Retrograde Analyse des 3-3 Endspiels beim Brettspiel Mühle

June 11, 2021

Hausarbeit von Benedikt Funke (3537931) und Niclas Kaufmann (8209645).

1 Einleitung

Im Rahmen dieser Hausarbeit soll eine retrograde Analyse des Brettspiels Mühle im Endspiel durchgeführt werden. Als Endspiel wird hier die letzte Phase des Spiels bezeichnet, wo beide Spieler nur noch drei Steine auf dem Spielbrett haben. Für die Analyse soll eine Endspieldatenbank implementiert werden, die daraufhin für jeden Zustand im Endspiel bestimmen kann, ob ein Spieler garantiert gewinnen kann und wie.

Die Hausarbeit wird im Fach Wissensbasierte Systeme des sechsten Semester im Studienfach Angewandte Informatik der DHBW Mannheim geschrieben. Die Arbeit ist eine Fortsetzung der Studienarbeit "*Studienarbeit zur Erstellung einer künstlichen Intelligenz zum Spielen des Brettspiels Mühle*". Die Studienarbeit hat eine künstliche Intelligenz für das Brettspiel mit den Algorithmen *Minimax* und *Alpha-Beta-Pruning* implementiert. Die grundlegende Implementierung des Spiels wird auch in der Hausarbeit weiterverwendet.

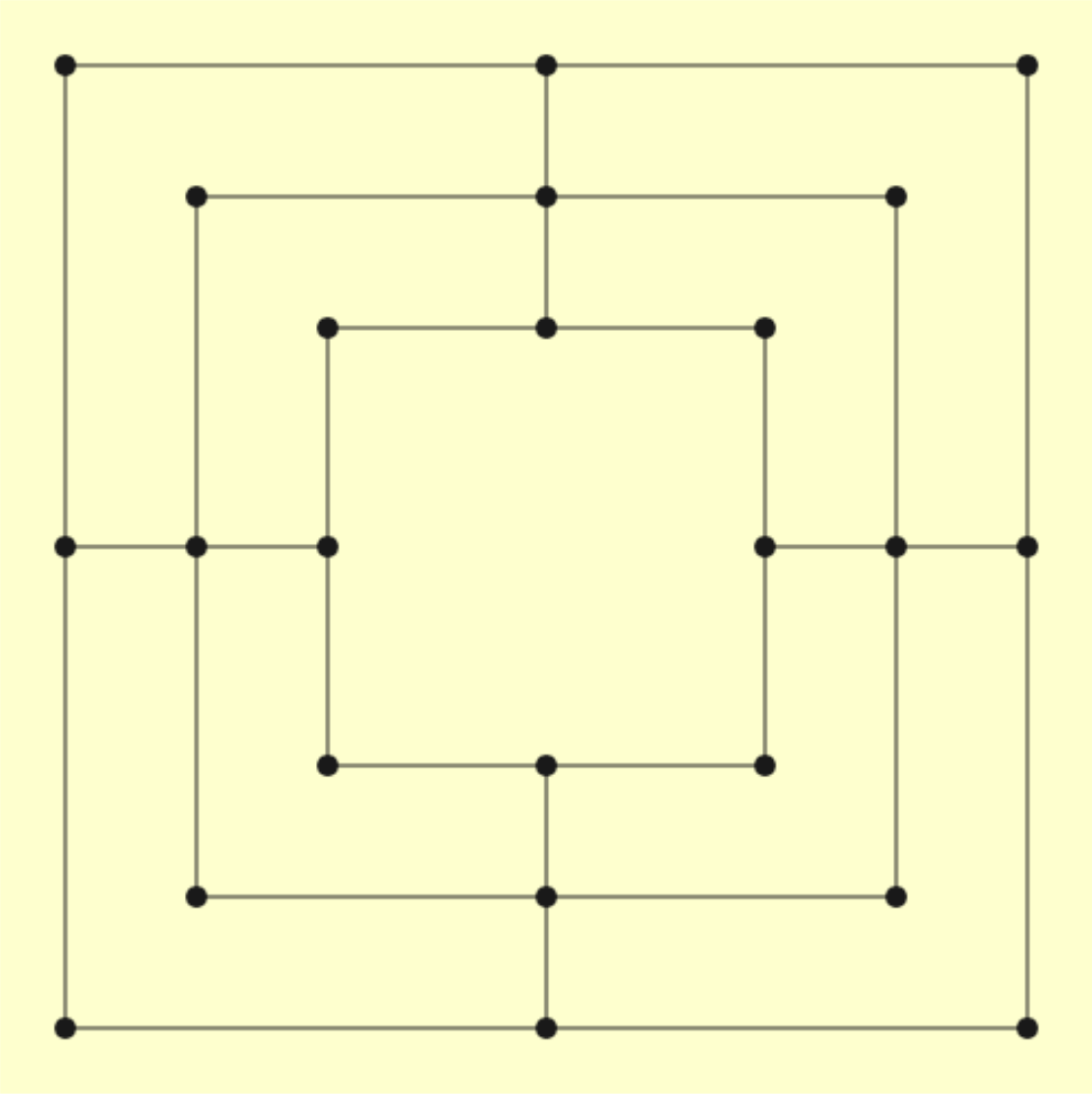
Am Anfang der Hausarbeit wird das Brettspiel Mühle erklärt und was eine Endspieldatenbank ist. Anschließend wird die Implementierung dieser besprochen und wie die Datenbank in das Spiel integriert werden kann. Zum Abschluss der Arbeit wird das 3-3 Endspiel von Mühle analysiert und es sollen Verbesserungsvorschläge gegeben werden, wie die Implementierung verbessert werden kann.

