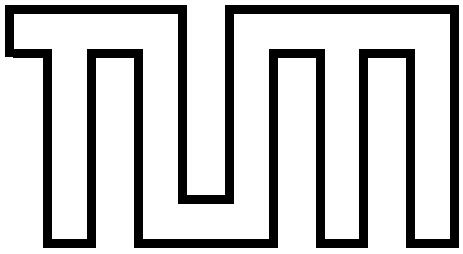


Technische Universität München

Fakultät für Informatik

Forschungs- und Lehreinheit Informatik IX

Adversariale Suche für optimales Spiel: Der Minimax-Algorithmus und die Alpha-Beta-Suche

Proseminar

Julius Adorf

Betreuer: Lars Kunze, Dominik Jain

Abgabetermin: 2. Dezember 2009

Zusammenfassung

Die Minimax-Suche ist ein klassisches Entscheidungsverfahren für den nächsten Zug in einem Zwei-Personen-Nullsummenspiel. Die Alpha-Beta-Suche macht die Minimax-Suche praxistauglich, indem sie bei der Suche Zugfolgen nicht mehr untersucht, die das Ergebnis nicht mehr beeinflussen können. Diese Arbeit ist eine Einführung in beide Algorithmen. Es werden der theoretische Hintergrund beleuchtet, Suchheuristiken, Herausforderungen bei der Implementierung und der Einsatz bei GNU Chess besprochen.

Inhaltsverzeichnis

1 Motivation

Der Minimax-Algorithmus und die Alpha-Beta-Suche sind klassische Mittel, um Computergegner für sogenannte adversariale Nullsummenspiele zu entwickeln. Beide Algorithmen bauen auf Ideen auf, die z.B. Schachspielern schon bekannt sind. Um ihren nächsten Zug zu bestimmen, versetzen sich Schachspieler gelegentlich in die Lage des Gegners, um zu bestimmen, welche Antwort dieser auf einen eigenen Zug wählen würde. Nach einem ähnlichen Prinzip funktioniert auch der Minimax-Algorithmus. Nun gibt es aber im Schach sehr viele mögliche Züge, so dass man im Spiel nicht sehr weit vorausrechnen kann, selbst mit einem Rechner nicht. Findet man als Schachspieler jedoch eine Zugfolge, die zu einer guten Stellung führt, so kann man sich getrost die Betrachtung von einigen anderen Zugfolgen sparen. Hier liegt die Idee der Alpha-Beta-Suche.

2 Spieltheoretischer Hintergrund

Der Minimax-Algorithmus findet die optimale Antwort auf jede Stellung bei optimalem Spiel beider Spieler. Was überhaupt optimal ist, muss man zuvor allerdings definieren. Die Beleuchtung der Spieltheorie gibt auch Aufschluss über Fragen, unter welchen Bedingungen sich der Minimax-Algorithmus einsetzen lässt, und zeigt, welche Zusammenhänge zwischen dem Gewinn des einen und dem Verlust des anderen Spielers bestehen.

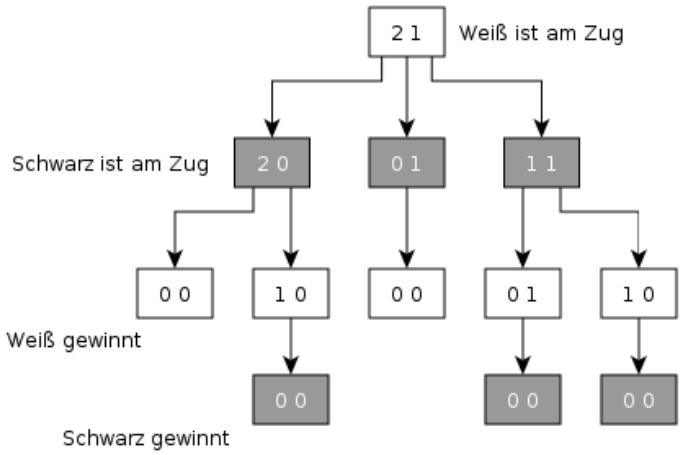


Abbildung 1: ein Spielbaum des Nim-Spiels mit zwei Stapeln.

2.1 Spielbäume

In der Spieltheorie ist ein *Spielbaum* ist ein azyklischer gerichteter Graph, dessen Knoten *Stellungen* des Spiels darstellen. Dabei kann die gleiche Stellung an verschiedenen Knoten

des Spielbaums auftreten. Jeder Knoten hat zudem die Eigenschaft, dass ein bestimmter Spieler an der Reihe ist. Dieser Spieler kann sich für eine der möglichen Aktionen entscheiden, die durch Kanten modelliert werden. Jede Spielstellung ist entweder eine *Endstellung* oder hat eine Menge von *Nachfolgestellungen*, die in einem *Halbzug* des Spiels erreicht werden können.

Das Nim-Spiel kann mit einem Spielbaum modelliert werden. Im Nim-Spiel ziehen zwei Spieler abwechselnd Streichhölzer von Stapeln. In einem Halbzug zieht ein Spieler ein oder mehrere Streichhölzer von einem der Stapel. Derjenige, der das letzte Streichholz zieht, verliert. Diese Verlustbedingung ist beim Nim-Spiel äquivalent zu der Bedingung, dass ein Spieler gewinnt, wenn er am Zug ist und nur leere Stapel vor sich liegen hat.

Ein mathematisches Modell ist nutzlos, wenn man mit ihm keine Fragen über das von ihm dargestellte System beantworten kann. Dazu gehören Fragen aus der Sicht eines Spielers wie.:

- Welche Stellungen sind Gewinnstellungen für mich?
- Welche Strategie ist für mich am besten?
- Stört es mich, wenn der Gegner meine Strategie kennt?
- Muss ich mich auf meinen Gegner einstellen?

2.2 Gewinnstellungen

Mit dem Modell des Spielbaums können *Gewinnstellungen* erkannt werden. Im Nim-Spiel in Abbildung 1 mit der Ausgangsposition $(2, 1)$ kann man feststellen, ob Weiß einen Gewinn erzwingen kann und sich also in einer Gewinnstellung befindet. Eine Gewinnstellung liegt für Weiß genau dann vor, wenn er entweder in der aktuellen Stellung nach Spielregel gewonnen hat oder einer seiner Züge zu einer Verluststellung von Schwarz führt. Eine Verluststellung für Schwarz wiederum liegt genau dann vor, wenn seine Stellung nach Spielregel verloren ist oder wenn alle seine Züge zu einer Gewinnstellung für Weiß führen. Da das Nim-Spiel kein Unentschieden kennt, ist automatisch jede Stellung entweder eine Gewinnstellung für Weiß oder für Schwarz.

2.3 Nutzwerte

Es gibt andere Spiele, bei denen bei Ende des Spiels nicht nur zwischen Gewinn, Unentschieden und Niederlage unterschieden wird. Stattdessen zieht ein jeder Spieler einen *Nutzwert* aus einer Endstellung. Hier kann man immer noch von Gewinnstellungen sprechen, als Spieler wird es das Ziel sein, den größten Nutzwert für sich selbst zu erzielen. Man beachte, dass die Definition der Nutzwerte noch nichts über eine Abhängigkeit der Nutzwerte zweier Spieler in einer Endstellung aussagt.

2.4 Nullsummenspiel

Das Nim-Spiel ist ein typisches Nullsummenspiel. In einem Nullsummenspiel ist der Gewinn eines Spielers mit dem Verlust des Gegners äquivalent. Genauer: ein Spiel ist genau dann ein Nullsummenspiel, wenn für jede Endstellung der Nutzwert für den einen Spieler mit dem negativen Nutzwert für den anderen Spieler übereinstimmt. Die Existenz eines Unentschiedens in einem Spiel ändert also an der Nullsummeneigenschaft nichts.

