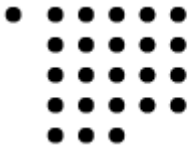


Fachhochschule Köln
University of Applied Sciences Cologne

Praktikum: Algorithmische Anwendungen

im WS 2005/2006

25.01.2006 – Gruppe A rot

Projektthema: Alpha-Beta-Suche

Jens Krenkers 110 36 925
Meik Romberg 110 36 567

Einleitung

Die Alpha-Beta-Suche ist ein Verfahren, welches den klassischen Minimax-Algorithmus verbessert, in dieses Verfahren Pfade im Baum verwirft, welche für das weitere Spielgeschehen unwichtig sind.
Die Idee, die hinter der Modifikation steht, ist die folgende: Während des Suchvorgangs für den nächsten Spielzug muss nicht jeder Spielzug (nicht jeder Node des Suchbaumes) abgelaufen werden, um zu einer korrekten Entscheidung zu gelangen. Anders ausgedrückt: Wenn der ausgewählte Pfad (Spielzug) ein schlechteres Ergebnis liefert als die aktuelle beste Wahl, dann ist der erste Spielzug, welchen der Gegner durchführen kann, welcher immer noch schlechter ist, als der beste Spielzug des Betrachters, der letzte Spielzug, den man einer Betrachtung unterziehen muss. Der Gegner wird letztendlich diesen Zug wählen. Siehe dazu folgende Abbildung:

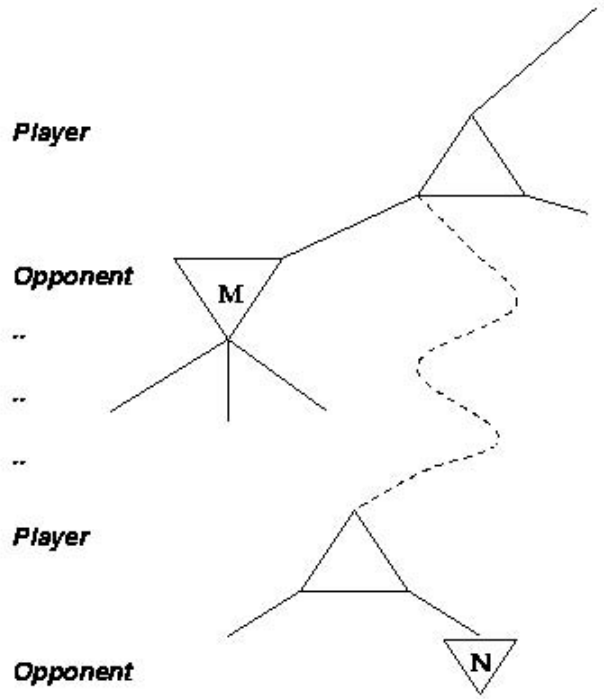


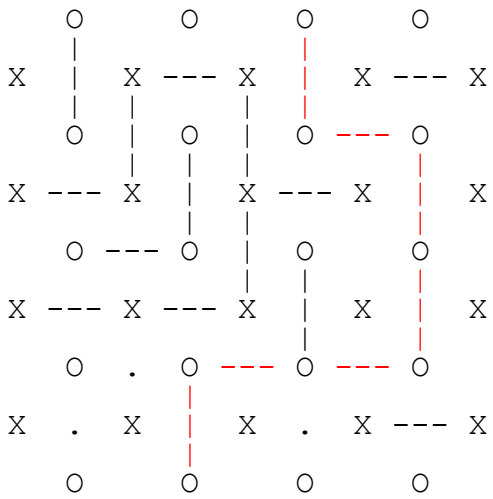
Abb. 1: Falls für den Spieler der Spielzug M besser ist als Zug N, dann wird niemals Spielzug N erreicht.

Eine detailliertere Beschreibung und Darstellung eines Beispielbaumes folgt später.

Betrachten wir zunächst ein Spielbeispiel, wie Tic-Tac-Toe oder Gale, um die Idee zu vertiefen. Das Spiel Gale verwendet in der Regel eine 9x9-Matrix, in der die Zeilen abwechselnd mit O- und X-Symbolen gefüllt werden, wobei immer das nächste Feld frei bleibt. Eine weitere Bedingung ist, dass die Zeilen versetzt zueinander angeordnet werden:

	O		O		O		O	
X		X		X		X		X
	O		O		O		O	
X		X		X		X		X
	O		O		O		O	
X		X		X		X		X
	O		O		O		O	
X		X		X		X		X
	O		O		O		O	

Ein Spieler spielt die O-Symbole, der andere die X-Symbole. Ziel des Spiels aus Sicht von Spieler O ist es, eine geschlossene Linie von einem beliebigen O in der ersten Zeile zu einem beliebigen O in der letzten Zeile zu generieren. Spieler X versucht analog eine geschlossene Linie von der ersten Spalte zur letzten Spalte zu erreichen. Dabei verbindet jeder Spieler jeweils abwechselnd zwei gleiche Symbole miteinander. Es dürfen sich jedoch keine Verbindungen überkreuzen.
Bei diesem Spiel kann es kein Unentschieden (Remis) geben, denn wenn ein Spieler nicht mehr die Chance hat zu gewinnen, jedoch in irgendeiner Weise an jeder Stelle sein Weg abgeschnitten ist, dann muss bereits der andere Spieler gewonnen haben. Hierzu ein Beispiel:



Die rote geschlossene Linie stellt die Gewinnstellung für Spieler O dar.

Theoretische Grundlagen

Für eine solche Spielbaumsuche benötigt man eine statische Bewertungsfunktion, welche eine Spielstellung als Eingabe auf eine ganze Zahl abbildet. In der Literatur findet man verschiedene Varianten der Bewertung. Wir verwenden hier die Variante, dass ein großer Wert eine bessere Bewertung für den Spieler, der am Zug ist, darstellt, während ein kleiner Wert einen Vorteil des Gegners charakterisiert.
Handelt es sich bei der Position um eine **Siegstellung**, wird ein sehr **hoher Wert** zurückgegeben. Zusätzlich wird ein **früher Sieg höher bewertet**, als einer, der erst später erreicht wird, siehe Baum.

Es ist bei der Suche möglich eine maximale Tiefe anzugeben, die durchlaufen werden soll.
Ein **Knoten** stellt im Baum eine **Spielstellung** bzw. ein Bewertung dar; die Kanten entsprechen dem Zug, der von einer Spielstellung zur nächsten führt.
Ein **Blatt** ist stellvertretend entweder für eine **Gewinnstellung oder** für eine **Spielstellung**, für die die **maximale Suchbaumtiefe erreicht** ist.
In einem Blatt entspricht die Bewertung tatsächlich dem Ergebnis der Bewertungsfunktion. Die Bewertung der inneren Knoten ergibt sich jeweils abwechselnd je Ebene als das Minimum oder Maximum der Bewertung seiner Kinder.

Die einzelnen **Ebenen** charakterisieren zudem die **abwechselnden Spielzüge**. Der von uns betrachtete Spieler wird dementsprechend versuchen, als Ergebnis eines eigenen Zuges die Bewertung zu maximieren und die Bewertung für die Auswahl, die dem Gegner bleibt, zu minimieren (analog der von uns vorgestellten Präsentation aus dem letzten Praktikum, siehe auch Wikipedia zum Thema Alpha-Beta-Suche bzw. Minimax-Algorithmus).

