Spielbäume

Proseminar Theoretische Informatik

Joschka Heinrich*, TU Dresden

31. März 2018

I. Einführung

Was sind Spielbäume und wozu können sie verwendet werden? In dieser Ausarbeitung wird das Konzept des Spielbaumes formal eingeführt und anhand des Spiels *Tic-Tac-Toe* beispielhaft erläutert. Mit der Erklärung des Alpha-Beta-Prunings als Optimierung des Minimax-Verfahrens wird eine Anwendung von Spielbäumen illustriert.

II. Spielbäume

Um im Folgenden mit Spielbäumen arbeiten zu können, führen wir das Konzept zunächst formal ein und definieren dazu unter anderem den Begriff des *Spiels*, der *Konfiguration* und des *Spielbaums*.

i. Definition

Alle folgenden Betrachtungen nehmen wir aus der Perspektive eines gewinnorientierten Spielers namens Max vor, der sich einer Anzahl Gegner gegenüber sieht. Es ist also das Ziel, Züge für Max so zu finden, dass dessen Gewinn maximiert, bzw. der Gewinn der Gegner minimiert wird. Wir gehen dabei davon aus, dass auch alle Gegner optimale Entscheidungen treffen. Wir vereinfachen die Betrachtung von Spielen, indem wir uns auf solche ohne Zufallskomponente, d.h. reine Strategiespiele mit vollkommener Information und Spiele mit zwei Kontrahenten – Max und ein zweiter Spieler Min – beschränken. Mit „der Gegner“ ist also im Folgenden stets Min gemeint.

Ein **Spiel** $S = (R, k_0, F)$ ist nun durch Regeln, in Form einer endlichen Menge R von legalen Spielzügen (als Funktionen auf den Konfigurationen), eine Anfangskonfiguration $k_0 \in K$ und eine Reihe möglicher Endkonfigurationen $F \subset K$ gegeben, mit K, der Menge aller *zulässigen* Konfigurationen. Eine **Konfiguration** $k \in K$ repräsentiert dabei einen möglichen Zustand des Spieles, bestehend aus einer Beschreibung wiederum der Zustände aller relevanten Spielelemente

*joschka.heinrich@tu-dresden.de, PGP: B40E 67C7 FF62 C860 7854 A778 6FB9 666F 1147 A401

(bspw. die Position der Zeichen auf dem Tic-Tac-Toe-Feld) inklusive des Spielers, der als nächster an der Reihe ist (bei uns entweder Max oder Min).

In Abgrenzung zur Menge der *legalen* Spielzüge können wir uns beliebige andere Spielzüge vorstellen, die zwar möglich, allerdings in dem betrachteten Spiel nicht erlaubt sind. Analog dazu sind über K hinaus weitere Konfigurationen denkbar, die allerdings nicht zulässig sind, d.h. in einem regelkonformen Spiel niemals auftreten können. Dieser Zusammenhang wird in Abb. 1 veranschaulicht.

Durch Anwenden eines legalen Spielzuges auf eine Konfiguration gelangen wir zu einer neuen Konfiguration. Ein legaler Spielzug kann also als eine Funktionen $r : K \to K$ verstanden werden. Seien $u, v \in K$ Konfigurationen und $r \in R$ ein legaler Spielzug, mit $v = r(u)$, dann heißt v **Kindkonfiguration** von u (bezüglich r) und u **Elternkonfiguration** von v (bezüglich r).

Wenn das Anwenden eines Spielzuges zu einer neuen Konfiguration führt, also $u \neq v$ gilt, dann heißt r **anwendbar** auf u.[1] Die Teilmenge von R, der auf eine Konfiguration $k \in K$ anwendbare Spielzüge, bezeichnen wir mit R_k. Wir erhalten daraus die **Menge der Kindkonfigurationen** $N(k) \subset K$ mit $N(k) = \{r(k) \mid r \in R_k\}$. Auf eine Endkonfiguration $k_f \in F$ sind keine Spielzüge anwendbar, da das Spiel mit Erreichen einer dieser Konfigurationen als beendet gilt: $N(k_f) = \varnothing$, da $R_{k_f} = \varnothing$

Die Menge K definieren wir nun induktiv über die Kindkonfiguration:

- k_0 ist Element von K.
- Wenn $k \in K$, dann auch alle $k' \in N(k)$.

