Dameprogramme: eine praktische Analyse des Minimax-Algorithmus

Carsi González, Ana

Praktische Einführung in Betriebssysteme und Rechnernetze
ana.carsi.gonzalez@math.uni-giessen.de

7. April 2024

Zusammenfassung

Dieser Bericht stellt das Verfahren, das befolgt wurde, um einen Künstliche-Intelligenz-Ansatz für das Damespiel zu programmieren.

I. Einleitung

Heutzutage ist eines des größten Forschungsgebiete der modernen künstlichen Intelligenz im Erstellen von Computerspielen für beliebte Spiele und jetzt hat die im Bereich der Algorithmen und Datenstrukturen entwickelte Forschung dazu geführt, dass die Agenten in der KI gegen jeden menschlichen Spieler gewinnen können.

Beispielsweise, der Sieg von AlphaGo [1] gegen einen 18-fachen Go-Weltmeister war ein wichtiger Meilenstein in der künstlichen Intelligenz und definitiv ein Beweis dafür. Go ist weitaus komplexer als das Damespiel mit mehr als 10^{172} möglichen Brettkonfigurationen (während das Damespiel nur 10^{18} hat)[2]. Vor AlphaGo und seinen Nachfolgern hatte kein Go-Programm einen professionellen Go-Spieler besiegt, und schließlich verallgemeinerte DeepMind den AlphaGo Zero-Algorithmus (genannt AlphaZero) und wandte ihn mit herausragenden Ergebnissen auf die Spiele Go, Chess und das Damespiel.[3] Im zweiten Abschnitt beschreiben wir die Grundlagen für das Design des Algorithmus und später beschreiben wir den Algorithmus und seine Effizienz im Abschnitt 3. Am ende wird ein ansatz in python gezeigt.

II. Beschreibung der Ziele

Das Damespiel ist ein *Zwei-Spieler-Nullsummenspiel*, denn in einer mathematischen Darstellung: ein Spieler gewinnt (+1) und ein anderer Spieler verliert (-1) oder beide gewinnen nicht.

A. Regeln für das Damespiel n-ter Ordnung

In diesem Spiel setzt jeder Spieler $n^2 - n$ Steine auf dem $2nx2n$-Brett, das abwechselnd hell und dunkel gefärbte Felder besitzt. Obwohl wir den Algorithmus für ein $2nx2n$-Brett entwickeln könnten, beschränken wir uns bei diesem Ansatz auf den beliebten Fall *8x8*. In diesem Ansatz haben wir weiße und rosa Steine verwendet, wobei die weißen dem KI-Spieler entsprechen.

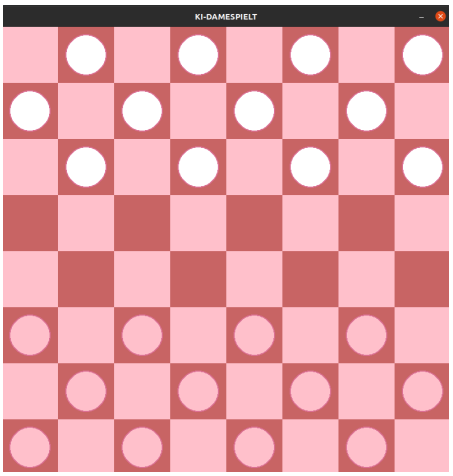

Abbildung 1: *8x8-Brett zu benutzen.*

Die Bewegung eines Steines kann vorwärts auf ein leeres diagonal sein. Falls das

Feld von einem gegnerischen Stein besetzt ist, aber das nächste dahinter frei, kann der gegnerische Stein übersprungen und vom Brett genommen werden. Falls ein Stein nach einem Sprung erneut springen kann, muss er das machen.

Wenn ein Stein die am weitesten vorne liegende Reihe (bekannt als "Königsreihe") erreicht, wird er ein König und die Runde ist vorbei. Ein König kann Steine in alle vier diagonalen Richtungen bewegen und schlagen.

Man gewinnt, wenn der Gegner nicht mehr ziehen kann weil seine Steine geschlagen sind oder festsitzen. Das Spiel kann auch für unentschieden erklärt werden.

