Hausarbeit: Rote-Learning anhand einer künstlichen Intelligenz für das Brettspiel Mühle

Robert Franzke (7222940) und Joost Ole Seddig (8713483)

1 Einleitung

Der α-β-Pruning Algorithmus versucht durch die Auswertung der zukünftig möglichen Züge den Besten auszuwählen. Da der Suchraum jedoch zu groß ist um jedes mögliche Spiel zu betrachten, muss an einem bestimmten Punkt das Vorausschauen beendet werden und eine Schätzung statt dem tatsächlichen Wert verwendet werden. Diese Aufgabe übernimmt die in dem Notebook `nmm-heuristic` implementierte Heuristik. Dies bedeutet dass eine künstliche Intelligenz, die den α-β-Pruning Algorithmus implementiert, je besser ist, desto mehr Schritte in die die Zukunft geschaut werden kann. Die Laufzeit α-β-Pruning Algorithmus nimmt jedoch mit zunehmender Tiefe, aufgrund der sehr schnell ansteigenden Anzahl der Zustände, deutlich zu. Somit ist eine Neuberechnung der Schäzung bei jedem Zug nicht praktikabel.

An diesem Punkt setzt das Rote-Learning an: Indem der α-β-Pruning Algorithmus um eine elementare Form des Lernens erweitert wird, wird dessen Effektivität stark gesteigert. Grundlegend werden bei der Verwendung von Rote-Learning alle Zustände, die jemals besucht wurden, zusammen mit den dazugehörigen errechneten Schätzungen abgespeichert. Anstatt die Wertschätzung dieser Zustände bei jedem Zug neu zu berechnen, können diese nun aus dem Speicher abgerufen werden. Dies spart vor allem bei einer hohen Suchtiefe viel Rechenzeit ein. Diese Einsparung der Rechenzeit kann darauf verwendet werden, weitere Zustände zu berechnen und somit die Qualität der Schätzungen zu erhöhen. Da mit jedem Spiel mehr Zustände und deren Schätzungen gespeichert werden, verbessert sich das Ergebnis des Algorithmus über Zeit. Es tritt also ein Lerneffekt ein.

1.1 Implementierung

Um das oben genannte Prinzip von Rote-Learning in Python zu implementieren, wurde auf eine bereits fertige Implementierung des α-β-Pruning Algorithmus für das Spiel Mühle zurück gegriffen. Dieser wurde im Rahmen der Studienarbeit entwickelt und musste für die Verwendung von Rote-Learning nur noch geringfügig angepasst werden. Um zu evaluieren ob Rote-Learning einen Vorteil gegenüber einem nicht trainierten Algorithmus herbeiführt, werden folgende Notebooks benötigt:

- `nmm-cache` implementiert eine schnelle, persistierbare Transpositionstabelle (auch Cache genannt);
- `nmm-rote-training` implementierung ein Klasse für den Trainingsteil des Rote-Learnings;
- `nmm-alpha-beta-pruning` ist eine bereits vorhandene Implementierung des α-β-Pruning Algorithmus;

- `nmm-tournament` ist eine ebenfalls bereits vorhandene Implementierung zur Evaluation, ob ein per Rote-Learning trainierter Algorithmus besser abschneidet als ein nicht trainierter Algorithmus.

[ ]:
```
%run ./nmm-cache.ipynb
%run ./nmm-rote-training.ipynb
%run ./nmm-alpha-beta-pruning.ipynb
%run ./nmm-tournament.ipynb
```

Desweitern wird das Paket `memory_profiler` eingebunden, welches die Überwachung der Auslastung des Arbeitsspeichers ermöglicht.

[ ]:
```
from memory_profiler import memory_usage
```

2 nmm-cache: Persistente Transpositionstabelle

Der Training Prozess des Rote-Learnings ist sehr zeitaufwendig, da viele Spiele gespielt werden müssen. Damit dieser Prozess nicht vor jedem Spiel ausgeführt werden muss, ist es sinnvoll die trainierte Transpositionstabelle persistent auf der Festplatte abspeichern zu können. Es gibt mehrere Möglichkeiten diese Transpositionstabelle zu implementieren, jedoch ist die Geschwindigkeit hier einer des ausschlaggebendsten Faktoren. Um zu ermitteln wie die Tabelle implementiert werden soll, wurde also ein Geschwindigkeitsvergleich erstellt. Dieser ist im Notebook `nmm-rote-performance-testing` zu finden. Das Ergebnis dieses Vergleiches hat ergeben, dass sich ein Python-Dictionary im Arbeitsspeicher gut eignet. Dieses kann ebenfalls in einer binären Datei auf einem Datenträger persistiert werden.

