

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

PHYSIKALISCH-ASTRONOMISCHE FAKULTÄT  
THEORETISCH-PHYSIKALISCHES INSTITUT

**Monte-Carlo-Tree-Search:  
Statistische Physik und Spieltheorie**

BACHELORARBEIT  
zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)

vorgelegt von Robert Kummer  
geboren am 29. April 1997 in Arnstadt  
Matrikelnummer: 159782



Betreut durch: Prof. Dr. Bernd Brügmann  
Zweitgutachterin: Sarah Renkhoff



## **Abstract**

Gegenstand dieser Arbeit ist es, Monte-Carlo-Methoden zu beschreiben und deren Anwendungsmöglichkeiten im Bereich der Spieltheorie zu zeigen. Dazu werden mehrere Monte-Carlo-Algorithmen für die Auswahl von Spielzügen implementiert und am Beispiel des Brettspiels Carcassonne getestet. Es wird gezeigt, dass der UCT-Ansatz zu den besten Ergebnissen führt, wenn er ausreichend Iterationen durchführen kann. Aufgrund der dafür nötigen Rechenzeit ist es jedoch nicht möglich, dass die KI unter normalen Spielbedingungen gegen einen fortgeschrittenen menschlichen Spieler gewinnt.



# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
<b>2 Monte-Carlo-Methoden</b>	<b>2</b>
2.1 Historisches . . . . .	2
2.2 Simple-Monte-Carlo . . . . .	2
<b>3 Monte-Carlo-Markov-Ketten</b>	<b>3</b>
3.1 Markov-Ketten . . . . .	3
3.2 Stationäre Verteilung . . . . .	4
3.3 Detailed-balance-Bedingung . . . . .	4
3.4 Das Verfahren . . . . .	5
<b>4 Metropolis-Hastings-Algorithmus</b>	<b>6</b>
<b>5 Statistische Physik</b>	<b>7</b>
<b>6 Metropolis-Algorithmus</b>	<b>7</b>
<b>7 Monte-Carlo-Methoden zur Optimierung</b>	<b>8</b>
<b>8 Monte-Carlo-Methoden in Machine-Learning</b>	<b>8</b>
<b>9 Monte-Carlo-Tree-Search</b>	<b>9</b>
9.1 Spielbäume . . . . .	9
9.2 Der UCT-Algorithmus . . . . .	10
9.3 Pseudocode . . . . .	13
9.4 Parallelisierung . . . . .	13
9.5 Subtree-Erhaltung . . . . .	14
<b>10 Anwendung am Brettspiel Carcassonne</b>	<b>14</b>
10.1 Spieltheoretische Eigenschaften . . . . .	14
10.1.1 Perfekte Informationen . . . . .	14
10.1.2 Determinismus . . . . .	15
10.2 Abschätzung der Spielkomplexität . . . . .	15
10.2.1 Zustandsraum-Komplexität . . . . .	15
10.2.2 Spielbaum-Komplexität . . . . .	15
<b>11 Implementierung</b>	<b>16</b>
11.1 Allgemeines . . . . .	16
11.2 Codeübersicht . . . . .	16
11.3 KI-Spieler . . . . .	17
11.3.1 Simple-Monte-Carlo . . . . .	17
11.3.2 Flat-UCB . . . . .	17
11.3.3 UCT . . . . .	17
11.4 Evaluation . . . . .	17
<b>12 Auswertung</b>	<b>18</b>
12.1 Flat-UCB . . . . .	18
12.2 UCT . . . . .	23

12.3 UCT gegen menschlichen Spieler . . . . .	29
12.4 Vergleiche mit Heyden . . . . .	30
<b>13 Fazit und Aussicht</b>	<b>32</b>

## **Abbildungsverzeichnis**

1	Konstruktion von Übergangswahrscheinlichkeiten bei Markov-Ketten . . . . .	5
2	Spielbaumbeispiel . . . . .	9
3	Grafische Darstellung des MCTS-Algorithmus . . . . .	11
4	Von Flat-UCB und vom Zufallsspieler gesetzte Meeples . . . . .	19
5	Von Flat-UCB und vom Zufallsspieler erspielte Punkte . . . . .	19
6	Von Flat-UCB und vom Zufallsspieler erspielte Punkte pro Meeple . . . . .	20
7	Von Flat-UCB und Simple-MC gesetzte Meeples . . . . .	21
8	Von Flat-UCB und Simple-MC erspielte Punkte . . . . .	22
9	Von Flat-UCB und Simple-MC erspielte Punkte pro Meeple . . . . .	22
10	Von UCT und Flat-UCB gesetzte Meeples . . . . .	23
11	Von UCT und Flat-UCB erspielte Punkte . . . . .	24
12	Von UCT und Flat-UCB erspielte Punkte pro Meeple . . . . .	25
13	Kartenauswahl für verkürztes Spiel . . . . .	26
14	verkürzte Spielverläufe, Flat-UCB beginnt gegen UCT . . . . .	27
15	verkürzte Spielverläufe, UCT beginnt gegen Flat-UCB . . . . .	28
16	weiterer verkürzter Spielverlauf, UCT beginnt gegen Flat-UCB . . . . .	29

## **Tabellenverzeichnis**

1	Ergebnisse aus 20 Spielen zwischen Flat-UCB und einem Zufallsspieler . . . . .	18
2	Ergebnisse aus 50 Spielen zwischen Flat-UCB und Simple-MC . . . . .	20
3	Ergebnisse aus 50 Spielen zwischen UCT und Flat-UCB . . . . .	23

# 1 Einleitung

Als Google Deepminds künstliche Intelligenz (KI) AlphaGo am 12. März 2016 den Go-Profi Lee Sedol sehr eindeutig zum dritten Mal in Folge schlug, war das eine Sensation. KIs, die in der Lage sind, professionelle Go-Spieler zu besiegen, wurden von Experten erst zehn Jahre später erwartet. Einige behaupteten sogar, dass dies nie möglich werden würde.<sup>1</sup>

AlphaGo nutzte eine modifizierte Version des Monte-Carlo-Tree-Search-Algorithmus (MCTS) [19], ein Verfahren, das in seiner Standardform bereits vorher mit Erfolg für Go-Anwendungen genutzt wurde [10]. MCTS basiert auf Monte-Carlo-Methoden (MC) und hat damit seinen Ursprung in der statistischen Physik.

Ziel dieser Arbeit ist es, Monte-Carlo-Methoden für das Arbeiten mit Spielbäumen zu implementieren und deren Anwendung am Beispiel des Spieles Carcassonne zu testen. Die Ergebnisse sollen mit denen der bisher einzigen ähnlichen Arbeit von C. Heyden [13] verglichen werden.

Die Arbeit beschreibt zu Beginn MC-Methoden und deren Anwendung in der statistischen Physik und dabei besonders bei der Erzeugung von Stichproben entsprechend der Boltzmann-Verteilung. Im zweiten Teil wird auf die spieltheoretischen Merkmale von Carcassonne eingegangen sowie eine Implementierung von verschiedenen Monte-Carlo-Algorithmen für die Auswahl von möglichen Spielzügen vorgenommen. Die KIs werden dann in Spielen gegeneinander getestet, miteinander verglichen und schließlich findet der Vergleich zu Heyden statt.

---

<sup>1</sup>Quelle: <https://www.sciencemag.org/news/2016/01/huge-leap-forward-computer-mimics-human-brain-beats-professional-game-go>  
(Artikel, zuletzt abgerufen am 8. Oktober 2019)

## 2 Monte-Carlo-Methoden

Das Gebiet der Monte-Carlo-Methoden umfasst alle Methoden, bei denen versucht wird ein analytisch nicht oder nur schwierig lösbares Problem mithilfe von vielen Wiederholungen von Zufallsexperimenten numerisch zu lösen. Die Grundlage dafür stellt das Gesetz der großen Zahlen dar, wonach sich die relative Häufigkeit des Ergebnisses eines Zufallsexperiments unter gleichartigen Wiederholungen des Experiments um die theoretische Wahrscheinlichkeit des Ergebnisses stabilisiert.

### 2.1 Historisches

Ihren Ursprung haben Monte-Carlo-Methoden zum größten Teil in Los Alamos kurz nach Ende des zweiten Weltkrieges [17]. Stanislaw Ulam hatte die Idee, das Verhalten eines Systems zu simulieren, um das Verhalten approximiert zu beschreiben, statt nach analytischen Lösungen zu suchen. Die Idee ging aus einem Krankheitsfall hervor, während welchem Ulam das Kartenspiel Canfield Solitaire spielte und herausfinden wollte, mit welcher Wahrscheinlichkeit er ein Spiel gewinnt. Da es nicht möglich war, alle möglichen Spielausgänge durchzurechnen, hatte er die Idee, zufällige Spiele zu simulieren und entsprechend der Regeln durchzuspielen, um das Spielergebnis zu berechnen. Würde man das oft genug wiederholen, würde dies eine empirische Näherung an die Gewinnchance für ein Startblatt ermöglichen [17].

Eine erste Anwendung fand diese Klasse von Methoden, welche nach dem Stadtteil „Monte Carlo“ von Monaco benannt wurden, in der Simulation von Neutronenbewegungen und -multiplikationsraten bei Kernspaltungsprozessen in Atombomben.

### 2.2 Simple-Monte-Carlo

Die Grundidee von Simple-Monte-Carlo ist es, eine reelle Zahl  $r$  als den Erwartungswert  $\mathbb{E}$  einer Funktion  $f$  einer Zufallsvariablen  $X$  darzustellen:

$$r = \mathbb{E}(f(X)). \quad (1)$$

Sei nun  $X_1, \dots, X_n$  eine Stichprobe von Zufallsvariablen der Verteilung von  $X$ . Mithilfe von Realisierungen  $x_i$  der Zufallsvariablen  $X_i$  und dadurch mit den Funktionswerten  $f(x_i)$  lässt sich der Erwartungswert mit dem folgenden erwartungstreuen Schätzer schätzen:

$$r \approx \bar{f} = \frac{1}{n} \sum_{i=1}^n f(x_i), \quad (2)$$

wobei  $\bar{f}$  aufgrund des Gesetzes der Großen Zahlen für  $n \rightarrow \infty$  gegen  $r$  konvergiert. Dieser Schätzer hat den Standardfehler

$$\sigma_{\bar{f}} = \frac{\sigma}{\sqrt{n}}, \quad (3)$$

wobei  $\sigma$  die Standardabweichung der wahren Verteilung der Funktion  $f$  ist [11]. Man sagt auch, Simple-Monte-Carlo hat die Konvergenzordnung  $\mathcal{O}(1/\sqrt{n})$ . Der Faktor  $\sqrt{n}$  im Nenner sagt aus, dass beispielsweise viermal so viele Samples von der Funktion  $f$  benötigt werden, um den Fehler zu halbieren, 100 mal so viele Samples, um den Fehler um den Faktor 10 zu verkleinern.

Um den Fehler zu verringern, kann statt einer Erhöhung der Samplezahl auch ein Modell gewählt werden, welches zu einem kleineren Wert von  $\sigma$  führt. Anstrengungen in diesem Bereich fasst man unter dem Begriff der Varianzreduktion zusammen.

Im Vergleich zu anderen numerischen Methoden, beispielsweise Methoden zur Integralberechnung, ist die Ordnung  $\mathcal{O}(1/\sqrt{n})$  für Problemstellungen in wenigen Dimensionen äußerst langsam. Der Vorteil von Monte-Carlo-Methoden ist, dass der Schätzwert mit zunehmender Samplezahl unabhängig von der Dimension  $d$  konvergiert. Im Vergleich zur Simpsonregel, welche eine Funktion mit der Ordnung  $\mathcal{O}(1/n^{4/d})$  integriert, konvergiert die Monte-Carlo-Approximation ab einem  $d > 8$  schneller. Das Simpsonverfahren leidet unter dem sogenannten „curse of dimensionality“.

Wenn entsprechend der korrekten Verteilung von  $X$  gesampelt werden kann, dann entsteht nicht das Problem, dass sich viele Samples in Raumbereichen befinden, welche kein hohes statistisches Gewicht haben. Daher werden also Algorithmen benötigt, welche es ermöglichen, entsprechend beliebiger Zufallsverteilungen zu samplen.

Es war um 1950 bereits möglich, Pseudozufallszahlen zu erzeugen, beispielsweise mit der middle-square-Methode [17]. Der Computer konnte Zufallszahlsequenzen zwischen 0 und 1 erzeugen, welche gleichverteilt waren. Wenn die (stetige) Gleichverteilung simuliert werden kann, gelingt mittels Inversionsmethode oder Rejection-Methoden auch die Simulation anderer Verteilungen [8]. Diese Methoden waren aber nicht universell einsetzbar und ebenfalls ungeeignet für Wahrscheinlichkeitsverteilungen in vielen Dimensionen [17]. Die Erzeugung von Stichproben mittels Markov-Ketten löst diese Probleme.

### 3 Monte-Carlo-Markov-Ketten

Monte-Carlo-Markov-Chain-Methoden (MCMC) sind Methoden, welche mittels einer Markov-Kette (MK) Stichproben aus bestimmten Verteilungen ziehen, mit welchen dann Monte-Carlo-Approximationen durchgeführt werden können. Dies funktioniert, indem eine Markov-Kette gebildet wird, welche die gewünschte Verteilung als ihre stationäre Verteilung hat. Im Folgenden soll auf die dafür zu Grunde liegende Mathematik eingegangen werden.

#### 3.1 Markov-Ketten

Gegeben sei eine Familie  $Y = (X_t)_{t \in \mathbb{N}}$  von Zufallsvariablen  $X$ , vom Wahrscheinlichkeitsraum  $(\Omega, \Sigma, P_A)$  in einen Messraum  $(\Omega', \Sigma')$ . Die Zufallsvariablen  $X$  haben die Verteilung  $P_X$  und können nur Werte aus dem Zustandsraum  $\Sigma' = \{s_1, s_2, \dots\}$  annehmen.  $Y$  heißt (diskrete) Markov-Kette, wenn für die Übergangswahrscheinlichkeiten

$$Pr(X_{t+1} = s_{j_{t+1}} | X_t = s_{i_t}, X_{t-1} = s_{j_{t-1}}, \dots, X_0 = s_{j_0}) = Pr(X_{t+1} = s_{j_{t+1}} | X_t = s_{j_t}) \quad (4)$$

gilt.  $X_t$  ist hierbei eine Zufallsvariable zum Zeitindex  $t$  und  $s_{j_t} \in \Sigma'$  der j-te Zustand zum Zeitindex  $t$ . Diese Bedingung heißt Markov-Bedingung. Eine Markov-Kette ist also ein stochastischer Prozess, der diese Markov-Bedingung erfüllt. Die Übergangswahrscheinlichkeiten hängen nur vom aktuellen Zustand der Kette und nicht von der Vorgeschichte der Kette ab. (Unter einer Markov-Kette n-ter Ordnung versteht man eine MK, bei der die Übergangswahrscheinlichkeiten von den n vorherigen Zuständen abhängen. In dieser Arbeit werden allerdings nur Markov-Ketten erster Ordnung betrachtet).

Markov-Ketten werden homogen genannt, wenn ihre Übergangswahrscheinlichkeiten mit der Zeit konstant bleiben.

Eine Markov-Kette heißt irreduzibel, wenn für alle Zustände  $i, j \in \Sigma'$  gilt, dass die Wahrscheinlichkeit in endlicher Zeit von  $i$  zu  $j$  zu gelangen größer als Null ist.

Trotz aller Zufälligkeit kann eine Markov-Kette Periodizität aufweisen. Als Periode  $d(i)$  eines Zustandes  $i \in \Sigma'$  bezeichnet man den größten gemeinsamen Teiler der Menge der Rückkehrzeiten  $t$  zu  $i$ , wenn  $i$  Startzustand ist. Haben alle Zustände der MK die Periode  $d$ , dann hat die Markov-Kette die Periode  $d$ . Hat die Kette die Periode eins, so heißt diese aperiodisch. Wird ein Zustand fast sicher unendlich oft besucht, heißt dieser rekurrent, sonst transient. Sind alle Zustände der Kette rekurrent, heißt die Kette rekurrent, gleiches gilt für Transienz.