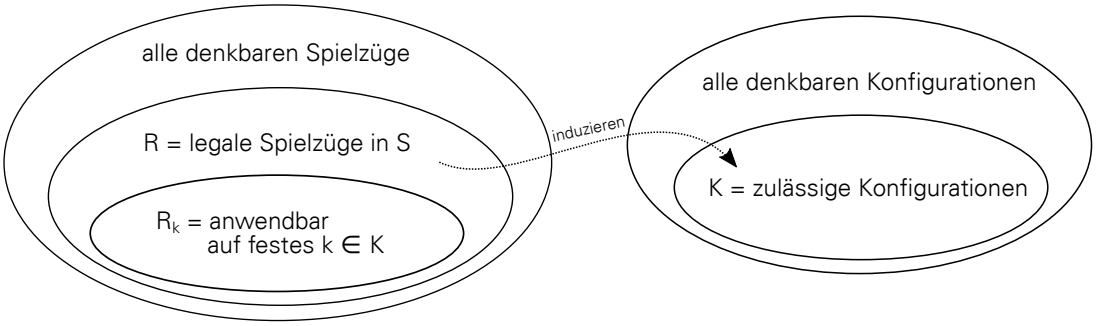

Abbildung 1: Dieses Mengendiagramm zeigt die Teilmengenbeziehungen der Spielzüge einerseits und der Konfigurationen andererseits, wobei K induktiv über R definiert ist.

Alle zulässigen Konfigurationen lassen sich also aus der Anfangskonfiguration und den legalen Spielzügen ableiten. Zu jeder Konfiguration $k \in K \quad F$ gehört eindeutig eine Menge von Kindkonfigurationen $N(k)$ und zu jeder Konfiguration $k \in K \quad k_0$ eine Elternkonfiguration k'. Diese Beziehungen können durch einen Graphen anschaulich dargestellt werden.

Ein **Spielgraph** ist ein gerichteter Graph $G(V, E)$ mit:

- Knoten $V = K$ und
- Kanten $E = \bigcup_{u \in K} \{(u, v) \mid v \in N(u)\}$

Zusätzliche soll eine über R aus k_0 generierte Konfiguration durch wiederholtes Anwenden legaler Spielzüge nicht erneut erreicht werden können. Wir gelangen also im weiteren Spielverlauf

[1] Das bedeutet nicht notwendigerweise, dass sich die Konfiguration der Spielelemente verändert. Zwei Spielkonfigurationen können sich auch darin unterscheiden, welcher Spieler an der Reihe ist.

nie zu einer Situation, die bereits aufgetreten ist. Dies können wir allerdings leicht erreichen, indem wir bspw. die Zugnummer oder sogar alle vorherigen Konfigurationen mit in die Konfigurationsbeschreibung aufnehmen. So werden Zyklen im Graphen aufgelöst.

Es folgt damit insbesondere, dass der Graph G zyklenfrei, also ein Baum ist. Im **Spielbaum**[2], mit k_0 als Wurzel und F als Blätter, kann so bspw. ein Spielverlauf mit n Zügen als Pfad $(k_0, k_1, \dots, k_n \in F)$ aufeinander folgender Kindkonfigurationen dargestellt werden.

ii. Am Beispiel Tic-Tac-Toe

Um obige Definitionen zu veranschaulichen, wenden wir sie nun auf das Spiel Tic-Tac-Toe an. Das Tic-Tac-Toe-Spielfeld besteht aus einem 3×3-Raster mit neun Feldern, in die zwei Spieler abwechselnd ihre Zeichen setzen. Dies sind üblicherweise „X“ bzw. „O“. Im Folgenden wird Max mit „X“ und Min mit „O“ spielen.

Ziel jedes Spielers ist es, drei der eigenen Zeichen nebeneinander zu setzen, d.h. in einer Reihe, Spalte oder Diagonale, und gleichzeitig zu verhindern, dass der Gegner seine Zeichen in dieser Weise setzen kann. Das Spiel endet entweder, wenn einer der beiden Spieler gewinnt, sobald er dieses Ziel erreicht, oder wenn kein Zug mehr möglich ist, da alle neun Felder belegt sind. Das Spiel geht in diesem Fall unentschieden aus.