B. Relevante Punkte für die Umsetzung

Die Zustände dieses Spiels sollen in einem Entscheidungsbaum (nämlich *Spielbaum*) dargestellt werden, durch den die Handlungsmöglichkeiten der Spieler genau definiert werden. Dies führt dazu, die Realisierung des Spiels als **Suchproblem** zu betrachten, wo die einzelnen Zustände voll zugänglich sind. Unter der Annahme des besten Spiels von beiden Seiten ist das Ziel der Suche, einen Pfad von der Wurzel zum höchstwertigen Blattknoten zu finden. Die Minimax-Regel wird verwendet, um den Wert dieser Ergebnisse zurück in den Ausgangszustand zu propagieren, wo die Wurzel des Baums ist der Anfangszustand und die Blattknoten sind die Endzustände.

III. Beschriebung des Algorithums

A. Minimax Verfahren

Der zu verwendende Ansatz ist der Suchalgorithmus Minimax, der es dem Programm ermöglicht, auf mögliche zukünftige Positionen zu schauen, bevor es entscheidet, welche Bewegung an der aktuellen Position ausgeführt werden soll.

Um dies zu veranschaulichen, nehmen wir an, dass für jede Position (Situation im Brett) nur zwei mögliche Züge zur Auswahl stehen (zwei getrennte Zweige), an deren Ende zwei weitere neue Positionen stehen, bei denen der andere Spieler nicht gerade an der Reihe ist . Wir können den Baum weiter erweitern, bis wir entweder das Ende des Spiels erreichen oder uns entscheiden aufzuhören, weil es zu lange dauern würde, tiefer zu gehen. In jedem Fall müssen wir am Ende des Baums eine **statische Bewertung** dieser Endpositionen durchführen, die versucht abzuschätzen, wie gut die Position für eine Seite ist, ohne weitere Züge zu machen. Bei diesem Ansatz addieren wir die Werte der verbleibenden weißen Steine und subtrahieren die verbleibenden schwarzen (rosa) Steine [beachten Sie A.1].

Daher begünstigen große Werte Weiß und kleine Werte begünstigen Schwarz. Aus diesem Grund versucht Weiß immer, die Bewertung zu **maximieren**, während Schwarz versucht, sie zu **minimieren**.

Stellen Sie sich vor, wir bewerten sie und sie kommen als -1 und 3 heraus. Da im vorherigen Level Weiß an der Reihe war, sich zu bewegen, und Weiß versucht, den Wert zu maximieren, kann Weiß den Wert 3 erhalten. Nehmen wir an, dass wir auf der anderen Seite das folgende Ergebnis haben. Da Schwarz (Rosa) dann an der Reihe ist, wählt Schwarz (Rosa) den Zug, der zur niedrigsten Bewertung führt, also 3. Selbst wenn Schwarz (Rosa) den besten Zug bekommt, erhält Weiß immer noch ein 3.

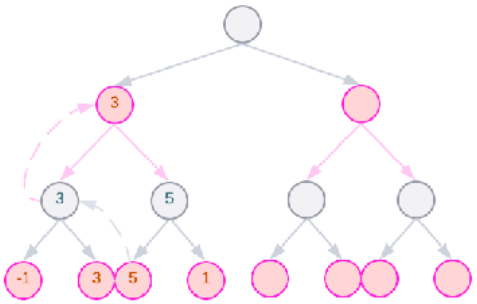

Abbildung 2: *Beispiel für den Entscheidingungsbaum, wobei Weiß der AI-Maximierenspieler ist.*

B. Baum Stutzen

Da die Spielbäume, die mehr als ein paar Ebenen umfassen, sehr umfangreich werden können, wurde die *Alpha-Beta-Pruning* (beschnei-

den und zurechtstutzen) des Suchbaums ergänzt.

