Technology
Arts Sciences
TH Köln

Informatikprojekt

Untersuchung von KI Agenten im Spiel Othello

Erstellt von:
Julian Cöln
Studiengang: Allgemeine Informatik
Matrikelnummer: 11117175
und
Yannick Dittmar
Studiengang: Allgemeine Informatik
Matrikelnummer: 11117676

Datum der Abgabe: 04.12.2019

Betreuung: Prof. Dr. Wolfgang Konen

Technische Hochschule Köln
Fakultät für Informatik und Ingenieurswissenschaften

Inhaltsverzeichnis

1. Einleitung 4

2. Das Spiel Othello 4
2.1. Das Spiel Othello 4
2.2. Das General Board Game Framework 5
2.3. StateObservation 5
2.4. PlayAgent 5

3. Agenten 6
3.1. Benchmarkplayer 6
3.1.1. Implementierung im GBG 7
3.2. NTuple 9
3.2.1. Funktionsweise 9
3.3. Implementierung im GBG 9
3.4. NTuple-Netzwerke 11
3.5. Edax 12
3.5.1. Edax im GBG 12
3.5.2. CommandLineInteractor 13
3.5.3. CommandLineReader 13

4. Vergleiche der Agenten 15
4.1. Round-Robin-Tournament 15
4.2. n-Ply Option 15
4.3. Agenten vs. Benchmarkplayer 16
4.3.1. Parametrisierung 16
4.3.2. NTuple-Netzwerke-Lernerfolge 16
4.4. Agenten vs. Agenten 17
4.4.1. Parametrisierung 17
4.4.2. Vergleich mit 5.000 Trainingsiterationen 17
4.4.3. Vergleich mit 50.000 Trainingsiterationen 18
4.4.4. Vergleich mit 200.000 Trainingsiterationen 18
4.5. Vergleich gegen Edax 19

5. Fazit 20

A. Anhang 22
A.1. Designvorgang der NTuple 22
A.2. Bestimmung der Parametrisierung 22

Abbildungsverzeichnis

1. Verschiedene Spielzustände zur Erläuterung der Spielregeln 5
2. a) Heur Spieler, b) Bench Spieler [vdRW13, S. 5] 6
3. Jaskowski NTuple-Netzwerk [Jaś14, S. 88, Abb.1 modifizert] 11
4. selbsterstellte NTuple-Netzwerke mit verschiedenen NTuple-Größen . . . 12
6. Ergebnisse der NTuple-Netzwerke bei wenigen Trainingsiterationen . . . 17
7. Ergebnisse der NTuple-Netzwerke mit höheren Trainingsiterationen . . . 19

Tabellenverzeichnis

1. Parametrisierung im GBG 16
2. Parametrisierung im GBG 17
3. Ergebnis Round-Robin-Tournament 5.000 Trainingsiterationen 18
4. Ergebnis Round-Robin-Tournament 50.000 Trainingsiterationen 18
5. Ergebnis Round-Robin-Tournament 200.000 Trainingsiterationen 19

1. Einleitung

Manche intelligente Agenten können Spielzustände mindestens genauso gut wie Menschen evaluieren und entscheiden, ob diese noch zu gewinnen sind oder ob eine Niederlage unmittelbar bevor steht. Dabei sind heutige starke Agenten in der Lage, bestmögliche Spielzüge wesentlich schneller als ein Mensch zu erkennen und dem Gegner sogar Fallen zu stellen.
Erste große Erfolge im Brettspiel Othello konnte der Agent Logistello im Jahre 1997 erreichen, als er den heute mehrfachen Weltmeister Takeshi Murakami in einer Serie von 6 Spielen zu 0 schlagen konnte [Bur97]. Ab diesem Zeitpunkt konnte vermutet werden, dass intelligente Agenten nicht nur in diesem Spiel die Oberhand gewinnen, sondern in allen strategischen Brettspielen mehr oder weniger erfolgreich anwendbar sind. Alpha Go, ein von Google DeepMind entwickelter Agent für das Brettspiel Go, schlug im Jahr 2016 den Profispieler Lee Sedol 4 zu 1 [Bor16].
Diese Arbeit untersucht verschiedene Agenten, die auf das Brettspiel Othello angewandt werden. Das Hauptaugenmerk liegt auf der Entwicklung eines selbstlernenden Agenten, der nach vergleichsweise wenigen Trainingsiterationen besonders stark spielt.
Dabei stellen sich folgende Fragen:

- Kann ein Agent erstellt und erfolgreich trainiert werden, sodass dieser über die Spielweise eines einfachen statischen Agenten hinausreicht?
- Besteht die Möglichkeit, diese Agenten untereinander zu vergleichen um die besten zu Spieler zu finden?
- Sind die trainierten Agenten in der Lage, den besonders stark spielenden Agenten „Edax" zu schlagen?

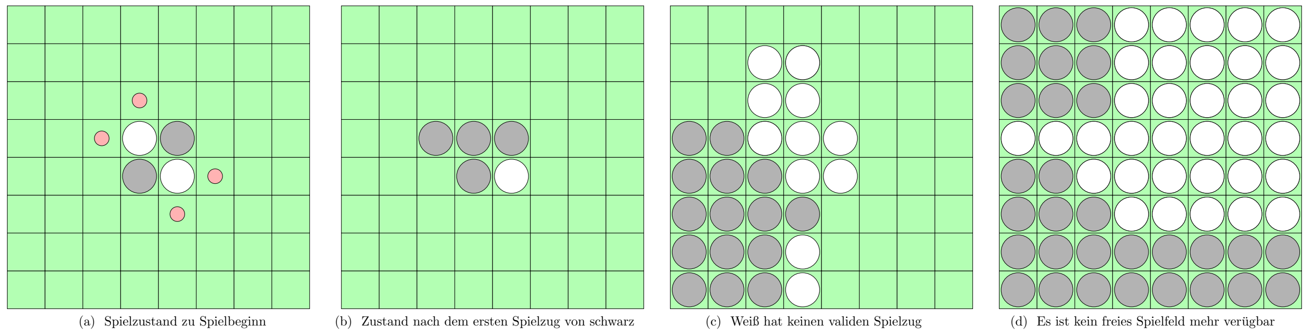
2. Das Spiel Othello

Das folgende Kapitel beschreibt den Spielaufbau und die Spielregeln von Othello. Des Weiteren erfolgt eine Einführung in das verwendete Framework "General Board Game", welches zur Implementierung des Spiels genutzt wird.