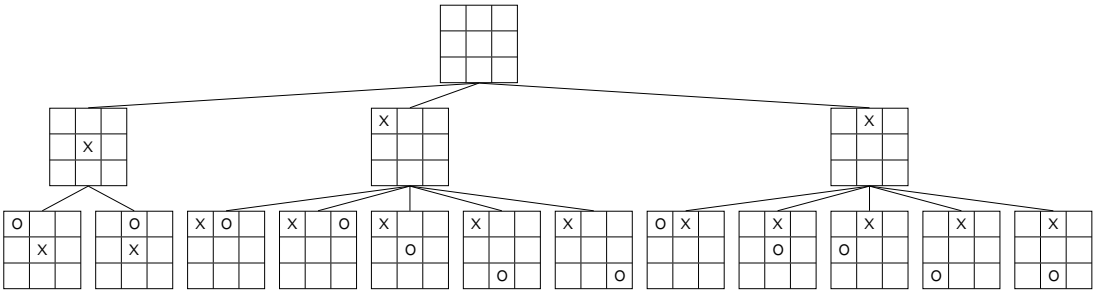

Abbildung 2: Ein Tic-Tac-Toe-Spielbaum der Tiefe 2, bei dem die Anzahl der Konfigurationen bereits durch Ausnutzung von Symmetrien optimiert wurde.

III. Zugplanung

Es stellt sich nun die Frage, wie sich aus den bekannten möglichen Zügen, die sich aus einer Spielsituation ergeben, der beste Zug auswählen lässt. Max wählt seine Züge stets so, dass sein Vorteil maximiert wird, wenn er am Zug ist. Da angenommen wird, dass Min genauso handelt, heißt das auch, dass er jenen von Mins Zügen annehmen muss, der ihm den größten Nachteil bringen wird.[3]

[2] Wir benutzen also im Folgenden – abweichend der Terminologie der Hauptquelle[1] – , mit den hier getroffenen Annahmen, „Spielgraph“ synonym zu „Spielbaum“. Davon abzugrenzen ist der Begriff „Suchbaum“. Während der *Spielbaum* ein theoretisches Modell von großer (Speicher-)Komplexität ist, wird der *Suchbaum* zur Laufzeit generiert und bildet kein vollständiges Spiel ab, ist aber gerade dadurch ebenfalls garantiert zyklenfrei.

[3] In Anlehnung an [2] tragen Max und Min auch genau aus diesem Grund ihre Namen: ist Max an der Reihe, wird der Nutzen *maximiert*; ist es Min, wird er *minimiert*; stets aus der Sicht von Max.

Dieses Vorgehen führt zum **Minimax-Verfahren**[4], das wir im Folgenden genauer betrachten. Wir werden einige seiner Optimierungen kennenlernen, um auch unvollständige Echtzeitentscheidungen treffen zu können.

i. Minimax-Verfahren

Zunächst führen wir eine **Gewinnfunktion** $g : F \to \mathbb{N}$ ein, die die Blätter des Spielbaumes bewertet und ihnen einen Wert zuordnet, wie günstig dieser Ausgang des Spiels für Max wäre.

Des Weiteren benötigen wir zwei Typen von Knoten in unserem Spielbaum: **Max-Knoten** (im Folgenden mit $\triangle$ gekennzeichnet), an denen Max am Zug ist und der Nutzen maximiert wird und analog dazu **Min-Knoten** ($\triangledown$). In unserem Tic-Tac-Toe-Szenario alternieren die Knoten-Typen mit jedem Halbzug[5], da die Spieler sich stets abwechseln.