### 3.2 Stationäre Verteilung

Ist eine Markov-Kette irreduzibel und aperiodisch, konvergiert die Verteilung der möglichen Zustände der Kette gegen eine stationäre Verteilung  $\pi$ . Sind die Zustände der Kette entsprechend der stationären Verteilung verteilt, ändert sich diese Verteilung für die nachfolgenden Schritte nicht mehr. Für die stationäre Verteilung gilt, dass die Wahrscheinlichkeit für den Zustand  $j$  gleich ist, wie die aufsummierten Wahrscheinlichkeiten dafür, sich in einem anderen Zustand  $i$  zu befinden und dann mit der Übergangswahrscheinlichkeit  $p_{ij}$  zu  $j$  zu wechseln:

$$\pi(\{j\}) = \pi_j = \sum_{i \in Z} \pi(\{i\}) p_{ij}. \quad (5)$$

Wie erwähnt, soll nun eine Markov-Kette erzeugt werden, welche eine gewünschte Wahrscheinlichkeitsverteilung als ihre stationäre Verteilung hat. Eine nicht notwendige, stärkere (siehe Gl. (7)), aber oft leichter zu implementierende Bedingung dafür, dass eine Verteilung stationär ist, ist die detailed-balance-Bedingung.

### 3.3 Detailed-balance-Bedingung

Eine Markov-Kette mit Übergangswahrscheinlichkeiten  $p_{ij}$  heißt reversibel bezüglich einer Verteilung  $\pi$ , wenn diese die folgende Bedingung erfüllt:

$$\pi_i p_{ij} = \pi_j p_{ji}. \quad (6)$$

Für eine Verteilung bezüglich welcher die Kette reversibel ist gilt:

$$\sum_i \pi_i p_{ij} = \sum_i \pi_j p_{ji} = \pi_j \sum_i p_{ji} = \pi_j. \quad (7)$$

Das ist genau die Bedingung (5) für die stationäre Verteilung.

Die detailed-balance-Bedingung sagt also aus, dass die Wahrscheinlichkeit, in Zustand  $i$  zu sein und dann zu  $j$  zu wechseln, gleich groß ist wie die Wahrscheinlichkeit, erst in  $j$  zu sein und dann zu  $i$  überzugehen.

### 3.4 Das Verfahren

Um zu verstehen, wie nun eine Markov-Kette mit passenden Übergangswahrscheinlichkeiten erzeugt wird, soll das folgende Beispiel herangezogen werden.

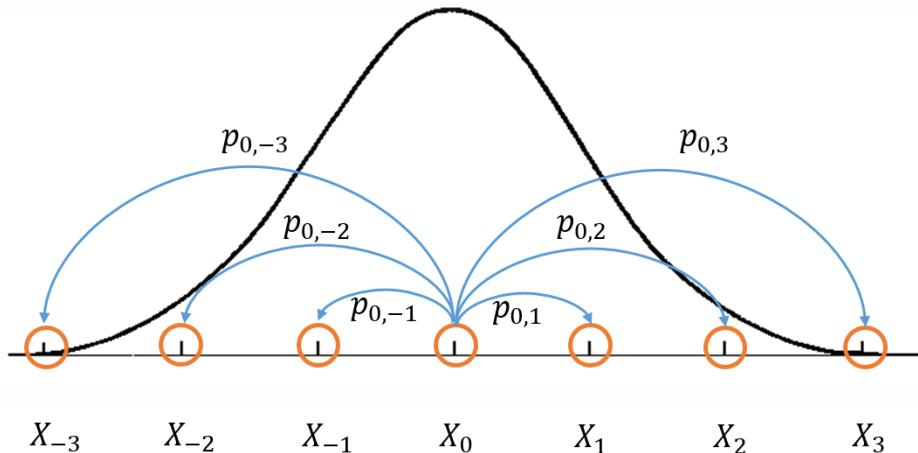


Abb. 1: Darstellung zur Verdeutlichung der Konstruktion von Übergangswahrscheinlichkeiten<sup>2</sup>.

Die Abb. 1 zeigt als schwarze Linie die Wahrscheinlichkeitsdichte  $p(x)$  der Verteilung  $P$ , welche approximiert werden soll. In diesem Beispiel sind eigentlich alle Punkte der x-Achse mögliche Zustände der Markov-Kette. Es liegt also eine unendlich große Zustandsmenge vor. Zur Veranschaulichung wurden jedoch bestimmte Zustände und deren Übergänge hervorgehoben. Man möchte die Übergangswahrscheinlichkeiten zwischen den  $x$ -Zuständen nun so designen, dass sich die Markov-Kette größtenteils in Bereichen aufhält, wo  $p(x)$  groß ist und wenig Zeit in Regionen verbracht wird, wo  $p(x)$  klein ist, also generell proportional zu  $p(x)$ . Wenn die Schritte der Markov-Kette lang genug simuliert werden, soll sich die Verteilung von Zuständen der Wahrscheinlichkeitsverteilung immer weiter annähern.

Die Schritte des Grundalgorithmus für das Samplen mittels MCMC sind:

1. Starte bei einem beliebigen Startzustand  $x$ .
2. Springe zu Punkt  $x'$  mit einer bestimmten Wahrscheinlichkeit, wenn nicht, bleibe im Zustand  $x$ .
3. Gehe zu Schritt 2, bis es  $T$  Übergänge gab.
4. Notiere den aktuellen Zustand  $x'$  und gehe zurück zu Schritt 2.

Die letzten beiden Schritte sind nötig, wenn man unabhängige Stichproben samplen möchte, da die Samples sonst zwar annähernd korrekt verteilt sind, jedoch direkt voneinander abhängig sind. Es wird dadurch eine gewisse Anzahl von Samples nicht berücksichtigt, um Unabhängigkeit zu simulieren. Der Wert  $T$  muss dabei abhängig von den Details der Targetdistribution und der Konstruktion der Übergangswahrscheinlichkeiten gewählt werden und hängt von der Autokorrelation benachbarter Samples ab [12].

<sup>2</sup>Bildquelle: <http://bjlkeng.github.io/images/mcmc.png>

(zuletzt abgerufen am 7. Oktober 2019)

Um nun die Übergangswahrscheinlichkeiten richtig zu konstruieren, gibt es verschiedene Möglichkeiten. Eine davon wird durch den Metropolis-Hastings-Algorithmus beschrieben.

## 4 Metropolis-Hastings-Algorithmus

Es soll eine Markov-Kette so über einem Zustandsraum konstruiert werden, dass sie die gewünschte Verteilung  $P$  als ihre stationäre Verteilung hat. Im folgenden ist der Algorithmus, basierend auf einem diskreten Zustandsraum, dargestellt. Für eine allgemeinere Formulierung auf stetigen Zustandsräumen empfiehlt sich die Ausarbeitung von Chib and Greenberg [6].

Die MK soll die stationäre Verteilung  $\pi$  haben, welche einzigartig sein soll. Die Forderung nach Einzigartigkeit verlangt nach Ergodizität (Aperiodizität und positive Rekurrenz aller Zustände) der Kette.

Da  $\pi$  eine stationäre Verteilung ist, erfüllt sie die detailed-balance-Bedingung (Gl. (6)). Mit den Übergangswahrscheinlichkeiten  $Pr(x'|x)$  vom Zustand  $x$  in den Zustand  $x'$  kann man diese auch schreiben als

$$Pr(x'|x)\pi(x) = Pr(x|x')\pi(x'). \quad (8)$$

Die Idee ist, die Übergangswahrscheinlichkeit als Produkt zweier Wahrscheinlichkeiten zu schreiben:

$$Pr(x'|x) = g(x'|x)A(x',x), \quad (9)$$

wobei  $g(x'|x)$  die Dichte einer bedingten Wahrscheinlichkeitsverteilung ist, welche angibt, mit welcher Wahrscheinlichkeit der Zustand  $x'$  vorgeschlagen wird, unter der Bedingung, dass man sich im Zustand  $x$  befindet. Die Akzeptanzrate  $A(x',x)$  gibt die Wahrscheinlichkeit an, den Übergang von  $x$  zu  $x'$  zuzulassen (zu akzeptieren).

Stellt man diese Definition um und fügt sie in (8) ein, erhält man:

$$\frac{A(x',x)}{A(x,x')} = \frac{\pi(x')g(x|x')}{\pi(x)g(x'|x)}. \quad (10)$$

Die Dichte  $g(x'|x)$  kann je nach Aufgabenstellung gewählt werden, beispielsweise als beim Zustand  $x$  zentrierte Normalverteilung. Davon abhängig muss nun noch  $A(x,x')$  so gewählt werden, dass (10) erfüllt wird.

Die Metropolis-Hastings-Wahl lautet:

$$A(x',x) = \min(1, \frac{\pi(x')g(x|x')}{\pi(x)g(x'|x)}). \quad (11)$$

Ist  $\frac{\pi(x')g(x|x')}{\pi(x)g(x'|x)}$  kleiner, oder gleich 1, ist das Reziproke  $\frac{\pi(x)g(x'|x)}{\pi(x)g(x|x')}$  größer als 1 und daher  $A(x,x')$  gleich 1 und die Bedingung ist erfüllt. Ist der erste Bruch größer als 1, dann erfüllt sich die Bedingung ebenfalls.

Der Algorithmus sieht nun vor, dass bei jedem Schritt, der Verteilung  $g(x'|x)$  entsprechend, ein neuer Kandidat  $x'$  erzeugt wird, den man mit Wahrscheinlichkeit  $A(x',x)$  annimmt. Bei einer Annahme wird das nächste  $x_{t+1} = x'$  gesetzt. Wenn der Kandidat abgelehnt wird, bleibt die Kette im alten Zustand ( $x_{t+1} = x_t$ ). Durch diese Vorgehensweise konvergiert die empirische Verteilung der Zustände  $x_0, \dots, x_T$  nun gegen die gewünschte Verteilung  $P$ . Die Samples aus dem Markov Ketten Verfahren kann man nun für Monte-Carlo-Approximationen weiterverwenden.

## 5 Statistische Physik

Die Boltzmann-Statistik beschreibt Wahrscheinlichkeiten für Zustände eines Systems, welches sich im Wärmeaustausch mit einem Wärmereservoir befindet. Über das mikrokanonische Ensemble erhält man mit Hilfe der Annahme, dass im abgeschlossenen Gesamtsystem alle Zustände a priori gleich wahrscheinlich sind, folgende Wahrscheinlichkeit dafür, das System in einem Zustand mit Energie  $E_i$  zu messen:

$$p_i = g_i \frac{e^{-\beta E_i}}{\sum_{j \in \Omega} e^{-\beta E_j}}. \quad (12)$$

Dabei ist  $\Omega$  der Zustandsraum und der Term im Nenner stellt die kanonische Zustandssumme  $Z$  dar. Der Entartungsgrad  $g_i$  gibt die Anzahl der Zustände an, welche dieselbe Energie  $E_i$  haben. Da Zustände gleicher Energie gleich wahrscheinlich sind, teilt man diesen Ausdruck durch  $g_i$ , um die Wahrscheinlichkeit dafür zu erhalten, das System genau im Zustand  $i$  vorzufinden:

$$\tilde{p}_i = \frac{e^{-\beta E_i}}{Z}. \quad (13)$$

Mithilfe dieser Ausdrücke lassen sich verschiedene physikalische Erwartungswerte berechnen, beispielsweise der der Energie:

$$\langle E \rangle = \sum_{i \in \Omega} E_i \tilde{p}_i = \frac{\sum_{i \in \Omega} E_i e^{-\beta E_i}}{\sum_{j \in \Omega} e^{-\beta E_j}}. \quad (14)$$

In realen Anwendungen ist es (aufgrund der Größe des Zustandsraumes  $\Omega$ ) nicht möglich die Zustandssumme auszurechnen. Um eine Annäherung an den tatsächlichen Wert zu erhalten, könnte beispielsweise gleichverteilt aus dem Zustandsraum gesampelt werden:

$$\langle E \rangle \approx \frac{\sum_{i=1}^M E_i e^{-\beta E_i}}{\sum_{i=1}^M e^{-\beta E_i}}. \quad (15)$$

Wählt man diese Methode, so werden sehr häufig Samples auftreten, für welche der Boltzmann-Faktor  $e^{-\beta E_i}$  sehr klein ist, da die Verteilung in den vielen Dimensionen oft stark lokalisiert ist. Diese Samples haben kein hohes statistisches Gewicht und man hat daher wenig relevante Zustände für die Mittelwertbildung [17].

Es wäre besser, wenn man direkt entsprechend der richtigen Verteilung der Zustände samplen würde. So könnte man den Erwartungswert mit dem klassischen erwartungstreuen Schätzer (Gl. (2))

$$\langle E \rangle \approx \frac{1}{M} \sum_{i=1}^M E_i \quad (16)$$

schätzen, indem man die Energiewerte  $E_i$  zu den Samples  $\omega_i$  berechnet. Dieser Schätzer hat entsprechend die Konvergenzordnung  $\mathcal{O}(1/\sqrt{M})$ .

Der Algorithmus, welcher zur Erzeugung einer Markov-Kette genutzt wird, welche die Boltzmann-Verteilung als ihre stationäre Verteilung hat, wird Metropolis-Algorithmus genannt.

## 6 Metropolis-Algorithmus

Der Metropolis-Algorithmus ist ein spezieller Metropolis-Hastings-Algorithmus. Er wurde 1953 von Nicholas Metropolis et al. publiziert [16] und erzeugt Zustände eines Systems entsprechend der Boltzmann-Verteilung.

Im Folgenden wird der Algorithmus für den Fall beschrieben, dass das System von einem mehrdimensionalen Ort  $\vec{x}$  abhängt. Der Ort  $\vec{x}$  sei kontinuierlich und der aktuelle Ort nach  $i$  Iterationen wird mit  $\vec{x}_i$  bezeichnet. Der Metropolis-Algorithmus ergibt sich dann durch Wiederholung der folgenden Schritte:

1. Ein neuer Ort  $\vec{y} = \vec{x}_i + r \cdot \vec{q}$  wird ausgewählt, wobei  $\vec{q}$  ein Zufallsvektor aus Komponenten zwischen -1 und +1 und  $r$  ein fest gewählter Suchradius ist.
2. Die Energiedifferenz  $\Delta E := E(\vec{y}) - E(\vec{x}_i)$  wird berechnet. Der Zustand  $\vec{y}$  wird nun mit der Wahrscheinlichkeit  $p_A = \min(1, \exp(-\frac{\Delta E}{kT}))$  angenommen. Das heißt, dass die neue Position in jedem Fall akzeptiert wird, wenn sie energetisch günstiger ist, also  $\Delta E \leq 0$  gilt.

Auch hier sind die Samples korreliert und es kann die Verwerfungsmethode aus 3.4 genutzt werden. Der Metropolis-Algorithmus erzeugt Systeme im kanonischen Zustand.

## 7 Monte-Carlo-Methoden zur Optimierung

Der Metropolis-Algorithmus kann auch als stochastisches Optimierungsverfahren zum Finden eines globalen Minimums einer Wertelandschaft verwendet werden. Hierzu wird die Temperatur  $T$  variiert. Es wird mit einer hohen Temperatur begonnen, so dass mit frühen Schritten ein möglichst großes Gebiet der Wertelandschaft besucht wird. Anschließend wird die Temperatur langsam abgesenkt, wodurch sich mit immer höherer Wahrscheinlichkeit einem Minimum genähert wird, da der Wechsel in energetisch ungünstigere Zustände immer unwahrscheinlicher wird. Dieser Algorithmus, mit von der Zeit abhängiger Temperatur, heißt simulierte Abkühlung (simulated annealing) und hat ihren Ursprung in der Physik der Abkühlung von Körpern. Beim Übergang vom flüssigen in den kristallinen Zustand nähert sich die Atomverteilung bei ausreichender Zeit sehr stark der energetisch günstigen gleichmäßigen Einkristallstruktur an.

Das Verfahren ähnelt dem Bergsteigeralgorithmus („hill climbing“), akzeptiert jedoch, im Gegensatz zu diesem, auch Schritte weg vom nächsten Minimum, sodass das „Hängen bleiben“ in lokalen Minima vermieden wird. Mit großem Erfolg wird dieser Algorithmus zur Lösung komplexer Optimierungsprobleme, wie z.B. des „traveling sales-man“ Problems eingesetzt [15].

