Technology
Arts Sciences
TH Köln

KI-Agenten für das Spiel 2048: Untersuchung von Lernalgorithmen für nichtdeterministische Spiele

Bachelorarbeit

ausgearbeitet von

Johannes Kutsch

11090517

zur Erlangung des akademischen Grades

Bachelor of Science

vorgelegt an der

Technischen Hochschule Köln

Campus Gummersbach

Fakultät für Informatik und

Ingenieurwissenschaften

Prüfer:

Prof. Dr. rer. nat. Wolfgang Konen,

Prof. Dr. Martin Eisemann

18. August 2017

Zusammenfassung

Die nachfolgende Bachelorarbeit zeigt die Ergebnisse einiger Untersuchungen von Lernalgorithmen für das Spiel 2048. Dabei wurden vier Agenten, ein Monte Carlo (MC) Agent, ein Monte Carlo Tree Search (MCTS) Agent, ein Monte Carlo Tree Search Expectimax (MCTSE) Agent und ein Temporal Difference NTuple (TD-NTuple) Agent, genauer untersucht. Ziel dieser Arbeit ist es, die Stärken und Schwächen der Agenten im Hinblick auf 2048 genauer zu untersuchen und allgemeine Erkenntnisse abzuleiten.

MC Agent: Die Ergebnisse des MC Agenten können, trotz seiner Einfachheit, oder vielleicht grade Wegen seiner Einfachheit, mit den Ergebnissen der anderen getesteten Agenten mithalten. Dabei benötigt der MC Agent die niedrigste Rechenleistung aller Agenten.

MCTS Agent: Aufgrund der nichtdeterministischen Anteile im Spiel 2048 ist die Grundform des MCTS-Algorithmus nicht geeignet, da sie die Zufallselemente des Spieles 2048 nicht richtig abbilden kann. Um die Zufallselemente richtig abbilden zu können, wurde ein MCTSE Agent entwickelt und implementiert.

MCTSE Agent: Durch den MCTSE Agenten konnte das Problem des MCTS Agenten gelöst werden. Der Agent erreicht jedoch kaum bessere Ergebnisse als ein MC Agent und benötigt hierfür mehr Rechenleistung.

TD-NTuple Agent: Mit den zur Verfügung stehenden Mitteln war es nicht möglich einen erfolgreichen TD-NTuple Agenten zu trainieren. Der beste trainierte Agent ist in etwa so gut wie ein MCTS Agent. Durch die Trainingsversuche konnte jedoch gezeigt werden, wie rechenaufwändig und komplex es ist einen erfolgreichen TD-NTuple Agenten für das Spiel 2048 zu trainieren.

Durch die Nutzung der Agenten kann das Ziel des Spieles 2048, das Erstellen einer Kachel mit dem Wert 2048, meistens erreicht werden, allerdings erreichen diese Agenten, verglichen mit den besten Agenten, eine eher geringe Punktzahl. Zusätzlich konnten folgende wesentliche Erkentnisse ermittelt werden:

1: Aufgrund der nichtdeterministischen Anteile sowie des großen Spielbaumes des Spieles 2048 ist die Sicherheit (Certainty) bei der Auswahl der nächsten

Aktion oft sehr gering. Diese konnte durch die Nutzung eines Majority Votes mehrerer Agenten leicht verbessert werden.

2: Durch Fachwissen in Form von Heuristiken konnten die Ergebnisse des MCTSE Agenten leicht verbessert werden. Da das genaue Einstellen der Heuristiken erst zum Ende der Arbeit in Form einer Konzeptstudie durchgeführt wurde, ist nicht auszuschließen, dass das Ergebnis in Zukunft noch weiter verbessert werden kann.

3: Durch die Nutzung des in Java integrierten `ExecutorService` konnten einige der getesteten Agenten erfolgreich auf mehrere Kerne aufgeteilt werden. So kann die Rechenleistung zeitgemäßer Mehrkernarchitekturen besser genutzt werden, wodurch die Geschwindigkeit dieser Agenten deutlich ansteigt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

MC Monte Carlo . . . 2

MCTS Monte Carlo Tree Search . . . 2

MCTSE Monte Carlo Tree Search Expectimax . . . 2

TD-NTuple Temporal Difference NTuple . . . 2

Certainty Sicherheit . . . 2

KI künstliche Intelligenz . . . 10

GBG General Board Gaming . . . 14

Rolloutdepth Spieltiefe . . . 19

Treedepth Baumtiefe . . . 27

Exploitation Ausbeutung . . . 28

Exploration Erkundung . . . 28

UCT Upper Confidence bounds applied to Trees . . . 29

UCTN UCT Normalised . . . 41

Maxnodes maximale Anzahl an Expectimaxknoten . . . 43

CMA-ES Covariance Matrix Adaptation Evolution Srategy . . . 67

1 Einleitung

1.1 Motivation

Das Erstellen von künstlichen Intelligenzen (KI's) für Brettspiele ist eine faszinierende Aufgabe. Ein Problem, wie etwa der nächste Zug in einem Schachspiel, das für einen Menschen sehr komplex erscheint, kann von einer guten KI in kürzester Zeit nahezu perfekt gelöst werden. Das Entwickeln einer guten KI ist meist ein sehr zeitaufwändiger und schwieriger Prozess. Es ist deshalb wünschenswert, nicht für jedes neue Problem eine neue, spezialisierte, KI entwickeln zu müssen, sondern einmal eine KI zu entwickeln, welche universal einsetzbar ist und bei neuen Spielen nur gering oder sogar gar nicht angepasst werden muss. Diese Arbeit beschäftigt sich hauptsächlich mit einer Variante dieser Agenten, welche die MC Methode als Grundlage nutzt. Durch die Wahl des Spieles 2048, welches einen sehr einfachen Satz von Regeln aufweist, jedoch einen um ein vielfaches größeren Spielbaum als komplexe Spiele wie Go besitzt, in Kombination mit Monte Carlo Algorithmen, entsteht ein einzigartiges, vielversprechendes und noch wenig erforschtes Problemfeld. Dabei stellen sich vor allem folgende Fragen:

1: Gibt es Problematiken, die für MC-basierte Agenten beim Lösen nichtdeterministischer Spiele auftreten, und wie können diese gelöst werden?

2: Welche Ergebnisse können mit MC-basierten Agenten mit welchem Rechenaufwand für das Spiel 2048 erreicht werden?

3: Inwieweit können diese Ergebnisse noch verbessert werden, etwa durch die Anwendung von Fachwissen oder aber auch durch allgemeine Methoden?

4: Welche Ergebnisse erreichen MC-basierte Agenten im Vergleich zu anderen Agenten, wie etwa einem TD-Tuple-Agent,[1] und worin bestehen die Vor- und Nachteile der MC Agenten gegenüber diesen?

[1] Siehe Kapitel 6, S. 55.

1.2 Verwandte Arbeiten

Im Folgenden werden einige Arbeiten, in welchen verschiedene Agenten für das Spiel 2048 getestet wurden, vorgestellt.

1: Den bisher größten Erfolg beim Lösen von 2048 gelang Jaskowski [2016] mit einem durchschnittlichen Punktestand von `609.104` Punkten am Spielende. Dazu wurde Delayed Temporal Coherence Learning, Multi-State Weight Promotion, Redundant Encoding und Carousel Shaping eingesetzt. Bei diesem Ergebnis ist jedoch zu beachten, dass der Agent auf einem Hochleistungsrechner mit 24 Kernen über einen Zeitrum von bis zu 6 Tagen trainiert wurde.

2: Ein leicht schlechteres Ergebnis erreichte Chao-Chin et al. [2014] durch die Kombination von Machine Learning und einer Game Tree Search, mit einem durchschnittlichen Punktestand von `291.597` Punkten. Bemerkenswert ist, dass die von Chao-Chin et al. [2014] entwickelte KI ungefähr 800 Aktionen pro Sekunde ausführt, deutlich mehr als die meisten anderen für das Spiel 2048 getesteten Agent.

3: Durch die Nutzung von einer Expectimax Optimierung in Verbindung mit mehreren Heuristiken gelang es Xiao [2014] einen Agenten zu entwickeln, der im Median 387.222 Punkte erreicht. Dabei wird das Feld mit dem Wert `32.768` in jedem dritten Spiel erreicht.

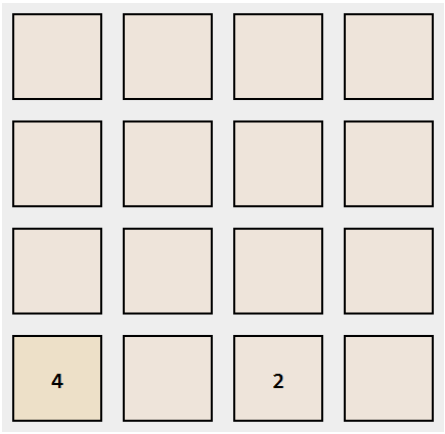
4: Ronenz [2014] untersuchte als erster die Performance des MCTS Agenten bei dem Spiel 2048. Er erzielte dabei ähnliche Ergebnisse wie mit den MC Agenten in dieser Arbeit erzielt wurden.

2 Überblick

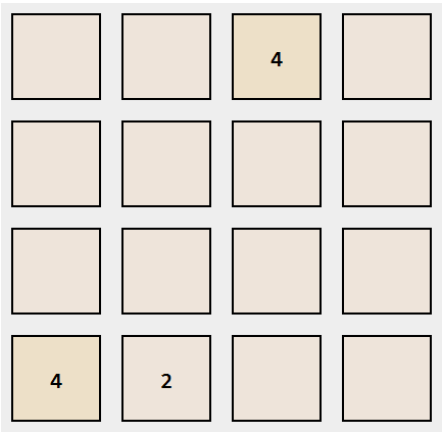
Im folgenden Kapitel wird ein kurzer Überblick über die Grundlagen der Arbeit gegeben. Dies umfasst eine Erläuterung der Funktionsweise des Spiele 2048, eine kurze Einführung in das, zum Implementieren der Agenten genutzte, General Board Gaming Framework, eine Zusammenfassung der untersuchten Agenten und eine Beschreibung der genutzten Evaluationsmethoden.

2.1 Das Spiel 2048

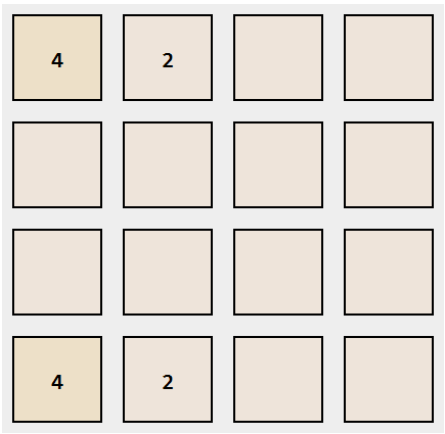
2048 ist ein nichtdeterministisches, rundenbasiertes Rätselspiel, welches von einem einzelnen Spieler auf einem Computer oder Mobilgerät im Browser oder als App gespielt werden kann. Das Spiel wurde ursprünglich von Gabriele Cirulli entwickelt, welcher das Spiel 1024 von Veewo Studio als Inspiration verwendete [Cirulli 2014]. Das Spiel 2048 kann auf der Webseite `https://gabrielecirulli.github.io/2048/` gespielt werden.
2048 wird auf einem Spielfeld, welches aus 4x4 Kästchen besteht, gespielt. Auf dem Spielfeld liegen Kacheln welche Zweierpotenzen (ausgenommen der 1) darstellen. Zum Spielbeginn werden zwei Kacheln zufällig auf dem Spielfeld platziert. Diese haben jeweils in 90% der Fälle den Wert 2 und in 10% der Fälle den Wert 4 (Abbildung 1a). Die Kacheln können gemeinsam in eine von 4 Richtung verschoben werden. Nachdem die Kacheln verschoben wurden, erstellt sich auf einem zufälligen, freien Feld eine neue Kachel. Diese hat ebenfalls in 90% der Fälle den Wert 2 und in 10% der Fälle den Wert 4 (Abbildung 1b & Abbildung 1c). Werden zwei Kacheln mit derselben Zahl aufeinander geschoben, kombinieren sie sich zu einer Kachel mit ihrer Summe (Abbildung 1d). Der Spieler erhält für das Kombinieren von Kacheln Punkte welche dem Wert der neuen Kachel entsprechen. Sobald keine Aktion mehr möglich ist, hat der Spieler verloren und das Spiel wird beendet.
Ziel des Spiels ist das Erstellen einer Kachel mit der Zahl 2048. Das Spiel endet jedoch nicht schon mit der Erstellung dieser Kachel, sondern erlaubt es dem Spieler noch weiterzuspielen, um eine möglichst hohe Punktzahl zu erreichen. Die maximal mögliche Punktzahl beträgt dabei 3.932.156 Punkte [Asusfood 2015]. Aktuelle AI's erreichen maximal Punktzahlen von etwa 840.000 Punkten [Olson 2015].



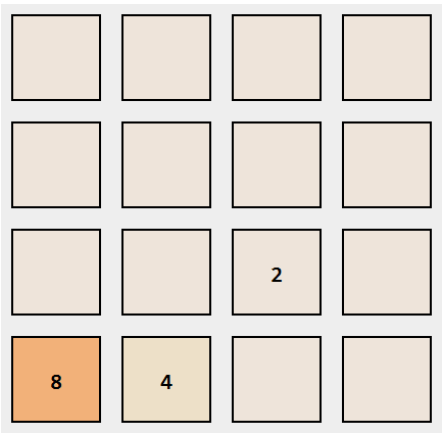
(a) Ein initialer Spielzustand s0 des Spieles 2048.



(b) Der Spielzustand s1 nachdem die Aktion links für den Spielzustand s0 ausgeführt wurde. In der ersten Zeile ist eine neue Kachel mit dem Wert 4 entstanden.



(c) Der Spielzustand s2 nachdem die Aktion links für den Spielzustand s1 ausgeführt wurde. In der ersten Zeile ist eine neue Kachel mit dem Wert 2 entstanden.



(d) Der Spielzustand s3 nachdem die Aktion runter für den Spielzustand s2 ausgeführt wurde. Die Kacheln mit den Werten 4 und 2 haben sich kombiniert und Kacheln mit den Werten 8 und 4 erzeugt. Der Spieler erhält als Belohnung 12 Punkte. In der dritten Zeile ist eine neue Kachel mit dem Wert 2 entstanden.

Abbildung 1: Beispielhafter Spielablauf des Spieles 2048.

Eine große Herausforderung bei der Entwicklung einer KI für 2048 ist der Zufallsfaktor beim Erstellen einer neuen Kachel nach jeder Aktion. Jede Aktion kann so zu einer Vielzahl an verschiedenen Spielzuständen führen. Dies führt dazu, dass nicht vorausgesagt werden kann, wie genau ein Spielbrett nach einer Aktion aussieht. Eine weitere Schwierigkeit entsteht durch die Menge der möglichen Aktionssequenzen. Auf den ersten Blick scheint das Spiel relativ simpel zu sein, da es maximal vier Aktionen (hoch, runter, links und rechts) gibt, zwischen denen sich ein Agent entscheiden muss. Da jedoch nach jeder Bewegung eine neue Kachel auf einem freiem Feld entsteht, kann jede dieser Aktionen zu einer Vielzahl von Spielzuständen führen, welche alle eine neue Aktionssequenz initiieren. Die Anzahl an Aktionssequenzen, also die Größe eines Spielbaumes, kann durch die Funktion b^d angenähert werden. Dabei steht `b` für die Breite des Spieles (Anzahl an möglichen Aktionen pro Spielzustand) und `d` für die Tiefe des Spieles (Anzahl an Zügen). Für das Spiel Schach ($b \approx 35$, $d \approx 80$) ist $b^d \approx 10^{108}$ und für das Spiel Go ($b \approx 250$, $d \approx 150$) ist $b^d \approx 10^{359}$ [Silver et al. 2016, S. 1f]. Bei 2048 werden bereits bei einem Spiel von nur etwa 80.000 Punkten Werte von $b \approx 32$ und $d \approx 3.650$ erreicht. Somit umfasst der Spielbaum für 2048 $\approx 10^{5.493}$ Aktionssequenzen. [Kutsch 2017, S. 5-7]

2.2 General Board Gaming Framework

Das General Board Gaming (GBG) Framework wurde von Konen [2017b] in der Programmiersprache Java entwickelt, um eine einheitliche Plattform zu bieten, auf welcher Agenten und Spiele aufeinandertreffen können.

> *"A common problem in GBG is the fact, that each time a new game is tackled, the AI developer has to undergo the frustrating and tedious procedure to write adaptations of this game for all agent algorithms. "* [Konen 2017b, S. 3]

Für die Implementation eines neuen Spieles in das GBG Framework müssen die Interfaces `StateObservation` und `GameBoard` implementiert werden. Das Interface `GameBoard` kümmert sich dabei um die graphische Darstellung des Spielbrettes, während das Interface `StateObservation` die Spiellogik übernimmt. Es stellt also die verfügbaren Aktionen bereit, weiss wann das Spiel vorbei ist und kann das Spielbrett von einen Spielzustand in den nächsten überführen. [Konen 2017b, S. 4].

Um einen neuen Agent zu implementieren muss das Interface `PlayAgent` implementiert werden. Das Interface `PlayAgent` repräsentiert einen Agent welcher ein Spiel im GBG Framework spielen kann [Konen 2017b, S. 4]. Im GBG Framework waren bereits vor Beginn der Arbeit einige Agenten implementiert, unter anderem eine Version des MCTS Agenten und einer Version des TD-NTuple Agenten. [Kutsch 2017, S. 8]

2048 im GBG Framework

Bei dem Spiel 2048 handelt es sich zwar nicht um ein Brettspiel im klassischem Sinn, wie etwa Schach, Vier Gewinnt oder Tic Tac Toe, allerdings besitzt es trotzdem alle Kriterien, die ein Spiel erfüllen muss, damit es in das GBG Framework integriert werden kann. 2048 ist, wie die meisten klassischen Brettspiele, rundenbasiert und wird nicht in Echtzeit gespielt, das heißt es wird eine Aktion ausgeführt, woraufhin das Spielbrett entsprechend modifiziert wird und auf die nächste Aktion gewartet wird. Es ist für den Spielablauf irrelevant, wie viel Zeit zwischen den einzelnen Aktionen vergeht. Außerdem können alle verfügbaren Aktionen für einen Spielzustand zu jedem Zeitpunkt ermittelt werden. [Kutsch 2017, S. 9]

2.3 Untersuchte Agenten

Im Verlauf der Arbeit wurden vier Agenten für das Spiel 2048 getestet. Drei dieser Agenten basieren auf der Monte Carlo Methode. Zusätzlich wurde getestet welche Ergebnisse mit dem bereits im GBG Framework integrierten TD-NTuple Agent erreicht werden können.

MC Agent: Der MC Agent wurde während den Anfängen der Arbeit als Übung entwickelt und stellt eine simple Form des MCTS Agenten dar. Ursprünglich war es nicht geplant, den MC Agenten näher zu untersuchen, dies änderte sich jedoch, als mit ihm bessere Ergebnisse als mit dem MCTS Agenten erzielt werden konnten. Der MC Agent erreicht im Schnitt `51521` Punkte am Spielende.

MCTS Agent: Mithilfe des MCTS Agenten wurden bereits übermenschliche Leistungen in Schach und ein schwaches Amateurlevel in Go erreicht [Silver et al. 2016, S. 2]. Prinzipiell sollte er somit für Spiele mit einem großen Spielbaum, wie 2048, gut geeignet sein. Während der Bearbeitung stellte sich jedoch heraus, dass der MCTS Agent für nichtdeterministische Spiele erhebliche Schwächen[2] aufweist. Der MCTS Agent erreicht im Schnitt `34713` Punkte am Spielende.

MCTSE Agent: Der MCTSE Agent wurde im Rahmen dieser Arbeit neu entwickelt um die Probleme des MCTS Agenten mit nichtdeterministischen Spielen zu beheben. Er basiert auf dem MCTS Agenten und erweitert diesen um eine Chance-Komponente, welche sich um den Zufallsfaktor von nichtdeterministischen Spielen kümmert. Der MCTSE Agent erreicht im Schnitt `56989` Punkte am Spielende.

TD-NTuple Agent: Im Gegensatz zu den anderen getesteten Agenten muss der TD-NTuple Agent vor seiner Nutzung trainiert werden. Der TD-NTuple Agent wurde untersucht um zu ermitteln welche Ergebnisse mit einem Agenten, der nicht auf der Monte Carlo Methode basiert, erreicht werden können und wie hoch der dafür benötigte Trainingsaufwand ist. Der TD-NTuple Agent erreicht im Schnitt etwa `29900` Punkte am Spielende.

Zwei der untersuchten Agenten, der MCTS und der TD-NTuple Agent, waren bereits zum Beginn der Bearbeitung im GBG Framework implementiert und konnten so ohne größeren Aufwand untersucht werden. Der MC und der MCTSE Agent wurden neu entwickelt und mussten somit noch in das GBG Framework implementiert werden. Durch die Integration in das Framework können sie nun ohne größeren Aufwand für verschiedene, im GBG Framework implementierte, Spiele genutzt werden.

[2] Siehe Kapitel 4.4, S. 35.

2.4 Evaluationsmethoden

Im Folgenden wird ein kurzer Überblick über die beiden Evaluationsmethoden, welche zur Evaluation der Agenten entwickelt und eingesetzt wurden, gegeben. Beim betrachten der Evaluationsergebnisse sollte beachtet werden, dass diese aufgrund der Natur des Spiele 2048 einer hohen Schwankung[3] unterliegen. Wegen der Dauer einzelner Evaluationen ist es nicht möglich gewesen die Anzahl an Evaluationen weiter zu erhöhen um eine genaueres Ergebnis zu erlangen. Die Ergebnisse geben somit meistens eher eine grobe Richtlinie an.

Methode 1: Bei der ersten Methode werden mehrere Spiele nacheinander gespielt und anschließend wird die durchschnittlich erreichte Punktzahl dieser Spiele dazu genutzt, den Agenten zu bewerten. So kann man sich relativ einfach einen Eindruck darüber verschaffen, wie gut ein Agent das Spiel 2048 spielt und wo er im Vergleich zu anderen Agenten steht.[4]

Methode 2: Bei der zweiten Evaluationsmethode wird ermittelt, wie sicher sich ein Agent bei der Auswahl der nächsten Aktion ist. Idealerweise wählt ein Agent für den selben Spielzustand immer die selbe, vermutlich beste, Aktion aus. In diesem Fall würde die Sicherheit (Certainty) des Agentens für diesen Spielzustand `100%` betrage. Das andere Extrem wäre eine Certainty von `0%`, welche angibt, dass für einen Spielzustand jede verfügbare Aktion gleich oft ausgewählt wurde. Der Agent verhält sich also äquivalent zu einem Zufallsagenten.[5]

Um die Certainty zu ermitteln wurden einzelne Spielzustände mehrmals untersucht. Die Ergebnisse der Untersuchungen wurden anschließend anhand mehrere Kriterien, wie der Anzahl an verfügbaren Aktionen und der Anzahl freier Felder, gruppiert. Die Certainty einer Gruppe entspricht der durchschnittlichen Certainty aller Ergebnisse dieser Gruppe. Damit die untersuchten Spielzustände möglichst realistisch sind, wurden sie während mehrerer normaler Spiele gesammelt und für die spätere Evaluation gespeichert.

Weitere Informationen zur Berechnung der Certainty befinden sich im Anhang auf Seite 77.

[3] In der Regel beläuft es sich hierbei um etwa `+-1000 Punkte`.
[4] Dargestellt z.B. in Abbildung 4, S. 24 und Abbildung 5, S. 25.
[5] Dargestellt z.B. in Abbildung 6, S. 26 und Abbildung 23, S. 88.

3 Monte Carlo Agent

Im folgenden Kapitel wird eine kurze Einführung in die Monte Carloe Methode, der Basis aller Monte Carlo basierten Agenten, gegeben. Anschließend wird die Funktionsweise des MC Agenten erläutert und es wird evaluiert mit welchen Einstellungen der Agent das beste Ergebnis für das Spiel 2048 erreichen kann.

3.1 Monte Carlo Methode

Die von Nick Metropolis benannte Monte Carlo (MC) Methode wurde erstmals 1953 von ihm auf dem MANIAC eingesetzt. Die MC Methode ist eine Anwendung der Gesetze der Wahrscheinlichkeit und Statistik und wurde ursprünglich zum Lösen von Physikproblemen eingesetzt. [Anderson 1986, S. 96]

> *"The essence of the method is to use various distributions of random numbers, each distribution reflecting a particular process in a sequence of processes such as the diffusion of neutrons in various materials, to calculate samples that approximate the real diffusion history."* [Anderson 1986, S. 96]

In der KI-Entwicklung wird die MC-Methode eingesetzt um mit wenig Programmieraufwand und Fachwissen über ein Spiel, welches ein Agent lösen soll, zu einem guten Ergebnis zu gelangen. Das Spiel wird, ausgehend vom aktuellen Spielzustand, durch zufällige Züge zu Ende gespielt. Dies passiert mehrmals, oft mehrere tausend Mal. Aufgrund der Ergebnisse dieser Zufallszüge wird dann die nächste Aktion eines Agent ermittelt. [Kutsch 2017, S. 8]

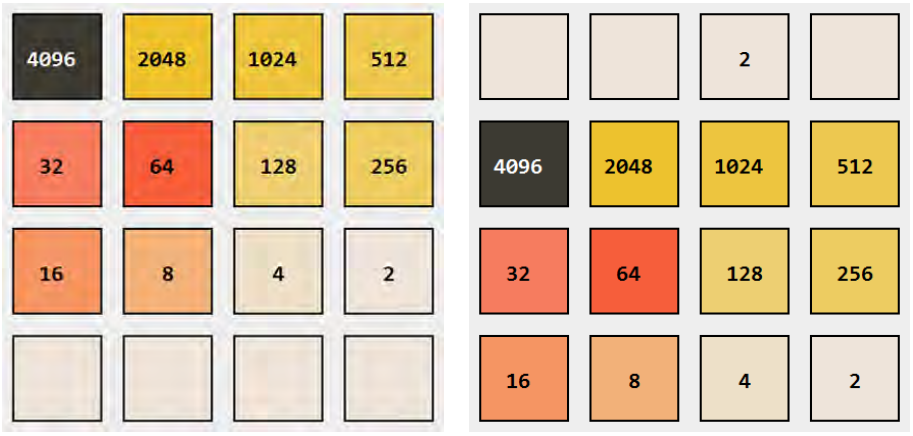
3.2 Funktionsweise des Monte Carlo Agenten

Algorithm 1 MC

```
1:  function MC(s_0)
2:      while within computational budget do
3:          for each a ∈ A(s_0) do
4:              s ← s_0
5:              s ←ADVANCE(s; a)
6:              s ←ROLLOUT(s)
7:              V(a) ← V(a) + V(s)
8:      return argmax_{a∈A(s_0)} V(a)
9:
10: function ROLLOUT(s)
11:     while NOT(s terminal OR maxMoves) do
12:         choose a ∈ A(s) uniformly at random
13:         s ←ADVANCE(s; a)
14:     return s
```

Die Funktionsweise des MC Agent kann grob in zwei Phasen unterteilt werden. In der ersten Phase, der Simulation, wird das Spiel mehrmals durch Zufallsbewegungen bis zum Spielende oder einer vorgegebenen Spieltiefe (Rolloutdepth) gespielt. In der zweiten Phase, der Evaluation, werden die gesammelten Informationen der Zufallsspiele evaluiert und es wird anhand dieser Daten die nächste Aktion des Agenten bestimmt.
Wenn der MC Agent aufgefordert wird, den nächsten Zug für einen Spielzustand zu ermitteln, wird als erstes überprüft ob es für den aktuellen Spielzustand nur eine mögliche Aktion gibt. Wenn dies der Fall ist kann diese direkt zurückgegeben werden, die Aktion ist die einzige Möglichkeit in den nächsten Spielzustand zu gelangen.[6] Ein Durchlauf des kompletten Suchprozesses ist somit nicht nötig. [Kutsch 2017, S. 10]

[6] Siehe Abbildung 2, S. 20.