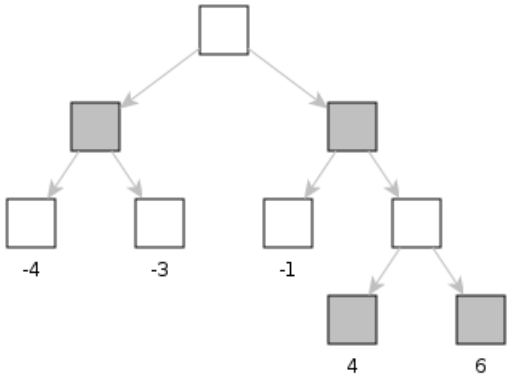


Abbildung 2: ein Baum eines Nullsummenspiels mit Minimax-Werten der Endstellungen - die Nutzwerte aus Sicht des weißen Spielers.

In Abbildung 2 ist der Spielbaum eines imaginären Nullsummenspiels dargestellt. Jeder Endstellung wird aus Konvention der Nutzwert für den weißen Spieler zugewiesen. Um den Nutzwert aus Sicht des schwarzen Spielers zu erhalten, muss man lediglich das Vorzeichen ändern.

2.5 Optimale Strategien

In der Spieltheorie ist eine *Strategie* eine vollständige Spezifikation wie sich ein Spieler in allen möglichen Spielstellungen verhält. In einem Spiel, in dem der Zufall keine Rolle spielt, ist mit der Angabe der Strategien beider Spieler der Ausgang des Spiels eindeutig bestimmt. Eine Strategie eines Spielers ist entweder eine *reine Strategie*, wenn die Strategie für jede Spielstellung, in der er an der Reihe ist, eine bestimmte Antwort festlegt, oder eine *gemischte Strategie*, wenn die Strategie nur eine Wahrscheinlichkeitsverteilung für die Züge in jeder Stellung angibt. Letztere sind z.B. bei der Betrachtung von Spielen wie Münzwurf oder Stein-Schere-Papier wichtig.

Die Abbildungen 3 und 4 zeigen je eine mögliche Strategie für Weiß und für Schwarz. Dabei muss man beachten, dass eine Strategie ein zeitloser Plan ist, der vor dem Spielablauf schon feststeht. Die Strategie beschreibt das Verhalten des Spielers komplett, und unabhängig vom Spielverlauf. Eine Strategie sieht für jede mögliche Stellung einen Zug vor.

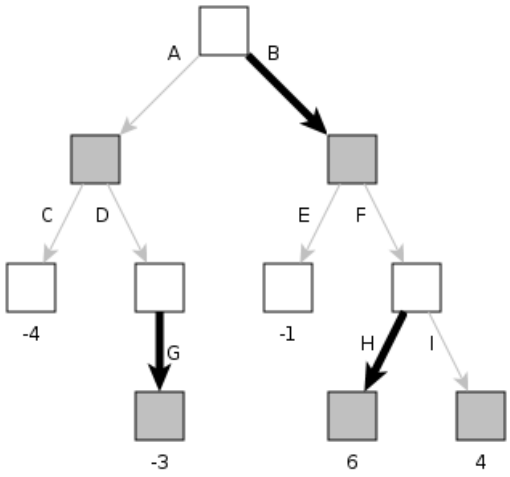


Abbildung 3: (B,G,H) - eine mögliche Strategie von Weiß

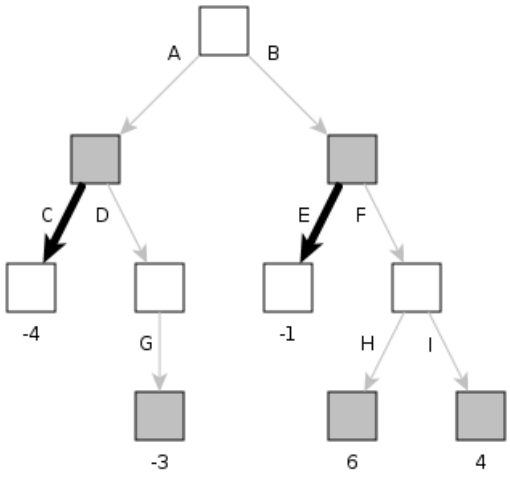


Abbildung 4: (C,E) - eine mögliche Strategie von Schwarz

Die Kombination der Strategien von Weiß und Schwarz determiniert den Ausgang des Spieles. Im Beispiel führt das Spiel zu einer Verluststellung für Weiß mit einem Nutzwert von -1 und aufgrund der Nullsummeneigenschaft zu einer Gewinnstellung für Schwarz mit einem Nutzwert von 1.

Um verschiedene Strategien zu vergleichen, kann man diese für beide Spieler aufzählen und in einer Matrix die Nutzwerte aus Sicht des weißen Spielers (MAX) eintragen. Die Strategien von Weiß sind in den Zeilen gegen die Strategien von Schwarz in den Spalten aufgetragen.

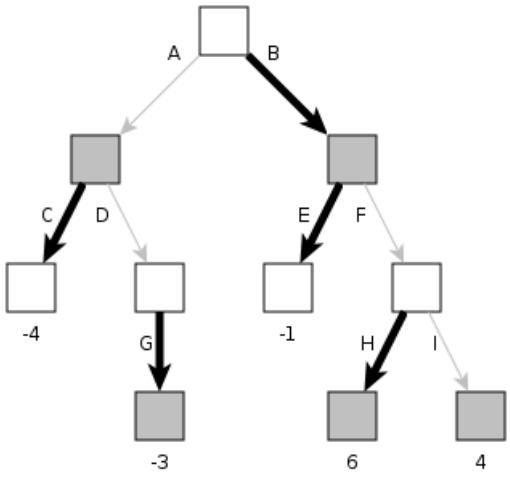


Abbildung 5: Strategien von Weiß und von Schwarz kombiniert.

	(C,E)	(C,F)	(D,E)	(D,F)
(A,G,H)	-4	-4	-3	-3
(A,G,I)	-4	-4	-3	-3
(B,G,H)	-1	6	-1	6
(B,G,I)	-1	4	-1	4

Auf welche Strategien sollten sich die Spieler festlegen? Die Entscheidung als Weißer wäre einfach, wenn man die Strategie des Gegners kennen würde. Dann könnte man sich eine Strategie aussuchen, die mit der Strategie von Schwarz kombiniert den größten Nutzwert für Weiß liefert. Die Strategie bleibt jedoch in der Regel im Verborgenen und man braucht ein anderes Auswahlkriterium. Man könnte spekulativ darauf bauen, dass der Gegner Fehler macht. Möchte man allerdings auf Nummer sicher gehen, so möchte man eine Strategie finden, die unabhängig von der Strategie des Gegners ist.

Eine sichere Strategie kann man sich aus dem Spielbaum herleiten. In Abbildung 5 z.B. würde Weiß (MAX) stets Zug H spielen und nicht Zug I. Schwarz (MIN) wird den Zug E spielen und nicht F, da F Weiß die Chance gibt, ein Spielende mit Wert 6 herbeizuführen, Zug E jedoch zu einer Gewinnstellung führt. Verfolgt man diese Logik bis zur Wurzel des Spielbaums, so erhält man für Weiß eine sichere Strategie und den *Minimax-Wert* des Spiels.