2.1 Cache

Die Klasse `Cache` implementiert eine Transpositionstabelle die persistiert werden kann. Der Konstruktor erhält folgenden Parameter:

- `max_size` gibt an, wie viele Einträge sich maximal im Cache befinden dürfen, bevor Einträge mit dem geringsten Rekursionslimit entfernt werden.
- `path` gibt an, aus welcher Datei der Cache geladen werden soll. Ist dieser nicht gesetzt wird ein neuer leerer Cache initiiert.

[ ]:
```
class Cache:
    def __init__(self, max_size = 1_000_000, path: str = None):
        self.max_size = max_size
        self.cache = {}
        if path:
            self.load(path)
```

Für Entwicklungszwecke wird eine Stringdarstellung für die Klasse `Cache` implementiert. Hierzu wird durch die Funktion `__repr__` ein String zurückgegeben, der alle Parameter der Klasse beinhaltet.

[ ]:
```
def __repr__(self: Cache) -> str:
    return f"Cache(size={len(self.cache)}, max_size={self.max_size})"

Cache.__repr__ = __repr__
del __repr__
```

Damit die Zustände und die Werte direkt als Bytes gespeichert und wieder ausgelesen werden können, ist das Paket `struct` nötig. Dieses ermöglicht `float` Werte zu `bytes` zu konvertieren. Das Paket `tqdm` ermöglicht eine simple Fortschrittsanzeige.

[ ]:
```
import struct
from tqdm.notebook import tqdm
```

Die Methode `convert_state_to_bytes` konvertiert einen Zustand in ein Byte-Array der Länge 7. Der Zustand wird dabei immer auf den weißen Spieler normiert. Durch die Normierung müssen die Zustände nur noch für einen Spieler gespeichert werden, worduch für die gleiche Menge an effektiv verwendbaren Zuständen nur noch die Hälfte des Speichers benötigt wird. Diese Optimierung ist jedoch nur effektiv, wenn beide Spieler die gleiche Anzahl an Steinen haben. In der ersten Phase liegt bei dem weißen Spieler (meist) ein Stein mehr auf dem Spielbrett als bei dem schwarzen Spieler. Deshalb bringt die Normierung zu Beginn in den meisten Fällen nichts, im späteren Verlauf ist sie jedoch sinnvoll.

Dabei werden folgende Argumente erwartet:

- `state` $\in$ *States*;
- `player` $\in$ *Player*.

Das resultierende Byte-Array besteht aus:

- `bytes[0]` beinhaltet den Stapel, wobei die ersten 4 Bits vom weißen und die letzten 4 Bits vom schwarzen Spieler belegt sind;
- `bytes[1:3]` beinhaltet die Steine des weißen Spielers auf dem Spielbrett, beginnend mit dem inneren Ring und dem letzten Stein des Ringes;
- `bytes[4:7]` beinhaltet die Steine des schwarzen Spielers auf dem Spielbrett im gleichen Format.

[ ]:
```
def convert_state_to_bytes(state, player):
    players = ['w','b']
    if player == 'b':
        players = ['b','w']
        state = (state[0][::-1],state[1])
    byte_data = ((state[0][0] << 4) | state[0][1]).to_bytes(1,'big')
    for player in players:
        for ring in range(2,-1,-1):
            ring_byte = 0
            for cell in range(7,-1,-1):
                ring_byte <<= 1
                if state[1][ring][cell] is player:
                    ring_byte |= 1
            byte_data += ring_byte.to_bytes(1,'big')
```

```
    return byte_data
```

Die Methode `write` schreibt einen Zustand in die Transpositionstabelle. Dabei wird der Zustand auf den weißen Spieler normiert. Es werden folgende Argumente erwartet:

- `state` $\in$ *States*;
- `player` $\in$ *Player*;
- `limit` $\in \mathbb{N}_0$;
- `value` $\in [-1.0, 1.0]$;
- `alpha` $\in [-1.0, 1.0]$;
- `beta` $\in [-1.0, 1.0]$.