Jedem Knoten u im Spielbaum wird nun ein **Minimax-Wert** $minimax(u) \in \mathbb{N}$, der dem Nutzen aus Sicht von Max entspricht, zugeordnet. Dabei ergibt sich $minimax(u)$ rekursiv aus den Minimax-Werten der Kindknoten $N(u)$ und wird mit einer Tiefensuche[6] wie folgt rekursiv berechnet:

$$minimax(u) = \begin{cases} g(u) & \text{wenn } u \in F \\ \max\limits_{v \in N(u)} minimax(v) & \text{wenn } u \text{ Max-Knoten} \\ \min\limits_{v \in N(u)} minimax(v) & \text{wenn } u \text{ Min-Knoten} \end{cases}$$

Die Rekursion endet mit dem Erreichen einer Endkonfiguration, dann wird die Gewinnfunktion angewendet. Je nachdem, in welcher Ebene (Min oder Max) des Baumes wir uns befinden, wird danach beim rekursiven Aufstieg entsprechend das Minimum bzw. Maximum der Kindknoten zum Elternknoten weitergegeben. Schließlich wählt Max den Zug mit dem größten Minimax-Wert als seinen nächsten Zug aus. (vgl. Abb. 3)

Würden wir tatsächlich den gesamten Spielbaum mit diesem Verfahren durchsuchen, ergibt sich mit der maximalen Tiefe m des Baumes und einem Verzweigungsgrad b eine Zeitkomplexität von $\mathcal{O}(b^m)$ sowie eine Speicherkomplexität von $\mathcal{O}(bm)$. Die Speicherkomplexität beträgt lediglich $\mathcal{O}(m)$, wenn nur der aktuelle Pfad im Speicher gehalten wird.

Da schon bei wenig komplexen Spielen wie Tic-Tac-Toe, mit einer überschaubaren Zuganzahl, die Anzahl der möglichen Konfigurationen zu groß werden würde, um in annehmbarer Zeit Züge zu berechnen,[7] da der gesamte Baum durchlaufen werden muss, kann das Verfahren ohne Optimierungen nicht angewendet werden. Vor allem für wesentlich komplexere und längere Spiele, wie Schach, kommen daher unter anderem folgende Optimierungen zum Einsatz.

[4] erstmals 1912 von E. Zermelo erwähnt, 1913 veröffentlicht[3]

[5] Ein Zug entspricht einer Runde, in der sowohl Max und Min einmal an der Reihe waren und besteht aus zwei Halbzügen.

[6] Als zusätzliche Optimierung, die sich auch aus weiteren Gründen anbietet, wie wir sehen werden, bietet sich dafür in der Praxis die Verwendung der *iterativen Tiefensuche* an.

[7] Der Tic-Tac-Toe-Spielbaum hat ohne Optimierung $9! = 362880$ Knoten, obwohl es „nur“ 5478 verschiedene Konfigurationen gibt, die sich allerdings in insgesamt 255168 möglichen Spielverläufen wiederholen. (Ein Spiel kann auch schon mit weniger als 9 Zügen beendet werden.) Unter Ausnutzung von Symmetrien erhalten wir 765 Konfigurationen und 31896 Spielverläufe.

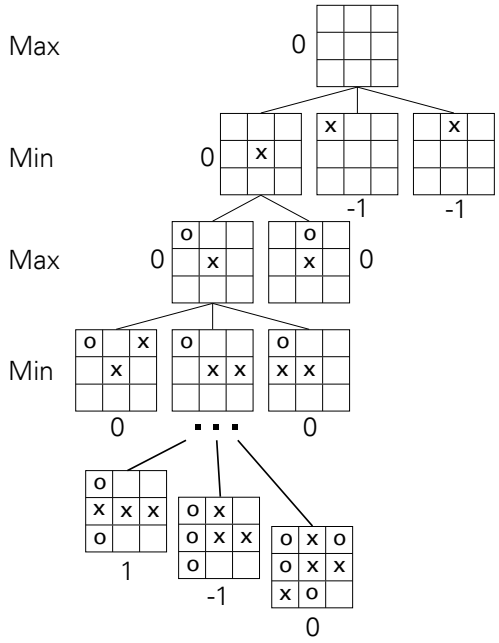

Abbildung 3: Ein Ausschnitt eines hypothetischen Spielbaumes, an dem das Prinzip des Minimax-Verfahrens nachvollzogen werden kann: eine Gewinnfunktion bewertet Endkonfigurationen mit 1 bei Gewinn, -1 bei Niederlage und 0 bei Unentschieden; die Minimax-Werte werden entsprechend des Algorithmus' bis zur Wurzel propagiert; Max wählt schließlich den ersten Zug mit dem Kreuz in der Mitte, da der Minimax-Wert hier am größten ist. Der Minimax-Wert ist nicht 1, da es keine Gewinnstrategie für Tic-Tac-Toe gibt. D.h. bei zwei rationalen Spielern wird stets ein Untenschieden erzielt werden.