2 Theorie

2.1 Mühle

Das Spiel Mühle ist ein Brettspiel für zwei Spieler, welches schon weit vor Christus gespielt worden sein soll. Somit zählt es neben Spielen wie Schach und Go zu den ältesten uns bekannten Brettspielen, welches noch gespielt wird. In dieser Hausarbeit wird das Spiel nach den offiziellen Turnierregeln des Weltmühlespiel Dachverband gespielt. Im englischen Sprachraum wird Mühle als *Nine men's morris* bezeichnet.

Das Spielbrett besteht aus drei Quadraten, die durch vier Verbindungslinien verbunden sind. An den Eck- und Verbindungspunkten können die Spielsteine platziert werden. Insgesamt gibt es für jeden Spieler neun Spielsteine, in der Regel in den Farben weiß und schwarz. In der folgenden Abbildung ist ein leeres Spielbrett dargestellt.



Das Spiel gliedert sich in drei Spielphasen:

1. Zu Beginn des Spiels ist die **Setzphase**. Jeder Spieler darf nacheinander einen Stein beliebig auf einen freien Platz setzen. Solange bis alle neun Steine der Spieler auf dem Brett befinden. Somit hat die erste Phase eine vorgebene Länge von 18 Zügen.
2. Die zweite Phase ist die **Zugphase**, diese folgt nach der Setzphase. In dieser Phase ziehen die Spieler ihre Steine auf ein freies Nachbarfeld. Diese Phase hat keine feste Dauer und endet entweder, indem das Spiel vorbei ist oder die dritte Phase beginnt.
3. Die dritte Phase ist die **End-** oder **Flugphase**. Diese beginnt für einen Spieler, sobald er nur noch drei Spielsteine auf dem Brett hat. Somit kann sich ein Spieler schon in der Endphase befindet, wobei der Gegenspieler noch in der Zugphase ist. In dieser Phase darf ein Spieler seinen Stein beliebig auf freie Felder bewegen und ist nicht auf Nachbarfelder begrenzt.

Das Spiel ist vorbei, sobald ein Spieler verloren hat. Ein Spieler hat genau dann verloren, wenn

- er keinen regulären Zug mehr ausführen kann (er also von dem Gegenspieler eingebaut wurde), oder

- er weniger als drei Steine auf dem Spielfeld hat (nicht in der Setzphase).