[ ]:
```
def write(self: Cache, state, player: str, limit: int, value: float, alpha:␣
↪float, beta: float) -> None:
    state = convert_state_to_bytes(state, player)
    key = state + limit.to_bytes(1,'big')
    if player == 'b':
        value, alpha, beta = -value, -beta, -alpha
    value = struct.pack("d", value) + struct.pack("d",alpha) + struct.
↪pack("d",beta)
    self.cache[key] = value

Cache.write = write
del write
```

Die Methode `read` liest einen vorher gespeicherten Zustand aus der Transpositionstabelle aus. Falls der Zustand nicht vorhanden ist wird `None` zurückgegeben. Dabei wird die Normierung auf den weißen Spieler rückgängig gemacht. Folgende Argumente werden erwartet:

- `state` $\in$ *States*;
- `player` $\in$ *Player*;
- `limit` $\in \mathbb{N}_0$.

Zurückgegeben wird ein Tripel bestehend aus:

1. `value` $\in [-1.0, 1.0]$;
2. `alpha` $\in [-1.0, 1.0]$;
3. `beta` $\in [-1.0, 1.0]$.

[ ]:
```
def read(self: Cache, state, player: str, limit: int) -> (float,float,float):
    state = convert_state_to_bytes(state, player)
    key = state + limit.to_bytes(1,'big')
    result = self.cache.get(key)
    if not result:
        return None
    value = struct.unpack("d", result[:8])[0]
    alpha = struct.unpack("d", result[8:16])[0]
    beta = struct.unpack("d", result[16:24])[0]
    if player == 'b':
```

```
        value, alpha, beta = -value, -beta, -alpha
    return (value, alpha, beta)

Cache.read = read
del read
```

Die Methode `clean` prüft ob der Cache seine maximale Größe überschritten hat. Ist dies der Fall werden Einträge beginnend mit dem Rekursionslimit $limit = 0$ aus dem Cache entfernt. Solange die Transpositionstabelle weiterhin ihre maximale Größe überschreitet, wird das minimale Rekursionslimit erhöht und die unterschreitenden Einträge gelöscht.

[ ]:
```
def clean(self: Cache):
    min_limit = 0

    while len(self.cache) > self.max_size:
        min_limit += 1

        pre_len = len(self.cache)
        self.cache = {
            key: value
            for key, value in self.cache.items()
            if key[7] > min_limit
        }

        print(f"Increased min_limit to {min_limit} and deleted {pre_len -␣
 ↪len(self.cache)} entries. " + \
              f"Cache is now {len(self.cache)} entries big.")

    return min_limit != 0

Cache.clean = clean
del clean
```

Die Methode `save` persistiert den Cache auf dem Dateisystem. Dafür wird folgendes Argument erwartet:

- `path` beschreibt den Pfad zur zu schreibenden Datei im Dateisystem.

[ ]:
```
def save(self: Cache, path: str):
    with open(path, "wb") as file:
        for key,value in tqdm(self.cache.items()):
            file.write(key)
            file.write(value)

Cache.save = save
del save
```

Die Methode `load` lädt einen zuvor persistierten Cache aus einer Datei, die sich auf dem Dateisys-

tem des Computers befindet. Dafür wird folgendes Argument erwartet:

- `path` beschreibt den Pfad zur zu lesenden Datei im Dateisystem.

[ ]:
```
def load(self: Cache, path: str):
    if not os.path.isfile(path):
        print(f'Failed to load cache from file {path}!')
        return
    with open(path, "rb") as file:
        while True:
            key = file.read(8)
            value = file.read(24)
            if not key or not value:
                break
            self.cache[key] = value
    print(f'Successfully loaded cache from file {path}!')
Cache.load = load
del load
```

3 nmm-rote-performance-testing: Geschwindigkeit Test der Transpositionstabelle

Die Klasse `Cache` aus dem Notebook `nmm-cache` implementiert eine Transpositionstabelle die persistiert werden kann. Um zu ermitteln wie diese implementiert werden soll, wurde zwischen zwei Möglichkeiten abgewogen.

1. Zum einen steht die Verwendung einer externen Key-Value-Datenbank zur Auswahl. Diese wird in einem weiteren Container auf dem selben Computer ausgeführt, dadurch werden die Netzwerklatenzen minimiert.
2. Die zweite Möglichkeit ist das Speichern im Arbeitsspeicher in einem Python-Dictionary. Dieses kann anschließend exportiert und auf einen persistenten Datenträger geschrieben werden.

Um zu ermitteln welche Methode verwendet werden sollte, wird im Nachfolgenden die Geschwindigkeit der jeweiligen Methode getestet und bewertet.