2.1. Das Spiel Othello

Othello ist ein deterministisches, rundenbasiertes Strategiespiel für zwei Spieler. Das Spielbrett besteht aus 64 Spielfeldern, die jeweils 8 Felder in der Horizontalen und Vertikalen zählen. Ziel des Spiels ist, nach Ende der Platzierung des letzten möglichen Spielsteines die höhere Anzahl an Spielsteinen auf dem Spielbrett zu besitzen. Der Startzustand S_0 ist durch vier platzierte Steine auf den 4 mittleren Spielfeldern gegeben. Dabei müssen die gleichen Farben jeweils diagonal zueinander gelegt werden. Mögliche Spielzüge des ersten Zuges für den Spieler mit den schwarzen Spielsteinen bildet ein roter Punkt in der Mitte der benachbarten Spielfelder ab (1a). Jeder Spielstein muss so platziert werden, dass er mit einem weiteren eigenen Stein mindestens einen gegnerischen

Spielstein einschließt. Ausgehend von einem Spielzug, ändern alle neu eingeschlossenen gegnerischen Spielsteine die Farbe zu der des zuletzt gelegten Spielsteins (1b). Ist für einen Spieler kein gegnerischer Spielstein einzuschließen, so darf in dieser Runde kein weiterer Spielstein platziert werden und der Gegner ist erneut an der Reihe (1c). Sobald beide Spieler keine gültigen Züge mehr haben oder kein freies Feld mehr verfügbar ist, endet die Partie (1d).



(a) Spielzustand zu Spielbeginn (b) Zustand nach dem ersten Spielzug von schwarz (c) Weiß hat keinen validen Spielzug (d) Es ist kein freies Spielfeld mehr verügbar

Abbildung 1: Verschiedene Spielzustände zur Erläuterung der Spielregeln

2.2. Das General Board Game Framework

Das General Board Game Framework, kurz ,,GBG", ist ein von Wolfgang Konen entwickeltes Framework, um künstliche Intelligenz im Anwendungsbereich der Brettspiele zu erforschen [Kon17]. Der Umfang des GBG beinhaltet Spiele wie ,,2048", ,,Tic-Tac-Toe" und ,,Vier Gewinnt". Um ein Spiel dem GBG hinzuzufügen sind folgende Klassen und Interfaces besonders wichtig:

2.3. StateObservation

Eine *StateObservation* stellt den aktuellen Spielzustand dar. Es beinhaltet eine Darstellung des Spielfeldes mit allen Spielsteinen und kann mit der Methode *getAvailableActions* alle verfügbaren Züge bestimmen. Des Weiteren kann es Spielzüge mit *advance* simulieren oder ausführen und mit Hilfe der *getScore*-Methode Spielzustände bewerten.[Kon17]

2.4. PlayAgent

Das *PlayAgent*-Interface beinhaltet die Funktionalitäten für einen Agenten. Die wichtigste Methode ist die *getNextAction2*-Methode. Diese erwartet einen Spielzustand als *StateObservation* und gibt den bestmöglichen Spielzug zurück. [Kon17]

3. Agenten

Dieses Kapitel befasst sich mit den untersuchten Agenten und deren Funktionsweisen sowie der Implementierung im GBG. Besondere Aufmerksamkeit wird dem Benchmarkplayer, dem NTuple-Agenten und dem Edax-Programm gewidmet.

3.1. Benchmarkplayer

Der Benchmarkplayer [vdRW13] wird zur Evaluation des Trainingsfortschritts benutzt, um zu überprüfen, ob der lernende Agent eine Strategie entwickelt hat, die einen Gegner mit mittlerem Spielniveau schlagen kann. Der Agent erhält einen Spielzustand und simuliert jeden möglichen Zug einmal. Jeder der daraus resultierende Spielzustand wird durch die Evaluierungsfunktion anhand der Wertetabellen 2a oder 2b ausgewertet:

$$V = \sum_{i=1}^{64} c_i w_i \tag{3.1}$$

Der Parameter c_i steht für den Spielstein, wobei er den Wert $+1$ für den Agenten selbst, -1 für einen gegnerischen Spielstein und 0 für den Fall, dass dieses Spielfeld noch von keinem Spieler belegt ist, annimmt. Der Index gibt die Position auf dem Spielbrett an. Der zweite Parameter w_i steht für das zugeordnete Gewicht aus einer der beiden Wertetabellen. V ist der interne Punktewert des Spielzustandes für den Agenten. Bei eindeutigen Spielzuständen wird dieser Agent immer die gleichen Entscheidungen treffen. Stehen jedoch mehrere Züge mit der gleichen internen Punktewertung zur Auswahl, wird ein Zug zufällig ausgewählt.

100	-25	10	5	5	10	-25	100
-25	-25	2	2	2	2	-25	-25
10	2	5	1	1	5	2	10
5	2	1	2	2	1	2	5
5	2	1	2	2	1	2	5
10	2	5	1	1	5	2	10
-25	-25	2	2	2	2	-25	-25
100	-25	10	5	5	10	-25	100

(a) Heurplayer

80	-26	24	-1	-5	28	-18	76
-23	-39	-18	-9	-6	-8	-39	-1
46	-16	4	1	-3	6	-20	52
-13	-5	2	-1	4	3	-12	-2
-5	-6	1	-2	-3	0	-9	-5
48	-13	12	5	0	5	-24	41
-27	-53	-11	-1	-11	-16	-58	-15
87	-25	27	-1	5	36	-3	100

(b) Benchplayer

Abbildung 2: a) Heur Spieler, b) Bench Spieler [vdRW13, S. 5]

3.1.1. Implementierung im GBG

Der Benchmarkplayer erweitert die Klasse *AgentBase*, welche die Stammfunktionalitäten des zu erstellenden Agenten bereithält. Zusätzlich werden noch die beiden Interfaces *PlayAgent* und *Serializable* implementiert. Durch Letzteres ist es möglich den Agenten zu exportieren und zu importieren und dadurch in einem Turnier mitwirken zu lassen. Der Benchmarkplayer benötigt die zwei Kernmethoden *getNextAction2* und *evaluateState*, welche im Fall des Benchmarkplayers eine Funktionalität der *getNextAction2*-Methode kapselt. Die *getNextAction2*-Methode erhält eine spielspezifische *StateObservation* und wählt dann anhand der oben genannten Evaluierungsfunktion den bestmöglichen Spielzug für den übergebenen Spielzustand aus. Eine detaillierte Ansicht der Funktionsweise ist dem folgenden Pseudocode zu entnehmen, der eine genauere Implementierung beider Methoden vorstellt.

