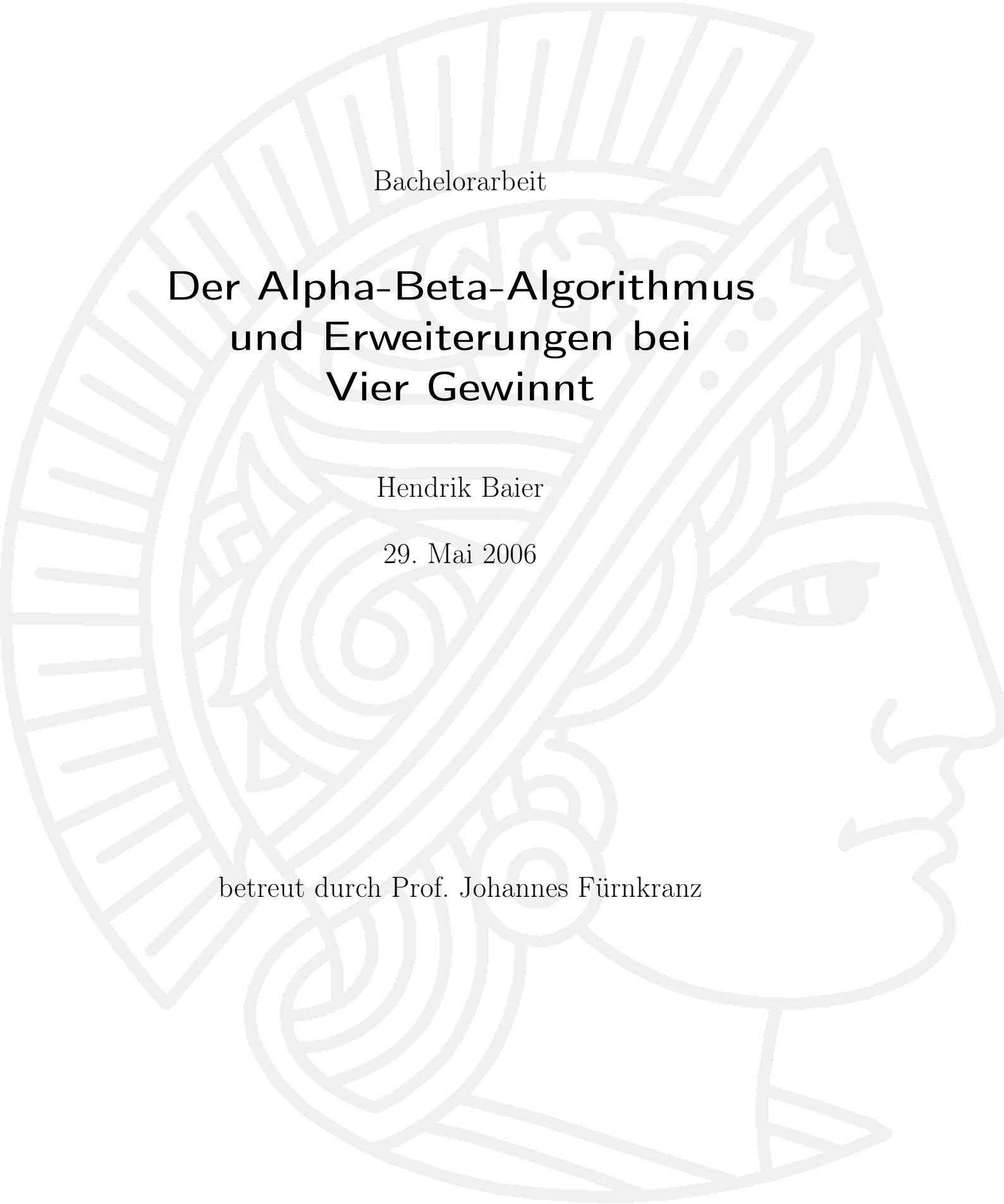


Bachelorarbeit

Der Alpha-Beta-Algorithmus und Erweiterungen bei Vier Gewinnt

Hendrik Baier

29. Mai 2006

betreut durch Prof. Johannes Fürnkranz

Hiermit versichere ich, daß ich die Arbeit selbständig verfaßt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.
Offenbach, den 18. April 2006

Verbesserte und überarbeitete Version:
Offenbach, den 29. Mai 2006

Vier Gewinnt ist gelöst: Bei perfektem Spiel beider Spieler gewinnt Weiß. James D. Allen gab dieses Ergebnis am 1. Oktober 1988 in einem Posting in rec.games.programmer bekannt, Victor Allis seine unabhängig gefundene Lösung nur 15 Tage später (siehe [1],[3]). Dennoch ist es ein für menschliche Spieler nach wie vor spannendes und taktisch überraschend herausforderndes Spiel; und darüber hinaus bietet es sich geradezu an, um Algorithmen, die im wesentlichen in der Schachprogrammierung entwickelt wurden, in relativ klarer Form zu implementieren, abzuwandeln und zu erweitern. Es weist einerseits vergleichsweise einfache Regeln und wenig algorithmisch schwierig umzusetzende Feinheiten auf; andererseits ist es nicht trivial lösbar und sein Spielbaum tief und verzweigt genug, um aus dem Alpha-Beta-Algorithmus und seinen Erweiterungen signifikante Spielstärke zu ziehen. Dies ist das Ziel des Programms MILTON und der folgenden Arbeit.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einführung

> Das Spiel ist das einzige, was Männer wirklich ernst nehmen. Deshalb sind Spielregeln älter als alle Gesetze der Welt.
> *Peter Bamm*

1.1 Vier Gewinnt

Bei Vier Gewinnt (im Englischen meist „Connect Four“[1], auch „Plot Four“ oder „The Captain´s Mistress“[2]) handelt es sich in den Begriffen der mathematischen Spieltheorie, ebenso wie bei Schach, Dame, Go usw., um ein Zwei-Spieler-Nullsummenspiel mit perfekter Information; und zwar in Form eines Baumspiels. Dies bedeutet zunächst, daß zwei Spieler im Verlauf des Spiels abwechselnd ziehen (den beginnenden Spieler nennen wir im Folgenden *Weiß*, den nachziehenden *Schwarz*) – hierbei ist es im Gegensatz zu manch anderem Spiel, wie z.B. Go, nicht erlaubt, zu passen, es kann also ein Zugzwang-Phänomen auftreten, bei dem ein Spieler seine Situation durch jeden ihm möglichen Zug verschlechtern muß. Weiterhin bedeutet der Begriff, daß keine kooperativen Züge existieren, die beide Spieler gleichzeitig stärken, sondern jeder Zug den einen Spieler genauso stark fördert, wie er den anderen behindert – die *Summe* der Nutzeneffekte beider Spieler also immer gleich Null ist. Schließlich sagt er aus, daß der Zustand des Spiels jederzeit für beide Spieler *vollständig* einsehbar ist, im Gegensatz etwa zu den meisten Kartenspielen.

1.2 Regeln

Vier Gewinnt wird auf einem senkrecht aufgestellten Brett mit sieben Spalten und sechs Reihen gespielt. Jeder Spieler besitzt 21 Scheiben seiner Farbe. Die Spieler ziehen, indem sie eine solche Scheibe in eine nicht vollständig angefüllte Spalte ihrer Wahl einwerfen, so daß jene das niedrigste unbesetzte Feld in der Spalte einnimmt. Ein Spieler gewinnt, indem er vier seiner eigenen Scheiben in einer *ununterbrochenen* Reihe – waagrecht, senkrecht oder diagonal – plaziert, was das Spiel beendet (Abb. 1.2); das Spiel endet un-

[1] Unter diesem Namen veröffentlichte Milton Bradley, nach dem das hier betrachtete Programm benannt wurde, das Spiel 1974 und sorgte für seine weite Verbreitung.

[2] Der Legende nach war Captain James Cook auf seinen berühmten Reisen auf der H.M.S. Endeavour und der H.M.S. Resolution abends so oft unauffindbar, daß die Crew witzelte, er müsse eine Geliebte in der Kabine verbergen. Als die Crew herausfand, daß er sich die Zeit mit einigen Partien Vier Gewinnt gegen den Naturforscher Sir Joseph Banks und den Botaniker Daniel Solander vertrieb, bekam das Spiel den Namen „The Captain´s Mistress“.

entschieden, wenn das Brett vollständig gefüllt wird, ohne daß ein Spieler zuvor gewinnen konnte.

1.3 Notation und Terminologie

In Übereinstimmung mit [2] verwende ich für die 42 möglichen Züge eine schachähnliche Notation, wobei die Spalten mit Buchstaben von a bis g und die Reihen mit Zahlen von 1 bis 6 gekennzeichnet werden, entsprechend Abbildung 1.1. Weiterhin verwende ich die Begriffe *weiße Drohung* für ein Feld, das bei der Belegung durch Spieler Weiß dessen Gewinn bedeuten würde (Abb. 1.3), sowie *akute weiße Drohung* für eine Drohung, die Weiß im unmittelbar nächsten Zug füllen könnte (Abb. 1.4); für Schwarz analog. Eine *gerade Drohung* schließlich ist eine Drohung in einer geradzahligen Reihe (2, 4, 6), eine *ungerade Drohung* liegt entsprechend in einer ungeradzahligen (1, 3, 5).

Es ist zu beachten, daß die Züge eines Spiels gemeinhin wie folgt notiert werden: 1. ⟨weiß⟩ ⟨schwarz⟩, 2. ⟨weiß⟩ ⟨schwarz⟩, 3. ⟨weiß⟩ ⟨schwarz⟩ und so weiter. Daraus folgt, daß Weiß und Schwarz jeweils strenggenommen nur *Halbzüge* ausführen. Um Verwechslungen mit der umgangssprachlichen Bezeichnung „Zug“ für die Aktion eines einzelnen Spielers zu vermeiden, hat sich in der Informatik das englische Wort *ply* (statt *half move*) eingebürgert; ich benutze im Folgenden *Zug*, wenn ich einen einzelnen Spieler betrachte, und *Halbzug*, wenn ich den Bezug zum Spielprozeß betonen möchte.

Schlußendlich ist in diesem Dokument unter *Eröffnung* diejenige Phase des Spiels zu verstehen, die aus den ersten 8 Halbzügen besteht und in welcher dem Computer vollkommenes spieltheoretisches Wissen mit Datenbankhilfe zur Verfügung steht, unter *Endspiel* jene, in der dem Computer perfektes Spiel möglich ist, indem er das Spiel bis zum Ende durchrechnet, und unter *Mittelspiel* die dazwischenliegende Phase, in der die verwendeten Algorithmen ihre volle Stärke ausspielen.

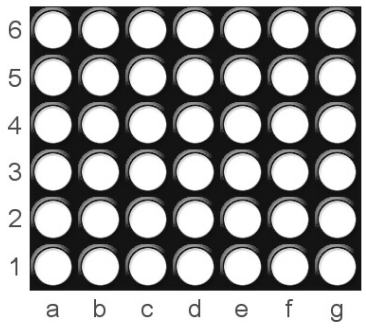


Abbildung 1.1: Spielbrett

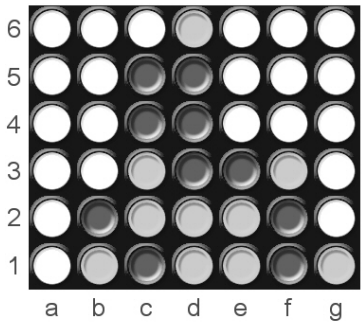


Abbildung 1.2: Gewinn für Schwarz: c5, d4, e3, f2

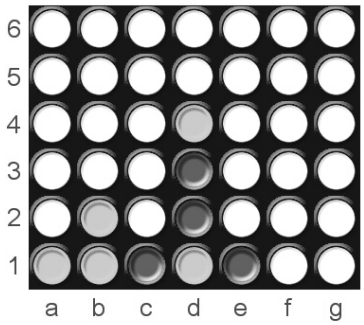


Abbildung 1.3: weiße Drohung: c3

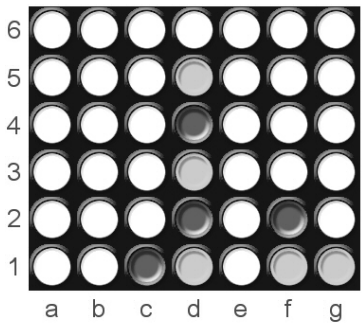


Abbildung 1.4: akute weiße Drohung: e1

2 Grundlegende Suchtechniken

Allem Anfang wohnt ein Zauber inne.
Hermann Hesse

2.1 Minimax und Negamax

Rational agierende Spieler verhalten sich bei Baumspielen auf die folgende Weise: Sie denken Halbzug für Halbzug, abwechselnd einen eigenen und einen des Gegners, voraus, indem sie die jeweils legalen Züge im Geiste der Reihe nach nachvollziehen und den erreichbaren Positionen Werte zuweisen (Gewinn für mich – ausgeglichen/unentschieden – Gewinn für den Gegner). Dabei beachten sie, daß sie selbst in jedem Zug ihren Vorteil zu *maximieren* versuchen, während der Gegner diesen *minimieren* will – nämlich, indem er für sich nach demselben Grundsatz handelt.

Der Name *Baumspiel* rührt daher, daß man sich diesen Prozeß als eine Suche in einem sogenannten *Spielbaum* vorstellen kann: In diesem bildet die aktuell auf dem Brett befindliche Spielstellung den Wurzelknoten; Positionen, die von dort aus mit einem Zug erreichbar sind, dessen Nachfolgerknoten; diese haben wiederum jeweils eine Reihe von ihnen ausgehender Äste bzw. Kanten, für jeden legalen Zug eine; und so weiter, bis hin zu den Blättern des Baumes – Endpositionen, die spieltheoretisch entscheidbar sind als Gewinn für Weiß, Gewinn für Schwarz oder unentschieden (Abb. 2.1).

Die ursprüngliche, unbeschränkte Form des *Minimax-Algorithmus* stellt eine Tiefensuche dar, die diesen Baum vollständig traversiert (Pseudocode in Listing 2.1). Die Funktion `Evaluate()` gibt hier den Wert einer Endposition aus der Sicht von Weiß zurück – einen sehr hohen Wert für einen weißen Sieg, einen sehr niedrigen Wert für einen schwarzen Sieg, Null für unentschieden. `Max()` beschreibt das Verhalten von Weiß – er versucht diese Werte zu maximieren – und `Min()` das von Schwarz, der das Gegenteil versucht. Befindet man sich also in einer Stellung, in der Weiß am Zug ist, ruft man `Max()` auf; alle legalen Züge für Weiß werden generiert und der Reihe nach „ausprobiert“, indem in jeder daraus unmittelbar resultierenden Stellung, in der Schwarz an der Reihe ist, `Min()` aufgerufen wird, und von dort aus nach Durchführung jedes Zuges jeweils wieder `Max()`; so bewegen sich die beiden wechselseitig rekursiven Funktionen nach unten im Spielbaum, bis sie zu den Blättern kommen, deren Werte mit `Evaluate()` bestimmt und nach oben gereicht werden. Jede Instanz der Funktion `Min()` wählt den *geringsten* der Werte aus, die von ihren rekursiven Aufrufen zurückkommen und somit von ihrer Position aus erreichbar sind, und jede der Funktion `Max()` den *größten*, bis schließlich der Wert der Wurzelposition aus der Sicht von Weiß zurückgegeben und derjenige Zug, der zu diesem Wert führt, als der beste für Weiß ausgewählt wird (Abb. 2.2).

