Spielbäume

Proseminar Theoretische Informatik

Joschka Heinrich*, TU Dresden

29. März 2018

Zusammenfassung

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 Minmax-Verfahrens und anderer Heuristiken wird eine Anwendung von Spielbäumen illustriert.

I. Einführung

Motivation. Überblick Aufbau Text.

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

Nullsummenspiel

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

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

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 von legalen Spielzügen R, 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 (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. Durch Anwenden eines legalen Spielzuges auf eine Konfiguration gelangen wir zu einer neuen Konfiguration. Dieser Zusammenhang wird in Abb. 1 veranschaulicht. 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).

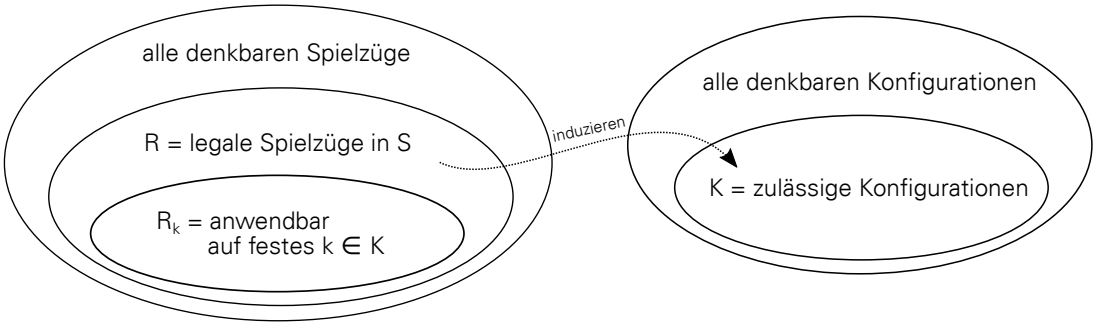

Abbildung 1: Das Venn–Diagramm zeigt die Teilmengenbeziehungen der Spielzüge einerseits und der Konfigurationen andererseits, wobei K induktiv über R definiert ist.

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

Die Menge K definieren wir nun induktiv über die Kindkonfiguration:

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

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

[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.

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

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

Als zusätzliche Vereinfachung schließen wir aus, eine über R aus k_0 generierte Konfiguration durch erneutes Anwenden legaler Spielzüge wieder erreichen zu können. Wir gelangen also im weiteren Spielverlauf nie zu einer Situation, die bereits aufgetreten ist.[2] Daraus folgt unmittelbar, dass G zyklenfrei ist.

Damit ist der Graphen G insbesondere ein Baum[3], mit k_0 als Wurzel und F als Blätter. So kann über den Baum entlang legaler Spielzüge traversiert werden und bspw. ein kompletter Spielverlauf mit n Zügen als Pfad $(k_0, k_1, \ldots, k_n \in F)$ subsequenter 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.

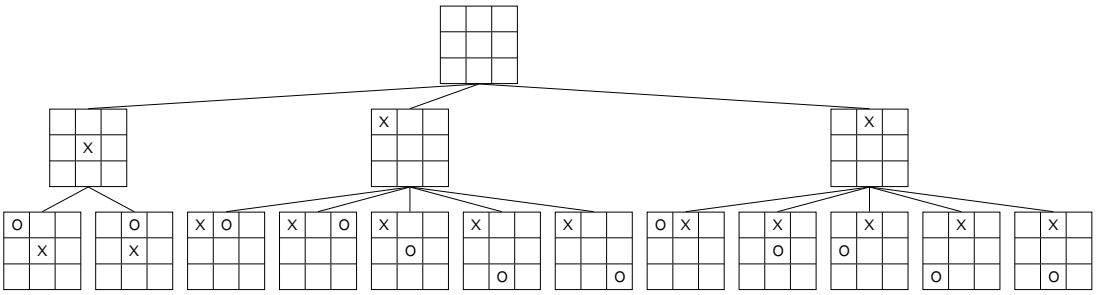

Abbildung 2: Ein Tic-Tac-Toe-Spielbaum der Tiefe 2, bei dem die Anzahl der Konfigurationen bereits durch Ausnutzung von Symmetrien optimiert wurde.

[2] Diese Einschränkung schließt viele Spiele wie bspw. Schach von der folgenden Betrachtung aus, da dort durch legale Spielzüge Konfigurationen reproduziert werden können. Viele der Aussagen lassen sich dennoch auch auf diese Art Spiele übertragen.