(a) Der Spielzustand s0 für den nur die Aktion runter möglich ist, da sich der Spielzustand durch die Aktionen links, hoch und rechts nicht ändern würde.

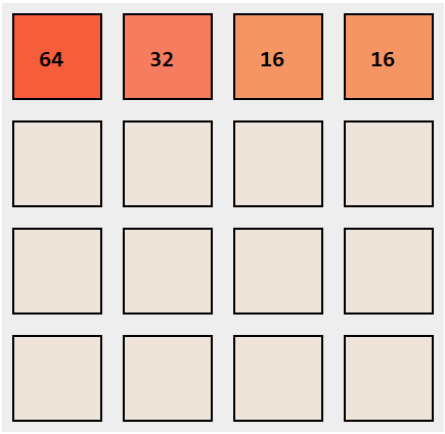
(b) Der Spielzustand s1 nachdem die Aktion runter ausgeführt wurde. In der ersten Zeile ist eine neue Kachel mit dem Wert 2 entstanden. Es sind nun die Aktionen links, hoch und rechts möglich.

Abbildung 2: Ein Spielzustand für den nur eine Aktion möglich ist.

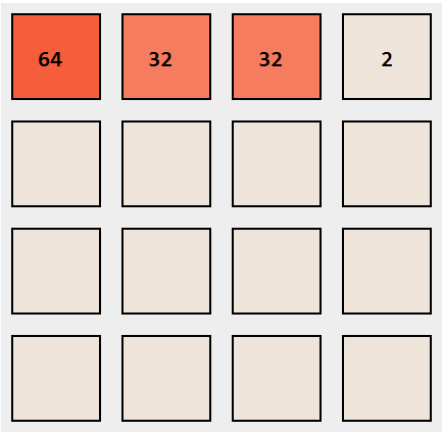
Phase 1: Simulation

Falls es für einen Spielzustand mehr als eine mögliche Aktion gibt, beginnt nun die erste Phase. Der aktuelle Spielzustand wird für jede mögliche Aktion einmal kopiert. Anschließend wird auf diesen Kopien jede mögliche Aktion einmal ausgeführt[7]. Nun liegen mehrere Spielzustände vor, welche jeweils einen möglichen Spielzustand nach jeder möglichen Aktion repräsentieren. Im Fall von `x` möglichen Aktionen liegen also `x` neue Spielzustände vor.
Nachdem die Spielzustände vorbereitet wurden, werden diese an einen Random Agent übergeben. Dieser führt solange Zufallsaktionen aus, bis das Spiel entweder beendet wurde oder die Anzahl an Zufallszügen der vorgegebenen Spieltiefe (Rolloutdepth) entspricht. Eine Begrenzung der Anzahl an Zufallszügen durch eine Rolloutdepth ist sinnvoll um die Rechenzeit zu begrenzen und sicherzustellen, dass der Spielzustand relevant bleibt.[Kutsch 2017, S. 11]

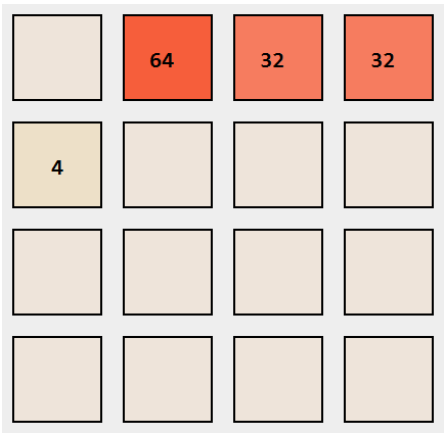
[7] Siehe Abbildung 3, S. 21.



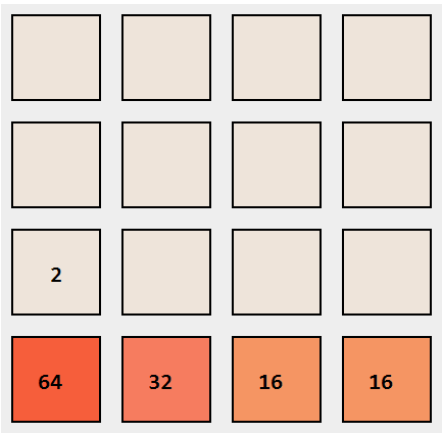
(a) Ein initialer Spielzustand s0 der an den MC Agent übergeben wird. Es sind die Aktionen links, rechts und runter möglich.



(b) Der Spielzustand s1 nachdem die Aktion links für den Spielzustand s0 ausgeführt wurde. In der ersten Zeile ist eine neue Kachel mit dem Wert 2 entstanden.



(c) Der Spielzustand s2 nachdem die Aktion rechts für den Spielzustand s0 ausgeführt wurde. In der zweiten Zeile ist eine neue Kachel mit dem Wert 4 entstanden.



(d) Der Spielzustand s3 nachdem die Aktion runter für den Spielzustand s0 ausgeführt wurde. In der dritten Zeile ist eine neue Kachel mit dem Wert 2 entstanden.

Abbildung 3: Ein Spielzustand wird an den MC Agenten übergeben.

Phase 2: Evaluation

Nachdem die erste Phase abgeschlossen wurde, folgt nun die Evaluation der Spielzustände. Dazu werden die Spielzustände zunächst anhand der Aktion gruppiert, welche in der ersten Phase als erstes für sie ausgeführt wurde. Danach wird für jede dieser Gruppen die durchschnittliche Game Score aller Spielzustände ermittelt. Die Gruppe, welche die höchste durchschnittliche Game Score aufweist, bestimmt nun die nächste Aktion des Agenten. Für 2048 hängt die Game Score meistens von dem aktuellen Punktestand des Spieles ab, Heuristiken können diese jedoch verändern oder überschreiben. Weitere Informationen zur Game Score befinden sich im Anhang auf Seite 78.
Während der Entwicklung wurde außerdem die Möglichkeit in Betracht gezogen, anstelle der Game Score die durchschnittlich benutzte Rolloutdepth als Grundlage für die Ermittlung der nächsten Aktion zu benutzen. Die Aktion, welche die höchste durchschnittliche Rolloutdepth aufweist, ist die Aktion, welche das längste Spiel hervorbringt und somit meistens auch den höchsten Punktestand erreicht. Diese Möglichkeit erwies sich aus drei Gründen als suboptimal und wurde deshalb relativ schnell wieder verworfen. [Kutsch 2017, S. 11-13]

1: Bei einer geringen maximalen Rolloutdepth weisen viele Spielzustände die selbe Rolloutdepth auf und liefern so unabhängig davon, ob die Aktionen, die zu den Spielzuständen führen, gut oder schlecht sind, das selbe Ergebnis. [Kutsch 2017, S. 13]

2: Es ist deutlich schwieriger, den Spielausgang bei nicht zu Ende gespielten Spielzuständen durch Heuristiken abzuschätzen, da diese nicht direkt im Spielzustand die Score modifizieren könnten sondern in den Agenten implementiert werden müssten. [Kutsch 2017, S. 13]

3: Aufgrund der Natur des Spieles 2048 steigen die Punkte relativ konstant mit der Anzahl an Spielzügen an, also wird unabhängig davon ob die Game Score oder die Rolloutdepth genutzt wird, meistens die selbe Aktion ausgewählt. [Kutsch 2017, S. 13]

3.3 Evaluation des Monte Carlo Agenten und Analyse der Parameter

Der MC Agent erreicht mit optimalen Einstellungen durchschnittlich ungefähr `51521` Punkte am Spielende. Im Gegensatz zu den anderen untersuchten Agenten, besitzt der MC Agent eher wenige Parameter die justiert werden können, es können nur die Tiefe der Rolloutdepth sowie die Anzahl an Iterationen verändert werden.

Iterationen: Die Iterationen geben an, wie oft jede verfügbare Aktionen eines Spielzustandes untersucht werden soll. Bei `1000` Iterationen wird eine gute Balance zwischen benötigter Simulationsdauer und dem Punktestand am Spielende erreicht.

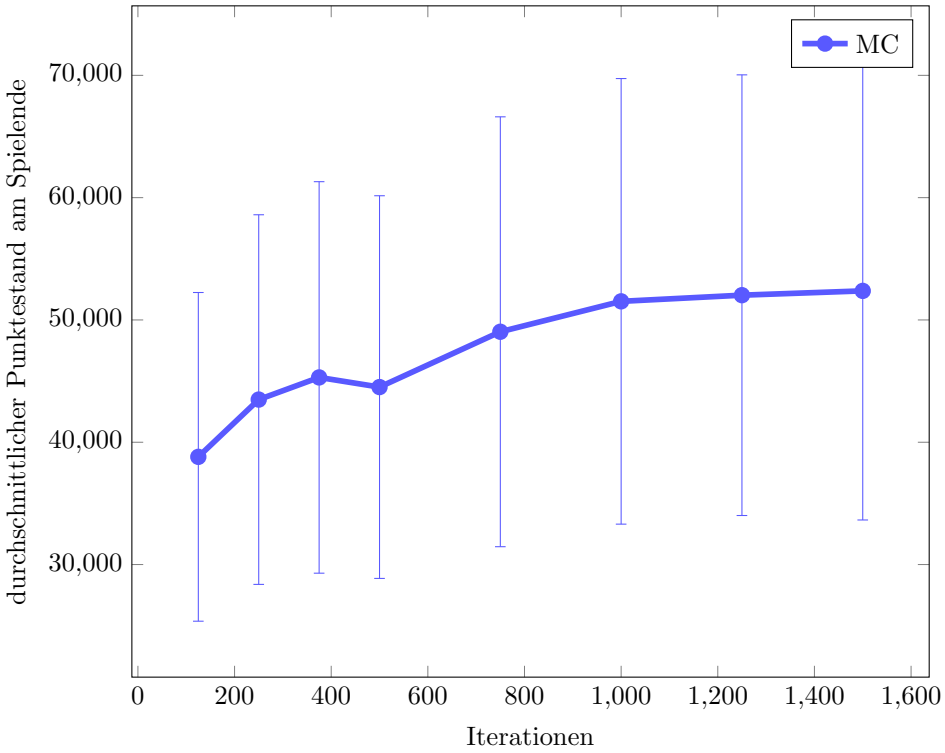
Rolloutdepth: Die Rolloutdepth gibt an, wie viele Aktionen der Random Agent pro Spielzustand maximal ausführen soll. Der MC Agent erreicht optimale Ergebnisse bei einer Rolloutdepth von etwa `20`.

Iterationen

Anders, als bei vielen anderen Parametern, bringt eine Steigerung der Iterationen immer eine Verbesserung des Ergebnisses mit sich[8]. Durch mehr Iterationen wird die Chance erhöht, dass ein guter Spielzug erkannt wird, wodurch die Certainty des Agenten und der Punktestand am Spielende steigen. Da eine Erhöhung der Iterationen jedoch auch die Simulationsdauer erhöht, muss ein Kompromiss zwischen einem guten Ergebnis und einer akzeptablen Simulationsdauer eingegangen werden.
Um die Simulationsdauer gering zu halten, ist es also wichtig abzuschätzen, wie viele Iterationen ein Agent benötigt, um einen Spielzustand erfolgreich zu evaluieren. Um die verfügbare Rechenleistung für den MC Agenten bestmöglich zu verteilen habe ich mich dafür entschieden, die Anzahl an Iterationen mit der Anzahl an verfügbaren Aktionen zu multiplizieren. Bei einem Spielzustand mit doppelt so vielen Aktionen, ist der zu untersuchende Spielbaum auch meistens doppelt so groß. Eine Verdoppelung der Iterationen führt nun dazu, dass der größere Spielbaum genauso gründlich wie der kleinere untersucht werden kann.

[8] Siehe Abbildung 4, S. 24.

Eine Verknüpfung mit der Anzahl an leeren Feldern wurde ebenfalls in Erwägung gezogen, jedoch nicht implementiert. Spielzustände mit vielen freien Feldern sind in der Regel nicht kritisch, da sie noch weit vom Spielende entfernt sind und sollten deshalb nicht mehr Rechenleistung als Spielzustände, die kurz vor dem Spielende auftreten, erhalten. [Kutsch 2017, S. 19-20]



Für den MC Agenten mit einer Rolloutdepth von 20, gemessen über 50 Spiele. Die Fehlerbalken stellen eine Standartabweichung dar.

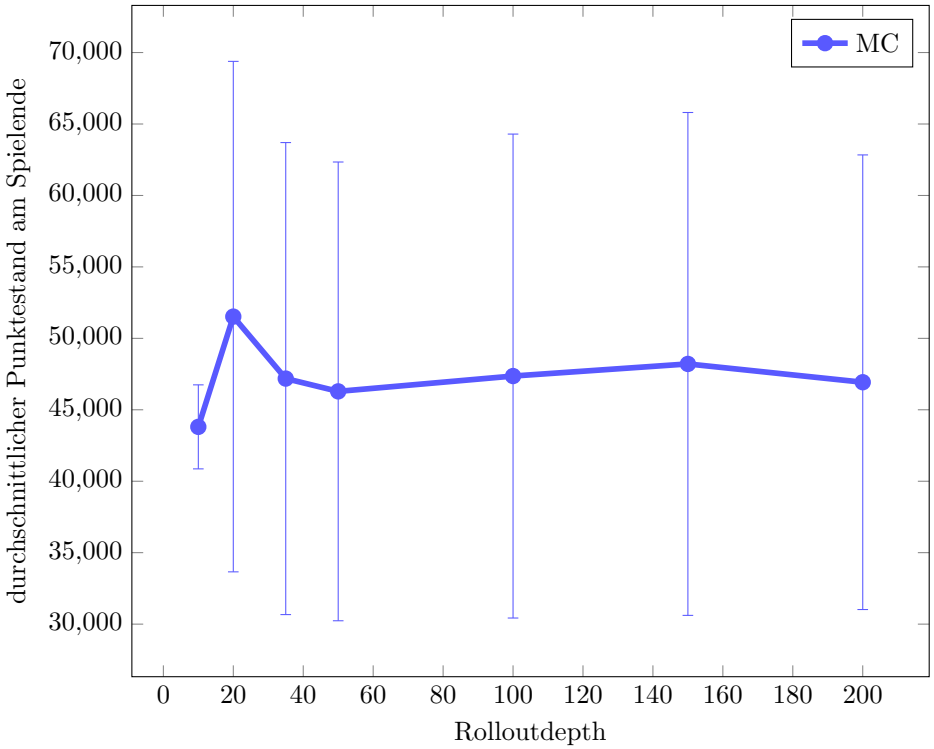
Abbildung 4: Durchschnittlichen Punktzahl des MC Agenten am Spielende für verschieden Iterationen.

Rolloutdepth

Wie in Abbildung 5 ersichtlich, befindet sich die optimale Roloutdepth des MC Agenten bei ungefähr 20. Die Ursache hierfür liegt vermutlich in dem eingangs angesprochenen breiten Spielbaum des Spieles 2048. Ohne eine Limitierung der Rolloutdepth ist dieser einfach so groß, dass er durch die Simulationen nicht dicht genug abgesucht werden kann. Die niedrigere Rolloutdepth sorgt dafür, dass sich die Simulationen nur auf die nächsten 20 Spielzustände konzentrieren

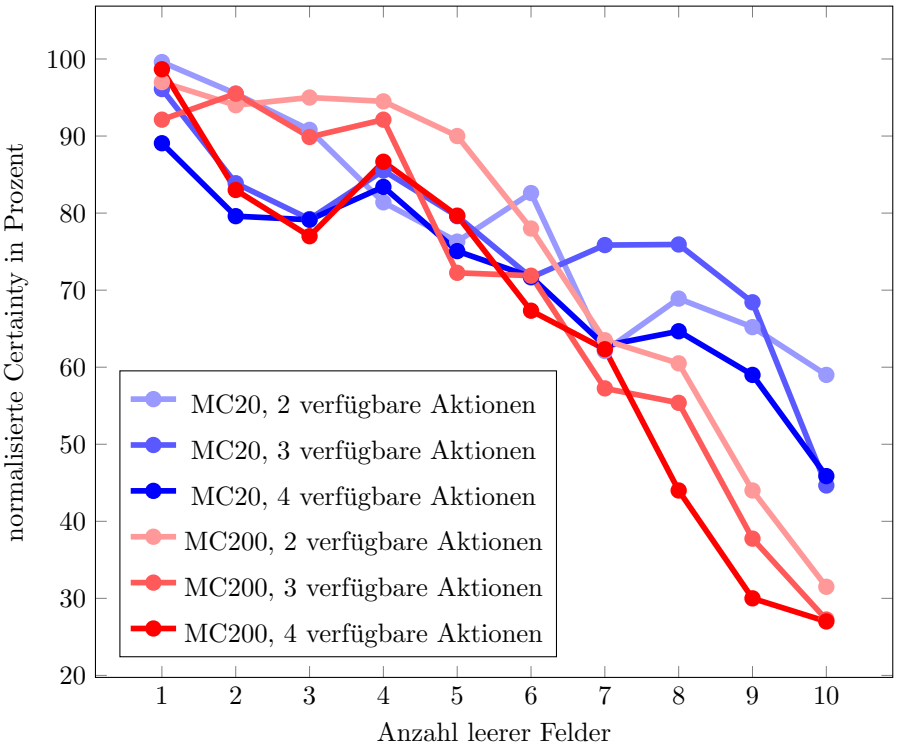
und der Agent die Aktion auswählt, welche nach 20 Aktionen das beste Ergebnis liefert.

Dieses Ergebnis wird durch die, in Abbildung 6 dargestellte, Certainty des MC Agenten mit einer Rolloutdepth von 20 und 200 bestätigt. Bei einer Rolloutdepth von 20 erreicht der MC Agent eine durchschnittliche Certainty von `75,09%`, bei einer Rolloutdepth von 200 beträgt die durchnittliche Certainty `69,83%`. Wie zu erwarten fällt die Certainty des Agentens mit einer höheren Anzahl von leeren Feldern ab, allerdings unterscheidet sich die Certainty bei der selben Anzahl an leeren Feldern und einer unterschiedlichen Anzahl von verfügbaren Aktionen nicht stark. Dies liegt vermutlich an der variablen Anzahl an Iterationen, welche Spielzuständen mit mehr verfügbaren Aktionen eine größere Anzahl an Iterationen zuweist.[Kutsch 2017, S. 29-30]



Für den MC Agenten mit 1000 Iterationen, gemessen über 50 Spiele. Die Fehlerbalken stellen eine Standartabweichung dar.

Abbildung 5: Durchschnittliche Punktzahl des MC Agenten am Spielende für verschiedene Rolloutdepths.



Für den MC Agenten mit 1000 Iterationen pro verfügbare Aktion, einer Rolloutdepth von 20 oder 200 und einer Treedepth von 10, gemessen über 20 Spielzustände je Knotenpunkt.

Abbildung 6: Normalisierte Certainty für verschiedene Spielzustände beim MC Agenten mit einer Rolloutdepth von 20 oder 200.

4 Monte Carlo Tree Search Agent

Im folgenden Kapitel wird die Funktionsweise des Monte Carlo Tree Search (MCTS) Agenten erläutert und evaluiert mit welchen Einstellungen der Agent das beste Ergebnis für das Spiel 2048 erreichen kann. Des Weiteren wird das Problem des MCTS Agenten mit nichtdeterministischen Spielen beschrieben, welches vermutlich der Grund für die schlechte Performance des MCTS Agenten ist. Am Ende des Kapitels folgt ein kurzes Fazit in dem der MCTS Agent mit dem MC Agent verglichen wird und mögliche Lösungen für das Problem des MCTS Agenten mit nichtdeterministischen Spielen erläutert werden.

4.1 Funktionsweise des Monte Carlo Tree Search Agenten

Algorithm 2 MCTS

```
1: function MCTS(s_0)
2:     create root node v_0 with state s_0
3:     while (within computational budget) do                ▷ Main Loop
4:         V_leaf ← TREEPOLICY(v_0)                          ▷ Selektion und Expansion
5:         Δ ← ROLLOUT(s(Vleaf))                             ▷ Simulation
6:         BACKUP(Vleaf, Δ)                                  ▷ Backpropagation
7:     return BESTACTION(v_0)                                ▷ Auswahl der nächsten Aktion
```

Der Monte Carlo Tree Search (MCTS) **Agent!** ist ähnlich wie ein Spielbaum aufgebaut. Er besteht aus mehreren Knotenpunkten `k`, welche jeweils Informationen über den Wert v_k und die Anzahl der Besuche des Knotenpunktes n_k enthalten. Während der Evaluation eines Spielzustandes wird dieser Baum bis zu einer vorher festgelegten Baumtiefe (Treedepth) aufgebaut. Ähnlich wie der MC **Agent!** durchläuft der MCTS **Agent!** bei der Untersuchung eines Spielzustandes mehrere Phasen, die Selektion, die Expansion, die Simulation und die Backpropagation (Abbildung 7). Diese werden so lange durchlaufen bis eine gewünschte Anzahl von Iterationen erreicht wurde oder eine festgelegte Zeitspanne vergangen ist. [Chaslot 2010, S. 18][Kutsch 2017, S. 13-14]

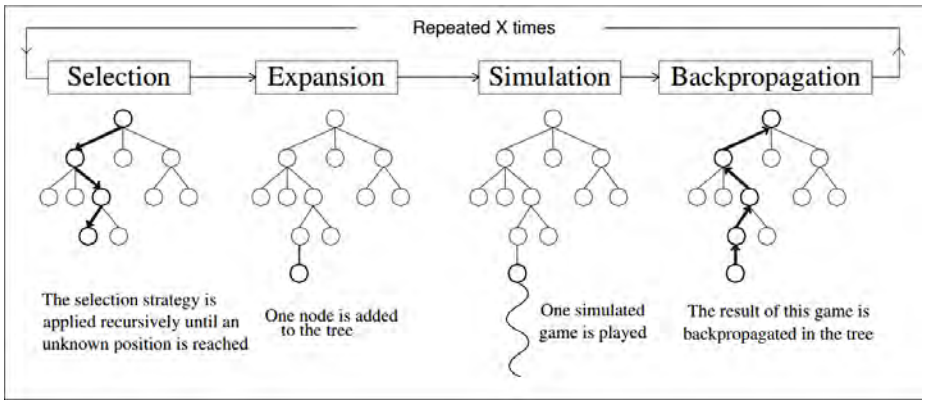


Abbildung 7: Funktionsweise des Monte Carlo Tree Search Agenten. [Chaslot 2010, S. 18]

Phase 1: Selektion

Algorithm 3 MCTS Selektion

```
1: function TreePolicy(v)
2:     while NOT(v terminal OR maxMoves) do
3:         if v NOT fully expanded then
4:             return Expand(v)
5:         else
6:             v ← BestChild(v, K)
7:     return v
8:
9: function BestChild(v)                                   ▷ UCT
10:    return argmax_{v'∈children(v)} Q(v')/N(v') + K sqrt(ln N(v)/N(v'))
```

Während der Selektion ist es das Ziel des Monte Carlo Tree Search (MCTS) **Agent!**s zum Ende des Baumes zu gelangen. Dazu wählt er eine Selektionsstrategie und bewegt sich anhand dieser Strategie rekursiv durch den Baum. Die Selektionsstrategie kontrolliert dabei die Balance zwischen der Ausbeutung (Exploitation) und der Erkundung (Exploration). Auf der einen Seite besteht die Aufgabe darin, den Weg zu finden, welcher aktuell den größten Wert v aufweist (Exploitation), allerdings soll der Agent auch neue Wege ausprobieren

(Exploration), da aufgrund der Ungenauigkeit der Evaluation nicht sichergestellt werden kann, dass sich hinter den neuen Wegen kein besseres Ergebnis verbirgt.
Der im GBG Framework implementierte Monte Carlo Tree Search (MCTS) **Agent!** benutzt als Selektionsstrategie die sogenannte Upper Confidence bounds applied to Trees (UCT). Wenn `I` die Menge an Knoten ist, die vom aktuellem Knoten `p` durch eine Aktion erreicht werden können, wählt UCT entsprechend der Formel 1 einen Knoten `k` aus der Menge `I` aus. [Chaslot 2010, S. 21][Kutsch 2017, S. 15]

$$k = \underset{i \in I}{\operatorname{argmax}} \frac{v_i}{n_i} + K \times \sqrt{\frac{\ln n_p}{n_i}} \tag{1}$$

Wenn v_i der Wert des Knotens `i`, n_i die Anzahl an Aufrufen von `i` und n_p die Anzahl an Aufrufen von `p` ist. `K` ist ein Koeffizient der durch Ausprobieren manuell eingestellt werden muss.