ii. Optimierung mittels Heuristik

Damit nicht mehr der gesamte (und potenziell sehr tiefe) Teilbaum der Kindkonfigurationen rekursiv bis zu den Blättern durchsucht werden muss, um die Minimax-Bewertung eines Knotens k weiter oben im Baum zu erhalten, führen wir nun eine **Heuristik** ein. Diese ermöglicht es, auch Nicht-Endkonfigurationen einen *geschätzten* Nutzen zuzuordnen, basierend auf Merkmalen der Konfiguration. Der Spielbaum muss dann nur noch in einer festen Suchtiefe t analysiert werden.[8] Den Baum, den wir ausgehend von einer beliebigen Konfiguration des Spielbaumes in der Tiefe t zur Laufzeit effizient aufbauen, nennen wir **Suchbaum**.

Diese Heuristik $h(k)$ muss (1) die gleiche Ordnung wie die Gewinnfunktion $g(k)$ erzeugen, d.h. ebenso einen Gewinn besser als ein Unentschieden und dieses besser als eine Niederlage werten. Sie muss (2) effizient sein, sonst ist die Optimierung nicht sinnvoll, und muss (3) möglichst genau sein, d.h. Werte für Nicht-Endkonfigurationen schätzen, die möglichst nah am tatsächlichen Nutzen liegen.

[8] Bisher haben wir so lange Knoten expandiert (d.h. deren Kinder analysiert) bis eine Endkonfiguration erreicht war, da nur eine Gewinnfunktion zur Bewertung zur Verfügung stand. Da mit der Heuristik nun jede Konfiguration bewertet werden kann, ist dies nicht mehr nötig. Es stellt sich allerdings die Frage nach einem **Expansionskriterium**; d.h., wann es sinnvoll ist, einen Knoten noch weiter zu expandieren. Wir arbeiten hier weiter mit dem Kriterium einer festen Suchtiefe t, allerdings lassen sich weitere Expansionskriterien finden, die zu besseren Entscheidungen führen können. Die Suchtiefe kann auch über den Spielverlauf hinweg variiert werden.

Eine Heuristik für Tic-Tac-Toe könnte bspw. mit $h_1(k) = A_X(k) - A_O(k)$ definiert werden, wobei A_X bzw. A_O jeweils die Summe an möglichen Horizontalen, Vertikalen und Diagonalen bezeichnet, die mit drei der entsprechenden Zeichen („X“ oder „O“) vervollständigt werden könnten. (s. Abb. 4)