[3] 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.

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 Zug von Min antizipieren muss, der ihm den größten Nachteil bringen wird.[4]. Dieses Vorgehen führt zum **Minmax-Verfahren**[5], das wir im Folgenden genauer betrachten. Auch werden wir einige dessen Optimierungen kennenlernen, um unvollständige Echtzeitentscheidungen treffen zu können.

i. Minmax-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 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 werden soll und analog dazu **Min-Knoten** ($\triangledown$). In unserem Tic-Tac-Toe-Szenario alternieren die Knoten-Typen mit jedem Halbzug[6], da die Spieler sich stets abwechseln.

Jedem Knoten u im Spielbaum wird nun ein **Minmax-Wert** $minmax(u) \in \mathbb{N}$, der dem Nutzen aus Sicht von Max entspricht, zugeordnet. Dabei ergibt sich $minmax(u)$ rekursiv aus den Minmax-Werten der Kindknoten $N(u)$ und kann wird mit einer Tiefensuche[7] wie folgt rekursiv berechnet:

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

Die Rekursion endet mit dem Erreichen einer Endkonfiguration, dann wird die Gewinnfunktion angewendet. Je nach dem in welcher Ebene (Min oder Max) des Baumes wir uns befinden wird beim folgenden Aufstieg entsprechend das Minimum bzw. Maximum der Kinder nach oben weitergegeben. Schließlich wählt Max den Zug mit dem größten Minmax-Wert als seinen nächsten Zug. (vgl. Abb. 3)

Würde 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. Ohne Optimierungen müsste der Baum vollständig durchlaufen werden. Da schon bei sehr einfachen Spielen wie Tic-Tac-Toe, mit wenigen Freiheiten und einer überschaubaren Zuganzahl, die Anzahl der möglichen

[4] 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.

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

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

[7] 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.

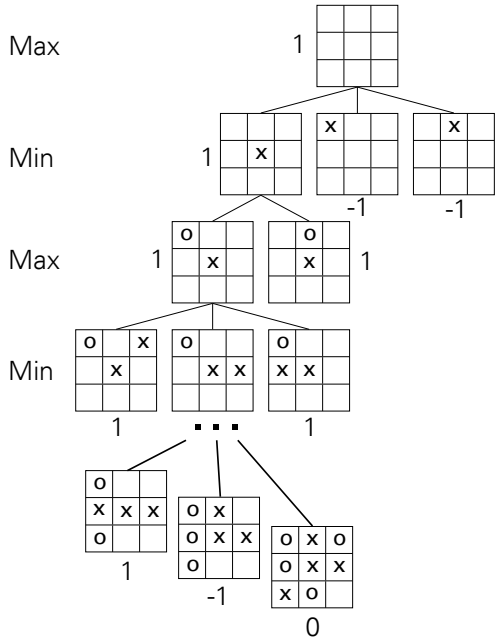

Abbildung 3: Ein Ausschnitt eines hypothetischen Spielbaumes an dem das Prinzip des Minmax-Verfahrens nachvollzogen werden kann: eine Gewinnfunktion bewertet Endkonfigurationen mit 1 bei Gewinn, −1 bei Niederlage und 0 bei Unentschieden; die Minmax-Werte werden entsprechend des Algorithmus' bis zur Wurzel propagiert; Max wählt schließlich den ersten Zug mit dem Kreuz in der Mitte, da der Minmax-Wert hier maximal ist.

Konfigurationen zu groß wird, um in annehmbarer Zeit Züge zu berechnen[8], kann das Verfahren so nicht angewendet werden. Vor allem bei Agenten für wesentlich komplexere und längere Spiele, wie Schach, kommen daher unter anderem folgende Optimierungen zum Einsatz.

ii. Optimierung mittels Heuristik

Damit nicht mehr der gesamte (und potenziell sehr tiefe) Teil-Baum der Kindkonfigurationen rekursiv bis zu den Blättern durchsucht werden muss, um die Minmax-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.[9] Den Baum, den wir ausgehend von einer beliebigen Konfiguration des Spielbaumes in der Tiefe t zur Laufzeit effizient aufbauen nennen wir **Suchbaum** (siehe dazu auch [3]).

Diese Heuristik $h(k)$ muss (1) die gleiche Ordnung wie die Gewinnfunktion $g(k)$ erzeugen, d.h.

