Chess AI

Tool- und Plugin Projekt von Kristofer Schmitz (schmitkr, 962643)

Hintergrund

Motivation

Künstliche Intelligenz, ein sehr komplexes und interessantes Thema. Heute aus dem Alltag kaum weg zu denken, dachte ich mir, ich könnte mich an dem Thema versuchen.
Für diesen Zweck habe ich ein simples Schachspiel, welches die Basis Konzepte besitzt, und eine dazu passende künstliche Intelligenz geschrieben.
Mit den hier benutzten Algorithmen verspricht die AI, gerade für Laien, eine gute Herausforderung zu sein.

Struktur

Hauptspiel

Ich habe versucht den Kern des Spiels möglichst übersichtlich und effizient zu gestalten, damit ich später keine Probleme bei der Implementierung der AI habe. Gefährlich können hier lange Zugriffszeiten auf Daten werden, wenn man einen großen Baum traversiert und in jedem Schritt eine Abfrage der aktuellen Board Situation braucht, wie es hier der Fall ist. Die von mir benutzten Datenstrukturen sind möglicherweise nicht optimal, erledigen ihre Aufgabe aber zur Genüge.

GameManager

Hier wird der Schwierigkeitsgrad ausgewählt und eine neue Session erstellt. Wie es sich für GameManager gehört handelt es sich um ein Singleton, der auch für die Szenenwechsel verantwortlich ist.

SessionManager

Der SessionManager steuert das Spielgeschehen. Wird mit der Maus geklickt, wird dieser Klick hier verarbeitet. So wird entweder eine neue Figur gewählt, abgewählt oder eine neuer Move initialisiert. Nach jeder Bewegung wird eine AI Bewegung erzeugt.

ChessBoard

Das ChessBoard verfügt über zwei Dictionaries, in denen alle ChessPieces und Tiles gespeichert sind, sowie über die wichtigsten Methoden des Schachspiels.

ChessPiece & Tile

Diese zwei Klassen beinhalten alle nötigen Informationen, welche eine Spielfigur, bzw das Feld braucht (Position, Farbe etc.).

GameSetup

Eine Helferklasse, mit deren Hilfe man das Schachbrett aufbaut und die Schachfiguren auf den richtigen Positionen erscheinen lässt.

Move

Die Move - Klasse beinhaltet alle wichtigen Informationen für die Bewegung einer Schachfigur.

ChessPieceMovements

ChessPieceMovement ist die Oberklasse jeder Movementklasse der unterschiedlichen Figuren. Hier kriegt man eine Liste an gültigen Moves zurück, welche dem Bewegungsmuster der Figuren entspricht.

AI - Part

Um zu einer anspruchsvollen AI zu gelangen, kann man schrittweise vorgehen. Ich habe diese Schritte in den verschiedenen Schwierigkeitsgraden verpackt.

IAI

AI Interface. Es beinhaltet die Methoden EvalBoard, Minimax, PerformFakeMove und UndoFakeMove, welche implementiert werden müssen.

AICore

Die Oberklasse der verschiedenen AI-Grade implementiert das Interface IAI. Vorhanden sind hier die wichtigsten Methoden. SetPieceValue() setzt bei Erzeugung des Objektes die Werte der zugehörigen Schachfiguren, welche für die Berechnung des Board Wertes wichtig sind. In der Methode EvalBoard wird dieser Board Wert berechnet.
PerformFakeMove und UndoFakeMove simulieren Bewegungen auf dem Brett, damit die AI mehrere Züge in die Zukunft schauen kann.

RandomAI

Der erste und einfachste Schritt. Die AI nimmt sich aus dem Pool der möglichen Moves einen zufälligen heraus und führt ihn aus.

EasyAI

Hier entscheidet sich die AI für den Move, welcher den größten Wert der EvalBoard Methode liefert. Sie wird demnach immer, wenn möglich, die gegnerische Figur mit dem höchsten Wert schlagen.

NormalAI

Die in der NormalAI implementierten AI kann man schon als fertige AI bezeichnen. Implementiert wurde die AI mit dem Minimax Algorithmus (dazu später mehr). Die AI schaut mehrere Züge in die Zukunft und ermittelt folgend den besten Zug.