Jedem Spieler ist klar, was er erreichen möchte: mit einer *Maxmin*-Strategie seinen mindestens erreichbaren Gewinn maximieren oder mit einer *Minmax*-Strategie den maximal erreichbaren Gewinn des Gegners minimieren [6]. In Nullsummenspielen fallen allerdings beide Strategien zusammen, wie das Minimax-Theorem von John v. Neumann im Jahr 1928 aussagt [10].

Im Beispiel sichert die Strategie (B,G,H) Weiß einen Minimax-Wert von -1, was einer glimpflichen Niederlage entspricht. Schwarz trifft mit (C,E) die beste Wahl. Die Strategien

(B,G,H) und (C,E) bilden ein sogenanntes *Nash-Equilibrium*, ein Gleichgewicht, in dem keiner der Spieler einen Vorteil erreichen könnte, wenn er sich einseitig für eine andere Strategie entschieden hätte.

Der Minimax-Algorithmus findet für beide Spieler Strategien, die zusammen ein Nash-Equilibrium bilden. Zu den schönen Eigenschaften der gefundenen Strategie gehört, dass sie dem Spieler einen minimalen Nutzwert garantiert, egal welche Strategie der Gegner gewählt hat. Spielt der Gegner suboptimal, wählt er also eine andere Strategie als durch die Minimax-Suche vorhergesagt, so kann unser erwarteter minimaler Nutzwert dadurch niemals schlechter werden.

3 Minimax-Algorithmus

Das unmittelbare Ziel des Minimax-Algorithmus ist es, den Minimax-Wert für die aktuelle Stellung zu berechnen. Auch hier gilt, dass für den weißen Spieler (MAX) ein positiver Minimax-Wert Gewinn bedeutet. Für alle Endstellungen ist der Minimax-Wert gleich dem Nutzwert für MAX. Für alle anderen Stellungen wird der Minimax-Wert rekursiv aus den Endstellungen berechnet.

$$minmax(r) = \begin{cases} utility(r), & \text{falls } r \text{ Endzustand} \\ max\{minmax(s) \mid s \in \Gamma(r)\}, & \text{falls } r \text{ MAX-Knoten} \\ min\{minmax(s) \mid s \in \Gamma(r)\}, & \text{falls } r \text{ MIN-Knoten} \end{cases}$$

Die Funktion Γ liefert alle Nachfolgestellungen einer Stellung im Baum. Die Funktion, die entscheidet ob ein Stellung ein Spielende darstellt, und die Unterscheidung zwischen MIN- und MAX-Knoten ist in der mathematischen Darstellung implizit. Eine Implementierung gibt in der Regel zusätzlich zum Minimax-Wert den Zug zurück, der zu der Stellung mit diesem Wert geführt hat.

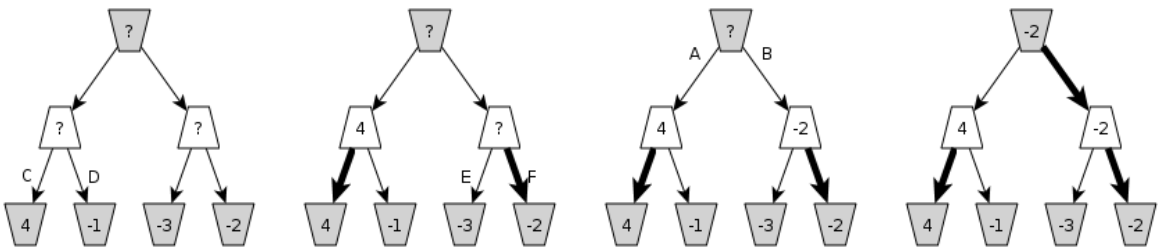


Abbildung 6: Die Schritte des Minimax-Algorithmus in einem Spielbaum.

Der Minimax-Algorithmus ist eigentlich nur eine Tiefensuche. Betrachtet man zum Beispiel die beiden Züge C und D in Abbildung 6 aus Sicht von Weiß. Dieser bevorzugt den maximalen Spielwert, den er erzielen kann, also 4. Genauso verfährt Weiß mit den

beiden anderen Knoten und bevorzugt hier F. Schwarz kann nun aus den beiden weißen MAX-Knoten schlussfolgern, dass er sich in der Ausgangsstellung für den Zug B entscheiden muss, da dieser ihm einen wesentlich günstigeren Minimax-Wert von -2 für die Ausgangsstellung garantiert.

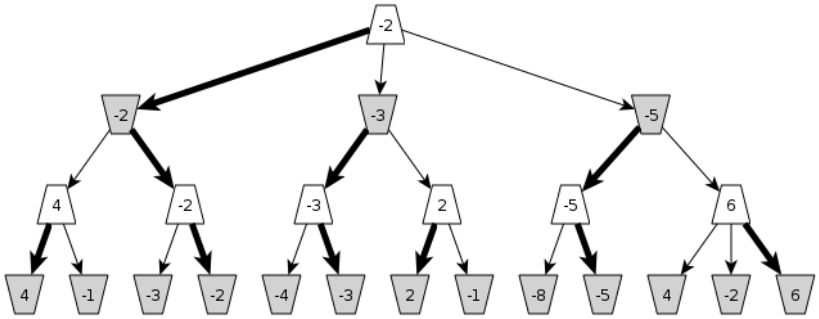


Abbildung 7: Ein vom Minimax-Algorithmus durchsuchter Spielbaum. Die berechneten Züge geben für Weiß und Schwarz Strategien eines Nash-Equilibriums an.

3.1 Negamax-Implementierung

Im Minimax-Algorithmus versuchen beide Spieler in ihrer Stellung den Zug so auszuwählen, dass der Minimax-Wert des Knotens maximiert bzw. minimiert wird. Die dafür notwendige Fallunterscheidung führt zu Duplikation in der Implementierung, die mit der Negamax-Variante des Algorithmus vermieden werden kann. In der Negamax-Variante wählen die Spieler stets denjenigen Zug, der dem Gegner den meisten Schaden zufügt. Hier kommt die Voraussetzung der Nullsummeneigenschaft ins Spiel, die für die Äquivalenz der Gewinnmaximierung und Schadensmaximierung sorgt. Für einen MAX-Knoten sind Ausgabe von Minimax und Negamax gleich.

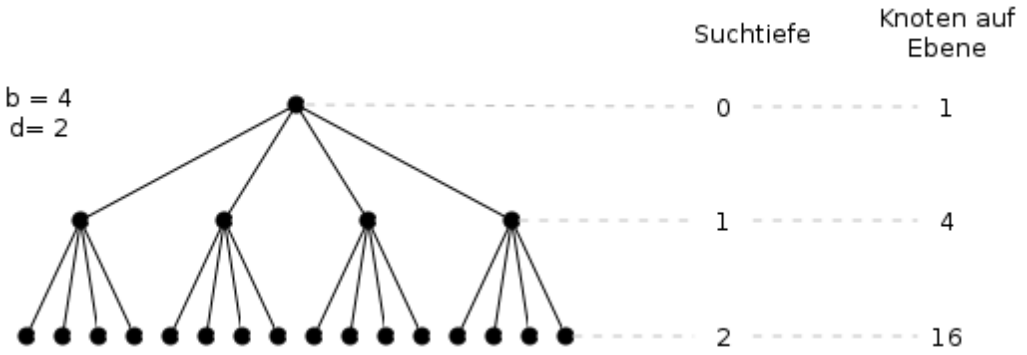
$$\text{mit Weiß: } p = 1 \text{ bzw. Schwarz: } p = -1$$

$$negamax(r, p) = \begin{cases} p * utility(r), & \text{falls } r \text{ ein Endzustand ist} \\ max\{-negamax(s, -p) \mid s \in \Gamma(r)\}, & \text{sonst} \end{cases}$$

3.2 Kombinatorische Explosion

So elegant und einfach die Minimax-Suche ist, so wenig praxistauglich ist sie leider auch in ihrer reinen Form. Die Suchbäume sind in der Regel viel zu groß, als dass man sie komplett durchsuchen könnte. Um die Größe eines Spielbaums zu beschreiben, wird ein Spielbaum mit seinem Verzweigungsgrad und der Tiefe des Baumes parametrisiert.

