Ausarbeitung

Philipp Bock und Theodor Rauch

17. Januar 2021

Inhaltsverzeichnis

1 Hintergrund

1.1 Ansatz

Wir verwenden für unsere Lösung einen Ansatz aus der Spieltheorie, den sogenannten Spielbaum oder wie es in der Fachsprache heißt: Die Extensivform des Spiels. Aber was heißt das überhaupt, wenn man diesen Begriff noch nie gehört hat. Im Endeffekt bauen wir rekursiv einen Baum auf, der alle möglichen Spielzüge bis zu einer bestimmten Tiefe enthält, berechnen an dieser Tiefe dann eine Bewertungsfunktion und suchen uns dann den Zug aus, der uns zur potenziell besten Spielsituation bringt. Das klingt zwar im ersten Moment recht rechenaufwändig, wird aber im Abschnitt Vereinfachungen optimiert.

1.1.1 Min-Max-Baum

Die praktische Anwendung des Spielbaumes erfolgt im Normalfall durch einen Min-Max Baum. Dabei wird davon ausgegangen, dass zwei Spieler mit allen Informationen über das Spiel gegeneinander spielen (Wie wir das Problem der zwei Spieler umgehen in Multi-Minimax). Dabei geht man davon aus, dass diese Spieler immer optimal spielen. Wir nennen den einen Max und den anderen Min. Max ist dabei meist man selbst und der Gegner ist Min. Möchte man nun einen Zug planen, ruft man für Min alle möglichen Züge auf, dann rufen diese wieder alle für Max auf und dann wieder für Min. Damit baut man einen Rekursionsbaum auf, der erst abbricht, wenn eine definierte Rekursionstiefe erreicht wird. An den Enden der Rekursion wird nun der potenzielle Spielstand ausgewertet und eine bewertende Nummer zurückgegeben. Je höher diese Nummer, desto besser für Max und umgekehrt für Min. Wenn nun die Rekursion zurückläuft, wählen sich Min und Max abwechselnd immer den besten Zug von denen aus, die der jeweils Andere vorher ausgewählt hat und so weiter, bis wir zu jedem der ursprünglich möglichen Züge eine Bewertung haben und wählen den besten.

1.1.2 Spielstandsbewertung

Die Spielstandsbewertungsfunktion muss so aufwändig wie nötig und so einfach wie möglich sein. Wäre sie zu schwach, gehen wir von einer komplett falschen Annahme über die Qualität einer Spielsituation für uns aus. Ist sie zu aufwändig können wir nicht annähernd so weit ïn die Zukunft schauenünd der Vorteil, den wir dadurch haben ist sehr gering. Wir haben uns dafür entschieden, sie möglichst einfach zu halten. Dazu kam noch, dass wir recht spät mit dem Projekt angefangen hatten und nicht mittendrin unser Konzept komplett ändern konnten, mehr dazu Aber in der Auswertung.

1.2 Vereinfachungen

1.2.1 Motivation

Die Motivation ist eigentlich klar. Gehen wir davon aus, dass wir in jedem rekursiven Aufruf jede mögliche Stellung der Spieler durchprobieren haben wir bei bis 6 Spielern und bis zu 5 möglichen Zügen pro Spieler 30 Funktionsaufrufe in einer Funktion und bei einem Spielbaum von Tiefe t bis zu 30^t Blätter, in denen dann die Spielbewertungsfunktion aufgerufen wird. Und über die rekursiven Aufrufe möchte man sich dann sowieso keine Gedanken machen. Da unser Algorithmus darauf basiert möglichst weit in die Zukunft zu schauen, sollte t auch möglichst hoch werden.

1.2.2 Rust

An dieser Stelle wird nun die Wahl der Programmiersprache erläutert. Rust ist eine Systemprogrammiersprache, die wir aufgrund folgender Eigenschaften gewählt haben:

1. Geschwindigkeit. Rust besitzt keine VM wie Python oder Java und verwendet und kommt ohne Garbage Collector aus. Da in manchen Funktionen bei uns jede Mikrosekunde zählt, kommt uns das gerade recht.

2. Sicherheit. Der Compiler von Rust in Verbindung mit der teilweise Strikten Referenzweitergabe macht Laufzeitfehler sehr selten.