HardAI

Das Prinzip der NormalAI wird hier weiter ausgebaut. Zusätzlich zu dem Minimax Algorithmus zum Auswählen des besten Zuges werden hier noch die Positionen der Figuren auf dem Board mit Werten versehen und ausgewertet.

Algorithmen

Um eine einfache Schach AI zu implementieren, muss man dieser mitteilen, welche Züge gut und welche schlecht sind.
Aus der Sichtweise der AI gilt es, folgende Fragen zu beantworten:

- Welche Zugmöglichkeiten habe ich?
- Wie sieht die Situation nach dem Zug aus?
- Welche Figur lohnt sich am meisten zu zerstören?
- Lohnt es sich diese Figur zu zerstören, wenn folgend meine Figur zerstört wird?
- Was wird der Gegner nach meinem Zug machen?
- Wann habe ich gewonnen?
- Welcher Weg führt mich zum Sieg?

EvalBoard

Die Fragen nach der Wertigkeit der Figuren lässt sich leicht mit diesem Algorithmus beantworten. Ziel ist es, den Wert des Boards wieder zu geben. Je nach implementierung kann man hier die Rollen tauschen. Ich habe es dementsprechend geregelt, dass alle Figuren der AI einen positiven und alle Figuren des Gegners (Spielers) einen negativen Wert besitzen. Diese Werte werden addiert und zurück gegeben.
Nun kann man nach jedem Zug die Werte des Boards vergleichen. Grundlegend, je höher der Wert, desto vorteilhafter ist dies für die AI.
In der Easy AI Einstellung gehe ich nur nach diesem Algorithmus vor, weshalb immer, wenn möglich, die gegnerische Figur geschlagen wird. Bei mehreren Möglichkeiten zum Schlagen wird die Figur mit dem höchsten Wert zerstört.

EvalBoardExtended

Für die schwierigste AI habe ich die EvalBoard Methode erweitert. Zusätzlich zu den Werten der Figuren werden noch die Positionen ausgewertet. Meine benutzte Verteilung findet man auf der Chess Programming Wiki Seite (9). Grob gesagt werden hier Schach-Weisheiten

ausgenutzt wie zum Beispiel, dass man den Bishop und Knight eher auf der Mitte des Boards platzieren will.

Minimax

Der von mir benutzte Algorithmus, um den besten Zug der AI zu berechnen, nennt sich Minimax. Mit diesem Algorithmus ist es möglich, mehrere Züge bis zu einer bestimmten Tiefe zu simulieren. Jeder Schritt versucht das bestmögliche Ergebnis aus der wechselnden Perspektive beider Spieler zu berechnen. Aus der Sicht der AI sucht der Algorithmus den maximalen Wert und wählt diesen aus, folgend wird im nächsten Schritt der minimale Wert gewählt, da, wie oben beschrieben, in jedem simulierten Zug die EvalBoard Methode aufgerufen wird. Es ist im Interesse des maximierenden Spielers, den Wert möglichst hoch zu bringen, während der minimierende Spieler das Gegenteil versucht.

Warum Minimax?

“Der Minimax-Algorithmus ist ein Algorithmus zur Ermittlung der optimalen Spielstrategie für endliche Zwei-Personen-Nullsummenspiele mit perfekter Information.”(1).

Nullsummenspiel und perfekte Information sind hierbei Begriffe aus der Spieltheorie.

“Nullsummenspiele beschreiben in der Spieltheorie Situationen, also Spiele im verallgemeinerten Sinne, bei denen die Summe der Gewinne und Verluste aller Spieler zusammengenommen gleich null ist.”(2).

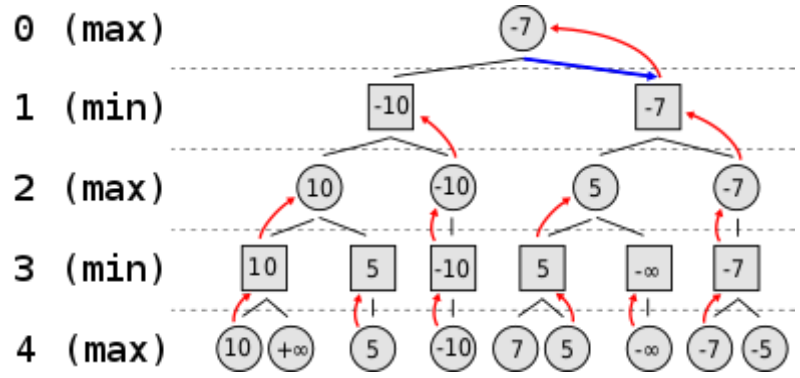
“[...] ein Spiel [besitzt] perfekte Information, wenn jedem Spieler zum Zeitpunkt einer Entscheidung stets das vorangegangene Spielgeschehen, d. h. die zuvor getroffenen Entscheidungen seiner Mitspieler sowie die zuvor getroffenen Zufallsentscheidungen, bekannt sind.”(3).