Aus der Nullsummenbedingung, also der Tatsache, daß eine Nutzenmaximierung aus

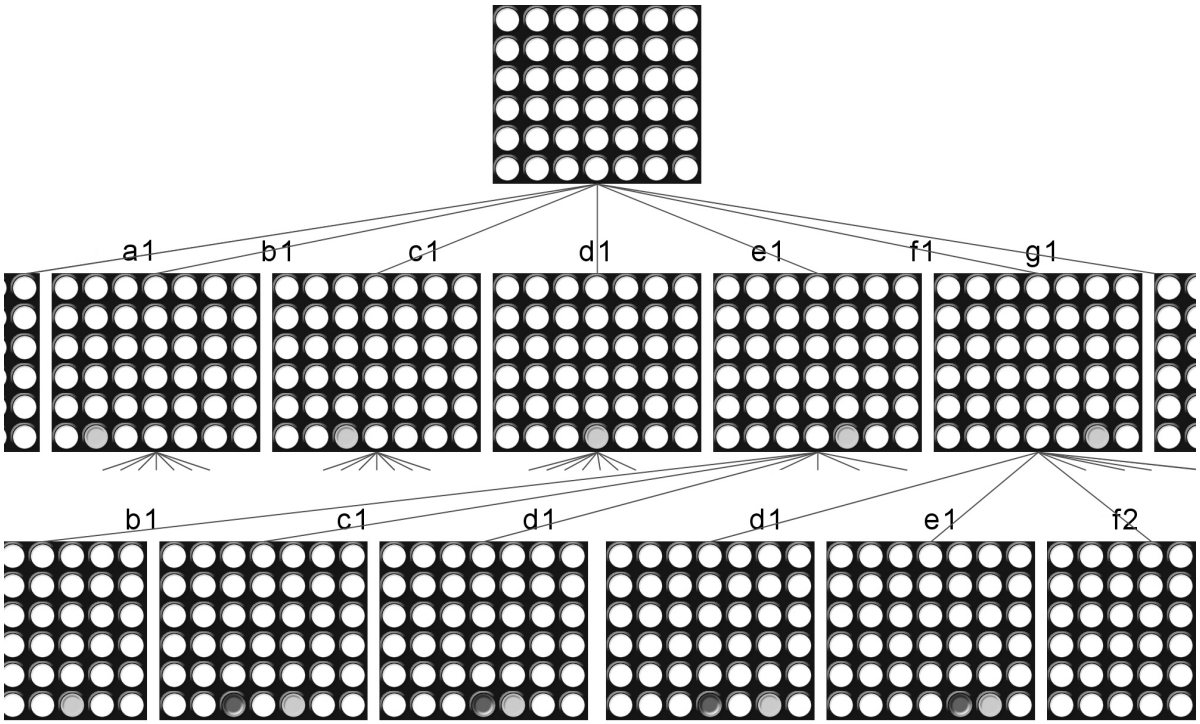


Abbildung 2.1: Ausschnitt aus den ersten zwei Ebenen des Spielbaums

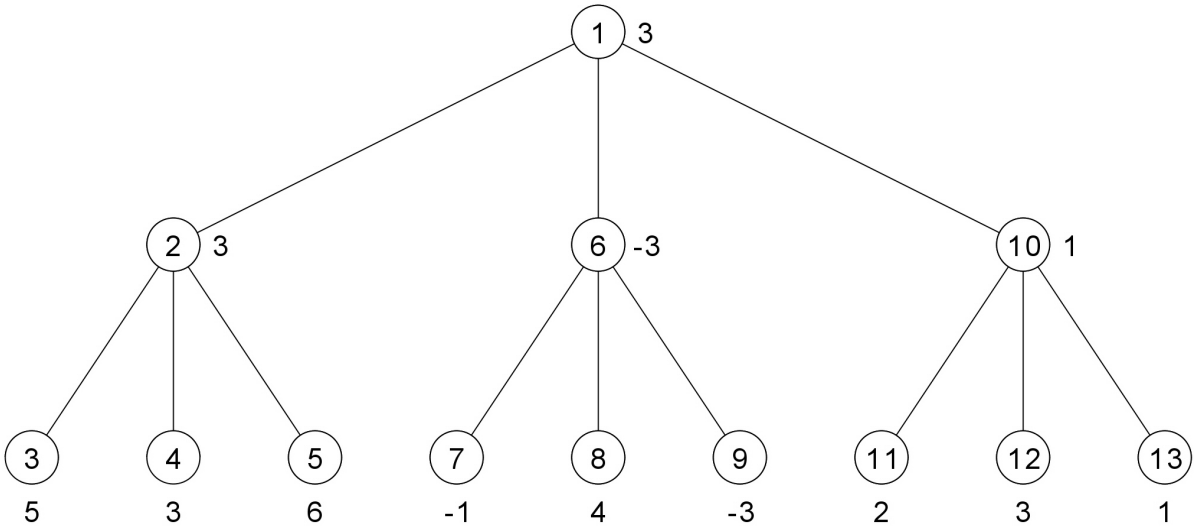


Abbildung 2.2: Minimax-Tiefensuche am Beispiel eines abstrakten Spielbaums. Neben den Knoten stehen ihre Werte aus der Sicht des maximierenden Spielers; er ist in der Wurzelposition am Zug. Die Zahlen in den Knoten bezeichnen die Reihenfolge, in der sie besucht werden.

```
int Max(Position p) {

     int best = -INFINITY;
     if(endPosition(p)) return evaluate(p);
     Move[] moves = generateMoves(p);
     for(int i=0; i<moves.length; i++) {
          makeMove(p, moves[i]);
          int value = Min(p);
          undoMove(p, moves[i]);
          if(value > best) best = value;
     }
     return best;

}

int Min(Position p) {

     int best = +INFINITY;
     if(endPosition(p)) return evaluate(p);
     Move[] moves = generateMoves(p);
     for(int i=0; i<moves.length; i++) {
          makeMove(p, moves[i]);
          int value = Min(p);
          undoMove(p, moves[i]);
          if(value < best) best = value;
     }
     return best;

}
```

Listing 2.1: Minimax unbeschränkt

Sicht von Weiß für Schwarz eine Minimierung seines Nutzens ist und umgekehrt, läßt sich eine weit elegantere Formulierung des Algorithmus ableiten, indem man die Endpositionen nicht stets aus der Sicht von Weiß, sondern aus der Sicht des *am Zug befindlichen Spielers* bewertet. Die sich ergebende *Negamax*-Version des Minimax-Algorithmus (Pseudocode in Listing 2.2) versucht, den Nutzen des gerade am Zug befindlichen Spielers zu maximieren; wenn sie einen Wert an die aufrufende Funktion zurückgibt, wird dieser *negiert*, da er einen Halbzug weiter oben im Suchbaum aus der Sicht des jeweils anderen Spielers betrachtet wird – eine sehr gute Stellungsbewertung wird für diesen zu einer sehr schlechten und vice versa, woraufhin er wiederum, ob Schwarz oder Weiß, eine Maximierung vornimmt. Negamax traversiert den Baum in exakt derselben Reihenfolge wie Minimax und gibt den gleichen Wert zurück, doch er beschreibt das Verhalten beider Spieler anders als Minimax auf dieselbe Weise, was intuitiv einleuchtet; er ist zudem übersichtlicher und leichter wartbar.

Im Hinblick auf die Anwendbarkeit des Algorithmus in der Praxis muß allerdings eine wichtige Einschränkung gemacht werden: Spielbäume sind üblicherweise viel zu groß, als daß man sie tatsächlich „bis zum Boden“, d.h. bis zu spieltheoretisch entschiedenen Positionen, durchsuchen könnte (der vollständige Schachbaum, aus der Startposition des Spiels heraus aufgebaut, enthält zum Beispiel weit über 10^{100} Knoten). Es ist aber dennoch möglich, das Minimax-Prinzip für ein solches Spiel zu verwenden – indem man, genau wie ein menschlicher Spieler, *nur einige Züge* vorausdenkt. Man begnügt sich also damit, den Baum nur bis zu einer vorgegebenen Tiefe aufzubauen, und modifiziert

```
int Negamax(Position p) {

     int best = -INFINITY;
     if(endPosition(p)) return evaluate(p);
     Move[] moves = generateMoves(p);
     for(int i=0; i<moves.length; i++) {
          makeMove(p, moves[i]);
          int value = -Negamax(p);
          undoMove(p, moves[i]);
          if(value > best) best = value;
     }
     return best;

}
```

Listing 2.2: Negamax unbeschränkt

```
int Negamax(Position p, int depth) {

     int best = -INFINITY;
     if(endPosition(p) || depth<=0) return evaluate(p);
     Move[] moves = generateMoves(p);
     for(int i=0; i<moves.length; i++) {
          makeMove(p, moves[i]);
          int value = -Negamax(p, depth-1);
          undoMove(p, moves[i]);
          if(value > best) best = value;
     }
     return best;

}
```

Listing 2.3: Negamax beschränkt

die statische Bewertungsfunktion dahingehend, daß sie mittels Heuristiken *geschätzte* Werte für Stellungen zurückgeben kann, die sich mitten im Spielverlauf und nicht an dessen Ende befinden – nicht mehr nur eines von drei möglichen Ergebnissen, sondern sinnvollerweise ein ganzes Spektrum von Werten: je wahrscheinlicher ein Gewinn für den am Zug befindlichen Spieler, desto höher, je wahrscheinlicher ein Verlust, desto niedriger. Mit dem jetzt benötigten zusätzlichen Parameter `depth` ergibt sich der Pseudocode von Listing 2.3, der bereits den Kern eines funktionsfähigen Programms darstellt – evaluiert wird hier nicht mehr erst an den Blättern des Baumes, sondern an den Knoten, an denen die verbleibende Suchtiefe auf Null gefallen ist, den Knoten des sogenannten *Horizonts*. Man beachte, daß dieses Grundgerüst vom tatsächlich gespielten Spiel völlig abstrahiert; ob hiermit Schach, Dame oder Vier Gewinnt gespielt wird, bleibt in der Implementierung von Zugerzeugung, Zugdurchführung und Positionsevaluation verborgen.

2.2 Alpha-Beta

Wir betrachten die in Abb. 2.2 visualisierte Suche. Minimax/Negamax durchsucht den gesamten Baum, um den (idealisierenderweise die Korrektheit von `Evaluate()` vorausgesetzt) beweisbar besten Pfad auszuwählen, der vom am Zug befindlichen Spieler er-

zwungen werden kann. Die Anzahl der Knoten, die hierbei betrachtet werden muß, mag noch nicht überwältigend erscheinen; der *Verzweigungsfaktor* des Spielbaums, d.h. die Anzahl der von jedem Nicht-Blatt ausgehenden Kanten, beträgt bei Vier Gewinnt im Durchschnitt jedoch knapp 5, im Schach sogar circa 35. Solche *buschigen* Bäume führen bei Suchtiefen, die für eine vernünftige Spielstärke erforderlich sind, rasch zu einer Explosion des Suchraums.

Glücklicherweise ist es jedoch möglich, den *effektiven* Verzweigungsfaktor, d.h. die Anzahl der tatsächlich betrachteten Kanten pro innerem Knoten, und damit die Anzahl der insgesamt betrachteten Knoten drastisch zu reduzieren, *ohne* dabei die Qualität der Suche zu schmälern, also ohne auf das immer noch beweisbar richtige Ergebnis zu verzichten. Die Methode dazu basiert auf der Beobachtung, daß viele der von Minimax/-Negamax betrachteten Stellungen gar nichts zum Ergebnis (der Wahl des besten Zweiges an der Wurzel) beitragen.

Hierzu ein Beispiel: Man stelle sich ein Spiel vor, bei dem es zwei Spieler namens A und B sowie einige Taschen gibt, die B gehören und mit Gegenständen von unterschiedlichem Wert gefüllt sind. A darf eine Tasche auswählen – dies ist sein „Zug“ – und B muß ihm daraufhin aus der ausgewählten Tasche einen Gegenstand von Bs Wahl aushändigen – dies ist sein „Zug“. Natürlich versucht B, seinen Verlust zu minimieren, er wird A also aus der ausgewählten Tasche den jeweils wertlosesten Gegenstand aussuchen. As Plan ist es also, nicht etwa die Tasche mit dem wertvollsten Gegenstand auszuwählen - diesen wird er ohnehin nicht bekommen – sondern die Tasche, deren wertlosester Gegenstand immer noch mehr Wert besitzt, als die jeweils wertlosesten der anderen Taschen. A übernimmt bei seinen Überlegungen abwechselnd also seine eigene, maximierende, und Bs minimierende Strategie. Wie verfährt A aber genau, wenn er die Taschen durchsucht?

Nach dem Minimax-Algorithmus durchsuchte er alle Taschen vollständig und träfe daraufhin seine Wahl – sicherlich die korrekte, aber erst nach langer Zeit. Also geht A anders vor: Er durchsucht die erste Tasche vollständig. Nehmen wir an, er findet darin eine teure Armbanduhr, einen 20-Euro-Schein und den Schlüssel für einen Neuwagen. Er weiß: Würde er diese Tasche wählen, bekäme er von B mit Sicherheit den 20-Euro-Schein. Als nächstes schaut er in die zweite Tasche und sieht darin einen Apfel. Dieser Apfel ist wertloser als der 20-Euro-Schein, den A aus der ersten Tasche bekäme – also wird er die zweite Tasche nicht wählen, *unabhängig* vom Wert der weiteren Gegenstände darin. Wie wertvoll oder wertlos diese sind, spielt keine Rolle, denn A kennt bereits eine untere Schranke für seinen erzielbaren Nutzen – 20 Euro – und eine obere Schranke für den Nutzen aus Tasche zwei – den Wert eines Apfels; B wird ihm ja ohnehin nichts wertvolleres aus dieser Tasche auswählen, falls es so etwas überhaupt gibt. A hat sich also ein wenig Arbeit gespart und kann bei Tasche 3 mit der Suche fortfahren.

Der *Alpha-Beta-Algorithmus* (Pseudocode in Listing 2.4) funktioniert rekursiv exakt auf diese Art und Weise. Hierfür verwendet er zwei weitere Parameter, die während der Suche verwendet und manipuliert werden: `alpha` repräsentiert den höchsten bisher bekannten Gewinn, den der am Zug befindliche Spieler auf irgendeine Art und Weise erzwingen kann – alles, was kleiner oder gleich `alpha` ist, ist nutzlos, da man durch den bisherigen Suchverlauf bereits eine Strategie kennt, die *aus Sicht des aktuellen Spielers* `alpha` erzielt. `alpha` stellt also eine untere Schranke dar. `beta` hingegen repräsentiert

```
int AlphaBeta(Position p, int depth, int alpha, int beta) {

     if(endPosition(p) || depth<=0) return evaluate(p);
     Move[] moves = generateMoves(p);
     sort(moves, p);
     for(int i=0; i<moves.length; i++) {
          makeMove(p, moves[i]);
          int value = -AlphaBeta(p, depth-1, -beta, -alpha);
          undoMove(p, moves[i]);
          if(value >= beta) return beta;
          if(value > alpha) alpha = value;
     }
     return alpha;

}
```

Listing 2.4: Alpha-Beta-Suche

den höchsten bisher bekannten Verlust, den der *Gegner* des aktuellen Spielers akzeptieren muß – es ist also bereits eine Strategie für den Gegner bekannt, ein Ergebnis zu erzielen, das *aus seiner Sicht* nicht schlechter als `beta` ist. `beta` stellt also für den aktuellen Spieler eine obere Schranke dar. Wenn die Suche einen Pfad betrachtet, der für den ziehenden Spieler einen Wert von `beta` oder mehr zurückgibt, ist dieser Pfad zu gut – der betreffende Spieler wird keine Chance zur Verwirklichung einer solchen Strategie bekommen. Im obigen Beispiel: In dem Moment, in dem A den Zug zu überdenken beginnt, Tasche zwei auszuwählen, beträgt sein `alpha`-Wert +20 Euro, da er durch eine bereits bekannte Strategie einen Gewinn von 20 Euro erzwingen kann (Tasche eins wählen). Sein `beta`-Wert beträgt theoretisch noch unendlich – was der für B größte Verlust ist, den er ertragen muß, ist noch unbekannt (in den weiteren Taschen können beliebig wertvolle Gegenstände sein). In dem Moment, in dem aus Sicht von B die Möglichkeit betrachtet wird, A den Apfel zu geben, liegt Bs `beta`-Wert bei −20 Euro, da er bereits weiß, daß A keinen kleineren Gewinn als 20 Euro wird akzeptieren müssen (was aus Sicht von B ein Verlust von 20 Euro ist). Sein `alpha`-Wert liegt aber noch bei minus unendlich – er kann sich, bevor die übrigen Taschen nicht untersucht wurden, nicht sicher sein, wie klein er seinen Verlust halten kann, d.h. was sein höchster erreichbarer Nutzen ist. Da nun aber der Verlust des Apfels für B einen höheren Nutzen darstellt als der Verlust der 20 Euro, bedeutet dies eine Überschreitung seines `beta`-Werts und somit die Gewißheit, daß man diese Möglichkeit nicht weiter in Erwägung ziehen muß – A wird sie zu verhindern wissen.

Das Verhalten der Suche in jedem Knoten hängt also von den Ergebnissen ab, welche die durchsuchten Züge zurückgeben. Gibt ein Zug einen Wert kleiner oder gleich `alpha` zurück, vergißt man ihn, da man schon eine Strategie kennt, die zu einer Position mit dem Wert `alpha` führt (*fail-low*). Gibt ein Zug einen Wert größer oder gleich `beta` zurück, vergißt man den gesamten aktuellen Knoten[1] – der Gegner wird eine Position nicht zulassen, in der ein so guter Wert erreicht werden kann (*fail-high*). Diese Situation bezeichnet man als beta-Cutoff, und aus ihr resultiert die Überlegenheit des Alpha-Beta-

[1] Man beachte, daß hiermit auch der gesamte *Teilbaum* unter dem aktuellen Knoten abgeschnitten wird (*Pruning*).

Algorithmus gegenüber Minimax. Bekommt man von einem Zug jedoch einen Wert, der zwischen `alpha` und `beta` liegt, so hat man einen neuen lokal besten Zug und merkt ihn sich, indem man die Obergrenze dessen, was man zu erzielen bereits sicher sein kann (also die Untergrenze dessen, was man schlußendlich akzeptieren wird), in `alpha` speichert und diese Grenze somit vor der Durchsuchung der weiteren Züge anhebt.