3.1 Vorbereitungen

Bevor die Möglichkeiten getestet werden können, müssen folgende Pakete geladen werden:

- `redis` implementiert die Kommunikation mit der Redis-Datenbank;
- `struct` übersetzt `float` Werte in `bytes`;
- `random` ermittelt Zufallswerte zum testen;
- `tqdm` zeigt den Fortschritt an.

[ ]:
```
import redis
import struct
import random
from tqdm.notebook import tqdm
```

Des Weiteren wird die Klasse `CacheTest` angelegt, welche als Interface für die jeweilige Implementierung dient. Zwei Funktionen werden vorgegeben:

- `write` schreibt die Werte `value`, `alpha` und `beta` an dem Schlüssel `state` und `player` in den Cache;
- `read` ließt die mit `write` gespeicherten Werte aus oder gibt `None` zurück, falls der Cache diese nicht beinhaltet.

[ ]:
```
class CacheTest():
    def write(self, state: int, player: bool, limit: int, value: float, alpha:␣
↪float, beta: float) -> None:
        pass

    def read(self, state: int, player: bool, limit: int) -> (float, float,␣
↪float):
        """Value, Alpha, Beta"""
        pass
```

3.2 Python-Dictionary

Der native Python Cache besteht aus einem einfachen Python-Dictionary in welches die Key-Value-Paare gespeichert werden.

[ ]:
```
class PythonCache(CacheTest):
    def __init__(self):
        self.cache = {}
        pass

    def write(self, state: int, player: bool, limit: int, value: float, alpha:␣
↪float, beta: float) -> None:
        key = state.to_bytes(8,'big') + player.to_bytes(1,'big') + limit.
↪to_bytes(1,'big')
        value = struct.pack("d", value) + struct.pack("d",alpha) + struct.
↪pack("d",beta)
        self.cache[key] = value

    def read(self, state: int, player: bool, limit: int) -> (float,float,float):
        """Value, Alpha, Beta"""
        key = state.to_bytes(8,'big') + player.to_bytes(1,'big') + limit.
↪to_bytes(1,'big')
        value = self.cache.get(key)
        return (
            struct.unpack("d", value[:8])[0],
            struct.unpack("d", value[8:16])[0],
            struct.unpack("d", value[16:24])[0],
        ) if value else None
```

3.3 Redis

Der Redis-Cache implementiert eine Transpositionstabelle auf basierend auf der Key-Value-Datenbank *Redis*. Die Übersetzung der Werte zu Byte-Arrays erfolgt äquivalent zu dem Python-Cache. Die Werte werden danach mit Hilfe des `redis` Pakets in der lokalen Redis-Datenbank gespeichert.

[ ]:
```
class RedisCache(CacheTest):
    r = redis.Redis(host='redis')
    def write(self, state: int, player: bool, limit: int, value: float, alpha:␣
↪float, beta: float) -> None:
        key = state.to_bytes(8,'big') + player.to_bytes(1,'big') + limit.
↪to_bytes(1,'big')
        value = struct.pack("d",value) + struct.pack("d",alpha) + struct.
↪pack("d",beta)
        self.r.set(key,value)

    def read(self, state: int, player: bool, limit: int):
        key = state.to_bytes(8,'big') + player.to_bytes(1,'big') + limit.
↪to_bytes(1,'big')
        value = self.r.get(key)
        return (
            struct.unpack("d", value[:8])[0],
            struct.unpack("d", value[8:16])[0],
            struct.unpack("d", value[16:24])[0],
        ) if value else None
```

3.4 Test

Als Vorbereitung des Tests werden zunächst Instanzen der einzelnen `CacheTest` Implementierungen erstellt. Außerdem werden Einstellungen für die Tests vorgenommen:

- `count` ist die Anzahl der Elemente, die in die Caches geschrieben werden sollen;
- `start` bis `end` ist das Intervall in dem Zustände generiert werden. Ein Zustand ist in diesem Test jedeglich eine Zahl.

[ ]:
```
pc = PythonCache()
rc = RedisCache()

count =     1_000_000
start = 1_000_000_000
end   = start+count
```

3.4.1 Test der Schreibgeschwindigkeit

Durch die Einstellung `count` gesteuert werden zufällige Werte in den Caches gespeichert. In diesem Fall sind es eine Millionen Einträge.

[ ]:
```
%%time
print('Python Cache')
for state in tqdm(range(start, end)):
    for player in [True, False]:
        for limit in range(4):
            pc.write(state, player, limit, random.random(), random.random(),␣
 ↪-random.random())
```

[ ]:
```
%%time
print('Redis Cache')
for state in tqdm(range(start, end)):
    for player in [True, False]:
        for limit in range(4):
            rc.write(state, player, limit, random.random(), random.random(),␣
 ↪-random.random())
```

Im Vergleich zu dem `PythonCache` ist der `RedisCache` bei dem Schreibvorgang der Einträge etwa um den Faktor 90 langsamer.