Gegeben sei ein gleichförmiger Spielbaum $T = (V, E)$. Ein gleichförmiger Spielbaum besitzt für jeden inneren Knoten den gleichen Verzweigungsgrad b und alle Blätter befinden sich in der gleichen Tiefe d. In einem solchen Baum wächst die Zahl der Knoten exponentiell mit der Tiefe des Baumes.



Der Minimax-Algorithmus ist eine Tiefensuche und sucht den Baum $T = (V, E)$ in der Zeit $O(\max\{|V|, |E|\}) = O(|V|)$ ab. Die Suche benötigt zwar lineare Zeit, jedoch wächst der Spielbaum exponentiell mit zunehmender Tiefe, woraus mit $b > 1$ eine eine Komplexität des Minimax-Algorithmus von $O(b^d)$ folgt. Begründung: Die Zahl der Knoten berechnet sich über die Summe der Anzahl der Knoten einer jeden Ebene:

$$|V| = \sum_{k=0}^{d} b^k \tag{1}$$

Die größte Potenz von b in der Summe ist für $b > 1$ stets größer als die Summe aller niedrigeren Potenzen:

$$\sum_{k=0}^{d-1} b^k < b^d \quad (\text{mit } b > 1) \tag{2}$$

Mit der Definition aus [9]:

$$|V| \in O(b^d) \quad \Leftrightarrow \quad \exists c, d_0 \in N_0 : \forall d > d_0 : 0 \leq |V| \leq cb^d \tag{3}$$

Mit (1), (2), $d_0 = 0$ und $c = 2$ in (3) eingesetzt, zeigt sich, dass der Minimax-Algorithmus zumindest für gleichförmige Spielbäume eine Komplexität von $O(|V|) = O(b^d)$ besitzt. Dieses Wachstum macht die direkte Anwendung in Spielen sehr schwierig.

Schon bei einem kleinen Spiel wie Tic-Tac-Toe besitzt der Spielbaum eine Tiefe von 9 und einen anfänglichen Verzweigungsgrad von 9, der mit wachsender Tiefe allerdings abnimmt.

Es ist klar, dass man für dieses banale Spiel keinen Minimax-Algorithmus zur Entscheidung braucht, da einfache Faustregeln zu gutem Spiel genügen. Trotzdem ist Tic-Tac-Toe ein illustratives Beispiel. Eine obere Schranke für die Zahl der Knoten des Spielbaums erhält man mit

$$\sum_{k=0}^{9} \frac{9!}{(9-k)!} = 986410$$

Die untere Schranke lässt sich mathematisch leider nicht so einfach bestimmen. Eine empirische Ermittlung ergab einen Spielbaum von immerhin 549946 Knoten für ein triviales Spiel auf einem 3x3-Brett! Beim Tic-Tac-Toe bewegt sich diese Größe noch im Rahmen der Möglichkeiten für die vollständige Berechnung, aber bei kaum einem anderen Spiel ist die Tiefe und der Verzweigungsgrad des Spielbaums derart begrenzt. Schach zum Beispiel besitzt einen Verzweigungsgrad von etwa 35 [8]. Deshalb ist der Minimax-Algorithmus allein für sich genommen nicht praxistauglich. Für Abhilfe sorgen die Alpha-Beta-Suche und eine Reihe weiterer Techniken.

4 Alpha-Beta-Suche

Die Alpha-Beta-Suche ist eine Methode, in der bestimmte Teile des Spielbaums gar nicht erst durchsucht werden, ohne dabei die Korrektheit des Ergebnisses zu gefährden [4]. Hinter der Alpha-Beta-Suche steckt die Idee, dass man sich manche Varianten nicht mehr ansehen muss, weil sie klar schlechter sind als bisher gefundene.

Die Alpha-Beta-Suche funktioniert wie der Minimax-Algorithmus, nur dass mit jeder Stellung ein veränderliches $\alpha\beta$-Fenster assoziiert ist, das mit einem Intervall $[\alpha;\beta]$ notiert wird. Man betrachte das Minimalbeispiel: Sei S ein MAX-Knoten, so beschreibt α den bisher größten gefundenen Minimax-Wert einer Nachfolgestellung von S, und β den bisher kleinsten Minimax-Wert einer der linken Geschwisterknoten von S. Nun beschränkt dieses Fenster die zu untersuchenden Züge von S. Denn überschreitet der Minimax-Wert einer Nachfolgestellung von S die β-Schranke, so heißt dies nichts anderes, als dass sich MIN im Vorgängerknoten von S nicht für den Zug nach S entscheiden würde, und eine weitere Untersuchung von Nachfolgestellungen sich als zwecklos herausstellt. Die β-Schranke liefert demnach in einem MAX-Knoten die Information, ab wann sich MIN wegen einer besseren Alternative nicht mehr für diesen MAX-Knoten entscheiden würde. Die Argumentation lässt sich analog auch auf einen MIN-Knoten übertragen und vor allem auch rekursiv anwenden.

