Der Alpha-Beta-Algorithmus

Maria Hartmann

19. Mai 2017

1 Einführung

Wir wollen für bestimmte Spiele algorithmisch die optimale Spielstrategie finden, also die Strategie, die für den betrachteten Spieler zum optimalen Ergebnis führt. Wir betrachten Spiele mit folgenden Eigenschaften:

(i) Es spielen genau zwei Spieler gegeneinander.

(ii) Das Spiel ist ein Nullsummenspiel.

(iii) Es ist ein Spiel mit perfekter und vollkommener Information.

Dazu zunächst einige Definitionen der relevanten Begriffe:

Definition 1.1. Nullsummenspiel
Ein Spiel ist ein Nullsummenspiel, wenn die Summe aus Gewinn und Verlust aller Spieler nach Spielende Null ergibt, das heißt der Gewinn eines Spielers ist der Verlust bzw. die Niederlage eines anderen.

Definition 1.2. perfekte Information
Ein Spiel ist ein Spiel mit perfekter Information, wenn allen Spielern das vorangegangene Spielgeschehen vollständig bekannt ist.

Definition 1.3. vollkommene Information
Ein Spiel ist eines mit vollkommener Information, wenn allen beteiligten Spieler alle Regeln und möglichen Spielzüge bekannt sind.

Bekannte Beispiele für solche Spiele sind z.B. Schach oder Tic Tac Toe.

2 Spielbäume

Alle möglichen Abläufe solcher Spiele können mithilfe von Spielbäumen dargestellt werden. Jeder Knoten repräsentiert dabei einen möglichen Spielzustand. Ein Spielzustand ist definiert durch folgende Informationen:

(i) Spieler am Zug. Zur visuellen Vereinfachung wird bei zwei Spielern für einen Spieler der Knoten kreisförmig gezeichnet und für den anderen quadratisch.

(ii) Aktueller Spielstand, z.B. Anordnung der Figuren auf dem Schachbrett.

(iii) Erlaubte Spielzüge aus dem aktuellen Zustand heraus. Jeder dieser Spielzüge ist ein Kind des betrachteten Knotens im Spielbaum. Wenn ein Spieler im aktuellen Spielstand gewonnen hat, ist das Spiel beendet und es gibt keine weiteren Züge.

In einem vollständig konstruierten Spielbaum, in dem alle möglichen Züge dargestellt sind, ist jeder Pfad von der Wurzel zu einem Blatt ein möglicher Spielverlauf.

Beispiel 2.1:
Als Beispiel betrachten wir einen Ausschnitt aus einem Spielbaum für das Spiel Tic Tac Toe. Der aktuelle Spielstand wird durch Abbildung des Spielbretts abgebildet; der aktuelle Spieler am Zug ist hier nur implizit erkennbar.

3 Minimax-Algorithmus

3.1 Vorbetrachtungen

In dem so konstruierten Spielbaum wollen wir nun den für unseren Spieler optimalen Pfad finden. Das geschieht mithilfe des *Minimax-Algorithmus*. Als Entscheidungsbasis wird dabei jedem erreichbaren Spielzustand ein Zahlenwert zugeordnet, der umso höher ist, je besser der Spielzustand für unseren Spieler ist, und entsprechend niedriger für für uns schlechte und damit für den Gegner gute Spielzustände. Die tatsächliche Gewichtung der Spielstände hängt dabei vom jeweiligen betrachteten Spiel ab. Zustände, in denen der betrachtete Spieler gewinnt, sollen dabei offensichtlich den höchsten Wert haben und solche, in denen er verliert den niedrigsten. Mit der festgelegten Gewichtung sind dann die Werte aller Endzustände (aller Blätter im Spielbaum) eindeutig bestimmt. Von den Blättern aus aufsteigend werden dann die Gewichte der inneren Knoten abgeleitet. Dazu zunächst folgende Überlegungen:

- Unser Spieler versucht offensichtlich, den Spielausgang mit dem höchstmöglichen Wert zu erzielen. Er strebt den *maximalen* Wert an; daher wollen wir ihn als **max-Spieler** bezeichnen.
- Unter der Annahme, dass der Gegner ebenfalls perfekt spielt, wird er hingegen in jedem Zug den Zug mit dem am niedrigsten gewichteten Spielausgang wählen. Er wählt also den *minimalen* Zug, daher nennen wir ihn **min-Spieler**.