Phase 2: Expansion

Algorithm 4 MCTS Expansion

```
1: function EXPAND(v)
2:     choose a ∈ untried actions from A(s(v))
3:     add new child v_new to v
           with s(v_new) = ADVANCE(s(v); a)
           and a(v_new) = a
4:     return v_new
```

Nachdem der Agent zu einem Ende des aktuellen MCTS Baumes navigiert ist, folgt nun die Expansion. Während dieser Phase wird ein neuer Knoten zu dem Baum hinzugefügt. Aus Performancegründen ist es meistens nicht möglich den gesamten Spielbaum im MCTS **Agent!**s abzubilden, sodass eine bestimmte Strategie verfolgt werden muss, welche entscheidet wann der Baum durch neue Knoten erweitert wird.
Die bekannteste Strategie erstellt für jedes simulierte Spiel einen einzigen neuen Knoten. Dieser Knoten wird an der ersten neuen Position erstellt, welche

während der Traversierung des MCTS Baumes erreicht wird. Eine weitere Strategie ist es, denn Baum vor der Suche bis zu einer bestimmten Treedepth aufzubauen.
Der im GBG Framework integrierte MCTS Agent benutzt eine Strategie, welche Merkmale beider Strategien aufweist. Während jeder Iteration wird versucht, den MCTS Baum um einen Knoten zu erweitern, allerdings nur bis zu einer vorher festgelegten Treedepth. Sollte diese Treedepth erreicht sein, wird kein neuer Knoten erstellt und der aktuelle Knoten wird ausgewählt. Wenn ein neuer Knoten erstellt wurde, wird der neue Knoten ausgewählt. Anschließend wird mit dem ausgewähltem Knoten die Simulation durchgeführt. [Kutsch 2017, S. 14-15]

Phase 3: Simulation

Algorithm 5 MCTS Simulation

1: **function** ROLLOUT(s)
2: $s_{ref} = s$
3: **while** NOT(s terminal OR maxMoves) **do**
4: choose $a \in$A(s) uniformly at random
5: $s \leftarrow$ ADVANCE($s; a$)
6: **return** reward for state s relative to s_{ref}

Während der Simulation wird der, dem ausgewähltem Knoten zugeordnete, Spielzustand durch das Ausführen von Aktionen, welche durch den Agenten ausgewählt werden, bis zum Spielende oder dem Erreichen einer bestimmten Rolloutdepth gespielt. Die Auswahl der Aktionen geschieht in den meisten Fällen zufällig, allerdings gibt es auch Ansätze, bei denen die Aktionen durch Heuristiken ausgewählt werden.
Bei der Wahl der richtigen Simulationsstrategie ist es wichtig, ein gutes Gleichgewicht zwischen der Exploration und der Exploitation zu finden (siehe Selektion). Ist die Auswahl der nächsten Aktion zu zufällig, findet zu viel Exploration statt, wird sie allerdings zu stark durch Heuristiken geleitet, wird der durchsuchte Bereich stark eingeschränkt und es findet zu viel Exploitation statt. [Chaslot 2010, S. 22-23][Kutsch 2017, S. 15]

Phase 4: Backpropagation

Algorithm 6 MCTS Backpropagation

```
1: function BACKUP(v; Δ)
2:     while v is not null do
3:         N(V) ←N(v) + 1
4:         Q(V) ←Q(v) + Δ
5:         v ←parent of v
```

Nachdem ein Spiel simuliert wurde, wird das Ergebnis der Simulation des Endknotens an alle Knoten, die zum Erstellen dieser Simulation durchlaufen wurden, weitergegeben. Dazu wird das Ergebnis der Simulation zu dem bereits im Knoten gespeicherten Wert `v` hinzugefügt. Zusätzlich wird der Zähler `n` für die Anzahl an Aufrufen in jedem durchlaufenen Knoten um 1 erhöht. Die Wert eines Knotens `k` kann außerdem jederzeit durch Formel 2 berechnet werden. [Chaslot 2010, S. 23][Kutsch 2017, S. 15]

$$v_k = \sum_{i \in children(k)} v_i \tag{2}$$

Wenn v_k der Wert des aktuelle Knotens `k` und v_i der Wert des Knotens `i` ist.

Auswahl der nächsten Aktion

Nachdem der MCTS Baum aufgebaut und die Simulationen durchgeführt wurden, muss die nächste Aktion des **Agent!**s ausgewählt werden. Dazu werden die Knoten der ersten Generation, also die vom Wurzelknoten ausgehenden Kinderknoten, untersucht und es wird der beste dieser Knoten ausgewählt. Für die Auswahl des besten Knotens gibt es mehrere Strategien:

Max Child: Das Max Child ist der Kinderknoten `K` mit dem höchsten durchschnittlichen Wert $\frac{v_K}{n_k}$. [Chaslot 2010, S. 25]

Robust Child: Das Robust Child ist der Kinderknoten `K` mit der höchsten Anzahl an Aufrufen n_K. [Chaslot 2010, S. 25]

Robust-Max Child: Das Robust-Max Child ist eine Kombination der Strategien Max Child und Robust Child. Es wird der Kinderknoten `K`, welcher sowohl

die höchsten Anzahl an Aufrufen n_K als auch den höchsten durchschnittlich Wert $\frac{v_K}{n_k}$ besitzt, ausgewählt. Wenn kein Robust-Max Child vorhanden ist, werden weitere Simulationen durchgeführt bis ein Robust-Max Child vorhanden ist. [Chaslot 2010, S. 25]

Secure Child: Das Secure Child maximiert eine untere Konfidenzgrenze. Dazu wird für jeden Kinderknoten `k` ein Wert `w` berechnet, welcher eine Kombination aus der durchschnittlichen Wert $\frac{v_K}{n_k}$ und der Wurzel aus der Anzahl an Aufrufen n_K des Kinderknotens ist. Die Formel zur Berechnung dieses Wertes lautet `w =` $\frac{v_k}{n_k} + \frac{L}{\sqrt{n_k}}$, wobei L eine Konstante ist, welche beeinflusst, wie stark die Anzahl an Aufrufen gewichtet wird. [Chaslot 2010, S. 25]

Der im GBG Framework integrierte MCTS Agent verwendet die erste Strategie, also das Max Child. [Kutsch 2017, S. 16]

4.2 Evaluation des Monte Carlo Tree Search Agenten und Analyse der Parameter

Der MCTS Agent erreicht mit optimalen Einstellungen durchschnittlich ungefähr `34713` Punkte am Spielende. Wie bereits beim MC Agenten kann die Tiefe der Rolloutdepth sowie die Anzahl an Iterationen verändert werden. Zusätzlich kann noch eine Treedepth und der Koeffizient `K` der UCT Formel eingestellt werden.

Iterationen: Durch die Iterationen wird angegeben, wie oft der MCTS Agent die in Abbildung 7 dargestellten Phasen durchläuft. Um die genutzte Rechenleistung vergleichbar zum MC Agenten zu halten, wurde sich dafür entschieden `3500` Iterationen zu nutzen.

Rolloutdepth: Die Rolloutdepth gibt an, wie viele Zufallsaktionen während der Simulation maximal ausgeführt werden sollen. Der MCTS Agent erreicht optimale Ergebnisse bei einer Rolloutdepth von etwa `150`.[9]

[9] Siehe Abbildung 15, S. 53.

Treedepth: Durch die Treedepth kann die maximale Anzahl an Ebenen des MCTS Agenten begrenzt werden. Aufgrund des Problems des MCTS Agenten mit nichtdeterministischen Spielen werden die besten Ergebnisse bei einer Treedepth von 1 erreicht.

Koeffizient von UCT: Dieser Koeffizient (`K` in der Formel 1) gibt an, wie stark die Aufrufe eines Knotens während der Selektion gewichtet werden. Da dieser Koeffizient die Performance des MCTS Agenten nur sehr gering beeinflusst habe ich mich dafür entschieden ihn auf dem Standartwert von `1.4142135623730951` zu lassen. Später stellte sich heraus, dass die UCT Formel für 2048 immer den am wenigsten besuchten Knoten auswählt, weshalb der Koeffizient irrelevant ist.[10]

Iterationen

Anders als beim MC Agenten ist eine Skalierung der Iterationen mit der Anzahl an verfügbaren Aktionen nicht für den MCTS Agenten implementiert. Daraus ergaben sich einige Schwierigkeiten bei der Evaluation gesamter Spiele. Wenn man beiden Agenten die selbe Anzahl an Iterationen zuweisen würde, würde der MC Agent deutlich mehr Rechenleistung benötigen. Um die verbrauchte Rechenleistung beider Agenten vergleichbar zu halten habe ich mich dafür entschieden, die Anzahl an Iterationen für den MCTS Agenten um den Faktor `3,5` auf `3500` zu erhöhen Dies entspricht etwa der durchschnittlichen Anzahl an verfügbaren Aktionen. Somit ist die genutzte Rechenleistung beider Agenten vergleichbar. Bei der Evaluation einzelner Spielzustände, also dem Errechnen der Certainty, wurden die MCTS Agenten einzeln für jeden Spielzustand erstellt, sodass die Anzahl an Iterationen an den Spielzustand angepasst werden konnte und mit der Anzahl an verfügbaren Aktionen skaliert. [Kutsch 2017, S. 20-21]

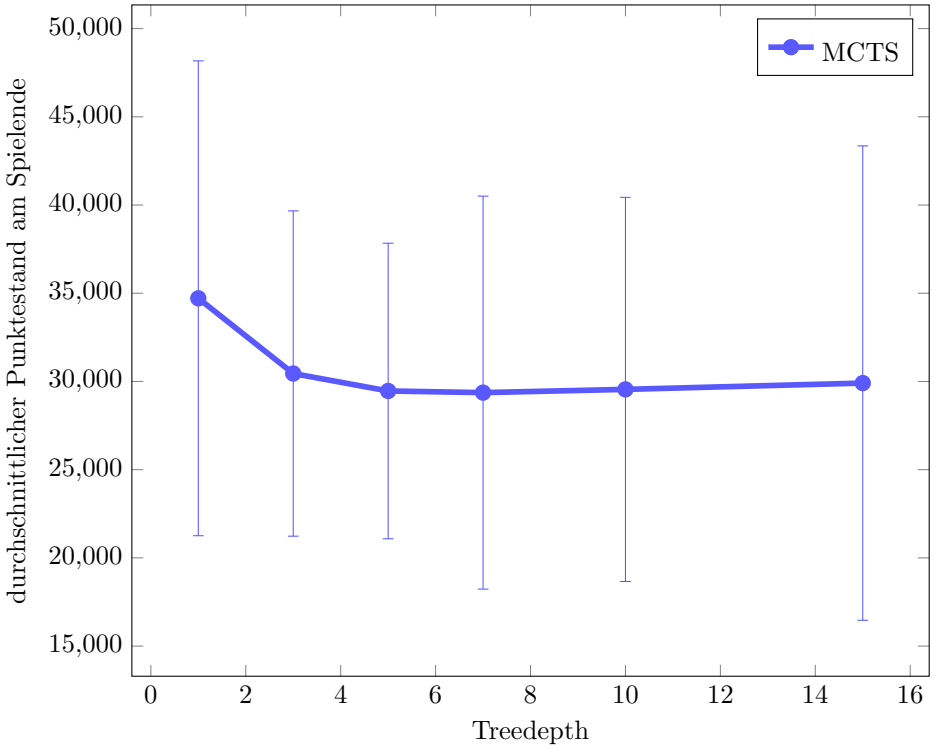
Treedepth

Die Limitierung der Treedepth führt dazu, dass nach Erreichen der eingestellten Treedepth keine weiteren Knoten erstellt werden und die Simulation in einem Knoten wiederholt wird. Dadurch wird das Ergebnis dieser Simulation genauer. Für Spiele mit einem breitem Spielbaum ist dies oft besser, da die Ergebnisse

[10] Siehe Kapitel 5.1, S. 40.

der Rollouts bei ihnen oft sehr ungenau sind und durch einen weiteren Rollout genauer werden.

In Abbildung 8 ist die Performance des MCTS Agenten für verschiedene Treedepths dargestellt. Der durchschnittliche Punktestand am Spielende ist für den MCTS Agenten bei einer Treedepth von `1` am besten. Dies liegt vermutlich an dem Problem des MCTS Agenten mit nichtdeterministischen Spielen. Die in Abbildung 23 auf Seite 88 dargestellte Certainty bestätigt dieses Ergebniss. Sie beträgt bei einer Treedepth von 1 im Durchschnitt `52,2%`, bei einer Treedepth von 10 sind es im Durchscnitt `48,08%`. Da Aufgrund des breiten Spielbaumes eine Treedepth von mehr als 5 selten erreicht wird, wird die tatsächlich genutzte Treedepth bei einem Maximum von 10 nicht limitiert.



Für den MCTS Agenten mit 3500 Iterationen und einer Rolloutdepth von 150, gemessen über 50 Spiele. Die Fehlerbalken stellen eine Standartabweichung dar.

Abbildung 8: Durchschnittliche Punktzahl des MCTS Agenten am Spielende für verschiedene Treedepths.

4.3 Zwischenfazit

Bisher wurden zwei Agenten für das Spiel 2048 untersucht, der MC Agent und der MCTS Agent. Beide Agenten schaffen es regelmäßig das Spiel 2048 zu gewinnen, also die Kachel mit dem Wert 2048 zu erreichen, der MC Agent gewinnt das Spiel zu $\approx$ `98%`, der MCTS Agent zu $\approx$ `84%`. Der MC Agent bildet des weiteren in $\approx$ `55%` aller Spiele eine Kachel mit dem Wert 4096 und sehr selten — deutlich unter `1%` — eine Kachel mit dem Wert 8192. Der MCTS Agent bildet die Kachel mit dem Wert 4096 in ungefähr jedem zehnten Spiel.
Diese Ergebnisse sind überraschend, da eigentlich zu erwarten war, dass der deutlich komplexere MCTS Agent bessere Ergebnisse als der relativ simple MC Agent erreicht. Der Grund für diese Diskrepanz wurde nach langem Suchen in einem Problem des MCTS Agenten mit nichtdeterministischen Spielen gefunden.

4.4 Problem des Monte Carlo Tree Search Agenten mit nichtdeterministischen Spielen

Der im GBG Framework implementierte MCTS Agent ist nicht für die Untersuchung von nichtdeterministischen Spielen, wie z.B. 2048, geeignet. Diese Problematik resultiert aus dem Zufallsfaktor beim Erstellen einer neuen Kachel. Aktuell führt dies dazu, dass eine Aktion auf einem Spielzustand s_0 zu mehreren neuen Spielzuständen s_1 bis s_x (mit $x = 2a$, wenn a die Anzahl an leeren Feldern ist) führt. Da der MCTS Agent beim Erstellen des Baumes die relevanten Spielzustände jeweils in den Knoten speichert, wird jedoch pro Knoten, also pro Aktion, jeweils nur einer dieser Spielzustände gespeichert und für die weitere Evaluation des ursprünglichen Spielzustandes berücksichtigt. Auch für die Simulation wird der gespeicherte Spielzustand genutzt. Dies führt dazu, dass der Agent einen sehr stark eingeschränkten Spielbaum evaluiert.
Das Problem wurde in Abbildung 9 schematisch dargestellt. Wenn der Agent die Aktion, die zum Spielzustand s_0 führt, auswählt, speichert er in dem zugehörigen Knoten nicht den Spielzustand s_0, sondern einen Spielzustand, welcher von der Spielumgebung gewählt wurde und auf dem bereits die neue Zufallskachel erzeugt wurde. In diesem Beispiel ist das der Spielzustand s_2. Anschließend führt

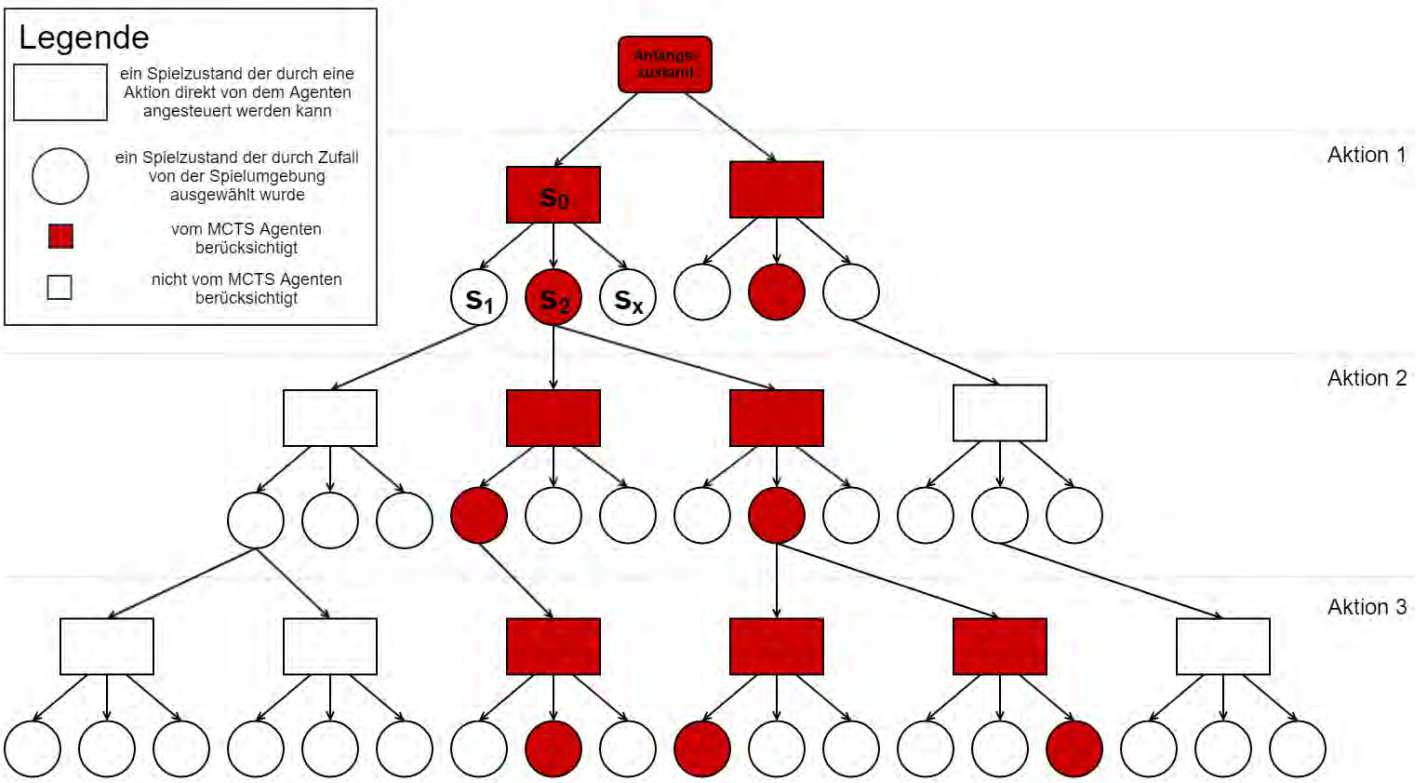


Abbildung 9: Der für den MCTS Agenten eingeschränkte Spielbaum.

der Agent seinen nächsten Schritt vom Spielzustand s_2 ausgehend aus. Entscheidet sich der Agent nun in einem anderem Durchlauf wieder die Aktion auszuführen, welche zu dem Spielzustand s_0 führt, wird der nächste Spielzustand nicht erneut von der Spielumgebung gewählt, sondern der Agent greift direkt auf den gespeicherten Spielzustand s_2 zu und führt seinen nächsten Schritt von diesem Spielzustand ausgehend aus. Somit werden Spielzustände, welche von den Spielzuständen s_1 oder s_x ausgehen, nie erreicht. [Kutsch 2017, S. 16-18]

4.5 Lösungen für das Problem des MCTS Agenten mit nichtdeterministischen Spielen

Browne et al. [2012, S. 12-14] stellen einige Möglichkeiten für einen nichtdeterministischen MCTS Agenten vor, allerdings scheinen diese Möglichkeiten nicht wirklich für das Spiel 2048 geeignet zu sein. Zur Lösung dieses Problems wurden deshalb 2 weitere Ansätze erarbeitet:

2048 als Zweipersonenspiel betrachten: Die Betrachtung des Spieles 2048 als Zweipersonenspiel war mein erster Ansatz zum Lösen des Problemes. Dabei führt der MCTS Agent wie gewohnt seine Aktion aus, allerdings wird das Erstellen einer zufälligen Kachel durch die Umgebung als Aktion eines anderen Spielers simuliert. Dieser wird durch einen Random Agenten dargestellt, welcher zufällig eine Aktion auswählt, also eine Kachel mit dem Wert zwei oder vier, entsprechend der Regeln, zufällig auf dem Spielfeld platziert.

Eine ähnliche Vorgehensweise wird auch von Vryniotis [2014] beschrieben, allerdings nutzt er anstelle des Zufallsagenten einen minimierenden Agenten, der die Kachel immer auf der für den Agenten schlechtesten Position platziert. So wird sichergestellt, dass der Agent, der 2048 spielt, immer eine gute Aktion auswählt, unabhängig davon an welcher Position die neue Kachel im echtem Spiel entsteht.

MCTSExpecitmax: Bei dem MCTSE Agenten wird der MCTS Baum durch einen MCTSExpecitmax Baum ersetzt. Dieser besteht aus alternierenden Ebenen von Expectimax und Chanceknoten. Ein Expectimax Knoten stellt dabei eine vom Agetenen ausgewählte Aktion dar. Ein Chanceknoten stellt ein daraus resultierendes zufälliges Ereignis dar, wie etwa das erstellen einer Zufallskachel in 2048. Dieses Prinzip wurde bereits erfolgreich von Szita et al. [2009] für das Spiel Siedler von Catan und von Van den Broeck et al. [2009] für Poker angewandt.

Ich habe mich dafür entschieden, den MCTSE Agenten zu testen, da dieser bereits für andere nichtdeterministische Spiele erfolgreich eingesetzt wurde. Der erste Ansatz würde vermutlich auch funktionieren und ein ähnliches Ergebnis hervorbringen, allerdings wäre es deutlich komplizierter ihn zu implementieren.

5 Monte Carlo Tree Search Expectimax Agent

Im folgendem Kapitel wird die Funktionsweise des Monte Carlo Tree Search Expectimax (MCTSE) Agenten erläutert und evaluiert mit welchen Einstellungen der Agent das beste Ergebnis für das Spiel 2048 erreichen kann. Dieser Agent wurde bereits erfolgreich von Szita et al. [2009] zum Lösen des Spieles Siedler von Catan und von Van den Broeck et al. [2009] zum Lösen des Spieles Poker eingesetzt. Der MCTSE Agent basiert auf den Prinzipien des MCTS Agenten und behebt das Problem, dass der MCTS Agent beim lösen nichtdeterministischer Spiele aufweist. Im weiteren Verlauf des Kapitels wird beschrieben, mit welchen Mitteln die Geschwindigkeit des MCTSE Agenten verbessert werden konnte. Am Ende des Kapitels folgt ein kurzes Fazit, in welchem der MCTSE Agent mit dem MCTS Agenten und dem MC Agenten verglichen wird.

5.1 Funktsionsweise des Monte Carlo Tree Search Expectimax Agenten

Algorithm 7 MCTSE

```
1: function MctsExpectimax(s_0)
2:     create root node v_0 with state s_0
3:     while (within computational budget) do                ▷ Main Loop
4:         V_leaf ← TreePolicy(v_0)                          ▷ Selektion und Expansion
5:         Δ ← Rollout(s(V_leaf))                            ▷ Simulation
6:         Backup(V_leaf, Δ)                                 ▷ Backpropagation
7:     return BestAction(v_0)                                ▷ Auswahl der nächsten Aktion
```

Da der MCTSE Agent auf dem MCTS Agenten aufbaut, ähnelt er diesem stark in seiner Funktionsweise. Er durchläuft ebenfalls mehrmals die Phasen Selektion, Expansion, Simulation und Backpropagation.
Im Gegensatz zum MCTS Agenten besteht der MCTSE allerdings nicht nur aus einem Knotentyp, sondern aus alternierenden Ebenen von Expectimaxknoten[11] und Chanceknoten. Dabei wird eine vom Agenten ausgewählte Aktion immer durch einen Expectimaxknoten und mehrere Chanceknoten dargestellt.

[11] Bei Mehrpersonenspielen werden die Expectimaxknoten teilweise durch Expectiminknoten ersetzt, abhängig davon welcher Spieler die Aktion auswählt. Es sind somit bei Mehr-

Der Expectimaxknoten repräsentiert die ausgewählte Aktion und hat mehrere Chanceknoten als Kinder, welche jeweils eine der möglichen nichtdeterministischen Spielzustände darstellen, die dass Spielbrett aufgrund der vom Agenten ausgewählten Aktion annehmen kann. Eine schematische Darstellung des MCTSE Spielbaumes wird in Abbildung 10 dargestellt.

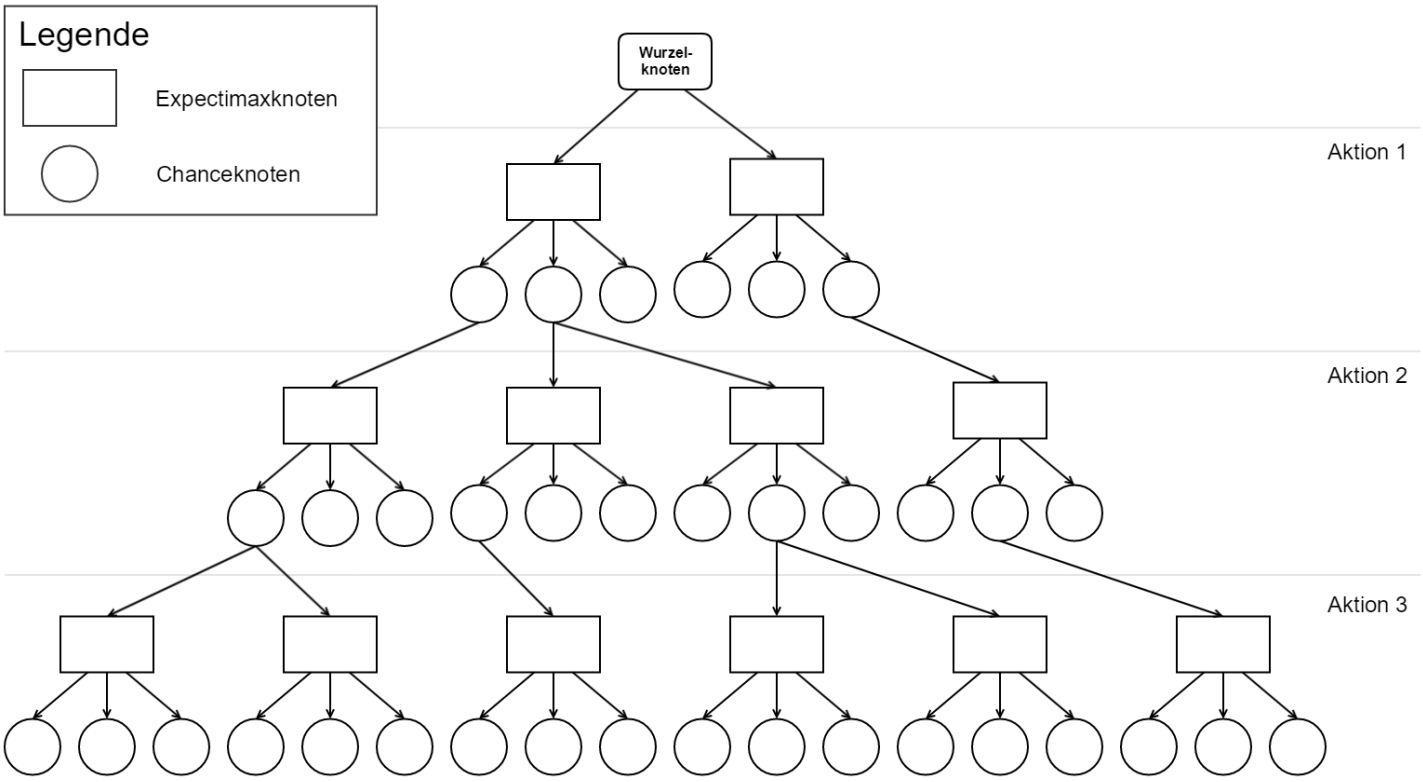


Abbildung 10: Schematische Darstellung des MCTSE Spielbaumes.

personenspielen mit Expectimaxknoten auch immer die Expectiminknoten gemeint, abhängig davon welcher Spieler die Aktion vorgibt.