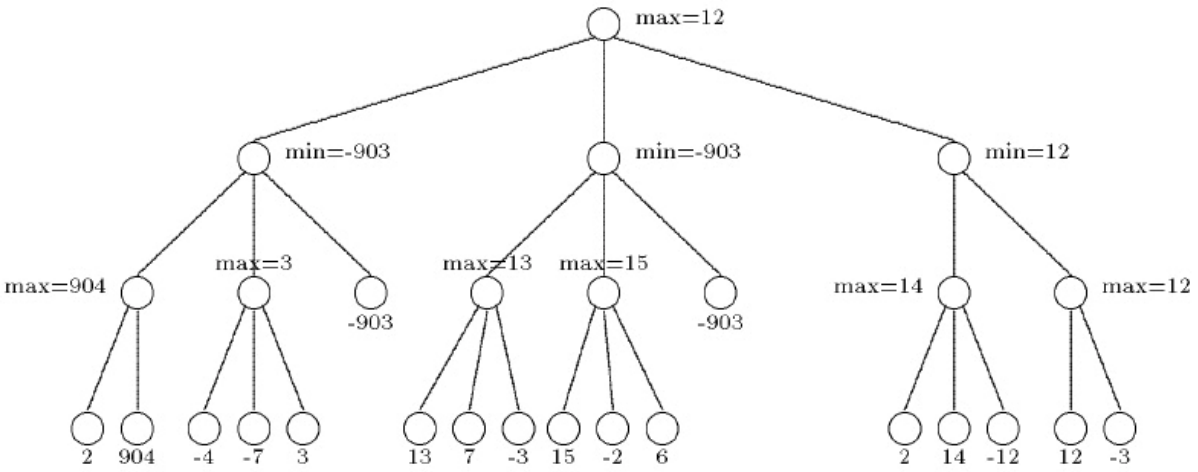


Abb. 2: Ein Baumausschnitt mit Bewertung der einzelnen Knoten. Ein Vaterknoten erhält dabei von unten nach oben, je nach Ebene die größte (max-) bzw. die niedrigste Bewertung (min-Wert) aller seiner Söhne (vgl. z.B. ganz rechten Pfad).

In diesem Beispielbaum wird man den rechten Zug wählen, da er den größtmöglichen Gewinn verspricht. Die klassische Min-Max-Suche wird angewendet auf diesen Baum jedes Blatt bewerten und die Min- und Max-Werte für die inneren Knoten berechnen. Eine Analyse dieses Laufzeitverhaltens erfolgt in einer späteren Praktikumssitzung.

Der **Alpha-Beta-Algorithmus** optimiert diese Baumsuche, indem solche Zweige, die bereits einen Wert liefern, welcher den Wert des zu vergleichenden Knotens nicht mehr verbessern können, nicht mehr weiter untersucht werden, siehe **schwarze Knoten** (Blätter) im Baumdiagramm.

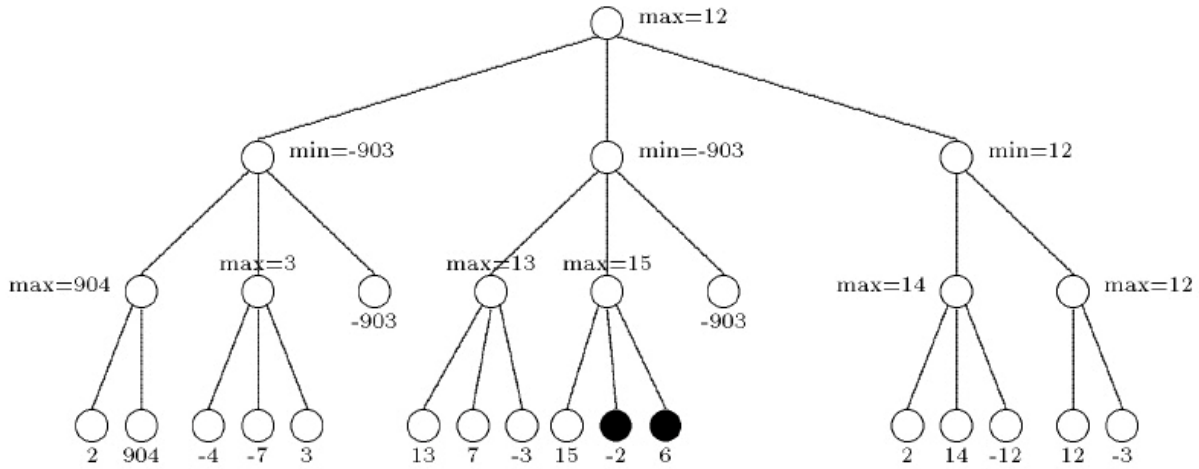


Abb. 3: Beim Alpha-Beta-Cut werden Knoten vernachlässigt, welche den Wert der zu vergleichenden Knotens nicht mehr verbessern können.

Sobald klar ist, dass dieser Zweig mindestens einen Max-Wert von 15 bringt, und somit das darüberliegende Minimum, welches bis dahin mit 13 bewertet war, nicht mehr verbessert werden kann, müssen die zwei markierten Knoten nicht mehr untersucht werden.

Vertiefung des Minimax-Prinzips anhand des TicTacToe-Baumes

Vorüberlegungen: Beim Spiel Tic-Tac-Toe kommt ein quadratisches Spielfeld mit 9 Feldern zum Einsatz. Zwei Spieler setzen abwechselnd jeweils Kreuze oder Kreise. Wer zuerst eine Linie (senkrecht, waagerecht oder diagonal) mit jeweils 3 Kreuzen bzw. Kreisen erzeugt, gewinnt das Spiel.
Folgerungen: Der resultierende Spielbaum hat folglich eine maximale Tiefe von 9. Im Baum werden alle möglichen Spiele betrachtet (siehe Abbildung 4). In den Blättern hat entweder der betrachtete Spieler oder der Gegner gewonnen oder es kommt zum Remis (Anmerkung: Dieser Zustand wird nahezu immer erreicht, wenn man den

Computer mit Hilfe eines implementierten Alpha-Beta-Algorithmus gegen sich selbst antreten läßt).