Algorithm 1: getNextAction2 bestimmt die bestmögliche Aktion für den Agenten, die für den aktuellen Spielzustand wählbar ist.

```
1  function getNextAction2 (so)
   Input  : StateObservation so
2
   Output: Action bestAction
3
4  bestAction ⟵ null
5  action ⟵ null
6  newStateobservation ⟵ null
7  bestScore ⟵ −∞
8  score ⟵ null
9  actions[] ⟵ so.getAvailableActions()
10 count ⟵ 0
11 foreach action ∈ actions do
12 |   newStateobservation ⟵ stateobservation.copy()
13 |   score ⟵ evaluate(newSo, action)
14 |   if score > bestScore then
15 |   |   bestscore ⟵ score
16 |   |   bestaction ⟵ action
17 |   |   count ⟵ 1
18 |   else if score == bestScore then
19 |   |   count ⟵ count + 1
20 |   |   if random.nextDouble() < (1.0/count) then
21 |   |   |   bestscore ⟵ score
22 |   |   |   bestAction ⟵ action
23 end
24 return bestAction
```

Algorithm 2: evaluateState wertet den bereits erweiterten Spielzustand durch eine Wertetabelle aus

```
1  function evaluateState (so, action) Input : StateObservation so, Action action
2  ; Output: Double score
3  ; currentScore ⟵ null
4  player ⟵ so
5  so.advance(action)
6  gameState[][] = so.getCurrentGameState
7  newSo ⟵ null
8  bestScore ⟵ −∞
9  score ⟵ null
10 actions[] ⟵ so.getAvailableActions()
11 count ⟵ 0
12 foreach action ∈ actions do
13 |   newSo ⟵ so.copy()
14 |   score ⟵ evaluateState(newSo, action)
15 |   if score > bestScore then
16 |   |   bestscore ⟵ score
17 |   |   bestaction ⟵ action
18 |   |   count ⟵ 1
19 |   else if score == bestScore then
20 |   |   count ⟵ count + 1
21 |   |   if random.nextDouble() < (1.0/count) then
22 |   |   |   bestscore ⟵ score
23 |   |   |   bestAction ⟵ action
24 end
25 return bestAction
```

3.2. NTuple

Neben dem Benchmarkagenten sind im Umfang des GBGs mehrere weitere Agententypen enthalten. Zum Einen gibt es den naiven ,,Random"-Agenten, der zufällig einen erlaubten Zug auswählt und spielt. Zum Anderen gibt es Agenten wie den ,,MaxN"-Agenten, der eine klassische Implementierung des Minimax Algorithmus darstellt oder den ,,Monte-Carlo-Tree-Search"-Agenten, der die Präzision der Baumsuche mit der Allgemeingültigkeit von zufälligen Stichproben kombiniert [BPW+12]. Die dritte Art von Agenten sind die ,,Temporal-Difference"-Agenten, welche durch reinforcement learning (bestärkendes Lernen) nach einigen Trainingseinheiten starke Ergebnisse erzielen können. Einer dieser Agenten ist der ,,NTuple"-Agent, welcher im Folgenden näher beschrieben wird.

3.2.1. Funktionsweise

Das Konzept der NTuple wurde ursprünglich in den 1950er Jahren von Woody Bledsoe und Iben Browning entwickelt, um es Maschinen zu ermöglichen Muster wie Buchstaben zu erkennen und zu kategorisieren [BB59]. Simon Lucas hat mit diesem Algorithmus in Kombination mit Temporal Difference Learning einen Othello Agenten erstellt, welcher nach 500 Trainingseinheiten in der Lage ist, die Benchmarkplayer zu schlagen.[Luc08]
Ein NTuple ist eine Sammlung von n Punkten auf dem Spielbrett, jeder Punkt liegt auf einem Spielfeld (3a). Diese können entweder durch den Entwickler vordefiniert sein oder durch zufälliges ,,Laufen" von Feld zu Feld prozedual zur Laufzeit erstellt werden (3d). Jedes NTupel wird acht mal auf dem symmetrischen Spielbrett platziert, wobei das Brett dabei durch Rotationen und Spiegelungen in die symmetrischen Zustände gebracht wird. Durch das Ausnutzen der Spielbrettsymmetrie kann die Effizienz aller NTuple gesteigert werden [Jaś14].
Mit Hilfe der so erzeugten Netzwerke kann das aktuelle Brett bewertet werden. Dazu wird der Wert jedes NTuple in einer Lookup-Tabelle ausgelesen und summiert. Die Lookup-Tabelle beinhaltet dabei alle (3^n) möglichen Kombinationen der NTuple, für jedes Feld in der Kette mit jeder Feldfarbe (schwarz, weiß und farblos) eins. Die Summe aller Bewertungen der NTupel ergeben den Gesamtwert eines Spielbretts und mit diesem kann der Agent dann über Temporal Difference Learning die Lookup-Tabelle anpassen und so aus dem aktuellen Zug lernen.