Phase 1: Selektion

Algorithm 8 MCTSE Selektion

```
1: function TreePolicy(v)
2:     while NOT(v terminal OR maxMoves) do
3:         if v NOT fully expanded then
4:             if NOT maxNumberTreeNodes reached then
5:                 return Expand(v)
6:             else
7:                 return v
8:         else
9:             v_e ← BestChild(v)        ▷ Einen Expektiomaxknoten auswählen
10:            s' ← f(s(v), a(v_e))   ▷ Einen nichtdeterministischen Spielzustand
    mit der Aktion des Expectimaxknotens erzeugen
11:            if ∃ Child(v_e) with s' then
12:                v ←  Child(v_e)
13:            else
14:                add new child v_new(v_e, S')        ▷ Einen neuen Chanceknoten
    erstellen
15:                v ← v_new(v_e, S')
16:    return v
17:
18: function BestChild(v)                                          ▷ UCTN
19:    return argmax_{v'∈children(v)} Q(v')/N(v') 1/m + K sqrt(ln N(v)/N(v'))    ▷ Siehe Gleichung 3
```

Während der Selektion ist es das Ziel des Agenten einen Chanceknoten für die Expansion auszuwählen. Dazu wird im aktuell ausgewählten Chanceknoten überprüft, ob jede aktuell mögliche Aktion bereits durch einen Expectimaxknoten dargestellt wird. Ist dies nicht der Fall, bleibt der aktuelle Knoten ausgewählt und es wird zur Expansion übergegangen. Wenn jedoch jede Aktion durch einen Expectimaxknoten dargestellt wird, wird einer der Expecimaxknoten anhand einer Selektionsstrategie ausgewählt und die zu dem Knoten gehörende Aktion wird auf den aktuellen Spielzustand angewandt. Anschließend navigiert der Agent in den ausgewählten Expectimaxknoten und ordnet dem neuen Spielzustand einen Chanceknoten zu. Falls noch kein Chanceknoten vorhanden ist, der

den Spielzustand repräsentiert, wird ein neuer erstellt. Nun wählt der MCTSE Agent den entsprechenden Chanceknoten aus und beginnt den Prozess von vorne.
Als Selektionsstrategie, wird wie im MCTS Agenten, UCT verwendet. Die UCT Formel wurde jedoch leicht verändert, da sie für Spiele entwickelt wurden, welche eine Game Score von -1, 0 oder 1 zurückgeben, jedoch nicht für das Game Score System von 2048. Bei 2048 entspricht die Game Score dem aktuellen Punktestand geteilt durch den maximal möglichen Punktestand von 3.932.156 Punkten [Asusfood 2015], wozu bei verlorenen Spielen -1 adiert wird. Dies führt dazu, dass die bei einem Rollout erreichte Game Score zum Spielbeginn sehr gering ist und mit Fortlaufen des Spieles immer weiter ansteigt.
Dies ist problematisch, da so innerhalb der UCT Formel zu Spielbeginn, aufgrund der niedrigen Game Score, immer die Anzahl der Aufrufe entscheidend für die Auswahl des nächsten Knotens ist. Ein niedrigerer K Wert könnte dieses Problem kurzfristig lösen, da so die Game Score höher gewichtet wird. Da die Game Score im späteren Spielverlauf allerdings höhere Werte als zum Spielbeginn erreicht, würde das Problem dort umgedreht werden und die Game Score wäre hauptsächlich ausschlaggebend für die Auswahl des nächsten Knotens. Zur Lösung des Problemes wurden zwei Ansätze erarbeitet:

1: Der in der UCT Formel genutzte Wert eines Knotens wird mit dem maximal bei einem Rollout erreichten Wert normalisiert. So ist der in der UCT Formel genutzte Wert während des gesamten Spieles auf einer Skala von -1 bis 1 angelegt.[12] Der für die UCTN Formel genutzte Wert des Knotens ($\frac{v_i}{n_i m}$) ist also davon abhängig, wie nahe der Wert des Knotens am potentiell besten Ergebnis eines Rollouts (m) liegt. Diese Normalisierung des Wertes beginnt nach 100 Iterationen, vorher wird während der Selektion die einfache UCT Formel genutzt, welche meistens den am wenigsten besuchten Knoten auswählt. So wird sichergestellt, dass genügend Rollouts durchgeführt wurden und der zur Normalisierung genutzte Wert möglichst genau ist. Die UCT Normalised (UCTN) Formel ist im Folgenden abgebildet:

[12] Die Game Score kann negativ werden, da sie bei verlorenen Rollouts $-1 + \frac{aktuellerPunktestand}{maximalerPunktestand}$ beträgt. (Siehe 9.2, S. 78)

$$k = \underset{i \in I}{\operatorname{argmax}} \frac{v_i}{n_i m} + K \times \sqrt{\frac{\ln n_p}{n_i}} \tag{3}$$

Wenn v_i der durchschnittliche Wert des Knotens `i`, n_i die Anzahl an Besuchen von `i`, n_p die Anzahl an Besuchen von `p` und `m` der höchste bei einem Rollout in diesem Durchlauf erreichte Wert ist. `K` ist ein Koeffizient der durch Ausprobieren manuell eingestellt werden muss.

2: Alternativ zu der im ersten Ansatz vorgestellten Normalisierung des Wertes, könnte der Agent mit einem höherem `K` Wert initialisiert werden, welcher während des Spielverlaufes langsam verringert wird. Dadurch würde die Gewichtung von der Game Score und der Anzahl an Aufrufen innerhalb der UCT Formel während des Spieles an den steigenden Punktestand angepasst werden.

Da ich den Aufwand geeignete Formel für den zweiten Ansatz zu ermitteln als relativ groß einschätze, habe ich mich dafür entschieden den ersten Ansatz zu implementieren. In Tabelle 1 ist deutlich zu sehen, dass durch die Nutzung der UCTN Formel Knoten mit einem höherem durchschnittlichen Wert bevorzugt aufgerufen werden. Dies ist bei der unveränderten UCT Formel nicht der Fall.

	Ø Wert	UCT	UCTN
Knoten 1	732	1000	566
Knoten 2	862	1000	941
Knoten 3	1024	1000	1763
Knoten 4	823	1000	730

Tabelle 1: Anzahl an Aufrufe der ersten Generation von Expectimaxknoten für UCT und UCTN bei 3500 Iterationen.

Phase 2: Expansion

Algorithm 9 MCTSE Expansion

```
1: function EXPAND(v)
2:     choose a ∈ untried actions from A(s(v))
3:     add new EXPECTIMAX child v_e to v with a(v_e) = a
4:     add new CHANCE child v_new to v_e
           with s(v_new) = ADVANCE(s(v); a)
           and a(v_new) = a
5:     return v_new
```

Nachdem ein Chanceknoten ausgewählt wurde, folgt nun die Expansion. Während der Expansion wird eine mögliche Aktion ausgewählt, die noch nicht durch einen Chanceknoten repräsentiert wird. Anschließend wird ein Expectimaxknoten erstellt, der diese Aktion repräsentiert. Wie in der Selektion wird nun diese Aktion für den aktuellen Spielzustand ausgeführt und der Agent navigiert in den neuen Expectimaxknoten. Dort wird ein neuer Chanceknoten erstellt, der den aktuellen Spielzustand repräsentiert. Ausgehend von diesem Chanceknoten wird nun in Phase 3 die Simulation gestartet.
Um zu entschieden, ob überhaupt eine Expansion stattfinden soll, nutzt der MCTSE Agent eine ähnliche Strategie wie der MCTS Agent. Zusätzlich zu der bereits im MCTS Agenten integrierten Treedepth kann jedoch auch eine maximale Anzahl an Expectimaxknoten (Maxnodes) angegeben werden. Um die Treedepth für den MCTS Agenten und den MCTSE Agenten vergleichbar zu halten, zählt bei dem MCTSE Agenten nur jede zweite Ebene zur Ermittelung der Treedepth, also nur die Ebenen mit Expectimaxknoten. Falls entweder die Treedepth oder die Maxnodes erreicht wurden, wird die Expansion übersprungen und es wird erneut eine Simulation im aktuellen Knoten ausgeführt. Dadurch wird der im Knoten gespeicherte Wert genauer.

Phase 3: Simulation

Die Simulation des MCTSE Agenten verläuft identisch zu der auf Seite 30 beschriebenen Simulation des MCTS Agenten.

Phase 4: Backpropagation

Die Backpropagation des MCTSE Agenten verläuft identisch zu der auf Seite 31 beschriebenen Backpropagation des MCTS Agenten. Wichtig ist dabei, das der Wert `v` und die Anzahl an Aufrufe `n` jeweils in den entsprechenden Expectimax und Chanceknoten aktualisiert werden.
Zusätzlich zu der im MCTS genutzten Backupstrategie wurde eine Strategie getestet, bei welcher jeweils nur der niedrigste mögliche Wert in einem Knoten gespeichert wird. Der Agent sollte sich also für die Aktion entscheiden, welche im schlechtesten Fall das beste Ergebnis hervorbringt. Zur Berechnung des Wertes eines Knotens wurde die Formel 4 genutzt. Es zeigte sich allerdings sehr schnell, dass dieser Ansatz für das Spiel 2048 ungeeignet ist, da der Agent in Testspielen nie mehr als ≈ 5000 Punkte erreichen konnte.

$$v_k = \min_{i \in children(k)} v_i \tag{4}$$

Wenn v_k der Wert des aktuelle Knotens `k` und v_i der Wert des Knotens `i` ist.

Auswahl der nächsten Aktion

Bei der Auswahl der nächsten Aktion wird ebenfalls die im MCTS benutzte Strategie, das Max Child, genutzt. Ich gehe davon aus, dass — ähnlich wie von Chaslot [2010, S. 25] für GO beobachtet — keine signifikanten Unterschiede zwischen den auf Seite 31 beschriebenen Strategien festzustellen sind und habe mich deshalb dagegen entschieden, die anderen Strategien zu implementieren und zu testen.

5.2 Evaluation des Monte Carlo Tree Search Expectimax Agenten und Analyse der Parameter

Der MCTSE Agent erreicht mit optimalen Einstellungen durchschnittlich ungefähr `56989` Punkte am Spielende. Zusätzlich zu den Parametern Rolloutdepth, Treedepth, Anzahl an Iterationen und dem Koeffizienten `K` der UCT Formel, welche auch beim MCTS Agenten justiert werden mussten, kann eine maximale Anzahl an Expectimaxknoten eingestellt werden.

Diese Parameter wurden sowohl mit UCT als auch mit UCTN als Selektionsstrategie untersucht. Interessant ist hierbei, dass UCTN nicht immer besser als UCT ist. Je nachdem wie die Parameter eingestellt sind, werden entweder mit UCTN oder UCT bessere Ergebnisse erzielt. Es ist also manchmal besser, die am wenigsten untersuchte Aktion genauer zu untersuchen und manchmal besser die vielversprechenste Aktion genauer zu untersuchen.

Iterationen: Durch die Iterationen wird angegeben, wie oft der MCTSE Agent die Phasen Selektion bis Backpropagation durchläuft. Um die genutzte Rechenleistung vergleichbar mit der vom MC und MCTS Agenten genutzten Rechenleistung zu halten, wurde entschieden ebenfalls `3500` Iterationen zu nutzen.

Rolloutdepth: Die Rolloutdepth gibt an wie viele Zufallsaktionen während der Simulation maximal ausgeführt werden sollen. Ähnlich wie bereits beim MCTS Agenten werden sowohl für UCT als auch für UCTN die besten Ergebnisse bei einer Rolloutdepth von etwa `150` erreicht.[13]

Treedepth: Durch die Treedepth kann die maximale Anzahl an Ebenen mit Expectimaxknoten für den MCTSE Agenten begrenzt werden. Sowohl für UCT als auch für UCTN werden die besten Ergebnisse ohne eine Begrenzung der Treedepth erreicht.

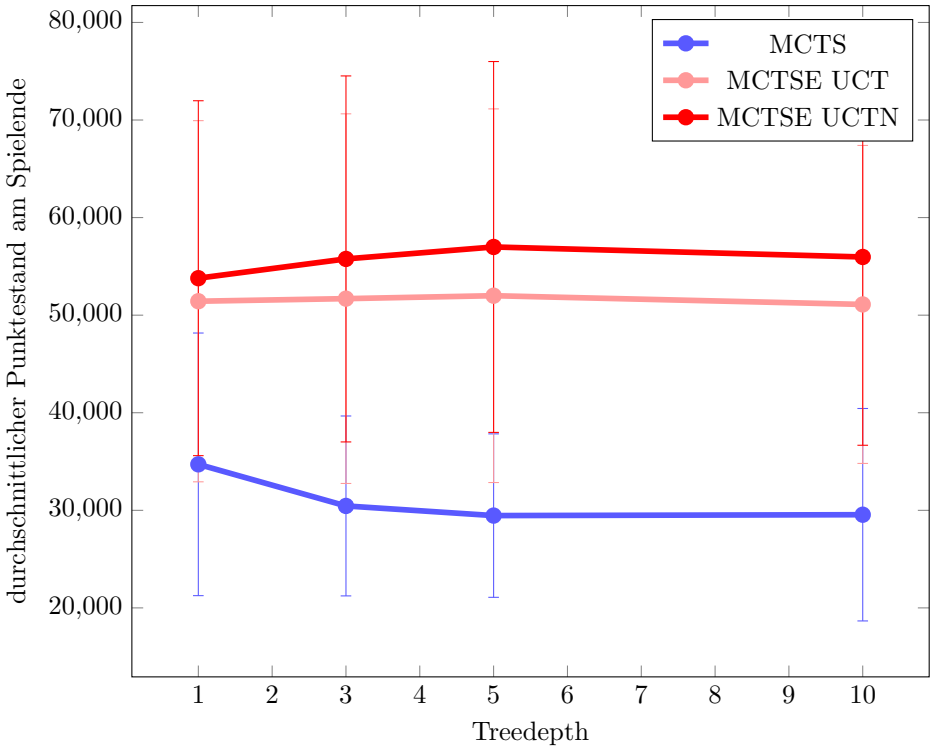
Maxnodes: Die Maxnodes limitieren die Anzahl an Expectimaxknoten, die während der Expansion erstellt werden können. Für UCTN werden die besten Ergebnisse mit etwa `500` Maxnodes erreicht, für UCT werden die besten Ergebnisse ohne eine Limitierung der Maxnodes erreicht.

Koeffizient von UCT(N): Der Koeffizient der UCT(N) Formel gewichtet die Anzahl der Aufrufe des Knotens gegenüber des Wertes eines Knotens. Für UCTN wird mit dem Standardwert von $\sqrt{2}$ das beste Ergebnis erreicht, für UCT wir mit einem Koeffizienten von $\approx 0,0566$ das beste Ergebnis erreicht.

[13] Siehe Abbildung 15, S. 53.

Treedepth

Die Größe der Treedepth hat, wenn man Extremfälle wie eine Treedepth von 1 ignoriert, keinen signifikanten Einfluss auf die Performance des MCTSE Agenten. Dies liegt vermutlich daran, dass die ursprüngliche Funktion der Treedepth, die Größe des Baumes zu limitieren, größtenteils von dem Parameter Maxnodes übernommen wurde. Erwähnenswert ist jedoch, dass im Gegensatz zum MCTS Agenten, eine Treedepth von 1 das Ergebnis des MCTSE Agenten nicht verschlechtert. Dies liegt daran, dass das Problem des MCTS Agenten mit nichtdeterministischen Spielen im MCTSE Agenten behoben wurde und eine höhere Treedepth nun die Certainty erhöht.[14]



Für den MCTSE Agenten mit 3500 Iterationen, einer Rolloutdepth von 150 und 500 Maxnodes, für den MCTS Agenten mit 3500 Iterationen und einer Rolloutdepth von 150, gemessen über 50 Spiele. Die Fehlerbalken stellen eine Standartabweichung dar.

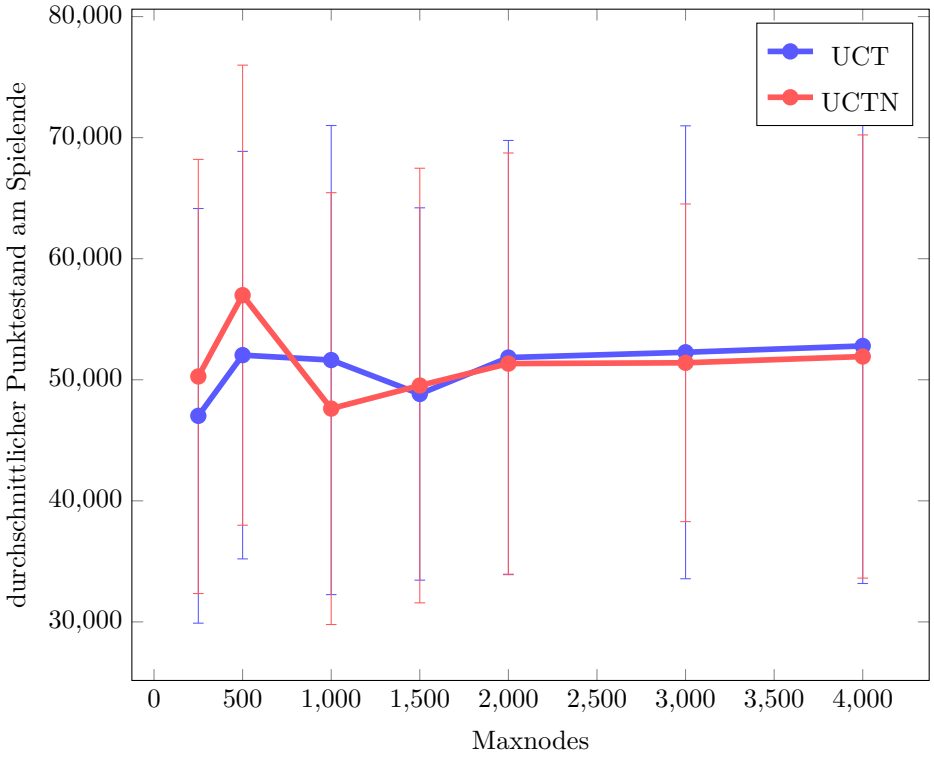
Abbildung 11: Durchschnittliche Punktzahl am Spielende für verschiedene Treedepths.

[14] Vergleiche Abbildung 23, S. 88 mit Abbildung 24, S. 89.

Maxnodes

Der beim MCTS Agenten beobachtete Trend, dass eine limitierte Anzahl an Knoten das Ergebnis verbessert, setzt sich auch beim MCTSE für die Selektionsstrategie UCTN fort. So wird das beste Ergebnis für UCTN bei nur etwa 500 Expectimaxknoten erreicht. Bei dieser Anzahl an Maxnodes werden im Durchschnitt 7 Rollouts für jeden Expectimaxknoten durchgeführt.
Erstaunlicherweise wird das schlechteste Ergebnis ebenfalls für UCTN, bei bereits 1000 Maxnodes, erreicht. Dies ist auch die Anzahl an Maxnodes, bei der UCTN erstmals schlechtere Ergebnisse als UCT erreicht. Ab 1500 Maxnodes liegen die Ergebnisse beider Selektionsstrategien bis zu dem Maximum von 4000 Maxnodes — also keiner Begrenzung der Expectimaxknoten — sehr nahe beieinander. Die UCT Strategie erzielt dabei meistens leicht bessere Ergebnisse.
Für UCT scheint die Anzahl an Maxnodes insgesamt keinen so großen Einfluss auf das Ergebnis wie für UCTN zu haben. Mit dieser Strategie werden bei 500, 1000, 2000, 3000 und 4000 Maxnodes immer relativ genau 52000 Punkte erreicht. Lediglich bei einer sehr geringen Anzahl von 250 Maxnodes und 1500 Maxnodes[15] werden schlechtere Ergebnisse erziehlt.

[15] Hierbei handelt es sich vermutlich um eine Anomalie.



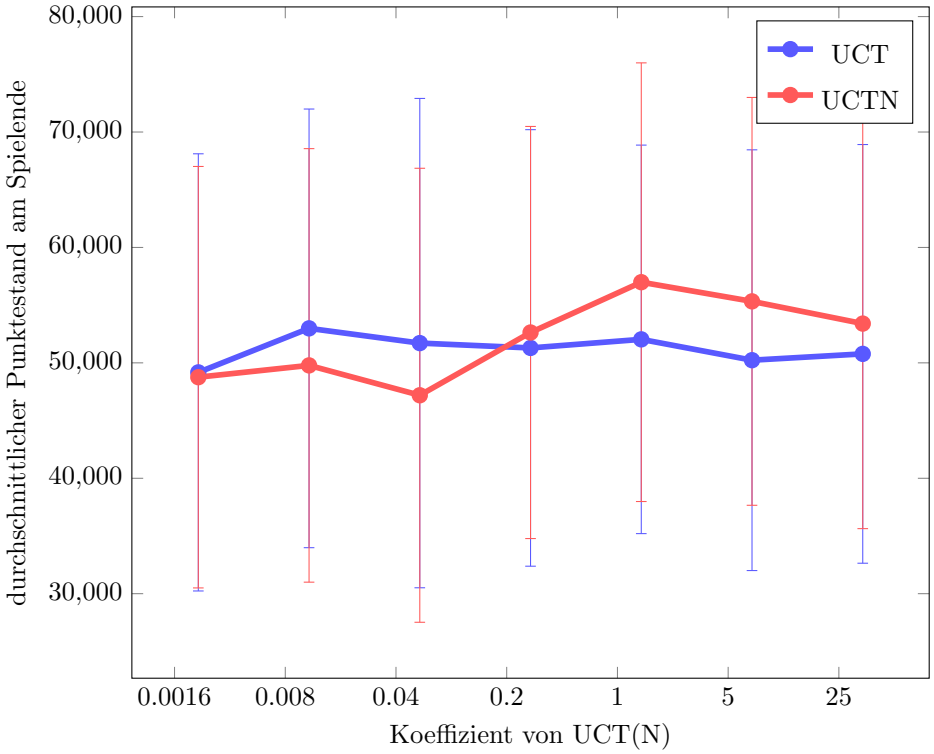
Für den MCTSE Agenten mit 3500 Iterationen, einer Rolloutdepth von 150 und einer Treedepth von 10, gemessen über 50 Spiele. Die Fehlerbalken stellen eine Standartabweichung dar.

Abbildung 12: Durchschnittliche Punktzahl am Spielende für eine unterschiedliche Anzahl von Maxnodes.

Koeffizient von UCT(N)

Der Koeffizient der UCTN Formel gibt an, wie stark die Anzahl der Aufrufe, im Vergleich zu dem Wert eines Knotens, während der Selektion gewichtet werden. Bei einem höherem Koeffizienten ist die Anzahl der Aufrufe eine Knotens wichtiger, bei einem niedrigerem Koeffizienten ist der Wert eines Knotens wichtiger. Wie in Abbildung 13 ersichtlich, werden für UCTN mit der Standarteinstellung von $\sqrt{2}$ bereits optimale Ergebnisse erreicht. Für UCT werden die besten Ergebnisse bei einem deutlich geringerem Wert von $\approx 0,0566$ erreicht. Dieses Ergebnis war zu erwarten, da der durchschnittliche Wert $\frac{v_i}{n_i}$ eines Knotens für UCTN immer im Bereich $[-1..1]$ liegt, wobei 1 für mindestens einen Rollout erreicht wird und meistens viele andere Rollouts in der Nähe liegen. Für UCT

wird das Ergebnis eines Rollouts stattdessen durch den höchsten möglichen Punktestand von `3.932.156` Punkten dividiert. Zum kritischstem Zeitpunkt, welcher bei etwa `50.000` Punkten, dem bilden der Kachel mit dem Wert `4096` auftritt, entspricht der maximale Wert eines Knotens unter UCT $\approx 0,013$. Ein Koeffizient von $\approx 0,0566$ für UCT bewirkt zu diesem Zeitpunkt etwa das selbe Verhalten wie ein Koeffizient von $\sqrt{2}$ bei UCTN.
Beim betrachten dieser Ergebnisse sollte jedoch beachtet werden, dass der ermittelte Koeffizient für UCT nur bei den aktuell vom Agenten erreichten Ergebnissen optimal ist. Sollte der Agent in Zukunft bessere Ergebnisse erreichen und es eine neue kritischste Spielsituation geben, etwa dem erstellen einer Kachel mit dem Wert `8192`, muss der Koeffizient für UCT neu ermittelt werden. Als alternative bietet sich deshalb UCTN an, da hier die Gewichtung während des Spielens an den aktuellen Punktestand angepasst wird. Da mit UCTN ohnehin bessere Ergebnisse als mit UCT erreicht werden, gibt es aktuell keinen Grund UCT zu nutzen.



Für den MCTSE Agenten mit 3500 Iterationen, einer Rolloutdepth von 150, einer Treedepth von 10 und 500 Maxnodes, gemessen über 50 Spiele. Die Fehlerbalken stellen eine Standartabweichung dar.

Abbildung 13: Durchschnittliche Punktzahl am Spielende für verschiedene Werte des Koeffizienten K von UCT und UCTN beim MCTSE Agenten.