Um das Spiel gewinnen zu können, müssen die Spieler versuchen, Mühlen zu bauen. Eine Mühle besteht aus drei Steinen, die sich auf einer Linie befinden. Es gibt 16 verschiedene Möglichkeiten eine Mühle auf dem Spielbrett zu bauen (jeweils vier auf den Seitenlinien der drei Quadrate und auf den vier Verbindungslinien). Hat der Spieler eine Mühle gebaut, darf er von dem Gegenspieler einen Spielstein vom Brett entfernen. Dabei ist darauf zu achten, dass sich der Spielstein nicht in einer Mühle befindet (Ausnahme: Alle Spielsteine des Gegenspielers befinden sich in einer Mühle). Es ist in jeder Spielphase möglich eine Mühle zu bilden. In der Setzphase kann ein Spieler mit einem Zug zwei Mühlen gleichzeitig bilden, dann darf er auch zwei Steine von dem Gegenspieler entfernen.

2.2 Endspieldatenbank

Eine Endspieldatenbank enthält vollständiges Wissen über ein Spielbrett mit einer maximalen Anzahl an Steinen auf dem Brett. So können in dem Endspiel perfekte Spielzüge ohne Rechenzeit gespielt werden. Endspieldatenbanken haben einen großen Speicherbedarf, weil die Anzahl an möglichen Spielstellungen mit mehr Steinen auf dem Brett extrem wächst. So gibt Ralf Gasser in seinem Paper an, dass es in der Zug- und Endphase des Spiels unter Berücksichtigung von Spiegelungen und ungültigen Zuständen noch 7.673.759.269 Zustände gibt [RG97].

2.2.1 Algorithmus zur Erstellung von Endspieldatenbanken

Schon 1912 soll der Mathematiker Ernst Zermelo auf einem Mathematikerkongress den folgenden Algorithmus zur Herstellung von Endspieldatenbanken vorgestellt haben. Dieser Algorithmus findet auch heutzutage noch Anwendung und lässt sich in vier Schritten theoretisch beschreiben. Die genaue Implementierung und Abwandlung für das Brettspiel Mühle wird in dem Kapitel *Implementierung* beschrieben.

Schritt 1: Es werden alle gültigen Zustände mit nicht mehr als n Steinen auf dem Spielbrett erzeugt.

Schritt 2: Im zweiten Schritt werden alle Gewinnstellungen für weiß gesucht:

1. Es werden alle Zustände gesucht, bei denen schwarz direkt verloren hat.
2. Es werden alle Zustände gesucht, an denen weiß am Zug ist und *mindestens ein* Zug zu einer Stellung unter 1. führt. Bei diesen Zuständen hat schwarz in einem Zug verloren.
3. Es werden alle Zustände gesucht, bei denen schwarz am Zug ist und bei dem *jeder* Zug zu einer Stellung aus 2. führt. Hier kann schwarz eine Niederlage in einem Zug nicht verhindern.
4. Es werden alle Zustände gesucht, an denen weiß am Zug ist und *mindestens ein* Zug zu einer Stellung unter 3. führt. Bei diesen Zuständen hat schwarz in zwei Zügen verloren.
5. Es werden alle Zustände gesucht, bei denen schwarz am Zug ist und bei dem *jeder* Zug zu einer Stellung aus 2. oder 4. führt. Hier kann schwarz eine Niederlage in zwei Zügen nicht verhindern.

6. Es werden alle Zustände gesucht, an denen weiß am Zug ist und *mindestens ein* Zug zu einer Stellung unter 5. führt. Bei diesen Zuständen hat schwarz in drei Zügen verloren.

7. Es werden alle Zustände gesucht, bei denen schwarz am Zug ist und bei dem *jeder* Zug zu einer Stellung aus 2., 4. oder 6. führt. Hier kann schwarz eine Niederlage in drei Zügen nicht verhindern.

   usw.