Wie ebenfalls aus dem Beispiel ersichtlich ist, tauschen `alpha` und `beta` beim rekursiven Aufruf der Funktion ihre Plätze und werden negiert: As `alpha` von 20 wird zu Bs `beta` von -20, As `beta` von unendlich wird zu Bs `alpha` von minus unendlich.

Im Gegensatz zum Minimax-Algorithmus, der in jedem Fall alle Knoten des Suchbaums durchwandert, ist die Performance des Alpha-Beta stark von der Reihenfolge abhängig, in der die Züge untersucht werden. Betrachtet er in jedem Knoten die möglichen Züge in aufsteigender Reihenfolge – zuerst den schlechtesten, zuletzt den besten – wird es nie einen beta-Cutoff geben, und der Algorithmus wird genauso ineffizient sein wie Minimax. Durchsucht er den besten Zug jedoch immer zuerst, entspricht der effektive Verzweigungsfaktor in etwa der Quadratwurzel aus dem tatsächlichen Verzweigungsfaktor, was in einer *massiven* Verbesserung der Suche resultiert (bis zu doppelter Suchtiefe bei gleicher Knotenzahl; für Details siehe [6]). Die Abbildungen 2.3 und 2.4 demonstrieren den Unterschied.

Da die Notwendigkeit der Suche aber darauf basiert, daß man den besten Zug in einer gegebenen Position nicht a priori weiß, wird das Thema Move Ordering, also Zugsortierung, noch gesondert behandelt werden.

2.3 Principal-Variation-Suche

Für das vom Aufruf `Alphabeta(p, alpha, beta, depth)` zurückgegebene Ergebnis gilt: `alpha` $\leq$ Ergebnis $\leq$ `beta`. Ist das Ergebnis echt größer `alpha` und echt kleiner `beta`, dann liegt ein exakter Wert vor; fällt das Ergebnis jedoch auf `alpha`, so handelt es sich dabei um eine obere Schranke (*fail-low* $\rightarrow$ tatsächlicher Wert $\leq$ `alpha`), und fällt es schließlich auf `beta`, so hat man eine untere Schranke für das gesuchte Resultat (*fail-high* $\rightarrow$ tatsächlicher Wert $\geq$ `beta`). Üblicherweise verlangt man für die Wurzelstellung einen exakten Wert und ruft daher `AlphaBeta(p, -infinity, +infinity, depth)` auf, um dem Resultat keine ungewissen Grenzen zu setzen.

Je näher `alpha` und `beta` jedoch beieinanderliegen, je enger das *Suchfenster* also gesetzt wird, desto höher steigt die Cutoff-Rate, d.h. desto schneller läuft die Suche tendenziell, während die Wahrscheinlichkeit, einen exakten Wert zurückzuerhalten, sinkt. Sucht man nun mit einem leeren Suchfenster, d.h. mit `beta` = `alpha` + 1, so erhält man gar keine exakten Resultate mehr, sondern testet lediglich auf die Überschreitung der Schranke `alpha`: Man erhält Resultat = `beta` $\rightarrow$ fail-high $\rightarrow$ tatsächlicher Wert $\geq$ `beta` = `alpha` + 1 $\rightarrow$ tatsächlicher Wert $>$ `alpha` oder aber Resultat = `alpha` $\rightarrow$ fail-low $\rightarrow$ tatsächlicher Wert $\leq$ `alpha`. Diese spezielle Konstellation von `alpha` und `beta` bezeichnet man als *Minimal Window* oder *Nullfenster*. Auf der Geschwindigkeit dieser Nullfenstersuchen basiert die folgende Verfeinerung des Alpha-Beta-Algorithmus.

Jeder Knoten in einer Alpha-Beta-Suche gehört zu einer von drei Typen (siehe [12]):

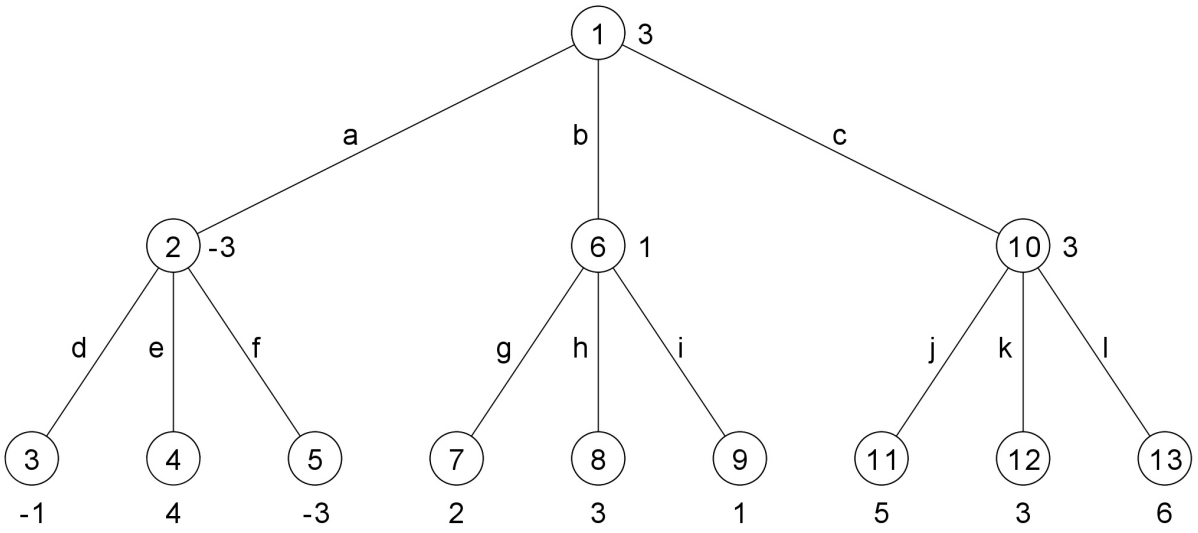


Abbildung 2.3: Alpha-Beta mit schlechter Zugsortierung. An keiner Stelle im Suchbaum erlauben die bisher bekannten Schranken, einen Teilbaum abzuschneiden.

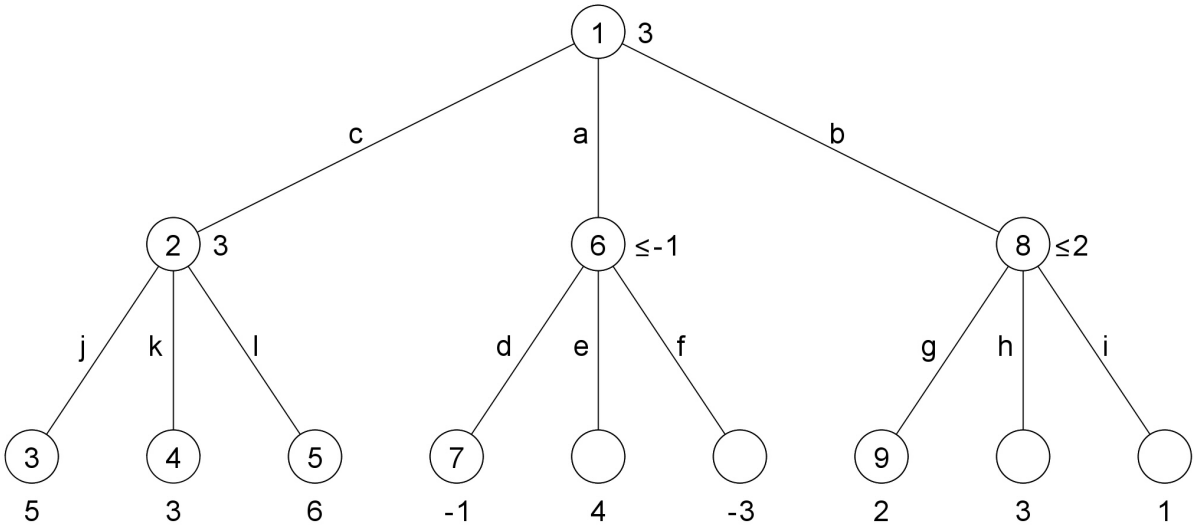


Abbildung 2.4: Alpha-Beta mit guter Zugsortierung. Nach der Evaluation des siebten Knotens ist sicher, daß der sechste Knoten für den maximierenden Spieler weniger wert ist als der zweite - nämlich höchstens -1. Man sagt auch: Zug d widerlegt Zug a. Also müssen die Züge e und f nicht mehr betrachtet werden. Ebenso widerlegt Zug g Zug b.

1. Alpha-Knoten. Jeder von hier aus mögliche Zug gibt einen Wert kleiner oder gleich `alpha` zurück, d.h. keiner der möglichen Züge ist gut genug, etwas am Suchergebnis zu ändern.

2. Beta-Knoten. Mindestens einer der Züge gibt einen Wert größer oder gleich `beta` zurück, was zum Cutoff führt.

3. Principal-Variation-(PV-)Knoten. Mindestens einer der Züge bekommt einen Wert größer `alpha`, keiner jedoch einen größer gleich `beta`. Dies bedeutet, daß der betreffende Knoten, soweit man den Suchbaum bisher betrachtet hat, zur *Principal Variation* (auch: *Principal Continuation*) gehört, also zur erwarteten Zugabfolge bei rationalem Spiel beider Gegner.

Geht man nun von einer guten Zugsortierung aus, so läßt sich schon nach Durchsuchen des ersten Zuges eine Vermutung darüber anstellen, welcher Knotentyp vorliegt. Verursacht der erste Zug einen fail-high, liegt ein Betaknoten vor und man ist fertig. Verursacht der erste Zug jedoch einen fail-low, so liegt *wahrscheinlich* ein Alphaknoten vor, aufgrund der Annahme, daß kein besserer Zug in der Zugliste nachfolgt. Ebenso handelt es sich *vermutlich* um einen PV-Knoten, wenn der erste Zug einen Wert zwischen `alpha` und `beta` zurückgibt.

Hieraus ergibt sich die Idee, alle weiteren Züge nach dem ersten gefundenen PV-Zug nicht mehr auf ihren tatsächlichen Wert hin zu untersuchen, sondern zunächst lediglich mit Hilfe einer schnellen Nullfenstersuche, die um den Wert des PV-Zuges zentriert ist, auf eine Überschreitung dieses Wertes zu testen. Ergibt jene Suche einen fail-low, ist der aktuelle Zug dem PV-Zug unterlegen – mit welchem Wert genau, ist irrelevant. Die schnelle Nullfenstersuche gibt einem genügend Information, diesen Zug zu ignorieren. Anderenfalls, also bei einem fail-high, hat man einen neuen besten Zug gefunden, kennt aber seinen tatsächlichen Wert noch nicht; man ist also gezwungen, den aktuellen Zug ein weiteres Mal mit einem „normalen“, durch `alpha` und `beta` beschränkten Suchfenster zu betrachten.

Dieses *Principal-Variation-Suche* (PVS) genannte Verfahren (Pseudocode in Listing 2.5) bedeutet natürlich nur dann im Durchschnitt einen Fortschritt gegenüber dem reinen Alpha-Beta-Algorithmus, wenn die Anzahl der nötigen Doppelsuchen, also die Anzahl der fail-highs aus den Nullfenstersuchen, konstant niedrig bleibt. Daher ist es in noch stärkerem Maße von einer sehr guten Zugsortierung in jedem einzelnen Knoten abhängig.

In MILTON wurde PVS - in einer Variante, die nicht schon ab dem zweiten, sondern erst ab dem vierten möglichen Zug eine Nullfenstersuche anwendet, was sich in der Praxis als überlegen erwiesen hat - als Option zusätzlich zum gewöhnlichen Alpha-Beta realisiert. Die Knotenersparnis der Nullfenstersuchen stellte sich als relativ gering heraus. Für die Ermittlung aller in dieser Arbeit enthaltenen statistischen Werte und bei der Durchführung vergleichender Tests wurde PVS nicht berücksichtigt.

```
int PVSearch(Position p, int depth, int alpha, int beta) {

     boolean foundPV = false;
     if(endPosition(p) || depth<=0) return evaluate(p);
     Move[] moves = generateMoves(p);
     sort(moves, p);
     for(int i=0; i<moves.length; i++) {
          makeMove(p, moves[i]);
          int value;
          if(foundPV) {
               value = -PVSearch(p, depth-1, -alpha-1, -alpha);
               if((value > alpha) && (value < beta))
                    value = -PVSearch(p, depth-1, -beta, -alpha);
          } else {
               value = -PVSearch(p, depth-1, -beta, -alpha);
          }
          undoMove(p, moves[i]);
          if(value >= beta) return beta;
          if(value > alpha) {
               alpha = value;
               foundPV = true;
          }
     }
     return alpha;

}
```

Listing 2.5: Principal-Variation-Suche

2.4 MILTONS Grundgerüst

Anhand des nachfolgenden Codes werde ich im folgenden Stück für Stück den tatsächlichen Kern von MILTONs Suchalgorithmus entwickeln. Hierzu noch einige Anmerkungen:

1. Verlorene Positionen werden mit $-1000 + \textit{Anzahl der Scheiben}$ bewertet, um ferne Verluste den nahen vorzuziehen.

2. Züge werden als einfache Integer kodiert; im Array `moves` ist der letzte Eintrag stets `moves[7]` und gibt die Anzahl der legalen Züge an.

3. Das Argument `move` wurde eingeführt, um den jeweiligen Halbzug, der von einem Suchknoten in Tiefe x zu einem Tochterknoten in Suchtiefe $x + 1$ führt, erst in diesem Tochterknoten tatsächlich durchführen zu müssen (im Gegensatz zu den vorhergehenden Beispielen in Pseudocode, die das Ziehen stets vor den rekursiven Aufruf des Tochterknotens setzten). Dies hat den Vorteil, daß die Methode `isWon()` jenen Halbzug als Argument erhalten und somit weit effizienter bestimmen kann, ob mit ihm ein Vierer geschlossen wurde, d.h. ob der Tochterknoten einer entschiedenen Stellung entspricht.[2]

4. Das Argument `pv` dient der Ermittlung der Principal Variation. In der bisher betrachteten Form gibt der Alpha-Beta-Algorithmus lediglich eine Bewertung der

[2] Ohne die Information über den letzten durchgeführten Halbzug muß man alle 69 möglichen Vierer überprüfen, mit ihr je nach Spielfeld nur zwei bis elf.

Wurzelposition zurück – von weit größerem Interesse ist jedoch der Zug, den der Algorithmus in dieser Stellung wählen würde, also der erste Zug in jener Reihe voraussichtlich optimaler Züge beider Seiten, aus der die zurückgegebene Bewertung resultiert.

Um also die PV zu ermitteln, kommt eine einfach verkettete Liste von Zügen zum Einsatz. Ruft man `AlphaBeta` mit einer Suchtiefe von Null auf, so wird die aktuelle Position einfach evaluiert und der entsprechende Wert zurückgegeben – da kein Zug ausgeführt wurde, hat die PV für diesen Knoten die Länge Null. Erfolgt der Aufruf jedoch mit einer höheren Tiefe und ergibt sich für einen Zug ein Score zwischen `alpha` und `beta`, so wird eine PV erstellt, die aus diesem Zug und der rekursiv ebenso entstandenen PV desjenigen Tochterknotens besteht, welcher diesen Score zurückgab.[3]

[3] [12] merkt an, daß sich die Principal Variation nach durchgeführter Suche auch aus den Hash Tables rekonstruieren lassen müßte, da diese den besten gefundenen Zug jeder PV-Position enthalten. Hieraus könnte sich eine effizientere, aber (wie alles im Zusammenhang mit Hash Tables) schwieriger zu entwickelnde und zu wartende Lösung ergeben.

```
public int AlphaBeta(Position p, int depth, int alpha, int beta, MoveList pv, int move){

     p.makeMove(move);

     if(p.isWon(!p.toMove,move)) {
          pv.length = 0;
          return -1000+p.numberOfDiscs();
     }
     if(p.isDraw()) {
          pv.length = 0;
          return 0;
     }
     if(depth <= 0) {
          pv.length = 0;
          return evaluate(p);
     }

     MoveList mypv = new MoveList();

     int[] moves = generateMoves(p);
     sort(moves, p);

     for(int i=0;i<moves[7];i++) {
          int value = -AlphaBeta(p, depth-1, -beta, -alpha, mypv, moves[i]);
          p.undoMove(moves[i]);
          if(value>=beta) {
               return beta;
          }
          if(value>alpha) {
               alpha = value;
               pv.setFirst(new MoveListElement(moves[i]));
               if(mypv.length!=0) {
                    pv.setRest(mypv);
               }
               pv.length = mypv.length + 1;
          }
     }
     return alpha;
}
```

Listing 2.6: Die Grundform von Miltons Alpha-Beta

3 Fortgeschrittene Suchtechniken

I will go anywhere, provide it be forward.
David Livingstone

3.1 Quiescence Search

Bei der Betrachtung der Minimax- und Alpha-Beta-Algorithmen wurde bereits erwähnt, daß die Korrektheit des Suchergebnisses ganz wesentlich von der Korrektheit der Evaluationsfunktion abhängt, da ihre Resultate von den Horizontknoten bis zur Wurzel durchgereicht werden und den Suchprozeß steuern. Naturgemäß kann eine *statische* Evaluation einer Spielposition in den meisten Fällen nur einen heuristischen Wert, einen aus der Theorie des Spiels mehr oder weniger wohlbegründeten Näherungswert liefern, der von *statischen Eigenschaften* der Stellung abhängt – wie im Schach etwa der Materialbalance und verschiedenen positionellen Vor- und Nachteilen der Spieler. Daher erhält man mit einer solchen Suche, die zentrale dynamische Aspekte des Spiels vernachlässigt, nur unbefriedigende Resultate – es stellt sich der sogenannte *Horizonteffekt* ein; im Schach beispielsweise, wenn in der betrachteten Horizontposition gerade ein Materialabtausch im Gange ist. Wenn Spieler A im letzten Halbzug einen Springer von Spieler B geschlagen hat, wird die Position von `evaluate()` als vielversprechend für A eingestuft, der im Materialwert vorne liegt. Dabei wird jedoch übersehen, daß B im unmittelbar nächsten Halbzug einen Turm gewinnt und dieser Tausch für A eher unvorteilhaft sein dürfte.