5.3 Verbesserung der Geschwindigkeit des MCTSE Agenten

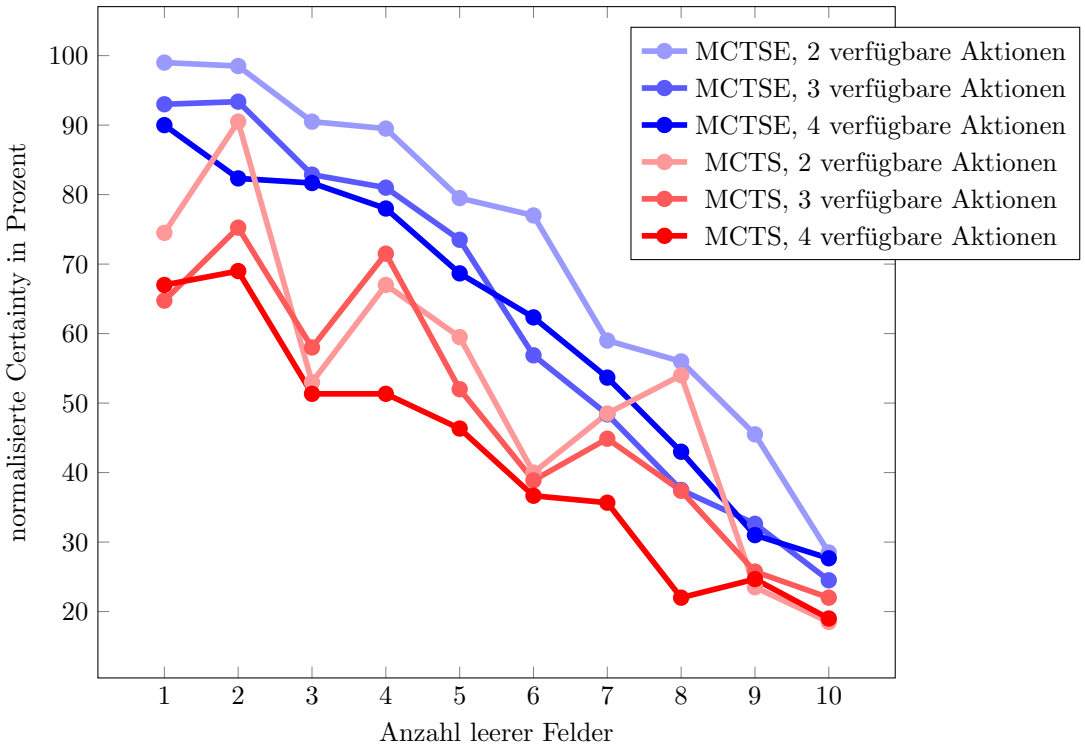
Zusätzlich zu der eingangs beschriebenen Version des MCTSE Agenten, welche beim Durchlaufen des Spielbaumes jede Aktion auf das Spielfeld anwenden muss um einen Spielzustand zu erhalten, der einem Chanceknoten zugeordnet werden kann, wurde eine schnellere Version des MCTSE Agentens entwickelt. Diese nutzt das Interface `StateObservationNondeterministic.java`, welches die Methoden `void advanceDeterministic(Types.Actions action)`, `void advanceNondeterministic()` und `Types.Actions getNextNondeterministicAction()` bereitstellt.[16]. Die hier beschriebene alternative Version reduziert nur die benötigte Rechenleistung des Agenten, beide MCTSE Varianten sind im Bezug auf die erreichte Punktzahl äquivalent.
Die Methode `void advanceDeterministic(Types.Actions action)` wird beim Erstellen eines Expectimaxknotens dazu genutzt, einen Spielzustand zu erzeugen, für den nur die deterministische Aktion ausgeführt wurde. Der Spielzustand repräsentiert somit genau den Zustand des Expectimaxknotens und kann in ihm gespeichert werden.
Nachdem ein neuer Expectimaxknoten erstellt wurde, wird mit der Methode `Types.Actions getNextNondeterministicAction()` zufällig eine der möglichen Zufallsaktionen dieses Expectimaxknotens ausgewählt. Der zu dieser Aktion gehörende Chanceknoten wird erstellt und die Aktion wird mit der Methode `void advanceNondeterministic()` auf den Spielzustand angewandt. Anschließend wird der neue Spielzustand im Chanceknoten gespeichert.
Wird nun in einem neuem Durchlauf eine Aktion ausgewählt, welche zu einem bereits existierendem Expectimaxknoten führt, muss der Spielzustand dieses Knotens nicht erneut hergestellt werden, sondern der im Expectimaxknoten gespeicherte Spielzustand kann einfach kopiert werden. Anschließend wird in dem kopierten Spielzustand mit der Methode `Types.Actions getNextNondeterministicAction()` erneut eine Zufallsaktion ermittelt. Wenn diese Aktion bereits durch einen Chanceknoten dargestellt ist, kann einfach der im Chanceknoten gespeicherte Spielzustand für den weiteren Durchlauf genutzt werden. Falls die Aktion noch nicht durch einen Chanceknoten dargestllt wird, muss ein neuer Chanceknoten erstellt werden, die nichtdeterministische Aktion wird auf den aktuellen Spielzustand angewandt und der neue Spielzustand wird anschließend

[16] Siehe Kapitel 9.6, S. 87.

im Chanceknoten gespeichert.

Dadurch, dass beim Durchlaufen des Spielbaumes nur dann ein neuer Spielzustand erstellt werden muss wenn ein neuer Knoten erstellt wird, konnte die Geschwindigkeit des MCTSE Agenten um etwa 4%[17] erhöht werden.

5.4 Zwischenfazit



Für den MCTS und MCTSE Agenten mit je 1000 Iterationen pro verfügbarer Aktion, einer Rolloutdepth von 150 und einer Treedepth von 10, gemessen über 20 Spielzustände je Knotenpunkt. Der MCTSE Agent nutzt außerdem 500 Maxnodes und die Selektionsstrategie UCTN.

Abbildung 14: Normalisierte Certainty für verschiedene Spielzustände bei MC und MCTSE Agenten mit einer Rolloutdepth von 20.

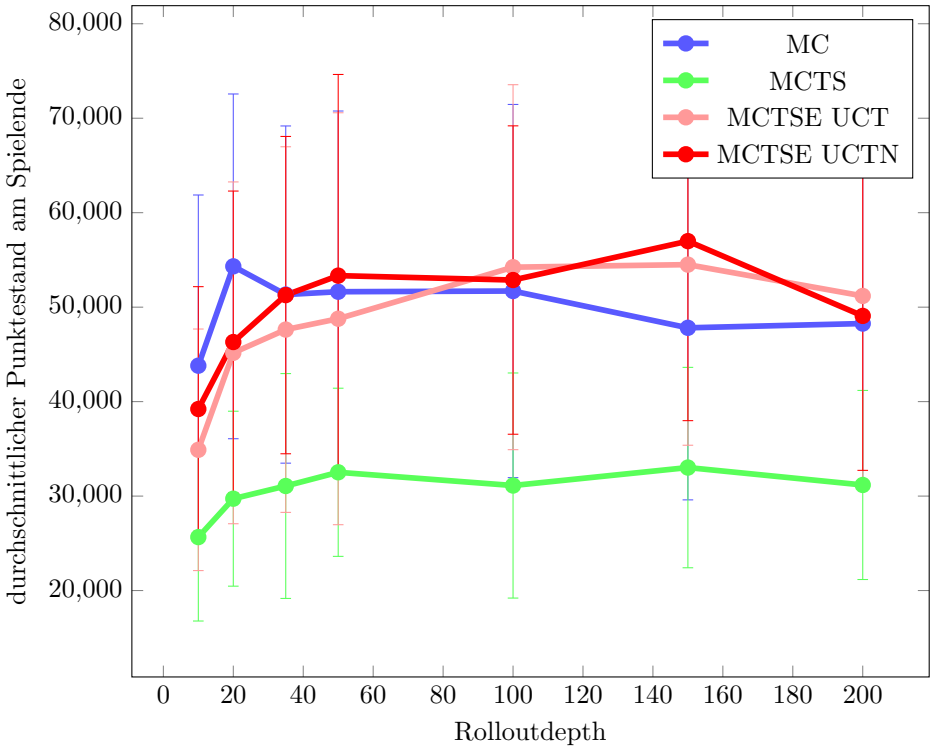
Durch den MCTSE Agenten konnte das Problem des MCTS Agenten erfolgreich behoben werden. Während der MCTS Agent im Schnitt 34731 Punkte erreicht hat, erreicht der MCTSE Agent mit 56989 Punkten ein $\approx 65\%$ höheres Ergebnis. Diese Verbesserung spiegelt sich ebenfalls in der Certainty beider Agenten

[17] Bei 3500 Iterationen, einer Rolloutdepth von 20, einer Treedepth von 10 und 3500 Maxnodes

wieder. In Abbildung 14 ist ersichtlich, das der MCTSE Agent mit einer Certainty von durchschnittlich $\approx 65,5\%$ deutlich über der durchschnittlichen Certainty von $\approx 48,1\%$ des MCTS Agenten liegt.

Im Vergleich zu den `51521` Punkten des MC Agenten erreicht der MCTSE Agent ebenfalls bessere Ergebnisse. Dies liegt vor allem an der verbesserten Selektionsstrategie UCTN, welche im Vergleich zu UCT die Ergebnisse des MCTSE Agenten um $\approx 5\%$ erhöht. Anzumerken ist jedoch, dass der MC Agent bei einer niedrigen Rolloutdepth mit Abstand die besten Ergebnisse erzielt.[18] Da der Rollout einen nicht unerheblichen Anteil des Rechenanteils der Agenten ausmacht, arbeitet der MC Agent deshalb deutlich schneller.

Trotz dieser guten Ergebnisse hat der MCTSE Agent ein relativ großes Pro-



Für den MC Agenten mit 1000 Iterationen, für den MCTS Agenten mit 3500 Iterationen und einer Treedepth von 10, für den MCTSE Agenten mit 3500 Iterationen, 500 Maxnodes und einer Treedepth von 10, gemessen über 50 Spiele. Die Fehlerbalken stellen eine Standartabweichung dar.

Abbildung 15: Durchschnittliche Punktzahl am Spielende für verschiedene Rolloutdepths beim MC, MCTS und MCTSE Agenten.

[18] Siehe Abbildung 15, S. 53.

blem. Durch die Chancenodes wird der Branchingfaktor des Baumes sehr stark erhöht. Bei `3500` Iterationen geht der Baum auch ohne eine Begrenzung der Maxnodes selten über mehr als 4 Ebenen. Im Vergleich zum MCTS Baum, welcher für jede Aktion einen Knoten besitzt, besitzt der MCTSE Baum zwei Ebenen pro Aktion.
Auf der ersten Ebene wird, wie beim MCTS Baum, ein Expectimaxknoten pro Aktion erstellt, auf der zweiten Ebene existieren dann allerdings für jede mögliche Aktion $2 \cdot \mathit{Emptytiles}$ Chanceknoten. Ein Knoten des MCTSE Baumes verzweigt sich also, je nach Spielzustand, in bis zu `112` Chanceknoten. Die Maxnodes begrenzen dieses starke Branching etwas, jedoch wird dadurch zeitgleich die Treedepth des Baumes stark reduziert.

6 TD-NTuple Agent

Im folgenden Kapitel wird die Funktionsweise des Temporal Difference NTuple (TD-NTuple) Agenten zusammengefasst und evaluiert mit welchen Einstellungen der Agent das beste Ergebnis für das Spiel 2048 erreichen kann. Im Gegensatz zu den bisher getesteten Agenten, muss der TD-NTuple Agent trainiert werden. Es soll hauptsächlich untersucht werden, mit wie viel Aufwand ein trainierbarer Agent beim Lösen des Spieles 2048 zu welchem Ergebnis kommt. So können die Ergebnisses der MC Agenten anschließend besser eingeordnet werden. Es wurde sich dafür entschieden einen TD-NTuple Agenten als Vergleichsagenten zu nutzen, da Jaskowski [2016] mit einem ähnlichen Agenten bisher die besten Erfolge beim Lösen von 2048 erzielen konnte. Der Agent war außerdem bereits im GBG Framework implementiert und konnte somit sofort genutzt werden.

6.1 Funktionsweise des Agenten

Anders als die bisher untersuchten Agenten muss der TD-NTuple Agent trainiert werden. Während des Trainings ist es das Ziel des Agenten eine ideale Belohnungsfunktion für jeden Spielzustand zu ermitteln, mit welcher der Agent Spielzuständ bewerten kann. Normalerweise ist diese Belohnungsfunktion vor dem Trainingsbeginn nur für terminale Spielzustände bekannt. In der Regel ist das Ergebnis dieser Funktion dort `+1` wenn der Agent gewonnen hat, `0` bei einem unentschieden und `-1` wenn der Agent verloren hat. Bei 2048 ist das Ergebnis dieser Funktion am Spielende abhängig von der erreichten Punktzahl ($\frac{score}{maxscore}$). Im laufe des Trainings ist es das Ziel, das der Agent diese Belohnungsfunktion für Spielzustände zu erlernt, wo die Funktion vor dem Trainingsbeginn unbekannt oder ungenau ist. Wenn der Agent anschließend ein Spiel spielt, versucht er nun den Wert dieser Funktion zu maximieren.[Thill et al. 2012]

Temporal Difference learning

Temporal Difference learning wird während des Trainings genutzt um die Belohnungsfunktion für nichtterminale Spielzustände zu erlernen. Um diese Funktion zu erlernen beginnt der untrainierte Agent ein Trainingsspiel zu spielen. Dabei wählt er die Aktion aus, die zu dem Spielzustand führt der den Wert der Belohnungsfunktion maximiert. Falls mehrere Spielzustände die selbe höchste Belohnung ausgeben, etwa im ersten Durchlauf, wird einer dieser Aktionen zufällig ausgewählt. Nachdem der Agent eine Aktion ausgewählt und ausgeführt hat, wird die Belohnungsfunktion des vorherigen Spielzustandes aktualisiert. Zum Anpassen dieser Belohnung wurden zwei Methoden getestet:

1: Bei der ersten Methode wird das Ergebnis der Belohnungsfunktion für den vorherigen Spielzustand entsprechend der Formel 5 geändert.

$$v_s = v_s + \alpha(\gamma v_{s'} - v_s) \tag{5}$$

Wenn s der vorherige Spielzustand, s' der nachfolgende Spielzustand, α die Lernrate und γ die Zerfallsrate ist. [Sutton & Barto 1998]

2: Die zweite Methode kann für Spiele genutzt werden, bei denen eine Belohnung bereits ohne Training für jeden Spielzustand ermittelt werden kann. Bei 2048 ist diese Belohnung $\frac{score}{maxscore}$. Diese Methode funktioniert ähnlich wie die erste Methode, mit dem Unterschied, dass diese ermittelte Belohnung entsprechend der Formel 6 beim Aktualisieren der Belohnungsfunktion berücksichtigt wird. Jaskowski [2016] Nutzt ebenfalls diese Methode.

$$z_s = v_s + \alpha(r_{s'} + \gamma v_{s'} - v_s) \tag{6}$$

Wenn s der vorherige Spielzustand, s' der nachfolgende Spielzustand, α die Lernrate, r die beobachtete Belohnung für den neuen Spielzustand und γ die Zerfallsrate ist. [Sutton & Barto 1998]

Anschließend wird nun die Belohnungsfunktion aller vorher besuchten Spielzustände aktualisiert. Da die Belohnungsfunktion dieser Spielzustände nicht so stark verändert werden soll, wie die Belohnungsfunktion des letzten Spielzustandes, werden hierfür Eligibility Traces genutzt.
Jeder Spielzustand hat einen Wert `z`, welcher Auskunft darüber gibt, wann der Spielzustand das letzte Mal aufgerufen wurde. Dieser Wert ist zu Beginn eines

Trainingsspieles 0 und wird mit jedem Besuch des Spielzustandes um 1 erhöht. Zusätzlich wird der Wert jedes mal wenn eine Trainingsaktion ausgeführt wird für alle Spielzustände um den Faktor $\lambda \in [0..1]$ verringert. Daraus ergibt sich folgende Funktion, welche pro Trainingsaktion einmal für jeden alten Spielzustand ausgeführt wird:

$$z_s = \begin{cases} \gamma\lambda z_s + 1 & \text{für den aktuellen Spielzustand} \\ \gamma\lambda z_s & \text{für die alten Spielzustände} \end{cases} \tag{7}$$

Wenn z der Wert der Eligibility Trace, λ der Faktor um den die Eligibilty Trace verändert werden soll und γ die Zerfallsrate ist.[Sutton & Barto 1998, S. 163]

Anschließend kann dann die Belohnungsfunktion für alle alten Spielzustände verändert werden:

$$\delta = \begin{cases} \gamma v_{s'} - v_s & \text{für Methode 1} \\ r_{s'} + \gamma v_{s'} - v_s & \text{für Methode 2} \end{cases} \tag{8}$$

$$\Delta v_s = \alpha\delta z_s \tag{9}$$

Wenn z der Wert der Eligibility Trace, λ der Faktor um den die Eligibilty Trace verändert werden soll, γ die Zerfallsrate und α die Lernrate ist. [Sutton & Barto 1998, S. 164]

Durch die Aktualisierung der vorherigen Spielzustände wird so langsam die Belohnungsfunktion für jeden vom Agenten angesteuerten Spielzustand angepasst. Aufgrund des Zerfalles verändern sich Spielzustände, — besonders wenn die erste Methode genutzt wird — die näher an einem terminalem Zustand sind, schneller ihre Belohnungsfunktion als Spielzustände, die zu Beginn des Spieles auftreten. Da bei der zweiten Methode auch die Belohnungsfunktion von Spielzuständen, die zu Beginn eines Spieles aufgerufen werden, direkt zum Trainingsbeginn stark verändert wird, kann die benötigte Trainingszeit durch diese Methode weiter reduziert werden. Voraussetzung hierfür ist jedoch, dass bereits vor dem Trainingsbeginn eine ausreichend genaue Belohnung für jeden Spielzustand ermittelt werden kann.

NTuple Netzwerke

Problematisch an dem oben beschriebenen TD-learning Agenten ist die Anzahl der Spielzustände, für welche der Agent bei dem Spiel 2048 eine Belohnungsfunktion erlernen muss. Das Spiel 2048 hat ein Spielfeld von 4 mal 4 Kacheln welche im besten Fall je 18 verschiedene Werte annehmen. Somit müsste der Agent theoretisch eine Belohnungsfunktion für $(4 \cdot 4)^{18}$, also $\approx 4,7 \cdot 10^{21}$ Spielzustände ermitteln. Da aktuell maximal die Kachel mit dem Wert 8192 erreicht wird, ist diese Zahl in der Praxis etwas geringer, allerdings immer noch sehr groß. Es liegt für das Spiel 2048 außerdem eine achtfache Symmetrie[19] vor, wodurch die Anzahl an verschiedenen Spielzuständen weiter verringert wird. Trotzdem ist diese Anzahl der Spielzustände viel zu groß um alle Spielzustände in einer angemessenen Zeit evaluieren zu können.
Eine Lösung für dieses Problem ist ein NTuple Netzwerk. Durch dieses Netzwerk wird das Spielfeld in einzelne Teile (NTuple) zerlegt. Jedes dieser NTuple beobachtet dabei `n` verschiedenen Spielfeldpositionen. Anstatt die Belohnungsfunktion für den gesamten Spielzustand zu ermitteln, wird nur die Belohnungsfunktion der einzelnen NTuple ermittelt. Diese ist davon abhängig, welche Werte die einzelnen Tuple aktuell haben. Die Belohnung für einen Spielzustand setzt sich demzufolge aus der Summe der Belohnungen der einzelnen NTuple zusammen. [Thill et al. 2012, S. 4-5]
Es gibt mehrere Methoden ein NTuple Netzwerk zu erstellen:

Random Point: Beim Random Point werden dem NTuple bei der Erstellung zufällig Spielpositionen zugewiesen, bis der NTuple die gewünschte Anzahl an Spielpositionen beobachtet. [Thill et al. 2012, S. 5]

Random Walk: Beim Random Walk wird dem NTuple die erste Spielposition ebenfalls zufällig zugewiesen. Alle weiteren Spielpositionen sind nun allerdings mit der zuletzt zugefügten Spielposition benachbart. [Thill et al. 2012, S. 5]

Fixed: Bei dieser Methode werden die NTuple nicht zufällig vom Agenten erstellt sondern vor Trainingsbeginn von Hand erstellt. Ich habe mich dazu entschieden diese Methode mit dem 7-Tuple Netzwerk zu testen, das Jaskowski [2016, S. 4] zum trainieren des aktuell besten 2048 Agenten genutzt hat. Dieses ist in Abbildung 16 dargestellt.

[19] Das Spielfeld kann 4 mal rotiert und einmal gespiegelt werden.

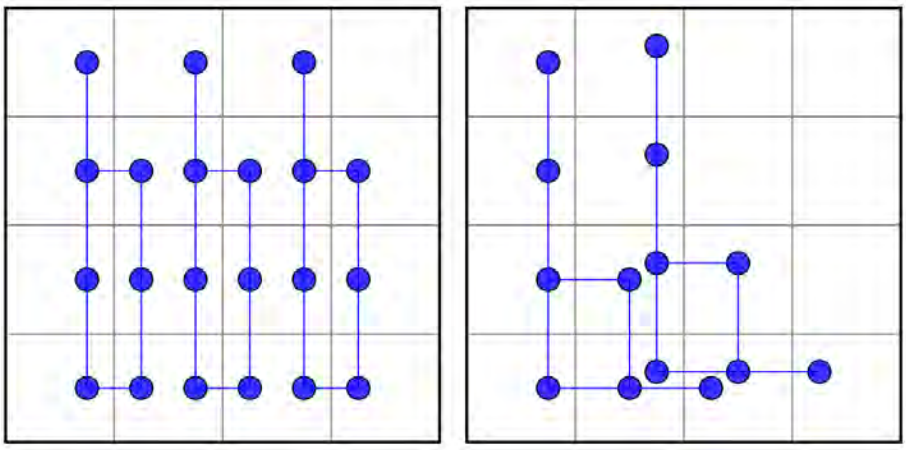


Abbildung 16: Das von Jaskowski [2016, S. 4] genutzte 7-Tuple Netzwerk. Die durch die Symmetrie entstehenden NTuple sind übersichtshalber nicht dargestellt.

6.2 Evaluation des TD-NTuple Agenten und Analyse der Parameter

Der TD-NTuple Agent erreicht durchschnittlich ungefähr `29900` Punkte am Spielende. Dabei besitzt der Agent sowohl für das Temporal Difference learning als auch für das NTuple Netzwerk eine Reihe von Einstellungsmöglichkeiten:

Anzahl an Trainingsspielen: Ähnlich wie bei der Anzahl an Iterationen kann nur schwer eine optimale Anzahl an Trainingsspielen ermittelt werden. Eine höhere Anzahl an Trainingsspielen verbessert das Ergebnis immer, braucht aber auch mehr Rechenleistung. Ich habe mich dafür entschieden zur Analyse der Parameter `100.000` Trainingsspiele und zum Trainieren des finalen Agenten `1.000.000` Trainingsspiele zu nutzen.

Temporal Difference learning:

Methode zur Anpassung der Belohnungsfunktion: Es wurden zwei Methoden zur Anpassung der Belohnungsfunktion getestet.[20] Dabei konnten, im Gegensatz zu Jaskowski [2016], mit der ersten Methode, also dem Anpassen der Belohnungsfunktionen am Spielende, deutlich bessere Ergebnisse erzielt werden.[21]

[20] Siehe S. 56.
[21] Vergleiche Abbildung 17, S. 62 mit Abbildung 29, S. 94.

epsilon: Der epsilon Wert gibt an, wie oft der Agent während des Trainings anstelle der besten Aktion eine zufällige Aktion ausführen soll. So wird sichergestellt, das der Agent nicht immer die selben Spielzustände erforscht und neue, unerforscht Spielzustände untersucht. Für epsilon ist initial ein hoher Wert von ungefähr `0,65` ideal, welcher sich im Verlauf des Trainings logarithmisch so ändert, das er am Trainingsende `0` beträgt.[22]

alpha: Der alpha Wert bestimmt die Gewichtung des Unterschiedes beim verändern der Belohnungsfunktion eines Spielzustandes. Er bestimmt also die Lerngeschwindigkeit des Agentens. Da die optimale Lernrate schwer von Hand einzustellen ist und sich im Verlauf des Trainings verändern kann, wird dieser Wert oft automatisch durch Algorithmen eingestellt. [Jaskowski 2016, S. 5] Da ein solcher Algorithmus nicht im GBG Framework integriert ist, wurde sich dafür entschieden die Lernrate dennoch von Hand einzustellen. Hier scheint ein Wert von ungefähr `0,001` optimal zu sein.[23]

gamma: Der gamma Wert gibt an wie weitsichtig der Agent sein soll. Bei einem geringem Wert wählt der Agent die Aktion aus, welche innerhalb kürzester Zeit das beste Ergebnis liefert. Bei einem hohem Wert wählt der Agent die Aktion aus, die langfristig das beste Ergebnis liefert. Hier ist ein Wert von `1`, also ein weitsichtiger Agent optimal.[24] [Sutton & Barto 1998, S. 50]

labmda: Der Wert labmda gewichtet die eligbility trace und gibt somit an, wie stark sich die Belohnungsfunktion von, in der Vergangenheit besuchten Spielzuständen, an die Belohnungsfunktion des aktuellen Spielzustandes anpassen soll. Hier ist ein Wert von `1` optimal.[25] [Sutton & Barto 1998, S. 163]

Ntuple Netzwerk:

Fixed oder Random NTuple: Ursprünglich war geplant das von Jaskowski [2016] genutzte fixe NTuple Netzwerk zum Trainieren des Agenten zu nutzen. Problematisch hierbei ist jedoch, dass bei einem NTuple Netzwerk dieser Größe ein Trainingsspiel am Anfang des Trainings bereits etwa 45 Sekunden dauert.[26] Da die Trainingsspiele im Verlauf des Trainings deutlich

[22] Siehe Abbildung 25, S. 90.
[23] Siehe Abbildung 26, S. 91.
[24] Siehe Abbildung 27, S. 92.
[25] Siehe Abbildung 28, S. 93.
[26] Auf einem Kern eines Intel Core i7 Prozessors @ 4.40GHz.

länger werden, wäre bereits bei 100.000 Trainingsspielen mit einer Trainingszeit von von wenigstens 100 Tagen zu rechnen.[27]
Als Alternative zu dem fixed NTuple Netzwerk wurde sich dafür entschieden die NTuple zufällig zu erstellen. Nach einigen schnellen Tests stellte sich heraus, das hier mit der Random Walk Methode deutlich bessere Netzwerke als mit der Random Point Methode erstellt werden können. Dies liegt vermutlich daran, dass potentielle Merges in einer Reihe und oft sogar nebeneinander liegen. Diese werden von benachbarten Tuplen besser erkannt.

Größe der Tuple: Es wurden NTuple Netzwerke mit 3 und 4-Tuplen getestet. Hierbei erziehlte das 3-Tuple Netzwerk erstaunlicherweise leicht besser Ergebnisse als das 4-Tuple Netzwerk. Da das 3-Tuple Netzwerk außerdem deutlich schneller als das 4-Tuple Netzwerk ist, wurde sich dafür entschieden dieses zum Training des Agenten zu nutzen. Ein 5-Tuple Netzwerk konnte aufgrund der langen Trainingsdauer nicht getestet werden.

Anzahl der Tuple: Sobald eine genügend hohe Anzahl von etwa 20 NTuplen erreicht ist, verbessern mehr NTuple den Agenten kaum. Da weite Tuple die Trainingszeit deutlich erhöhen, wurde sich dafür entschieden ein Netzwerk von 20 NTuplen zu nutzen.

Trainingsverlauf

In Abbildung 17 ist der Trainingsverlauf des TD-NTuple Agenten mit optimalen Parametern dargestellt. Der extreme Punkteverlust des Agentens zum Trainingsende ist vermutlich auf einen Bug zurückzuführen und wurde nicht bei der Bewertung des Agenten berücksichtigt. Da der Agent nicht von mir implementiert wurde, war es nicht möglich diesen Bug zu lokalisieren und zu beheben. Abgesehen von dem hohem Punkteabfall ist auffällig, dass der Punkteanstieg anders als zu erwarten verläuft. Es wäre zu erwarten, dass die erreichten Punkte des Agenten zu Beginn des Trainings stark ansteigen und diese Anstieg im Verlauf des Trainings stetig abnimmt. Stadtessen steigen die Punkte initial nur sehr gering an und der Anstieg nimmt im Verlauf des Trainings zu. Dies liegt vermutlich an dem initial relativ hohem epsilon Wert. Im Verlauf des Trainings

[27] Zum Vergleich: Jaskowski [2016, S. 5] nutzte für das Training seines Agentens 24 Kerne und eine Trainingszeit von etwa einer Woche.

nimmt der Anteil der Zufallsaktionen stetig ab und der Agent erreicht somit höhere Punktzahlen. Trotz der Tatsache, dass bei einem hohem epsilon Wert Spielzustände, mit einem hohem Punktestand, erst in späteren Trainingsspielen aufgerufen werden und somit nicht so gründlich evaluiert werden, ist das Training mit einem hohem epsilon Wert effektiver.[28]

Abbildung 17: Trainingsverlauf des TD-NTuple Agenten mit optimalen Einstellungen, einem 3-Tuple Netzwerk und Methode eins zur Anpassung der Belohnungsfunktion.