[8] 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.

[9] Bisher haben wir so lange Knoten expandiert, d.h. deren Kinder analysiert, bis eine Endkonfiguration erreicht war (im Fall des Spielbaumes, da nur eine Gewinnfunktion 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 guten **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.

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.

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)

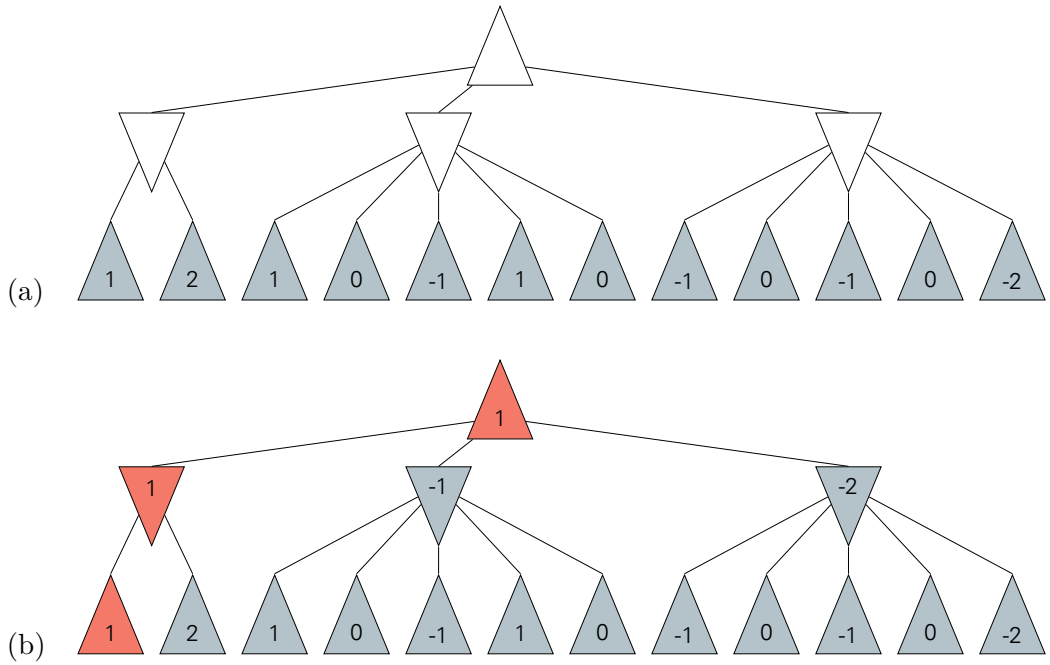

Abbildung 4: Wenden wir einen Suchbaum der Suchtiefe $t = 2$, ausgehend von der Startkonfiguration des leeren Feldes, aufbauen (vgl. Abb. 2) und dessen Blätter mit der Heuristik h_1 bewerten, erhalten wir die geschätzten Werte aus (a), die in (b) im gewohnten Modus zur Wurzel propagiert werden. Der Zug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

iii. Alpha-Beta-Pruning

Um die Komplexität des Minmax-Algorithmus weiter zu verringern, nehmen wir eine weitere Optimierung vor: das **Alpha-Beta-Pruning**[10]. Dieses erlaubt es, das Untersuchen von Teilbäume abzukürzen, ohne dabei gute Züge auszuschließen, indem Knoten nicht weiter expandiert werden, auch, wenn das Expansionskriterium erfüllt ist.

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 Minmax-Wertes $minmax(k)$ 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.

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

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

Wir können nun auf Basis der α und β Werte **Kürzungen (Cut)** vornehmen, d.h. Teilbäume aus dem Suchbaum löschen, sobald sich die beiden Schranken überschneiden: An einem Max-Knoten u – es wird hier stets der größte Minmax-Wert unter den Kindknoten gesucht – führen wir den s.g. α**-Cut** durch, sobald $\exists \beta_v : \beta_v \leq \alpha_u \ (v \in N(u))$. Analog dazu wird der β**-Cut** an einem Min-Knoten durchgeführt, wenn $\exists \alpha_v : \alpha_v \geq \beta_u \ (v \in N(u))$.

Beispiel, Grafik, Komplexität

IV. Zusammenfassung

Schlussfolgerung. Ausblick. Anwendbarkeit, Zufallsspiele

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`.