3.4.2 Testen der Lesegeschwindigkeit

Bei dem Test der Lesegeschwindigkeit wird gemessen, wie schnell die Werte für zufällig ermittelte Zustände zurück gegeben werden können. Dabei wird auf die oben generierten Einträge zurück gegriffen.

[ ]:
```
%%time
valid = 0
for count in tqdm(range(count)):
    state = random.randint(start, end-1)
    limit = random.randint(0, 3)
    player = random.choice([True, False])
    result = pc.read(state, player, limit)
    if result is None:
        print(state, player, limit)
    else:
        valid += 1
print(valid)
```

[ ]:
```
%%time
valid = 0
for count in tqdm(range(count)):
    state = random.randint(start, end-1)
    limit = random.randint(0, 3)
    player = random.choice([True, False])
    result = rc.read(state, player, limit)
    if result is None:
        print(state, player, limit)
```

```
    else:
        valid += 1
print(valid)
```

Während die Lesegeschwindigkeit beim `RedisCache` deutlich höher als die Schreibgeschwindigkeit ist, ist diese dennoch um etwa den Faktor 4 langsamer als die des `PythonCaches`.

3.5 Resumee der Cache Implementierungen

Sowohl die Lese- als auch die Schreibgeschwindigkeit ist beim `PythonCache` deutlich höher als beim `RedisCache`. Wobei die Geschwindigkeitsdifferenz beim Schreiben stärker ausfällt als beim Lesen. Dies ist höchst wahrscheinlich auf den Overhead der Verbindung zwischen Datenbank und dem Python-Programm zurückzuführen.

Aus diesem Grund wurde sich dafür entschieden, den Cache als Python-Dictionary zu implementieren und anschließend als Datei auf dem Datenträger zu persistieren. Hierbei wird in Kauf genommen, dass die wachsende Größe des Caches zu einer starken Auslastung des Arbeitsspeicher führen kann.

4 nmm-rote-training: Rote-Learning Training

Der Rote-Learning Algorithmus muss im wahrsten Sinne des Wortes auswendig lernen und das Spiel trainieren. Dazu werden möglichst viele Spiele hintereinander gespielt und die Transpositionstabelle mit immer mehr und genaueren Daten gefüllt.

Zunächst werden beide implementierte Algorithmen geladen: `AlphaBetaPruning` und `Minimax`. Die `Match` Implementierung aus dem `Tournament` Notebook kann ebenfalls wieder verwendet werden.

[ ]:
```
%run ./nmm-alpha-beta-pruning.ipynb
%run ./nmm-minimax.ipynb
%run ./nmm-tournament.ipynb
```

Um eine übersichtlichere Entwicklung zu ermöglichen, werden Typdefinitionen geladen, welche später im Code verwendet werden. Das Paket `tqdm` ermöglicht eine einfache Fortschrittsanzeige.

[ ]:
```
from typing import Optional, Union, List, Callable
from tqdm.notebook import tqdm
```

4.1 Training

Innerhalb eines Trainings spielt eine künstliche Intelligenz viele Male gegen sich selbst und erweitert so ihre Transpositionstabelle. Bei jedem Spiel gegen sich selbst ist eine höhere Rekursionstiefe erreichbar, da der Cache bereits Werte von vorherigen Runden beinhaltet. Da die Implementierung der Transpositionstablle die gespeicherten Werte auf den weißen Spieler normiert, ist es sogar effizient möglich, dass sich beide künstliche Intelligenzen (weiß und schwarz) einen Cache teilen.

Innerhalb eines Trainings werden nur die ersten 18 Runden eines jeden Spieles gespielt, da alle diese Runden gezwungenermaßen in der ersten Spielphase `Placing` sind. Jeder Spieler muss alle seine 9 Steine zu Beginn der Runde setzen. Dadurch werden innerhalb der 18 Runden alle Steine von beiden Spielern gesetzt. Es wird nur die Spielphase `Placing` trainiert, da diese Phase im Vergleich zur zweiten Phase `Moving` um einiges komplexer ist. Bei jedem Zug kann jeder neue Stein eines Spielers auf jedes freie Feld bewegt werden, in der `Moving` Phase sind hingengen nur benachbarte leere Felder möglich. Dadurch ist der Suchraum für die Algorithmen zu Beginn des Spieles besonders groß und die künstliche Intelligenz kann dementsprechend weniger Züge in die Zukunft schauen. Durch das Auswendiglernen dieser Phase gewinnen die Algorithmen einen größeren Vorteil.