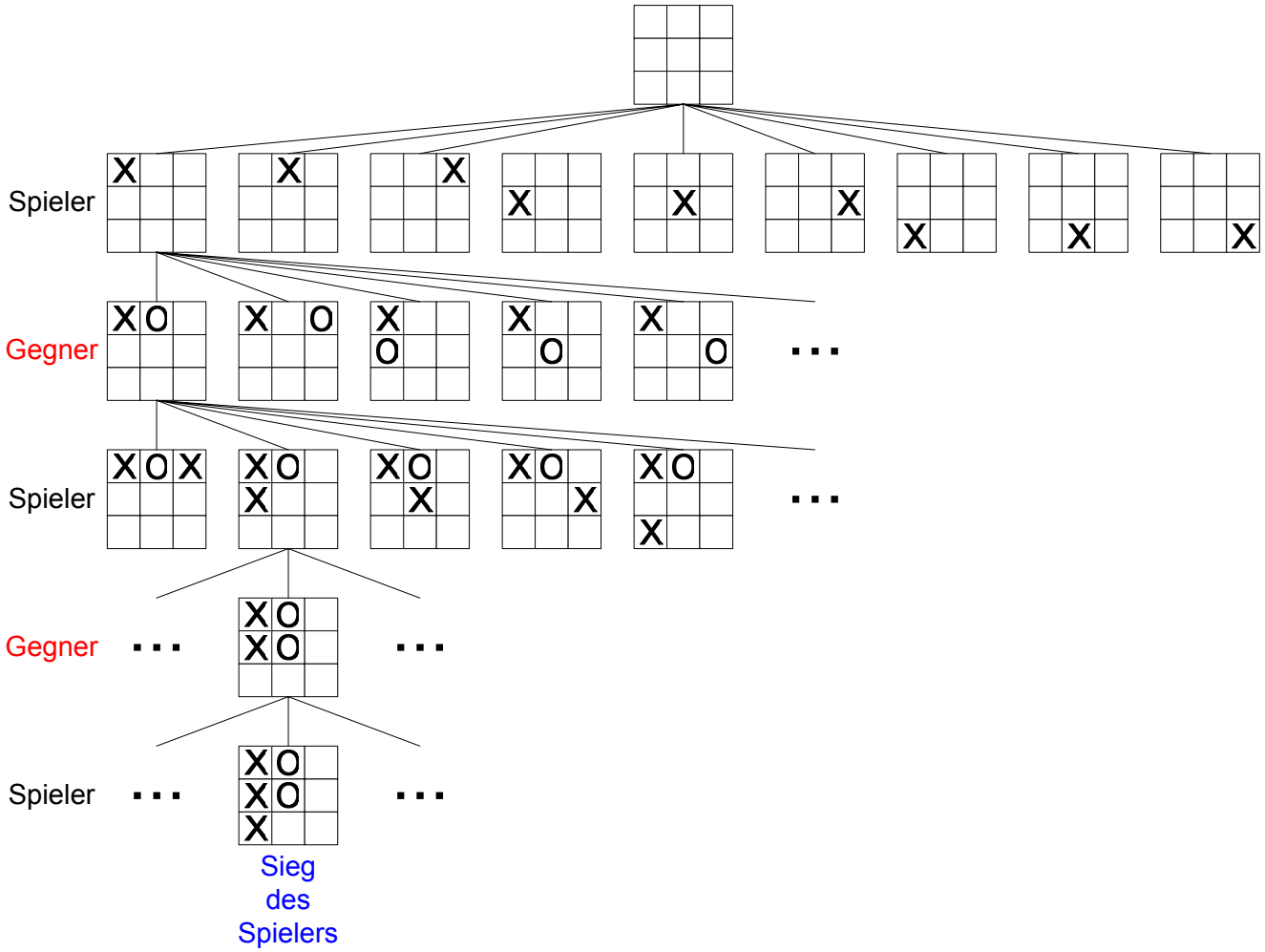


Abb. 4: Der vollständige Spielbaum für das Spiel TicTacToe

Daraus ergibt sich die Fragestellung: Wie kann man nun daraus folgern, welcher Zug am Anfang des Spiels der Beste ist?

Die Bewertung der Knoten

Die Idee dahinter besteht darin, ausgehend von den Blättern nach oben hin im Baum eine Bewertung vorzunehmen. Beim Spiel TicTacToe kann man eine recht einfache Bewertungsgrundlage ansetzen:

Bewertungsfunktion *B* mit Blatt *b*:

$$B(b) = \begin{cases} -1, \textit{falls der Gegner gewinnt}, \\ 0, \textit{falls unentschieden}, \\ 1, \textit{falls der Spieler gewinnt} \end{cases}$$

Die Bewertung eines Vaterknotens *v* wird wie folgt vorgenommen: *v* habe *n* Blätter $b_1 \ldots b_n$.
Wenn im Vaterknoten v der **Gegner am Zug** war, ist danach der Spieler am Zug und kann dann durch seinen Zug die beste Alternative aus den b_i wählen (z.B. Gewinnen).
Man setzt also demnach $B(v) = max \{ B(b_1), \ldots, B(b_n) \}$

Wenn im Vaterknoten v der **Spieler am Zug** war, ist die Situation gerade umgekehrt: Der Gegner kann die für sich beste Alternative wählen, also ist in diesem Fall
$B(v) = max \{ B(b_1), \ldots, B(b_n) \}$
Anschließend wird diese Bewertungsstrategie sukzessive bis zur Wurzel verfolgt, siehe Abbildung 5.

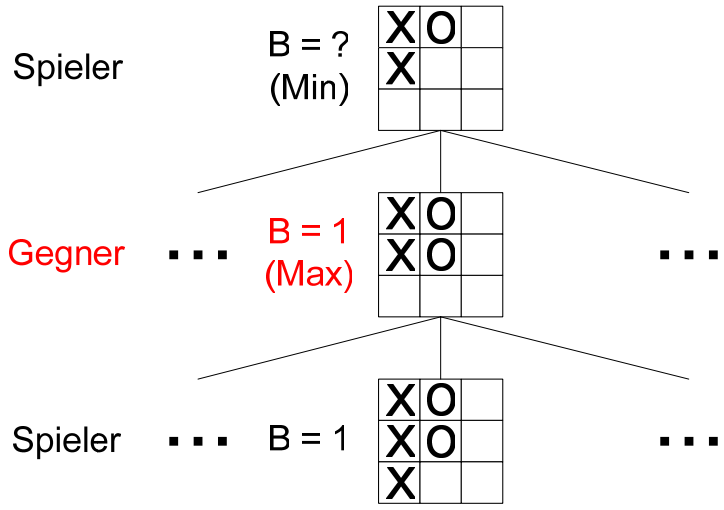


Abb. 5: Die Bewertung eines Pfades (Auszug)

Bewertungsfunktionen sind von Spiel zu Spiel verschieden. Beim Schach sieht eine Implementierung einer Bewertungsklasse weitaus komplexer aus.

Anwendung des Minimax-Algorithmus auf das Spiel TicTacToe und Bewertung der Knoten

Betrachten wir folgenden Ausschnitt aus dem Spielbaum für TicTacToe. Beide Spieler haben jeweils abwechselnd Spielzüge vorgenommen und das aktuelle Spielbrett sieht so aus, wie es die Wurzel des Baumes darstellt.
Jeder weitere Knoten repräsentiert eine weitere mögliche Spielsituation. Die Bewertung wird nun von unten nach oben vorgenommen. Nach der Bewertungsfunktion erhalten die Blätter demnach die Werte wie in Abbildung 6 dargestellt (Bewertungen für Blätter sind orangefarben, für Knoten blau).

An einem Knoten muß der Algorithmus jeweils eine Entscheidung zur Bewertung desselben treffen. Diese hängt unmittelbar von der Ebene ab, sprich, welcher Spieler ist am Zug. Für die tiefste Knotenebene (die 3. Ebene ab der Wurzel) ist der von uns betrachtete Spieler am Zug. Folglich findet der Max-Part des Algorithmus Anwendung und es wird der Max-Wert herausgesucht und in die entsprechenden Knoten geschrieben. Wie man an der Abbildung sieht, gibt es auf dieser Ebene eh jeweils eine einzige Möglichkeit (eine resultierende Spielsituation), wodurch diese Bewertung trivial ausfällt.
Auf der 2. Ebene ist nun für jeden Knoten eine Entscheidung notwendig. Da hier der Gegner (Symbol O) am Zug ist, findet der Min-Part des Minimax-Algorithmus Anwendung. Auf der obersten Ebene ist wiederum der von uns betrachtet Spieler am Zug und es ergibt sich als Wert für diesen Knoten eine 0.
Dieses Ergebnis geht einher mit der Beobachtung, daß es für den Spieler am günstigsten ist, auf jeden Fall den rechten Zug im Baum (also den rechten Pfad) zu wählen, da er in diesem Fall nicht verlieren kann. Es kommt im schlechtesten Fall zu einem Remis (0). Läuft der Gegner in die „Falle“, so gewinnt der Spieler sogar.