## 8 Monte-Carlo-Methoden in Machine-Learning

MCMC-Verfahren können im Machine-Learning (ML) Kontext beispielsweise dazu genutzt werden, um aus einer Auswahl möglicher Züge zu samplen. Große Bekanntheit in diesem Gebiet hat allerdings ein anderer MC-Algorithmus erlangt, der sogenannt Monte-Carlo-Tree-Search-Algorithmus (MCTS) [19] und speziell der UCT-Algorithmus („Upper Confidence Bounds Algorithm for Trees“). Dieser Algorithmus ist ein Suchalgorithmus für Entscheidungsprozesse auf Spielbäumen und stellt eine Alternative zur Durchsuchung der Bäume mittels Minimax oder Pruning-Methoden dar.

MCTS selbst wird nicht als ML-Algorithmus angesehen, besitzt aber einige Ähnlichkeiten mit dem ML-Gebiet des Reinforcement-Learning [21].

Für die Auswahl von möglichen Aktionen bei Spielbäumen bieten sich MC-Methoden auf viele Weisen an. B. Brügmann führte 1993 die erste Anwendung von Monte-Carlo-Methoden auf Spielbäume durch, genauer für das Brettspiel Go. Dabei wurde, mittels simulated annealing,

möglichen Zügen eine gewisse Wahrscheinlichkeit zugeordnet, entsprechend derer Spiele zufällig zu Ende gespielt wurden, um den besten nächsten Zug auszuwählen [4]. R.Coulom beschrieb die weitere Anwendung von MC-Methoden auf Spielbäume [7] und L. Kocsis and Cs. Szepesvári entwickelten den UCT-Algorithmus, welcher eine MCTS-Version ist, der Bandit-Ideen zur Beeinflussung der Entscheidungsfindung nutzt [14, 3]. Dieser Algorithmus wurde mit großem Erfolg in Go-Programme implementiert und weiterentwickelt [10]. Zuletzt wurde er in abgewandelter Form durch den Erfolg von Google Deepminds Go-KI AlphaGo, sowie weiterer Versionen wie AlphaGoZero [19] [20], bekannt.

## 9 Monte-Carlo-Tree-Search

### 9.1 Spielbäume

In diesem Abschnitt soll kurz das Konzept von Spielbäumen erklärt werden. Ein Beispielbaum ist in Abb. 2 gezeigt.

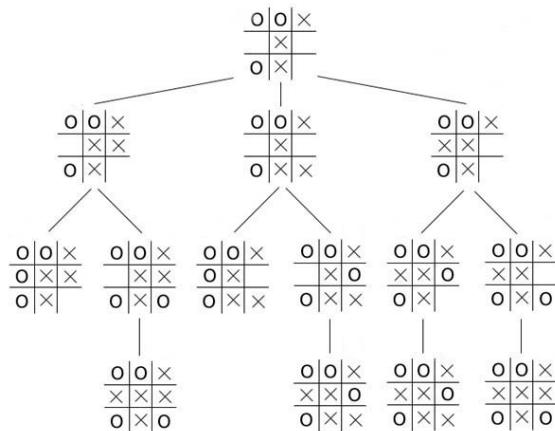


Abb. 2: Darstellung eines Spielbaumes für das Spiel TicTacToe<sup>3</sup>.

Ein Spielbaum ist ein Graph, dessen Knoten (Nodes) Spielzustände repräsentieren und dessen Kanten für mögliche Aktionen im Spiel stehen. Die Start-Node, von der aus der Baum beginnt, wird Root-Node genannt. Die Nodes auf der nächsten Ebene sind alle durch mögliche Aktionen ab der Root-Node erreichbar und werden daher als Kinder (Child-Nodes) der Root-Node bezeichnet. Nodes, die bisher keine Kinder haben, nennt man Leaf-Nodes und solche Nodes, die einen Endzustand repräsentieren (in Abb. 2 die unterste Node-Ebene), werden End-Nodes genannt. Diese End-Nodes können folglich auch keine weiteren Kinder haben.

Um von einer Root-Node ausgehend die bestmögliche nächste Aktion zu ermitteln, gibt es viele Algorithmen, welche Suchbäume erkunden. Ein berühmtes Beispiel ist der Minimax-Algorithmus, welcher versucht die Aktion zu finden, die dem Ausgangsspieler das bestmögliche Endergebnis garantiert, unter der Voraussetzung, dass der Gegenspieler perfekt spielt. Da es für diesen Algorithmus nötig ist, alle Enden des Spielbaumes zu untersuchen, um die jeweiligen Ergebnisse nach oben hin durchzureichen, ist er ungeeignet für sehr große und komplexe

<sup>3</sup>Bildquelle: <https://slideplayer.com/slide/13805878/85/images/6/A+part+of+the+game+tree+of+Tic+Tac+Toe.jpg> (bearbeitet, zuletzt abgerufen am 7. Oktober 2019)

Suchbäume. Es werden also Algorithmen benötigt, die es schaffen die Bäume effektiver und schneller zu untersuchen. Eine solche Algorithmenklasse sind MCTS-Verfahren.

## 9.2 Der UCT-Algorithmus

MCTS-Verfahren nutzen zufällig zu Ende simulierte Spiele, um die spieltheoretischen Werte der Nodes des Spielbaumes zu approximieren. Mit Hilfe dieser Werte wird ein partieller Aufbau des Spielbaumes gelenkt und eine Auswahlstrategie optimalerweise so verbessert, dass sie die beste nächste Aktion ermitteln kann [3].

Als UCT-Algorithmus bezeichnet man eine spezielle MCTS-Version, welche eine „Upper-Confidence-Bounds-Strategie“ (UCB) als Auswahlstrategie nutzt. Standardmäßig wird UCB1 genutzt [2, 3] (siehe Gl. 17).

UCT besteht aus vier Schritten, die solange wiederholt werden, bis idealweise der beste nächste Zug von einem gegebenen Spielstand aus ermittelt wurde.

**Selektion:** Jede Node speichert für den Algorithmus jeweils einen Visit-Count und einen Node-Wert. Der Visit-Count ist die Anzahl der Zufallssimulationen, die stattgefunden haben, nachdem die Node ausgewählt wurde. Der Node-Wert soll den spieltheoretischen Wert der Node angeben und dafür sorgen, dass vielversprechende Züge den Vorzug erhalten.

Jede UCT-Iteration beginnt damit, eine Child-Node nach der Root-Node auszuwählen. Es wird die Kind-Node gewählt, welche den folgenden UCB1-Wert [3] maximiert

$$\frac{v}{n} + c \sqrt{\frac{\ln N}{n}}. \quad (17)$$

Dabei steht  $v$  für den Wert der Node und  $n$  für den Visit-Count.  $N$  steht für den Visit-Count der Parent-Node, von der aus der Zug gespielt werden soll. Der Wert  $c$  stellt einen Gewichtungsparameter dar. Der erste Term steht für den Nutzen (exploitation) einer Aktion. Er ist größer für Züge, die auf lange Sicht häufig zu Siegen führen und daher einen hohen Wert aufweisen. Der hintere Term steht für das Erkunden (exploration). Der hintere Term ist groß für Nodes, die im Vergleich zu ihren Nachbarnodes noch nicht oft besucht wurden. Er sorgt dafür, dass auch solche Züge, die bisher als nicht sehr profitabel gelten, ab und zu untersucht werden, um zu verhindern, dass gute Züge unentdeckt bleiben. Der Parameter  $c$  wird üblicherweise auf  $\sqrt{2}$  festgelegt und ist für die Gewichtung des zweiten Terms verantwortlich. Durch die beiden Teilterme gibt es einen Kompromiss zwischen Exploration und Exploitation.

Der Auswahlprozess wird dann für die folgenden Knoten so oft wiederholt, bis eine End-Node erreicht ist, von deren Spielzustand bisher nur eine Simulation ausgeführt wurde.

Da das Ziel darin besteht, sich dem Minimax-Wert der Nodes anzunähern, also die besten Aktionen sowohl für den Startspieler als auch den Gegner auszuwählen, muss bei der Auswahl der Aktionen des Gegners darauf geachtet werden, dass dieser Züge spielt, die für ihn von Vorteil sind.

**Expansion:** Wenn eine End-Node erreicht ist, wird der Baum unter dieser Node für jeden möglichen nächsten Spielzustand um eine neue Kind-Node erweitert. Einer davon wird zufällig für den nächsten Schritt ausgewählt.

**Simulation:** Dieser Teil repräsentiert die erwähnte Bewertungsfunktion. Für den ausgewählten Knoten wird ein Spiel simuliert, indem zufällige Züge, beginnend mit dem Spielzustand der neuen Child-Node, ausgewählt werden. Das Ergebnis der Zufallssimulation wird für den folgenden Schritt benötigt.

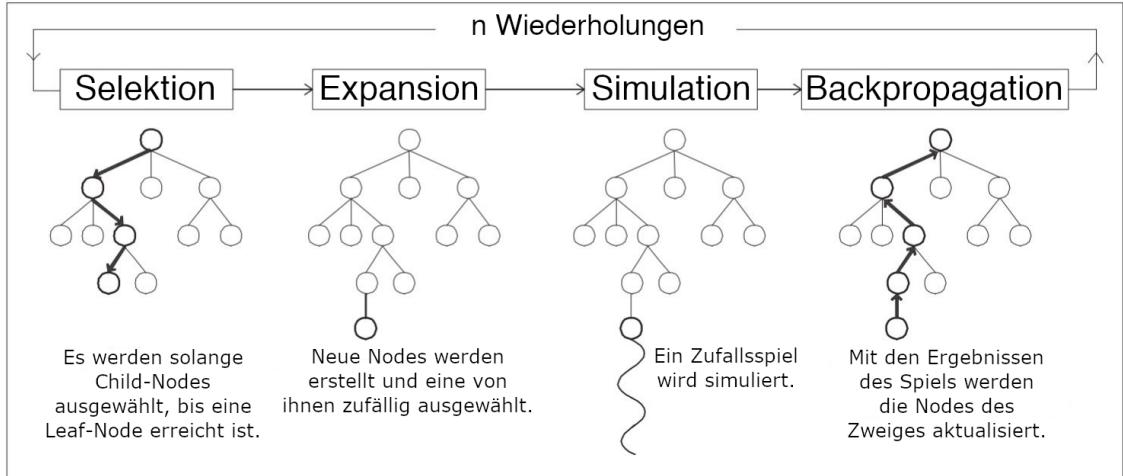


Abb. 3: Grafische Darstellung des Ablaufes des MCTS-Algorithmus<sup>4</sup>.

**Backpropagation:** Die Werte  $v$  und Visit-Counts  $n$  aller, im bisherigen Prozess durchlaufenen, Knoten werden nun wie folgt upgedated:

$$n = n + 1 \quad (18)$$

$$v = v + r, \quad (19)$$

wobei der Wert von  $r$  abhängig von der Bewertungsmethode ist. Es gibt zwei verschiedene Möglichkeiten, die Aktualisierung eines Knotenzustands zu implementieren. Erstens kann der Zustand in einem absoluten Sinne aus der Perspektive des Spielers, welcher ausgehend von der Root-Node als erster dran ist, evaluiert werden. In diesem Fall ist  $r = 1$  für die Nodes, welche sich auf Ebenen befinden, die nach Zügen des Root-Spielers erreicht werden. Für die anderen Nodes wird  $r = -1$  gesetzt. Im Falle eines Unentschiedens gilt für alle Nodes  $r = 0$ . Wählt man diese Variante, so muss in der Selektionsphase das Vorzeichen der Node-Werte von Nodes nach Gegneraktionen umgekehrt werden, da sonst das beschriebene Problem der Auswahl von falschen Aktionen auftreten würde.

Andererseits kann eine Bewertung durchgeführt werden, die immer vom Spieler abhängt, der die nächste Aktion ausführen muss. Für diese Variante gilt  $r = 1$ , wenn der Spieler, der vor der zu bewertenden Node dran ist, die aktuelle Simulation gewonnen hat,  $-1$ , wenn er verloren hat und  $0$  im Fall eines Unentschiedens.

Nach der Bewertung wird wieder zur Selektions-Phase beginnend von der Root-Node zurückgekehrt und die nächste Iteration begonnen. Der gesamte Prozess ist in Abb. 3 grafisch dargestellt.

<sup>4</sup>Quelle: [https://miro.medium.com/max/1491/1\\*Ntm0xHhJ5j0gsL9AdB2kNw.jpeg](https://miro.medium.com/max/1491/1*Ntm0xHhJ5j0gsL9AdB2kNw.jpeg)  
(bearbeitet, zuletzt abgerufen am 7. Oktober 2019)

Über viele UCT-Iterationen hinweg, soll durch diesen Algorithmus die Child-Node, zu welcher die vielversprechendste Aktion führt, am häufigsten besucht werden.

**Alternative Bewertungsmethoden:** Es kann sich anbieten, statt der beschriebenen Bewertung mit 1 für Sieg, -1 für Niederlage und 0 für Unentschieden auch andere Werte zur Beurteilung von Nodes zu wählen. Beispielsweise kann die Punktedifferenz der Spieler als Wert dienen. Für die spielerabhängige Bewertung im Fall von zwei Spielern würde jeder Node der folgende Wert auf den bisherigen aufaddiert werden:

$$v = v_{\text{Spieler}} - v_{\text{Gegner}}. \quad (20)$$

Dabei steht  $v_{\text{Spieler}}$  für die erspielte Punktzahl des Spielers, welcher vor der Node am Zug ist und  $v_{\text{Gegner}}$  für die Punkte des Spielers, welcher nach dieser Node seine Aktion auswählen darf.

**Finale Selektion** Nach einer Stopbedingung, welche beispielsweise durch das Erreichen eines Zeitlimits oder einer Maximalschrittanzahl gegeben sein kann, muss nun noch der Zug gewählt werden, der tatsächlich gespielt werden soll. Dabei wird standardmäßig nicht die Node mit dem höchsten Wert  $v$  gewählt, sondern die mit dem höchsten Visit-Count  $n$ , da dieser Wert nicht so sehr zu Ausreißern neigt wie der Wert [19]. Vor allem bei der beschriebenen alternativen Bewertungsmethode muss stark darauf geachtet werden, dass nicht nach den Werten ausgewählt wird, da selbst die Node der besten nächsten Aktion zu Niederlagen führen kann, wodurch ein negativer Wert auf den Wert der Node addiert wird. Da die Node am vielversprechendsten ist, wird so im Laufe der Iterationen häufig eine negative Zahl aufaddiert. Eine Auswahl nach den Werten würde daher eine schlechtere Aktion, die nicht so oft besucht wurde, bevorzugen.

Der Algorithmus kann so lange laufen, wie es der Nutzer für nötig hält. Zu jeder Zeit kann ein Zug als vermeintlich bester nächster Zug vorgeschlagen werden.

MCTS durchsucht nicht jeden Teilbaum deterministisch nach dem besten Ergebnis, sondern arbeitet stochastisch, um den besten nächsten Zug zu approximieren. Da vielversprechende Knoten häufiger besucht werden, wächst der Baum asymmetrisch. Es werden nicht alle Zweige besucht, wodurch sich der Algorithmus gut für Anwendung auf Probleme eignet, die eine hohe Spielbaumkomplexität aufweisen und daher nicht mehr effektiv von Bruteforce-Algorithmen bearbeitet werden können.

Mit genügender Laufzeit oder Rechenleistung wird der Algorithmus sicher den bestmöglichen nächsten Zug approximieren. Die benötigte Laufzeit ist immer abhängig vom zu lösenden Problem. Im Vergleich zu anderen Algorithmen, wie Minimax oder  $\alpha - \beta$ -pruning, welcher ebenfalls den Suchraum minimiert, benötigt MCTS keine explizite Evaluationsfunktion, um den Suchraum zu untersuchen. MCTS-Algorithmen kommen ohne spielstrategisches Vorwissen und Heuristiken aus.

Problematisch ist, dass der Algorithmus langsam konvergiert und gewisse Expertenzüge oft nicht kommen sieht, da diese erst auf lange Sicht spielentscheidend sein können. Letzteres könnte eine Ursache dafür sein, dass Deepminds AlphaGo im vierten Spiel gegen Lee Sedol verloren hat. Lee Sedol spielte einen brillanten Zug, den kaum ein anderer Experte in Betracht gezogen hatte. Darauf hin spielte die KI schlagartig schlechte Züge und schaffte es nicht mehr, den entstandenen Vorsprung aufzuholen<sup>5</sup>.

