OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

Fakultät
Informatik und Mathematik

Projektbericht

zum HSP1 im Wintersemester 2019/20 2019

Implementierung von Reversi mit dem AlphaZero-Ansatz

Autoren: simon1.hofmeister@st.oth-regensburg.de
naddia1.matsko@st.oth-regensburg.de
monika.silber@st.oth-regensburg.de
simon.wasserburger@st.oth-regensburg.de

Leiter: Prof. Dr. rer. nat. Carsten Kern

Abgabedatum: 15.03.2019

Inhaltsverzeichnis

1 Einleitung **1**

2 Related Work **2**
2.1 Monte Carlo Tree Search 2

3 Implementierung **4**
3.1 Monte Carlo Tree Search 4

4 Organisation **6**
4.1 Team und Aufgabenverteilung 6
4.2 Kommunikation 6
4.3 Versionskontrolle 6
4.4 OS, IDE und Programmiersprache 6
4.5 Testumgebungen 6
4.6 Projekt-Dokumentation 6

5 Fazit **7**

Anhang **I**

1 Einleitung

2 Related Work

2.1 Monte Carlo Tree Search

Der MCTS fand als Alternative zum Alpha-Beta-Search Anwendung. Da der Alpha-Beta-Ansatz einen geringen Verzweigungsgrad und eine angemessene Bewertungsfunktion fordert, ist dessen in Anwendung in Brettspielen, die diese Bedingungen nicht erfüllen, ungeeignet. Der MCTS hingegen bewies sich im Umgang mit solchen Situation [CBSS08]. Basierend auf einer Bestensuche und stochastischer Simulation [CBSS08] wählt der MCTS bei jedem Durchlauf den erfolgversprechendsten Knoten als nächsten Spielzug aus. Entsprechend wird ein Baum aufgebaut, in dem ein Knoten einen konkreten Spielzustand widerspiegelt. Dabei werden die drei Schritte Expansion, Simulation und Backpropagation durchgeführt [CBSS08]. Falls der aktuelle Spielzustand noch nicht als Knoten existiert, wird der Baum zunächst expandiert. Um nun die beste Aktion zu ermitteln, werden Spiele ausgehend vom aktuellen Zustand bis hin zum Spielende simuliert. Dabei werden valide Spielzüge zufällig ausgewählt. [Hierbei ist jedoch zu beachten, dass eine reine Zufallsauswahl, die impliziert, dass die Selektion aller Möglichkeiten gleich wahrscheinlich ist, in eher primitivem Spielverhalten resultiert. Mithilfe einer Heuristik können daher aussichtsreichere Spielzüge favorisiert werden.] Innerhalb eines Playouts durchlaufene Nodes werden schließlich aktualisiert, indem vermerkt wird, dass sie einmal mehr besucht wurden und welches Spielergebnis sich ergeben hat [CBSS08].

Anzumerken ist, dass zwei verschiedene Policies genutzt werden. Für die Erweiterung des Baums wird eine Tree Policy angewendet, die besagt, dass entsprechende Blattknoten an bereits vorhandene, unbesuchte Knoten angefügt werden. Des Weiteren legt die Default Policy die Simulation fest. Hierbei wird in einem nichtterminalen Spielzustand, der gewöhnlich dem neu hinzugefügt Blattknoten entspricht, ein zufälliges Spiel durchlaufen, um ein ein Spielergebnis zu ermitteln [BPW+12].

Der MCTS bleibt so lange aktiv, bis er unterbrochen wird, beispielsweise aufgrund von abgelaufener Rechenzeit. Der zu diesem Zeitpunkt als am erfolgreichsten ermittelte Knoten beziehungsweise Spielzug steht daraufhin fest [BPW+12].

Zu einem Knoten gehören der entsprechende Spielzustand, den er widerspiegelt, der Spielzug aus dem er resultierte, sowie der aus Simulationen resultierende Reward und wie oft er besucht wurde [BPW+12].

Entsprechend dem AlphaZero-Ansatz findet bei der Simulation keine zufällige Auswahl des Spielzuges statt, stattdessen wird eine Variante des Upper Confidence Bound (UCB) angewendet. Die konkrete Abwandlung ist der polynomiale UCB applied to Trees (UCT). Dieser errechnet sie wie folgt: Schließlich wird stets der Spielzug ausgewählt, der den maximalen UCT-Wert darstellt.

Die klassische Simulation von Spieldurchläufen entfällt im MCTS vollständig, da der ausgewählte Spielzug an das NN weitergegeben und dort evaluiert wird [SSS+17].

3 Implementierung

3.1 Monte Carlo Tree Search

Um den MCTS für Reversi zu realisieren wurden die Klassen MCTS und Node angelegt. Letztere enhält zwei überladene Konstruktoren zum Anlegen von Wurzel- und Kindknoten. Ein Node enthält zusätzliche Attribute. Sowohl der Elternknoten als auch eine ArrayList vom Typ Node, die die direkten Kinder enthält, werden abgespeichert. Die Anzahl, wie oft ein Knoten besucht wurde, wird in der Variable numVisited hinterlegt. Außerdem gespeichert wird das Ergebnis eines simulierten Spieldurchlaufs in simulationReward. Da der aktuelle Spielzustand durch das Environment, das ebenfalls die derzeitige Repräsentation des Playgrounds beinhaltet, definiert ist, wird dies in Node hinterlegt. Des Weiteren wird in dem Attribut nextPlayer festgehalten, welcher Spieler als Nächstes an der Reihe ist.