3.3. Implementierung im GBG

Das GBG bietet selbst eine Implementierung für NTuple. Um einen NTuple-Agenten anzulegen, muss dieser die Interfaces *XNTupleFuncs* und *Serializable* implementieren, wobei das Serializable Interface wie beim Benchmarkplayer zum Importieren und Exportieren für Turniere benötigt wird. Das Interface XNTupleFuncs definiert die folgenden zwölf Methoden:

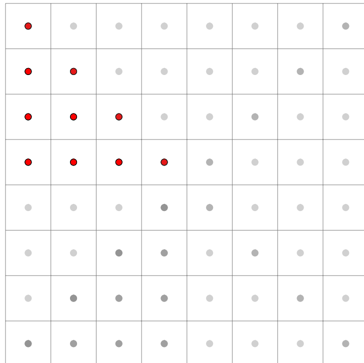
- *boolean instantiateAfterLoading();*

- *int[] makeBoardVectorEachCellDifferent();*
- *int getNumCells();*
- *int getNumPositionValues();*
- *int getNumPlayers();*
- *int[] getBoardVector(StateObservation so);*
- *int[][] symmetryVectors(int[] boardVector);*
- *int[] symmetryActions(int actionKey);*
- *int[][] fixedNTuples(int mode);*
- *String fixedTooltipString();*
- *int[] getAvailFixedNTupleModes();*
- *HashSet adjacencySet(int iCell);*

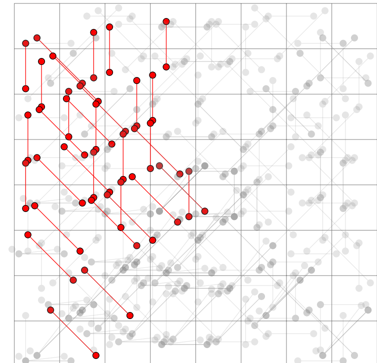
Die Basisklasse *XNTupleBase* implementiert bereits *instantiateAfterLoading* und *makeBoardVectorEachCellDifferent*, der Agent kann XNTupleBase erweitern, um auf die Methoden zuzugreifen. Die Methoden *getNumCells, getNumPositionValues, getNumPlayers* geben Auskunft zu Spielkonstanten, *getNumCells* gibt die Anzahl der Spielfelder zurück, bei Othello sind es immer 64 Felder. *getNumPlayers* gibt die Anzahl der Spielparteien des Spiels zurück, in diesem Fall zwei. Die Methode *getNumPositionValues* gibt die Anzahl der unterschiedlichen Feldzustände zurück. Da ein Feld entweder schwarz, weiß oder farblos sein kann ist der Rückgabewert drei. Denkbar ist auch eine Logik mit zwei farblosen Zuständen, mit der Unterscheidung, ob das Feld in diesem Zug erreichbar ist oder nicht. Für Othello hat sich diese Unterscheidung als nicht relevant herausgestellt, da alle erlaubten Felder durch eine Methode aufgerufen werden können. Die Interfacemethoden *getBoardVector, symmetryVectors, symmetryActions* beschreiben das aktuelle Spielfeld. *getBoardVector* wandelt den Spielzustand aus der übergebenen *StateObservation* in ein Integer Array um. Dieser Spielfeldvektor wird von *symmetryVectors* in symmetrische Vektoren umgewandelt. Um alle acht symmetrischen Vektoren zu erhalten muss das Spielfeld drei mal gedreht werden, um dann alle resultierenden Vektoren noch zu spiegeln. Zum Schluss müssen noch die Methoden *fixedNTuples, fixedTooltipString, getAvailFixedNTupleModes, adjacencySet* bereitgestellt werden. Bei den Methoden *fixedNTuples, fixedTooltipString, getAvailFixedNTupleModes* handelt es sich um Methoden, die es dem Entwickler ermöglicht vordefinierte NTuple-Netzwerke zu programmieren. *fixedNTuples* gibt dabei ein Netzwerk aus, abhängig von der Nutzerauswahl aus den verschiedenen Modi von *getAvailFixedNTupleModes*. Die Methode *adjacencySet* gibt alle Nachbarn einer Zelle zuück, was für die zufällige Erzeugung von NTuplen durch ,,laufen" über das Spielfeld verwendet wird.

3.4. NTuple-Netzwerke

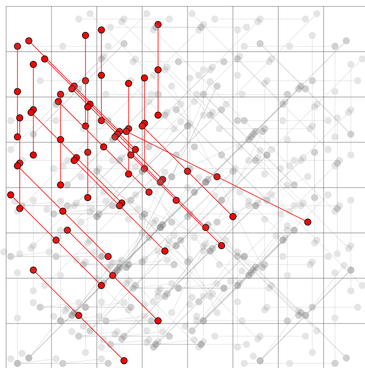
Die nachfolgenden Grafiken zeigen verschiedene NTuple-Netzwerke. Die durch Rotation und Spiegelung erweiterten Bordpositionen der Tuple werden in Grau dargestellt.



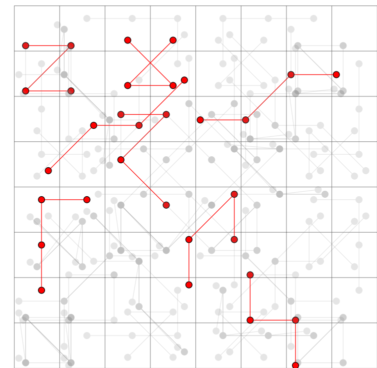
(a) 1er-Tupel-Netzwerk (30 Gewichte)



(b) 2er-Tupel-Netzwerk (288 Gewichte)

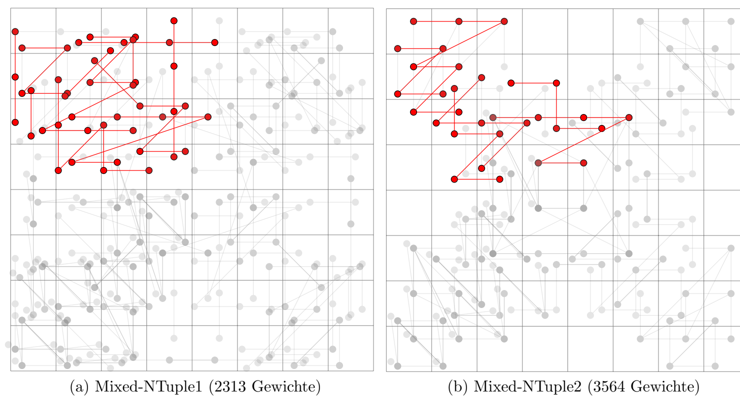


(c) 3er-Tupel-Netzwerk (648 Gewichte)



(d) 4er-Tupel-Netzwerk, zufällig generiert (648 Gewichte)

Abbildung 3: Jaskowski NTuple-Netzwerk [Jaś14, S. 88, Abb.1 modifizert]



(a) Mixed-NTuple1 (2313 Gewichte) (b) Mixed-NTuple2 (3564 Gewichte)