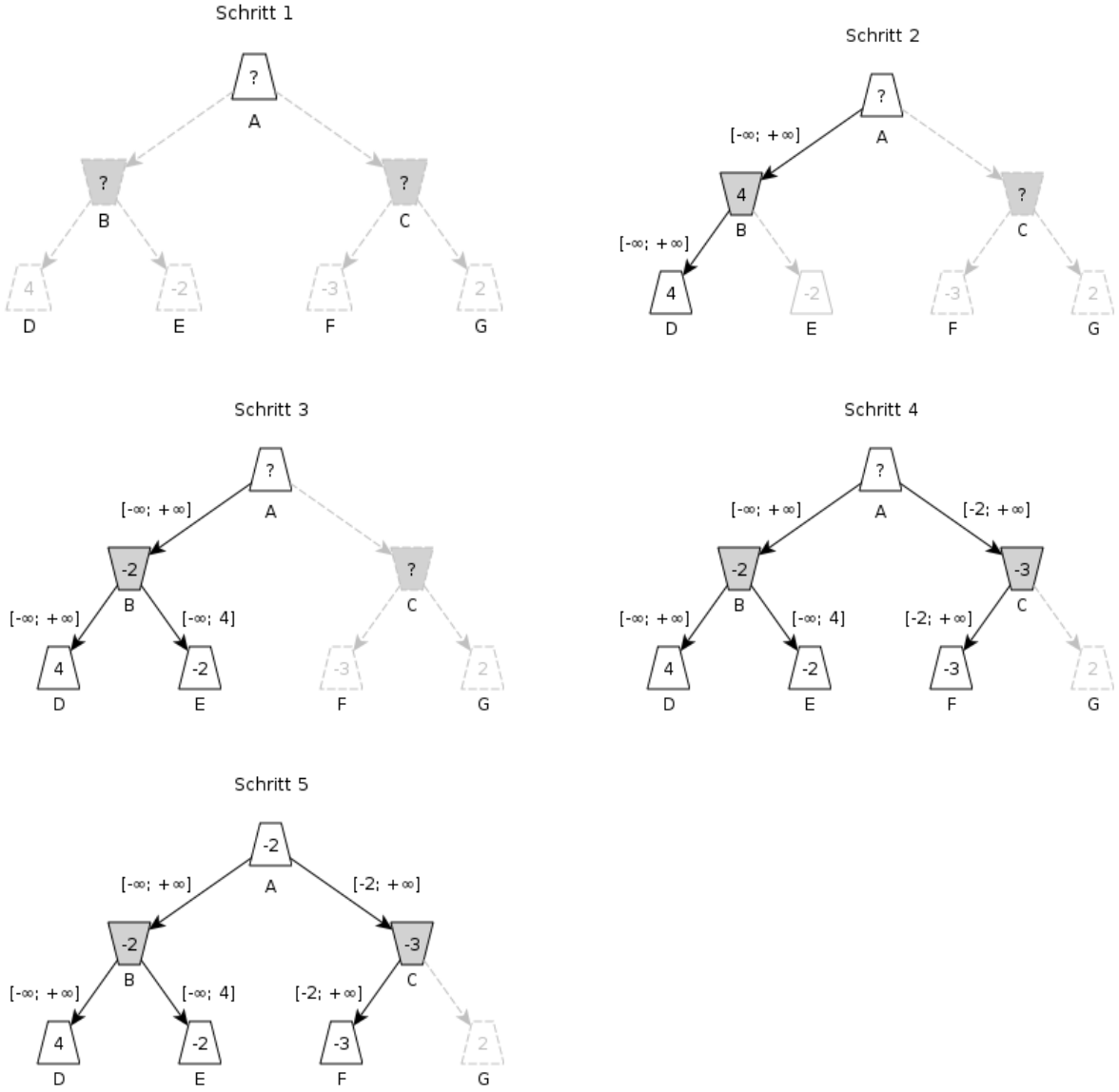


Abbildung 8: Schrittfolge einer Alpha-Beta-Suche. Erläuterung der Illustration: (1) Beginn bei der Ausgangsstellung A, es ist noch nichts über beste Züge bekannt. (2) Schwarz hat den Zug zu D gefunden, der ihm einen für B einen Minimax-Wert von 4 liefert. (3) Schwarz hat einen besseren Zug in B gefunden, der ihm für B einen Wert von -2 oder kleiner garantiert. (4) Schwarz entdeckt einen Zug in C mit Wert -3. Da Weiß in B bereits einen Zug gefunden hat, der mit -2 einen günstigeren Wert bietet als -3, braucht Stellung G nicht mehr untersucht werden. Denn hätte Stellung G einen Wert kleiner als -3, würde Weiß dennoch den Zug nach B bevorzugen. Hätte Stellung G einen Wert größer als -3, so würde Schwarz den Zug nach F wählen, und Weiß wäre ebenfalls besser beraten mit dem Zug nach B zu beginnen. (5) Die Suche ist zu Ende. In diesem Beispiel hat die Alpha-Beta-Suche einen Knoten weniger besucht. Wäre G ein Teilbaum und kein Endknoten, so wäre an dieser Stelle ein kompletter Teilbaum abgesägt worden.

Der Algorithmus ist ähnlich aufgebaut wie der Minimax-Algorithmus, lediglich über die α, β-Werte muss Buch geführt werden. Es folgt die Implementierung einer Alpha-Beta-Suche [3] in Python, das sich hier mindestens so gut wie Pseudo-Code lesen lässt:

```
def alphabeta_search(node):
    return max_value(node, -sys.maxint - 1, sys.maxint)

def max_value(node, alpha, beta):
    if is_leaf(node):
        return value(node)
    else:
        v = alpha
        for c in children(node):
            v = max(v, min_value(c, v, beta))
            if v >= beta:
                return v
        return v

def min_value(node, alpha, beta):
    if is_leaf(node):
        return value(node)
    else:
        v = beta
        for c in children(node):
            v = min(v, max_value(c, alpha, v))
            if v <= alpha:
                return v
        return v
```

Die hier nicht definierten Funktionen und Datenstrukturen sind im Anhang A aufgelistet. Für das Verständnis der Funktionsweise ist es wichtig, ein Gefühl für die Bedeutung des $\alpha\beta$-Fensters zu bekommen. In Abbildung 9 wird im MIN-Knoten C mit $\alpha\beta$-Fenster $[-2; \infty]$ die Untersuchung der Nachfolgestellungen abgebrochen, da sich herausstellt, dass MIN mit einem Zug von C nach D einen Wert von -3 außerhalb des Fensters erzielen kann, der so gut ist, dass sich MAX nicht für einen Zug nach C entscheiden würde. Die Kraft dieses Mechanismus steckt darin, dass diese Logik rekursiv angewandt wird.

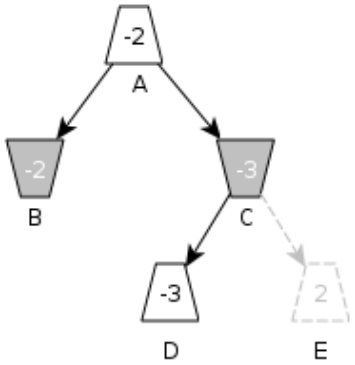


Abbildung 9: Ein Minimalbeispiel für einen durch die Alpha-Beta-Suche abgesägten Teilbaum.

4.1 Negamax-Implementierung

Im Alpha-Beta-Suchalgorithmus ist die Duplikation im Quelltext deutlich zu sehen. Für Anschauungszwecke erscheint die explizit zwischen MAX- und MIN-Knoten unterscheidende Variante aber sinnvoller als die Negamax-Variante. In einer tatsächlichen Anwendung wiederum ist die Negamax-Variante sicherlich zu bevorzugen. Es folgt die Negamax-Implementierung [3] der Alpha-Beta-Suche in Python. Der Wert `player = 1` steht für MAX, `player = -1` für MIN.

```
def alphabeta_negamax_search(node, player):
    return alphabeta_negamax(node, player, -sys.maxint - 1, sys.maxint)

def alphabeta_negamax(node, player, alpha, beta):
    if is_leaf(node):
        return player * value(node)
    else:
        v = alpha
        for c in children(node):
            v = max(v, -alphabeta_negamax(c, -player, -beta, -v))
            if v >= beta:
                return v
        return v
```

Der Algorithmus ist erstaunlich schlank. Die Herausforderungen liegen in der Wahl der Datenstrukturen und der Nachfolgerfunktion, die die möglichen Züge generiert. Werden die Nachfolgestellungen in zufälliger Reihenfolge ausgewählt, so betrachtet die Alpha-Beta-Suche etwa $O(b^{3d/4})$ der Knoten, eine wesentliche Verbesserung gegenüber der Minimax-Suche, die $O(b^d)$ Knoten untersucht. [8].

5 Heuristische Alpha-Beta-Suche

Die Alpha-Beta-Suche hat eine bessere asymptotische Komplexität als der Minimax-Algorithmus. Trotzdem können Spielbäume normalerweise nicht bis zu ihren Endstellungen - den Blättern - durchsucht werden. Stattdessen kann man eine bestimmte maximale Suchtiefe definieren. Das ist der *Suchhorizont*, an dem die Suche abgebrochen wird. Dafür muss man die exakte Nutzwertfunktion durch eine Evaluierungsfunktion ersetzen, die für eine gegebene Stellung am Suchhorizont einen geschätzten Nutzwert liefert. Diese Evaluierungsfunktion ist eine Heuristik, die schnell zu einem Wert gelangt und dabei die Exaktheit des Ergebnisses zu Gunsten der Zeitersparnis opfert. Die Güte und Geschwindigkeit der Auswertung einer Stellung spielt maßgeblich in das Ergebnis der Suche ein.