Eine Lösung für dieses Problem ist es, die statische Bewertungsfunktion möglichst nur auf Spielstellungen anzuwenden, für die sie geeignet ist: die sogenannten *quiescent nodes*, ruhigen Knoten. Hierfür verwendet man eine *Quiescence Search* oder Ruhesuche, die an die Knoten des Horizontes, wenn nötig, noch eine weitere Suchphase anhängt – nun aber nicht mehr in voller Breite, d.h. alle möglichen Züge betrachtend (was einfach auf einen weiteren Halbzug Suchtiefe hinausliefe), sondern nur noch auf diejenigen wenigen Züge konzentriert, welche die angesprochenen dynamischen Stellungsmerkmale auflösen. Erst an den Blättern dieser Ruhesuche wird statisch evaluiert.

Es wird hier also eine Form von *selektiver* Suche eingeführt: Die Selektivität in der Ruhesuche muß dabei so gewählt werden, daß sie schnelle Terminierung und zuverlässige Behandlung der wichtigsten dynamischen Bewertungsaspekte gleichzeitig gewährleistet. Im Schachspiel bedeutet dies gewöhnlich die Betrachtung von schlagenden, Schach gebenden oder verhindernden Zügen und Bauernumwandlungen.

MILTON verfügt über eine einfache, aber effektive Quiescence Search (Listing 3.1). Sie dient insbesondere als ein Schutz gegen taktische Drohungsketten, d.h. mehrere aufeinanderfolgende akute Drohungen, mit deren Hilfe der Gegner letztlich in eine ausweglose Situation (z.B. eine Doppeldrohung) gezwungen werden kann, wenn ihre Anzahl seinen

```
public int AlphaBeta(Position p, int depth, int alpha, int beta,
                     double extension, MoveList pv, int move) {

     p.makeMove(move);

     if(p.isWon(!p.toMove,move)) {
          pv.length = 0;
          return -1000+p.numberOfDiscs();
     }
     if(p.isDraw()) {
          pv.length = 0;
          return 0;
     }

     if(depth <= (0-extension)) {
          int acute2 = acuteThreat(p, p.toMove);
          if(acute2!=-1) {
               pv.length = 0;
               return 1000-p.numberOfDiscs()-1;
          }
          int acute = acuteThreat(p, !p.toMove);
          if(acute==-1) {
               pv.length = 0;
               return evaluate(p, false);
          }
          extension += 2;
          int value = -AlphaBeta(p, depth-1, -beta, -alpha, extension, mypv, acute);
          p.undoMove(acute);
          if(value>=beta) {
               return beta;
          }
          if(value>alpha) {
               alpha = value;
               pv.setFirst(new MoveListElement(akut));
               if(mypv.length!=0) {
                    pv.setRest(mypv);
               }
               pv.length = mypv.length + 1;
          }
          return alpha;
     }

     MoveList mypv = new MoveList();

     int[] moves = generateMoves(p);
     sort(moves, p);

     for(int i=0;i<moves[7];i++) {
          int value = -AlphaBeta(p, depth-1, -beta, -alpha, extension, mypv, moves[i]);
          p.undoMove(moves[i]);
          if(value>=beta) {
               return beta;
          }
          if(value>alpha) {
               alpha = value;
               pv.setFirst(new MoveListElement(moves[i]));
               if(mypv.length!=0) {
                    pv.setRest(mypv);
               }
               pv.length = mypv.length + 1;
          }
     }
     return alpha;
}
```

Listing 3.1: MILTONs Alpha-Beta mit Quiescence Search

Suchhorizont überschreitet. Gelangt die Suche am Horizont an, also an einer Position mit einer verbleibenden Suchtiefe von kleiner oder gleich Null, so wird zunächst auf akute Drohungen des aktuellen Spielers getestet. Existiert eine solche, würde er im nächsten Halbzug gewinnen, und die Stellung erhält einen entsprechenden Rückgabewert (≈11% der Fälle). Existiert keine, werden akute Drohungen des anderen Spielers gesucht. Nun ergeben sich zwei Möglichkeiten: Entweder auch dieser besitzt keine – dann gilt die Position als ruhig und wird regulär evaluiert (≈67%). Oder aber der Gegner hat eine akute Drohung – in diesem Fall wird die verbleibende Suchtiefe um zwei Halbzüge erhöht und somit die Quiescence Search gestartet bzw. verlängert (≈22%).

Zwei Halbzüge erlauben es dem aktuellen Spieler, die akute Drohung zu verhindern, und seinem Gegner, dessen Drohungskette nach Möglichkeit fortzusetzen; am somit wiederum erreichten neuen Suchhorizont wird der beschriebene Test ein weiteres Mal durchgeführt usw. Theoretisch besteht also die Gefahr einer Suchexplosion - in der Praxis aber wird dieses Risiko durch die taktische Stärke der Ruhesuche mehr als ausgeglichen.

Um den praktischen Nutzen der Ruhesuche wie auch den anderer Features unter realistischen Spielbedingungen evaluieren zu können, setzte ich einen *self-play*-Test ein, der Spiele aus 49 verschiedenen Startpositionen umfaßte (erste Züge 1. a1 a2, 1. a1 b1, 1. a1 c1, ... , 1. b1 a1, ... , 1. g1 g2 bereits auf dem Brett). Jedes dieser Spiele spielte MILTON *mit* dem jeweiligen Feature zweimal gegen MILTON *ohne* das jeweilige Feature, wobei die Rollen des weißen und schwarzen Spielers zwischen beiden Durchläufen getauscht wurden. Die Zeitbeschränkungen wurden hierfür vergleichbar gestaltet, um einen erhöhten Zeitaufwand als Ursache erhöhter Spielstärke auszuschließen.

Im Falle der Quiescence Search ergab der self-play-Test ein deutliches Gewinnverhältnis von 47 : 34 gegen den Algorithmus ohne ein entsprechendes Verfahren.

3.2 Search Extensions

Wie im Abschnitt über Minimax bereits ausgeführt, wird der Suchbaum aufgrund dessen exponentiellen Wachstums mit der Suchtiefe im allgemeinen nicht vollständig, sondern nur bis zu den Horizontknoten entwickelt. Ein Parameter, der für die verbleibende Tiefe steht, wird zu diesem Zweck von rekursivem Aufruf zu rekursivem Aufruf stetig um einen Halbzug verringert. Daher haben alle von einem Knoten mit Tiefenparameter T ausgehenden Pfade zum Horizont (oder dem Level, auf dem die Ruhesuche beginnt) die Länge T, soweit sie nicht zuvor enden oder abgeschnitten werden.

Suchverfahren, die solch uniforme Bäume traversieren, bieten sich zwar für mathematische Studien und Beweisführungen an, im tatsächlichen Spiel weisen sie aber große taktische Schwächen auf; wie ein menschlicher Spieler die interessanteren Spielzüge und Positionen tiefer durchdenkt als die weniger vielversprechenden, sollte auch ein guter Suchalgorithmus den Tiefenparameter *variabel* behandeln, indem er z.B. beim Durchsuchen einer taktisch schwierigeren Variation T unverringert weitergibt. Solche *Search Extensions* – Sucherweiterungen – führen zu völlig unvorhersehbaren Suchbäumen, da sie rekursiv mehrfach entlang eines einzigen Pfades auftreten können, und sind mit Vorsicht zu implementieren, um die Suche nicht unproportional aufzublähen – T stellt ja

jetzt nur noch eine untere Schranke für die Länge der Pfade bis zu den Horizontknoten dar. Erfahrungsgemäß erhöht sich jedoch die Spielstärke eines Programms, das beispielsweise anstelle eines weiteren Halbzugs uniformer Suchtiefe eine vergleichbare Suchzeit in Extensions investiert, dramatisch.

MILTONs Sucherweiterungen sind ein gutes Beispiel für die starke wechselseitige Abhängigkeit aller Elemente eines spielfähigen Alpha-Beta-Algorithmus. Ich implementierte nach verschiedenen Experimenten zunächst das folgende, relativ unkomplizierte Verfahren: Die verbleibende Suchtiefe wurde immer dann, wenn sich die legalen Zugmöglichkeiten eines Spielers auf einen Zug reduzieren ließen (entweder, weil nur noch eine Spalte frei war, oder aber, weil einer der Spieler eine akute Drohung hatte – darüber mehr im Kapitel über Zuggenerierung), um 0,5 Halbzüge erhöht. Diese sog. *Fractional Search Extensions* akkumulierten sich entlang der Suchpfade in demselben Funktionsargument wie die Extensions, welche die Ruhesuche verursachte. Auf diese Weise schien die Weiterverfolgung vieler taktisch interessanter Linien gesichert, aber auch ein gewisser Schutz gegen ein unkontrollierbares Anwachsen der Knotenzahl; denn die Suche wurde nur in solchen Teilbäumen vertieft, deren Größe in einem oder mehreren Halbzügen nicht gewachsen war.

Im self-play-Test ergab sich jedoch ein kaum signifikantes Ergebnis von 38 : 37 für den Algorithmus mit Sucherweiterungen. Es ließ sich feststellen, daß der Erfolg der Search Extensions stark davon abhing, ob im jeweiligen Spiel tiefe taktische Kombinationen eine Rolle spielten oder nicht; und darüber hinaus, daß die im vorigen Abschnitt beschriebene Quiescence Search oftmals bereits die von den Search Extensions erhofften Resultate erzielte. Durch die im Vergleich zu Schach deutlich einfachere taktische Struktur des Spiels überlappten sich offensichtlich die Wirkungen beider Features, so daß sich der doppelt erhöhte Suchaufwand nicht mehr lohnte – was die Frage aufwarf, welches Verfahren das bessere Verhältnis von Spielstärke zu Knotenzahl aufweisen konnte.

Die zunächst widersprüchlich erscheinenden Ergebnisse führten dazu, daß ich in dieser Frage auf die Round-Robin-Methode zurückgreifen mußte, d.h. auf jeweils einen self-play-Test zwischen allen Paarungen der vier möglichen Programmvarianten: MILTON mit Erweiterungen und Ruhesuche; mit Erweiterungen, aber ohne Ruhesuche; ohne Erweiterungen, aber mit Ruhesuche und schließlich MILTON ohne beide Features. Abb. 3.1 zeigt die Ergebnisse, die zur Streichung der Search Extensions in der aktuellen Version von MILTON führten. Ich führe ihren Code in Anhang B auf.

3.3 Iterative Deepening

Es ist sehr schwer, vorherzusagen, wie lange eine Suche bis zu einer im Voraus festgelegten Tiefe dauern wird. In komplexen Mittelspielpositionen mit vielen taktischen Varianten wird die Anzahl der Knoten, die man beispielsweise für eine Suche der Tiefe 12 braucht, um ein Vielfaches höher sein als in einer Endspielsituation, die relativ wenige Zugmöglichkeiten offenläßt. Dies bedeutet auch, daß man kaum abschätzen kann, welche Suchtiefe in einer vorgegebenen Zeitdauer erreichbar ist. *Iterative Deepening* ist eine simple, aber erstaunlich effiziente Methode, innerhalb einer gegebenen Zeitspanne

	(1)	(2)	(3)	(4)	Summe der Punkte
(1)	–	38 : 37	33 : 42	44 : 33	115
(2)	–	–	38 : 40	47 : 34	122
(3)	–	–	–	27 : 41	109
(4)	–	–	–	–	108

Abbildung 3.1: Quiescence Search und Search Extensions: (1) bezeichnet den Algorithmus mit beiden Features, (2) den mit ausschließlich Quiescence Search, (3) den mit ausschließlich Extensions, (4) den ohne beide Features. Der Spieler der jeweiligen Zeile ist als der erste Spieler zu verstehen, der Spieler der Spalte als der zweite.

so tief wie möglich zu suchen (Pseudocode in Listing 3.2).

Man sucht zunächst lediglich bis zur Tiefe eins. Liegt man hiernach noch unter der veranschlagten Suchzeit, sucht man ein weiteres Mal von vorne – diesmal bis zur Suchtiefe zwei. Dann bis zur Tiefe drei, und so fort, bis von den länger und länger werdenden Suchen die gewünschte Zeit überschritten wird; an dieser Stelle kann man abbrechen und den aus der letzten *vollendeten* Iteration gespeicherten Zug zurückgeben.[1]

In der Tat bietet dieses Vorgehen noch einen weiteren Vorteil: In Iteration x kann man auf das Wissen aus Iteration $x - 1$ zurückgreifen, genauer gesagt auf diejenigen Züge, die sich bis zur Suchtiefe $x - 1$ als die besten erwiesen haben (siehe Abschnitt über Hash Tables). Dies ist ein guter Ausgangspunkt für die Zugsortierung in Iteration x und sorgt in einigen Fällen in der Tat dafür, daß die Anzahl der benötigten Knoten für alle Suchiterationen von Tiefe 1 bis Tiefe T geringer ausfällt als die Anzahl der Knoten einer direkten Suche bis zur Tiefe T – bei Mittelspielzügen für $T = 12$ z.B. durchaus um rund 20%.

MILTON verwendet für die iterierte Suche statt eines Zeitlimits eine Grenze von einer Million Knoten, nach der eine `TooManyNodesException` geworfen wird (ein schnellerer Abbruch erfolgt nur bei eindeutigen Gewinn- oder Verlustzügen aus dem Eröffnungsbuch oder aber in einer Suchtiefe, welche die restliche Anzahl freier Felder übersteigt). Auf diese Weise kann man Veränderungen am Alpha-Beta-Algorithmus auch unter schwankender Systemauslastung vergleichen. Zum Testen technischer Optimierungen wie etwa der Einführung der Bitboards, deren Effizienz sich nicht in Veränderungen der Knotenzahl ausdrücken läßt, muß man allerdings selbstverständlich auf Zeitmessung zurückgreifen.

3.4 Aspiration Windows

Aspiration Windows, frei mit Erwartungsfenster übersetzbar, stellen eine weitere Verbesserung des Iterative Deepening dar (Listing 3.3). Sie beruhen auf einem ähnlichen Prinzip wie die bereits beschriebene Principal-Variation-Suche: Auf einer Veränderung

[1] Derzeit noch unrealisiert ist die Idee, auch aus vorzeitig abgebrochenen Iterationen noch Informationen zu entnehmen, z.B. wenn der darin vorläufig beste Zug bereits ein besseres Evaluationsergebnis liefert als der beste Zug aus der tiefsten vollständig durchgeführten Suche.

```
public int returnMove(Position arg) {

     Position p = new Position(arg);

     int lastRoundMove = -1;
     int lastRoundValue = -1000;
     int bestFoundValue = -1;
     searchDepth = 1;
     int alpha = -10000;
     int beta = 10000;
     MoveList pv = new MoveList();

     while(!(searchDepth>(42-p.numberOfDiscs()))) {

          pv = new MoveList();
          long startTime = 0;
          long duration = 0;

          try {

               startTime = System.currentTimeMillis();

               bestFoundValue = StartAlphaBeta(p,searchDepth,alpha,beta,0,pv);

               duration = System.currentTimeMillis() - startTime;

          } catch (TooManyNodesException t) {
               duration = System.currentTimeMillis() - startTime;
               System.out.println("Dauer: "+duration);
               return lastRoundMove;
          }

          lastRoundMove = pv.pop();
          lastRoundValue = bestFoundValue;

          System.out.println("Bewertung in Tiefe "+searchDepth+": "+bestFoundValue);
          System.out.println("Dauer: "+duration);
          System.out.println();

          if(bestFoundValue>900 || bestFoundValue<-900) {
               break;
          }

          searchDepth += 1;

     }

     return lastRoundMove;

}
```

Listing 3.2: Iterative Deepening. Die Methode `returnMove()` bildet den äußeren Rahmen für die Alpha-Beta-Suche. Ein einfaches Verfahren zur Zeitmessung ist hier bereits berücksichtigt; zur Methode `StartAlphaBeta` siehe Anhang A.

der Suchgrenzen in der Hoffnung, dadurch die Suche zu verkürzen, ohne sich zu verschätzen und die Veränderung rückgängig machen zu müssen.