Die meisten klassischen Gesellschaftsspiele fallen in diese Kategorie, so auch Schach, was den Algorithmus für das Vorhaben perfekt macht.

Der Name resultiert aus der Herangehensweise des Algorithmus, beim traversieren des Baumes abwechselnd den Wert zu mini- , bzw zu maximieren.

Beispiel

Der Minimax Algorithmus lässt sich am besten als einen Baum anschaulich darstellen.



In diesem Beispiel ist ein Binärbaum mit der Tiefe 4 gegeben. Jeder Knoten des Baumes stellt den Zug eines Spielers dar. Die Kreise sind die Züge des maximierenden Spielers, die Quadrate die des minimierenden Spielers.
Die roten Pfeile beschreiben den ausgewählten Zug und der blaue Pfeil die Entscheidung, also das Endergebnis.
Es handelt sich hier um einen rekursiven Algorithmus, also beginnt man die Betrachtung bei den Blättern des Baumes.
Auf der Ebene 4 befinden sich die berechneten Werte, welche in höhere Ebenen getragen werden.
Auf Ebene 3 entscheidet der minimierende Spieler. Der kleinere der beiden Werte der Kindknoten (Blätter) wird hier ausgewählt.
Danach entscheidet der Algorithmus auf Ebene 2 aus der Sicht des maximierenden Spielers, und wählt den größten Wert der direkten Kindknoten.
Folgend ist noch einmal der minimierende Spieler dran und letztendlich wird das Maximum der zwei Knoten aus Ebene 1 ausgewählt, welches das Endergebnis darstellt.

Pseudocode

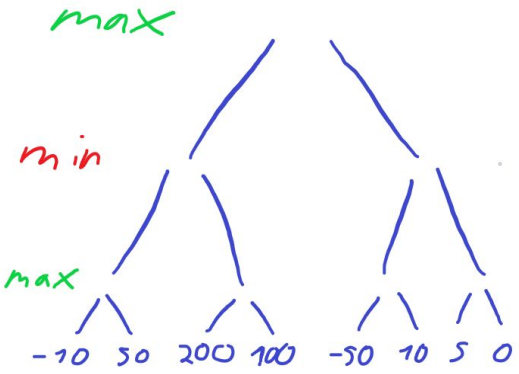
Was macht der Minimax Algorithmus im Detail? Dazu gibt es folgenden Pseudocode(4):

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

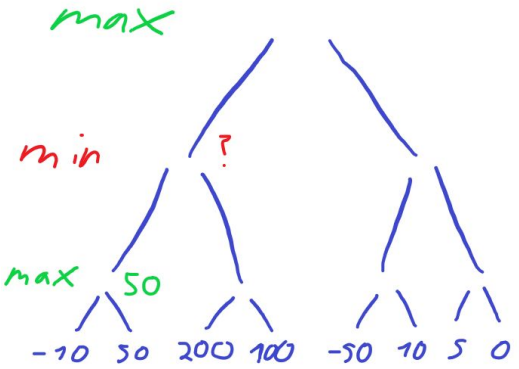
Der Code ist relativ simpel. Der Rekursionsanker befindet sich bei depth = 0, in welchem die Funktion einen Wert zurück gibt. In unserem Fall wäre dies der aktuelle Wert des Boards, welches durch die EvalBoard Methode berechnet wird.
Bei jeder Iteration wird zwischen maxi- und minimierenden Spieler unterschieden und der entsprechende Wert weitergegeben.