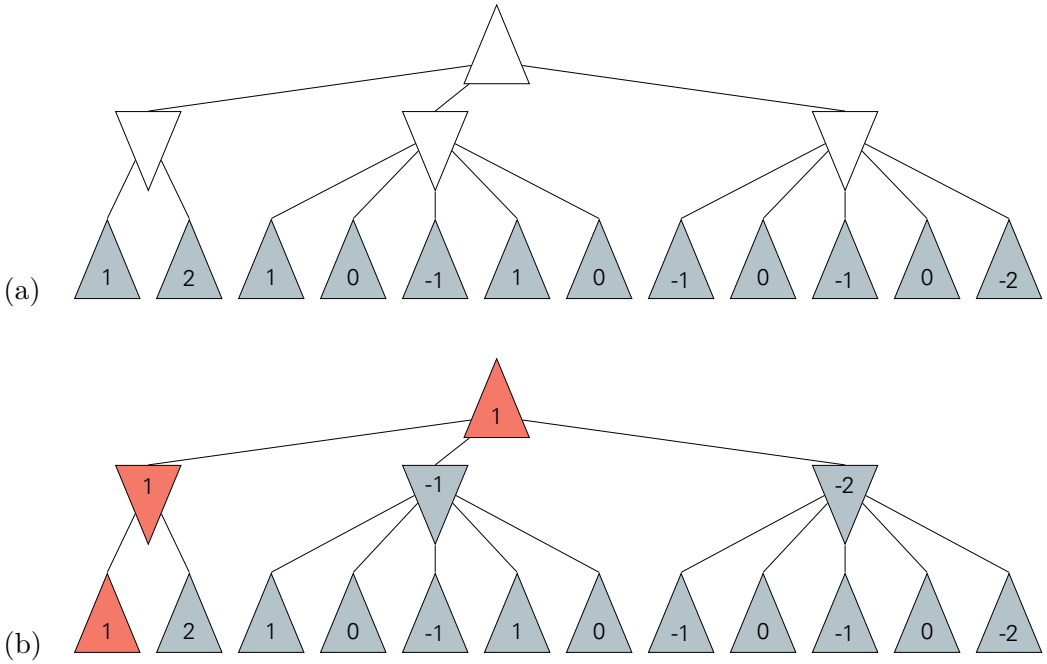

Abbildung 4: Wenden wir einen Suchbaum der Suchtiefe $t = 2$, ausgehend von der Startkonfiguration des leeren Feldes, aufbauen (vgl. Abb. 2) und dessen Blätter mit der Heuristik h_1 bewerten, erhalten wir die geschätzten Werte aus (a), die in (b) im gewohnten Modus zur Wurzel propagiert werden. Der Zug, den Max wählt, ist hervorgehoben.

iii. Alpha-Beta-Pruning

Um die Komplexität des Minimax-Algorithmus zu verringern, nehmen wir eine weitere Optimierung vor: das **Alpha-Beta-Pruning**[9]. Dieses erlaubt es, die Untersuchung von Teilbäumen abzukürzen, ohne dabei aussichtsreiche Züge auszuschließen, indem Knoten nicht weiter expandiert werden (auch, wenn das Expansionskriterium noch erfüllt wäre).

Dazu führen wir für jeden Knoten k zusätzlich einen **Alpha-** und **Beta-Wert** α_k und β_k ein. α_k ist dabei die *untere* Schranke des Minimax-Wertes für Max-Knoten und speichert die bisher *größte* Bewertung im Pfad von der Wurzel des aktuellen Suchbaumes zum Knoten k. Analog wird in β_k die *obere* Schranke für Min-Knoten gespeichert, d.h. die bisher *kleinste* Bewertung im Pfad zu k.

Die Alpha- und Beta-Werte werden mit $\alpha = -\infty$, $\beta = +\infty$ initialisiert und nach jeder untersuchten Konfiguration aktualisiert, sodass α an einem Max-Knoten stets maximiert und β an einem Min-Knoten stets minimiert wird.

Wir können nun auf Basis der Alpha- und Beta-Werte **Kürzungen (Cuts)** vornehmen, d.h. Teilbäume aus dem Suchbaum löschen. Diese können stets durchgeführt werden, sobald sich

[9] erstmals 1956 vorgestellt (McCarthy), auch „Alpha-Beta-Kürzung“, „α-β-Pruning“, „α-β-Cut“ oder „α-β-Suche“