Auch die letzte Phase `Flying` ist sehr komplex und würde vom Auswendiglernen profitieren, allerdings wird hierzu eine Endspieldatenbank implementiert. Diese Datenbank deckt die letzte Phase besser ab als das Auswendiglernen.

Die Klasse `Training` implementiert solch ein Training und benötigt mehrere Parameter:

Verpflichtend:

- `cache` ist die Instanz der Transpositionstabelle, die innerhalb des Trainings verwendet und damit trainiert werden soll;
- `artificial_intelligence` ist eine Funktion die eine `ArtificialIntelligence` Instanz produziert, die jeweils für den weißen und den schwarzen Spieler aufgerufen wird;

Optional:

- `trainings` ist die Anzahl der Runden in denen trainiert werden sollen;
- `seed_offset` ist die Zahl, die auf den Seed addiert werden soll um verschiedene Trainingsläufe erstellen zu können;
- `path_prefix` ist der Prefix der vor den Dateinamen geschrieben wird, um den Cache Zwischenstand zu speichern;
- `save_interval` ist das Intervall in dem der Cache auf der Festplatte zwischen gespeichert werden soll.

[ ]:
```python
class Training():
    def __init__(
        self,
        cache: Cache,
        artificial_intelligence: Callable[[Cache], ArtificialIntelligence],
        trainings: int = 100,
        seed_offset: int = 0,
        path_prefix: str = "training-",
        save_interval: int = 10
    ):
        self.cache = cache
        self.artificial_intelligence = artificial_intelligence
        self.trainings = trainings
        self.seed_offset = seed_offset
        self.path_prefix = path_prefix
        self.save_interval = save_interval
```

Für Entwicklungszwecke wird eine Stringdarstellung für die Klasse `Training` implementiert. Hierzu wird durch die Funktion `__repr__` ein String zurückgegeben, der alle Parameter der Klasse beinhaltet.

[ ]:
```
def __repr__(self: Training):
    return f"Training(cache={self.cache}, " + \
           f"artificial_intelligence={type(self.artificial_intelligence).
 ↪__name__}, " + \
           f"trainings={self.trainings}, seed_offset={self.seed_offset}, " + \
           f"path_prefix='{self.path_prefix}', save_interval={self.
 ↪save_interval})"

Training.__repr__ = __repr__
del __repr__
```

Die Funktion `train` trainiert die gespeicherte Transpositionstabelle indem für die Anzahl der zu spielenden Runden (`trainings`) ein Spiel (`Match`) erstellt wird. Dieses Spiel wird von zwei `ArtificialIntelligence` Instanzen gespielt, welche durch die gespeicherte Funktion `artificial_intelligence` erstellt werden. Pro Spiel werden maximal 18 Runden, also nur die `Placing` Phase gespielt. Nach `save_interval` Spielen sowie am Ende wird die Transpositionstabelle mit der Rundennummer auf der Festplatte zwischengespeichert.

[ ]:
```
def train(self: Training):
    for i in tqdm(range(self.trainings)):
        name = f"{self.path_prefix}{i+1:0>3}"
        random.seed(self.seed_offset + i)
        match = Match(
            white     = self.artificial_intelligence(cache = self.cache),
            black     = self.artificial_intelligence(cache = self.cache),
            max_turns = 18,
            name      = name
        )

        print(f"Training #{name}:")
        match.play()
        print(f" > {self.cache}")

        if (i+1) % self.save_interval == 0:
            print(f"> Saving to '{name}.cache'...")
            self.cache.save(f"{name}.cache")
        print()
    self.cache.save(f"{self.path_prefix}final.cache")

Training.train = train
del train
```

5 Anwendung

Um den bei Rote-Learning beschriebenen Lerneffekt zu erzielen und die Zustände zu Speichern, wurde eine Transpositionstabelle als Klasse `Cache` in dem Notebook `nmm-cache` implementiert. Diese Klasse verfügt über fünf Methoden zur Verwaltung des Caches:

- `write` speichert einen neuen Zustand und die errechneten Werte ab;
- `read` liest die gespeicherten Werte eines Zustandes aus dem Cache aus;
- `save` legt den Inhalt des Caches in einer Datei auf einem Datenträger ab;
- `load` lädt die Daten des Caches aus einer Datei auf dem Datenträger;
- `clean` löscht Einträge aus dem Cache, falls die Anzahl der Einträge `max_size` überschritten wurde.