Der Grundgedanke ist, daß das Resultat, das man für die betrachtete Position aus Iteration $x - 1$ erhalten hat, mit hoher Wahrscheinlichkeit eine gute Näherung für den Wert aus Iteration x ist. Also startet man Iteration x mit einem relativ engen Suchfenster um jenes Resultat herum. Wiederum gilt, wie bei PVS: Irrt man sich, bekommt man also keinen exakten Wert zurückgeliefert, sondern einen fail-high oder fail-low, muß man die Suche mit wiederum erweiterten Suchgrenzen wiederholen.

MILTON sucht daher nur in Tiefe eins notwendigerweise mit einem Startfenster, das alle möglichen Rückgabewerte umfaßt: [−10000; +10000]. Erhält das Programm einen Score zwischen 459 und 499 zurück – d.h. einen, der auf einen möglichen Gewinn hinweist – so legt es das Fenster für die nächste Suchiteration auf [Score; Score+2]; denn es geht davon aus, daß in dieser Iteration keine neuen Varianten gefunden werden und das vorige Suchergebnis lediglich in einer um einen Halbzug größeren Tiefe bestätigt werden wird, was zu einer Rückgabe von Score+1 führte. Erhält MILTON einen Wert größer als 600 zurück – d.h. einen, der einen sicheren Gewinn signalisiert – liegt das nächste Intervall bei [Score−1; Score+1], da sich Scores dieser Art nicht in Abhängigkeit von der Suchtiefe verändern (siehe Abschnitt über Interior Node Recognition). Für Verlustresultate geht man analog vor. Bei einem Rückgabewert von 0, der auf Unentschieden bzw. eine ungeklärte Stellung schließen läßt, ergibt sich ein nachfolgendes Fenster von [−2; +2]. Zum Verständnis dieses Absatzes betrachte man auch den Abschnitt über die Stellungsbewertung.

Im Falle eines fail-lows jeder dieser Varianten erweitert das Programm das Suchintervall wieder auf [−10000; Score+2], um die Suche in der entsprechenden Tiefe zu wiederholen; bei fail-highs auf [Score−2; +10000]. Sollte es zu *Suchinstabilität* kommen (siehe [12]), sollte sich also an einen fail-high direkt ein fail-low anschließen oder umgekehrt, was zu widersprüchlichen Aussagen über die Positionsbewertung führen würde, startet die Suche in der entsprechenden Tiefe erneut mit einem sicheren Fenster von [−10000; +10000].

Im self-play-Test zeigte sich ein Gewinnverhältnis von 39 : 36 für den Algorithmus mit Aspiration Windows gegenüber der Suche mit festem Suchintervall [−10000; +10000]. In einer Reihe von Testpositionen erwies sich die Knotenersparnis darüber hinaus als überzeugend, selbst wenn sie keinen spielentscheidenden Charakter hatte.

```
try {
     while(true) {

          startTime = System.currentTimeMillis();

          bestFoundValue = StartAlphaBeta(p,searchDepth,alpha,beta,0,pv,Zkey);

          duration = repeated? duration + System.currentTimeMillis() - startTime :
               System.currentTimeMillis() - startTime;

          if(bestFoundValue<=alpha && !repeated) {
               repeated = true;
               alpha = -10000;
               beta = bestFoundValue + 2;
               continue;
          }
          if(bestFoundValue>=beta && !repeated) {
               repeated = true;
               beta = 10000;
               alpha = bestFoundValue - 2;
               continue;
          }
          if(repeated && (bestFoundValue<=alpha || bestFoundValue>=beta)) {
               alpha = -10000;
               beta = 10000;
               continue;
          }
          if(bestFoundValue==0) {
               alpha = bestFoundValue - 2;
               beta = bestFoundValue + 2;
          }
          if(bestFoundValue>0 && bestFoundValue<600) {
               alpha = bestFoundValue;
               beta = bestFoundValue + 2;
          }
          if(bestFoundValue<0 && bestFoundValue>-600) {
               alpha = bestFoundValue - 2;
               beta = bestFoundValue;
          }
          if(bestFoundValue>600) {
               alpha = bestFoundValue-1;
               beta = bestFoundValue+1;
          }
          if(bestFoundValue<-600) {
               alpha = bestFoundValue-1;
               beta = bestFoundValue+1;
          }
          break;
     }
} catch (TooManyNodesException t) {
     duration = repeated? duration + System.currentTimeMillis() - startTime :
          System.currentTimeMillis() - startTime;
     System.out.println("Dauer: "+duration);
     return lastRoundMove;
}
```

Listing 3.3: Aspiration Windows. Dargestellt ist der zentrale try-catch-Block der Methode `returnMove()`.

4 Datenstrukturen

La perfection se compose de minuties.
Joseph Joubert

4.1 Bitboards

Beim Design jedes Spiels muß ein Weg gefunden werden, die Information über dessen aktuellen Zustand - das heißt im Falle von Vier Gewinnt, die Information über Farbe und Lage der bereits gesetzten Scheiben - in einer geeigneten Art und Weise zu codieren. Hierbei ist die Effizienz, mit der bestimmte Manipulationen (z.B. „mache Zug X“) und Abfragen (z.B. „hat Spieler Y gewonnen?“) an der Datenstruktur vorgenommen werden können, von größter Wichtigkeit. In der Schachprogrammierung hat sich nun schon früh herausgestellt, daß der naive Ansatz – das Brett durch ein Array von Integern zu repräsentieren, von denen ein jeder für ein leeres Feld oder für eine bestimmte Figur steht – den Effizienzanforderungen in vielerlei Hinsicht nicht genügt.

Stattdessen hat sich das *Bitboard*-Prinzip durchgesetzt (siehe [9]). Zur Veranschaulichung vergleiche man die Abbildungen 4.2 und 4.3: In beiden wurde die in Abbildung 4.1 dargestellte Spielstellung codiert. Die Variante in Abb. 4.2 verwendet hierfür ein Array von 42 Integern (je 32 bit), die Variante in Abb. 4.3 jedoch lediglich zwei Variablen vom Typ long (je 64 bit). Diese Variablen stehen einerseits für die weißen, andererseits für die schwarzen Scheiben, wobei die Bits 22 bis 63 die Felder des Spielbretts von a1 bis g6 repräsentieren: Jedes einzelne Bit zeigt das Vorhandensein (1) oder das Fehlen (0) einer weißen respektive schwarzen Scheibe auf seinem jeweiligen Feld an.

Obwohl diese Codierung nicht wie ein Schach-Bitboard den vollen Gebrauch von modernen 64-Bit-Architekturen machen kann – 2∗22 Bits bleiben ungenutzt – ist leicht zu erkennen, daß sie sehr wenig Speicherplatz erfordert. Wichtiger noch sind jedoch ihre *Geschwindigkeits*vorteile durch die effiziente Beantwortung häufig gestellter Anfragen. Ein simples Beispiel hierfür: Um festzustellen, ob Spieler Weiß eine Viererkette von a2 bis d5 gelegt und somit gewonnen hat, muß man in der naiven Implementierungsvariante vier Werte aus dem Brett-Array entnehmen und jeden einzelnen mit dem Wert für „weiße Scheibe“ vergleichen. Mit Bitboards genügt eine einzige bitweise und-Operation sowie ein Vergleich (siehe Abb. 4.4). Auf ähnliche Weise kann mittels bitweiser Operatoren das Setzen und Zurücknehmen von Zügen und vieles andere realisiert werden.

Die Verwendung von Bitboards verkürzte MILTONs Suchzeit im Durchschnitt auf rund 70%. In Anhang B sind zum Vergleich mit ihren Bitboard-Pendants einige Methoden der Array-Implementierung aufgeführt.

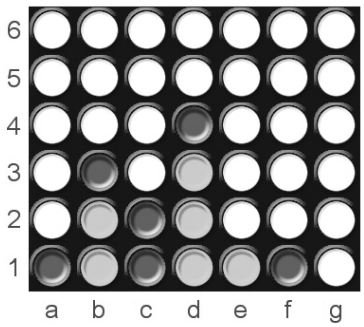


Abbildung 4.1: zu kodierende Spielstellung

`int board[][]`

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	2	0	0	0
0	2	0	1	0	0	0
0	1	2	1	0	0	0
2	1	2	1	1	2	0

Abbildung 4.2: Kodierung durch Integer-Array: 0 für „leeres Feld“, 1 für „weiße Scheibe“, 2 für „schwarze Scheibe“

`long whiteDiscs`

```
0000000000000000010001110000000000110000000000000000000000000000
```

`long blackDiscs`

```
0000000000010000000010000000110001000000010000000000000000000000
```

[g6 g5 g4 g3 g2 g1 f6 f5 f4 f3 f2 f1 e6 e5 e4 e3 e2 e1 d6 d5 d4 d3 d2 d1 c6 c5 c4 c3 c2 c1 b6 b5 b4 b3 b2 b1 a6 a5 a4 a3 a2 a1 + 22 leere Bits]

Abbildung 4.3: Kodierung mittels Bitboards: Nur zwei Variablen von Typ `long` werden benötigt

```
int x = WHITEDISC;
if(getDisc(0,1)==x && getDisc(1,2)==x && getDisc(2,3)==x && getDisc(3,4)==x) return true;
```

```
if((whiteDiscs & 0x0020202020000000L)==0x0020202020000000L) return true;
```

Abbildung 4.4: Hat Weiß den Vierer a2, b3, c4, d5 gesetzt?

4.2 Hash Tables

Man betrachte die Startposition von Vier Gewinnt. Von hier aus führen die Züge 1. e1 d1 2. f1 zu derselben Stellung wie 1. f1 d1 2. e1 (Abb. 4.5); es liegt eine *Transposition* der weißen Züge vor. Tatsächlich hat der Spielbaum in einer Tiefe von drei Halbzügen 343 Blätter – diese stellen aber nur 238 verschiedene Stellungen dar. (Von den 117649 Blättern in einer Tiefe von 6 Halbzügen unterscheiden sich gar nur 16422.) Hat man nun die untere Position von Abb. 4.5 bereits über 1. e1 d1 2. f1 erreicht und bewertet – direkt durch die Evaluationsfunktion oder indirekt durch tiefere Suche – ergibt es Sinn, diese nicht von neuem durchzurechnen, wenn man sie an anderer Stelle über 1. f1 d1 2. e1 erreicht.

Stattdessen ruft man den gesuchten Wert aus einer *Hash Table* (auch Transposition Table genannt) ab. Wann immer die Suche einen neuen Knoten erreicht, überprüft sie zuerst, ob passende und ausreichende Information über die betreffende Stellung bereits darin gespeichert ist: Wenn ja, kann sie als Horizontknoten betrachtet und der Wert aus der Hash Table für sie übernommen werden. Wenn nein, wird sie regulär durchsucht und das gefundene Ergebnis in einem neuen Hasheintrag gespeichert bzw. der vorhandene Eintrag im „Suchgedächtnis“ aufgefrischt.

Bei der Implementierung von Hash Tables besteht das entscheidende Problem in der Definition von *passender und ausreichender Information*. Zwei Bedingungen müssen hierfür erfüllt sein: Erstens muß die Suche, die den eingetragenen Wert ergeben hat, mindestens ebenso tief gewesen sein wie die Suche, für die er verwendet werden soll; hierfür benötigt jeder Hasheintrag neben dem Attribut `Wert` auch ein Attribut `Tiefe`. Zweitens ist aufgrund des Prinzips des Alpha-Beta-Algorithmus nicht von jeder Position ein exakter Wert bekannt (und benötigt), sondern oftmals nur eine untere oder obere Schranke; dies macht eine Flag notwendig, welche die Werte `hashexakt`, `hashalpha` (gespeicherter Wert ist untere Schranke) und `hashbeta` (gespeicherter Wert ist obere Schranke) annehmen kann.

Transposition Tables verhindern aber nicht nur redundante Berechnungen. Eine häufig mindestens ebenso wichtige Funktion wird durch ein weiteres Attribut jedes Hasheintrags realisiert: das Feld für den *besten Zug*. Hat man z.B. für einen Knoten, der bis zu einer Tiefe von 8 Halbzügen durchsucht werden soll, in der Hash Table nur Informationen über seinen Wert bis zur Tiefe 6, dann kann man diesen Wert zwar nicht unmittelbar übernehmen, sondern muß den Suchbaum regulär weiterverfolgen – der erste Zug jedoch, den man dabei berücksichtigen sollte, ist derjenige, der sich bei der 6-Halbzüge-Suche als der beste erwiesen hatte. Die Chancen sind relativ groß, daß er auch in einer tieferen Suche nicht widerlegt wird und somit die Zugsortierung optimiert. Man vergleiche hierzu

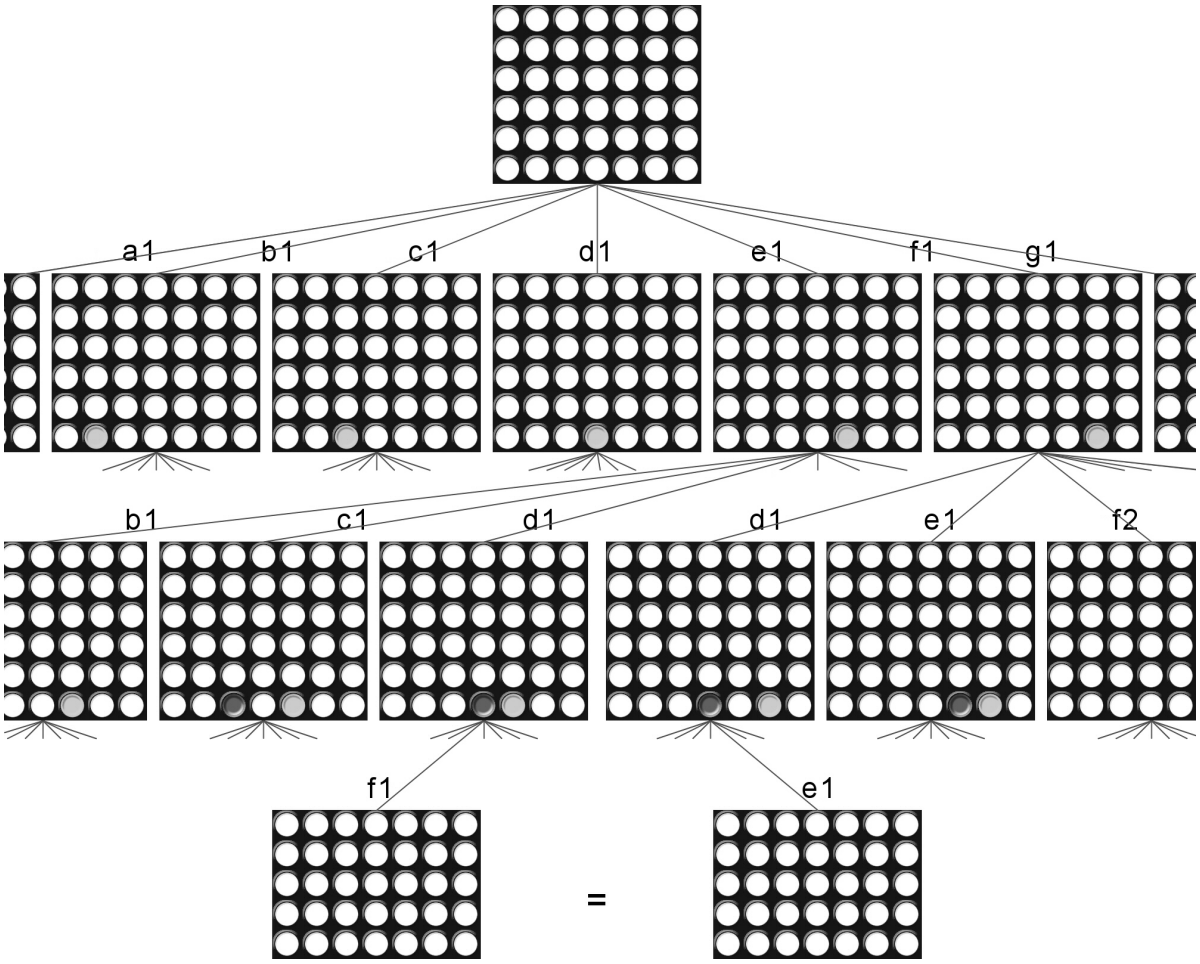


Abbildung 4.5: Transpositionen

die Abbildungen 4.6 und 4.7.[1]

Ein wichtiger Nebenaspekt des Hash-Table-Prinzips ist die Frage, wann ein Hasheintrag zu überschreiben ist; wie der Name ja bereits sagt, muß die extrem hohe Anzahl aller möglichen Spielstellungen mittels einer Hashfunktion auf eine Hashtabelle praktikabler Größe abgebildet werden (Details hierzu im Abschnitt über Zobrist Keys). Hierdurch kommt es zu Kollisionen zwischen verschiedenen Stellungen, die dieselbe Hashadresse beanspruchen. Zwei der einfachsten möglichen Ersetzungsstrategien sind: „Immer ersetzen“ (sorgt für stets aktuelle Hashinformationen, löscht aber unter Umständen wertvolle Ergebnisse aus tiefen Suchen, um bloße Evaluationsergebnisse zu speichern) und „Ersetzen bei ausreichender Tiefe“ (sorgt für ein Ansammeln aussagekräftigerer Werte aus tieferen Suchen – die allerdings veralten können und somit lediglich Platz blockieren). Hier besteht viel Raum für Experimente.