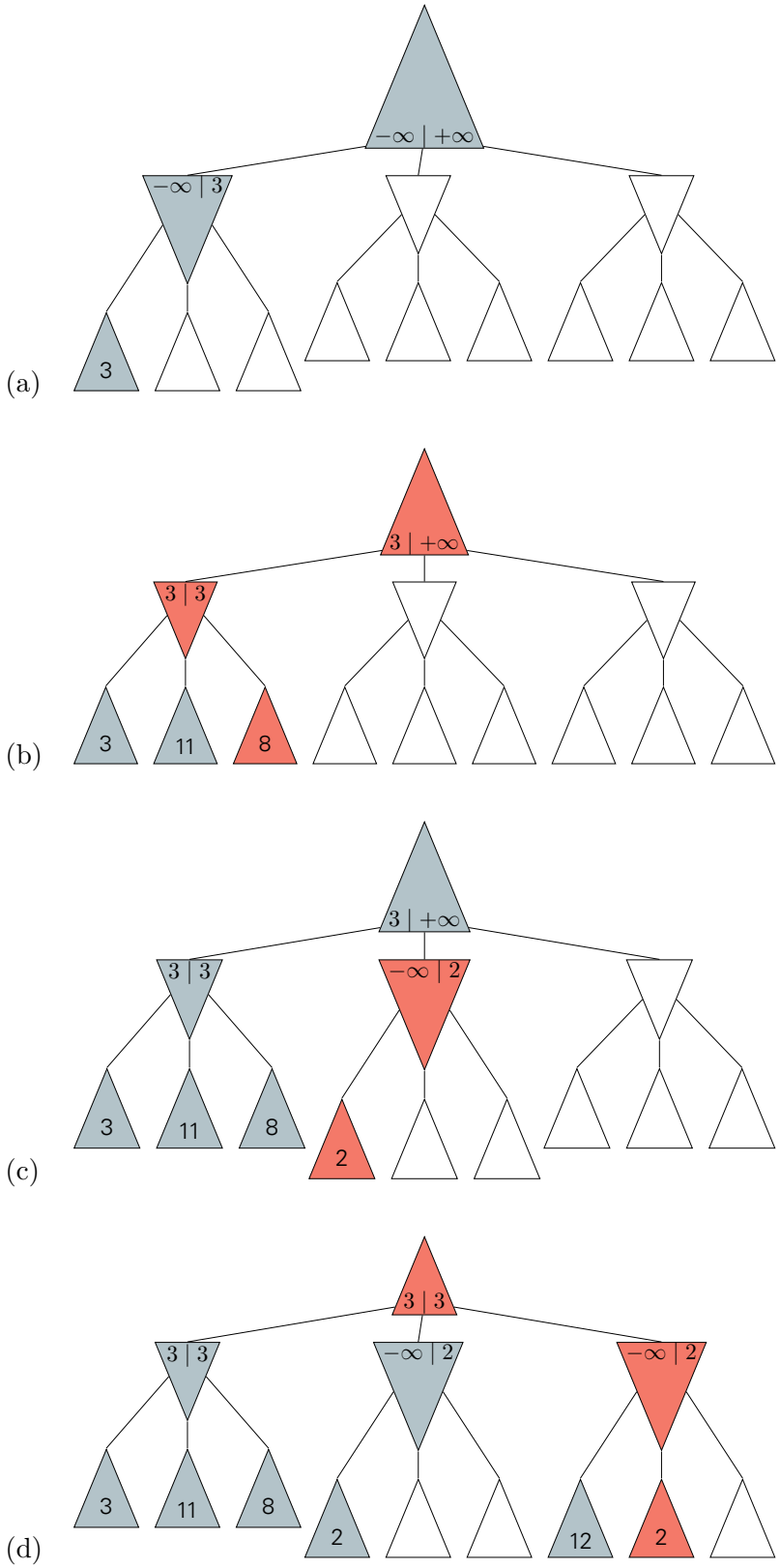

Abbildung 5: Schrittweiser Ablauf des Alpha-Beta-Prunings; links im Knoten ist der Alpha-, recht der Beta-Wert dargestellt.

die beiden Schranken überschneiden. An einem Max-Knoten u – es wird hier stets der größte Minimax-Wert unter den Kindknoten gesucht – führen wir den s.g. **Alpha-Cut** durch, sobald $\exists \beta_v : \beta_v \leq \alpha_u \ (v \in N(u))$. Analog dazu wird der **Beta-Cut** an einem Min-Knoten durchgeführt, wenn $\exists \alpha_v : \alpha_v \geq \beta_u \ (v \in N(u))$.

Dieses formal beschriebene Vorgehen wird durch folgende Überlegung zum Alpha-Cut intuitiver: Da auf einen Max-Knoten stets nur Min-Knoten in der nächsten Ebene des Suchbaumes folgen, müssen diese nicht weiter expandiert werden, wenn bekannt ist, dass ihr Minimax-Wert nie besser werden kann, als der eines bereits untersuchten Min-Knotens (und dessen Teilbaum). Dies ist immer dann der Fall, wenn *ein* Minimax-Wert gefunden wird, der den geringsten Wert eines anderen Min-Teilbaumes unterschreitet. Der Beta-Cut, kann analog-umgekehrt genauso verstanden werden.