6 GNU Chess

GNU Chess ist eine Schach-KI aus dem Open-Source-Bereich, an deren Suchalgorithmus Techniken und Erweiterungen im Zusammenhang mit der Alpha-Beta-Suche vorgestellt werden. Viele der Erweiterungen und Heuristiken können auch auf andere Spiele als Schach übertragen werden. Im folgenden werden Techniken erst allgemein vorgestellt und dann am Beispiel von GNU Chess in einen Kontext gesetzt.

Kurz vorweg: Das Programm ist in C geschrieben und der Quelltext ist sicherlich kein Kandidat für einen Schönheitspreis. Es ist verwunderlich, dass der relativ komplexe Code anscheinend ohne Tests auskommt, da keine Tests in den Quellen zu finden sind. Ungeachtet dessen ist viel Wissen in das Projekt eingeflossen, und so kann man aus dem Projekt einige Ideen extrahieren.

6.1 Datenstruktur

Das Spielbrett ist in Bitmasken gespeichert. Dabei passt praktischerweise das 8x8-Schachfeld mit seinen 64 Feldern genau in ein vorzeichenunbehaftetes 64-bit Long. Je eine solche 64-bit-Maske speichert alle Positionen eines bestimmten Figurentyps bzw. der Bauern einer bestimmten Farbe. Alle weißen Bauern z.B. sind in einer 64-bit-Maske mit einer 1 gekennzeichnet.

Über die aktuelle Brettstellung werden mehr Informationen als unbedingt notwendig gespeichert, um die Analyse der Stellung zu vereinfachen. Dazu gehört unter anderem die Materialsumme und die Information welche Felder von eigenen Figuren besetzt sind.

6.2 Suchalgorithmus

GNU Chess verwendet *Principal Variation Search* (PVS) als Suchalgorithmus. Er ist eine Weiterentwicklung der Alpha-Beta-Suche. Bis auf feine Unterschiede kursiert PVS auch unter dem Namen Negascout [3]. PVS spekuliert darauf, dass die von ihm in den Stellungen zuerst untersuchten Züge die besten Züge sind und er damit die Hauptvariante zuerst abläuft. Bei jedem Rekursionsschritt wird für den ersten Zug eine Suche mit vollem $\alpha\beta$-Fenster gestartet, für alle verbleibenden Züge der Stellung wird eine Suche mit einem *Nullfenster* durchgeführt. Mit einem solchen Nullfenster lässt sich feststellen, ob die Vermutung, dass der erste Zug der bessere ist, korrekt ist oder nicht. Für diese Berechnung ist weniger Aufwand nötig als bei der Berechnung des Minimax-Wertes. Stellt sich die Vermutung als falsch heraus, so muss Negascout die Züge mit vollem Suchfenster neu untersuchen. Negascout vertraut damit auf eine relativ gute Ordnung der Züge in jeder untersuchten Stellung. Negascout ist wesentlich günstiger als die reine Alpha-Beta-Suche [3].

Der Suchalgorithmus arbeitet bei GNU Chess auf einer globalen Datenstruktur. Das Schachbrett wird während der Suche vor einem Zug nicht kopiert. Stattdessen wird auf dem Brett ein Zug ausgeführt und nach dem rekursiven Aufstieg der Tiefensuche wieder rückgängig gemacht. Dies ist eine Möglichkeit, die sich bei einer Tiefensuche stets anbietet.

6.3 Sortierung der Züge

Für die Effektivität des Negascout-Algorithmus ist es wichtig, dass die besseren Züge zuerst untersucht werden. Generell gibt es für die Sortierung der Züge in einer Stellung verschiedene Ansätze. Eine mögliche Idee wäre die Evaluierungsfunktion der Blätter auch auf die inneren Knoten anzuwenden, um so die Züge zu sortieren. Ein zweiter Weg wäre gespeicherte Werte aus früheren Suchen zu Rate zu ziehen.

Eine dritte Möglichkeit stellt die *History-Heuristik* dar, eine dynamische Methode, die Züge im Verlauf der Suche bewertet und die gewonnenen Informationen sofort zur Sortierung einsetzt. Die History-Heuristik speichert in einer Tabelle die Bewertungen von Zügen ab. Beim Schach kann diese Tabelle eine 64x64-Matrix sein, die Züge von einem Feld auf ein anderes Feld mit Bewertungseinheiten be-

	a1	b1	c1	...	f8	g8	h8
a1	0	2	3	...	0	0	2
b1	1	0	7	...	0	0	0
c1	2	1	0	...	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
c4	0	0	1	...	0	12	0
d4	1	0	0	...	0	0	5
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
h8	2	0	0	...	0	4	0

Tabelle 1: Tabelle der History-Heuristik beim Schach. Nicht alle Zugkombinationen sind möglich. Der Zug c4-g8 hat eine hohe Bewertung.

legt. Die Information, mit welchen Figuren Züge ausgeführt wurden, ist absichtlich nicht enthalten. Wenn man den komplette Kontext mitabspeichert, also Zug und Stellung, so wird aus der History-Heuristik die Idee der Transpositionstabelle [5].

Eine spezielle Version der History-Heuristik ist die *Killer-Heuristik*. Die Killer-Heuristik merkt sich die besten Züge (killer moves) einer Suchtiefe, die im bisherigen Suchverlauf aufgetreten sind. Sowohl History-Heuristik als auch die Killer-Heuristik basieren auf der etwas spekulativen Erwartung, dass gute Züge in anderen Stellungen ebenfalls zu guten Ergebnissen führen. Die Wahrscheinlichkeit, dass ein Zug auch in einer anderen Stellung gut ist, ist größer, wenn die Stellungen sich ähnlich sind. Deshalb gehen Züge, die auf geringer Suchtiefe gefunden worden sind, mit einem wesentlich größeren Gewicht in die Bewertung ein [5].

GNU Chess sortiert Züge nach folgenden Kriterien. Zuerst werden Züge überprüft, zu denen es bereits einen Eintrag in der Transpositionstabelle in gleicher Zugtiefe gibt. Dann werden Schlagzüge überprüft, dabei werden Schlagzüge mit Figuren niedrigen Werts bevorzugt, und anschließend folgen die Killerzüge und letztlich die Züge aus der History-Tabelle.

6.4 Stellungsbewertung

In die eingebaute Heuristik zur Stellungsbewertung fließen Erfahrungswerte aus dem Schachspiel ein, für deren Verständnis gute Schachkenntnisse notwendig sind. Wer sich beim Schach nicht so gut auskennt, mag hier zumindest mitnehmen, dass die Sicht des Menschens, wie man gut Schach spielt auf die Evaluierungsfunktion übertragen wird. Dabei ist die Evaluierungsfunktion eine Überlagerung verschiedener Aspekte, die unterschiedlich stark gewichtet werden.

Die Evaluierungsfunktion von GNU Chess fährt unterschiedliche Strategien, je nachdem ob die untersuchte Stellung der Eröffnung, dem Mittelspiel oder dem Endspiel zugeordnet ist.

In der Eröffnung bestraft die Bewertungsfunktion frühe Bewegungen der Dame und des Turms (1.h4 ... 2. Th3 ... wird dadurch schon mal unterbunden), das Vorziehen von Flügelbauern und wiederholte Züge mit Leichtfiguren oder Zentrumsbauern.

In allen Spielphasen wird der Materialunterschied berücksichtigt und für jede Figur festgestellt, welche Macht sie ausübt, ob sie schwache Bauern angreift, sich gut bewegen kann oder aufgrund einer Fesselung gar nicht bewegen kann. Der Wert einer Spielfigur wird für die fünf Figurentypen und die Bauern einzeln festgestellt.