Der Schritt wird solange wiederholt, bis es keine neuen Zustände mehr gibt. Damit sind alle Stellungen gefunden, mit denen weiß bei maximal n Steinen auf dem Brett gewinnt.

Schritt 3: Im dritten Schritt werden alle Gewinnstellungen für schwarz gesucht, dies erfolgt analog wie Schritt 2.

Schritt 4: Alle Zustände, die nicht in Schritt zwei auftauchen, laufen bei einem perfekten Spiel von beiden Spielern auf ein Unentschieden hinaus, da keiner der beiden Spieler verlieren kann.

Der Algorithmus von Zermelo ist allgemein formuliert und es existieren einige Verbesserungen für den Algorithmus, besonders für das Schachspiel. Eine abgewandelte Implementierung für das 3-3 Endspiel für Mühle wird in dem nächsten Kapitel beschrieben.

3 Implementierung

In diesem Kapitel soll die Implementierung der Endspieldatenbank kommentiert werden. Für die Generierung müssen zwei Notebooks aus der Studienarbeit ausgeführt werden.

[ ]:
```
%run ./nmm-game.ipynb
%run ./nmm-symmetry.ipynb
```

Die Daten sollen in einer MongoDB gespeichert werden. Die MongoDB kann gestartet werden mit dem Docker Befehl `docker-compose up` in dem Verzeichnis `../retro-database`.

Um eine Verbindung zu der Datenbank herzustellen, muss die Variable `dbUrl` angepasst werden:

- Falls Jupyter Notebook ohne Docker ausgeführt wird, ist `'localhost'` zu verwenden;
- Falls Jupyter Notebook auch in einem Docker-Container ausgeführt wird, ist der Name des Docker-Containers der Datenbank zu verwenden. Dieser sollte bei Verwenden der Docker-Compose-Datei `'retro-database'` sein.

[ ]:
```
import pymongo
import math

dbUrl = 'mongodb://localhost:27017/'
# dbUrl = 'mongodb://retro-database:27017/'

localMongo = pymongo.MongoClient("mongodb://retro-database:27017/")
retroDb = localMongo["retro-database"]
retroCol = retroDb["retro-collection"]
```

Die Funktion `loadDataFromJson` dient dazu, die vorgenerierten Daten in die Datenbank einzulesen. Die Daten wurden bei der Abgabe der Hausarbeit erstellt und befinden sich in den beiden JSON-Dateien *../retro-database/retro-collection-1.json* und *../retro-database/retro-collection-2.json*. Bevor die Dateien in die Datenbank geladen werden, wird die Datenbank geleert.

[ ]:
```
from bson.json_util import loads

def loadDataFromJson():
    retroCol.delete_many({})
    for i in range(1, 3):
        with open(f'../retro-database/retro-collection-{i}.json', 'r') as file:
            arr = loads(file.read())
            retroCol.insert_many(arr)
```

Wenn man den folgenden Kommentar entfernt und die Funktion ausführt, werden alle Daten in die Datenbank geladen. Somit spart man sich das aufwendige Generieren.

[ ]:
```
# loadDataFromJson()
```

3.1 Generierung der Spielfelder

Die Endspieldatenbank soll alle Zustände speichern, in denen der weiße Spieler auf jeden Fall gewinnt. Dafür ist es notwendig, zunächst alle möglichen Spielfelder mit drei weißen und drei schwarzen Spielsteinen zu generieren, um im Nachhinein untersuchen zu können, ob diese zu den Gewinnzuständen gehören.

Die Funktion `generateBoards` nimmt ein leeres Spielfeld und befüllt dieses Stein für Stein. Sie erhält ein Argument:

- `boards` ist das Ausgangsspielbrett. Standardwert ist ein leeres Spielbrett `startBoard`

Die Funktion setzt jeden der sechs Steine auf jede der 24 Positionen des Spielfeldes. So wird sichergestellt, dass auch wirklich jede mögliche Kombination erstellt wird. Da es nicht ausgeschlossen werden kann, dass zwei Steine auf die selbe Position gesetzt werden und somit nicht die gewünschte 3-3-Kombination vorhanden ist, wird vor dem Speichern der Spielbretter überprüft, ob für beide Farben drei Steine platziert wurden.