6.3 Zwischenfazit

Trotz der eher schlechten Ergebnisse des TD-NTuple Agenten wurde durch die Untersuchung des Agentens ein Punkt relativ deutlich. Es ist sehr schwer und mit einer enormen Menge von Rechenleistung verbunden, einen guten TD-NTuple Agenten für das Spiel 2048 zu trainieren.

[28] Siehe Abbildung 25, S. 90.

Es ist außerdem wahrscheinlich, dass im TD-NTuple Agenten aktuell noch Bugs vorhanden sind, da der Agent zum Trainingsende von fast `33000` Punkten auf etwa `1040` Punkte absinkt, was relativ genau der Performance eines reinen Zufallsagenten entspricht. Des Weiteren ist aktuell die zweite Methode zur Anpassung der Belohnungsfunktion deutlich besser als die Erste. Da dies bei Jaskowski [2016] nicht der Fall ist, wirft dies die Frage auf ob diese korrekt implementiert ist.[29]

Nachdem diese Probleme behoben wurden, kann über weitere Schritte zum verbessern des TD-NTuple Agenten nachgedacht werden. So ist eine Koppelung des TD-NTuple Agenten mit einer Expectimax Suche oder auch einem MCTSE Agenten, bei dem der Rollout durch die Belohnungsfunktion des TD-NTuple Agenten ersetzt wird, möglich. Jaskowski [2016, S. 10] hatte mit einer solchen Koppelung bereits große Erfolge.

[29] Vergleiche Abbildung 17, S. 62 mit Abbildung 29, S. 94.

7 Verbesserung der Agenten

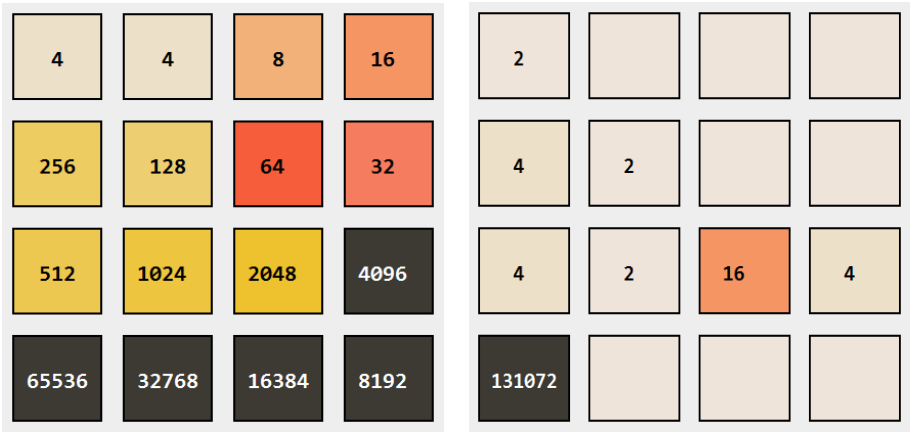
Im folgenden Kapitel werden zwei Methoden beschrieben, mit denen versucht wurde die Ergebnisse des MCTSE Agenten zu verbessern. Als erstes werden einige Heuristiken erläutert, welche zusätzlich zum Rollout bei der Ermittelung der Game Score im MCTSE Agenten eingesetzt werden können. Danach wird der Majority Vote beschrieben, bei dem mehrere Agenten parallel einen Spielzustand bewerten, was zu einer Erhöhung der Certainty des MCTSE Agenten führt. Abschließend folgt am Ende des Kapitels eine kurze Beschreibung einiger Ansätze zur Erhöhung der Geschwindigkeit der MC Agenten durch die Nutzung mehrerer CPU Kerne.

7.1 Verbesserung des MCTSE Agenten durch Heuristiken

Eine Heuristik ist eine Abschätzung des Spielausganges anhand eines Spielzustandes. Aufgrund dieser Abschätzung kann die beste Aktion für einen Spielzustand ermittelt werden oder eine Simulation innerhalb eines Agenten verbessert werden.
Die Entscheidung, ob Heuristiken während einer Simulation genutzt werden sollen, ist meist problematisch. Auf der einen Seite bringen sie Fachwissen in die Simulation, was das Ergebnis der Simulation verbessern kann, auf der anderen Seite wird nun mehr Rechenleistung benötigt, was diese verlangsamt. Diese Rechenleistung könnte auch dazu genutzt werden weitere Simulationen ohne Heuristiken durchzuführen, welche das Ergebnis ebenfalls verbessern würden. Es kann außerdem sein, dass das einer Heuristik zugrundeliegende Fachwissen nicht korrekt ist, was schwerwiegende Folgen hätte, welche im schlimmsten Fall nicht bemerkt werden.[Kutsch 2017, S. 32]
Anders als ursprünglich im Praxisprojekt angedacht, wurden die Heuristiken nicht am Ende eines Rollouts zur Veränderung der Game Score eingesetzt, sondern direkt in den MCTSE Agenten integriert. Dort ersetzen sie den Rollout und bieten eine alternative Möglichkeit einzuschätzen, wie gut der aktuelle Spielzustand ist.

7.1.1 Die Heuristiken

Aufgrund eigener Erfahrungen, einer Analyse des in Abbildung 18 dargestellten Spielzustandes und den von Xiao [2014] genutzten Heuristiken wurde 5 Heuristiken für das Spiel 2048 entwickelt. Diese besitzen eine Konstante, welche zur Gewichtung der Heuristik genutzt wird und können aktiviert bzw. deaktiviert werden.



(a) Der Spielzustand s0, eine Möglichkeit die Kacheln so anzuordnen, dass sie Problemlos ineinander geschoben werden können.

(b) Der Spielzustand s1 nachdem die Kacheln ineinander geschoben wurden.

Abbildung 18: Eine mögliche Lösung für die kritischste Spielsituation des Spieles 2048.

Bonus für potenzielle Merges Der Bonus für potenzielle Merges soll Spielzustände belohnen, bei denen zwei Kacheln mit den selben Werten nebeneinander liegen. Als Merge wird das ineinanderschieben zweier Kacheln bezeichnet. In der Heuristik werden nur Merges beachtet die tatsächlich möglich sind, wenn z.B. drei Kacheln mit dem Wert 4 nebeneinander liegen ist nur ein Merge möglich, obwohl Kacheln mit dem selben Wert in zwei Fällen nebeneinanderliegen. Die Höhe des Bonus entspricht dem Wert der Kacheln die ineinandergeschoben werden multipliziert mit einer Konstanten.

Bonus wenn die höchstes Kachel in einer Ecke liegt Diese Heuristik belohnt Spielzustände, bei denen die höchste Kachel in einer Ecke liegt. Die Be-

lohnung für diese Heuristik ist, ähnlich wie bei der "Merge" Heuristik, der Wert der Kachel multipliziert mit einer Konstante.

Bonus für eine fortlaufende, ansteigende Reihe von Zweierpotenzen Die Heuristik für eine fortlaufende, ansteigende Reihe von Zweierpotenzen erweitert die Heuristik für die höchste Kachel in einer Ecke. Wenn die höchste Kachel in einer Ecke liegt, wird geschaut ob eine fortlaufende, ansteigende Reihe von Zweierpotenzen ermittelt werden kann, welche mit dieser Kachel endet. Dabei kann diese Reihe auch Ecken beinhalten, ähnlich wie bei dem in Abbildung 18a dargestellten Spielzustand. Zur Ermittlung der Reihe wurden zwei Möglichkeiten implementiert, bei der Ersten muss die nächste Kachel in der Reihe genau den halbierten Wert aufweisen, bei der Zweiten reicht es aus, wenn die nächste Kachel einen niedrigeren Wert besitzt. Die Belohnung für eine solche Reihe ist der summierte Wert aller Kacheln dieser Reihe multipliziert mit einer Konstante. Es wird immer nur die Reihe mit dem höchsten Bonus für diese Heuristik beachtet.

Bonus für leere Felder Der Bonus für leere Felder dient dazu, den Agenten dazu zu bewegen, das Spielfeld möglichst leer zu halten und nicht das Ineinanderschieben von Kacheln zu verzögern. Es sollen also nicht lange Reihen erstellt werden, welche nie gelöst werden, da der Bonus für das Erstellen einer solchen Reihe den Punktegewinn aus dem Zusammenschieben dieser Reihe übersteigt. Der Bonus für diese Heuristik ist entweder die aktuelle Anzahl an Punkten multipliziert mit der Anzahl an leeren Feldern und einer Konstante, der Wert der höchsten Kachel multipliziert mit der Anzahl an leeren Feldern und einer Konstante oder 1 plus die Konstante zur Gewichtung der Heuristik hoch der Anzahl an leeren Feldern.

Bonus für den Rollout Die Rollout Heuristik führt einen Rollout aus, welcher genauso wie der, während der Simulation durchgeführte, Rollout des MCTSE Agenten funktioniert. Dieser Rollout bietet eine Einschätzung dafür, mit welchen Ergebnissen bei einem Weiterspielen dieses Spielzustandes zu rechnen ist. Die Belohnung der Heuristik ist die Game Score des Spielfeldes nach dem Rollout multipliziert mit einer Konstante.

7.1.2 Balancing der Heuristiken mittels einer Evolutionsstrategie

Zusätzlich zu dem in der Beschreibung der Heuristiken erwähnten Parameter, mit denen die Belohnung der Heuristik gewichtet werden kann, wurde für jede Heuristik ein boolean Wert implementiert, mit dem die Heuristik aktiviert und deaktiviert werden kann. Des Weiteren wurden für einige Heuristiken, wie etwa bei der Heuristik für leere Felder, mehrere Möglichkeiten integriert die Belohnung zu berechnen. Zwischen diesen Methoden kann ebenfalls mit einem Parameter gewechselt werden.
Insgesamt gibt es also 15 Parameter,[30] welche die Performance der Heuristiken beeinflussen. Da es sehr aufwändig ist diese Parameter genau per Hand einzustellen, wurde sich dafür entschieden dies mit einem Algorithmus zu machen. Anders als ursprünglich im Praxisprojekt angedacht wurde hierfür jedoch nicht ein genetischer Algorithmus sondern eine Covariance Matrix Adaptation Evolution Srategy (CMA-ES) genutzt. Diese wurde auch von Xiao [2014] zum balancen seiner Heuristiken genutzt. Im Gegensatz zu einem genetischem Algorithmus, bei dem sich die Population nur langsam und sehr zufällig durch die Mutation verändert, kann sich die Population bei einer CMA-ES bewegen und findet so vergleichsweise schnell ihr Optimum. In Abbildung 19 ist dieses Prinzip beispielhaft für ein zweidimensionales Problem dargestellt.[Hansen 2016]

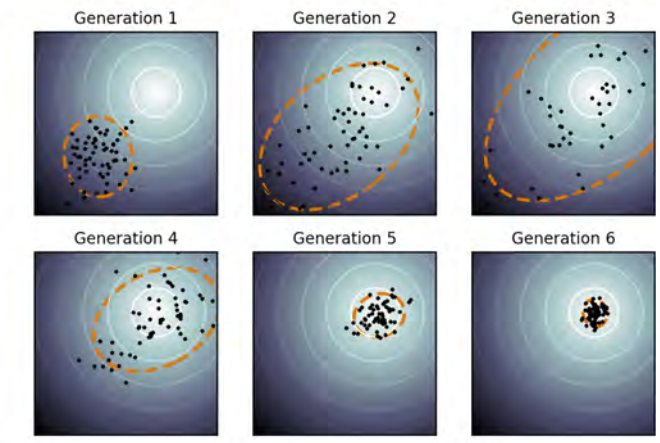
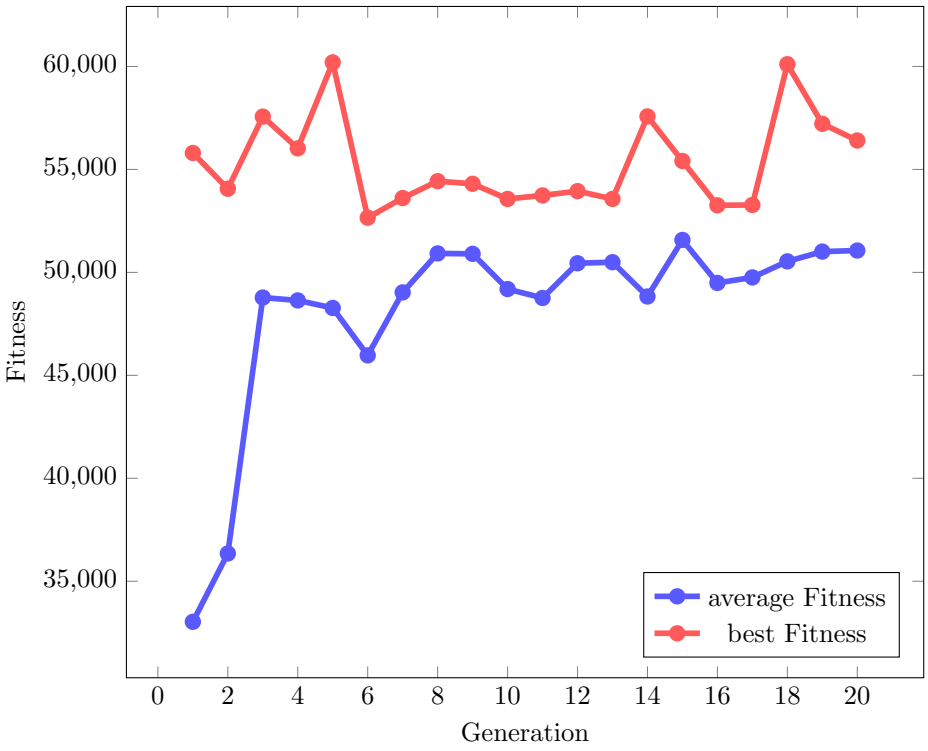


Abbildung 19: Lösung eines zweidimensionalen Problems mit der CMA-ES. [Sentenwolf 2013]

[30] Eine genaue Auflistung aller Parameter inklusive ihrer optimalen Werte befindet sich in Kapitel 9.4, S. 83.

Da die CMA-ES nur double Werte akzeptiert, mussten die Parameter der Heuristiken entsprechend umgewandelt werden. So entspricht true z.B. einem Wert `w` der kleiner als 0,5 ist und false einem `w` das größer oder gleich 0,5 ist ($w \in [0..1]$). Ein Umschalten zwischen den verschiedenen Methoden zur Ermittelung des Bonus innerhalb einer Heuristik wurde ähnlich implementiert.
In Abbildung 20 sind die Ergebnisse der CMA-ES dargestellt. Dabei ist erstaunlich wie schnell die CMA-ES herausarbeitet, welche Heuristikeinstellungen gut und schlecht sind. So wurden bereits in der fünften Generation die besten Einstellungen[31] mit einer Fitness von `60197` Punkten gefunden.



Für den MCTS Agenten mit 3500 Iterationen, einer Rolloutdepth von 150 und einer Treedepth von 10, gemessen über 25 Spiele pro Populationsmitglied. Eine Generation besteht aus 12 Mitgliedern.

Abbildung 20: Ergebnisse der CMA-ES beim Balancen der Heuristiken des MCTSE Agenten.

[31] Siehe Kapitel 9.4, S. 83.

7.1.3 Zwischenfazit

Bisher konnten durch Heuristiken im MCTSE Agenten nicht die Durchbrüche erzielt werden, welche in anderen Agenten, wie etwa dem Expectimax Agent von Xiao [2014], erreicht wurden. Trotzdem kann durch sie das Ergebnis des MCTSE Agenten, ohne einen deutlichen Anstieg an Rechenaufwand, um etwa `5%` bis `10%` erhöht werden.
Ein Grund für die vergleichsweise geringe Verbesserung könnte sein, dass die Fitness der einzelnen Mitglieder über nur 25 Spiele ermittelt wurde. Aufgrund der Natur des Spieles 2048 ist die Fitness somit relativ ungenau und weicht regelmäßig um `+-2000` Punkte von der tatsächlichen Fitness ab. Somit ist es der CMA-ES nicht möglich die besten Einstellungen exakt zu bestimmen sondern nur eine grobe Richtung vorzugeben. Es ist also anzunehmen, dass mit einer größeren verfügbaren Rechenleistung bessere Einstellungsmöglichkeiten ermittelt werden können, welche die Verbesserung durch Heuristiken weiter erhöht. Aufgrund des enormen Rechenaufwandes und der zur Verfügung stehenden Rechenleistung war dies im Rahmen dieser Arbeit jedoch nicht möglich.

7.2 Verbesserung der Certainty des MCTSE Agenten durch einen Majority Vote

Um die Certainty und somit auch die erreichte Punktzahl des MCTSE Agenten weiter zu verbessern, wurde ein Majority Vote für den MCTSE Agenten implementiert. Bei diesem wird der selbe Spielzustand mehrmals von verschiedenen MCTSE Agenten untersucht. Anschließend werden die Ergebnisse der einzelnen Agenten zusammengezählt und es wird die Aktion ausgeführt, welche von den meisten Agenten ermittelt wurde. Bei einem Gleichstand wird zufällig eine der am meisten vorgeschlagenen Aktionen ausgeführt.

Agenten	Ø Certainty	Ø Punkte
1	65,50%	56989
5	69,49%	58321
10	72,83%	60060

Tabelle 2: Certainty und Punkte des MCTSE Agenten im Majority Vote mehrerer Agenten.

Wie in Tabelle 2 ersichtlich ist, kann durch den Majority Vote die durchschnittliche Certainty und die durchschnittliche Punktzahl des MCTSE Agenten leicht erhöht werden. Dieses verbesserte Ergebnis nutzt jedoch deutlich mehr Rechenleistung, da ein Spielzustand mehrmals untersucht wird. Bei fünf Agenten wird die genutzte Rechenleistung bereits um den Faktor 5 erhöht. Dies wirft die Frage auf, ob vergleichbare Ergebnisse auch durch ein einfaches Erhöhen der Iterationen, also mit der selben genutzten Rechenleistung allerdings ohne einen Majority Vote, erreicht werden können.
Die Erhöhung der Iterationen des MCTSE Agenten um einen so großen Faktor hat vermutlich Auswirkungen auf die optimalen Werte der Parameter Maxnodes und Treedepth. Dies ist problematisch, da aufgrund der hohen Anzahl an Iterationen die Evaluationen extrem lange dauern und es einige Wochen dauern würde diese Parameter wieder optimal einzustellen. Ich habe mich deshalb dazu entschieden, die entsprechenden Parameter um den selben Faktor wie die Iterationen zu erhöhen. Dadurch bleibt das Verhältnis zwischen dem Aufbau des Baumes und Wiederholen von Evaluationen trotz einer erhöhten Anzahl an Iterationen identisch.
Die Ergebnisse dieser Evaluation sind in Tabelle 3 dargestellt.

Iterationen	Ø Certainty	Ø Punkte
3500	65,50%	56989
17500	82,41%	57236
35000	84,26%	57693

Tabelle 3: Certainty und Punkte des MCTSE Agenten bei einer erhöhten Anzahl an Iterationen.

Hierbei ist überraschend, dass eine Erhöhung der Iterationen zu einem wesentlich höherem Anstieg der Certainty führt als der Majority Vote. Erstaunlich ist auch, das bei der Erhöhung der Iterationen, trotz der deutlich höheren Certainty, im Schnitt weniger Punkte als beim Majority Vote erreicht wurden. Eine mögliche Erklärung hierfür könnte die erhöhte Anzahl an Maxnodes sein, da der Spielbaum des Agenten bei der Erhöhung der Iterationen so wesentlich größer wird als die Spielbäume des Majority Votes. Der Agent entscheidet sich so öfter für die selbe Aktion, welche jedoch nicht zwangsweise die beste Aktion ist.

Zwischenfazit

Abschließend kann gesagt werden, dass eine Erhöhung der Iterationen das Ergebnis nur sehr gering verbessert und nicht im Vergleich zur genutzten Rechenleistung steht. Auf einem Computer wo mehrere Prozesse auf verschiedenen Kernen laufen können, ohne das diese sich gegenseitig verlangsamen, wäre trotzdem ein Majority Vote sinnvoll, da dieser die Agenten auf mehrere Prozesse aufteilen könnte und so die verfügbare Rechenleistung des Computers besser nutzen könnte. Die geringe Erhöhung des Ergebnisses durch den Majority Vote wäre also ohne eine merkbare Reduzierung der Geschwindigkeit zu erreichen. Ein besseres Verhältnis von Zeit und Ergebnis des Agenten würde man jedoch vermutlich mit einem einzelnem MCTSE Agenten erreichen, welcher auf mehrere Kerne aufgeteilt ist und nur einen Baum erstellt.

7.3 Verbesserung der Geschwindigkeit der MC Agenten

Da die im GBG Framework integrierten Agenten nicht für die Nutzung mehrerer Kerne ausgelegt sind, gestalteten sich die ersten Evaluationen als sehr zeitaufwendig. Eine CPU mit 8 Kernen wurde nur zu etwa 15% ausgelastet. Es ist also wünschenswert, dass die Agenten mehrere Kerne nutzen und so die verfügbare Rechenleistung eines zeitgemäßen Mehrkernsystems besser ausnutzen können. Für dieses Problem wurden drei Lösungen entwickelt, von denen zwei erfolgreich waren:

1 Ein erster Versuch mehrere Kerne zu nutzen bestand darin, das Spiel 2048 auf mehrere Kerne aufzuteilen. Dies brachte zwar eine höhere CPU-Auslastung, allerdings brauchte die Evaluation erstaunlicherweise länger als vor dieser Änderung. Dies lag vermutlich daran, dass ein einzelner Zug sehr schnell abläuft (der MC Agent durchläuft auf einem Kern mehr als 1000 Simulationen pro Sekunde) und die Prozesse zum Verteilen der Aufgaben auf mehrere Kerne mehr Leistung benötigten als man durch das Aufteilen gewinnt.

2 Bei einem weiteren Ansatz wurden mehrere Spiele parallel gestartet und auf die verschiedenen Kerne aufgeteilt. Dies führte jedoch dazu, dass Race Conditions auftraten, da mehrere Spiele parallel auf den selben Agenten zugriffen. Dies konnte verhindert werden, indem für jedes Spiel ein

neuer Agent erzeugt wurde. Da in dem GBG Framework das Kopieren von Agenten nicht vorgesehen ist, muss dieser Vorgang für jeden Agenten einzeln implementiert werden. So konnte während der Evaluation eine CPU-Auslastung von 100% erzielt werden.

3 Da der zweite Ansatz nur während der Evaluation und nicht während des Spielens eine verbesserte CPU-Auslastung erzielt, wurde für den MC Agenten ein weiterer Ansatz umgesetzt. Hier werden erst alle Simulationen vorbereitet und anschließend für das Zufallsspielen auf verschiedene Kerne verteilt. Dies führt für den MC Agenten auch während des normalen Spielens zu einer CPU-Auslastung von fast 100%. Ein ähnlicher Ansatz wäre auch für die MCTS Agenten vorstellbar, allerdings deutlich aufwändiger, da man nicht einfach die Rollouts auf mehrere Kerne verteilen kann sondern einen Weg finden müsste den selben Spielbaum auf mehreren Prozessen zu modifizieren, ohne das Race Conditions auftreten. Eine mögliche Alternative wäre es auch einfach mehrere parallele Rollouts während der Simulation durchzuführen, so würde der Agent zwar nicht schneller, allerdings würde das Ergebnis der Rollouts genauer, wodurch das Ergebnis der MCTS Agenten vermutlich verbessert wird.

Die parallele Nutzung mehrerer Kerne wurde in allen Fällen durch den in Java integrierten `ExecutorService` implementiert. Diesem können `Callables` übergeben werden, welche Funktionen beinhalten, die auf mehreren Kernen parallel ausgeführt werden sollen. Eine Funktion eines `Callables` repräsentiert dabei im Falle des MC Agenten eine Simulation oder während der Evaluation der MCTS Agenten ein komplettes Spiel. Bevor die `Callables` an den `ExecutorService` übergeben werden können, müssen sie vorbereitet werden. Ihnen müssen die für die Simulation relevanten Parameter zugewiesen werden. Anschließend verteilt der `ExecutorService` die `Callables` auf verschiedene Kerne und führt die in ihnen enthaltenen Funktionen aus. Nachdem alle `Callables` ausgeführt wurden, werden die Ergebnisse der Simulationen zur weiteren Bearbeitung in einer Liste gespeichert. Ein ausführlich kommentiertes Beispiel einer Implementation des `ExecutorServices` anhand der Methode `getNextAction(StateObservation, vtable)` des MC Agenten befindet sich im Anhang auf Seite 84.[Kutsch 2017, S. 30-31]

8 Fazit

Abschließend werden die eingangs gestellten Forschungsfragen beantwortet und ein kurzer Ausblick auf weitere, nicht getestete, Möglichkeiten zur Verbesserung der Agenten gegeben:

Gibt es Problematiken, die für MC-basierte Agenten beim Lösen nichtdeterministischer Spiele auftreten, und wie können diese gelöst werden?

Der klassische MCTS Agent hat erhebliche Probleme damit nichtdeterministische Spiele zu lösen, da er keine nichtdeterministischen Elemente in seinem Spielbaum darstellen kann. Zur Lösung dieses Problems wurde ein MCTSE Agent entwickelt und implementiert, welcher die Möglichkeit hat einen nichtdeterministischen Spielbaum korrekt darzustellen.

Welche Ergebnisse können mit MC-basierten Agenten mit welchem Rechenaufwand für das Spiel 2048 erreicht werden?

Alle getesteten MC-basierten Agenten, also der MC, der MCTS und der MCTSE Agent, bilden regelmäßig eine Kachel mit dem Wert 2048, wodurch das Spiel 2048 gewonnen wird. Wenn man das Ziel der Agenten weiter fasst und versucht eine möglichst hohe Punktzahl zu erreichen, liegen der MC Agent (`51521` Punkte) und der MCTSE Agent (`56989` Punkte) deutlich vor dem MCTS Agenten (`34713` Punkte). Hierbei sollte beachtet werden, dass der komplexe MCTSE Agent nur leicht bessere Ergebnisse als der einfache, selbst entwickelte, MC Agent erreicht. Dabei benötigt der MCTSE Agent sogar deutlich mehr Rechenleistung als der MC Agent. Dieses Ergebnis ist ziemlich überraschend, man würde eigentlich erwarten das der komplexe MCTSE Agent deutlich bessere Ergebnisse als der einfache MC Agent erzielt.