In MILTON setze ich drei verschiedene Hash Tables ein, deren Größe auf eine beliebige Zweierpotenz angepaßt werden kann: Table 1 verwendet die Immer-ersetzen-Strategie, Table 2 verwendet Ersetzen-bei-größerer-Tiefe, und Table 3 dient ausschließlich zum Speichern exakter Evaluationen aus der Interior Node Recognition, wie sie im Kapitel über Stellungsbewertung beschrieben wird. Falls man bei der Hashsuche sowohl in Table 1 als auch Table 2 einen Treffer erhält, hat es sich experimentell bewährt, den Eintrag aus Table 1 zu bevorzugen. Hits in Table 3 überschreiben alles; diese Hash Table ist die einzige, die nicht vor jedem Zug gelöscht wird, um ein Veralten der Information sowie verschiedene Suchinkonsistenzen zu vermeiden. Ihre Funktion ist im wesentlichen die Beschleunigung des Endspiels.

Wie zahlreich Transpositionen in Vier Gewinnt auftreten, verdeutlichen die folgenden Zahlen: Bei einer iterierten Suche bis zur Tiefe 12 sparen die Transposition Tables in MILTON im Mittelspiel rund 80-85% der Knoten. Da durch den zusätzlichen Aufwand für Hasheinträge und Hashabfragen die durchschnittliche Suchzeit pro Knoten um bis zu 20% steigt, wurde der self-play-Test entsprechend angepaßt, indem der Alpha-Beta-Version mit Hash Tables 15% weniger Knoten pro Zug zugestanden wurden. Das Ergebnis war dennoch eindeutig: 50 : 25 für die Version mit Hash Tables.

4.3 Zobrist Keys

Jeder Eintrag in der Hash Table speichert Informationen über eine bestimmte Position. Im Verlaufe der Suche müssen diese Positionen viele Tausend Mal pro Sekunde mit der aktuellen Stellung verglichen werden; hierfür wären die Speicherung und der Vergleich jedes einzelnen Felds des Spielbretts bei weitem zu zeitaufwendig.

Stattdessen verwendet man eine Signatur, einen *Hashwert* (auch: Hashcode, Hashkey) von typischerweise 64 Bit Länge, der mittels einer speziellen *Hashfunktion* aus der Position errechnet wird. Dieser Wert repräsentiert die Position – und die letzten k Bits dieses Werts dienen in einer Hashtabelle der Größe 2^k zugleich als *Hashadresse*. Da nun

[1] Im Wurzelknoten des Suchbaums werden alle Hash Tables *nur* zur Zugsortierung getestet, da eine direkte Wertrückgabe mit einer leeren Principal Variation verbunden wäre und somit keine Zugentscheidung ermöglichte; siehe Anhang A.

mögliche Züge	**bester Zug durchsucht als**						
	erster	zweiter	dritter	vierter	fünfter	sechster	siebter
2	83,71%	16,29%					
3	82,45%	11,96%	5,59%				
4	79,87%	13,35%	4,89%	1,88%			
5	80,48%	12,19%	4,27%	1,94%	1,13%		
6	82,29%	9,91%	3,53%	2,06%	1,33%	0,87%	
7	83,32%	8,05%	2,94%	2,04%	1,68%	1,18%	0,78%

Abbildung 4.6: Zugsortierung mit Hash Tables. 90.881.838 Positionen des self-play-Tests wurden daraufhin untersucht, als wievielter unter allen möglichen Zügen der (die Korrektheit von `evaluate()` vorausgesetzt) beste Zug jeweils durchsucht wurde.

mögliche Züge	**bester Zug durchsucht als**						
	erster	zweiter	dritter	vierter	fünfter	sechster	siebter
2	60,43%	39,57%					
3	65,62%	23,97%	10,40%				
4	65,07%	22,64%	8,70%	3,59%			
5	69,50%	18,33%	6,93%	3,32%	1,93%		
6	72,91%	13,96%	5,53%	3,33%	2,38%	1,90%	
7	72,34%	11,71%	4,87%	3,66%	3,04%	2,34%	2,04%

Abbildung 4.7: Zugsortierung ohne Hash Tables. 141.016.772 Positionen wurden untersucht.

64 Bit nicht ausreichen, um eine beliebige Stellung eindeutig zu beschreiben, muß man die Möglichkeit von *Hashfehlern* (zwei verschiedene Stellungen mit identischem Hashcode) prinzipiell in Kauf nehmen, versucht diese aber durch das Design der Hashfunktion zu minimieren.

Zwei Anforderungen werden an die Hashfunktion gestellt: Erstens eine möglichst zufällige Verteilung der Spielstellungen über die Hashtabelle; auch ähnliche Stellungen sollten keine ähnlichen Hashindices aufweisen, um „Klumpenbildung“ und Kollisionen (zwei verschiedene Stellungen mit evtl. unterschiedlichem Hashcode, aber identischer Hashadresse) zu reduzieren. Zweitens eine möglichst effiziente Erzeugbarkeit der Hashkeys im Suchprozess. Beide Ziele werden von einer Hashfunktion, wie sie Al Zobrist 1970 an der University of Wisconsin vorstellte ([11] - für eine anschauliche Darstellung siehe [7]), auf elegante Weise realisiert. Nahezu alle heutigen Schachprogramme, wie auch MILTON, verwenden daher ein ähnliches Verfahren – man spricht von *Zobrist Keys*.

Ich lege hierfür ein Array von 84 64-Bit-Zufallszahlen an – jede steht für eine weiße oder eine schwarze Scheibe auf einem der 42 Spielfelder (Abb. 4.8). Den Hashwert einer gegebenen Position ermittle ich, indem ich diejenigen Zufallszahlen, die den vorhandenen Scheiben entsprechen, bitweise mittels exklusivem Oder verknüpfe[2] (Abb. 4.9; ich notiere die Operation als $\oplus$, es gilt also $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$). Um nun den Hashwert zu berechnen, der zu einem Tochterknoten dieser Stellung im Suchbaum gehört, genügt es also, eine einzige Exklusiv-Oder-Operation durchzuführen: Ich verknüpfe hierbei den Hashcode der ursprünglichen Stellung mit der Zufallszahl des auszuführenden Zugs. So ist es möglich, die Anwendung der Hashfunktion auf Spielpositionen inkrementell zu handhaben. Listing 4.1 zeigt den um Hash Tables mit Zobrist Keys erweiterten Alpha-Beta-Algorithmus.

4.4 Eröffnungsbuch

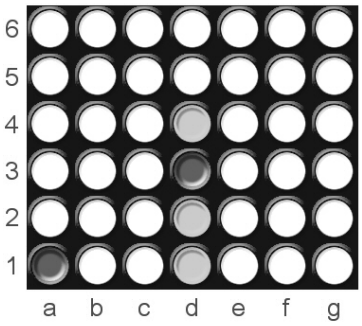
Die Heuristiken einer Evaluationsfunktion haben die Aufgabe, das Ergebnis anzuzeigen, das ein Spiel aus der von ihnen betrachteten Position heraus – soweit man dies rein statisch analysieren kann – mit größter Wahrscheinlichkeit haben wird. Ihre Aufgabe wird um so leichter, je näher die zu evaluierenden Stellungen dem *Spielende* rücken: Im Schach z.B. durch die deutlich reduzierte Figurenanzahl gegenüber dem Mittelspiel und durch die Möglichkeit der Anwendung spezieller Lösungsregeln auf gewisse Endspiele, in Vier Gewinnt durch die verringerte Anzahl freier Spielfelder, deren Belegung neue Chancen oder Bedrohungen für die Spieler bedeuten könnte. Dieser Sachverhalt ist ja der hauptsächliche Grund, weshalb eine möglichst tiefe Suche für die Spielstärke von Vorteil ist.

Auch die stärksten Suchverfahren müssen nun gewöhnlich kapitulieren, wenn es um die Zugentscheidung in einer so frühen Spielphase geht, daß der Spielbaum nicht tief genug entwickelt werden kann, um den Verlauf des Spiels annähernd realistisch einschätzen zu können. Nicht im Suchalgorithmus liegt hierbei das eigentliche Problem, sondern in

[2] Man beachte, daß es in Vier Gewinnt nicht nötig ist, den am Zug befindlichen Spieler mitzukodieren, da man ihn aus jeder Spielposition eindeutig bestimmen kann.

piece-square	**64-Bit-Zufallszahl**
Weiß auf a1	1010100000101011011111101011101100111111010001100011000000100100
Schwarz auf a1	1010111111110111100001000100011111010100011101101110110110111011
Weiß auf a2	0111001010111111101100111011100010101000000100011101010110111001
Schwarz auf a2	0000001000011011101111000010110000111110111010011010001000110100
Weiß auf a3	1110100000110010011010100101101010011111100010101010000101111010
...	...
Weiß auf d1	0101101001010111011111000001100011000011110110011000001000010110
Schwarz auf d1	1101100001111000010011001111110010010100011110001000001101010011
Weiß auf d2	0010000001100000111100011011001110001101110010100000000000001011
Schwarz auf d2	0000111101011001110011011111010011110011101010100100100001100100
Weiß auf d3	1000101100110111110001110110100000001001100001011000101001000000
Schwarz auf d3	1100101101011110100001001000100100011001000010011100001111011011
...	...
Schwarz auf g6	0110100110010001111100000001101100111010010111111100010100101110

Abbildung 4.8: Zobrist Keys: Die Piece-Square Table




```
  1010111111110111100001000100011111010100011101101110110110111011 (Schwarz auf a1)
⊕ 0101101001010111011111000001100011000011110110011000001000010110 (Weiß auf d1)
⊕ 0010000001100000111100011011001110001101110010100000000000001011 (Weiß auf d2)
⊕ 1100101101011110100001001000100100011001000010011100001111011011 (Schwarz auf d3)
⊕ 0100101001100010000110110011111100100101111110100000011110111011 (Weiß auf d4)
= 0101010011111100100101100101101111001000100110001010001100001010
```

Abbildung 4.9: Zobrist Keys: Bestimmung eines Hashcodes

```
public int AlphaBeta(Position p, int depth, int alpha, int beta,
     double extension, MoveList pv, long Zkey, int move) throws TooManyNodesException {

     if(nodecount>1000000) {
          throw new TooManyNodesException();
     }

     nodecount += 1;
     int hashflag = HASHALPHA;
     int index = (int)Zkey&(HASHSIZE-1);

     int hvalue = -1;
     bestMove = -1;
     int newbeta = beta;
     int newalpha = alpha;
     if(HashTable2[index].key == Zkey) {
          if(HashTable2[index].depth >= (depth+extension)) {
               if(HashTable2[index].flag==HASHEXACT) {
                    hvalue = HashTable2[index].value;
               }
               if(HashTable2[index].flag==HASHALPHA && HashTable2[index].value <= alpha) {
                    hvalue = alpha;
               }
               if(HashTable2[index].flag==HASHALPHA && HashTable2[index].value < beta) {
                    newbeta = HashTable2[index].value;
               }
               if(HashTable2[index].flag==HASHBETA && HashTable2[index].value >= beta) {
                    hvalue = beta;
               }
               if(HashTable2[index].flag==HASHBETA && HashTable2[index].value > alpha) {
                    newalpha = HashTable2[index].value;
               }
          }
          bestMove = HashTable2[index].move;
     }
     if(HashTable1[index].key == Zkey) {
          if(HashTable1[index].depth >= (depth+extension)) {
               if(HashTable1[index].flag==HASHEXACT) {
                    hvalue = HashTable1[index].value;
               }
               if(HashTable1[index].flag==HASHALPHA && HashTable1[index].value <= alpha) {
                    hvalue = alpha;
               }
               if(HashTable1[index].flag==HASHALPHA && HashTable1[index].value < beta) {
                    newbeta = HashTable1[index].value;
               }
               if(HashTable1[index].flag==HASHBETA && HashTable1[index].value >= beta) {
                    hvalue = beta;
               }
               if(HashTable1[index].flag==HASHBETA && HashTable1[index].value > alpha) {
                    newalpha = HashTable1[index].value;
               }
          }
          bestMove = HashTable1[index].move;
     }
     if(HashTable3[index].key == Zkey) {
          hvalue = HashTable3[index].value;
     }
     beta = newbeta;
     alpha = newalpha;
     if(hvalue != -1) {
          pv.length = 0;
          return hvalue;
     }
```

```
    if(p.isWon(!p.toMove,move)) {
        pv.length = 0;
        addHashEntries(Zkey, index, depth+extension, -1000+p.numberOfDiscs(), HASHEXACT, -1);
        return -1000+p.numberOfDiscs();
    }
    if(p.isDraw()) {
        pv.length = 0;
        addHashEntries(Zkey, index, depth+extension, 0, HASHEXACT, -1);
        return 0;
    }
    if(depth <= (0-extension)) {
        pv.length = 0;
        value = evaluate(p, false);
        addHashEntries(Zkey, index, depth+extension, value, HASHEXACT, -1);
        return value;
    }

    MoveList mypv = new MoveList();

    int[] moves = generateMoves(p, searchDepth-depth);
    sort(moves, p);

    if(moves[7]==1) {
        extension += 0.5;
    }

    for(int i=0;i<moves[7];i++) {
        Zkey = Zkey^(p.toMove ? zobristKeys[moves[i]][0] : zobristKeys[moves[i]][1]);
        value = -AlphaBeta(p, depth-1, -beta, -alpha, extension, mypv, Zkey, moves[i]);
        p.undoMove(moves[i]);
        Zkey = Zkey^(p.toMove ? zobristKeys[moves[i]][0] : zobristKeys[moves[i]][1]);
        if(value>=beta) {
            if(moves[7]==1) {
                addHashEntries(Zkey, index, depth+extension-0.5, value, HASHBETA, moves[i]);
            }
            else {
                addHashEntries(Zkey, index, depth+extension, value, HASHBETA, moves[i]);
            }
            return beta;
        }
        if(value>alpha) {
            hashflag = HASHEXACT;
            alpha = value;
            pv.setFirst(new MoveListElement(moves[i]));
            if(mypv.length!=0) {
                pv.setRest(mypv);
            }
            pv.length = mypv.length + 1;
        }
    }
    if(moves[7]==1) {
        addHashEntries(Zkey, index, depth+extension-0.5, alpha, hashflag,
            (hashflag == HASHEXACT ? pv.pop() : -1));
    }
    else {
        addHashEntries(Zkey, index, depth+extension, alpha, hashflag,
            (hashflag == HASHEXACT ? pv.pop() : -1));
    }
    return alpha;

}
```

Listing 4.1: MILTONs Alpha-Beta mit Hash Tables

der Schwierigkeit der Positionsbewertung in den Horizontknoten – der, wenn ein *Eröffnungszug* gesucht wird, meist in einer noch recht offenen Phase des Mittelspiels verläuft. Die Problematik wird noch dadurch verschärft, daß gerade jene schwer einzuschätzenden Eröffnungszüge wesentlichen Einfluß auf den Spielverlauf haben; mit ihnen werden die grundlegenden strategischen Entscheidungen getroffen.

Daher können Computer, ebenso wie Menschen, nicht auf eine gute Kenntnis der Eröffnungstheorie verzichten. Ebenso wie für Schach oder Go eine reiche Eröffnungsliteratur existiert, durch deren Studium ein starker menschlicher Spieler das Spiel schon früh in eine von ihm favorisierte Richtung lenken kann, so wurden für Computer entsprechende *Eröffnungsbücher* konzipiert, die dem Suchalgorithmus in der Eröffnungsphase des Spiels offline berechnetes und verifiziertes spieltheoretisches Wissen zur Verfügung stellen.

In MILTON verwende ich eine von John Tromp [14] stammende Eröffnungsdatenbank, deren Berechnung auf den Sun und SGI Workstations am Centrum voor Wiskunde en Informatica in Amsterdam ungefähr 40.000 Stunden in Anspruch nahm. Sie enthält alle nichttrivialen Stellungen mit acht Scheiben und deren Wert (Gewinn für Weiß, Gewinn für Schwarz oder Unentschieden), wobei unter „nichttrivial“ zu verstehen ist, daß keiner der Spieler eine akute Drohung besitzt (Ausschnitt in Abb. 4.10).[3] Listing 4.2 zeigt den um Eröffnungsbuchzugriffe erweiterten Code des Alpha-Beta-Algorithmus. Anzumerken ist hierzu, daß asymmetrische Stellungen in Tromps Datenbank nur in jeweils einer Variante (gespiegelt an der d-Spalte) vorhanden sind, um Speicherplatz zu sparen; dies erfordert in der Realisierung unter Umständen zwei Zugriffe pro Stellung. Für Details betrachte man den Code in Anhang A.