---

<sup>5</sup>Quelle: <https://web.archive.org/web/20161116082508/https://gogameguru.com/lee-sedol-defeats->

### 9.3 Pseudocode

Im Folgenden ist der UCT-Algorithmus für die spielerabhängige Bewertung dargestellt.

---

#### Algorithm 1 UCT

---

**Require:** Spielstartzustand  $s_0 = s(0)$   
erstelle Root-Node  $n_0$  für  $s_0$   
**while** noch Rechenzeit übrig **do**

// Selektion  
**while** Node  $n$  hat Children **do**  
     $n \leftarrow \underset{n' \in \text{Children von } n}{\operatorname{argmax}} \left( \frac{V'}{N'} + \sqrt{\frac{2 \ln N}{N'}} \right)$   
**end while**

// Expansion  
**if**  $n$  repräsentiert keinen Endzustand **then**  
    **for** alle möglichen nächsten Spielzustände **do**  
        erstelle ein neues Child für  $n$   
    **end for**  
     $n \leftarrow$  ein zufälliges neues Child  
**end if**

// Simulation ab  $s = s(n)$   
**while**  $s$  ist kein Endzustand **do**  
    wähle Aktion zu nächstem  $s'$  zufällig und spiele diese  
     $s \leftarrow s'$   
**end while**  
berechne Spielergebnis  $r$  für den Endzustand  $s$

// Backpropagation  
**while**  $n$  hat Parent-Node **do**  
     $N \leftarrow N + 1$   
    **if** Sieger macht Aktion vor  $n$  **then**  
         $V \leftarrow V + 1$   
    **else**  
         $V \leftarrow V - 1$   
    **end if**  
**end while**  
**end while**

---

### 9.4 Parallelisierung

Um den Auswahlprozess des UCT-Algorithmus zu beschleunigen, kann man Teilprozesse parallelisieren. Chaslot et al. beschreiben dazu drei Möglichkeiten [5]. Bei der Leaf-Parallelisierung werden, nach der Expansionsphase gleichzeitig mehrere Zufallsspiele simuliert. Die Ergebnisse

werden dann gemittelt und im Baum wieder nach oben propagierte. Die Root-Parallelisierung arbeitet mit mehreren Suchbäumen, wobei jeweils einer pro Thread parallel zu den anderen aufgebaut wird. Zur Auswahl des Root-Childs mit der zu spielenden Aktion werden die Werte der Root-Children, die für dieselbe Aktion stehen, zusammenaddiert und die Node mit dem höchsten Gesamt-Visit-Count gewählt. Außerdem schlagen die Autoren die sogenannte Tree Parallelization vor, welche allerdings aufgrund der Nutzung von Mutex-Verfahren zu komplex für die Implementierung in dieser Arbeit ist.

UCT mit Root-Parallelisierung ist nicht äquivalent zum Standard-UCT-Algorithmus, sondern zum sogenannten Ensemble-UCT-Algorithmus [3].

## 9.5 Subtree-Erhaltung

Durch die Auswahl der vielversprechendsten Child-Node geht man im Suchbaum eine Ebene weiter nach unten und anschließend eine weitere durch den Zug des Gegners. In der Berechnung für die Child-Nodes wurde der Suchbaum weiter als bis zu diesen zwei Ebenen aufgebaut und es ist sinnvoll, den bereits berechneten Unterbaum („subtree“) für die Berechnung der nächsten eigenen Aktion zu behalten. Dadurch würden trotz gleicher Laufzeit insgesamt mehr Iterationen für den nächsten Entscheidungsprozess zur Verfügung stehen.

In der nicht parallelisierten Implementierung kann man dazu einfach den Baum unter der ausgewählten Node behalten. Um in der parallelen Version auch einen Teil des Baumes zu erhalten, wird aus den parallel erstellten Bäumen derjenige gespeichert, in welchem die Node nach der ausgewählten Aktion am häufigsten besucht wurde.

# 10 Anwendung am Brettspiel Carcassonne

Für eine Anwendung des MCTS-Algorithmus wurde das Brettspiel Carcassonne ausgewählt und programmiert. Ziel des Spieles ist es, durch Anlegen von Landschaftskarten und das Besetzen von Gebieten auf den Karten möglichst viele Punkte zu erspielen. Für eine genaue Regelbeschreibung wird auf die offizielle Spielanleitung verwiesen<sup>6</sup>.

Aus Gründen der Vereinfachung wurde nur das Grundspiel ohne Erweiterungen und Mini-Erweiterungen<sup>7</sup> implementiert.

## 10.1 Spieltheoretische Eigenschaften

Carcassonne ist ein diskretes, nicht deterministisches Spiel mit perfekten Informationen für zwei bis fünf Spieler. In dieser Arbeit wurden Agenten für das Spiel gegen einen Gegner implementiert.

### 10.1.1 Perfekte Informationen

Zu jeder Zeit des Spieles können beide Spieler den kompletten Spielzustand, bis auf die zufälligen nächsten Karten, überblicken. Kein Spieler verbirgt Informationen vor dem anderen (beispielsweise verdeckte Karten). Man nennt das Spiel daher ein Spiel mit perfekten Informationen.

<sup>6</sup>[https://www.hans-im-glueck.de/\\_Resources/Persistent/29c0ba5d1199419745fb69b01e87482761ed947f/CarcBasis\\_II\\_Regel\\_DE\\_16\\_Final.pdf](https://www.hans-im-glueck.de/_Resources/Persistent/29c0ba5d1199419745fb69b01e87482761ed947f/CarcBasis_II_Regel_DE_16_Final.pdf)

<sup>7</sup>Die Mini-Erweiterungen „Der Fluss“ und „Der Abt“

### **10.1.2 Determinismus**

In dem Spiel ziehen die Spieler abwechselnd zufällig aus den übrigen Karten, wobei manche davon häufiger vorkommen als andere (eine Übersicht aller Karten und deren Häufigkeiten findet sich im Regelwerk). Das Spiel ist also vom Zufall des Kartenziehens abhängig und damit nicht deterministisch.

Dies stellt ein Problem für die Anwendung des standard UCT-Algorithmus dar. Mögliche Aktionen sind nämlich durch den nicht deterministischen Spielcharakter abhängig von den gezogenen Karten, wodurch kein klassischer Suchbaum aufgebaut werden kann.

Es gibt mehrere Möglichkeiten, MCTS in modifizierter Form trotzdem auf nicht deterministische Spiele anzuwenden [3]. Viele dieser Modifizierungen haben gemein, dass das Spiel determiniert wird [3]. Der Zufall wird allgemein dadurch behoben, dass alle vom Zufall abhängigen Größen von vornherein einen bestimmten Wert annehmen. Carcassonne wird determiniert, indem eine Reihenfolge der Karten festgelegt wird. Dadurch ist für den Algorithmus klar, dass nach  $n$  Zügen immer wieder dieselbe Karte kommt. Im Spiel gegen einen Menschen, welcher jedoch meistens nicht weiß, welche Karte als nächstes kommt (es sei denn er weiß gegen Ende, welche Karten noch übrig sein müssen), stellt dies jedoch einen unfairen Vorteil dar.

Um wieder allgemeiner spielen zu können, gibt es verschiedene Möglichkeiten, beispielsweise den HOP-Algorithmus [3]. In diesem werden für mehrere Spiele zufällig deterministisch Kartentreihenfolgen festgelegt, welche dann vom Standard-UCT-Algorithmus bearbeitet werden. Danach wird über alle Spiele gemittelt, um die allgemein beste Aktion zu ermitteln und nicht die, welche für eine bestimmte Reihenfolge von Karten die beste ist.

## **10.2 Abschätzung der Spielkomplexität**

### **10.2.1 Zustandsraum-Komplexität**

Die Zustandsraum-Komplexität ist die Anzahl von möglichen Spielsituationen, welche vom Startzustand mit erlaubten Aktionen erreicht werden können. Heyden [13] ermittelte in ihrer Masterarbeit für Carcassonne eine stark unterschätzende Untergrenze für die Zustandsraumkomplexität von  $5 \cdot 10^{40}$  möglichen Zuständen. Schon diese Mindestgrenze ist so groß, dass sie zeigt, dass es nicht möglich ist, Carcassonne mittels eines Minimax-Algorithmus zu lösen.

### **10.2.2 Spielbaum-Komplexität**

Die Komplexität des Spielbaumes ist die Anzahl an möglichen Spielverläufen und entspricht damit der Anzahl der Leafnodes des kompletten Spielbaumes. Sie berechnet sich aus dem sogenannten „branching factor“, welcher die durchschnittliche Anzahl an Auswahlmöglichkeiten für die nächste Aktion darstellt. Die Spielbaum-Komplexität ist im Regelfall deutlich größer als die Zustandsraum-Komplexität, da dieselben Zustände in verschiedenen Verläufen an unterschiedlichen Stellen vorkommen können. Dies kann in Carcassonne beispielsweise dadurch passieren, dass die Karten für einen Zustand in anderer Reihenfolge gezogen, aber identisch gelegt und besetzt werden.

Heyden ermittelte als Schätzwert für die Spielbaum-komplexität von Carcassonne einen Wert von  $8.8 \cdot 10^{194}$  möglichen Spielverläufen.

Im Vergleich zu diesen Schätzungen besitzt Schach eine ähnliche Zustandsraum-Komplexität ( $\sim 10^{47}$ ), aber eine deutlich kleinere Spielbaum-Komplexität ( $\sim 10^{123}$ ) [18]. Die normale  $19 \times 19$  Go-Variante hat eine Zustandsraum-Komplexität von  $\sim 10^{172}$  und eine Spielbaum-Komplexität von  $\sim 10^{360}$  [1]. Go ist aufgrund seines großen Spielbrettes deutlich komplexer als Carcassonne. Jedoch wurden wie beschrieben mit modifizierten MCTS-Algorithmen kürzlich übermenschlich gut spielende Agenten für Go konstruiert [19, 20]. Es sollte daher theoretisch möglich sein, einen ähnlichen Agenten für Carcassonne zu konstruieren.

## 11 Implementierung

### 11.1 Allgemeines

Unter einer Aktion werden im Folgenden die Auswahl einer freien Anlegestelle mit der Anzahl an Rotationen vor dem Anlegen und die Positionierung einer Figur verstanden. Die Figuren im Spiel werden auch Meeples genannt.

### 11.2 Codeübersicht

Im Rahmen dieser Arbeit wurden sowohl das Spiel als auch die Monte-Carlo-Algorithmen in Python implementiert. Der Code sowohl für die Spielimplementierung als auch für die KI-Algorithmen sowie Anweisungen zum Nutzen der Funktionen befinden sich im Github-Repository <https://github.com/T3K14/Carcassonne>.

Der Code läuft unter Python 3.7 und verwendet, neben Modulen der Standardbibliothek, die Packages numpy (v. 1.17.2) und matplotlib (v. 3.1.1), sowie alle dafür benötigten Drittmodule, welche bei der Installation automatisch mitinstalliert werden sollten.

Im Skript `main_functions.py` befinden sich die zwei Hauptfunktionen für das Spielen von Carcassonne mit KIs. Die Funktion `ai_vs_ai` ist dazu da, um verschiedene KI-Spieler gegeneinander spielen zu lassen, wobei jeweils verschiedene Hyperparameter eingestellt werden können. Zu diesen zählen beispielsweise die Auswahl der Konstante  $c$  oder die Angabe der Rechenzeit<sup>8</sup>. Es ist einstellbar, wie viele Spiele zwischen den KIs gespielt werden sollen. Zu jedem Spiel wird eine Logdatei in den Ordner `simulations` geschrieben sowie eine Auswertungsdatei, welche die Ergebnisse aller Spiele zusammenfasst und Statistiken enthält.

Die zweite Funktion `human_vs_ai` ermöglicht es, als menschlicher Spieler gegen einen KI-Gegner zu spielen. Ein Userinterface wurde nicht dazu implementiert und die Befehlseingaben erfolgen über die Python Console. Nach jedem Spielzug wird in dieser Funktion ein Bild ausgegeben, welches das aktuelle Spielfeld darstellt. Da in dieser Grafik jedoch keine Meeplepositionen angezeigt werden, wird empfohlen, das Spiel nebenher mit echtem Spielzubehör mitzuspielen.

Für lange Laufzeiten auf mehreren Threads wird empfohlen, den Speicher des Computers im Blick zu behalten, da der Suchbaum ein bis zwei GB in Anspruch nehmen kann und Python daher immer mehr Speicher beansprucht. Durch gleichzeitiges, paralleles Berechnen mehrerer Bäume wurde teilweise soviel Speicher benötigt, dass die Simulation auf einem macOS mit einem Intel(R) Core(TM) i5 abgebrochen werden musste.

---

<sup>8</sup>Näheres dazu kann im Readme des Repositorys nachgelesen werden.

## 11.3 KI-Spieler

Im Rahmen dieser Arbeit wurden vier künstliche Spieler implementiert. Ein Zufallsspieler, welcher seine Züge zufällig auswählt, ein Simple-MC-Spieler, ein Flat-UCB-Spieler, sowie eine, auf dem UCT-Algorithmus basierende KI.

### 11.3.1 Simple-Monte-Carlo

Ein Standard-Monte-Carlo-Ansatz für die Bestimmung des nächsten Zuges ist es, jede vom aktuellen Zustand mögliche Aktion gleich oft zu spielen (abhängig von der verfügbaren Rechenzeit) und nach jeder gespielten Aktion zufällig zu Ende zu simulieren, um das Spielergebnis zu ermitteln. Dieses Ergebnis kann dann dazu genutzt werden, die gespielte Aktion zu bewerten. Am Ende wird dann die Aktion gespielt, welche in den Simulationen zu den besten Ergebnissen geführt hat.

### 11.3.2 Flat-UCB

In dieser Arbeit wurde außerdem eine leicht modifizierte Version der Simple-MC-Methode implementiert. Statt alle nächsten Aktionen gleich häufig zu untersuchen, werden die Aktionen entsprechend der Bandit-Auswahlmethode gewählt. Zu jeder Aktion werden die Anzahl an Untersuchungen und die Spielergebnisse der Spiele, die nach der Aktion simuliert wurden, gespeichert. Bei dieser Methode wird kein Spielbaum aufgebaut und man nennt sie daher auch Flat-UCB [3].

### 11.3.3 UCT

Schließlich wurde der UCT-Algorithmus mit der beschriebenen, spielerabhängigen Bewertung von Nodes implementiert. Aus Gründen der limitierten Rechenleistung wurde in den simulierten Testspielen zu Beginn deterministisch eine feste Kartenreihenfolge festgelegt, mit welcher der UCT-Algorithmus arbeiten kann. Da dadurch in Spielen gegen menschliche Spieler ein unfaire Vorteil entsteht, wird die Reihenfolge der Karten für den Menschen ebenfalls offengelegt. Eine optionale Root-Parallelisierung sowie die Erhaltung von Subtrees wurden mit implementiert.

## 11.4 Evaluation

Für Carcassonne bietet es sich an, die Punktedifferenzen der Spieler zu verwenden, um die Nodewerte zu aktualisieren (siehe Gl. (20)). Durch das Vorzeichen bleiben weiterhin Sieg und Niederlage als solche erkennbar, jedoch werden Züge höher bewertet, die oft zu höheren Punktzahlen führen. Es wird also nicht bloß bewertet, ob der Algorithmus gewinnt oder nicht, sondern auch, wie hoch das jeweilige Ergebnis ausfällt. Das gibt dem Agenten mehr Spielraum gegen gut spielende Gegner. Spielt die KI nämlich nur auf Sieg, wird sie auch schlechte Züge spielen, wenn sie der Meinung ist, dass auch diese sie zum Sieg führen.

Diese Evaluationsmethode wurde für den Simple-MC-, den Flat-UCB-, sowie für den UCT-Algorithmus verwendet.

Aufgrund dieser Evaluationsart wird final die Aktion der Node gespielt, welche am häufigsten besucht wurde (siehe Abs. 9.2).

## 12 Auswertung

Zu jedem simulierten Spiel zwischen zwei KIs wird eine Datei erzeugt, welche den Spielverlauf und das Ergebnis enthält. Es kann angegeben werden, ob die KIs jeweils ein bestimmtes Zeitlimit oder eine Anzahl an Iterationen haben, um den nächsten Zug zu berechnen. Bei einer Gesamtzahl von  $n$  Spielen zwischen zwei KIs beginnt jede die Hälfte aller Spiele.