Abbildung 4: selbsterstellte NTuple-Netzwerke mit verschiedenen NTuple-Größen ¡¡¡¡¡¡¡ HEAD

Die Bordpositionen der Tuple der NTuple-Netzwerke Mixed-NTuple1 und Mixed-NTuple2 sind eigen ausgewählt. Der Unterschied besteht darin, dass Mixed-NTuple2 nur 6 Tuple mit einer Länge ≥ 4 beinhaltet, die sich auf 2x4er, 2x5er, 1x6er und 1x7er Tuple aufteilen. Mixed-NTuple1 hingegen beinhaltet die folgenden 11 NTuple Längen 2x2er, 1x3er, 4x4er, 2x5er, 2x6er. =======

¿¿¿¿¿¿¿ 34a3fda8cff4695a0a9c2fdbdd2077e3c5406457

3.5. Edax

Edax ist ein sehr starkes Othello Programm entwickelt von Richard Delorme in der Programmiersprache C. An das GBG-Framework angebunden, wurde die Windows-Release-Version 4.4. Die hervorstechenden Eigenschaften sind, wie vom Author selbst beschrieben:
,,Edax is a very strong othello program. Its main features are:

- fast bitboard based and multithreaded engine.
- accurate midgame-evaluation function.
- opening book learning capability.
- text based rich interface.
- multi-protocol support to connect to graphical interfaces or play on Internet (GGS).
- multi-OS support to run under MS-Windows, Linux and Mac OS X." [Del17]

3.5.1. Edax im GBG

Edax wird im GBG wie ein normaler Agent behandelt. Da es eine eigenständige Konsolenanwendung ist, implementiert die *getNextAction2*-Methode nicht wie der Benchmarkplayer die Agentenlogik. Stattdessen simuliert der Agent ein Spiel gegen das Konsolenprogramm Edax, indem der GBG-Agent den Zug seines Gegners gegen Edax spielt

und dessen Antwort als den eigenen Zug an das GBG zurück meldet.
Um die Funktionsweise des Agenten zu erklären, muss auf die Funktionsweise der Edaxanwendung eingegangen werden: Edax bietet vier verschiedene Spielmodi an, die über den Befehl *mode* $< 0 - 3 >$ eingestellt werden können. In den Modi ,,0" und ,,1" spielt Edax automatisch auf eine menschliche Eingabe, mit dem einzigen Unterschied, dass einmal der Spieler und einmal der Computer anfängt. Im Modus ,,2" spielt Edax auf beiden Seiten automatisch und im Modus ,,3" spielt Edax gar nicht automatisch. Mit dem Befehl *go* kann ein Zug von Edax angefordert werden. Dieses Feature wird für den Edax-Agenten benutzt (5). Des Weiteren kann mit den Befehlen *level* $< level >$ und *move-time* $< time >$ die Suchtiefe von Edax und die Zeit pro Zug für Edax eingestellt werden.

3.5.2. CommandLineInteractor

Um eine Kommunikation zwischen dem GBG und der Konsolenanwendung zu ermöglichen, wird ein *CommandLineInteractor* verwendet. Der *CommandLineInteractor* ist in der Lage, eine Konsolenanwendung auszuführen, Befehle an die Kommandozeile zu schicken und die Antwort darauf zurück zu geben. Zur Initialisierung benötigt der *CommandLineInteractor* den Pfad zur Anwendung, den Namen der Anwendung und optional eine Regular Expression um die Antwort zu spezifizieren. Zusätzlich gibt es die Methoden *sendCommand* und *sendAndAwait*. Die *sendCommand*-Methode wird zum Initialisieren von Edax verwendet, um den Spielmodus festzulegen und die Schwierigkeit einzustellen. *sendAndAwait* wird zur Zugabfrage verwendet, da der Agent einen Zug als Antwort erwartet.

3.5.3. CommandLineReader

Die Antwort einer Konsolenanwendung wird wiederum im *CommandLineReader* realisiert. Nach einem *sendAndAwait* Aufruf wird die Ausgabe auf der Konsole gelesen und gegebenenfalls mit der zuvor definierten Regular Expression gekürzt an den CommandLineIteractor weitergegeben.
Dieses Ergebnis wandelt der GBG-Agent in einen gültigen Zug um und spielt diesen.

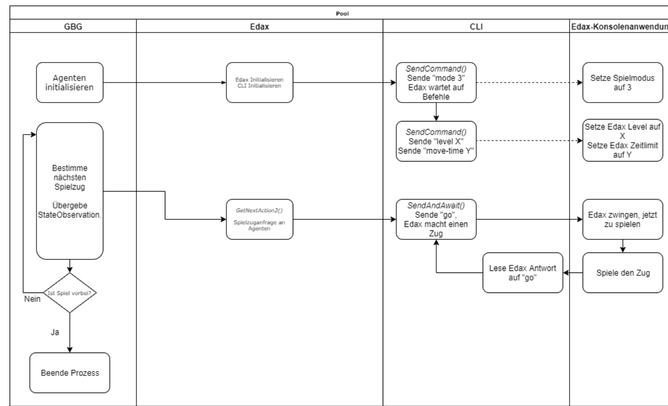


Abbildung 5: Lebenszyklus für den angebundenen Edax2-Agenten

4. Vergleiche der Agenten

Die im vorherigen Kapitel vorgestellten NTuple werden durch das GBG-Framework [Kon17] angewendet und anschließend in einem Round-Robin-System getestet. Jeder Agent trainiert verschiedene Anzahlen an Iterationen gegen sich selbst und tritt dann in seiner Gewichtsklasse gegen die anderen Agenten an. Nach der Bewertung der Agenten werden die zwei besten mit dem Edax-Agenten verglichen.

4.1. Round-Robin-Tournament

Das Round-Robin-Tournament ist ein Ligasystem und bietet einen Maßstab, um verschiedene Agenten unterschiedlichster Stärken gegeneinander antreten zu lassen, um den Agenten mit dem höchsten Spielverständnis herauszufiltern. Jeder Agent tritt gleich oft gegen alle anderen Agenten an, wobei darauf geachtet wird, dass jeder die selbe Anzahl an Eröffnungszügen spielt. Eine Agentenpaarung besteht aus insgesamt 1000 Spielen, damit der zufällige Gewinn eines eigentlich schwächeren Agenten als Ausreißer nicht zu sehr in das Ergebnis mit einfließt.