Im self-play-Test erwies sich, welch essentielle Bedeutung die ersten acht Halbzüge für den späteren Verlauf des Spiels haben: Der Algorithmus ohne Eröffnungsbuch unterlag mit 23 : 55.

```
...
x,b,b,b,b,b,o,x,b,b,b,b,b,b,b,b,b,b,x,o,o,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,o,x,b,b,b,b,b,d
x,b,b,b,b,b,o,x,b,b,b,b,b,o,b,b,b,b,b,x,o,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,o,x,b,b,b,b,b,w
x,b,b,b,b,b,o,x,b,b,b,b,b,b,b,b,b,b,x,o,b,b,b,b,b,o,b,b,b,b,b,b,b,b,b,b,o,x,b,b,b,b,b,w
x,b,b,b,b,b,o,x,o,b,b,b,b,b,b,b,b,b,x,o,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,o,x,b,b,b,b,b,l
...
```

[a1 a2 a3 a4 a5 a6 b1 b2 b3 b4 b5 b6 c1 c2 c3 c4 c5 c6 d1 d2 d3 d4 d5 d6 e1 e2 e3 e4 e5 e6 f1 f2 f3 f4 f5 f6 g1 g2 g3 g4 g5 g6 + Wert]

Abbildung 4.10: Ausschnitt aus dem Eröffnungsbuch: **x** für „weiße Scheibe“, **o** für „schwarze Scheibe“, **b** für „leeres Feld“ – **w** für „Gewinn für Weiß“, **l** für „Gewinn für Schwarz“, **d** für „Unentschieden“.

[3] Wie Lars Bremer [15] richtig herausstellt, ist dies eine etwas vorschnelle Definition von Trivialität; denn während Positionen mit einer akuten *weißen* Drohung in der Tat im nächsten, also neunten Halbzug entschieden werden können – Weiß kann schließen und gewinnt – ergeben sich aus Positionen mit einer akuten *schwarzen* Drohung, die Weiß aber im nächsten Zug verhindern kann, teilweise keineswegs triviale Spiele. Daher kann MILTON in vielen Fällen nicht an der Acht-Halbzüge-Grenze anhalten, auch wenn das Eröffnungsbuch aktiviert ist.

```
.
.
.

if(p.isDraw()) {
     pv.length = 0;
     addHashEntries(Zkey, index, depth+extension, 0, HASHEXACT, -1);
     return 0;
}
if(opening) {
     if(p.numberOfDiscs()==8) {
          value = searchBook(p);
          if(value != -1) {
               pv.length = 0;
               addHashEntries(Zkey, index, depth+extension, value, HASHEXACT, -1);
               return value;
          }
     }
}
if(depth <= (0-extension)) {
     pv.length = 0;
     value = evaluate(p, false);
     addHashEntries(Zkey, index, depth+extension, value, HASHEXACT, -1);
     return value;
}

.
.
.
```

Listing 4.2: Eröffnungsbuchzugriff in MILTONs Alpha-Beta

5 Zuggenerierung und Zugsortierung

Aans nach'm annern, wie die Kleeß 'gesse wern.
hessisches Sprichwort

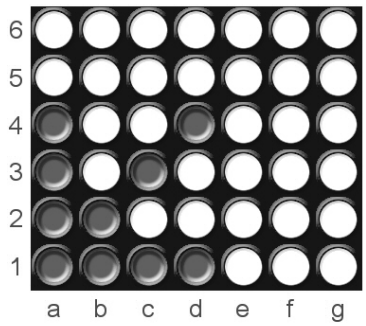
5.1 Grundlagen

Im Kapitel über grundlegende Suchtechniken wurde bereits mehrfach auf die Bedeutung einer möglichst guten Zugsortierung für die Wirksamkeit des Alpha-Beta-Pruning und seiner algorithmischen Verfeinerungen hingewiesen. Wie kommt man nun zu einer realistischen Einschätzung der Qualität eines Zuges – insbesondere unter den strengen Effizienzbedingungen, denen Algorithmen unterworfen sind, welche in jedem Knoten des Suchbaums angewendet werden müssen?

In MILTON ziehe ich für diese Einschätzung zwei Quellen zu Rate: Primär die besten Züge aus den *Transposition Tables*, soweit für die betrachtete Stellung vorhanden; und sekundär eine Sortierung nach der *Lage* der legalen Züge auf dem Spielbrett. Hierfür verwende ich ein statisches Array (Abb. 5.1a), das jeden Zug von a1 bis g6 mit der Anzahl der möglichen Vierer bewertet, die sich *theoretisch* durch dieses Feld legen ließen - minimal drei und maximal dreizehn (Beispiel in Abb. 5.1b).[1]

3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

(a) Feldwerte zur Zugsortierung



(b) Beispiel: Feldwert von a1 ist drei

Abbildung 5.1: statische Feldwerte

[1] Die untere und obere Hälfte des Spielbretts weisen symmetrische Möglichkeiten auf. Experimentell hat sich jedoch bewährt, die untere Hälfte zu bevorzugen; daher bewertet MILTON tatsächlich Felder in der oberen Hälfte mit *Anzahl mögl. Vierer* $* 2$ und Felder in der unteren Hälfte mit (*Anzahl mögl. Vierer* $* 2) + 1$.

Dieses schnelle Sortierverfahren erwies sich bisher gegenüber allen komplexeren Algorithmen als überlegen. Zu erwähnen wäre hier der zunächst vielversprechend erscheinende Ansatz, statt des erwähnten Arrays für jeden Spieler ein eigenes Array zu speichern, das im Verlaufe der Suche dynamisch mit Information über die in jedem einzelnen Knoten für den jeweiligen Spieler *tatsächlich* noch möglichen Vierer versorgt wird (Beispiel in Abb. 5.2). Der Wert, den man der Zugsortierung zugrundelegen könnte, wäre in diesem Fall z.B. die Summe der eigenen Vierer, in die sich ein Zug einfügt, und der gegnerischen Vierer, die dieser Zug verhindert.

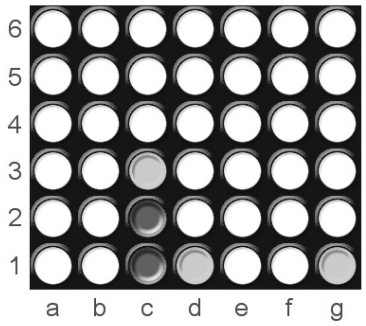
Wie die folgenden Zahlen beweisen, stellte dies jedoch keinen Fortschritt dar: Im self-play-Test gewann das dynamische Verfahren zwar mit 34 : 32 gegen den simpleren Algorithmus mit statischem Array – wenn man jedoch den im Durchschnitt um 38% höheren Suchaufwand pro Knoten berücksichtigte, indem man die Anzahl der Knoten pro Zug entsprechend reduzierte, verlor es mit 34 : 40. Solange sich also keine Methode findet, die dynamischen Arrays deutlich effizienter zu behandeln, bleibt MILTON bei der guten statischen Lösung.

Zur Zuggenerierung lediglich zwei Anmerkungen: Wie schon im Abschnitt über Search Extensions erwähnt, gibt `GenerateMoves()` nur dann alle legalen Züge zurück, wenn keiner der beiden Spieler eine akute Drohung besitzt (≈70% der Fälle). Besitzt der am Zug befindliche Spieler (mindestens) eine solche, können alle anderen Teilbäume abgeschnitten werden, da kein besserer Zug existieren kann als einer, der den Gewinn im nächsten Halbzug ermöglicht (≈8% der Fälle); hat sein Gegner jedoch (mindestens) eine (≈22% der Fälle), so sind ebenfalls alle anderen Teilbäume irrelevant, da a) im Falle genau einer akuten Drohung nur ein einziger Zug den Verlust im nächsten Halbzug verhindern kann, und b) im Falle mehrerer akuter Drohungen jeder beliebige Zug zum Verlust im nächsten Halbzug führt und somit die Betrachtung eines einzigen ebenfalls genügt.[2]

5.2 Killer Moves

Ein weiteres Verfahren aus der Schachprogrammierung, welches die Zugsortierung verbessern und so die Anzahl der beta-Cutoffs maximieren soll, ist die *Killer-Heuristik*. Sie basiert auf der Idee, daß ein Zug, der sich in einer bestimmten Stellung in Tiefe x im Suchbaum als der beste erwiesen hat, auch in anderen Stellungen der Tiefe x funktionieren könnte und daher, falls er auch dort legal ist, zuerst ausprobiert werden sollte. Man betrachte zum Beispiel Abbildung 5.3: Zieht Schwarz 6. ... a1, so kann Weiß mit 7. d3 eine Doppeldrohung aufbauen, die für Schwarz den Verlust nach zwei Halbzügen bedeutet. Auch Schwarz' Züge b4, c4, f1 und g2 lassen sich mit 7. d3 widerlegen; er ist ein *Killer Move*. Immer, wenn die Suche die entsprechende Tiefe erreicht, lohnt es sich also, d3 als ersten Zug zu untersuchen.

[2] Weil der Alpha-Beta-Algorithmus auf dem Konzept der Tiefensuche basiert, ist zu jedem Zeitpunkt im Verlauf der Suche nur ein einzelner Zweig des Suchbaums von Interesse. Daher erzeugt `GenerateMoves()` nicht wirklich bei jedem Aufruf ein neues Array von Zügen, sondern überschreibt ein zweidimensionales Array, dessen erster Index der Suchtiefe entspricht.



(a) Beispielposition

3	4	5	7	5	4	3
4	6	7	10	8	5	4
4	8	9	13	9	7	5
5	7	9	11	10	8	5
3	4	0	6	6	5	4
2	1	0	3	3	3	3

(b) Feldwerte für Weiß

3	4	4	7	5	3	3
3	6	6	10	6	6	4
4	6	8	8	11	8	3
4	5	0	9	8	6	4
4	4	5	7	7	5	3
1	2	1	0	1	2	0

(c) Feldwerte für Schwarz

Abbildung 5.2: dynamische Feldwerte

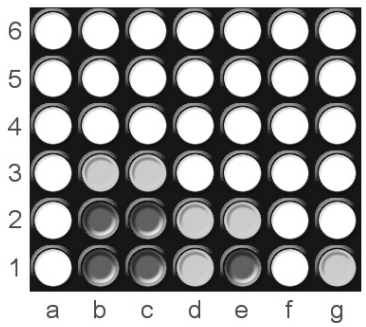


Abbildung 5.3: Killer Move in Tiefe 2: d3

Zwei Varianten der Heuristik lassen sich unterscheiden, weshalb viele Schachprogramme auch zwei Killer Moves für jede Suchtiefe bestimmen: Man kann für jede Tiefe ein Array im Speicher halten, in dem man während des Suchprozesses die *Anzahl* der Cutoffs jedes möglichen Zugs mitzählt – anhand dieses Arrays läßt sich der über die gesamte bisher untersuchte Breite des Suchbaums wahrscheinlichste Killer-Kandidat ermitteln. Man kann aber auch stets denjenigen Zug bevorzugen, der den *letzten* Cutoff auf gleicher Tiefe bewirkt hat, und sich auf die relative Nähe und damit Ähnlichkeit der Stellungen zueinander verlassen. In jedem Fall hat die Sortierung nach Killer Moves eine höhere Priorität als die statische Sortierung, aber eine niedrigere als die Sortierung nach den besten Zügen aus den Hash Tables.

Beide Verfahren haben sich in MILTON allerdings als wenig vielversprechend erwiesen. Sei es, weil derartige Killer Moves in Vier Gewinnt deutlich seltener auftreten als im Schach, oder sei es, weil die Zugerzeugung für die meisten relevanten Fälle ohnehin nur einen Zug zurückgibt – keine der vielen überprüften Varianten konnte in puncto Sortierungsqualität, Knotenzahl oder self-play gegen MILTON ohne Killer Moves bestehen.

6 Stellungsbewertung

A guess? You, Spock? That's extraordinary.
James Tiberius Kirk

6.1 Evaluationsfunktion

Die zentrale Rolle der Evaluationsfunktion und ihre Verknüpfung mit dem Suchalgorithmus wurden in den vorhergehenden Abschnitten schon verschiedentlich besprochen. `evaluate()` führt eine statische Analyse der Positionen an den Blättern des Suchbaums durch und gibt numerische Werte zurück, die Bewertungen dieser Positionen aus der Sicht des am Zug befindlichen Spielers darstellen; Bewertungen, die innerhalb des Alpha-Beta-Algorithmus miteinander verglichen und so zur Steuerung des Suchvorgangs benutzt werden, Bewertungen, deren Güte letzten Endes die Qualität der Zugentscheidung bestimmt. Während Alpha-Beta in seiner Grundform vom betrachteten Spiel unabhängig ist und zur Zugerzeugung lediglich die Spielregeln benötigt werden, konzentriert sich in den Heuristiken von `evaluate()` das Wissen des Programmierers über *Strategie und Taktik*. Die Entscheidung, welche Eigenschaften einer Stellung hier Berücksichtigung finden sollen, wie man diese effizient ermitteln kann und wie ihre Relevanz gegeneinander abzuwägen ist, ist seine eigentlich interessanteste und kreativste Aufgabe.

In der Schachprogrammierung wird die Evaluationsfunktion quasi ausnahmslos als *Linearkombination*, d.h. als gewichtete Summe, diverser *Features* realisiert. Als wichtigstes Feature einer Stellung gilt hier die Materialbalance, gefolgt beispielsweise von Mobilität, Zentrumskontrolle, Entwicklung der Figuren, Bauernstruktur, Königssicherheit usw.; für jedes einzelne Feature können Punkte vergeben oder abgezogen werden, teilweise in Abhängigkeit von der Spielsituation oder sogar von anderen Features. Deep Blues[1] Schach-Chip erkannte z.B. circa 8000 verschiedene Einzelmerkmale, deren Feintuning durch die Einschätzungen von Schachgroßmeistern und die Erfahrungen aus Tausenden von Testläufen vorgenommen wurde.

Diese Methode liegt im Schach so zwingend nahe und ist so erfolgreich, daß eine Diskussion von Alternativen nicht mehr stattfindet. Auch ich bin für Vier Gewinnt zunächst von einem derartigen Ansatz ausgegangen: Hierbei legte ich die Unterscheidung zwischen geraden und ungeraden Drohungen zugrunde, die in den zur Strategie von Vier Gewinnt verfügbaren Texten (insbesondere [2]) als das wesentliche Element menschlicher Stellungseinschätzung dargestellt wird. Es entstanden also mehrere Varianten von Evaluationsfunktionen, die jeweils gerade und ungerade Drohungen beider Spieler zählten, diese nach Höhe gewichteten (Drohungen auf unteren Reihen lassen sich tendenziell eher

[1] Deep Blue war 1997 der erste Schachcomputer, der unter Turnierbedingungen gegen einen amtierenden Schachweltmeister gewann.

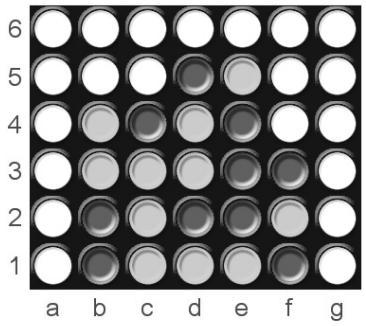
realisieren als solche auf oberen) und besondere Boni für *ungerade weiße* und *gerade schwarze* Drohungen verteilten; denn betrachtet man eine einzelne Spalte mit einer Drohung und nimmt an, daß das restliche Brett im Laufe des Spiels beliebig mit Scheiben gefüllt werden könne, ohne den Spielausgang entscheidend zu beeinflussen – akzeptiert also die Hypothese, daß keine relevanten „versteckten“ Drohungen mehr existieren außer jener, die man bereits kennt – so wird wegen der geraden Spaltenhöhe stets Weiß die ungeraden, Schwarz die geraden Felder besetzen.

Weitere Experimente umfaßten Bewertungsfunktionen, die Punkte für das Besetzen einzelner Felder verteilten – je zentrumsnäher, desto höher die Punktzahl – oder auch Funktionen, welche die Anzahl der theoretisch verbliebenen Möglichkeiten beider Spieler zur Realisierung eines beliebigen Vierers in die Betrachtung einbezogen.