Da es für das Mühlespiel egal ist, ob beispielsweise der 1. oder der 3. weiße Stein auf der Position `(1, 4)` liegt, werden die Spielbretter in ein globales `set` gespeichert. Dadurch wird sichergestellt, dass ein Spielbrett nicht zwei mal abgespeichert ist.

Die Entscheidung, die Spielbretter global abzuspeichern wurde bewusst getroffen, um bei Änderungen an anderen Funktionen nicht die Generierung der Spielbretter neu starten zu müssen, welche einiges an Zeit benötigt.

[ ]:
```
startBoard = ((' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '),
    (' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '),
    (' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '))
boards = set()
```

```
def generateBoards(board=startBoard):
    global boards
    count = 0
    for whiteOneX in range(0,3):
        for whiteOneY in range(0, 8):
            count += 1
            print('Progress: ' + str(count) + '/24')
            for blackOneX in range(0,3):
                for blackOneY in range(0, 8):
                    for whiteTwoX in range(0,3):
                        for whiteTwoY in range (0, 8):
                            for blackTwoX in range(0,3):
                                for blackTwoY in range(0, 8):
                                    for whiteThreeX in range(0,3):
                                        for whiteThreeY in range(0,8):
                                            for blackThreeX in range(0,3):
                                                for blackThreeY in range(0,8):
                                                    board1 = place(board,␣
↪(whiteOneX, whiteOneY), 'w')
                                                    board2 = place(board1,␣
↪(blackOneX, blackOneY), 'b')
                                                    board3 = place(board2,␣
↪(whiteTwoX, whiteTwoY), 'w')
                                                    board4 = place(board3,␣
↪(blackTwoX, blackTwoY), 'b')
                                                    board5 = place(board4,␣
↪(whiteThreeX, whiteThreeY), 'w')
                                                    board6 = place(board5,␣
↪(blackThreeX, blackThreeY), 'b')
                                                    white = countStones(('',␣
↪board6), 'w')
                                                    black = countStones(('',␣
↪board6), 'b')
                                                    if (white == 3 and black ==␣
↪3):
                                                        boards.add(board6)
```

3.2 Symmetrien

Die Funktion `getWeightForState` berechnet eine Ganzzahl eines Zustands basierend auf den Positionen der Steine auf dem Brett. Sie hat zwei Argumente

- `state` ist ein Zustand eines Spiels;
- `minWeight` ist das aktuell kleinste Gewicht. Das Argument ist optional und der Standardwert ist ∞.

Die Funktion iteriert über alle 24 Felder des Spielbretts und erhöht bei jeder Position einen Zähler `counter`. Steht auf einer Position ein weißer Stein, wird das Gewicht des Zustands um $2^{counter}$

erhöht. Steht auf der Position ein schwarzer Steint, erhöht sich das Gewicht um $2^{counter+24}$.

Nach jeder Position wird überprüft, ob das aktuelle Gewicht das kleinste Gewicht überschreitet. Ist das der Fall, kann die Schleife vorher abbrechen, weil der Zustand nicht mehr als kleinster Zustand in Frage kommt.

[ ]:
```
def getWeightForState(state, minWeight = math.inf):
    weight = 0
    counter = 1
    for r in range(3):
        for c in range(8):
            weight += (2**(counter if state[1][r][c] == 'w' else counter + 24)␣
↪if state[1][r][c] != ' ' else 0)
            counter += 1
            if (weight > minWeight):
                return weight
    return weight
```

Die Funktion `getUniqueStateForSymmetry` berechnet einen eindeutigen Zustand für eine Menge von Symmetrien eines Zustands. Die Funktion hat ein Argument

- `state` ist ein Zustand eines Spiels.