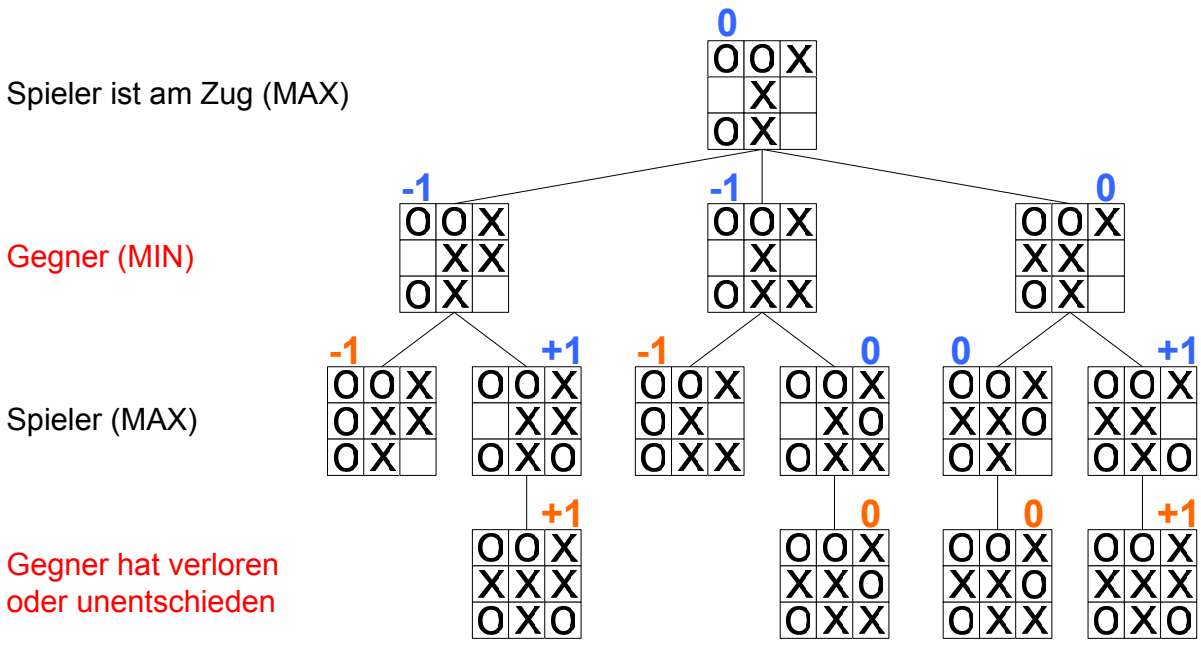


Abb. 6: Die Bewertung eines Teilbaumes detailliert dargestellt

Im Falle des TicTacToe-Spiels wird der Alpha-Beta-Algorithmus für diesen Teilbaum ein ähnliches Ergebnis liefern, da der Wertebereich für dieses Spiel recht eingeschränkt ist.

Schlußfolgerungen zum Minimax-Algorithmus

Wie man gesehen hat, bestimmt der Wert an der Wurzel, wer (unter Annahme einer optimalen Spielstrategie des von uns betrachteten Spielers) das Spiel gewinnt.
Jedes Strategiespiel (mit Ausnahme von Spielen, bei denen zusätzlich der Zufall analysiert werden muß, wie z.B. bei Backgammon) ist auf diese Weise lösbar. Einziges Problem sind komplexere Spiele mit großen Spielbäumen.
Wie am vorgestellten Beispiel (TicTacToe) aufgezeigt, betrachtet der Minimax-Algorithmus in diesem Fall **9!** (Fakutät) Blattknoten. Der Rechenaufwand dafür ist noch akzeptabel.
Im Falle des Schachspiels sieht der Aufwand schon ganz anders aus: Im Schnitt kann man von ca. 35 Verzweigung pro Nicht-Blattknoten ausgehen. Ingesamt gibt es 10^{40} Knoten zu analysieren, was nicht mehr berechenbar ist.

Verbesserung des Minimax-Algorithmus

Die Hauptprobleme bei komplexen Spielen (Schach, Backgammon etc.) stellen die Tiefe und die Breite des Suchbaums dar. Folglich ist es erstrebenswert, einen solchen Baum von vornherein zu verkleinern, z.B. durch Beschneiden der Tiefe.
Hierzu bricht man ab einer gewissen Suchtiefe ab. Falls dabei ein Nicht-Blattknoten vorliegt, benötigt man eine gewisse Heuristik zur Bewertung. Beim Schach zieht man beispielsweise den Figurenvorteil heran. Diese Heuristik ist der aufwendige Teil bei guten Spielimplementierungen. Am Minimax-Algorithmus selbst kann man nichts mehr dahingehend modifizieren.
Außerdem ändert sich Regel bei komplexen Spielen und der Umsetzung der dazugehörigen Spielregeln der Wertebereich der Bewertungsfunktion B (nicht mehr -1, 0, 1, wie im Falle des TicTacToe's).
Eine weitere Möglichkeit der Verkleinerung des Suchbaums ist das Beschneiden in der Breite. Dabei werden Symmetrien ausgenutzt (im Falle von TicTacToe Spiegel- und Punktsymmetrien). Des weiteren könnte man den Baum als azyklischen gerichteten Graphen betrachten und kontrollieren, ob es bereits die gleiche Spielsituation schon einmal in einem anderen Unterbaum gegeben hat. In diesem Falle kann man dann das Ergebnis von dort verwenden.
Eine effektive Methode zur Beschneidung in der Breite ist eben die Alpha-Beta-Beschneidung (auch Alpha-Beta-Suche oder engl. Alpha-Beta-Cut oder Alpha-Beta-Pruning).

Der Alpha-Beta-Algorithmus

Nachdem ein Basisverständnis der Abläufe vermittelt wurde, kann man nun den Algorithmus einer genaueren Untersuchung unterziehen. Vorweg folgen einige Definitionen, welche im Algorithmus verwendet werden. Danach folgt eine Darstellung des Algorithmus in Pseudocode.

Definitionen

Alpha (α):
Minimales Ergebnis, welches der Spieler MAX auf jeden Fall erreicht

Beta (β)
Maximales Ergebnis, welches der Spieler MAX sich gegen einen aufmerksamen Gegner erhoffen kann

Pseudocode

```
function ALPHABETA (v:Spielbrett, alpha, beta, tiefe, tmax:integer) :integer;

var m, j :integer;

begin
        if ((v ist Gewinnstellung) or (tiefe = tmax)) then
                return Bewertung(v);
        repeat
                generiere Nachfolgepositionen v.j von v;

                m:=ALPHABETA (v.j, alpha, beta, tiefe+1, tmax);

                if ((v ist MAX-Knoten) and (m > alpha)) then
                        alpha:=m;
                if ((v ist MIN-Knoten) and (m < beta)) then
                        beta:=m;
                if (alpha >= beta) then
                        Cutoff;
                        break;          /* Schleife verlassen */
        until (alle Nachfolgepositionen betrachtet);

        if (v ist MAX-Knoten) then
                return alpha;
        if (v ist MIN-Knoten) then
                return beta;
end;
```

Eine erste Implementierung in Java

```
public static int alphaBeta (MinMaxNode node, int depth, int alpha, int beta)
{
    if (node.isLeaf() || depth <= 0)
        return node.evaluate();

    Iterator moves = node.getMoves();

    if ( node.isMinNode() )
    {
        while ( moves.hasNext() )
        {
            int val = alphaBeta ( (MinMaxNode)moves.next(), depth-1, alpha, beta);
            if ( val < beta )
                beta = val;
            if alpha >= beta
                return beta;
        }
        return beta;
    }
    else
    {
        while ( moves.hasNext() )
        {
            int val = alphaBeta ( (MinmaxNode)moves.next(), depth-1, alpha, beta);
            if ( val > alpha )
                alpha = val;
            if alpha >= beta
                return alpha;
        }
        return alpha;
    }
}
```

Cut-Off - Der Alpha-Beta-Algorithmus im Einsatz:

Die folgenden Screenshots des JavaApplets zeigen exemplarisch einen Ablauf des Algorithmus in der Baumstruktur.