4.2. n-Ply Option

Die n-Ply Option bestimmt die Suchtiefe, dazu wird der aktuelle Spielzustand s_0 um jeden möglichen Spielzug erweitert bis s_n erreicht ist. Der Spielzug, der zu dem höchsten internen Punktewert des Agenten führt wird anschließend ausgewählt. Der nächste Zug baut diesen Suchbaum wieder aus. Das bedeutet, dass nicht eine einmalig festgelegte Reihenfolge an Spielzügen strikt eingehalten wird, sondern diese nach Verhalten des Gegeners angepasst werden. Im GBG ist diese Option als Wrapper für jeden Agenten verfügbar.

4.3. Agenten vs. Benchmarkplayer

Dieses Unterkapitel untersucht die Lernerfolge der NTuple-Netzwerke gegen die Benchmarkplayer. Jaskowski zeigt, dass die gezeigten NTuple bereits nach wenigen hundert Trainingsiterationen durch evolutionäre Optimierung stark gegen die heuristischen Agenten spielen[Jaś14]. Es soll gezeigt werden, dass ebenfalls gute Ergebnisse durch bestärkendes Lernen erreicht werden können.

4.3.1. Parametrisierung

Parameter	Wert	Parameter	Wert
Alpha init	0.5	Epsilon init	0.1
Alpha final	0.5	Epsilon final	0.0
Lambda	0.0	Gamma	1.0
Output Sigmoid	true	numEval	25

Tabelle 1: Parametrisierung im GBG

4.3.2. NTuple-Netzwerke-Lernerfolge

Die nachfolgende Grafik zeigt den prozentualen Lernerfolg der NTuple-Netzwerke nach 250, 500, 750 und 1000 Trainingsspielen. Die rote Linie zeigt das 2er-NTuple-Netzwerk (1b), welches bei diesem Vergleich mit einer Gewinnrate von 97.7% am besten abschneidet. Auf dem zweiten Platz liegt das Mixed-NTuple1-Netzwerk (4a) mit 96.15% in blau. Die grüne Linie zeigt den Lernerfolg des 3er-NTuple-Netzwerks und Mixed-NTuple2-Netzwerk (4b) ist in gelb abgebildet. Das Mixed-NTuple2-Netzwerk (gelb) besitzt wie in der Abbildung gezeigt nur NTuple mit einer Länge von ≥ 4 Bordpositionen. Der Verlauf der gelben Kurve ist ein Indikator dafür, dass dieses Tuple-Netzwerk mit nur wenigen Trainingsiterationen wesentlich schlechter lernt als beispielsweise das 2er-NTuple-Netzwerk.

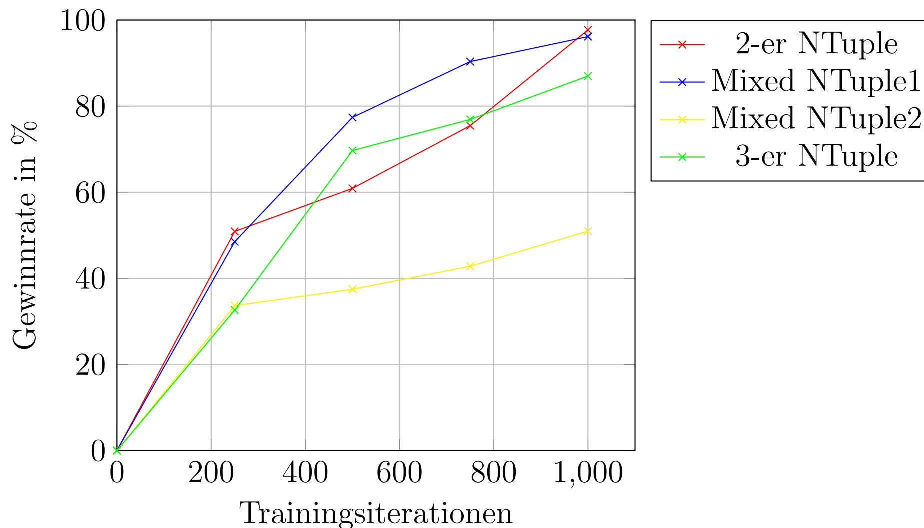


Abbildung 6: Ergebnisse der NTuple-Netzwerke bei wenigen Trainingsiterationen

4.4. Agenten vs. Agenten

Dieses Unterkapitel untersucht, wie Agenten in einem Double-Round-Robin-Tournament von 1000 Spielen untereinander spielen.

4.4.1. Parametrisierung

Damit alle Agenten in der gleichen Testumgebung trainieren und eine Gleichwertigkeit gegeben ist, wurden alle Agenten mit den folgenden Parametern aus der nachstehenden Tabelle trainiert.

Parameter	Wert	Parameter	Wert
Alpha init	0.2	Epsilon init	0.05
Alpha final	0.2	Epsilon init	0.0
Lambda	0.4	Gamma	1.0
Output Sigmoid	true	numEval	1000

Tabelle 2: Parametrisierung im GBG

4.4.2. Vergleich mit 5.000 Trainingsiterationen

Die folgenden Tabelle veranschaulicht den Vergleich der verschiedenen trainierten NTuple-Agenten untereinander mit dem Heur- und Bench-Spieler. Dabei ist zu sehen, dass die das 2er-NTuple-Netzwerk mit wenigen Trainingiterationen mit einer Gewinnrate von 88.42% am stärksten spielt.

Agent	Gewonnen	Unentschieden	Verloren	Gewinnrate in %
2er-NTuple	10367	486	1147	88.42
mixed-NTuple1	7955	26	4019	66.40
3er-NTuple	7428	11	4561	61.95
rand-NTuple	5984	3	6013	49.88
mixed-NTuple2	3991	500	7509	35.34
Heur-Spieler	2952	111	8937	25.06
Bench-Spieler	2733	43	9224	22.95

Tabelle 3: Ergebnis Round-Robin-Tournament 5.000 Trainingsiterationen

4.4.3. Vergleich mit 50.000 Trainingsiterationen

In dem zweiten Vergleich ist die Tendenz zu erkennen, dass jetzt die größeren Netzwerke die oberen Plätze einnehmen. Die Differenz der Gewinnrate zwischen dem ersten und zweiten Platz mit 9,16%.