Um die einzelnen Werte für die Zustände im Cache abzulegen, werden sowohl die Zustände als auch die Werte in ein Byte-Array konvertiert. Insgesamt ist ein Eintrag des Caches somit 32 Bytes groß. Um zu verhindern, dass der Cache zu groß wird um ihn im Arbeitsspeicher oder auf dem Datenträger speichern zu können, wurde der Parameter `max_size` eingeführt. Dieser begrenzt den Cache auf eine maximale Anzahl an Einträgen. Um die Performance zu steigern, wird dies jedoch nicht beim jedem Aufruf der `write` Methode geprüft sondern nur beim Aufruf der Methode `clean`. Dies kann dazu führen, dass zwischen den Aufrufen der `clean` Methode mehr Einträge im Cache vorhanden sind, als durch `max_size` erlaubt. Wird der Cache auf dem Datenträger abgelegt, entspricht die Dateigröße genau der Anzahl an Elementen multipliziert mit 32 Bytes. Die Größe das Caches im Arbeitsspeicher ist jedoch um einen Faktoren von ca. 6 größer, da Python weiteren Arbeitsspeicher, beispielsweise für den Datentyp, reserviert.

Der folgende Befehl erstellt einen Cache mit einer Größe von 50 Millionen Zuständen.

```
[ ]: cache = Cache(max_size = 50_000_000)
```

5.1 Anbindung an α-β-Pruning

Da im α-β-Pruning Algorithmus bereits die Verwendung eines In-Memory-Caches vorgesehen ist, mussten nur kleine Anpassungen vorgenommen werden. Es musste sichergestellt werden, dass sowohl zum Schreiben als auch zum Lesen der Werte aus dem Cache die richtigen Methoden aufgerufen werden, sowie dass ein Cache entweder als Parameter übergeben werden kann oder ein neuer instanziiert wird. Nach jeder Berechnung der nächsten besten Züge mit Hilfe der `bestMoves` Methode, wurde der Aufruf der `clean` Methode hinterlegt, um die Größe des Caches anzupassen.

5.2 Traingingsprozess

Um die Transpositionstabelle zu trainieren und somit das Rote-Learning umzusetzen, wird die Klasse `Training` aus dem Notebook `nmm-rote-training` verwendet. Dazu muss zuerst eine Methode erstellt werden, welche den Algorithmus für die künstliche Intelligenz konfiguriert und generiert. Diese Methode wird `artificial_intelligence_generator` genannt und generiert eine AlphaBetaPruning Instanz, welche den übergebenen Cache und eine benutzerdefinierte Heuristiken verwendet. Die verwendeten Heuristiken wurden bereits im Rahmen der Studienarbeit ermittelt.

```
[ ]: def artificial_intelligence_generator(cache: Cache):
         customWeights = HeuristicWeights(stones = 3, stash = 3, mills = 2,␣
     ↪possible_mills = 1)
         return AlphaBetaPruning(cache = cache, weights = customWeights)
```

Für den Cache wurde sich für eine maximale Größe von 50.000.000 Einträgen entschieden. Dies resultiert in eine maximal Größe des Caches von ca. 1,6Gb auf dem Datenträger. Des Weiteren werden die Standardwerte der Training-Klasse verwendet. Das heißt, dass ingesamt 100 Spiele gespielt werden und der Seed für den Zufallsgenerator nicht angepasst wird. Alle 10 Spiele wird der Cache auf dem Datenträger gespeichert und wird der Prefix `training-` für den Dateinamen verwendet.

```
[ ]: training = Training(
         cache = cache,
         artificial_intelligence = artificial_intelligence_generator,
         path_prefix = 'training-small-'
     )
```

Der Trainingsprozess wird durch den Aufruf der Funktion `train` gestartet und dauert mit der oben genannten Konfiguration in etwa 4,5 Stunden. Dabei werden bis zu 10 Gb Arbeitsspeicher benötigt.

```
[ ]: mem_usage = memory_usage(training.train)
     print('Maximum memory usage: %s MB.' % max(mem_usage))
```

5.2.1 Trainings Limitierungen

Auf den ersten Blick scheint es so, dass man den Cache unendlich weiter trainieren kann. Dies jedoch mit der aktuellen Implementierung nicht zielführend. Ab einem gewissen Trainingszeitpunkt ist das Rekursionslimit der Einträge im Cache so hoch, dass dieses durch weiteres Training nicht erneut erreicht werden können. Dies ist lässt sich gut an nachfolgendem Beispiel erkennen:

Durch die geringe Größe des Caches sowie dem kleinen Wert für `max_states` kommt man sehr schnell an den Punkt, an dem das Cache-Limit auf 3 erhöht wird. Dadurch werden fast alle Einträge aus dem Cache gelöscht und es verbleiben nur noch 48 im Cache. Dies passiert mehrere Male und ist ein Zeichen dafür, dass der Cache nicht mehr weiter trainiert werden kann, da dieser zu klein ist um genügend Einträge zu halten, damit mehr Einträge mit `limit = 3` berechnet werden können bevor der `max_states` Wert überschritten wird.

```
[ ]: t = Training(
         cache                  = Cache(max_size = 1_000),
         artificial_intelligence = lambda cache: AlphaBetaPruning(
             cache      = cache,
             max_states = 1_000
         ),
         path_prefix            = "max-learning-",
         save_interval          = 1
     )
```

```
t.train()
```

5.3 Auswertung

Bei der Auswertung wird auf die Klasse `Tournament` aus dem Notebook `nmm-tournament` zurück gegriffen. Diese Klasse besitzt die Methode `play`, welche es ermöglicht verschiedene Algorithmen gegeneinander Antreten zu lassen und die Ergebnisse zu speichern.

Um zu ermitteln ob die Rote-Learning-Methode einen Vorteil gegenüber einem normalen Cache bietet, tritt eine über 100 Spiele trainierte Trainspositionstabelle gegen eine untrainierte Transpositionstabelle an. Dies wird zehn mal mit verschiedenen Seeds wiederholt. Das verändern der Seeds sorgt dabei für einen veränderten Spielverlauf. Somit sollte ermittelt werden können, ob durch das Training eine Verbesserung der künstlichen Intelligenz eintritt und wenn ja, wie deutlich diese Verbesserung ausfällt.

[ ]:
```
for i in range(10):
    Tournament(
        [
            lambda: AlphaBetaPruning(
                name='New Cache',
                weights = HeuristicWeights(stones = 3, stash = 3, mills = 2,␣
↪possible_mills = 1),
                cache = Cache(max_size = 50_000_000)
            ),
            lambda: AlphaBetaPruning(
                name='Trained Cache',
                weights = HeuristicWeights(stones = 3, stash = 3, mills = 2,␣
↪possible_mills = 1),
                cache = Cache(max_size = 50_000_000, path =␣
↪'training-small-final.cache')
            )
        ],
        instances_per_round = 1,
        seed_offset         = i,
        name                = f"small-full-{i}-seed"
    ).play()
```

Die Auswertung hat ergeben, dass von insgesamt von 20 gespielten Spielen 10 gewonnen wurden. Bei 5 lief es auf ein Unentschieden herraus und 5 wurden verloren. Damit lässt sich mit ziemlicher Sicherheit sagen, dass das Training der ersten Spielphase durch Rote-Learning einen Vorteil im gesamten Spielverlauf bietet.

6 Fazit

Das Ziel dieser Arbeit war die Implementierung des in dem Paper `Some Studies in Machine Learning Using the Game of Checkers` von `A.L.Samuel` beschriebene Prinzip von Rote-Learning zu erarbeiten. Das Ziel wurde erfolgreich durch eine Python-Implementierung er-

reicht und durch eine Verbesserung der Leistung einer künstlichen Intelligenz bestätigt. Im Laufe der Arbeit wurde sich neben der Implementierung in Python auch mit den dem α-β-Pruning Algorithmus sowie verschiedenen Methoden zur Persistierung eines Caches beschäftigt.

Während in dem Paper jedoch zwei Methoden beschrieben werden, um Einträge aus einem zu großen Cache zu löschen, wird sich in dieser Arbeit nur auf eine Methode konzentriert. Im Paper wird zum einen die Aufruf-Frequenz oder zum anderen das Rekursionslimit des Eintrags. Diese Arbeit behandelt nur das Löschen von Einträgen basierend auf dem Rekursionslimit.

Dennoch lässt sich sagen, dass diese Arbeit ein Erfolg war, da eine deutliche Verbesserung des α-β-Pruning mit einem trainierten Cache erreicht werden konnte, sodass nach 100 Trainingsrunden nur noch 25% der Spiele gegen eine nicht trainierte künstliche Intelligenz verloren wurden.