Farbgebung (Helle Farbtöne sind vom Algorithmus betrachtete Knoten oder Blätter):

Grün = Spieler (Maximum)
Rot = Gegenspieler (Minimum)
(x,y) = (Alpha,Beta) Werte

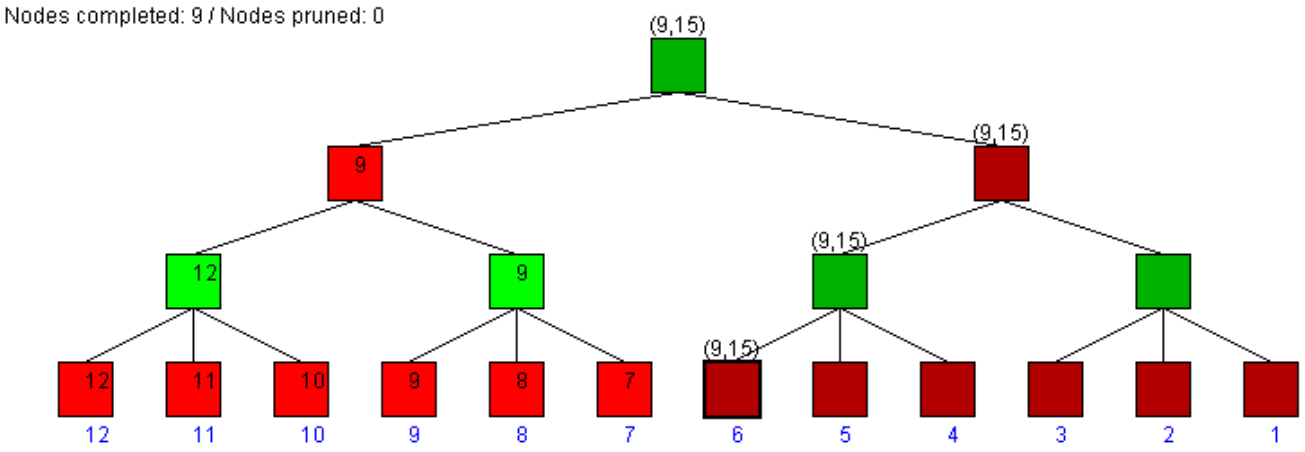


Abb. 7: Zustand nach Abarbeitung des linken Teilbaumes

Diese Abbildung zeigt den Baum im Zustand nach der Abarbeitung des linken Teilbaumes. Die hellroten Blätter zeigen, dass zuvor alle Blätter betrachtet worden sind. In den hellgrünen Knoten in Level 3 wurde jeweils das Maximum aus den Blättern darunter gespeichert. Der hellrote Knoten mit der 9 speichert das Minimum der Kinder, da es sich um den Zug des Gegenspielers handelt.

In der Ausgangsposition oben wird das erste Blatt des rechten Teilbaumes mit den zuvor ermittelten Alpha-Beta-Werten verglichen.

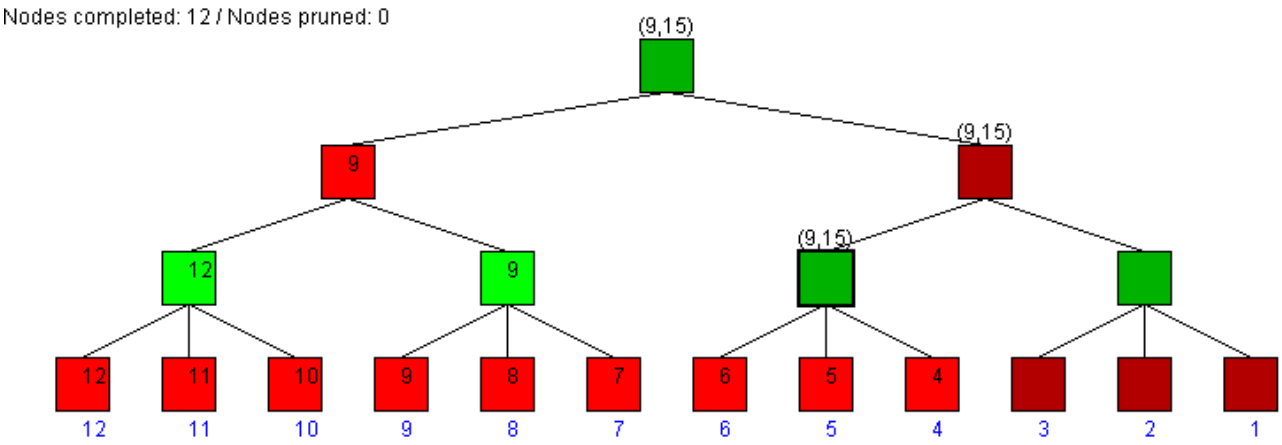


Abb. 8: Kein höherer Wert in den Blättern gefunden

Diese Situation zeigt, dass alle drei, jetzt hellroten Blätter des Teilbaumes betrachtet worden sind. Jeder Wert ist kleiner als der Alpha-Wert.

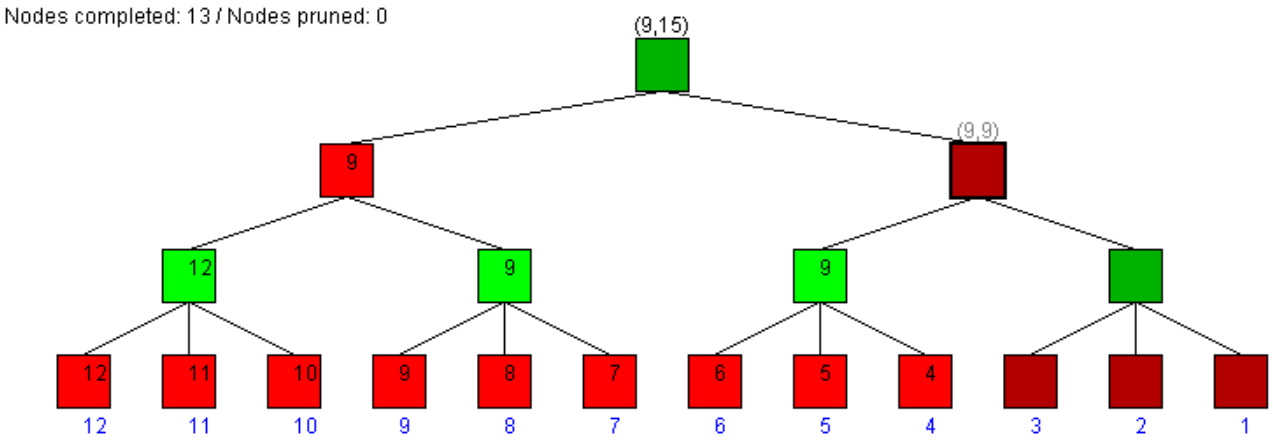


Abb. 9: Alter Alpha-Wert wird gespeichert. Beta-Wert des roten Min-Knotens wird auf den Rückgabewert des Teilbaumes gesetzt.

Es wurde kein höherer Wert ermittelt. Der **alte** Alpha-Wert wird in den Knoten übernommen, **obwohl keiner der folgenden Blätter diesen Wert hat!** An dieser Position wird der Beta-Wert im darüber liegenden Minimum-Knoten mit dem Rückgabewert des Maximum-Knotens ersetzt. Diese Regel sieht der Algorithmus vor, wenn der Rückgabewert aus der Rekursion kleiner als der Beta-Wert ist, denn der Gegner hat somit einen besseren „schlechten Zug“ gefunden.