GNU Chess bestraft Ausflüge des Königs in der Eröffnung und im Mittelspiel, wechselt im Endspiel aber dazu über, Könige im Zentrum zu belohnen, da sie im Endspiel zu den mächtigsten Figuren gehören. Könige, die auf (halb-)offenen Linien stehen, sind ein Malus. Einen Bonus gibt es für ein intaktes Bauernschild, das Einsperren der Türme in

den Ecken wird ungern gesehen. Am eigenen Königsflügel sollte das Verhältnis zwischen angreifendem Material und verteidigenden Figuren nicht zu groß sein.

Die Dame gehört ins Zentrum und sollte auf keinen Fall in eine Fesselung geraten. Ein oder zwei Türme oder gar ein Turm und eine Dame auf der siebten Reihe sind von großem Vorteil, solange sich noch Bauern auf der Reihe befinden oder ein König auf der achten, analog gilt natürlich das Gleiche für den schwarzen Spieler. Türme auf (halb-)offenen Linien und Türme hinter eigenen Freibauern gehen gut, Türme davor schlecht in die Bewertung ein.

Beim Springer wird auf Mobilität geachtet. Einen Bonus gibt es für einen ewigen Springer auf einem Vorposten. Für Läufer gibt es ein kleines Plus, falls beide Läufer noch auf dem Brett sind. Positiv werden Läuferfianchetti und Läufer auf Vorposten gewertet. GNU Chess überprüft die typischen Läuferfallen auf a2, h2 bzw. a7 und h7.

In die Bewertung der Bauernstruktur geht ein, ob ein Spieler Freibauern, rückständige Bauern, Doppelbauern, Isolanis oder verbundene vorgerückte Freibauern besitzt. Angriffe auf die Basis einer Bauernkette werden ebenfalls berücksichtigt. Bauernangriffe werden als gut eingestuft, wenn Weiß und Schwarz auf verschiedene Seiten rochiert haben.

Zur Bewertung einer Endspielstellung wird zum Beispiel festgestellt, ob der König es noch in das Quadrat eines Bauerns schaffen kann, um eine Umwandlung des Bauern in eine Figur zu verhindern. Ebenso wird das Konzept der Opposition der Könige formalisiert und in die Stellungsbewertung mit aufgenommen.

In die Bewertung gehen weitere Schachweisheiten ein. Wenn man beim Schach im Vorteil ist, sollte man Figuren tauschen und das Tauschen von Bauern vermeiden. Ebenso riechen zwei ungleichfarbige Läufer nach einem Unentschieden.

Die Evaluierungsfunktion von GNU Chess ist komplex. Etwas Hilfe bei der Analyse ist durch die Duplikation von Information in der Repräsentation des Spielbretts gegeben.

6.5 Transpositionstabellen

Ein Spielbaum hat zwar eindeutig unterscheidbare Knoten, eine Situation auf dem Spielbrett kann jedoch an verschiedenen Stellen im Spielbaum auftreten. Beim Schach passiert dies durch einfache Zugumstellung. Solche mehrfach auftretenden Spielsituationen schaffen unnötige Arbeit, würde man sie erneut untersuchen. Deshalb können die Bewertungen von Stellungen in einer *Transpositionstabelle* gespeichert werden [1].

Deshalb verwendet auch GNU Chess Transpositionstabellen. Jeder Eintrag in der Tabelle enthält Informationen über den Nutzwert, das $\alpha\beta$-Fenster, den Halbzug und die Suchtiefe in der die Stellung aufgetreten ist, und natürlich, welcher Spieler an der Reihe war. Bei den Transpositionstabellen gibt es ein großes Hindernis: Man möchte in der Regel weder

Stellungen abspeichern, noch Stellungen vergleichen. GNU Chess verwendet deshalb ein spezielles Hashverfahren: das *Zobrist-Hashing* [2].

Beim Zobrist-Hashing werden Spielstellungen auf Hashwerte abgebildet, und über diese kann auf die Informationen in der Transpositionstabelle zugegriffen werden. Zu den besonderen Eigenschaften dieses Hashverfahrens gehört, dass inkrementelle Änderungen am Spielbrett sich inkrementell auf den Hashwert übertragen. Ein wohl unvermeidbarer Nachteil besteht darin, dass Kollisionen nicht ausgeschlossen werden können.

Das Hashverfahren benötigt eine Menge S aus gleichverteilten Zahlen s in einem Intervall $[0;\ 2^n - 1]$. Die Zahlen stellen Attribute dar, die bestimmten Spielsituationen zugeordnet sind. Das Hashverfahren sieht vor, einer Menge von Attributen je einen Hashwert zuzuordnen. Der Hashwert einer Teilmenge aus S ergibt sich aus der XOR-Verknüpfung aller ihrer Attribute.

Das Verfahren soll anhand eines Spieles erklärt werden. Tic-Tac-Toe eignet sich wegen der Übersichtlichkeit zur Demonstration, das Problem mit den Kollisionen wird erst einmal vernachlässigt. Gegeben seien zwei Matrizen $X = (x_{i,j})$ und $C = (c_{i,j})$ zu je neun Attributwerten aus $[0;\ 31]$. Der Wert eines Attributs ist eine fünfstellige Dualzahl. Ein Spielposition besitzt ein Attribut $x_{i,j}$ (bzw. $c_{i,j}$) genau dann, wenn das Feld (i, j) mit einem Kreuz (bzw. Kreis) markiert ist.

X	0	1	2
0	00010	01111	00001
1	11001	11101	10001
2	00011	11011	10101

C	0	1	2
0	10100	01101	01001
1	10011	11010	00101
2	11100	01100	01011

Tabelle 2: Attribute werden auf Integerwerte abgebildet.

Einer Spielposition in Tic-Tac-Toe wird eine Teilmenge dieser Attribute zugeordnet. Entsprechend kann das Hash-Verfahren dann dieser Teilmenge mit der XOR-Verknüpfung einen Hashwert zuweisen. Die Anfangsstellung entspricht der leeren Menge, und soll für dieses Beispiel den Hashwert 00000 besitzen. Im Folgenden wird illustriert, wie ein solches Hashing im Spiel verläuft.

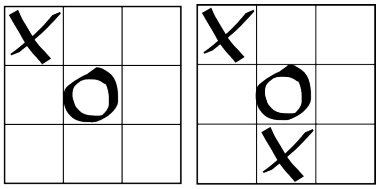


Abbildung 10: Zobrist-Hashing bei Tic-Tac-Toe. Die Position 1 besitzt die Attribute $x_{0,0}$ und $c_{1,1}$. Position 2 besitzt zusätzlich noch das Attribut x_{21}.

Eine explizite Berechnung der Hashwerte für Position 1 ergibt:

$$\begin{aligned} h_1 &= 00000 \circ x_{00} \circ c_{11} \\ &= x_{00} \circ c_{11} \\ &= 00010 \circ 11010 \\ &= 11000 \end{aligned}$$

Vergleicht man die Attributmengen der Positionen 1 und 2, so unterscheiden sie sich nur um das Attribut x_{21}, das besagt, dass unten mittig ein Spieler ein Kreuz gesetzt hat. Der Hashwert für Position 2 läßt sich demnach aus Position 1 ableiten, indem man h_1 einfach mit x_{21} verknüpft.

$$\begin{aligned} h_2 &= h_1 \circ x_{21} \\ &= 11000 \circ 11011 \\ &= 00011 \end{aligned}$$

Das Verfahren ist sehr einfach und die Hashwerte sind schnell zu berechnen. Problematisch wird es, wenn es zu einer Kollision kommt. Das kann zu zwei Fehlern führen. Zum einen kann es passieren, dass Einträge in der Transpositionstabelle überschrieben werden. Das ist noch harmlos. Zum anderen jedoch kann für eine Stellung eine Bewertung aus der Transpositionstabelle gelesen werden, die eigentlich zu einer anderen Stellung gehört. Dies ist unter Umständen fatal, denn wenn es schlecht läuft, wird der Fehler bis zur Wurzel des Spielbaums durchgereicht. Die Wahrscheinlichkeit für einen solchen Fehler kann jedoch kontrolliert werden. Dazu berechnet man für die Stellungen einen zweiten Hashwert mit anderen Attributwerten. Dieser Hashwert, dessen Länge nicht durch die Größe der Transpositionstabelle beschränkt ist, wird im Eintrag der Tabelle abgelegt. Auf diese Art und Weise kann man die Fehlerwahrscheinlichkeit auf ein gewünschtes Maß reduzieren.