Inwieweit können diese Ergebnisse noch verbessert werden, etwa durch die Anwendung von Fachwissen oder aber auch durch allgemeine Methoden?

Um zu schauen inwieweit die Ergebnisse des besten Agenten, dem MCTSE Agenten, weiter verbessert werden können, wurden verschiedene Verbesserungen implementiert und getestet:

Heuristiken: Durch Heuristiken wurde Fachwissen in den Agenten gebracht, der Agent kann also nicht mehr für andere Spiele eingesetzt werden. Anders als bei der von Xiao [2014] erstellten Expectimax Optimization konnte das Ergebnis des MCTSE Agenten durch die Heuristiken nur um etwa `5%` bis `10%` erhöht werden. Da sich die Justierung der Parameter als sehr aufwändig erweist, kann es jedoch sein, dass die Heuristiken noch nicht ihr volles Potential entfaltet haben. Es ist jedoch eher unwahrscheinlich, dass eine Parameterkombination gefunden wird, durch die die Heuristiken in Zukunft noch nennenswert verbessert werden können.

Majority Vote: Der Majority Vote zielt darauf ab, die Certainty des MCTSE Agenten zu verbessern. Hier zeigte sich allerdings, dass die Certainty des Agenten durch ein einfaches Erhöhen der Iterationen deutlich stärker verbessert werden kann, als durch einen Majority Vote mehrerer Agenten. Erstaunlicherweise wird jedoch die erreichte Punktzahl, trotz einer niedrigeren Certainty, durch den Majority Vote stärker erhöht als durch ein Erhöhen der Iterationen.

Zusammenfassend kann also gesagt werden, dass das Ergebnis des MCTSE Agenten, trotz mehrerer verschiedener Ansätze, nicht nennenswert verbessert werden konnte. Mit MC-basierten Agenten können im besten Fall etwas mehr als `60000` Punkte erreicht werden. Hierfür muss jedoch Fachwissen in Form von Heuristiken oder aber ein sehr hoher Rechenaufwand im Majority Vote eingesetzt werden. Ohne diese Verbesserungen werden etwa `56989` Punkte erreicht.
Eine mögliche Erklärung für dieses vergleichsweise schlechte Ergebnis ist der hohe Zufallsfaktor, gekoppelt mit dem breiten Spielbaum und einer sehr langen Spiellänge, beim Spiel 2048. Das Ergebnis eines Rollouts ist dort nur Eine, meistens sehr schlechte, von Milliarden Möglichkeiten wie das Spiel weiter verlaufen könnte. Da die Chance einen guten Spielablauf bei einem Rollout zu entdecken sehr gering ist, wählt der Agent immer den besten der simulierten schlechten Spielabläufe, allerdings selten einen wirklich guten Spielablauf. Diese These wird dadurch bestätigt, dass die vom Agenten ausgewählten Aktionen meistens sehr zufällig wirken und er nicht, wie etwa ein guter TD-NTuple Agent, eine langfristige Strategie verfolgen kann, die zum Meistern von 2048 notwendig ist.

Welche Ergebnisse erreichen MC-basierte Agenten im Vergleich zu anderen Agenten, wie etwa einem TD-Tuple-Agent, und worin bestehen die Vor- und Nachteile der MC Agenten gegenüber diesen?

Um die Ergebnisse der MC Agenten besser einordnen zu können wurde ein TD-NTuple Agent erstellt. Hierbei hat sich herausgestellt, dass das Erstellen eines solchen Agenten mit einem enormen Rechenaufwand einhergeht und ein guter Agent nicht mit den zur Verfügung stehenden Mitteln erstellt werden konnte.

Aus diesem Versuch lässt sich ein großer Vorteil des MCTSE Agenten erkennen, da dieser im Vergleich zum langen Training eines TD-NTuple Agenten nicht trainiert werden muss. Der MCTSE Agent braucht jedoch während des Spielens mehr Rechenleistung, was zu einem langsameren Spielen des Agenten führt, diese ist im Vergleich zum Training des TD-NTuple Agenten allerdings sehr gering.

Zusammenfassend kann also gesagt werden, dass MC Agenten eine schnelle Möglichkeit bietet um mittelmäßige Erfolge in 2048 zu erreichen, allerdings nicht mit einem langen und gut trainierten TD-NTuple Agenten, wie dem von Jaskowski [2016], mithalten können. Wenn jedoch nur wenig Rechenleistung zur Verfügung steht bieten sie eine adäquate Alternative zum Trainieren eines TD-NTuple Agenten.

Da es nicht Ziel der Arbeit war den weltbesten 2048 Agenten zu erstellen, sondern zu ermitteln, welche Ergebnisse generelle, nicht auf 2048 spezialisierte Agenten, die allein durch Self-Play lernen, für das Spiel 2048 erreichen und welche Stärken und Schwächen sie aufweisen, kann man trotz der, im Vergleich zu anderen Arbeiten, eher mittelmäßigen Ergebnisse sagen, dass das Ziel der Arbeit erreicht wurde.

Ausblick

Es wurden in der Arbeit bereits viele Möglichkeiten getestet, die MC-basierten Agenten zu verbessern, sodass nur noch an wenigen Stellen Potential zur Verbesserung der Agenten vorhanden ist. Eine dieser Stellen ist die in Kapitel 7.3 angeschnittene parallele Nutzung mehrerer Kerne der CPU. Die Geschwindigkeit des MCTSE Agenten könnte erhöht werden indem er, ähnlich wie der MC Agent, für die Nutzung mehrerer Kerne verändert wird. Hierdurch würde der Agent, bei einem gängigen Rechner mit 8 Kernen etwa um den Faktor 8 schneller. Die hierdurch gewonnene Rechenleistung könnte dazu eingesetzt werden, den Majority Vote ohne größere Einbußen in der Geschwindigkeit des Agenten zu nutzen.
Des Weiteren könnte noch getestet werden, ob das Ergebnis der MC-basierten Agenten durch eine Kombination aus Heuristiken und Majority Vote weiter verbessert werden kann. Wahrscheinlich ist jedoch, dass die MC Agenten bei etwa `60.000` Punkten ihr Maximum erreichen und bessere Ergebnisse nur mit anderen Ansätzen erreicht werden können.
Anders als bei den MC Agenten bieten sich für die TD-NTuple Agenten noch viele Möglichkeiten zur Verbesserung an. So könnte der TD-NTuple Agent mit einem Expectimax oder sogar einem MCTSE Agenten, bei dem der Rollout durch die Belohnungsfunktion des TD-NTuple Agenten ersetzt wird, gekoppelt werden. Diese Methode wurde bereits von Jaskowski [2016] erfolgreich eingesetzt. Ein weiterer Grund für die eher schwachen Ergebnisse des TD-NTuple Agenten könnte durch mehr verfügbare Rechenleistung sowie einem Training, das mehrere Kerne nutzt, behoben werden, da es so möglich wäre den Agenten über mehr Trainingsspiele zu trainieren. Die Anzahl an Trainingsspielen die in dieser Arbeit absolviert wurden, waren im Vergleich zu Jaskowski [2016] sehr gering.

9 Anhang

9.1 Normalisierung der Certainty

Die Certainty sagt aus, wie oft ein Agent sich bei der Evaluierung des selben Spielzustandes für die selbe, vermutlich beste, Aktion entscheidet, also wie sicher sich der Agent bei der Auswahl einer Aktion ist.
Wenn einem Agenten ein Spielzustand mit zwei verfügbaren Aktionen `(a, b)` übergeben wird und er wählt Aktion `a` in `75%` der Fälle und Aktion `b` in `25%` der Fälle, ergibt sich daraus eine Certainty von `75%`. Wenn ihm ein Spielzustand mit vier verfügbaren Aktionen `(a, b, c, d)` übergeben wird und er wählt Aktion `a` in `70%` der Fälle und Aktion `b, c und d` in jeweils `10%` der Fälle, ergibt sich daraus eine Certainty von `70%`.
Da sich diese Certainty bei zwei möglichen Aktionen zwischen `50%` und `100%` und bei vier möglichen Aktionen zwischen `25%` und `100%` befindet, ist sie nicht genormt. Es ist außerdem nicht sofort ersichtlich, wo ein Agent sich im Vergleich zu einem Zufallsagenten, welcher bei zwei verfügbaren Aktionen eine Certainty von `50%` und bei vier verfügbaren Aktionen eine Certainty von `25%` erreichen würde, befindet. Um diese Probleme zu beheben, wurde die Certainty `(`C`)` mithilfe der Formel 10 in eine normalisierte Certainty (C_n) umgerechnet.

$$C_n = \frac{nC - 1}{n - 1} \tag{10}$$

C_n ist die normalisierte Certainty, wenn C die Certainty und n die Anzahl an verfügbaren Aktionen sind.

Die normalisierte Certainty hat, unabhängig von der Anzahl an verfügbaren Aktionen, einen Wert von `0%`, wenn sich der Agent äquivalent zu einem Zufallsagenten verhält, also jede Aktion gleich oft auswählt, und einen Wert von `100%`, wenn der Agent jedes mal die selbe Aktion auswählt. Wenn einem Agenten also ein Spielzustand mit zwei verfügbaren Aktionen `(a, b)` übergeben wird und er wählt Aktion `a` in `75%` der Fälle und Aktion `b` in `25%` der Fälle, ergibt sich daraus eine normalisierte Certainty von `50%`. Wenn ihm ein Spielzustand mit vier verfügbaren Aktionen `(a, b, c, d)` übergeben wird und er wählt Aktion `a` in `70%` der Fälle und Aktion `b, c und d` in jeweils `10%` der Fälle, ergibt sich daraus eine normalisierte Certainty von $66,\overline{66}\%$.
Außerhalb dieses Kapitels ist, wenn nicht anders angegeben, mit dem Begriff

Certainty nicht die tatsächlich ermittelte Certainty sondern die besser verständliche normalisierte Certainty gemeint. [Kutsch 2017, S. 23-24]

9.2 Verschiedene Möglichkeiten zur Ermittlung der Game Score und ihre Auswirkungen auf den MC Agenten

Auf den ersten Blick scheint die Game Score, also das Ergebnis einer Simulation, eine relativ simple Angelegenheit zu sein. Wenn das Spiel gewonnen wurde, wird eine positive Belohnung, also z.B. 1 zurückgegeben, ist das Spiel verloren wird eine negative Bestrafung, also z.B. -1, zurückgegeben. Ein Unentschieden würde dann ein Ergebnis von 0 liefern. Komplizierter wird dies jedoch, wenn es nicht Ziel des Spieles ist, das Spiel zu gewinnen, sondern eine möglichst hohe Punktzahl zu erreichen, wie es in dem Spiel 2048 der Fall ist.
Eine traditionelle Ermittelung der Game Score würde hier dazu führen, dass diese anfangs fast ausschließlich einer Bestrafung von -1 entsprechen würde, da es aufgrund des tiefen Spielbaumes extrem unwahrscheinlich ist mit einer Zufallssimulation das Spiel zu gewinnen, also eine Kachel mit dem Wert 2048 zu erstellen. Dadurch werden alle verfügbaren Aktionen gleich bewertet und der Agent würde zufällige Aktionen ausführen. Mit sehr viel Glück überlebt der Agent lange genug um diese Phase zu verlassen und es gelingt den ersten Simulationen, das Spiel zu gewinnen. Diese Simulationen liefern nun ein Ergebnis von 1, wodurch die nächste Aktion nicht mehr zufällig gewählt wird. Sobald der Agent diesen Punkt erreicht hat, stehen die Chancen relativ gut dafür, dass er das Spiel gewinnt. Nachdem der Agent das Spiel gewonnen hat, liefern jedoch alle Simulationen ein Ergebnis von 1. Die nächste Aktion wird also wieder zufällig gewählt und das Spiel wird relativ schnell beendet.
Als Alternative zur traditionellen Variante habe ich mich dafür entschieden, die Game Score mit dem tatsächlichen Punktestand des Spieles zu verknüpfen.[32] Um das traditionelle Schema von -1 für das schlechteste Ergebnis und +1 für das beste Ergebnis beizubehalten habe ich mich dafür entschieden, den Punktestand des Spieles durch den maximal möglichen Punktestand von 3.932.156 Punkten [Asusfood 2015] zu teilen und so einen normalisierten Punktestand `PN` zwischen

[32] Ein ähnliches Prinzip wurde auch von Roelofs [2012] für das Spiel Siedler von Catan angewandt. Hier sind die Siegespunkte eines Spielers ein wichtiger Faktor der Game Score.

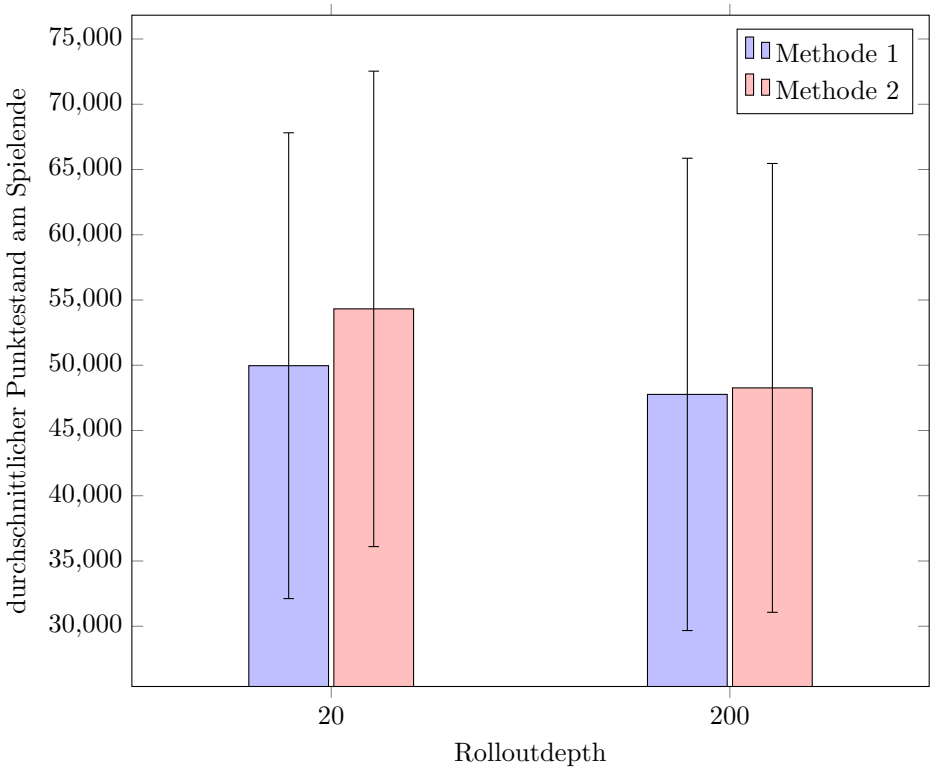
0 und 1 zu erhalten. Anschließend wurden zwei Methoden zur Ermittelung der Game Score entwickelt:

Methode 1: Bei dieser Methode wird nicht zwischen einer beendeten und einer nicht beendeten Simulation unterschieden. Das Ergebnis einer Simulation ist immer der normalisierte Punktestand `PN`. Diese Methode wurde ebenfalls von Jaskowski [2016] eingesetzt.

Methode 2: Bei der zweiten Methode wird zwischen einer beendeten und einer nicht beendeten Simulation unterschieden. Das Ergebnis einer beendeten Simulation ist `-1+PN`, das Ergebnis einer nicht beendeten Simulation ist `PN`. Eine Simulation wird dabei als beendet betrachtet, wenn für ihren Spielzustand keine Aktion mehr verfügbar ist. Für eine nicht beendete Simulation sind noch Aktionen verfügbar. Dies kann nur geschehen, wenn eine Simulation ihre maximale Rolloutdepth erreicht. Im Gegenzug zu der ersten Methode werden so Simulationen, welche noch nicht beendet wurden, also noch nicht ihren maximal möglichen Punktestand erreicht haben, deutlich stärker belohnt, als solche welche diesen während der Simulation erreichen und kein Potential für einen höheren Punktestand aufweisen

Ein Vergleich beider Methoden für den MC Agenten wird in der Abbildung 21 aufgezeigt. Für eine hohe Rolloutdepth sind kaum Unterschiede zwischen den Methoden erkennbar, bei einer niedrigen Rolloutdepth zeigt sich jedoch, dass mit der zweiten Methode deutlich bessere Ergebnisse erzielt werden können. Die fehlenden Unterschiede bei einer Rolloutdepth von 200 lassen sich dadurch erklären, dass bei dieser Rolloutdepth nur sehr selten eine Simulation nicht zu Ende gespielt wird und sich die Methoden deshalb nicht unterscheiden.[33] Eine ähnliches Tendenz konnte auch für den MCTS und MCTSE Agenten festgestellt werden, mit dem Unterschied, dass diese bessere Ergebnisse bei einer hohen Rolloutdepth erreichen.

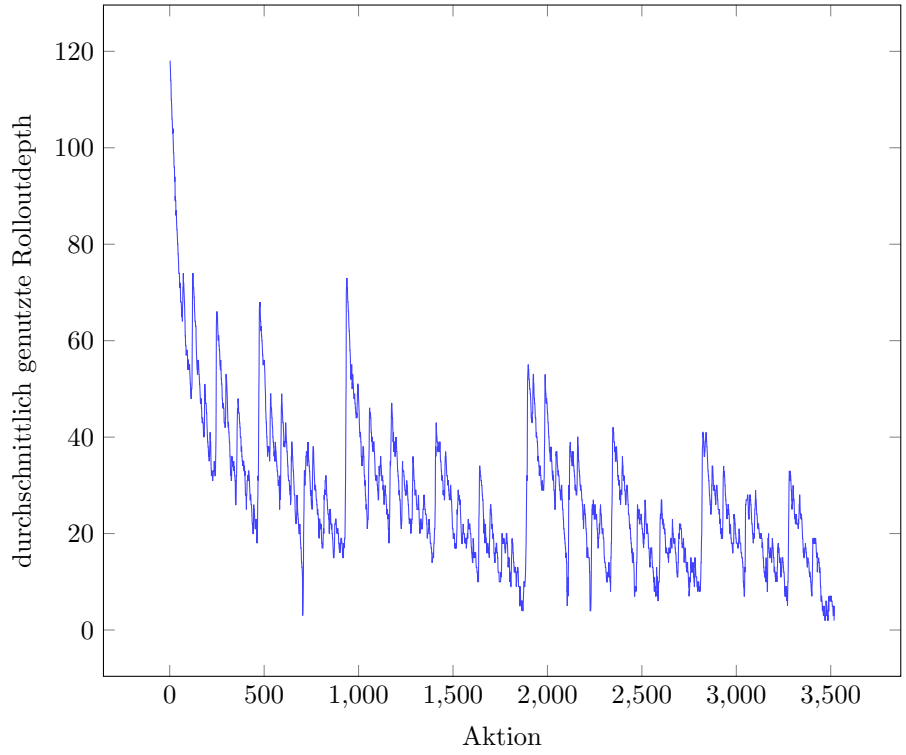
[33] siehe Abbildung 22, S. 81.



Für den MC Agenten mit 1000 Iterationen, gemessen über 50 Spiele. Die Fehlerbalken stellen eine Standartabweichung dar.

Abbildung 21: Vergleich der Methoden zur Ermittelung der Game Score für den MC Agenten bei einer Rolloutdepth von 20 und 200.

9.3 Tatsächlich genutzte Rolloutdepth der Monte Carlo Agenten



Für den MC Agenten mit 1000 Iterationen und einer Rolloutdepth von 10000 (quasi Endlos), gemessen über 1 Spiel mit ≈ 80000 Punkten.

Abbildung 22: Durchschnittliche genutzte Rolloutdepth des MC Agenten.

In Abbildung 22 wurde die durchschnittlich genutzte Rolloutdepth des MC Agenten für jede Aktion über ein Spiel von ≈ 80.000 Punkten gemessen. Da der Rollout bei allen Monte Carlo Agenten (also MC, MCTS und MCTSE) gleich abläuft, ist bei einer Messung der genutzten Rolloutdepth der anderen Agenten ein ähnliches Ergebnis zu erwarten. Auffällig ist, dass diese Rolloutdepth starken Schwankungen unterliegt. Schaut man sich den Graphen bei der ≈ 475, ≈ 950 und ≈ 1900 Aktion an, verdreifacht sich die durchschnittlich genutzte Rolloutdepth von weniger als 20 auf mehr als 60. An diesen Stellen wird eine kritische Spielsituation, also das Bilden einer Kachel mit dem Wert 1024 (bei der ≈ 475 Aktion), 2048 (bei der ≈ 950 Aktion) oder 4096 (bei der

≈ 1900 Aktion), erfolgreich gelöst und es entsteht wieder viel Platz auf dem Spielfeld. Demzufolge können die Zufallssimulationen länger überleben und die Rolloutdepth steigt an. In abgeschwächter Form kann dieses Phänomen auch an weiteren Punkten beobachtet werden, etwa bei der ≈ 1400 Aktion, wo der Agent zusätzlich zu der bereits vorhandenen Kachel mit dem Wert 2048 eine Kachel mit dem Wert 1024 erstellt. Zum Spielende steht der Agent ebenfalls kurz vor einer kritischen Spielsituation, dem Bilden der Kachel mit dem Wert 8192, jedoch schafft er dies nicht erfolgreich. Diese Beobachtung ist interessant, da der Agent eine hohe Rolloutdepth nur in nicht kritischen Situationen nutzt, bei denen eine Aktion selten spielentscheidend ist. In den kritischen Situationen, wo die nächsten Züge darüber entscheiden, ob das Spiel weitergeht oder beendet wird, ist die tatsächlich genutzte Rolloutdepth des Agenten sehr gering. [Kutsch 2017, S. 28-29]

9.4 Parameter zur Justierung der Heuristiken

```
1  //empty tiles
   public boolean enableEmptyTiles = true;
   public double emptyTilesWeighting0 = 0;
   public double emptyTilesWeighting1 = 0.7214131449043863;
5  public double emptyTilesWeighting2 = 0;
   public int emptyTilesMethod = 1;
   //valid methods are
   //0 => numeEmptyTiles^weighting,
   //1 => highestTileValue * numEmptyTile * weighting,
10 //2 => score * numEmptyTile * Weighting

   //highest tile in corner
   public boolean enableHighestTileInCorner = true;
   public double highestTileIncornerWeighting = 0.48066258192020583;
15

   //row
   public boolean enableRow = true;
   public double rowWeighting0 = 0.4778659656963287;
   public double rowWeighting1 = 0;
20 public int rowMethod = 0;
   //valid methods are
   //0 => tile in a row has to be lower then the previous tile,
   //1 => tile in a row is exactly half of the previous tile

25 //merge
   public boolean enableMerge = true;
   public double mergeWeighting = 0.8079838161879548;

   //rollout
30 public boolean enableRollout = true;
   public double rolloutWeighting = 0.8435028456190605;
```

9.5 parallele Simulation des MC Agenten

```
1  //der Executor Service, welcher die Prozesse auf verschiedene
   //Kerne verteilt, ein WorkStealingPool nutzt dafuer die
   //gesamte CPU Leistung
   private ExecutorService executorService = Executors.
       newWorkStealingPool();
5
   /**
    * Get the best next action and return it
    * @param sob           current game state (not changed on return)
    * @param vtable        must be an array of size n+1 on input,
10  *                      where n=sob.getNumAvailableActions(). On
    *                      output, elements 0,...,n-1 hold the score
    *                      for each available action (corresponding
    *                      to sob.getAvailableActions()). In addition,
    *                      vtable[n] has the score for the best action
15  * @return nextAction   the next action
    */
   public Types.ACTIONS getNextAction (StateObservation sob, double[]
       vtable) {
       //die Funktionen, welche auf die verschiedenen Kerne verteilt
       //werden sollen
20     List<Callable<ResultContainer>> callables = new ArrayList<>();
       //das Ergebnis dieser Funktionen
       List<ResultContainer> resultContainers = new ArrayList<>();
       //alle fuer den Spielzustand verfuegbaren Aktionen
       List<Types.ACTIONS> actions = sob.getAvailableActions();
25
       //es ist nur eine Aktion verfuegbar, diese kann sofort
       //zurueckgegeben werden
       if(sob.getNumAvailableActions() == 1) {
           return actions.get(0);
30     }

       //die Funktionen, welche anschliessend auf die verschiedenen
       //Kerne verteilt werden sollen, werden erstellt
       //fuer jede Iteration wird eine Funktion mit jeder verfuegbaren
35     //Aktion erstellt
       for (int j = 0; j < Config.ITERATIONS; j++) {
           for (int i = 0; i < sob.getNumAvailableActions(); i++) {

               //eine Kopie des Spielzustandes wird erstellt
40             StateObservation newSob = sob.copy();
```