Zuerst berechnet die Funktion alle Symmetrien des Zustands mit Hilfe der Funktion `findSymmetries`. Dann iteriert sie über alle Symmetrien und sucht den Zustand mit dem geringsten Gewicht. Das Gewicht eines Zustands wird mit der Funktion `getWeightForState` berechnet.

Der Sinn dieser Funktion ist, dass sichergestellt wird, dass bei einer Menge von symmetrischen Zuständen immer mit dem einem gleichen Zustand gearbeitet wird.

[ ]:
```
def getUniqueStateForSymmetry(state):
    symmetries = findSymmetries(state)

    state = None
    minWeight = math.inf
    for s in symmetries:
        weight = getWeightForState(s, minWeight)
        if weight < minWeight:
            state = s
            minWeight = weight
    return state
```

Die Funktion `getSymmetryUnique(boards)` erstellt eine Menge an einzigartigen Zuständen unter Berücksichtigung der Symmetrien. Sie erhält ein Argument:

- `boards` die unbereinigte Menge an Spielfeldern

`getSymmetryUnique` iteriert über jedes Spielfeld und erhält von der Funktion `getUniqueStateForSymmetry` das Spielfeld, welches unter allen Symmetrien die niedrigste Gewichtung (`weight`) aufweist und fügt dieses den `uniqueStates` hinzu. So wird sichergestellt,

das für alle Symmetrien nur ein Repräsentant abgespeichert wird. Als Rückgabe liefert die Funktion eine Menge, in der nur ein Repräsentant jeder Symmetrie abgespeichert ist.

Dies wird durchgeführt um die Menge der abzuspeichernden Zustände in der Datenbank zu verringern und somit die Gesamtperformance zu verbessern. Alle in dieser Funktion aussortierten Zustände können über Symmetrien durch den abgespeicherten Repräsentanten rekonstruiert werden.

[ ]:
```
def getSymmetryUnique(boards):
    uniqueStates = set()
    stones = (0, 0)
    for board in boards:
        state = tuple([stones, board])
        uniqueStates.add(getUniqueStateForSymmetry(state))
    return uniqueStates
```

3.3 Befüllen der Endspieldatenbank

Die Funktion `fillDb` führt alle notwendigen Schritte aus, um Zustände zu finden, welche immer zum Sieg führen und speichert diese in eine Datenbank ab. Die Funktion erhält ein Argument:

- `states` die Menge der zu überprüfenden Zustände

Die Funktion ist in zwei Teile gegliedert: 1. Suchen aller Zustände, in denen der weiße Spieler mit einem Zug gewinnen kann 2. Endlosschleife, bis keine neuen Zustände mehr gefunden werden: 1. Suchen aller Zustände, in denen der schwarze Spieler durch seinen Zug nur Zustände erreichen kann, bei welchem der weiße Spieler auf jeden Fall gewinnt. Diese Schritte werden auch als `halfSteps` bezeichnet, da sie für die Endspieldatenbank nur als Zwischenschritt eine Rolle spielen und nicht abgespeichert werden. 2. Suchen aller Zustände, in denen weiß durch einen Zug auf einen Zustand in $2A$ kommen kann. Diese Schritte werden auch als `fullSteps` bezeichnet, da diese Zustände zu einem Sieg von weiß führen und somit in die Endspieldatenbank abgespeichert werden.

Zünächst müssen einige Schritte zur Initialisierung vorgenommen werden: - Erstellen einer Zählvariable. Diese wird zusammen mit den Zuständen abgespeichert und gibt an, in maximal wie vielen Schritten der weiße Spieler gewinnt. - Erstellen von zwei Kopien der Menge `states` (`fullStates` und `halfStates`)

Die Kopien werden dafür verwendet, die zu überprüfenden Zustände bereitzuhalten. In `fullStates` sind alle Zustände gespeichert, welche für den weißen Spieler relevant sind. In `halfStates` alle, die für den schwarzen Spieler eine Rolle spielen. Zu Beginn haben beide Mengen den gleichen Inhalt. Wenn jedoch ein Zustand gefunden wurde, auf welchen die oben beschriebenen Kriterien zutreffen, wird dieser aus der Menge gelöscht. Nur so kann immer der möglichst schnellste Siegesweg für den weißen Spieler sichergestellt werden.