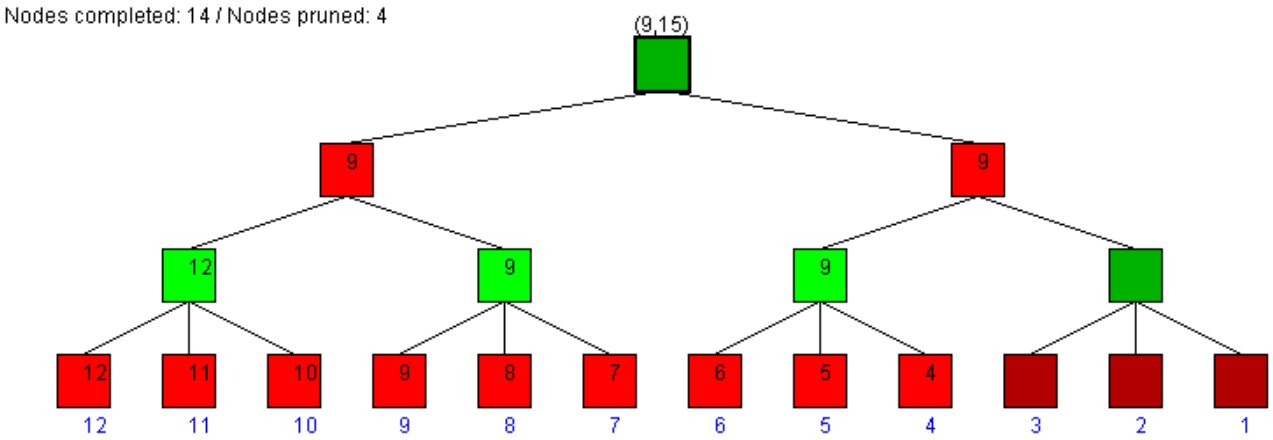


Abb. 10: Cut Off, der letzte Teilbaum wird ignoriert

An dieser Stelle tritt der Cut-Off ein. Da der Alpha-Wert größer oder gleich dem Beta-Wert ist, braucht kein weiterer Teilbaum mehr betrachtet zu werden. Denn egal, welcher höhere Wert noch in den folgenden Wurzeln oder Teilbäumen des Min-Knotens versteckt ist, auf dieser Ebene würde der Gegner den niedrigeren Wert wählen. Ein kleinerer Wert wird ebenfalls nicht mehr zum Vorteil des Opponenten gespeichert, da der Alpha-Wert nicht verändert wird, wenn kein höherer Wert ermittelt wird.

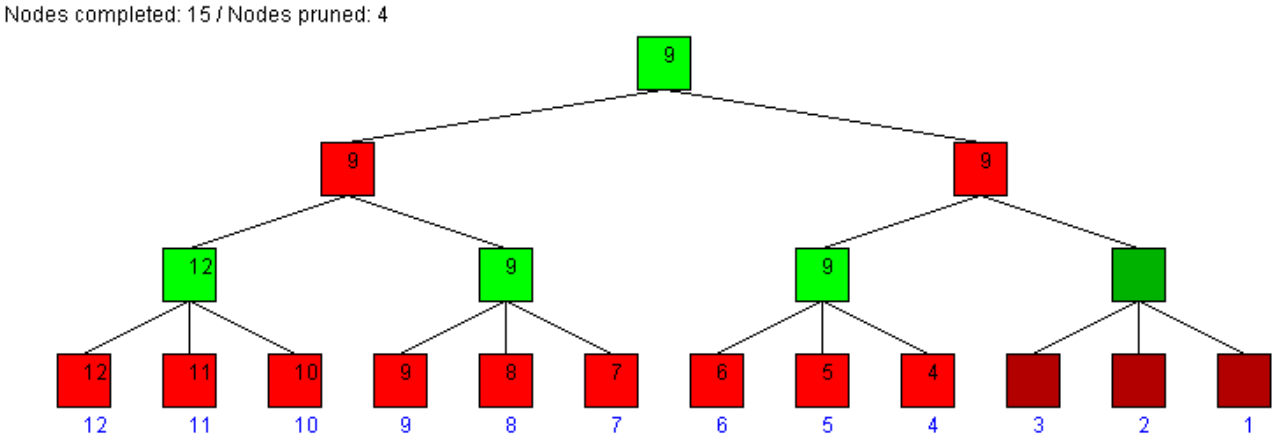


Abb. 11: Endzustand nach Abarbeitung des Baumes

Diese Abbildung zeigt den letzten Speichervorgang für die Wurzel. Der Rückgabewert ist „9“. Es wurde somit kein besserer Spielzug für den Spieler ermittelt und die „9“ wird als optimaler Zug in die Wurzel gespeichert.

Subjektive Erfahrungen zum Spielverlauf und zum Laufzeitverhalten der beiden Algorithmen

Wir haben das Spiel TicTacToe auf Basis einer vorhandenen, recht einfachen Implementierung mit den beiden Algorithmen Minimax und AlphaBetaSearch umgesetzt. Bei beiden Algorithmen wird die Suchtiefe im Baum durch die Variable **skill** beschrieben, die von außen per **PARAM** (HTML-Source) an das Applet übergeben oder im Source-Code hart codiert eingestellt wird.
Aufgrund der Einfachheit des Spiels TicTacToe hat man als Human Player gegen einen Computergegner grundsätzlich nur ein Gewinnchance bei Suchtiefen bis max. 5 (**skill = 5;**). Bei höheren Skill-Levels, also höher Suchtiefe im Baum, läuft das Spiel immer auf ein unentschieden hinaus, wenn dem Human Player kein Fehler unterläuft. Da der Spielbaum bei TicTacToe je Spielsituation noch recht überschaubar ist, lasten die Berechnungen durch die beiden Algorithmen einen aktuellen PC nicht aus, so dass subjektiv kein Geschwindigkeitsgewinn wahrgenommen wird. Den Unterschied soll hierbei eine genaue Laufzeitanalyse zeigen.

Des Weiteren haben wir in eine fertige Implementierung eines Schachspiels zusätzlich den Minimax-Algorithmus eingepflegt, um dort ebenfalls beide Algorithmen zu testen. Es fiel sofort auf, dass die Alpha-Beta-Suche deutlich schneller bei der Planung eines neuen Schachzuges des Computergegners arbeitete, als der Minimax-Algorithmus. Dies geht einher mit der massiv größeren Komplexität des Schachspiels.
Das Schachspiel ließ sich jedoch für genauere Untersuchungen des Minimax-Algorithmus nicht heranziehen, da bei Verwendung desselben die Spielzüge des Computergegners nicht selten mehr als 20 Minuten Rechenzeit benötigten.
Im Vergleich dazu benötigte der Alpha-Beta-Algorithmus einige Sekunden, im schlechtesten Fall wenige Minuten.
Letztendlich bestätigen diese Eindrücke unsere Vermutung, dass die Alpha-Beta-Suche bei komplexen Spielen eine deutliche Geschwindigkeitssteigerung erzielt.