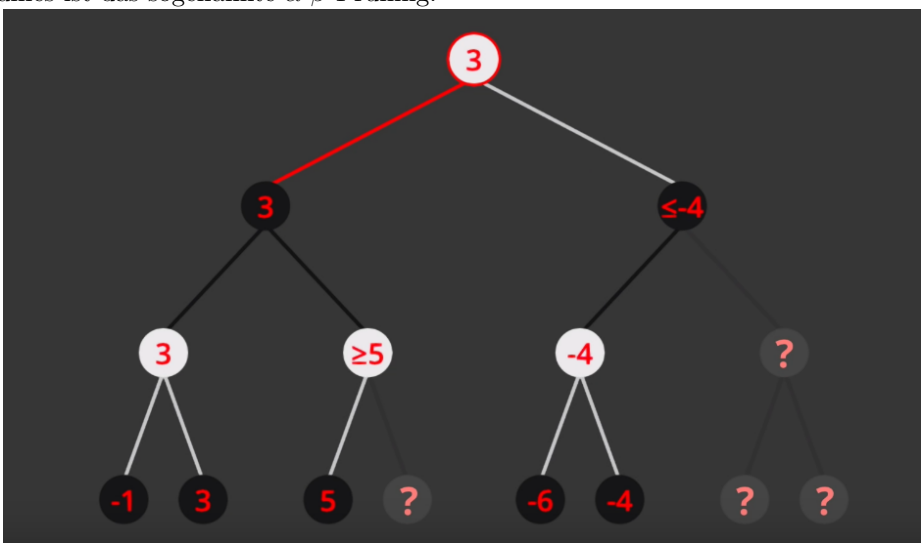
Die oft als ungewöhnlich angesehene Programmierweise in Rust hat uns nicht gestört, da wir aber bereits schon Projekte in Rust geschrieben haben.

1.2.3 Bitboards

Die erste Optimierung, die tatsächlich etwas mit unserem Algorithmus zu tun hat und nicht mit der Wahl der Programmiersprache sind sogenannte Bitboards. Dadurch, dass uns ja de facto egal sein kann gegen wessen Linie wir krachen, können wir diese einfach in Einsen umwandeln. Damit haben wir ein Feld an Einsen und Nullen. Dieses Wandeln wir in zwei Arrays mit Zahlen um, die - In Bitdarstellung - entweder die Reihen oder die Spalten des Feldes ergeben. Nun kann man Züge teilweise mit Bitoperationen berechnen und was liebt ein Informatiker lieber als Bitoperationen. Außerdem brauchen die Felder dann weniger Speicher.

1.2.4 α-β - Pruning

Einer der wohl wichtigsten und typischsten Verbesserungen eines Min-Max-Baumes ist das sogenannte α-β-Pruning.



In diesem Beispiel ist ein aufgebauter Min-Max-Baum zu sehen, bei dem weiß Max und schwarz Min ist. Schauen wir uns das Beispiel an: Wir bauen rekursiv den linken Teilbaum unter dem obersten Knoten auf, dann den rechten. Dabei fällt uns hier auf, dass unter der -4 der rechte Teilbaum ausgegraut ist. Weshalb ist das so? Max nimmt immer den höchsten Zug und Min den niedrigsten. Nun wissen wir, dass das Ergebnis im linken Teilbaum 3 ist und im rechten haben wir in dem einen Teil schon die -4 berechnet. Da also Max nur ein schlechteres Ergebnis als die -4 oder die -4 selbst nehmen würde, nehmen wir so oder so die Alternative 3 und der restliche Baum ist uns egal. Dieses Muster findet man recht oft und erspart sehr viel Zeit.

1.2.5 Multiminimax

Multi-mini-max ist ein Algorithmus zur Zug Berechnung aus Lecture Notes in Artificial Intelligence, genauer AI 2019: Advances in Artificial Intelligence. Er stellt ein neuen Ansatz im Spiel gegen mehrere Gegner*innen dar. Dabei wird sich jeder mögliche eigene Zug getrennt angeguckt und davon der beste genommen. Die einzelnen Zügen kriegen eine Bewertung, indem sie gegen jeden Gegner einzeln die Bewertung mit dem Minimax Algorithmus berechnen. Davon kriegt der Zug den schlechtesten Wert zugewiesen. Ein großer Vorteil hiervon ist die verhältnismäßig effiziente Laufzeit und Möglichkeiten zur Optimierung (Pruning, Move ordering).

2 Auswertung

2.1 Wahl des Algorithmus'

Der Algorithmus war uns aus mehreren vorherigen Projekten schon vertraut. In diesem Fall wie auch im vorherigen haben wir auch eine hohe Suchtiefe gesetzt. Im Nachhinein ist eine mächtigere Spielstandsbewertungsfunktion mit geringerer Baumtiefe vielleicht die bessere Lösung. Das müsste sich aber erst in einem ausgiebigen Test zeigen.

2.2 Qualität der Lösung

Aus einer theoretischen Perspektive ist die Lösung nicht sonderlich hochwertig. Es sind recht viele Berechnungen nötig und das Ergebnis ist nicht sonderlich Optimal. Der Algorithmus "denktßu kurz. Bei einem großen Spielfeld sind Langzeitstrategien wichtiger und egal wie viel Zeit wir haben, ab einer Suchtiefe von 14 oder höher dauert die Berechnung zu lange. Leider entspricht das nur 7 echten Zügen... Für die investierte Zeit (von Prügeleien mit der Verknüpfung von Rust, Python und Docker abgesehen) halten wir es aber für eine solide Basis, auf der mit einer besseren Spielstandsbewertung aufgebaut werden kann, damit sich der "Blick in die Zukunftäuch lohnt.

2.3 Probleme