Auf diese Weise kann die Suchzeit auf den aussichtsreicheren Teilbaum beschränkt und gleichzeitig eine tiefere Suche durchgeführt werden. Wie sein Vorgänger gehört er zur Klasse der Branch-and-Bound-Algorithmen. Die Optimierung reduziert die effektive Tiefe auf etwas mehr als die Hälfte des einfachen Minimax, wenn Knoten in optimaler oder nahezu optimaler Reihenfolge ausgewertet werden (beste Wahl für die Seite, die am Zug ist, zuerst an jedem Knoten geordnet). Jedoch wurde in diesem Fall kein Ordnungsbaumalgorithmus implementiert. Daher können die Ergebnisse nicht so optimal wie erwartet sein.

C. Analyse der Optimierung

Bei einem (durchschnittlichen oder konstanten) Verzweigungsfaktor b und einer Suchtiefe d Lagen ist die maximale Anzahl der bewerteten Blattknotenpositionen (wenn die Zugreihenfolge pessimal ist)

$$\mathcal{O}(bb...b) = O(bd) \tag{1}$$

dasselbe wie eine einfache Minimax-Suche. Wenn die Zugreihenfolge für die Suche optimal ist (das bedeutet, dass die besten Züge immer zuerst gesucht werden), beträgt die Anzahl der bewerteten Blattknotenpositionen etwa

$$\mathcal{O}(b1b1...b) \tag{2}$$

für ungerade Tiefe und

$$\mathcal{O}(b1b1...1) \tag{3}$$

für gleichmäßige Tiefe, also

$$\mathcal{O}(bd/2) = \mathcal{O}(b^{d/2}) = O(\sqrt{b^d}). \tag{4}$$

In dem letzteren Fall, wenn die Anzahl der Suchen gerade ist, wird der effektive Verzweigungsfaktor auf seine Quadratwurzel reduziert, oder äquivalent kann die Suche bei gleichem Rechenaufwand doppelt so tief sein. [6]

IV. Entwicklung in Python

Bei diesem Ansatz wurde in **pygame** eine Dame-GUI erstellt, um den Minimax-Algorithmus zu visualisieren. Zusätzlich zum Animieren des Bretts und der Figuren zeigt die GUI auch alle möglichen Züge an, die der aktuelle Spieler machen kann. Das Beobachten, wie der Algorithmus gegen sich selbst spielt, ist eine gute Möglichkeit, zu veranschaulichen, wie der Algorithmus das Spiel wahrnimmt.

A. Programmierung des Algorithmus

A.1 Statische Bewertung

Die erstellte Funktion bewertet die Nützlichkeit eines Knotens gemäß den Kriterien in A. Wir haben jedoch auch die Wahrscheinlichkeit priorisiert, einen König als Vorteil für den Spieler zu erhalten. Wie erwartet wird der weiße AI-Spieler versuchen, die Funktion zu maximieren und die Knoten zu wählen, die zu den größeren Werten führen.

Diese Funktion wurde in der Datei board.py erstellt, als eine Methode abhängig von der `position` (aktueller Brett).

```
# Method of static evaluation of a
    node - white implements AI
# @white: maximizes the algorithm
# @black: minimizes the algorithm
def evaluate(self):
    # Prioritise becoming a king-> if
        there is more pink kings than
        white kings this operation
        will substract
    # therefore it will be a
        disadvantage for white
    return self.white_left -
        self.pink_left +
        (self.white_kings * 0.5 -
        self.red_kings * 0.5)
```

A.2 Minimax mit Alpha-Beta-Pruning

Der Code für den Minimax-Algorithmus wurde in der Datei minimax.py implementiert, in der Funktion:

```
def minimax(position, depth,
    max_player, game){}
```

Kein Baum ist als Datenstruktur implementiert. Der Algorithmus liest jedoch die möglichen Züge, wie im vorherigen Abschnitt angegeben. `position` bezieht sich auf die tat-

sächliche Instanz des Bretts, wo gespielt wird, und game repräsentiert die Umgebung der Anwendung.