Tatsächliches Laufzeitverhalten der beiden Algorithmen

Die einfache Minimax-Suche, auf einen typischen Baum angewendet, wird jedes Blatt erzeugen und die Minimax-Werte für die inneren Knoten berechnen. In der Laufzeitanalyse wird im Allgemeinen nur die Anzahl der Blätter, also die Zahl der benötigten Aufrufe der Bewertungsfunktion benutzt. Die Zahl der inneren Knoten ergibt sich bei einem Baum, der jedes Blatt in der maximalen Tiefe hat, als kleiner als die Zahl der Blätter. Zudem wird angenommen, dass der Aufwand für die Bewertungsfunktion größer ist als der Verwaltungsaufwand für die inneren Knoten. Bei einem Verzweigungsfaktor ***b***, das heißt jede Stellung erlaubt ***b*** Züge und einer maximalen Tiefe ***n*** ist die Laufzeit für die einfache Minimax-Suche also $\Theta(b^n)$. Eine

Unterscheidung nach **Average Case** und **Worst Case** gibt es hier nicht, da der Minimax-Algorithmus immer den gesamten Baum durchläuft.
Wie bereits bekannt, bringt die Alpha-Beta-Suche im Vergleich dazu einen Fortschritt, da Zweige, die bereits einen Wert liefern, den ein Minimal- oder Maximal-Wert nicht mehr verbessern kann, nicht weiter betrachtet werden. Die Alpha- und Beta-Werte bilden dabei eine Art Schranken eines Fensterbereichs. Liegen Werte innerhalb dieses Fensterbereiches, so verkleinern sie den Fensterbereich und die Berechnung läuft weiter. Kommt es zu einer Überschneidung der Schranken, so tritt der Cutoff ein. In dem Fall macht es keinen Sinn mehr, weitere Knoten auf der gleichen Ebene (also Brüder) zu betrachten, da diese das Ergebnis nicht mehr verändern werden. Der Spieler wird eh den Zug wählen, der in der Tiefe 0 (Root) eine letzte Verbesserung des Alpha-Wertes gebracht hat.
Es ist jedoch nur schwer vorherzubestimmen, wie sich der Algorithmus bezogen auf die Laufzeit verhält, weil dies von vielen weiteren Faktoren abhängt, wie z.B. Spieltyp, Bewertungskomplexität, zu betrachtende Suchtiefe, eventuell Sortierung des Baumes und bereits vollzogene Spielzüge. Die Meinungen dazu gehen auch in der Literatur weit auseinander.
Nach D. Knuth und R. Moore ("An Analysis of Alpha-Beta Pruning", Artificial Intelligence, vol. 6, no. 4, pp. 293-326, 1975) ergibt sich für den Alpha-Beta-Algorithmus ein Laufzeit von:

$b^{\lceil n/2 \rceil} + b^{\lfloor n/2 \rfloor - 1}$, also ungefähr $b^{n/2}$.

Ein direkter Vergleich anhand eines Beispiels:
Es sei $b = 10$ und $n = 9$.
Daraus ergeben sich für den Minimax-Algorithmus 10^9 Operationen. Die Alpha-Beta-Suche benötigt hier $10^5 + 10^4$ Operationen.

```
Minimax:            1.000.000.000
Alpha-Beta:               110.000
```

Im **Worst Case** degeneriert die Alpha-Beta-Suche zu einem Minimax-Algorithmus, d.h. alle Knoten und Blätter müssen abgearbeitet werden, wodurch sich die bereits ermittelte Laufzeit von b^n ergibt.
Abschließend folgen einige Testergebnisse unserer Untersuchungen in Abhängigkeit vom Skill, also der Suchtiefe im Baum.

```
Minimax:
-------------------------
Skill:    - Besuchte Nodes
     9    - 549946
     8    - 59705
     7    - 45737
     6    - 24425
     5    - 8081
     4    - 2081
     3    - 401
     2    - 65
     1    - 9
```

```
AlphaBeta:
--------------------------------
Skill:    - Besuchte Nodes
     9    - 16811
     8    - 3531
     7    - 2835
     6    - 1973
     5    - 649
     4    - 283
     3    - 137
     2    - 32
     1    - 9
```

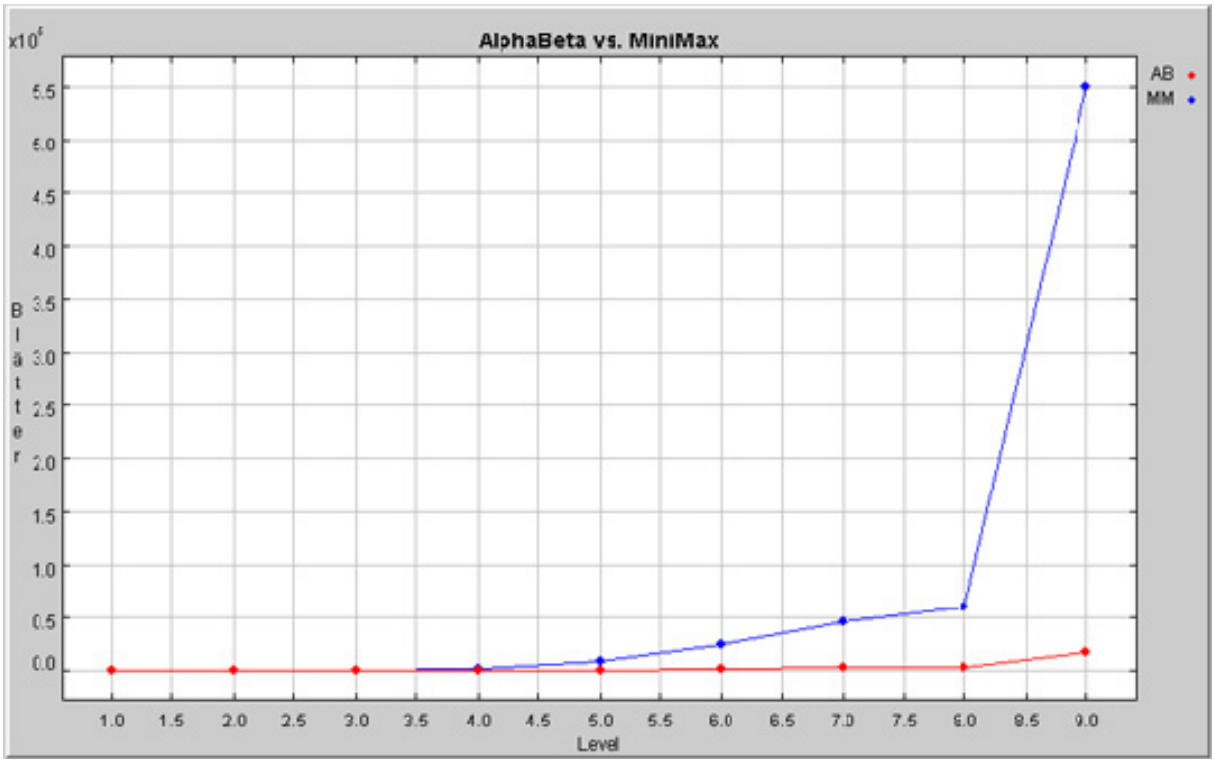


Abb. 12: Graphische Darstellung der Abhängigkeit der besuchten Blätter von der Suchtiefe

Dieser Laufzeitvergleich zeigt einen massiven Sprung vom achten zum neunten Level. Da die Komplexität bei einem Spiel mit höheren möglichen Baumtiefen (z.B. Gale oder Schach) mit jedem Level mindestens gleich große Sprünge macht, wird die Notwendigkeit des Alpha-Beta-Algorithmus in der Spielzuganalyse deutlich.

Primärliteratur

Russell, S. and Norvig, P. (1995). Artificial Intelligence: A Modern Approach. Prentice-Hall Inc., Englewood Cliffs, NJ, USA.

Knuth, D.E. and Moore, R.W. (1975). An Analysis of Alpha-beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293-326.

Sekundärliteratur

Akl, S.G. and Newborn, M.M. (1977). The Principal Continuation and the Killer Heuristic. *1977 ACM Annual Conference Proceedings*, pp. 466-473. ACM, Seattle.

Allis, L.V., Herik, H.J. van den and Herschberg, I.S. (1991). Which Games Will Survive? *Heuristic Programming in Artificial Intelligence 2: the second computer olympiad* (eds. D.N.L. Levy and D.F. Beal), pp. 232-243. Ellis Horwood, Chichester.