**Anmerkung:** Kurz vor Abgabe wurde noch ein Fehler in der Spiellogik gefunden, welcher dafür sorgte, dass in seltenen Fällen ein Spieler für eine Straße die doppelte Punktzahl erhält sowie die doppelte Anzahl an darauf gesetzten Meeples zurückbekommt. Da dieser Fehler weder besonders gravierend ist, noch oft aufgetreten sein kann und alle KIs diesen Fehler zu ihrem Vorteil nutzen konnten, unterscheiden sich die Simulationsergebnisse kaum von denen einer korrekten Implementierung. Für das Spiel zwischen dem UCT-Algorithmus und einem menschlichen Spieler wurde der Fehler behoben.

### 12.1 Flat-UCB

#### Flat-UCB gegen einen Zufallsspieler

Zu Beginn wurde die Flat-UCB-Methode getestet. Sie spielt zuerst gegen einen Gegner, welcher gleichverteilt zufällig eine von allen möglichen nächsten Aktionen auswählt. In Tabelle 1 sind die Ergebnisse aus 20 simulierten Spielen dargestellt, wobei der Flat-UCB-Algorithmus für die Berechnung jedes Zuges 500 Iterationen zur Verfügung hatte.

Tabelle 1: Spielergebnisse und durchschnittlich erreichte Punktzahlen (mit Standardabweichungen)

	random		Flat-UCB		# Draws
	Siege	Ø Punkte	Siege	Ø Punkte	
random beginnt	0	16,8 (s=4,3)	10	103,7 (s=16,0)	0
Flat-UCB beginnt	0	15,2 (s=5,9)	10	100,6 (s=18,9)	0
gesamt	0%	16,0 (s=5,1)	100%	102,2 (s=16,9)	0%

Der Flat-UCB-Algorithmus zeigt offensichtlich ein funktionierendes Spielverständnis. Er hat in den 20 Spielen durchschnittlich mehr als 6 mal so viele Punkte erspielt wie der zufällig spielende Gegner und alle Spiele gewonnen. In den Abbildungen 4, 5 und 6 sieht man genauer, wie diese Ergebnisse zustande kommen.

In Grafik 4 sieht man, dass die Flat-UCB-KI deutlich mehr Figuren setzt als der zufällig spielende Gegner. Lediglich auf Wiesen setzt der Zufallsspieler häufiger Meeples. Das liegt daran, dass es auf allen Karten (bis auf eine) die Möglichkeit gibt, eine Figur auf mindestens eine Wiese zu setzen. Dementsprechend wählt ein Zufallsspieler häufig eine solche Aktion. Der Flat-UCB-Spieler setzt zwar seltener Meeples auf Wiesen, tut dies aber deutlich strategischer, wie man in Diagramm 5 und 6 sieht. Mit allen Figuren, die der Spieler auf Wiesen setzt, erzielt er zusammen durchschnittlich 17 Punkte. Teilt man die zusammen erspielte Anzahl an Punkten pro Gebietsart durch die Anzahl gesetzter Figuren auf diese Gebietsart und mittelt diesen Wert über alle Spiele, erhält man die durchschnittlich pro Spiel erhaltenen Punkte pro Gebiets-Meeple. Dieser Wert beschreibt dann nur solche Spiele, bei denen die jeweilige KI mindestens eine Figur

auf ein bestimmtes Gebiet gesetzt hat.

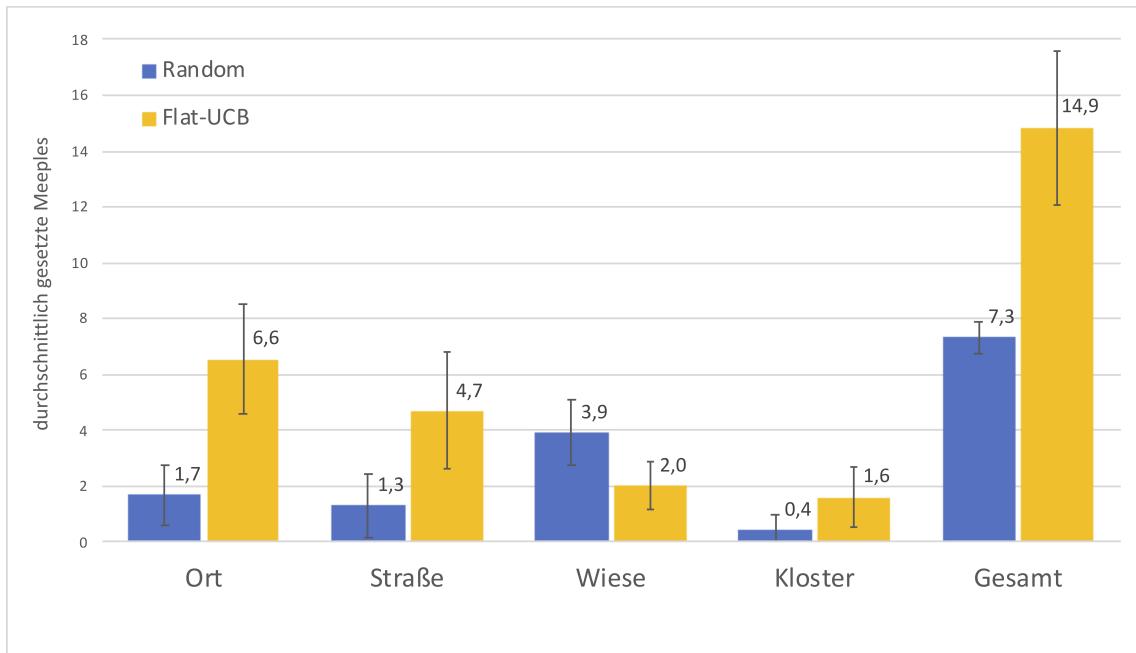


Abb. 4: Die durchschnittlich pro Spiel gesetzte Anzahl von Meeples auf die jeweiligen Gebiete (mit Standardabweichung).

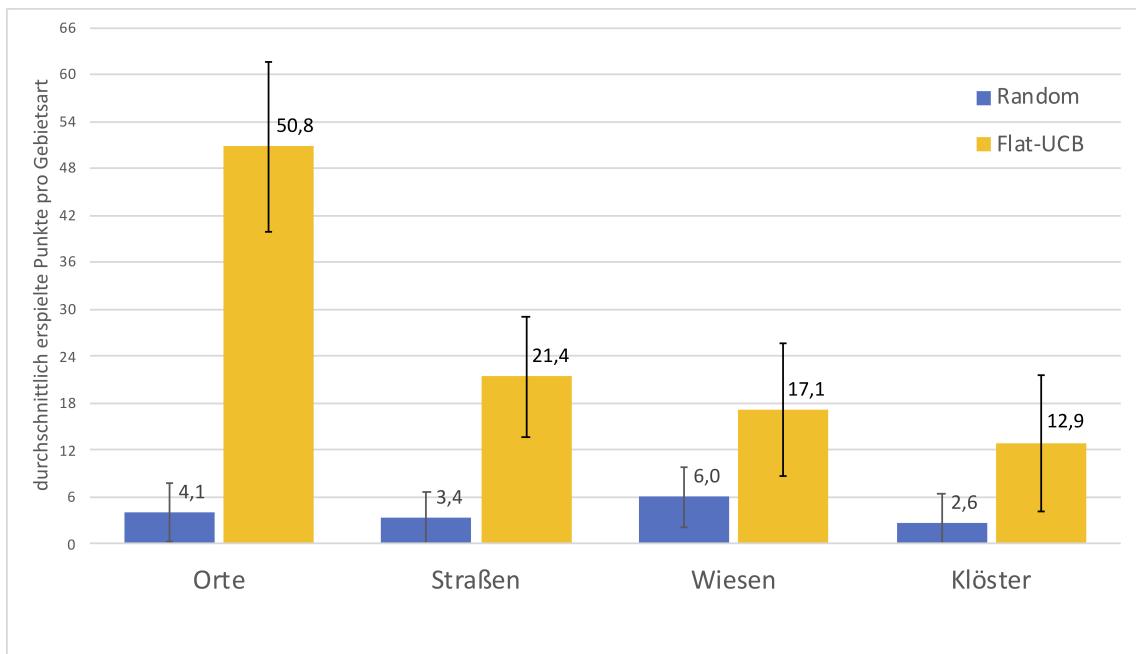


Abb. 5: Die durchschnittlich pro Spiel erzielte Punktzahl mit Gebieten, bei denen der jeweilige Spieler beteiligt war (mit Standardabweichung).

In Abbildung 6 ist zu erkennen, dass der Flat-UCB-Algorithmus so spielt, dass er, falls er einen Meeple auf eine Wiese setzt, damit durchschnittlich 9,3 Punkte pro Spiel generiert. Er nutzt zwar weniger Meeples für Wiesen, spielt aber so, dass er mit den wenigen deutlich mehr Punkte erzeugt als der Zufallsspieler. Für die übrigen Gebiete erspielt er zusätzlich zu der höheren Anzahl an gesetzten Meeples auch pro gesetzter Figur deutlich mehr Punkte.

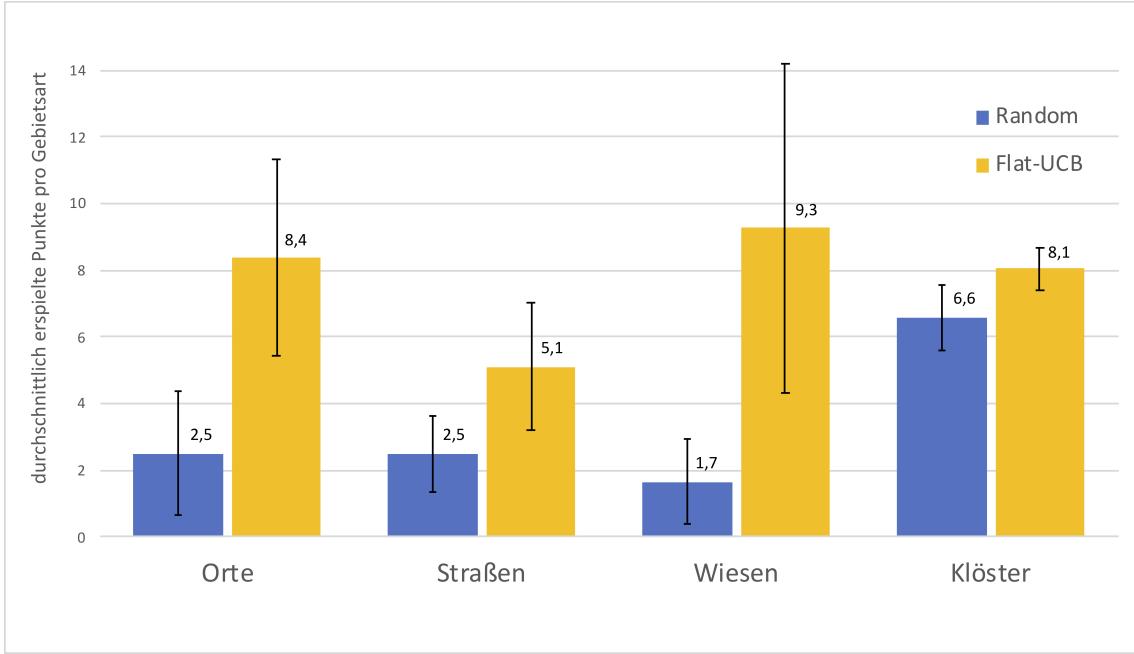


Abb. 6: Die durchschnittlich pro Spiel erzielte Punktzahl pro gesetztem Meeple auf ein Gebiet (mit Standardabweichung).

### Flat-UCB gegen Simple-MC

Zum Vergleich zwischen der KI, die auf einem klassischen MC-Ansatz basiert und dem Flat-UCB-Algorithmus, wurden 50 Spiele zwischen diesen KIs gespielt. Beide Algorithmen durften für jede Entscheidung 500 Spiele simulieren. Die Ergebnisse sind in Tabelle 2 dargestellt.

Tabelle 2: Spielergebnisse und durchschnittlich erreichte Punktzahlen (mit Standardabweichungen)

	Simple MC		Flat-UCB		# Draws
	Siege	Ø Punkte	Siege	Ø Punkte	
Simple-MC beginnt	7	67,4 (s=14,9)	18	82,2 (s=13,9)	0
Flat-UCB beginnt	7	70,4 (s=18,5)	18	78,8 (s=15,9)	0
gesamt	28%	68,9 (s=16,7)	72%	80,5 (s=14,9)	0%

Es ist zu erkennen, dass die Flat-UCB-KI durchschnittlich etwa 10 Punkte mehr pro Spiel erspielt und von den 50 gespielten Spielen über zwei Drittel gewonnen hat.

In Abbildung 7 erkennt man, dass der Flat-UCB-Spieler wieder mehr Meeples im Spiel nutzt. Der Wert ist mit 12,6 bei gleicher Laufzeit jedoch deutlich niedriger als der Durchschnittswert 14,9 der Spiele gegen den Zufallsspieler. Das liegt u.A. daran, dass ein guter Spieler gelegentlich

gegnerische Vorhaben sabotiert, statt eigene Punkte zu erspielen. Aufgrund der Spielsimulationen (Gl. (20)) soll der Spieler falls er führt die Aktionen spielen, welche dafür sorgen, dass der Abstand zum Gegner maximal wird, bzw. die Aktion, welche die Punktedifferenz verkleinert, falls er hinten liegt. Dabei kann es sich anbieten auf ein paar Punkte zu verzichten und eine Karte so zu legen, dass der Gegner nun besondere Karten benötigt, um beispielsweise eine große Stadt abschließen zu können. Im besten Fall bekommt er keine Karte mehr, die ihm die Fertigstellung erlaubt. Letzters ist aus Sicht des Gegners besonders wichtig, da dadurch ein Meeple verloren geht, welcher nicht mehr für neue Projekte genutzt werden kann.

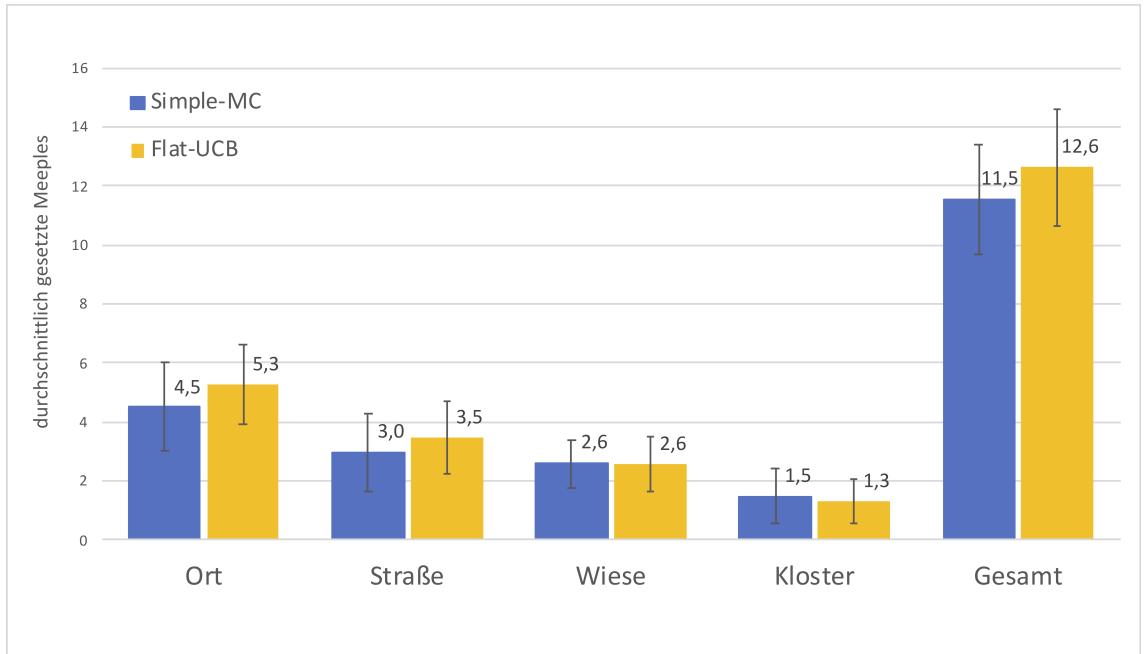


Abb. 7: Die durchschnittlich pro Spiel gesetzte Anzahl von Meeples auf die jeweiligen Gebiete (mit Standardabweichung).