Zuerst prüft es, ob wir das Ende des Baums erreicht haben. Wir überprüfen dies mit depth, die angibt, wie viele Ebenen des Baums wir untersuchen sollten. Je größer die Zahl, desto genauer ist der KI-Agent und desto länger dauert die Berechnung des Ergebnisses.

```
if depth == 0 or position.winner() !=
    None:
    # Return the static evaluation:
    # tuple evaluation and position
        associated alongside
    return position.evaluate(),
        position
```

Wenn der aktuelle Spieler weiß ist, versucht der Algorithmus in den nächsten Schritten, den nächsten Pfad zu maximieren und speichert das Maximum zwischen der aktuellen Bewertung und der vorherigen für jedes Kind. Später ruft es den Algorithmus rekursiv als der andere Spieler auf, um die nächste Ebene des Baums zu analysieren. Die Funktion verwendet eine zeitliche Variable für das Brett (pycopy), um die folgenden Positionen für jeden vorherigen Knoten im Baum zu berechnen. Dies bedeutet ein Speicheropfer, um die Effektivität des Spiels sicherzustellen.

```
# Maximize the score for white (AI)
if max_player:
    # Worst case scenario is negative
        infinity
    maxEval = float('-inf')
    best_move = None
    # For each child of position,
        recalculate the minimax with
        depth - 1
    for move in
        get_all_moves(position,
        WHITE, game):
        # Recursive algorithm to study
            the possible child nodes
            after making that move
        # Since this function returns
            both the optimal
            evaluation and the best
            move, we select only the
            first value
        # Change the color so that now
            we study the other
            alternative for the
            opponent
        evaluation =
            minimax_prun(move,
            depth-1, False, game,
            alpha, beta)[0]
        # Function to maximize
        maxEval = max(maxEval,
            evaluation)
        alpha = max(alpha, evaluation)
        # If the node leads to the
            best position, we save it
        if maxEval == evaluation:
            best_move = move
        # If the node should be prunned
        if beta <= alpha:
            break
    # The best move only matters to
        the program once we have
        reached the depth
    return maxEval, best_move
```

Für den rosa Spieler ist die Vorgehensweise umgekehrt.

A.3 Vorgaben des Designs

Die Anwendung wurde unter Verwendung der Entwurfsmuster Observer und Singleton implementiert. Darüber hinaus kommuniziert die Spielklasse Game mit dem Brett Board, das für die Durchführung der statischen Bewertung sowie für die Entscheidung über die geeigneten Züge für jeden Stein verantwortlich ist.

V. Abschluss

In diesem Dokument wurde die Programmierprozedur für das Damespiel unter Verwendung des Minimax- und Pruning-Algorithmus beschrieben. Die Auswirkung der Wahl des Datentyps auf die Effizienz des Algorithmus und die Vorteile der Verwendung von Pruning gegenüber anderen Alternativen wurden auch diskutiert.

Als zukünftige Arbeit wird jedoch vorgeschlagen, ein in den von [7] vorgeschlagenen Minimax-Algorithmus integriertes neuronales Netzwerk zu implementieren, das es ermöglicht, die Suche im Baum nach dem Training angemessen zu lenken, um die Fälle

zu berücksichtigen, in denen der Gegner die Bewegung nicht ausführt optimaler Weg.

Literatur

[1] Silver et al., Mastering the game of Go with deep neural networks and tree search, Nature, vol. 529, no. 7587, pp. 484-489, 2016. Available: 10.1038/nature16961 [Accessed 14 December 2020].

[2] Rendón, Arturo Alvarado, Matías. (2012). Pattern Recognition and MonteCarloTree Search for Go Gaming Better Automation.

[3] D. Silver et al., A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play, Science, vol. 362, no. 6419, pp. 1140-1144, 2018. Available: 10.1126/science.aar6404 [Accessed 14 December 2020].

[4] Shannon, C. E. (1950). Programming a computer for playing chess. Philisophical Magazine, 41:256

[5] Burgard W. et al., Foundations of Artificial Intelligence: 6. Board Games, Search Strategies for Games, Games with Chance, State of the Art. Albert-Ludwigs-Universit at Freiburg

[6] Viren Shah, J. Min-Max Algorithm, Institute of Technology Nirma University, Min-Mag-Alg

[7] Moriarty, David E. et al., Evolving Neural Networks to Focus Minimax Search, Department of Computer Sciences. The University of Texas at Austin, Austin, TX 78712 Focused Min-Max