Alpha-Beta-Pruning

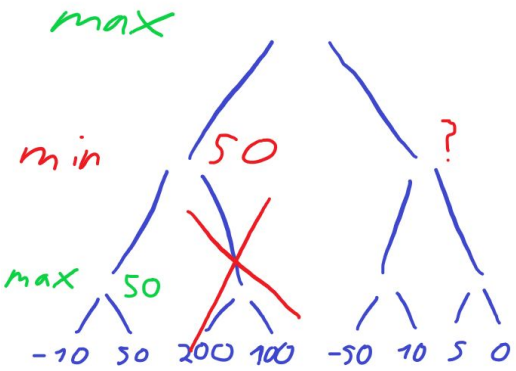
In einem Schachspiel gibt es bekanntermaßen mehr als zwei Zugmöglichkeiten, was den Baum sehr verbreitert. Dies wirkt sich exponentiell auf die Laufzeit aus und kann das Spiel, je nach Tiefe des Baumes, unspielbar machen.
Abhilfe schafft hier die Optimierung namens Alpha-Beta-Pruning.
Idee hierbei ist es, unnötige Pfade wegfallen zu lassen, um Rechenzeit einzusparen.
Kleines (selbstgezeichnetes) Beispiel:



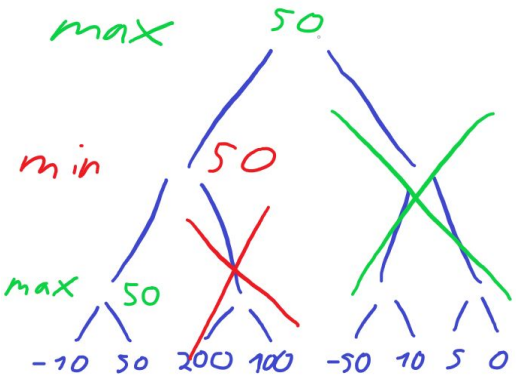
Hier ist ein Beispiel Minimax Baum aufgezeichnet. Die Blätter bestehen aus den Werten der einzelnen Moves.



Der Maximierende Spieler wählt den größten Wert aus. Darauf wird der minimierende Spieler den kleinsten Wert auswählen. Dafür müsste man jetzt den rechten Pfad evaluieren. Wir wissen aber, dass der Wert, den der minimierende Spieler auswählt, kleiner 50 sein muss. In dem Pfad gibt es jedoch keinen Wert kleiner als 50, weshalb dieser ignoriert werden kann.



Selbes Spiel noch einmal. Nun wählt der maximierende Spieler einen Wert aus. Dieser Wert muss größer 50 sein, sonst wird er nicht ausgewählt. Da es in dem rechten Teilbaum jedoch keinen Wert größer als 50 gibt, fällt auch dieser Weg.



Mit dieser Methode kann man einiges an Rechenaufwand einsparen, was gerade bei großen Bäumen die Performance stark beeinflussen kann.

Meine Implementierung

```
private int Minimax(int depth, int alpha, int beta, bool isMaximizing)
    {
        // recursion anchor
        if (depth <= 0)
        {
            // return Board Value
            return EvalBoard();
        }
        // currently maximizing player (AI)
        if (isMaximizing)
        {
            // has to be sufficient big negative number
            int bestMoveValue = -1000000;
            // List of all possible Moves of current Player
            List<Move> moveList =
                board.GetAllPossibleMovesOfColor(ChessPiece.Color.Black);
            // shuffle List to prevent move repetition
            moveList = moveList.OrderBy(a => Guid.NewGuid()).ToList();
```

```
            // iterate over all possible moves
            foreach (Move move in moveList)
            {
                // only to simulate move
                PerformFakeMove(move);
                // recursive call
                value = Minimax(depth - 1, alpha, beta, false);
                // undo simulated move
                UndoFakeMove();
                // check if move gives better value
                if (value > bestMoveValue)
                {
                    bestMoveValue = value;
                    // set bestMove if algorithm is at its root
                    if (depth == maxCalcDepth)
                    {
                        bestMove = move;
                    }
                    // set alpha
                    alpha = bestMoveValue;
                }
                // alpha - beta pruning
                if (value >= beta)
                {
                    break;
                }
            }
            return alpha;
        }
        // currently minimizing player
        else
        {
            // value has to be bigger than the biggest possible value of EvalBoard
            int bestMoveValue = 1000000;
            List<Move> moveList =
                board.GetAllPossibleMovesOfColor(ChessPiece.Color.White);
            moveList = moveList.OrderBy(a => Guid.NewGuid()).ToList();

            foreach (Move move in moveList)
            {
                PerformFakeMove(move);
                value = Minimax(depth - 1, alpha, beta, true);
                UndoFakeMove();
                // check if move gives worse value
                if (value < bestMoveValue)
                {
                    bestMoveValue = value;
                    // set beta
                    beta = bestMoveValue;
                }
                // alpha - beta pruning
                if (value <= alpha)
                {
                    break;
                }
            }
            return beta;
        }
    }
```