Insgesamt setzt der Flat-UCB-Spieler dennoch durchschnittlich über eine Figur mehr pro Spiel, als der Simple-MC-Algorithmus. Zusätzlich erspielt er auch mit jeder Gebietsart mehr Punkte pro Spiel, was in Abb. 8 ersichtlich ist. Vor allem mit Orten und Straßen verdient die Flat-UCB-KI mehr Punkte als der Simple-MC-Spieler. Insgesamt erspielt die Flat-UCB-KI durchschnittlich mit jedem gesetzten Meeple pro Gebietsart mehr Punkte. Mit 7,6 Punkten pro gesetztem Klostermeeple, bei der Simple-MC-KI und 7,9 Punkten für die Flat-UCB-KI (siehe 9) erreichen beide Algorithmen gute Werte, da ein Kloster maximal 9 Punkte einbringen kann. Die Klöster werden also entweder bereits an geeignete Stellen mit vielen Nachbarkarten angelegt, oder beide KIs legen viel Wert darauf, begonnene Klöster abzuschließen. Beides ist auch dadurch sinnvoll, dass man schnell wieder eine Figur zurückbekommt und weiter nutzen kann.

Es zeigt sich im Vergleich zu den Spielen gegen den Zufallsgegner, dass die Flat-UCB-KI gegen den stärkeren Gegner weniger Meeples setzt und damit weniger Punkte erspielt. Besonders bemerkbar, macht sich dies in den Werten für Orte und Wiesen. Da die Flat-UCB-KI gegen einen stärkeren Spieler spielt, kann die KI nicht so oft kleine Orte abschließen, welche der Zufallsspieler noch offen gelassen hätte. Außerdem ist es ihr nun nicht mehr so einfach möglich, sich mit wenigen Meeples an die wichtigen Wiesen anzuschließen. Der Flat-UCB-Spieler nutzt in den Spielen etwas mehr Figuren für Wiesen als zuvor, erspielt aber nur etwas mehr Punkte mit

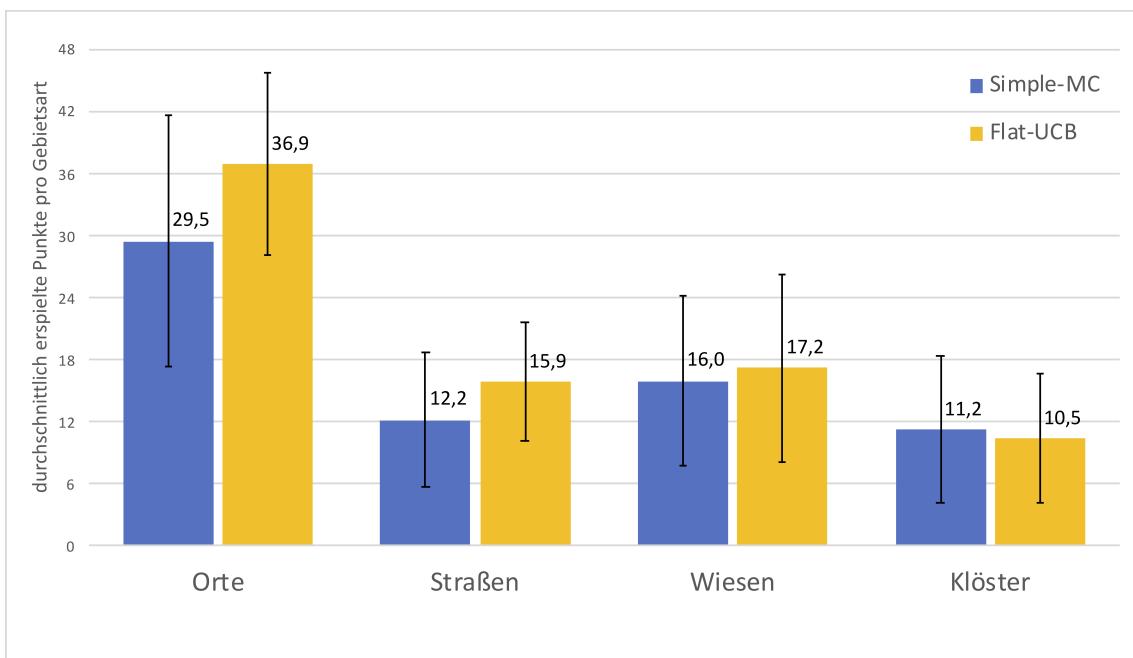


Abb. 8: Die durchschnittlich pro Spiel erzielte Punktzahl mit Gebieten, bei denen der jeweilige Spieler beteiligt war (mit Standardabweichung).

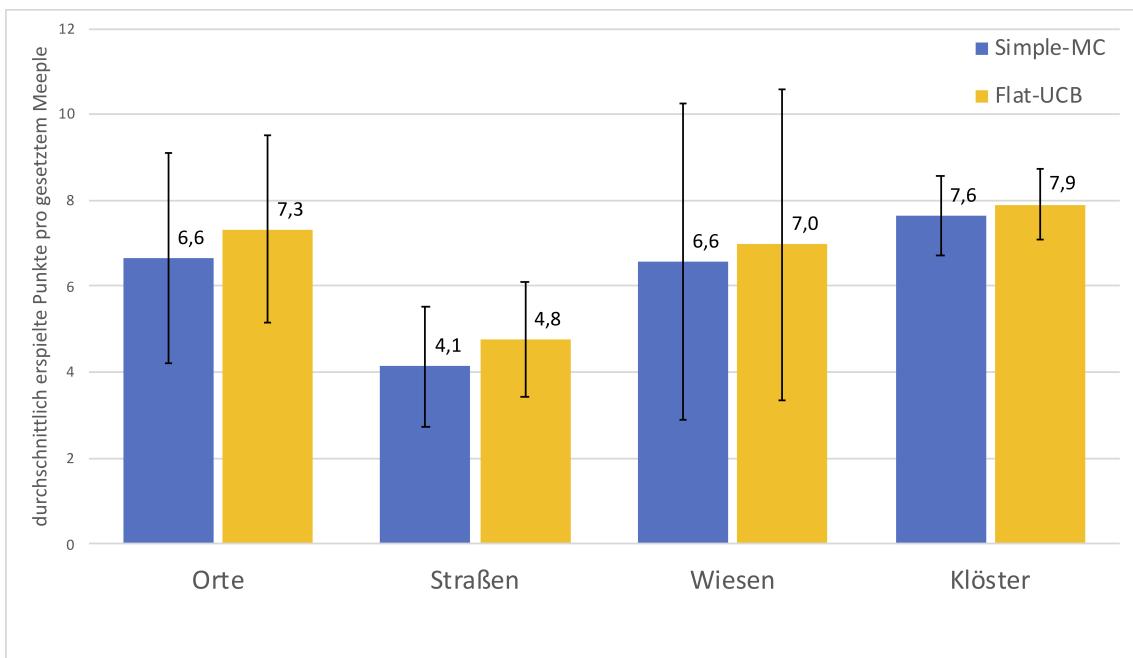


Abb. 9: Die durchschnittlich pro Spiel erzielte Punktzahl pro gesetztem Meeple auf ein Gebiet (mit Standardabweichung).

Wiesen, als sein Gegner.

Man hätte die KIs auch so programmieren können, dass sie immer, versuchen ihre eigenen Punkte zu maximieren. Das würde allerdings auch dazu führen, dass der Gegner häufig unge-

stört große Städte bauen kann und keine Meeples blockiert werden.

Der entscheidende Vorteil in diesen simulierten Spielen für den Flat-UCB-Algorithmus ist, dass er häufiger und strategischer Meeples auf Orte und Straßen platziert hat. Da Straßen generell nicht so viele Punkte einbringen (1 Punkte pro Straßenstück), wäre es jedoch sinnvoller, sich eher auf Orte und Wiesen zu konzentrieren. Durchschnittlich 12,6 gesetzte Figuren sind bereits ein guter Wert, aber bei 36 zu ziehenden Karten ist ein höherer Wert möglich.

## 12.2 UCT

### UCT gegen Flat-UCB im Standardspiel

Als nächstes wurde die UCT-KI getestet. Sie spielte 50 Spiele gegen den Flat-UCB-Algorithmus, da dieser bisher die besten Ergebnisse erzielt hat. Beide KI-Spieler nutzen als Auswahlkonstante den Wert  $c = \sqrt{2}$  und hatten 60 Sekunden Zeit, um ihren nächsten Zug zu berechnen. Die Ergebnisse sind in Tabelle 3 dargestellt.

Tabelle 3: Spielergebnisse und durchschnittlich erreichte Punktzahlen (mit Standardabweichungen)

	Flat-UCB		UCT		# Draws
	Siege	Ø Punkte	Siege	Ø Punkte	
Flat-UCB beginnt	11	78,2 (s=14,6)	14	78,6 (s=12,9)	0
UCT beginnt	9	75,5 (s=15,3)	15	76,6 (s=14,4)	1
gesamt	40%	76,9 (s=14,9)	58%	77,6 (s=13,6)	2%

Es ist zu erkennen, dass beide Spieler pro Spiel durchschnittlich fast gleich viele Punkte erspielt haben.

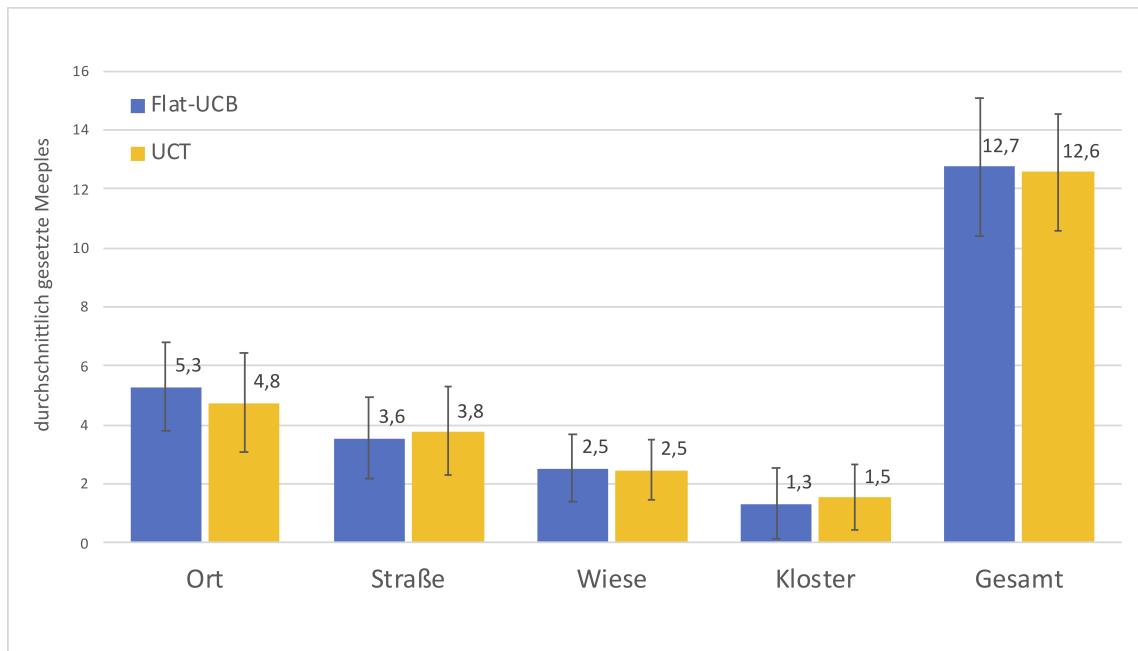


Abb. 10: Die durchschnittlich pro Spiel gesetzte Anzahl von Meeples auf die jeweiligen Gebiete (mit Standardabweichung).

In Abbildung 11 erkennt man, wie sich die Punkte verteilen. Der Flat-UCB-Spieler hat etwas mehr Punkte mit Orten erzielt, während die UCT-KI mehr mit Klöstern gepunktet hat. Der höhere Klosterwert für den UCT-Spieler kommt einerseits daher, dass dieser Spieler in den simulierten Spielen häufiger Klosterkarten gezogen hat und andererseits dadurch, dass er den übernächsten Zug bereits vorhersehen konnte (siehe Abs. 12.2) und so in der Lage war, noch einen Meeple durch Beenden eines Ortes, einer Straße, oder eines anderen Klosters zurückzuholen, um diesen dann im übernächsten Zug für das nächste Kloster verwenden zu können. Es kam in den Spielen oft vor, dass beide Spieler keine Figuren mehr übrig hatten, um sie auf ein gezogenes Kloster zu setzen, wodurch häufig viele Punkte verloren gingen.

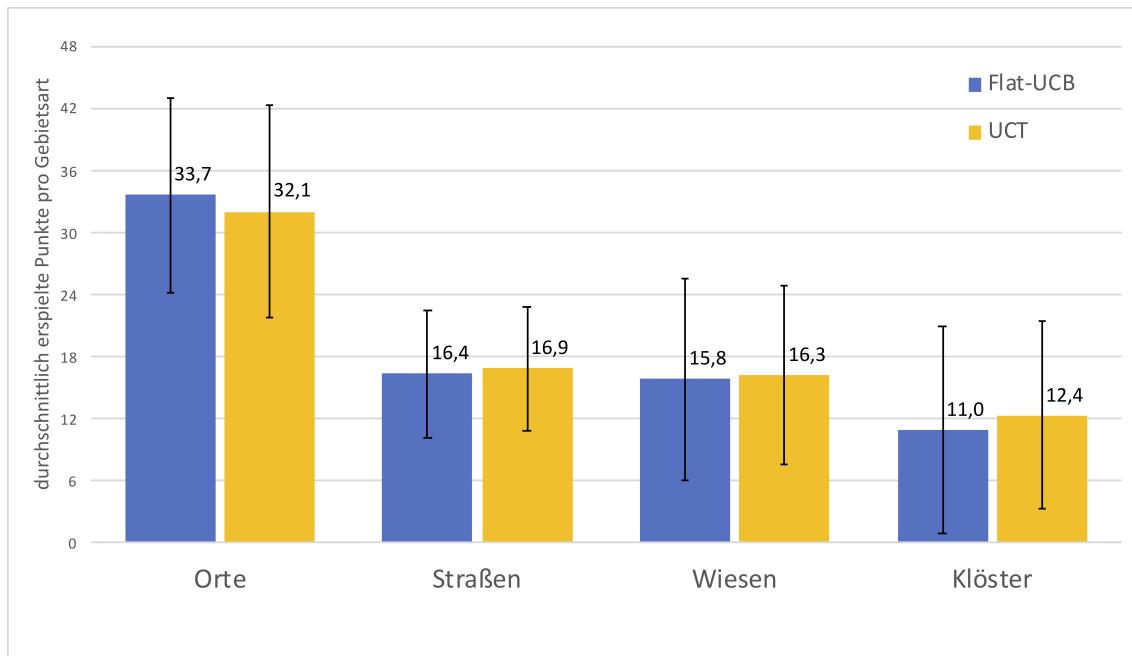


Abb. 11: Die durchschnittlich pro Spiel erzielte Punktzahl mit Gebieten, bei denen der jeweilige Spieler beteiligt war (mit Standardabweichung).

Ein gutes Beispieldispiel ist game1<sup>9</sup>. In diesem Spiel ist zu erkennen, dass beide Spieler nicht weitsichtig genug spielen. Auf kurze Sicht finden beide sehr gute Aktionen, was allerdings dazu führt, dass beide Spieler sehr schnell alle Mepples verbraucht haben und teilweise früh Figuren auf Gebiete stellen, die am Ende des Spiels wenig Punkte einbringen. Auf diesen Gebieten verbringen die Meeples dann fast das ganze Spiel und können nicht anders verwendet werden. Erfreulicherweise fällt auf, dass beide Spieler es mit ihren Strategien schaffen, gegnerische Bauvorhaben zu sabotieren, oder sich an wertvolle gegnerische Gebiete anzuschließen.