GNU Chess verwendet die hier vorgestellten Techniken. Die Attribute entsprechen den Positionen und Farben der Figuren. Auch andere Eigenschaften des Schachspiels werden als Attribute modelliert. Zum Beispiel gehen in den Hashwert die Informationen ein, ob ein Bauer gerade *en passant* genommen wurde. Anstelle des zweiten Hashwerts nehmen sie je nach voreingestellten Größe der Transpositionstabelle einen Teil der Bits eines einzigen Hashwerts, der dann in einem Eintrag der Tabelle nochmal komplett gespeichert wird.

6.6 Sucherweiterungen

In der Praxis ist man gezwungen, die Suche auf eine bestimmte Tiefe zu begrenzen. Das Absägen des Spielbaums auf bestimmter Tiefe führt jedoch zu unerwünschten Phänomenen, wie z.B. dem Horizont-Effekt und dem Abbruch der Suche in wilden Stellungen mit unklarem Ausgang.

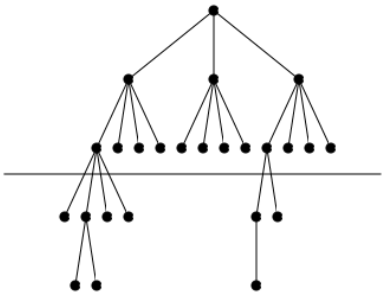


Abbildung 11: Selektive, erweiterte Suche über den Suchhorizont hinaus.

In unruhigen Stellungen wird lokal die Suchtiefe erhöht und weitergesucht, bis eine ruhige Stellung auftritt. Beim Schach vermeidet eine solche Erweiterung, dass eine Stellung bewertet wird, obwohl im nächsten Zug eine Figur geschlagen werden könnte. Im Englischen wird diese Suche als *quiescence search* bezeichnet.

Eine andere unangenehme Situation entsteht, wenn ein unvermeidlicher Zug des Gegners mit einer Hinhalte-Taktik über den Suchhorizont geschoben wird, mit dem Effekt, dass nun die Evaluierungsfunktion diesen unheilvollen Zug nicht mehr in der Stellungsbewertung berücksichtigt [8]. Diesen Effekt nennt man *Horizonteffekt*. Er könnte beim Schach in einem Szenario auftreten, wo ein schwarzer Bauer auf der zweiten Reihe kurz vor der Umwandlung steht und Weiß mit ein, zwei Schachansagen die Umwandlung verzögert, den Bauern jedoch langfristig nicht mehr aufhalten kann. Die beiden Schachansagen schieben den Umwandlungszug über den Suchhorizont.

7 Fazit

Die Alpha-Beta-Suche bietet die Grundlage für viele Algorithmen im Bereich der Nullsummenspiele für zwei Personen. Die Alpha-Beta-Suche hat zur vollständigen Lösung des Mühlespiel beigetragen [7]. Im Kampf um die Effizienz gibt es vielerlei Heuristiken und Verbesserungen, von denen einige in dieser Arbeit angesprochen wurden. Über den Nutzen von Heuristiken lässt sich nicht immer eine klare Aussage treffen, weshalb in der Literatur häufig Ergebnisse von Experimenten zu finden sind [5, 4]. Bei der Implementierung eines Suchalgorithmus ist die Evaluierungsfunktion eine der interessanten Baustellen. Hier fließen Erfahrungswerte ein, die überprüft und mit der richtigen Gewichtung versehen werden müssen [8]. Eine interessante, offene Frage bleibt für mich, ob sich der Erfolg der Alpha-Beta-Suche und ihrer Erweiterungen im Durchsuchen von Spielbäumen auch in andere Domänen übertragen lässt.

A Quelltextausschnitte

Für den Quelltextausschnitt, der die Implementierung der Alpha-Beta-Suche beschreibt, wurde auf einem explizit dargestellten Spielbaum in einer Tupelnotation gearbeitet. Die explizite Darstellung ist in einer Implementation sicherlich nicht zu finden, für Test- und Anschauungszwecke sind explizite Darstellungen jedoch hilfreich.

```
def node(*children):
    return (None, children)

def leaf(value):
    return (value, None)

def is_leaf(node):
    value, children = node
    return children == None

def value(node):
    value, children = node
    return value

def children(node):
    value, children = node
    return children
```

Der Spielbaum aus Abbildung 8, an dem die Alpha-Beta-Suche illustriert wurde, kann auf folgende Weise dargestellt werden.

```
def sample_tree():
    return node(
        node(
            leaf(4),
            leaf(-2)),
        node(
            leaf(-3),
            leaf(2)))
```

B Literaturhinweise

Sowohl die Minimax- als auch die Alpha-Beta-Suche sind sehr gut erforscht und es gibt dementsprechend eine große Auswahl an Literatur. Einen ersten Überblick über die Minimax- und die Alpha-Beta-Suche bieten S. Russell und P. Norvig in [8]. Eine sehr gute Einführung, Begriffsbildung und Einordnung existierender Algorithmen für Suchen in Spielbäumen bietet A. Reinefeld in [3]. Eine Übersicht über die möglichen Heuristiken gibt A. Junghanns in [1]. Eine kompakte und verständliche Einführung in die Spieltheorie stellen K. Leyton-Brown und Y. Shoham mit [6]. Informationen zu GNU Chess kann man direkt aus dem Quelltext entnehmen.

Literatur

[1] A. Junghanns: *Are There Practical Alternatives To Alpha-Beta in Computer Chess?* ICCA J., 21(1), März 1998.

[2] A. L. Zobrist: *A New Hashing Method with Application for Game Playing.* Technischer Bericht 88, Univ. of Wisconsin, 1970.

[3] A. Reinefeld: *Spielbaum-Suchverfahren.* Springer-Verlag, Berlin, 1989.

[4] D. E. Knuth und R. W. Moore: *An analysis of alpha-beta pruning.* Artificial Intelligence, 6:293–326, 1975.

[5] J. Schaeffer: *The History Heuristic and Alpha-Beta Search Enhancements in Practice.* IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-11(11):1203–1212, November 1989.

[6] K. Leyton-Brown und Y. Shoham: *Essentials of Game Theory.* Morgan & Claypool Publishers, 2008.

[7] R. Gasser: *Solving Nine Men's Morris.* Computational Intelligence, 12:24–41, 1996.

[8] S. J. Russell und P. Norvig: *Artificial Intelligence: A Modern Approach.* Prentice Hall, zweite Auflage, 2003.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest und C. Stein: *Introduction to Algorithms.* MIT Press, Cambridge, MA, 2001.

[10] T. Legendi und T. Szentivanyi (Herausgeber): *Leben und Werk von John von Neumann. Ein zusammenfassender Überblick.* Bibliographisches Institut, 1983.