Agent	Gewonnen	Unentschieden	Verloren	Gewinnrate in %
Mixed-NTuple1	9778	25	2197	81.59
Mixed-NTuple2	8674	34	3292	72.43
2er-NTuple	8288	0	3712	69.07
3er-NTuple	6533	0	5467	54,44
Rand-NTuple	3186	105	8709	26,99
Bench-Spieler	2677	115	9208	22,79
Heur-Spieler	2654	141	9205	22,70

Tabelle 4: Ergebnis Round-Robin-Tournament 50.000 Trainingsiterationen

4.4.4. Vergleich mit 200.000 Trainingsiterationen

Im Gegensatz zu den vorherigen Vergleichen mit deutlich weniger Iterationen, ist der Vorsprung der Gewinnrate der größeren selbsterstellten NTuple-Netzwerk Mixed-NTuple1 und Mixed-NTuple2 mit mehr 10.09% und 8.02 % deutlich. Die Benchmarkspieler haben die ungefähr selbe Gewinnrate und belegen wie in allen anderen Vergleichen auch hier die letzten Plätze, dabei sinkt die Gewinnrate der Benchmarkspieler immer weiter.

Agent	Gewonnen	Unentschieden	Verloren	Gewinnrate in %
Mixed-NTuple1	8360	145	3495	70.27
Mixed-NTuple2	8184	0	3816	68.2
3er-NTuple	7206	30	4764	60.18
2er-NTuple	6754	0	5246	56.28
rand-Ntuple	6201	17	5782	51.75
Bench-Spieler	2853	36	9381	21.67
Heur-Spieler	2484	228	9288	21.65

Tabelle 5: Ergebnis Round-Robin-Tournament 200.000 Trainingsiterationen

Alle drei Vergleiche der Agenten zeigen, dass die erstellten NTupel-Netzwerke Mixed-Ntuple1 und Mixed-NTuple2 stärker spielen als die drei anderen wesentlich kleineren, sobald diese eine höhere Anzahl an Iterationen ausgesetzt werden. Durch mehrere Iterationen wird also eine Strategie gewählt, die den kleineren überlegen ist.

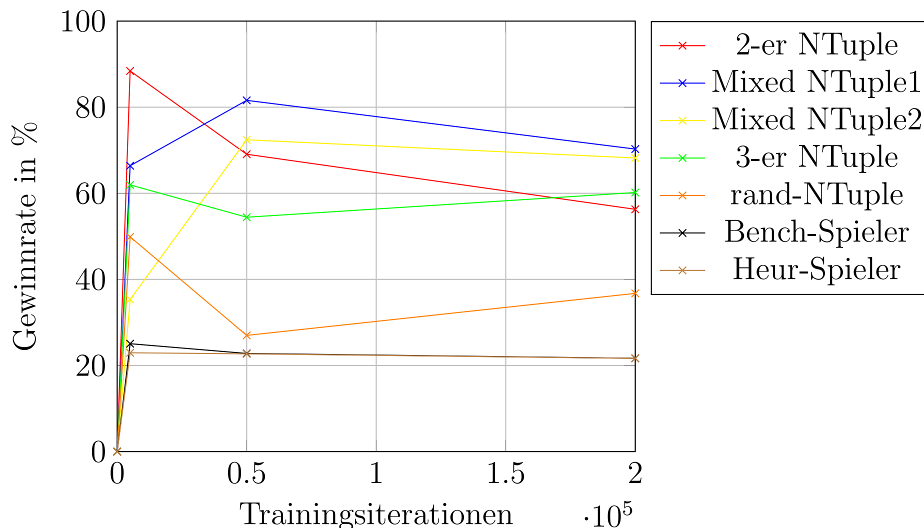


Abbildung 7: Ergebnisse der NTuple-Netzwerke mit höheren Trainingsiterationen

Die Abbildung zeigt, dass das stark spielende 2-er-NTuple Netzwerk bereits vor 50.000 Trainingsiterationen den Mixed-NTuple Netzwerken unterlegen ist.

4.5. Vergleich gegen Edax

Unser bester Agent aus dem Vergleich mit 200.000 Trainingsiterationen ist in der Lage den Edax-Agenten mit Suchtiefe 0 oder 1 als weißer Spieler ohne die Option n-Ply auszuwählen zu schlagen. Leider ist Edax ab einer Suchtiefe von 2 auch mit einer höheren n-Ply Option ($n \leq 4$) des getesteten Agenten nicht zu schlagen.

5. Fazit

Das Ziel dieser Arbeit war mit Hilfe des General Board Game Frameworks das Konzept der selbstlernenden Agenten im Kontext von Othello zu lernen, eigene Agenten dafür zu trainieren und diese miteinander zu vergleichen. Die Ergebnisse zeigen, dass sich NTuple Agenten dazu eignen Othello zu spielen. Durch den verallgemeinerten NTuple-Agenten des GBG-Frameworks war es möglich, Tuple zu finden, die das Spielniveau der statischen Benchmarkagenten übertreffen. Dabei hat sich gezeigt, dass die in dieser Arbeit trainierten Agenten unter beschriebener Parametrisierung, gegen die von Jaskowski beschriebene NTuple-Agenten im Ligasystem besser abschneiden. Das Ranking der verglichenen Agenten zeigt, dass sich die vorgestellten Benchmark-Agenten bei beiden vergleichen auf den letzten Plätzen befinden. Besonders erfreulich ist es, dass der stärkste NTuple-Agent ohne Suchtiefe in der Lage ist Edax mit einer Suchtiefe von einem Zug zu schlagen, solange er als zweiter Spieler reagiert. Das bedeutet, dass Edax mit maximaler Suchtiefe aktuell nicht zu besiegen ist.Jasokowski zeigt, dass das 2er-NTuple-Netzwerk bereits nach 1250 Trainingsiterationen sehr stark gegen den Heur-Spieler spielt. Die schnellen Lernerfolge der NTuple im Spiel Othello können bestätigt werden[Jaś14]. Unsere Ergebnisse zeigen, dass durch bestärkendes Lernen die Mixed-NTuple Netzwerke bereits ab 50.000 Trainingsiterationen in einem Round-Robin-Tournament besser abschneiden als das 2er-NTuple Netzwerk, was auf die größe der Netzwerke zurückzuführen ist.