Die Stärke des UCT-Algorithmus deutet sich dadurch an, dass er in game1 mehrfach Züge gespielt hat, welche in Kombination mit dem nächsten eigenen Zug zu einem größeren Spielvorteil geführt haben. Beispielsweise beendete er erst eine Straße, um eine Figur für eine weitere Straße zu bekommen. Die nächste Karte setzte der Algorithmus so, dass sie einen Ort des Gegners sabotierte und platzierte die Figur so auf eine kleine Straße, dass sie direkt wieder zum ihm zurückging. Dieser Meeple konnte dann, bei der nächsten Karte, wieder auf ein Kloster gesetzt werden. Vor allem gegen Ende des Spieles spielte der UCT-Algorithmus stärker, was

<sup>9</sup>Die Logdateien der einzelnen Spiele befinden sich im Git-Repository in den jeweiligen Simulationsordnern

damit zusammenhängt, dass die Zufallssimulationen mit jeder bereits gesetzten Karte schneller berechnet werden können, wodurch in der gegebenen Rechenzeit mehr UCT-Iterationen durchgeführt werden können.

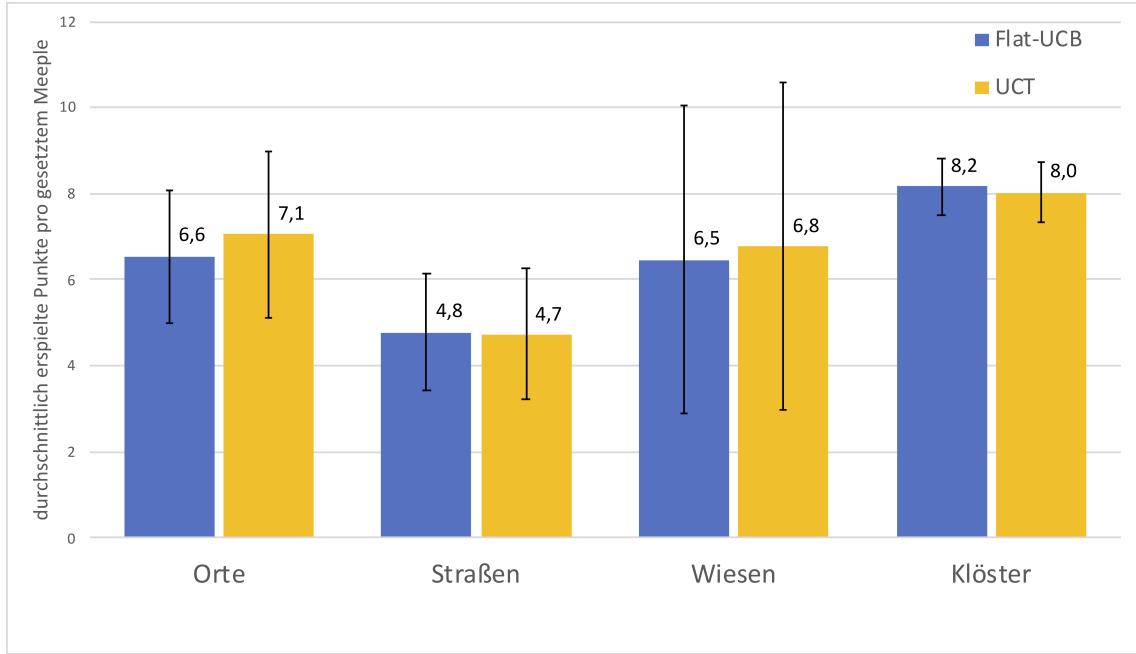


Abb. 12: Die durchschnittlich pro Spiel erzielte Punktzahl pro gesetztem Meeple auf ein Gebiet (mit Standardabweichung).

Aufgrund der vielen Karten, die gezogen werden, sollte die Möglichkeit, das Spiel zu beginnen, so lange keine Rolle spielen, bis die Spieler annähernd perfekt spielen können. Dies deutet sich in den Spielausgängen in den Tabellen 2 und 3 an. Die Anzahl der Siege und die Durchschnittspunkte beider Spieler bleiben unabhängig von der Startreihenfolge ungefähr gleich.

#### **Probleme dieser Implementierung:**

Allgemein ist zu sagen, dass 50 simulierten Spiele nur eine extrem kleine Stichprobe aus der Menge aller möglichen Spielverläufe sind und nur grobe Annäherungen an die tatsächlichen Durchschnittswerte erlauben. Es kann z.B. gut sein, dass der UCT-Spieler einfach häufiger Glück mit der Kartenreihenfolge hatte. Es gab genauso Spiele, bei denen der Flat-UCB-Spieler mit mehr als 30 Punkten Vorsprung gewann. Man darf also nicht zu viel in die Ergebnisse interpretieren.

Aufgrund des hohen branching-factors von Carcassonne kommt der UCT-Algorithmus in der gegebenen Laufzeit nicht sehr tief im Suchbaum. Für jede neue Node liegen durchschnittlich etwa 55 neue auf der nächsten Ebene [13]. Für die dritte Ebene kommt man bereits auf 170000 Nodes. All diese Child-Nodes müssen aufgrund des Visit-Counts im Nenner (Gl. (17)) mindestens einmal besucht werden, bevor der Bandit-Auswahlprozess auf Höhe dieser Ebene überhaupt erst starten kann.

Mit wenigen Iterationen funktioniert der UCT-Algorithmus daher kaum anders als der Flat-UCB-Algorithmus. Er schließt zwar durch die Bandit-Auswahl bereits auf der direkten Child-Ebene einen Teil des Suchbaums aus, jedoch reichen die Iterationsschritte nicht für ein vorausschauendes Spielen aus. Hinzu kommt, dass der Algorithmus durch den Aufbau des Such-

baumes komplexer als der Flat-UCB-Algorithmus ist und daher zusätzlich für jede Iteration mehr Rechenzeit benötigt.

Da möglicherweise sinnvolle Züge außerdem nicht aufgrund einer negativ geendeten Simulation vorzeitig ausgeschlossen werden sollen, muss man eine Erhöhung der Erkundungskonstante  $c$  in Betracht ziehen. Gerade zu Beginn eines Spieles, kann im Verlauf der Zufallssimulationen ein eigentlich guter Zug nach der ersten Zufallssimulation stark negativ bewertet werden. Bei der kleinen Standardkonstante von  $\sqrt{2}$  und kurzen Laufzeiten führt das dann dazu, dass die entsprechende Aktion nie mehr in Betracht gezogen wird.

Dem letzten Problem steuert die Parallelisierung etwas entgegen, da es nun auf allen Parallelbäumen gleichzeitig auftreten muss, um zu bewirken, dass der sinnvolle Zug überhaupt nicht beachtet wird. Bei der anschließenden Addition der Visit-Counts müssten allerdings die Bäume, welche den guten Zug erkannt haben so hohe Besuchszahlen dieser Node aufweisen, dass andere weniger gute, aber überall relativ häufig gespielte Züge nicht den Vorrang erhalten.

### UCT gegen Flat-UCB mit verkürztem Spiel

Um die Vorteile des UCT-Algorithmus aufzuzeigen, wurden 10 weitere Spiele zwischen ihm und dem Flat-UCB-Algorithmus gespielt. In diesen Spielen, wird eine stark verkürzte Version von Carcassonne mit 10 der 71 Karten gespielt (siehe Abbildung 13). Dabei bleibt die Kartenreihenfolge immer gleich und nach fünf Spielen wird der Startspieler getauscht. Der Spieler, der als zweiter zieht hat aufgrund der beiden Klöster, die er ziehen wird, einen Vorteil und die Möglichkeit durch perfektes Spiel immer zu gewinnen. Beide Algorithmen haben in diesen Spielen für jeden Zug 600 Sekunden Rechenzeit erhalten.



Abb. 13: Kartenauswahl für verkürztes Spiel. Obere Zeile: Karten, die der Startspieler ziehen wird, Untere Zeile: Karten des Gegenspielers.

Durch Untersuchen der Visit-Counts der Nodes beim UCT-Algorithmus, wurde ein Wert von  $c = 4$  gewählt. Bei niedrigeren Werten besteht die beschriebene Gefahr, dass gute Züge unentdeckt bleiben und bei zu hohen Werten, dass zu wenig Iterationen für vielversprechende Züge genutzt werden. Letzteres bedeutet, dass die Folgen der Züge, die letztendlich gespielt werden weniger gut untersucht sind, wodurch weniger weitsichtig gespielt wird. Für eine optimale Wahl der Konstante sollte man UCT-Algorithmen mit verschiedenen Konstanten gegeneinander spielen lassen und ermitteln, welcher Wert zu den besten Ergebnissen führt.

Bei dieser verkürzten Spiellänge und der hohen Laufzeit zeigt sich die Stärke des UCT-Algorithmus. In den fünf Spielen bei denen der Flat-UCB-Spieler anfing, konnte der UCT-Spieler den Kartenvorteil ausnutzen und gewann alle Spiele. Von den Spielen, in denen der UCT-Spieler startete, verlor er nur ein Spiel und schaffte es, in den übrigen Spielen Unentschieden zu spielen. Insgesamt wurden fünf verschiedene Spielverläufe beobachtet. Diese sind im Folgenden beschrieben.

**Flat-UCB beginnt:** Drei der fünf Spiele resultierten im Ergebnis von Abb. 14a, die anderen beiden endeten im Spielzustand von Abb. 14b. Die Aktion, welche für die unterschiedlichen Ergebnisse verantwortlich ist, ist der, in Abb. 14b markierte, vierte Zug des Flat-UCB-Spielers an den Koordinaten (0/3). Aufgrund der Annahme, dass der Gegner seine Aktionen zufällig wählt, nimmt der Flat-UCB-Spieler an, dass ihn auch dieser Zug häufig zum besten Ergebnis bringt. Wie auch immer er seine letzte Karte setzt, kann er nun jedoch nicht mehr verhindern, dass der UCT-Spieler mit seinem letzten Kloster sechs weitere Punkte erspielt (entweder auf (-1,1), oder (1,1), je nach dem, welcher Platz noch zugänglich ist).

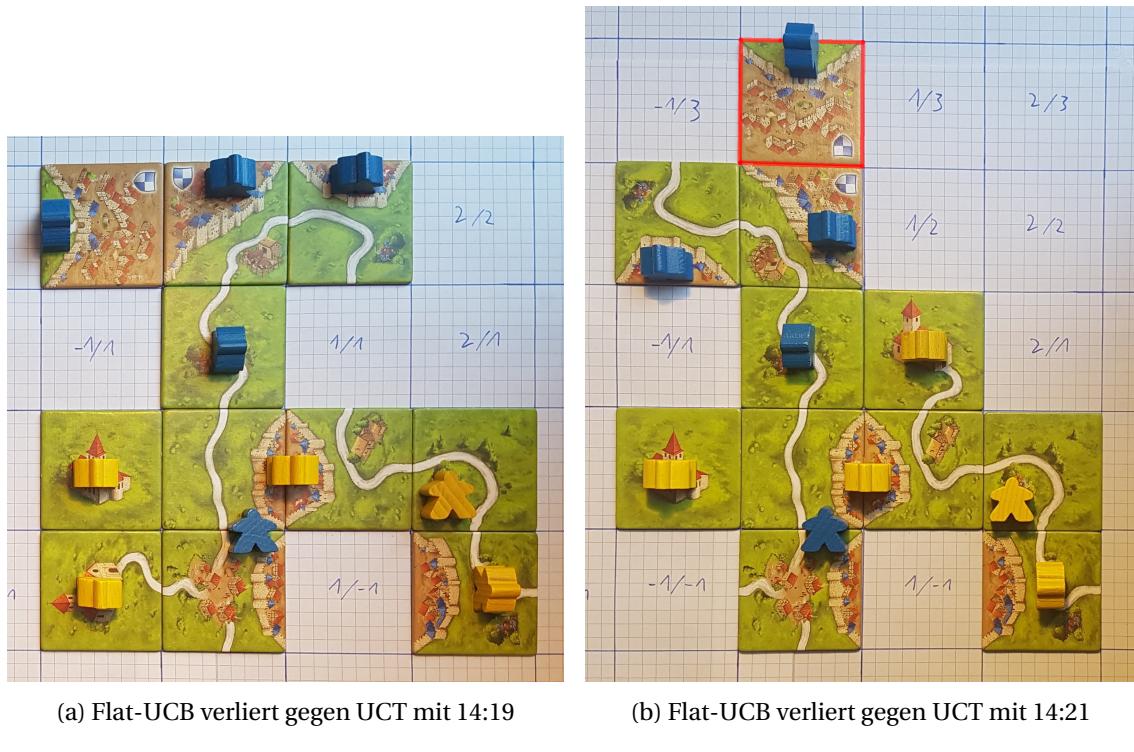


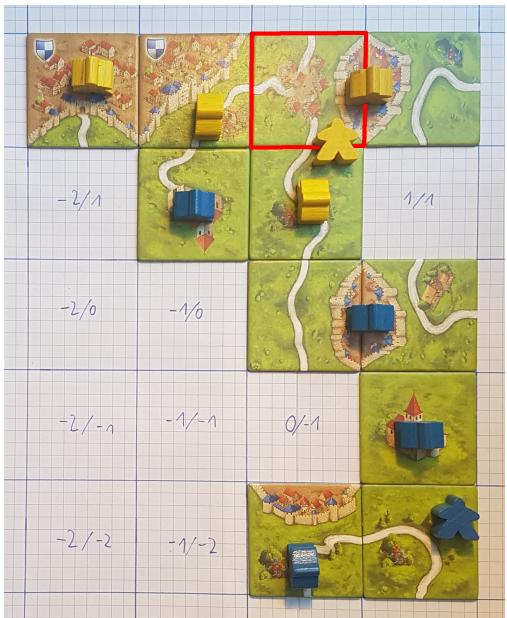
Abb. 14: Flat-UCB (blau) beginnt gegen UCT (gelb)

**UCT beginnt:** Die ersten beiden Züge sind, unabhängig davon welche KI begonnen hat, identisch gewesen. Der UCT-Spieler spielt seine erste Karte ebenso auf (0,1) und setzt eine Figur auf die Straße. In einer beobachteten Simulation, welche im Debugger lief, wurde die Node zu dieser Aktion von insgesamt 33236 Iterationen 11059 mal ausgewählt. Der Zug des Gegners auf (1,0) und die Besetzung des Ortes wurden mit etwa 30% dieser 11059 Iterationen bereits korrekt vorhergesehen. Statt der Aktion ((0,-1) mit einem Meeple auf der rechten Wiese), welche vorher von dem Flat-UCB-Algorithmus gewählt wurde, wird mit 9% der Selektionen bereits die

Aktion auf (0,2), mit einem Meeple auf derselben Wiese, präferiert (markiert in Abb. 15a). So-  
gar die besten möglichen Anlegestellen für das erste Kloster des Gegenspielers werden richtig  
vorhergesehen.

Mit diesem ersten Zug hat die UCT-KI also bereits die nächsten zwei bis drei Züge vorherge-  
sehen und auf Basis dieser Vorhersagen seine Entscheidung getroffen. Dies stellt einen großen  
Spielvorteil gegenüber dem Flat-UCB-Algorithmus dar. Da das Programm standardmäßig,  
und nicht im Debugger lief, wurden zusätzlich mehr Iterationen in der vorgegebenen Rechen-  
zeit geschafft.

Der Flat-UCB-Spieler spielt den erwarteten Zug, was dazu führt, dass durch das Speichern des  
Subtrees zusätzlich zu den berechneten 36229, die 3309 bereits berechneten Iterationen hin-  
zukommen. Für den zweiten Zug entscheidet sich der UCT-Algorithmus wie erwartet für die  
markierte Aktion, hat aber viele Iterationen in andere Aktionen, wie z.B. die vorher vom Ge-  
ner gespielte, investiert. Für die nächste Aktion des Gegners erwartet er das Kloster eher bei  
(-1,1) als bei (1,-1). Flat-UCB entscheidet sich jedoch in allen gespielten Partien für die zweite  
Variante, welche jedoch bei anschließendem perfektem Spiel kein Hindernis für einen sic-  
heren Sieg darstellt. Für die nächste eigene Karte werden bereits die Züge präferiert, welche auch  
in den gezeigten Spielen gewählt wurden. Auch hier trifft der Algorithmus Entscheidungen mit  
Rücksicht auf die nächsten zwei Ebenen im Spielbaum.