Auf dieser Basis wird das Gewicht eines inneren Knotens v des Spielbaums also rekursiv so bestimmt:

$$w(v) = \begin{cases} min(children(v)), & \text{falls Gegner am Zug} \\ max(children(v)), & \text{falls Spieler am Zug} \end{cases}$$

3.2 Pseudocode

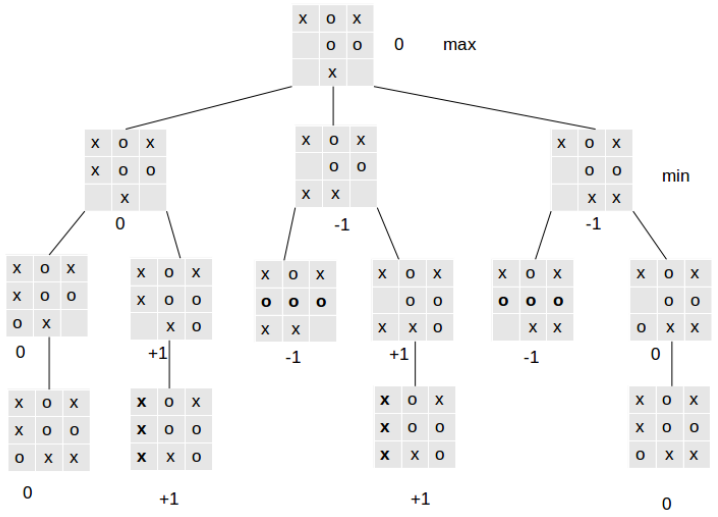
```
function minimax(node v);
if v is leaf then
    return w(v);
else if v is max-node then
    n=-∞;
    foreach child u of v do
        n=max(n, minimax(u))
    end
    return n;
else if v is min-node then
    n=+∞;
    foreach child u of v do
        n=min(n, minimax(u))
    end
    return n;
```

Algorithm 1: der Minimax-Algorithmus

3.3 Ein Beispiel

Beispiel 3.1:
Als Beispiel betrachten wir wieder das Spiel Tic Tac Toe. Die Gewichtung der Spielstände ist in diesem Fall sehr einfach; die einzigen für uns relevanten Ergebnisse sind Sieg, Niederlage und Unentschieden, ohne dass uns die weitere Anordnung der Spielsteine interessiert. Daher definieren wir die Gewichtung der Blätter wie folgt:

unser Spieler gewinnt →	$+1$
unentschieden →	0
unser Spieler verliert →	-1



3.4 Laufzeitanalyse

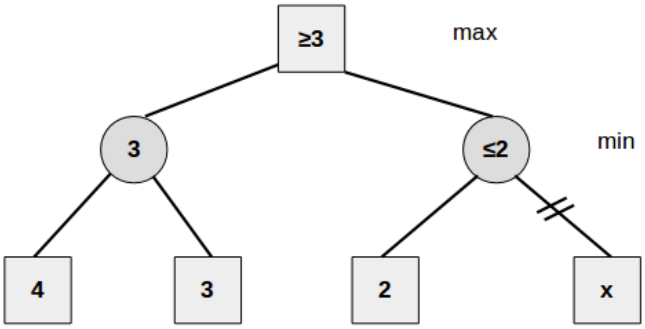
Der Minimax-Algorithmus durchläuft in jedem Fall alle Zweige. Im ungünstigsten Fall enden alle möglichen Spielverläufe erst nach d Zügen (d entspricht dann der Tiefe aller Blätter bzw. der Höhe des Baumes). Nehmen wir an, dass aus jeder Spielkonfiguration heraus immer genau w Züge möglich sind. Im resultierenden Spielbaum gibt es dann w Knoten mit der Tiefe 1, $w * w$ Knoten mit der Tiefe 2 etc. bis zu w^d Blättern in der untersten Ebene. Um alle Knoten zu durchlaufen, sind dann $T = \sum_{i=1}^{d} w^d$ Aufrufe des Algorithmus notwendig. Es gilt $\sum_{i=1}^{d} w^d \in O(w^d)$.
Für Spielbäume mit einem nichtkonstanten Verzweigungsfaktor wird als Approximation der Durchschnitt aller Verzweigungsfaktoren gebildet.

4 Verbesserung: Alpha-Beta-Algorithmus

4.1 Vorbetrachtungen

Es ist leicht erkennbar, dass der Minimax-Algorithmus zwar immer zu einem Ergebnis führt, aber dies bei umfangreicheren Spielen mit einem extrem hohen Rechenaufwand verbunden ist. Eine Analyse der Komplexität des Algorithmus zeigt, dass er in $O(w^d)$ liegt, wobei w der durchschnittliche Verzeigungsfaktor und d die Tiefe des Spielbaumes ist. Das ist zum Beispiel auch bei Schachcomputern die entscheidende Einschränkung, die die Anzahl der Züge, die vorausberechnet werden können, beschränkt. Daher wird der *Alpha-Beta-Algorithmus*, eine Modifikation des Minimax-Algorithmus, verwendet, um die Berechnung effizienter zu gestalten. Dieser Algorithmus schätzt den Wert der möglichen Endzustände auf dem betrachteten Zweig des Spielbaumes ab und bricht die Evaluation des Zweiges ab, sobald absehbar wird, dass er keinen Spielvorteil bietet, ohne diese unnötig zu Ende zu führen. Dazu folgende Überlegungen:

- Es müssen nicht notwendigerweise alle Zweige durchlaufen werden, um das beste Ergebnis zu erzielen. Gegebenenfalls können Zweige vernachlässigt werden.
- Die Auswahl des optimalen Zugs erfolgt beim Minimax-Algorithmus mittels *Tiefensuche*, d.h. von links nach rechts werden alle Zweige komplett durchlaufen. Während dieser Durchläufe ist der bisher beste gefundene Zug (der mit dem maximalen bzw. minimalen Wert) bekannt. Dazu betrachten wir folgendes Beispiel:



Wenn der min-Knoten im von der Wurzel aus rechten Teilbaum durchsucht wird, wissen wir bereits aus der Durchsuchung des linken Teilbaums, dass der Wurzelknoten **mindestens** den Wert 3 hat, auf keinen Fall aber kleiner (es ist ein max-Knoten). Daher kann die Evaluation des rechten Teilbaums abgebrochen werden, sobald erkannt wird, dass dessen min-Knoten einen Wert von maximal 2 haben wird, also für den Elternknoten nicht mehr relevant ist. Der tatsächliche Wert von x ist dabei unwichtig; der Zweig wird *weggeschnitten*.

Diese Beobachtungen macht sich der Alpha-Beta-Algorithmus zunutze. Der Algorithmus ist eine erweiterte Form des Minimax-Algorithmus, bei der zusätzlich als Parameter noch ein Intervall (α, β) übergeben wird, innerhalb dessen der exakte Wert des betrachteten Knoten von Interesse ist. α steht dabei für den größten bisher gefundenen Wert, den ein max-Node sicher hat, und β entsprechend für den kleinsten bisher gefundenen Wert für einen min-Node. Wird wie im Beispiel ein Wert außerhalb des Intervalls gefunden, kann dieser keinen Einfluss mehr auf den Endwert des Elternknotens haben und die Betrachtung des Zweiges kann beendet werden.

4.2 Pseudocode

Formalisiert sieht der Algorithmus dann so aus:

```
function alphabeta(node v, α,β);
if v is leaf then
    return w(v);
else if v is max-node then
    n=-∞;
    foreach child u of v do
        α=max(α, alphabeta(u,α,β));
        if α ≥ β then
            return α;
        end
    end
    return n;
else if v is min-node then
    n=+∞;
    foreach child u of v do
        β=min(β, alphabeta(u,α,β));
        if α ≥ β then
            return β;
        end
    end
    return n;
```

Algorithm 2: der Alpha-Beta-Algorithmus

4.3 Laufzeitanalyse

Es gibt Fälle, in denen der Alpha-Beta-Algorithmus keinen Zweig wegschneiden kann, nämlich wenn die Zweige nach Werten aufsteigend geordnet sind. In diesem (schlechtesten) Fall handelt der Algorithmus genau wie der Minimax-Algorithmus und hat daher dieselbe worst case-Komplexität $O(w^d)$.
Interessanter sind jedoch die Betrachtung des best und average case: Im besten Fall sind die Spielzüge absteigend geordnet, so dass der beste Zug der erste evaluierte Zweig ist (umgekehrt zum worst case).
Das hat zur Folge, dass in jedem betrachteten Unterbaum der erste Zweig von links den besten Wert liefert. Bei den betrachteten Knoten des Spielers, der als zweites gezogen hat, können dann im Regelfall alle weiteren möglichen Züge verworfen werden. Bei allen Zügen des ersten Spielers in einem linken Teilbaum müssen jedoch alle w möglichen Spielzüge betrachtet werden, da das Intervall (α, β) hier immer auf einer Seite durch $\pm\infty$ begrenzt ist, was nicht zu einem vorzeitigen Abbruch durch $\alpha \geq \beta$ führen kann. Es müssen also mindestens $(1 * w * 1 * w * ... * 1 * w) = w^{d/2}$ bei Bäumen gerader Höhe und $(1 * w * 1 * w * ... * 1) = w^{(d-1)/2}$ bei Bäumen ungerader Höhe betrachtet werden. Das liegt in $O(w^{\lfloor d/2 \rfloor})$.
Berechnungen ergeben für den average case eine Durchschnittslaufzeit in $O(w^{3d/4}$. Besonders bei umfangreicheren Berechnungen, z.B. beim Schach, bringt diese Verbesserung einen signifikanten Vorteil!