Literatur

[BB59] Woodrow Wilson Bledsoe and Iben Browning. Pattern recognition and reading by machine. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 225–232. ACM, 1959.

[Bor16] Steven Borowiec. AlphaGo seals 4-1 victory over Go grandmaster Lee Sedol. *The Guardian*, 15, 2016.

[BPW+12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[Bur97] Michael Buro. The Othello match of the year: Takeshi Murakami vs. Logistello. *ICGA Journal*, 20(3):189–193, 1997.

[Del17] Richard Delorme. edax-reversi, 10 2017. https://github.com/abulmo/edax-reversi, eingesehen am 08.11.2019.

[Jaś14] Wojciech Jaśkowski. Systematic n-tuple networks for Othello position evaluation. *ICGA Journal*, 37:85–96, 2014.

[Kon17] Wolfgang Konen. The GBG Class Interface Tutorial: General Board Game Playing and Learning. *Research Center CIOP (Computational Intelligence, Optimization and Data Mining), TH Köln–University of Applied Sciences, Tech. Rep*, 2017.

[Luc08] Simon M Lucas. Learning to play Othello with n-tuple systems. *Australian Journal of Intelligent Information Processing*, 4:1–20, 2008.

[vdRW13] Michiel van der Ree and Marco Wiering. Reinforcement learning in the game of Othello: learning against a fixed opponent and learning from self-play. In *2013 IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning (ADPRL)*, pages 108–115. IEEE, 2013.

A. Anhang

A.1. Designvorgang der NTuple

Der Designvorgang des in Abbildung 4a gezeigten NTuple-Agenten verlief iterativ. Jeder Iterationsschritt hatte dabei folgenden Aufbau: Zu erst wurde eine Änderung an der Tuple-Belegung vorgenommen, initial wurden die NTuple zufällig gewählt. Anschließend wurde der neue Agent in 200 000 Trainingseinheiten trainiert, um diesen, unter menschlicher Beobachtung, gegen Edax spielen zu lassen. Hat der Agent bei den Testspielen offensichtliche Fehlentscheidungen getroffen, der Agent hat also ein ,,schlechtes" Feld einem ,,besseren" Feld vorgezogen, musste das NTuple-Netzwerk an den entsprechenden Feldern angepasst werden. Entweder wurde das ,,schlechte" Feld abgeschwächt, indem ein Tuple von diesem Feld entfernt wurde oder das ,,bessere" Feld wurde durch ein neues Tuple stärker priorisiert. Nach etwa 30 solcher Änderungen war das Ergebnis das Mixed-NTuple1.

A.2. Bestimmung der Parametrisierung

Die Parameter der Agenten wurden in beiden Versuchen unterschiedlich gewählt, da jeweils eine andere Aufgabe im Vordergrund steht und mit anderen Einstellungen bessere Ergebnisse erzielt werden konnten. Der Vergleich gegen den Heur-Spieler besteht aus einer nur geringen Anzahl an Trainingsiterationen, wohingegen der Vergleich der Agenten untereinander mit einer höheren Iterationsanzahl getestet wird. Die Lernrate Alpha wurde anfangs auf 1.0 gesetzt und anschließend jeweils um 0.1 reduziert bis dieser Wert 0.1 erreicht hat. Während des Trainings ist dieser Wert konstant gehalten wurden.Eine steigende bzw. sinkende Lernrate die im Trainingsverlauf gegen 1.0 oder 0.0 läuft, hat sich nicht als Indikator für eine Steigerung der Performance erwiesen. Für den Test gegen den Heur-Spieler hat sich ein konstanter Wert von 0.5 für die Lernrate und für den Test, in dem die trainierten Agenten gegeneinander antreten, eine Lernrate von 0.2 als optimal herausgestellt. Der Abzugsfaktor Gamma wurde mit den Werten 1.0, 0.99, 0.98, 0.95 und 0.5 getestet. In beiden Versuchen zeigte sich, dass eine schnelle Belohnung in früheren Spielzuständen schlechtere Ergebnisse erzielen, als kein Abzug in späteren Spielzuständen. Ein hoher Epsilonwert ≥ 0.2 hat sich als nicht brauchbar herausgestellt, da sonst die Wahrscheinlichkeit einen Zufallszug auszuführen zu hoch ist. Unsere Agenten zeigen bei konstantem sowie steigendem Epsilon keine Erhöhung der Spielweise im Vergleich zu einem sinkenden gegen 0.0 laufenden Epsilon. Steigende Epsilonwerte wurden mit einer Obergrenze von 0.15 getestet. Zu Beginn wurden 0.4, 0.3, 0.2, 0.15, 0.1, 0.8 0.05, 0.02, 0.01 als Epsilon gewählt. Die Beobachtung zeigt, dass ein zu hoher Epsilonstartwert bei hohen Trainingsiterationen nicht zu stark ausschlägt, wenn dieser Wert gegen 0 läuft, jedoch bessere Ergebnisse erzielt werden können, wenn Epsilon niedriger angesetzt wird. Auch ein konstantes Epsilon hat nicht so gute Ergebnisse wie ein gegen 0.0 laufender Wert. Die stärksten Agenten sind im Heur-Spieler Vergleich, die bei denen 0.1 als Startwert und 0.0 als Endwert gewählt wurde und bei dem Vergleich der Agenten untereinander, die welche einen Startwert von 0.05 gegen 0.0 laufend haben.

Die Batch-Größe, die bestimmt nach wie vielen Trainingsiterationen ein Agent seine Gewichtung anpasst, wurde für die Agenten mit 200.000 Trainingsiterationen nach oben auf 1000 angepasst und für die schwächeren Agenten, die gegen den Heur-Spieler spielen von 100 auf 25 gemindert. Die Schrittgröße der Agenten mit hohen Iterationen wurde ausgehend von 500 bis 2000 mit einer Schrittgröße von 100 getestet. In dem Vergleich mit dem Heur-Spieler spielen die schwächsten Agenten nach 250 Trainingsiterationen. Deshalb wurde bei diesem Vergleich sukzessiv um 25 reduziert, sodass mindestens 10 Gewichtsanpassungen erfolgen können.