In Abb. 5 kann das Alpha-Beta-Pruning nachvollzogen werden:

(a) Alpha- und Beta-Werte werden wie beschrieben initialisiert. Der erste Min-Knoten wird expandiert und eine Heuristik angewendet, die den Wert 3 für den ersten Blattknoten schätzt. Dieser Wert ist damit der Beta-Wert dieses Min-Knotens, da kein kleinerer Wert bisher gefunden wurde.

(b) Da die Bewertungen beider anderer Blattknoten größer als 3 ist, ändert sich β nicht, und auch α wird auf 3 gesetzt. Damit erhalten wir im Startknoten $\alpha = 3$, da wir mindestens diesen ersten Kindknoten wählen könnten. Es müssen im Folgenden nur noch Min-Knoten weiter untersucht werden, die einen größeren Wert annehmen können.

(c) Im zweiten Teilbaum wurde ein Blattknoten mit 2 bewertet. Das bedeutet, dass der Min-Knoten keinen Wert > 2 mehr liefern kann. Dies kann daran abgelesen werden, dass der Beta-Wert dieses Min-Knotens den Alpha-Wert, des darüber liegenden Knotens unterschreitet. Für die verbleibenden beiden Blattknoten (ggf. inklusive ihrer Teilbäume) wird jetzt ein Alpha-Cut durchgeführt und sie müssen nicht untersucht werden.

(d) Im letzten Teilbaum wiederholt sich dieser Fall, allerdings erst, sobald ein Wert $< \alpha$ geschätzt wird, also beim zweiten Blattknoten. Da nun alle Min-Knoten untersucht wurden, kann auch der Beta-Wert des Wurzelknotens auf 3 gesetzt werden. Letztendlich wurden in diesem Beispiel drei Kürzungen durchgeführt.

Die Zeitkomplexität sinkt mit dieser Optimierung im besten Fall, d.h. bei optimaler Sortierung der Kindknoten, auf $\mathcal{O}(b^{\frac{m}{2}})$. Dies ist allerdings nur das theoretisches Limit, falls immer sofort gekürzt werden kann. Bei einer zufälligen Sortierung der Knoten erhalten wir eine Zeitkomplexität von $\mathcal{O}(b^{\frac{3}{4}m})$.

IV. Zusammenfassung

Wir haben den Grundlegenden Minimax-Algorithmus kennengelernt und festgestellt, dass dieser ohne Optimierungen nicht echtzeitfähig ist. Mit dem Alpha-Beta-Pruning wurde eine der wesentlichen Optimierungen vorgestellt.

Es ist nicht auszuschließen, dass in dadurch nicht untersuchten Teilbäumen ein höherer Gewinn für Max zu finden wäre, allerdings hätte auch eine vollständige Suche ohne Kürzungen diesen Zug nie ausgewählt. Grund dafür ist die Eigenschaft des Minimax-Algorithmus', keinerlei Risiko einzugehen. Die mit Alpha-Beta-Kürzung optimierte Variante wird also stets die selben

Ergebnisse, wie das bisherige Minimax-Verfahrens liefern. Wir sparen uns lediglich weitere Untersuchungen des Suchbaumes, wenn sich Teilergebnisse ohnehin nicht mehr ändern können.

Natürlich kann man das Verfahren noch weiter verbessern, so ist die Zugreihenfolge für das Alpha-Beta-Pruning entscheidend. Mit bspw. der iterativen Tiefensuche könnte man diese günstig wählen und darüber hinaus auch ggf. begrenzte Zeit best möglich ausnutzen.

Weitere Fragen und Erweiterungen des Szenarios lassen noch Raum zur Vertiefung dieses Themas und müssen hier leider offen bleiben: Was passiert, wenn Zufall im Spiel ist? Wie können wir Spiele mit mehr als zwei Spielern abbilden? Und wie lassen sich Allianzen zwischen mehreren Spielern modellieren?

Literatur

[1] Sascha Klüppelholz „Entwurfs- und Analysemethoden für Algorithmen – Skript zur Vorlesung“, Sommersemester 2016

[2] Russel, Norvig „Künstliche Intelligenz – Ein moderner Ansatz“, 3., aktualisierte Auflage, 2012

[3] Ernst Zermelo „Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels“, 1913

Diese Ausarbeitung und zugehörige Präsentation sind auch auf github zu finden:
`https://github.com/foobar0112/tic`.