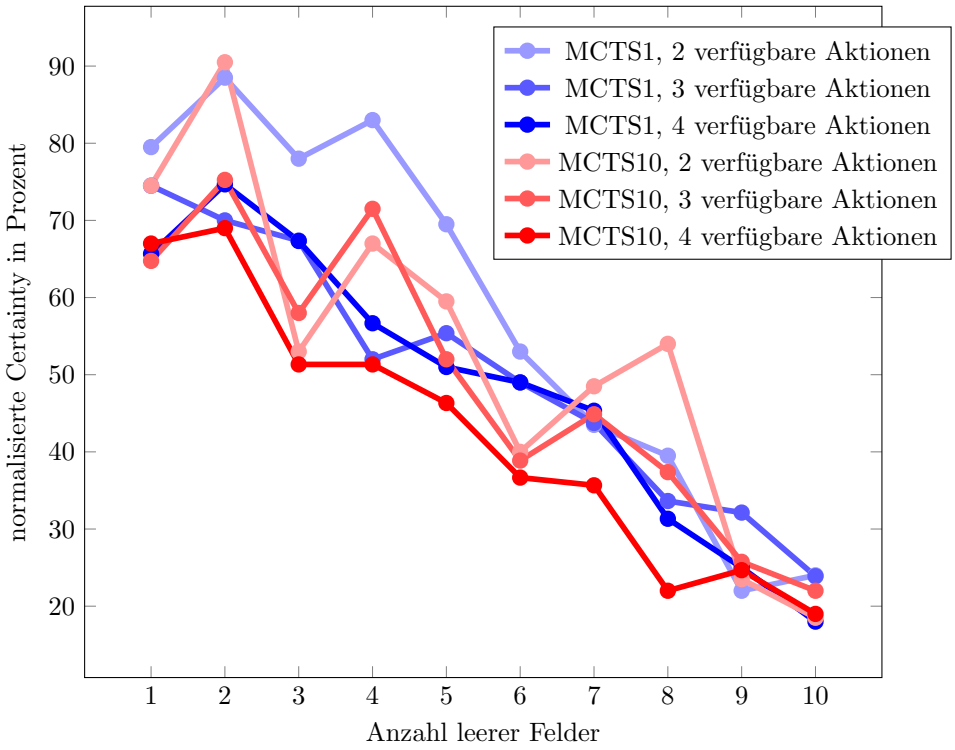
```
            //die Kennzeichnung der ersten Aktion muss
            //Zwischengespeichert werden, da auf den aktuellen
            //Wert der for-Schleife zum Zeitpunkt der Ausfuehrung
45          //des callables nicht mehr zugegriffen werden kann
            int firstActionIdentifier = i;

            //Die callables, also die Funktionen die spaeter auf
            //mehreren Kernen parallel ausgefuehrt werden, werden
50          //erstellt. Die callables werden hier nur erstellt und
            //erst mit dem Ausfuehren der Funktion
            //invokeAll(callables) auf executorService in Zeile 79
            //durchlaufen
            callables.add(() -> {
55
                //die erste Aktion wird ermittelt und auf dem
                //Spielzustand ausgefuehrt
                Types.ACTIONS firstAction = actions.get(
                    firstActionIdentifier);
                newSob.advance(firstAction);
60
                //der Random Agent wird erstellt und simuliert
                //ein Spiel
                RandomSearch agent = new RandomSearch();
                agent.startAgent(newSob);
65
                //das Ergebnis der Simulation wird in einem
                //ResultContainer zurueckgegeben
                return new ResultContainer(staticI, newSob);
            });
70      }
    }


    try {
75      //Der executorService wird aufgerufen und verteilt die
        //zuvor erstellen Simulationen auf alle Kerne der CPU.
        //Die Ergebnisse der Simulationen werden in einen stream
        //geschrieben.
        executorService.invokeAll(callables).stream().map(future->{
80          try {
                return future.get();
            }
            catch (Exception e) {
                throw new IllegalStateException(e);
```

```
85              }

                //jedes Ergebnis das in den stream geschrieben wurde wird
                //in der Liste resultContainers gespeichert
                }).forEach(resultContainers::add);
90          } catch (InterruptedException e) {
                e.printStackTrace();
            }

            //jeder resultContainer in der Liste resultContainers wird
95          //der zugehoerigen Aktion im vtable hinzuadiert
            for(ResultContainer resultContainer : resultContainers) {
                vtable[resultContainer.firstAction] += resultContainer.sob.
                    getGameScore();
                }
            }
100
            //hier wird die naechste Aktion sowie ihre Bewertung
            //gespeichert
            Types.ACTIONS nextAction = null;
            double nextActionScore = Double.NEGATIVE_INFINITY;
105
            //es wird jede im vtable gespeicherte Aktion durchlaufen
            for (int i = 0; i < sob.getNumAvailableActions(); i++) {

                //es wird die durchschnittliche Score jeder Aktion gebildet
110             vtable[i] /= Config.ITERATIONS;

                //es wurde eine Aktion mit einer hoeheren Score gefunden
                if (nextActionScore < vtable[i]) {
                    nextAction = actions.get(i);
115                 nextActionScore = vtable[i];
                }
            }

            //die beste Aktion wird zurueckgegeben
120         return nextAction;
        }
```

9.6 Interface StateObservationNondeterministic

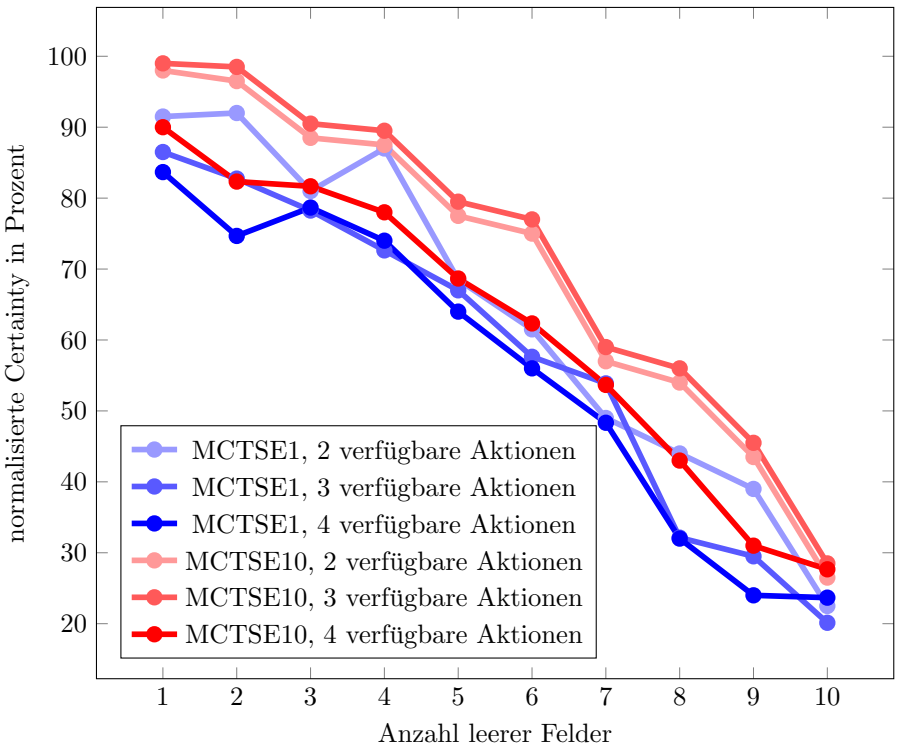
```
/**
 * This interface is used to implement nondeterministic games like
     2048. It extends the normal StateObservation interface.
 *
 * Created by Johannes on 17.05.2017.
 */
public interface StateObservationNondeterministic extends
    StateObservation {
    /**
     * Advance the current state to a new state using a
         deterministic Action
     *
     * @param action the action
     */
    void advanceDeterministic(Types.ACTIONS action);

    /**
     * Advance the current state to a new state using a
         nondeterministic Action
     */
    void advanceNondeterministic();

    /**
     * Get the next nondeterministic action
     *
     * @return the action, null if the next action is deterministic
     */
    Types.ACTIONS getNextNondeterministicAction();

    StateObservationNondeterministic copy();
}
```

9.7 Weitere Evaluationsergebnisse



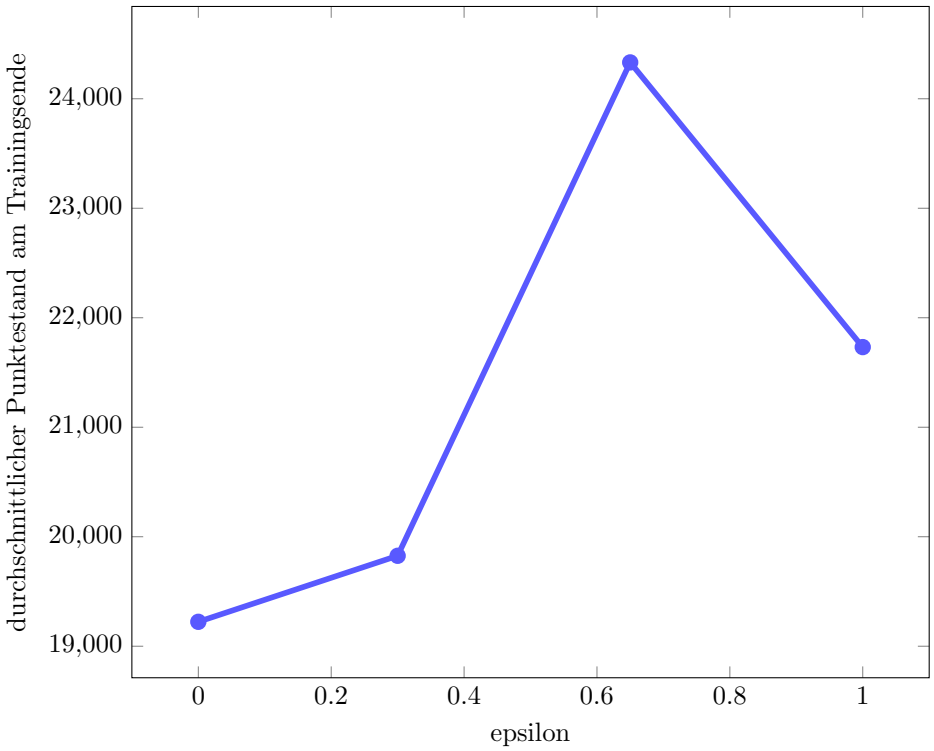
Für den MCTS Agenten mit 1000 Iterationen pro verfügbare Aktion, einer Rolloutdepth von 150 und einer Treedepth von 1 oder 10, gemessen über 20 Spielzustände je Knotenpunkt.

Abbildung 23: Normalisierte Certainty für verschiedene Spielzustände beim MCTS Agent mit einer Treedepth von 1 oder 10.



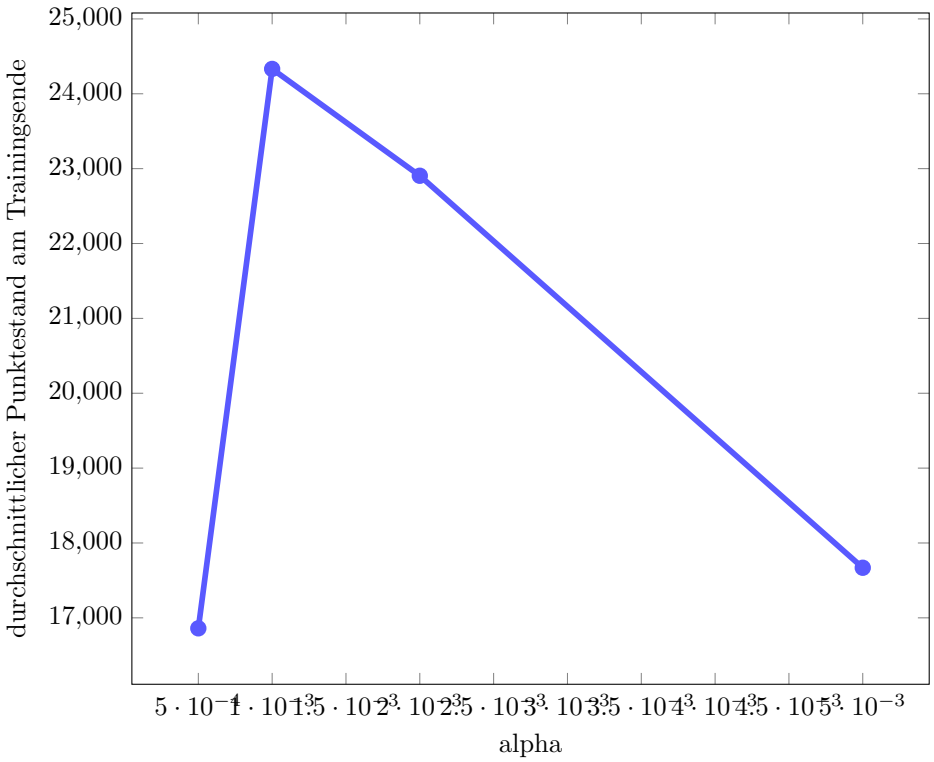
Für den MCTSE Agenten mit 1000 Iterationen pro verfügbare Aktion, einer Rolloutdepth von 150, 500 Maxnodes, einer Treedepth von 1 oder 10 und UCTN als Selektionsstrategie, gemessen über 20 Spielzustände je Knotenpunkt.

Abbildung 24: Normalisierte Certainty für verschiedene Spielzustände beim MCTS Agent mit einer Treedepth von 1 oder 10.



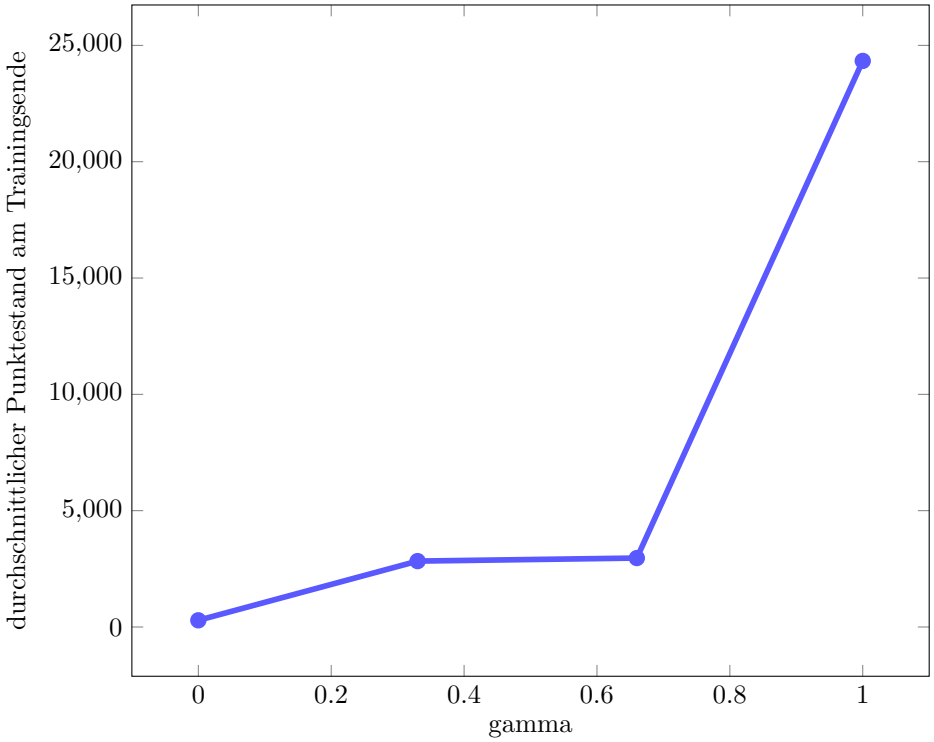
Für den TD-Ntuple Agenten über 100.000 Trainingsspiele mit einem alpha von 0,001, gamma von 1 und lambda von 0.

Abbildung 25: Durchschnittliche Punktzahl des TD-Ntuple Agenten am Trainingsende für verschiedene epsilon Werte.



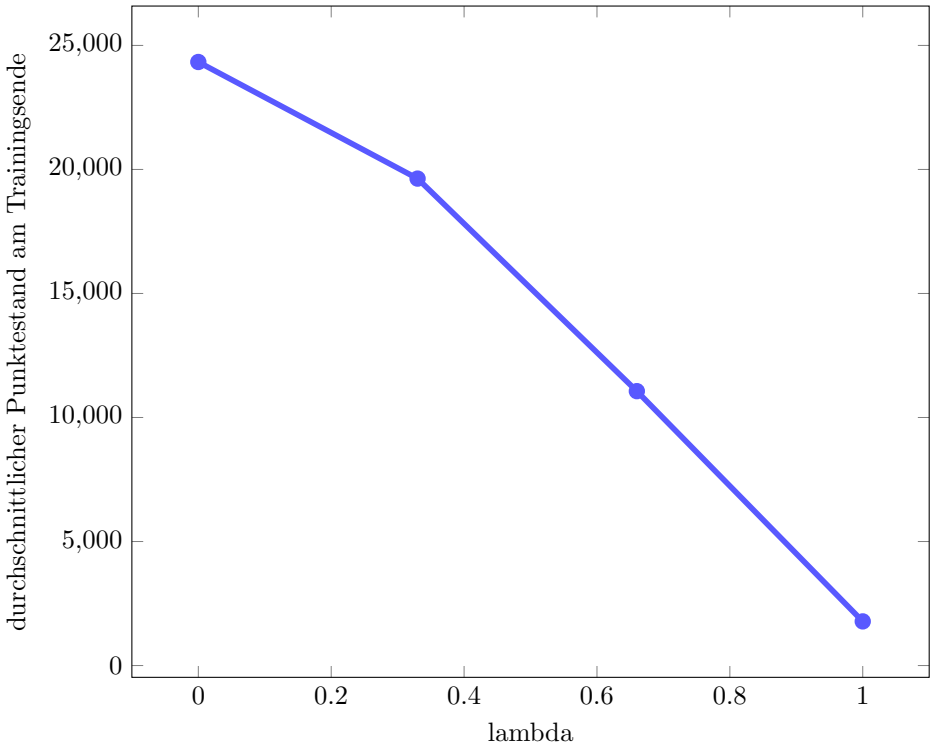
Für den TD-Ntuple Agenten über 100.000 Trainingsspiele mit einem epsilon von 0,65, gamma von 1 und lambda von 0.

Abbildung 26: Durchschnittliche Punktzahl des TD-Ntuple Agenten am Trainingsende für verschiedene alpha Werte.



Für den TD-Ntuple Agenten über 100.000 Trainingsspiele mit einem epsilon von 0,65, alpha von 0,001 und lambda von 0.

Abbildung 27: Durchschnittliche Punktzahl des TD-Ntuple Agenten am Trainingsende für verschiedene gamma Werte.



Für den TD-Ntuple Agenten über 100.000 Trainingsspiele mit einem epsilon von 0,65, alpha von 0,001 und gamma von 1.

Abbildung 28: Durchschnittliche Punktzahl des TD-Ntuple Agenten am Trainingsende für verschiedene lambda Werte.

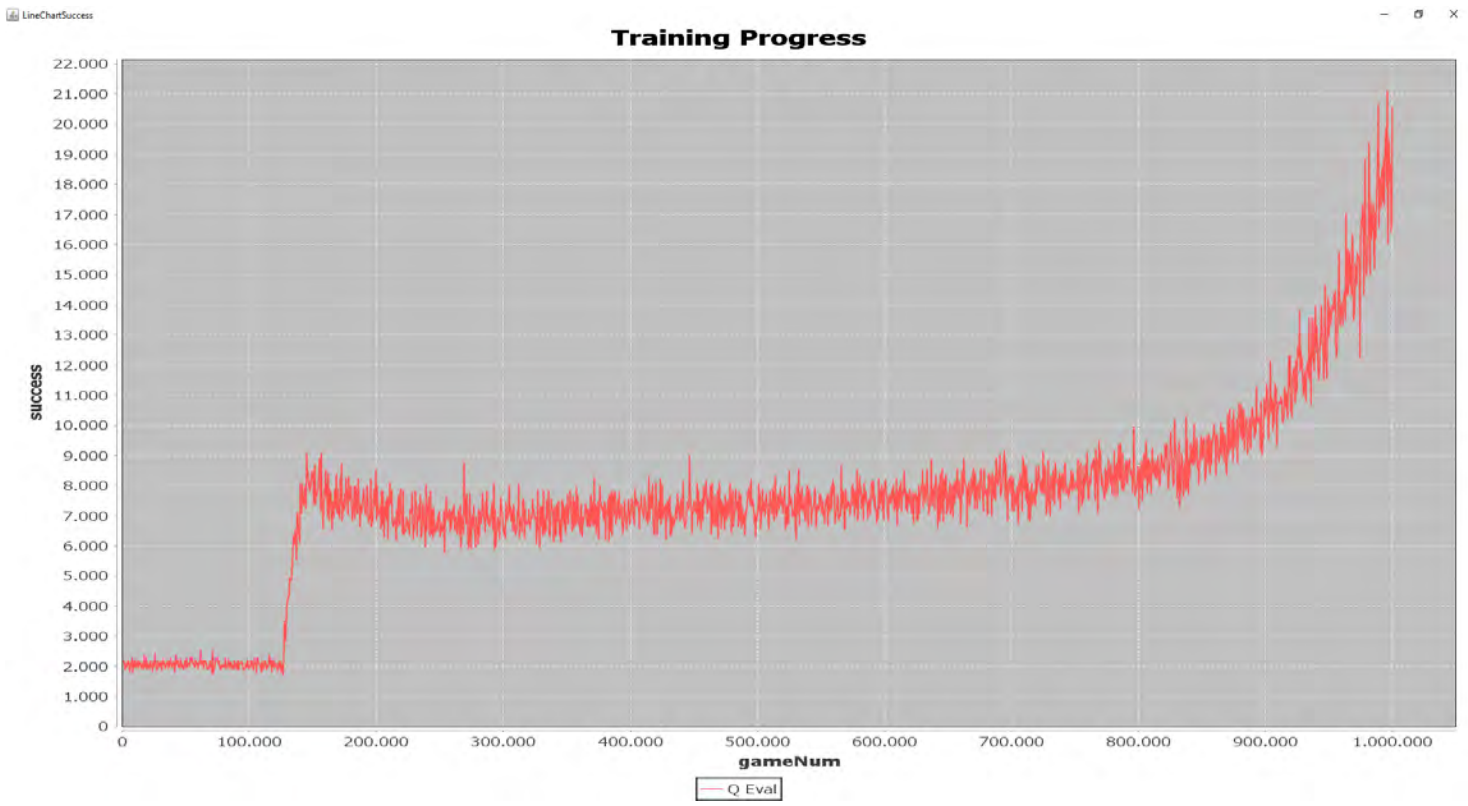


Abbildung 29: Trainingsverlauf des TD-NTuple Agenten mit optimalen Einstellungen, einem 3-Tuple Netzwerk und Methode zwei zur Anpassung der Belohnungsfunktion.

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Gummersback, 18. August 2017

Johannes Kutsch

Literatur

Anderson 1986

Anderson, Herbert L.: *Metropolis, Monte Carlo and the Maniac*. https://web.archive.org/web/20170215075934/http://library.lanl.gov/cgi-bin/getfile?00326886.pdf, 1986. – Aufgerufen: 16.07.2017

Asusfood 2015

Asusfood: *Highest possible Score for 2048*. https://www.reddit.com/r/2048/comments/214njx/highest_possible_score_for_2048_warning_math/, 2015. – Aufgerufen: 16.07.2017

Van den Broeck et al. 2009

Broeck, Guy Van d.; Driessens, Kurt; Ramon, Jan: Monte-Carlo tree search in poker using expected reward distributions. In: *Advances in Machine Learning* (2009), pages 367–381

Browne et al. 2012

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; Colton, S.: A Survey of Monte Carlo Tree Search Methods. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4 (2012), Nr. 1, pages 1–43. http://dx.doi.org/10.1109/TCIAIG.2012.2186810. – DOI 10.1109/TCIAIG.2012.2186810. – ISSN 1943–068X

Chao-Chin et al. 2014

Chao-Chin, Liang; Kun-Hao, Yeh; I-Chen, Wu: *CGI-2048*. https://web.archive.org/web/20140715061243/http://2048.aigames.nctu.edu.tw/, 2014. – Aufgerufen: 16.07.2017

Chaslot 2010

Chaslot, Guillaume Maurice Jean-Bernard: *Monte-Carlo Tree Search*. PhD Thesis, Maastricht University, https://web.archive.org/web/20170308132443/https://project.dke.maastrichtuniversity.nl/games/files/phd/Chaslot_thesis.pdf, 2010. – Aufgerufen: 16.07.2017

Cirulli 2014

Cirulli, Gabriele: *2048*. https://gabrielecirulli.github.io/2048/, 2014. – Aufgerufen: 16.07.2017

Hansen 2016

Hansen, Nikolaus: *The CMA Evolution Strategy*. https://www.lri.fr/~hansen/cmaesintro.html, 2016. – Aufgerufen: 25.07.2017

Jaskowski 2016

Jaskowski, Wojciech: *Mastering 2048 with Delayed Temporal Coherence Learning, Multi-State Weight Promotion, Redundant Encoding and Carousel Shaping*. IEEE Transactions on Computational Intelligence and AI in Games, https://www.researchgate.net/publication/301877490_Mastering_2048_with_Delayed_Temporal_Coherence_Learning_Multi-State_Weight_Promotion_Redundant_Encoding_and_Carousel_Shaping, 2016. – Aufgerufen: 16.07.2017

Konen 2017a

Konen, Wolfgang: *private Kommunikation*. 2017

Konen 2017b

Konen, Wolfgang: The GBG Class Interface Tutorial: General Board Game Playing and Learning / Research Center CIOP (Computational Intelligence, Optimization and Data Mining). Version: June 2017. http://www.gm.fh-koeln.de/ciopwebpub/Kone17a.d/TR-GBG.pdf. Cologne University of Applied Science, June 2017. – Technical Report

Kutsch 2017

Kutsch, Johannes: *KI-Agenten für das Spiel 2048: Untersuchung von Monte Carlo Algorithmen*. Praxisprojekt-Dokumentation, Cologne University of Applied Science, 2017

Mehta 2014

Mehta, Rahul: 2048 is (PSPACE) hard, but sometimes easy. In: *CoRR* abs/1408.6315 (2014). http://arxiv.org/abs/1408.6315

Olson 2015

Olson, Randy: *Artificial Intelligence has crushed all human records in 2048. Here's how the AI pulled it off*. http://www.randalolson.com/2015/04/27/artificial-intelligence-has-crushed-all-human-rec

ords-in-2048-heres-how-the-ai-pulled-it-off/, 2015. – Aufgerufen: 16.07.2017

Roelofs 2012

ROELOFS, G: *Monte carlo tree search in a modern board game framework.* Research paper available at univ. Maastricht nl, https://project.dke.maastrichtuniversity.nl/games/files/bsc/Roelofs_Bsc-paper.pdf, 2012. – Aufgerufen: 25.07.2017

Ronenz 2014

RONENZ: *Pure Monte Carlo game search for 2048.* http://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048/23853848#23853848, 2014. – Aufgerufen: 16.07.2017

Sentenwolf 2013

SENTENWOLF: *Concept of directional optimization in CMA-ES algorithm.* https://commons.wikimedia.org/wiki/File:Concept_of_directional_optimization_in_CMA-ES_algorithm.png, 2013. – Aufgerufen: 25.07.2017

Silver et al. 2016

SILVER, David; HUANG, Aja; MADDISON, Chris J.; GUEZ, Arthur; SIFRE, Laurent; VAN DEN DRIESSCHE, George; SCHRITTWIESER, Julian; ANTONOGLOU, Ioannis; PANNEERSHELVAM, Veda; LANCTOT, Marc et al.: *Mastering the game of Go with deep neural networks and tree search.* in Nature, https://web.archive.org/web/20170227114102/https://gogameguru.com/i/2016/03/deepmind-mastering-go.pdf/, 2016. – Aufgerufen: 16.07.2017

Sutton & Barto 1998

SUTTON, Richard S.; BARTO, Andrew G.: *Reinforcement learning: An introduction.* MIT press Cambridge, 1998

Szita et al. 2009

SZITA, István; CHASLOT, Guillaume; SPRONCK, Pieter: Monte-Carlo Tree Search in Settlers of Catan. In: *Proc. Adv. Comput. Games* (2009), pages 21–32

Thill et al. 2012

Thill, Markus; Koch, Patrick; Konen, Wolfgang: Reinforcement Learning with N-tuples on the Game Connect-4. In: *Parallel Problem Solving from Nature-PPSN XII* (2012), pages 184–194

Vryniotis 2014

Vryniotis, Vasilis: *Using Artificial Intelligence to solve the 2048 Game (JAVA code).* http://blog.datumbox.com/using-artificial-intelligence-to-solve-the-2048-game-java-code/, 2014. – Aufgerufen: 25.07.2017

Xiao 2014

Xiao, Robert: *Expectimax optimization for 2048.* http://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048/22498940#22498940, 2014. – Aufgerufen: 16.07.2017