Bei der Instanziierung durch die Konstruktoren werden sinnvolle initiale Werte vergeben. numVisited und simulationReward werden auf 0 beziehungsweise 0.0 gesetzt. Für den Wurzelknoten gilt, dass er keinen Parent besitzt, für alle weiteren Knoten wird der Parent übergeben und gesetzt. Die Children werden zunächst durch eine leere Liste initialisiert. Der nextPlayer wird ebenfalls übergeben und gesetzt. Außerdem zu erwähnen ist die Methode calculateUCT(), die den Upper Confidence Bound applied to Trees (UCT) für einen Knoten berechnet. Sie ermittelt zunächst die Exploitation-Komponente, indem sie den simulationReward durch die Anzahl an Besuchen dividiert. Zur Berechnung der Exploration wird die Wurzel aus dem Wert, wie oft der Parent besucht wurde geteilt durch den Wert die oft der aktuelle Knoten besucht wurde +1, gezogen. Um den Kompromiss zwischen diesen beiden Komponenten zu kontrollieren, wird die Exploration mit der A-priori-Wahrscheinlichkeit für einen Zug multipliziert. Diese wird im neuralen Netz trainiert. Hinsichtlich der Klasse MCTS ist festzuhalten, dass diese im der Klasse Agent über den Konstruktoraufruf instanziiert wird. Dieser verlangt das Environment und den Player als Übergabeparameter und legt daraufhin einen neuen Wurzelknoten, sowie eine leere ArrayList vom Typ Node, die die Blattknoten beinhaltet, die im späteren Verlauf simuliert werden müssen. Für die Simulation muss beachtet werden, dass das Environment geklont und somit eine tiefe Kopie erzeugt werden muss, damit der tatsächliche Spielzustand nicht unbeabsichtigt manipuliert wird. Dabei ist festzuhalten, dass dies zweimal stattfindet. Einmal, wenn ein neuer Baum aufgebaut wird, somit erhält der neue Wurzelknoten und ebenfalls jedes seiner Kinder jeweils einen eigenen Klon. Für die Kinderknoten gilt, dass diese ihre Environment-Instanz an ihre Kinder weitergeben und diese somit innerhalb derselben Instanz agieren. Um den MCTS zu starten, wird die Methode searchBestTurn() aufgerufen. Diese expandiert zunächst den Wurzelknoten, indem sie alle im aktuellen Zustand möglichen validen Züge ermittelt und durch diese iteriert. Die Methode getPossibleTurns() gibt diese zurück. Sie iteriert über das gesamte Spielfeld und prüft dabei mithilfe der Methode validateTurnPhase1 im Environment, an welcher Stelle ein gültiger Zug gemacht werden kann. Für jeden

dieser Züge wird der Kinderknoten angelegt, sowie als unbesuchte Blattknoten abspeichert. Außerdem wird ermittelt, welcher Spieler als Nächster einen Zug machen darf und der simulierte derzeitige Spielzustand anhand des Zuges aktualisiert. Daraufhin werden die unbesuchten Blattknoten, die am Anfang den Kinderknoten der Wurzel entsprechen, durchlaufen. Dabei wird in jedem dieser eine Simulation gestartet, die einen Spielverlauf bis zum Spielende anhand zufällig ausgewählter möglicher Züge durchspielt. Mithilfe der rekursiven Funktion simulate() erfolgt die Simulation eines Spiels, sowie die Backpropagation des Ergebnisses bei Spielende. Sie enthält letztendlich den Reward als Rückgabewert. Ein Zufallszug wird durch eine Instanz der Java-Klasse Random generiert. Hierbei wird ein zufälliger Integer erzeugt, der durch eine Modulo-Operation auf den Größenbereich abgestimmt wird, der dem der Anzahl der möglichen Züge entspricht. Die resultierende Zahl gibt den auszuwählenden Zug innerhalb der ArrayList an. Für einen neu gewählten Zug wird ein neuer Knoten angelegt und dessen Variablen, die zur Berechnung des UCT relevant sind, aktualisiert. Der Reward wird durch den rekursiven Aufruf der Funktion erhöht. Die Anzahl an Besuchen wird jeweils um den Wert eins erhöht. Wenn keine weiteren validen Folgezüge ermittelt werden können, bedeutet das das Spielende und der Reward für die Spielausgang wird anhand der Funktion rewardGameState() berechnet. Diese erhält als Parameter das Environment, sowie den Spieler, für den der Reward kalkuliert werden soll. Indem der gesamte Playground durchlaufen und gezählt wird, wie viele Steine vom übergebenen Spieler enthalten sind, errechnet sich die Bewertung des Spiels. Abschließend werden die Werte für Anzahl Besuche und Reward ebenfalls im Wurzelknoten aktualisiert.

4 Organisation

4.1 Team und Aufgabenverteilung
4.2 Kommunikation
4.3 Versionskontrolle
4.4 OS, IDE und Programmiersprache
4.5 Testumgebungen
4.6 Projekt-Dokumentation

5 Fazit

Literatur

[BPW+12] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, Samthrakis S., and S. Colton. A survey of monte carlo tree search methods. In *IEEE Transactions on Computational Intelligence and AI in games*, volume 4(1), pages 1–43, März 2012.

[CBSS08] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, Oktober 2008.

[SSS+17] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. In *nature*, volume 550(7676), pages 354–359, März 2017.

Anhang