Einen entscheidenden Schritt vorwärts machte MILTONs Spielstärke jedoch erst, als ich das traditionelle Schema der Evaluationsfunktion als Linearkombination zugunsten eines speziell für Vier Gewinnt entwickelten Bewertungsalgorithmus aufgab. Er basiert auf der Beobachtung, daß das Spielresultat unter der oben beschriebenen Voraussetzung, die bereits auf dem Feld befindlichen Drohungen seien die einzigen, die jenes Resultat bestimmen werden, tatsächlich auf unkomplizierte Weise bestimmt werden kann - und zwar *exakt*, nicht bloß näherungsweise, selbst in den komplexeren Fällen mit mehreren Drohungen in gleichen und/oder verschiedenen Spalten. Selbstverständlich handelt es sich bei der Stellungsbewertung nichtsdestoweniger um eine heuristische, da diese Grundannahme in vielen Fällen inkorrekt ist; insbesondere in Eröffnung und Mittelspiel. Zudem kann man mit diesem Algorithmus im allgemeinen nicht die optimalem Spiel entsprechende Endposition des Spiels errechnen – lediglich über Sieg, Unentschieden oder Niederlage kann entschieden werden. Die Resultate sind jedoch jenen klassischer Evaluationsfunktionen deutlich überlegen. Abb. 6.1 zeigt ein Beispiel hierfür.

MILTONs Algorithmus ermittelt zunächst die Lage aller weißen und schwarzen Drohungen. Von diesem Punkt an betrachtet er das Spielbrett nicht mehr; Spalten ohne Drohungen nimmt er implizit als gefüllt an. In die anderen Spalten aber „zieht“ er nun abwechselnd für beide Spieler: Hierbei wählen beide mit höchster Priorität Felder, die den eigenen Gewinn bedeuten, also akute eigene Drohungen; gibt es keine solchen, ziehen sie auf akute Drohungen des Gegners, um dessen Gewinn evtl. noch zu verhindern; hat kein Spieler eine akute Drohung, bevorzugen sie beliebige Felder, die weder direkt unter einer eigenen Drohung liegen (sonst könnte der Gegner diese im nächsten Halbzug füllen und den Vierer vereiteln) noch direkt unter einer gegnerischen (sonst würde der Gegner im nächsten Halbzug gewinnen); bleibt ihnen auch hierzu keine Möglichkeit mehr, ziehen sie auf Felder unter eigenen Drohungen, da ein solcher Zug zumindest nicht den unmittelbaren Verlust des Spiels nach sich zieht - allerdings zunächst, wenn möglich, in Spalten mit ungerader Anzahl freier Felder, denn dies bringt ggf. in der nächsten Spalte den Gegner in Zugzwang; es folgen Züge unter eigene Drohungen in Spalten mit gerader Anzahl freier Felder und, mit niederster Priorität, Züge unter gegnerische Drohungen. Steht einem der beiden Spieler schließlich kein Zug mehr offen, so sind die betrachteten Spalten vollständig gefüllt und das Spiel kann als unentschieden bewertet werden.

Entscheidend ist, daß die in jedem Schritt konkret gewählten Züge unter der Grundannahme der Evaluation keinerlei Relevanz für das Spielergebnis haben, sondern nur die



(a) Beispielposition

		s			w	
w						
w						
						s

(b) featurebasierte Bewertung: Gewinn für Schwarz. Markiert sind die Drohungen von Weiß ("w") und Schwarz („s").

		11				17
		10				16
		5			9	15
19		4			8	14
18		3			7	13
1		2			6	12

(c) MILTONs Bewertung: Gewinn für Weiß. Numeriert sind die Füllschritte des Algorithmus.

Abbildung 6.1: Vergleich zweier Evaluationsalgorithmen. Der klassische Evaluationsalgorithmus schätzt die Stellung als einen Gewinn für Schwarz ein, da Schwarz die tiefste Drohung besitzt, welche zudem noch in gerader Höhe liegt. MILTON erkennt jedoch korrekt, daß Schwarz irgendwann gezwungen sein wird, unter *irgendeine* Drohung seines Gegners zu spielen – wenn sich keine weiteren Drohungen ergeben. Dies trifft hier zu: Schwarz verliert tatsächlich in maximal zehn Halbzügen.

erwähnten Eigenschaften in Bezug auf die Lage eigener und fremder Drohungen. Dies macht den Aufbau eines Spielbaums für die Evaluation unnötig – tatsächlich genügt es, die „Füllhöhe" der Spalten hochzuzählen, ohne wirklich weiße und schwarze Scheiben zu setzen.

Für ein erwartetes Unentschieden wird schließlich Null zurückgegeben, für einen erwarteten Gewinn bzw. Verlust des (in der ursprünglichen Stellung) am Zug befindlichen Spielers aber ein Wert, der mit der Anzahl der schon gesetzten Spielscheiben steigt respektive fällt; denn die Wahrscheinlichkeit, daß die Heuristik korrekt schätzt, erhöht sich gegen Spielende, was eine Gewinnschätzung in größerer Suchtiefe interessanter macht, eine Verlustschätzung jedoch abschreckender.[2]

[2]Für einen erwarteten Gewinn wird $458 + \textit{Anzahl der Scheiben}$ zurückgegeben, für einen erwarteten Verlust $-458 - \textit{Anzahl der Scheiben}$.

6.2 Interior Node Recognition

Wie aus dem vorigen Abschnitt zu ersehen ist, existieren bestimmte Positionen, für die `evaluate()` unter der Annahme beiderseitigen perfekten Spiels eine sichere Aussage über Gewinn, Verlust oder Unentschieden machen kann: Diejenigen, in welchen sich die Lage der spielentscheidenden Drohungen auf dem Weg zum jeweiligen Spielende nicht mehr verändern wird. Zwei Überlegungen liegen nahe: Zum einen wäre es wünschenswert, solche Positionen erkennen zu können, um ihnen andere numerische Werte zuzuweisen als jene bloßer Schätzungen. Eine Stellung, deren spieltheoretischer Wert als „gewonnen“ bekannt ist, ist sicher höher zu bewerten als eine Stellung, für die eine Heuristik lediglich eine mehr oder weniger hohe Gewinnwahrscheinlichkeit nahelegt – niedriger aber als eine Stellung mit einem bereits realisierten Gewinn. Zum anderen ließe sich eine sehr große Anzahl Knoten einsparen, wenn man das Auftreten solcher Positionen nicht erst an den Blättern des Suchbaums, sondern bereits früher erkennen würde - sobald nämlich die Lage besagter spielentscheidender Drohungen feststeht, nicht aber unbedingt die jeder einzelnen Scheibe.

Hier greift die Grundidee der *Interior Node Recognition*. Sie besagt: „Brich die Suche unverzüglich ab und verlasse den gesamten Teilbaum, wenn du eine Position mit einem bekannten spieltheoretischen Wert erreichst (Unentschieden, Verlust oder Gewinn).“ [5] Das Hauptproblem bei der Umsetzung dieses zunächst trivial klingenden Konzepts besteht nun darin, einen Kompromiß zwischen der Erkennung möglichst vieler spieltheoretisch entscheidbarer Stellungen einerseits und der Effizienz dieser Erkennung andererseits zu finden, die ja in jedem Suchknoten, d.h. weit über hunderttausend Mal pro Sekunde stattfinden muß.

Der in MILTON verwendete Recognizer (Listing 6.1) setzt innerhalb des Alpha-Beta-Algorithmus an der Stelle zwischen der Suche in den Hash Tables, der Suche in der Eröffnungsdatenbank und der Erkennung von gewonnenen und unentschiedenen Positionen einerseits sowie der Entscheidung über Evaluierung oder Suchvertiefung andererseits an. Er ermittelt zuerst die Anzahl noch freier, also nicht vollständig gefüllter Spalten; gibt es derer nur noch eine, so werden sich keine neuen Drohungen mehr ergeben und der Suchbaum kann an dieser Position abgeschnitten werden. Ebenso verhält es sich, wenn es noch zwei freie Spalten gibt, diese jedoch durch mindestens drei Spalten voneinander getrennt sind, so daß neue Scheiben in der einen keine neuen Drohungen in der anderen bewirken können. Alle anderen Stellungen werden wie gewöhnlich behandelt; für diese schnell zu überprüfende Teilmenge entscheidbarer Positionen erfolgt jedoch unmittelbar ein Aufruf von `evaluate()` mit dem Argument `recognized`.

Im vorigen Abschnitt wurde erwähnt, daß die Evaluationsfunktion aufgrund ihres konservativen Vorgehens im allgemeinen keine genaue Endposition bestimmen kann. Tatsächlich gibt sie aber mehr Information zurück als „Gewinn“, „Verlust“ oder „Niederlage“: Man erhält für voraussichtlich verlorene Stellungen den exakten Abstand zum Spielende in Halbzügen sowie für voraussichtlich gewonnene Stellungen eine obere Schranke für diesen Abstand. In den recognizer scores werden diese Werte verwendet – ein nähe-

Verluste	Recog.-Verluste	Eval.-Ergebnisse	Recog.-Gewinne	Gewinne
-G . . . -G+X	-R . . . -R+Y	-E . . . 0 . . . +E	+R-Y . . . +R	+G-X . . . +G

mit

$$\text{-G} \leq \text{-G+X} < \text{-R} < \text{-R+Y} < \text{-E} < 0 < \text{+E} < \text{+R-Y} < \text{+R} < \text{+G-X} \leq \text{+G}$$

Abbildung 6.2: Slates Schranken für Bewertungen innerer Knoten

rer Gewinn bzw. ein fernerer Verlust gelten als wertvoller.[3] `evaluate()` orientiert sich hierbei an Slates Konzept für Interior-Node Score Bounds [8] (siehe Abb. 6.2).

Durch den Einsatz der Interior Node Recognition läßt sich die Suchbaumgröße um so stärker verringern, je näher man dem Spielende kommt; in der Eröffnungsphase hat sie in der Regel keine Wirkung. Für iterierte Suchen bis zur gleichen Tiefe ergaben einige Stichproben: ≈ -1 – 0,5% Verringerung im vierzehnten Halbzug, ≈ -1 – 5% Verringerung im sechzehnten Halbzug, ≈ -3 – 18% Verringerung im achtzehnten Halbzug, bis zu 50% im zwanzigsten. Im self-play-Test zeigte sich die Version mit Recognizer der Version ohne Recognizer mit 37 : 34 überlegen.

[3] Für einen erwarteten Gewinn wird $699 -$ *Abstand in Halbzügen* zurückgegeben, für einen erwarteten Verlust $-699 +$ *Abstand in Halbzügen*.

```
     .
     .
     .

//   Eröffnungsbuchzugriff

          int a = -1;
          int b = -1;
          int freeColumns = 0;
          for(int i=0;i<7;i++) {
               if(p.height[i]!=6) {
                    freeColumns += 1;
                    if(freeColumns==3)
                         break;
                    if(freeColumns==1)
                         a = i;
                    if(freeColumns==2)
                         b = i;
               }
          }
          if(freeColumns==1 || (freeColumns==2 && (b-a>=4))) {
               if(acuteThree(p,p.toMove)==-1 && acuteThree(p,!p.toMove)==-1) {
                    pv.length = 0;
                    value = evaluate(p, true);
                    addHashEntries(Zkey, index, depth+extension, value, HASHEXACT, -1);
                    return value;
               }
          }

//   Evaluierung

     .
     .
     .
```

Listing 6.1: Interior Node Recognizer in MILTONs Alpha-Beta

7 Ausblick

> Auf jedes Ende folgt wieder ein Anfang,
> auf jedes Äußerste folgt eine Wiederkehr.
> *Lü Bu We*

Bei der Entwicklung von MILTON habe ich fast täglich neue theoretische oder praktische Einsichten gewonnen und kleinere oder größere Ideen für Verbesserungen entwickelt. Einige davon konnte ich verwirklichen, andere bisher nicht; manche der umgesetzten Ideen haben sich wiederum ganz anders ausgewirkt als erwartet; das Projekt ist insofern, auch nach Fertigstellung der daraus entstandenen Bachelorarbeit, weit von seinem Abschluß entfernt.

Ein Beispiel von vielen für eine kleinere, eher technische Verbesserung wäre der Verzicht auf die Zuggenerierung, wenn aus der Hash Table bereits ein potentiell bester Zug entnommen werden konnte. Es dürfte ein wenig Rechenzeit sparen, diesen zunächst auszuprobieren und alle anderen Züge nur zu bestimmen, wenn er keinen fail-high erreicht.

Deutlich mehr Gewinn ermöglichte evtl. eine weitere Optimierung der Zugsortierung, z.B. durch das Bevorzugen von Zügen, die Dreierketten bilden und somit Drohungen aufbauen. Es wäre jedoch zu prüfen, ob hierdurch genug Knoten eingespart werden können, um den Zeitaufwand auszugleichen, den die Bestimmung der Dreierketten verursacht.

Auch die Möglichkeit, durch automatisierten Vergleich der in einigen tausend Positionen jeweils möglichen Züge und des jeweils besten Zuges auf bisher unberücksichtigte Muster zu stoßen, kann nicht ausgeschlossen werden.

Ein Beispiel für eine algorithmische Verbesserung, die aus der Theorie des zugrundeliegenden Spiels erwächst, wäre der Ausbau der Evaluationsfunktion. Es wäre z.B. in Endspielstellungen denkbar, für den aktuell zurückliegenden Spieler nicht nur die bestehenden Drohungen zu bestimmen, sondern auch die *potentiellen*, d.h. alle diejenigen, die noch nicht durch Scheiben des Gegners vereitelt sind. Zeigt sich nun, daß der Spieler seine Situation selbst durch das Realisieren all dieser Drohungen nicht verbessern könnte – wenn er also das Brett ohne einen weiteren Zug des Gegners mit Scheiben füllen dürfte oder aber unter einer anderen, noch zu bestimmenden optimistischen Grundannahme – so steht sein Verlust fest und kann entsprechend im Wert der Position reflektiert werden. Auch zwischen den zahlreichen als unentschieden klassifizierten Positionen könnte differenziert werden, indem festgestellt wird, ob für einen der Spieler evtl. keinerlei Gewinnmöglichkeiten mehr bestehen. In der Verbindung solcher Ansätze mit der Internal Node Recognition (Stichwort: Rückgabe von Schranken statt exakter Scores) – so schwierig sich diese aus Effizienzgründen gestalten dürfte – sehe ich großes Potential.

Weiterhin gibt es Konzepte aus der Schachprogrammierung, deren Anwendbarkeit auf Vier Gewinnt noch völlig unüberprüft ist: Beispielsweise das *Internal Iterative Deepening* (ein Verfahren, das Suchen mit geringer Suchtiefe einsetzt, um die Zugsortierung in

inneren Knoten des Spielbaums zu verbessern) oder das *Futility Pruning* (eine Methode, den Suchbaum einen Halbzug oder sogar zwei Halbzüge vor dem Horizont abzuschneiden, wenn absehbar ist, daß keine entscheidende Veränderung an der Bewertung mehr auftreten wird; siehe [5]), bis hin zu *MTD(f)*, laut seinem Entwickler Aske Plaat ein Alpha-Beta und PVS überlegenes Suchverfahren.

In vielerlei Hinsicht besteht also noch Raum für eine schnellere, tiefere und klügere Suche.

Literaturverzeichnis

Bücher und Artikel

[1] James D. Allen (1989): A Note on the Computer Solution of Connect-Four. *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad*, hrsg. von D. N. L. Levy und D. F. Beal, Ellis Horwood, Chichester; 134-135.

[2] James D. Allen (1990): Expert Play in Connect-Four.
http://homepages.cwi.nl/~tromp/c4.html

[3] L. V. Allis (1988): A Knowledge-based Approach of Connect-Four. The Game is Solved: White Wins. M.Sc. thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam.
http://www.farfarfar.com/games/connect_four/connect4.pdf

[4] D. J. Edwards und T. P. Hart (1963): The Alpha-Beta Heuristic. *M.I.T. Artificial Intelligence Memo*, No. 30 (ursprünglich herausgegeben als The Tree Prune (TP) Algorithm, 4. Dezember 1961).

[5] Ernst A. Heinz (2000): Scalable Search in Computer Chess. Vieweg Verlag, Wiesbaden.

[6] D. Knuth und R. Moore (1975): An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6; 293-326.

[7] David Levy und Monty Newborn (1991): How Computers Play Chess. Computer Science Press, New York.

[8] D. J. Slate (1984): Interior-node score bounds in a brute-force chess program. *ICCA Journal*, Vol. 7, No. 4; 184-192.

[9] D. J. Slate und L. Atkin (1977): Chess 4.5 – The Northwestern University Chess Program. *Chess Skill in Man and Machine*, hrsg. von P. Frey, Springer-Verlag, Berlin; 82-118.

[10] A. M. Turing (1953): Digital Computers Applied to Games. *Faster than Thought*, hrsg. von B. V. Bowden, Pitman, London; 286-310.

[11] Albert L. Zobrist (1970): A hashing method with applications for game playing. *Technical Report 88*, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin.

Internetseiten

[12] http://www.seanet.com/~brucemo/topics/topics.htm

[13] http://www.gamedev.net/reference/articles/article1014.asp
und die anderen Artikel der einführenden Reihe „Chess Programming“

[14] http://homepages.cwi.nl/~tromp/c4/c4.html

[15] http://www.lbremer.de/index.html