(a) Das Spiel endet 20:20, Unentschieden



(b) UCT verliert gegen Flat-UCB mit 20:23

Abb. 15: UCT (gelb) beginnt gegen Flat-UCB (blau)

Der UCT-Spieler spielt in allen drei Spielszenarien immer sehr ähnlich. Das unterschiedliche  
Bild von Spiel 16 kommt daher, dass die KI ihre dritte, markierte Karte auf (0,3) setzte. Aus  
Punktesicht sind beide gespielte Aktionen gleich sinnvoll, da der Spieler für die Straße und  
den Ort immer dieselbe Punktzahl bekommt, wodurch sich die Gesamtpunktedifferenz zum  
Gegner nicht ändert. Der zusätzliche Punkt für die UCT-KI in den Spielen 15a und 15b kommt  
lediglich dadurch, dass durch das letzte Kloster vom Flat-UCB-Spieler ein weiterer Straßenteil  
für eine der Straßen des UCT-Spielers hinzukommt.

Die Entscheidung, die dem Flat-UCB-Spieler im Spiel der Abb. 15b zum Sieg verholfen hat war, seine vorletzte Karte nicht auf (0,-2) zu legen. Dadurch hatte er die Möglichkeit, mit seinem letzten Kloster die beiden besetzten Wiesen zu verbinden und konnte drei Punkte für den oberen abgeschlossenen Ort dazugewinnen. Da er seine dritte Kurvenkarte ungünstig gedreht unter das Kloster gelegt hat, wäre es in Spiel 15a die bessere Alternative für ihn gewesen, seine vorletzte Karte einmal nach rechts zu rotieren, an (-2,0) anzulegen und die Figur auf die Straße zu setzen. Dadurch hätte er ebenfalls zwei Straßenpunkte bekommen, mit seinem ersten Kloster nicht weniger Punkte gemacht, da die Alternativkoordinaten trotzdem neben dem Kloster liegen und vor allem die bessere Position für sein zweites Kloster nicht blockiert. Aufgrund der vielen Möglichkeiten, das Spiel zufällig zu Ende zu spielen, haben die Iterationen für den Flat-UCB-Spieler nicht dazu ausgereicht, diesen, kurzfristig genauso wertvollen, Zug dem anderen vorzuziehen.



Abb. 16: UCT (gelb) beginnt und spielt 19:19 Unentschieden gegen Flat-UCB (blau)

Die verkürzten Simulationen zeigen, dass der UCT-Algorithmus ab einer gewissen Anzahl an Iterationen pro Zug in der Lage ist, vorausschauender zu spielen und den Flat-UCB-Algorithmus zu besiegen. UCT schafft es mit den gegebenen Ressourcen und 600 Sekunden Laufzeit etwa drei Züge vorherzusehen und aufgrund dieser Informationen seinen nächsten Zug auszuwählen. Der Flat-UCB-Algorithmus geht stattdessen von einem Zufallsspiel nach seinem Zug aus, wodurch er nicht mit einem starken Gegner rechnet und nur wenig weitsichtig spielt. Mit den, dem Autor gegebenen, Hardwaremöglichkeiten ist es für den UCT-Spieler jedoch nicht möglich, die Anzahl an nötigen Iterationen in akzeptabler Laufzeit zu berechnen. Insbesondere funktioniert dies nicht, wenn alle Karten zum Spielen genutzt werden, da dadurch die Zufallsimulationen deutlich mehr Rechenzeit benötigen.

### 12.3 UCT gegen menschlichen Spieler

Schließlich wurde ein Spiel zwischen der UCT-KI und dem Autor gespielt, welcher ein stark fortgeschrittener Carcassonespieler ist. Der UCT-Algorithmus durfte für jeden Zug 150 Sekunden auf 4 Threads auf einer Intel(R) Core(TM) i7-3770 CPU rechnen. Damit schafft er zu Beginn des Spieles etwa 1160 Iterationen pro Thread. Die Gesamtzahl von 4640 Iterationen steht in kei-

nem guten Verhältnis zu den über 33000 Iterationen zu Beginn der verkürzten Simulation, wodurch vor Allem zu Beginn des Spieles nicht mit vorausschauendem Spiel zu rechnen war. Im Gegensatz dazu hat ein menschlicher Spieler der einsehen kann welche Karten er noch ziehen wird den Vorteil, dass er genau wissen kann, ob er noch Karten bekommen wird mit denen er ein großes Gebiet abschließen oder sich an ein gegnerisches Gebiet anbinden kann. Der Autor konnte so durch eigenes Blockieren eine gegnerische Blockade verhindern, da er wusste, dass er später eine Karte ziehen würde, die es ihm ermöglichte einen Ort im Wert von 24 Punkten abzuschließen.

Das Spiel endete 126:80 für den menschlichen Spieler. Sowohl der Spielverlauf, als auch ein Bild des finalen Spielaufbaus finden sich in dem entsprechenden Verzeichnis im Git-Repository.

Es fällt auf, dass die UCT-KI zu Beginn des Spiels sehrzeitig alle verfügbaren Meeples spielt. Da sie nicht weit vorausschauen kann und nicht alle Stellen schützen kann, verliert sie auch hier durch Blockaden des Gegners schnell Figuren, die sie dadurch erst spät oder sogar nie mehr zurück bekommt. Kurzfristig findet sie immer gute bis sehr gute Züge und schafft es dadurch die Vorhaben des Gegners zu sabotieren, oder sich an wertvolle Wiesen anzuschließen.

Durch das frühe Setzen von Figuren auf Wiesen und das zusätzliche Verlieren von blockierten Meeples, hatte die KI gegen Ende des Spiels nicht mehr viel Spielraum, die steigende Anzahl an Iterationen effektiv zu nutzen. Statt mehrere kleinere Orte abzuschließen, hatte sie nur die Möglichkeit, ihre Ortskarten an einen riesigen Ort anzuschließen, welcher nicht abgeschlossen werden konnte. Die KI verstand es jedoch sehr gut, ein einfaches Anschließen des menschlichen Spieler an diesen Ort zu verhindern. So kam die KI durch ihre Wiesen und die große Stadt letztendlich doch auf das akzeptable Ergebnis von 80 Punkten.

Erwähnenswert ist auch, dass die KI den zu Beginn von Abs. 12 erwähnten Fehler in der Spiellogik in einem ersten Versuchsspiel gegen den Autor erkannte und für sich nutzte. Dadurch bekam sie neben vielen Punkten auch einen zusätzlichen Meeple, was natürlich einen sehr wertvollen Vorteil darstellt.

## 12.4 Vergleiche mit Heyden

In Heyden [13] ist es nicht klar, wie mit dem Problem des Nichtdeterminismus umgegangen wird. Aufgrund eigener Versuche bezüglich des branching-factors und der ähnlichen Ergebnisse, wird davon ausgegangen, dass das Problem ähnlich behandelt wurde, wie in dieser Arbeit. Heyden vergleicht in ihrer Arbeit den UCT-Algorithmus mit der Standardevaluierungsmethode und den mit der auch in dieser Arbeit genutzten Alternativmethode (siehe Gl. (20)). Sie bestätigt die getroffene Annahme, dass die alternative Methode zu besseren Ergebnissen führt.

Sie schreibt, dass in ihren Simulationen jedem Spieler lediglich sechs Minuten Rechenzeit pro Spiel zur Verfügung gestellt wurden. Das würde bedeuten, dass der UCT-Spieler nur etwa 10 Sekunden zum Berechnen des nächsten Zuges hat. Wenn die Berechnungen nicht auf einem damaligen High-End Computer stattgefunden haben (die Arbeit stammt aus dem Jahr 2009), wird die Anzahl an berechneten Iterationen kaum für ein weitsichtiges Spiel gereicht haben.

Der Spieler mit der effektiveren Bewertungsmethode setzte in ihren simulierten Spielen durchschnittlich 11,4 Meeples pro Spiel. Aufgrund der längeren Laufzeit ist der Wert 12,6 in den Spielen gegen den Flat-UCB-Spieler deutlich höher (siehe Abb. 10). Trotz der Laufzeitunterschiede und den unterschiedlich starken Gegner-KIs, ähneln sich die Verteilungen der gesetzten Meeples auf die einzelnen Gebietsarten deutlich. Die hier genutzte UCT-Version setzte auf alle Gebietsarten durchschnittlich etwas mehr Figuren, aber die Verteilung ist sehr ähnlich.

Im Vergleich der durchschnittlich erspielten Punkte pro speziellem Gebietsmeeple fällt deutlich auf, dass Heydens UCT-Algorithmus mit 10,6 fast 4 Punkte mehr pro Wiesenmeeple er-

spielt hat, als die UCT-KI in den 50 hier simulierten Spielen. Diese kam durchschnittlich lediglich auf 6,8 Punkte (siehe Abb. 12). Ein möglicher Grund ist, dass der Flat-UCB-Gegner in dieser Arbeit im Verhältnis mehr in Wiesen investiert hat, als es der Standard-UCT-Algorithmus in Heydens Spielen getan hat. Es fällt nämlich auf, dass der Wert für den Flat-UCB-Spieler dem des UCT-Spielers in den simulierten Spielen sehr ähnlich ist und auch beide etwa gleich viele Meeples auf Wiesen gesetzt haben, während der Gegner von Heydens UCT-KI weniger Figuren auf Wiesen setzte und auch pro Wiesenmeeple deutlich weniger Punkte erspielte.

Heyden vergleicht die UCT-Ergebnisse außerdem mit denen eines Simple-MC-Spielers mit verbesserter Evaluierung. Jedoch stammen die Werte des MC-Spielers nicht aus einem direkten Vergleich mit dem UCT-Algorithmus, sondern aus Spielen gegen einen Simple-MC-Spieler mit Standardevaluierung. Aufgrund von möglichen Sabotageaktionen, oder Anschließungen und weiteren Faktoren ist die Spielstärke eines Algorithmus von der Stärke des Gegners abhängig. Daher ist dieser Vergleich nicht stark aussagekräftig und der von ihr beschriebene Spielvorteil von UCT gegenüber dem Simple-MC-Algorithmus dürfte allein durch den UCB-Auswahlmechanismus auf der ersten Child-Ebene entstanden sein.

Spannend sind in Heydens Arbeit die Vergleiche mit anderen Spielbaum-Algorithmen wie Expectimax, jedoch werden dort den Spielern auch jeweils nur sechs Minuten zum Berechnen aller Züge zur Verfügung gestellt, wodurch der UCT-Algorithmus wahrscheinlich nicht sein Potential nutzen konnte.

## 13 Fazit und Aussicht

In der Arbeit konnte die erfolgreiche Anwendung von Monte-Carlo-Methoden auf das Spiel Carcassonne gezeigt werden. Die getesteten Monte-Carlo-Algorithmen sind in der Lage, auf kurze Sicht sinnvolle Entscheidungen zu treffen. Es zeigte sich, dass die mit einer Bandit-Selektion arbeitende Flat-UCB-KI mit gleicher Laufzeit bessere Ergebnisse erzielt, als der Standard-MC-Ansatz.

Der UCT-Algorithmus schaffte es mit einer Laufzeit von 60 Sekunden gegen den Flat-UCB-Spieler mit gleicher Rechenzeit nicht, ausreichend Iterationen durchzuführen, um einen Vorteil aus dem Erkunden des Spielbaumes zu ziehen. Mangels ausreichender Hardware musste die erfolgreiche Anwendung des UCT-Algorithmus an einer determinisierten und verkürzten Spielversion gezeigt werden. Dabei wurde bestätigt, dass die UCT-KI am erfolgreichsten spielt und die besten Ergebnisse erzielt, wenn sie genügend Algorithmusiterationen durchführen kann. Mit etwa 35000 Iterationen ist es dem UCT-Spieler möglich, zwei bis drei Züge korrekt vorherzusehen und seine Aktion aufgrund dieser Informationen auszuwählen.

Aufgrund der kaum vorhandenen Weitsichtigkeit haben sowohl UCT bei kürzeren Laufzeiten als auch die übrigen KIs das Problem, dass sie früh viele Figuren platzierten und dadurch teilweise durch Blockaden nicht wiederbekommen. Mit der gegebenen Zeit ist daher es nicht möglich, die UCT-KI in Echtzeit so laufen zu lassen, dass sie in der Lage ist fortgeschrittene menschliche Spieler zu besiegen.

Es gibt viele Möglichkeiten, MCTS-Algorithmen effektiver und schneller zu machen [3]. Beispielsweise können verbesserte Bandit-Auswahlmethoden eingesetzt werden oder das Zufallsspiel durch Heuristiken gelenkt oder gar durch Neuronale Netze ersetzt werden [19, 20]. Es wäre spannend zu versuchen, ähnliche Ansätze für Carcassonne umzusetzen. Für eine weitere, einfache Optimierung der Auswahlkonstante  $c$  des UCT-Algorithmus, könnten Versionen unterschiedlicher Werte gegeneinander spielen und so die Version mit der höchsten Siegesrate ermittelt werden.

Außerdem wäre ein Vergleich mit anderen Algorithmen wie Expectimax oder ähnlichen interessant, bei denen den KIs mehr Rechenleistung zur Verfügung steht.

Da der erfolgreiche Einsatz von Monte-Carlo-Methoden auf Spielbäume gelungen ist, wäre es wichtig zu überlegen welche anderen Problemstellungen aus der Forschung in Entscheidungsprozesse umformulierbar sind und dadurch von MCTS-Algorithmen bearbeitet werden können.

## Literaturverzeichnis

- [1] V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, 1994.
- [2] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.*, 47(2):235–256, 2002.
- [3] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfschagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on computational intelligence and AI in games*, 4(1), Mar. 2012.
- [4] B. Brügmann. Monte Carlo Go. Technical report, Max-Planck-Institut of Physics, München, 1993.
- [5] G. M.-B. Chaslot, M. H. Winands, and H. J. van den Herik. Parallel Monte-Carlo Tree Search. Technical report, University Maastricht, Jan. 2008.
- [6] S. Chib and E. Greenberg. Understanding the Metropolis-Hastings Algorithm. *The American Statistician*, 49(4):327–335, Nov. 1995.
- [7] R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games, 5th International Conference, CG 2006, Turin, Italy*, 2006.
- [8] L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986.
- [9] A. Fischer and C. Igel. Training Restricted Boltzmann Machines: An Introduction. *Pattern Recognition*, 47:25–39, 2014.
- [10] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Technical report, INRIA, 2006.
- [11] J. M. Hammersley and D. C. Handscomb. *Monte Carlo Methods*. Methuen, London, 1964.
- [12] W. K. Hastings. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika*, 57(1):97–109, 1970.
- [13] C. Heyden. Implementing a Computer Player for Carcassonne. Master’s thesis, Maastricht University, Dept of Knowledge Engineering, 2009.
- [14] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo Planning. In *Machine Learning: ECML 2006*, 2006.
- [15] M. P. McLaughlin. Simulated Annealing: This algorithm may be one of the best solutions to the problem of combinatorial optimization. *Dr. Dobb’s Journal*, pages 26–37, 88–91, Sep 1989.
- [16] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.
- [17] M. Richey. The Evolution of Markov Chain Monte Carlo Methods. *The American Mathematical Monthly*, 117(5):383–413, May 2010.

- [18] C. Shannon. Programming a Computer for Playing Chess. *Philosophical Magazine*, 41(314), Mar. 1950.
- [19] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529:484–489, Jan 2016.
- [20] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550:354–359, 2017.
- [21] T. Vodopivec, S. Samothrakis, and B. Ster. On Monte Carlo Tree Search and Reinforcement Learning. *JAIR*, 60:881–936, Dec 2017.

## **Danksagung**

Ich danke Prof. Brügmann für die Betreuung dieser Arbeit und besonders für die hilfreichen Anregungen und die konstruktive Kritik.

Ich danke meinen Eltern und Mara Weberschock die auf der Suche nach Rechtschreibfehlern meine Arbeit korrigiert haben.

Im speziellen danke ich meinen Eltern und meiner lieben Freundin Katrin für die Unterstützung in allen nur denkbaren Aspekten die letzten Jahre und die Aufmunterungen, wenn wieder ein Fehler im Programm aufgetaucht ist. Ohne Sie wäre mein Studium und als Abschluss diese Arbeit niemals möglich gewesen.



## **Selbstständigkeit und Veröffentlichung**

Ich erkläre, die vorliegende Arbeit selbstständig verfasst, und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Vonseiten des Verfassers bestehen keinerlei Einwände, diese Arbeit der Thüringer Universitäts- und Landesbibliothek zur öffentlichen Nutzung zur Verfügung zu stellen.

Jena, den 9. Oktober 2019

---

Robert Kummer