Mein Rekursionsanker befindet sich bei depth <= 0, in welchem die Methode EvalBoard aufgerufen wird und den Wert des Boards zurück gibt.
Bei jeder Iteration wird, wie in der Beschreibung des Algorithmus, zwischen max und min Player unterschieden.
Hier wird zuerst ein default Wert für die bestMoveValue angelegt und eine Liste von dem ChessBoard abgefragt, bestehend aus allen möglichen Moves der Schachfiguren des jeweiligen Spielers. Diese Liste wird gemischt, um Wiederholungen von Moves vorzubeugen..
Folgend wird über jeden möglichen Move der Schachfiguren gelaufen. Dieser Move wird mit PerformFakeMove simuliert, die Methode wird rekursiv erneut aufgerufen und danach mit UndoFakeMove wieder rückgängig gemacht.
Ist der berechnete Wert größer, bzw kleiner als der bisherigen bestMoveValue, so wird dieser Wert abgespeichert. Befindet sich der Algorithmus auf der obersten Ebene (1.Iteration), wird der ausgewählte Move abgespeichert. Dieser wird dann dem Hauptspiel übergeben und ausgeführt.
Durch Alpha-Beta-Pruning fallen jeweils die überflüssigen Pfade weg.

Fazit

Mit den hier beschriebenen Methoden ist es mir gelungen eine simple Schach AI zu schreiben. Während sie Profi Spielern nicht das Wasser reichen kann, wird sie Laien jedoch Probleme bereiten können.
Eine Problemstelle war und ist die Laufzeit. Lässt man die AI 4 Züge im voraus berechnen, verlangsamt dies die Spiel Performance spürbar. Jedoch ist die AI schon ab 3 Zügen ein ernst zu nehmender Gegner. Hier könnte man noch ein wenig rumbasteln, um die Laufzeit weiter zu verringern. Es sollte auf jeden Fall möglich sein, mit Alpha-Beta-Pruning noch mehr Rechenzeit einzusparen, indem man zusätzliche Teilbäume wegfallen lässt. Man könnte die Liste der Moves auch vorsortieren, aber ob dies spürbar wäre kann ich nicht genau sagen.
Wie oben beschrieben handelt es sich bei Minimax um einen Algorithmus, den man auf diverse andere Gesellschaftsspiele anwenden kann.

Bonus

Ich habe zusätzlich eine TicTacToe AI basierende auf meiner Schach AI geschrieben.

Quellen

(1) Seite „Minimax-Algorithmus“. In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 12. Januar 2020, 13:45 UTC. URL: https://de.wikipedia.org/w/index.php?title=Minimax-Algorithmus&oldid=195739275 (Abgerufen: 19. August 2020, 13:45 UTC)

(2) Seite „Nullsummenspiel“. In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 23. Januar 2018, 07:17 UTC. URL: https://de.wikipedia.org/w/index.php?title=Nullsummenspiel&oldid=173230670 (Abgerufen: 19. August 2020, 14:00 UTC)
(3) Seite „Spiel mit perfekter Information“. In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 10. November 2019, 09:56 UTC. URL: https://de.wikipedia.org/w/index.php?title=Spiel_mit_perfekter_Information&oldid=193918000 (Abgerufen: 19. August 2020, 14:01 UTC)
(4) Wikipedia contributors. Minimax. Wikipedia, The Free Encyclopedia. June 21, 2020, 10:33 UTC. Available at: https://en.wikipedia.org/w/index.php?title=Minimax&oldid=963711709. Accessed August 19, 2020.
(5) https://byanofsky.com/2017/07/06/building-a-simple-chess-ai/ (Abgerufen 21.08.2020)
(6) http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html (Abgerufen 21.08.2020)
(7) https://github.com/SkylerAlvarez/Chess-AI-Unity (Abgerufen 18.08.2020)
(8) https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/ (Abgerufen 17.08.2020)
(9) https://www.chessprogramming.org/Simplified_Evaluation_Function (Abgerufen 21.08.2020)