Allis, L.V. (1994). Searching for Solutions in Games and Artificial Intelligence. Ph.D. thesis, Rijksuniversiteit Limburg, Maastricht, The Netherlands.

Anshelevich, V.V. (2000). The Game of Hex: An Automatic Theorem Proving Approach to Game Programming. To appear in the Proceedings of *AAAI-2000*.

Blixen, C. von (2000). Lines-of-Action. *http://www.student.nada.kth.se/~f89-cvb/loa.html*

Bouton, C.L. (1901). Nim, a Game with a Complete Mathematical Theory. *Annals of Mathematics*, Vol. 2, No.3, pp. 33-39.

Breuker, D.M., Uiterwijk, J.W.H.M. and Herik, H.J. van den (1994). Replacement Schemes for Transposition Tables. *ICCA Journal*, Vol. 17, No. 4, pp. 183-193.

Breuker, D.M. (1998). Memory versus Search in Games. Ph.D. thesis. Universiteit Maastricht, Maastricht, The Netherlands.

Brudno, A.L. (1963). Bounds and Valuations for Abridging the Search of Estimates. *Problems of Cybernetics*. Vol. 10, pp. 225-241.

Donninger, C. (1993). Null Move and Deep Search: Selective-Search Heuristics for Obtuse Chess Programs. *ICCA Journal*, Vol. 16, No. 3, pp. 137-143.

Dyer, D. (2000). Lines of Action Homepage. *http://www.andromeda.com/people/ddyer/loa/loa.html*

Gray, S.B. (1971). Local Properties of Binary Images in Two Dimensions. *IEEE Transactions on Computers,* Vol. C-20, No. 5, pp. 551-561.

Groot, A.D. de (1946). Het Denken van den Schaker, een Experimenteel-psychologische Studie. Ph.D. thesis, University of Amsterdam, Amsterdam, The Netherlands. In Dutch.

Handscomb, K. (2000a) Lines of Action Strategic Ideas – Part 1. *Abstract Games.* Vol. 1, No. 1.

Handscomb, K. (2000b) Lines of Action Strategic Ideas – Part 2. *Abstract Games.* Vol. 1, No. 2.

Hartmann, D. (1988). Butterfly Boards. *ICCA Journal*, Vol. 11, Nos. 2/3, pp. 64-71.

Herik, H.J. van den (1983). Computerschaak, Schaakwereld en Kunstmatige Intelligentie. Ph.D. thesis, Delft University of Technology. Academic Service, Den Haag, The Netherlands. In Dutch.

Herik, H.J. van den and Herschberg, I.S. (1985). The Construction of an Omniscient Endgame Data Base. *ICCA Journal*, Vol. 8, No. 2, pp. 66-87.

Huberman, B.J. (1968). A Program to Play Chess End Games. *Technical Report no. CS-106.* Ph.D. thesis. Stanford University, Computer Science Department, USA.

Maltell, T. (2000). Renju International Federation. *http://www.lemes.se/renju/*

Marsland, T.A. (1986). A Review of Game-Tree Pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3-19.

Minsky, M. (1968). Semantic Information Processing. M.I.T. Press, Cambridge, MA, USA.

Minsky, M. and Papert, S. (1988). Perceptrons: An Introduction to Computational Geometry. M.I.T. Press, Cambridge, MA, USA.

Newell, A. and Simon, H.A. (1972). Human Problem Solving. Prentice-Hall Inc., Englewood Cliffs, NY, USA.

Nilsson, N.J. (1971). Problem-Solving Methods in Artificial Intelligence. McGraw-Hill Book Company, New York, NY, USA.

Sackson, S. (1969). A Gamut of Games. Random House, New York, NY, USA.

Samuel, A.L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, Vol. 3, No. 3, pp. 210-229.

Schaeffer, J. (1983). The History Heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16-19.

Schaeffer, J. and Lake, R. (1996). Solving the Game of Checkers. *Games of No Chance* (ed. J. Nowakowski). *MSRI Publications,* Vol. 29*,* pp. 119-133. Cambridge University Press, Cambridge, UK.

Schaeffer, J. (1997). One Jump Ahead: Challenging Human Supremacy in Checkers. Springer-Verlag, New York NY, USA.

Schaeffer, J. and Plaat, A. (1997). Kasparov versus Deep Blue: The Rematch. *ICCA Journal*, Vol. 20, No.2, pp. 95-101.

Shannon, C.E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, Vol. 41, No. 7, pp. 256-275.

Schrüfer, G. (1989). A Strategic Quiescence Search. *ICCA Journal*, Vol. 12, No. 1, pp. 3-9.

Tesauro, G. (1994). TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural Computation*, Vol. 6, No. 2, pp. 215-219.

Thompson, M. (2000). TwixT. *http://home.flash.net/~markthom/html/twixt.html*

Turing, A.M. (1953). Digital Computers Applied to Games. *Faster than Thought* (ed. B.V. Bowden), pp. 286-297. Pitman, London, UK.

Uiterwijk, J.W.H.M. (1992). The Countermove Heuristic. *ICCA Journal*, Vol. 15, No. 1, pp. 8-15.

Uiterwijk, J.W.H.M. (1995). Déjà vu. *Computerschaak*, Vol.15, No. 2, pp. 84-91. In Dutch.

Uiterwijk, J.W.H.M. and Herik, H.J. van den (2000). The Advantage of the Initiative. *Information Sciences*, Vol. 122, No. 1, pp. 43-58.

Von Neumann, J. and Morgenstern, O. (1944). Theory of Games and Economic Behavior. Princeton University Press, Princeton, NJ, USA.

Zobrist, A.L. (1970). A New Hashing Method with Application for Game Playing. Technical Report 88, Computer Science Department, The University of Wisconsin, Madison, WI, USA. Reprinted (1990) in *ICCA Journal*, Vol.13, No. 2, pp. 69-73.

Verwendete Literatur (Internet-Recherche)

Game playing with Minimax and Pruning, *http://www.cogs.susx.ac.uk/users/christ/crs/kr/lec05.html*

Decision Trees, *http://www.aaai.org/AITopics/html/trees.html*

Game Playing: Alpha-Beta-Pruning,
http://sern.ucalgary.ca/courses/CPSC/533/W99/presentations/L2_5B_Lima_Neitz/search.html

Parallele Alpha-Beta-Suche auf Spielbäumen,
http://www.thi.informatik.uni-frankfurt.de/~gramlich/Publications/alphabeta.ps.gz

Aleksandar Trifunovic. A parallel Chess Computer,
http://www.doc.ic.ac.uk/lab/labsrc_area/msc/MScProjects01-02/Conv/KKXKKat701.pdf

Prof. Dr. M. Gerndt, Dr. J. Weidendorfer, Dr. M. Walter (2005). Hochleistungsarchitekturen - Minimax und Alpha Beta Suche,
http://www.lrr.in.tum.de/~gerndt/home/Teaching/SS2005/Praktikum/Abalone/main.6.pdf

Yosen Lin (2003). Games Trees,
http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html

Bernhard Pfahringer. AI Techniques and Applications,
http://www.cs.waikato.ac.nz/~bernhard/316/

Minimax and Alpha-Beta Pruning,
http://students.cs.byu.edu/~cs670ta/Lectures/Minimax.html

Verwendeter SourceCode (Internet-Recherche)

Raul, Chess Game, Truman State University, Misouri, USA,
http://www2.truman.edu/~d1577/Chess/src/

Mark Watson (2005). Practical Artificial Intelligence Programming in Java,
http://www.markwatson.com/opencontent/