Vor jeder Überprüfung wird eine weitere Kopie der Menge erstellt, welche für den betrachtenden Spieler relevant ist. Im ersten Schritt wird nun also `fullStates` kopiert und in `_states` abgespeichert. Nun wird jeder `state` in `_states` betrachtet. Zunächst wird überprüft, ob der Zustand mögliche Mühlen für den weißen Spieler beinhaltet: - Ist dies nicht der Fall wird die weitere Betrachtung des Zustandes übersprungen, weil so kein Sieg in einem Schritt für weiß möglich ist. Dies funktioniert nur für das 3-3 Endspiel, weil die Spieler springen dürfen und sich nicht einbauen können. - Ist dies der Fall, werden alle nächstmöglichen Zustände generiert. Wird unter diesen einer

gefunden welcher zum Sieg von weiß führt, wird der Ausgangszustand zusammen mit dem Zielzustand in der Datenbank abgespeichert. Da für den Ausgangszustand nun der optimale Gewinnweg gefunden wurde, wird dieser aus der `fullStates` Menge gelöscht. weil er nicht erneut betrachtet werden muss. Des weiteren wird der Ausgangszustand in eine neue Menge `fullStep` gespeichert, welche für den 2. Schritt von `fillDb` notwendig ist.

Der zweite Teil der `fillDb`-Funktion besteht aus einer Schleife, welche so lange ausgeführt wird, bis für alle Zustände in `fullStates` ein idealer Lösungsweg gefunden wurde. Da dieser Fall äußerst unwahrscheinlich ist, wird die Schleife auch abgebrochen, wenn keine neuen Zustände für eine der beiden Unterschritte *2A* oder *2B* gefunden wurden.

In ersten Abschnitt der Schleife wird überprüft, ob ein Spielzug durch den schwarzen Spieler unweigerlich zu einem Sieg von schwarz führt. Dafür muss überprüft werden, ob alle möglichen Folgezustände `nextStates` des aktuell betrachteten Zustands `state` eine Teilmenge der Menge der Zustände ist, in denen weiß auf jeden Fall gewinnt. Um zu gewährleisten, dass auch alle Folgezustände vertreten sein können, müssen zunächst alle Symmetrien der Zustände in der Menge `fullStep` rekonstriert werden und werden in die Menge `enrichedFullStep` gespeichert. Wenn ein Zustand gefunden wurde, auf welchen die obige Beschreibung zutrifft, wird dieser der `halfStep` Menge hinzugefügt. Außerdem wird er aus der `halfStates` Menge entfernt, da bewiesen ist, dass er einen Sieg von weiß nicht verhindern kann.

Wenn keine `halfStep`s gefunden werden, wird die Suche nach neuen Zuständen abgebrochen.

Für die Suche nach Gewinnspielzügen für weiß ist es auch wieder notwendig, die `halfStep`s um die Symmetrien zu erweitern. Das Ergebnis wird in `enrichedHalfStep` gespeichert.

Nun wird für jeden noch vorhanden Zustand in `fullStates` überprüft, ob einer der möglichen Folgezustände zu einem Zustand in `halfStep` führt. Dieser wird zusammen mit dem passenden Folgezustand und der Information in wie vielen Zügen weiß maximal gewinnt in die Endspieldatenbank gespeichert. Außerdem wird der Ausgangszustand wieder der `fullStep` Menge hinzugefügt und aus der `fullStates` Menge gelöscht.

Wenn keine neuen `fullStep`s gefunden werden, wird die Suche nach neuen Zuständen abgebrochen.

Zusätzlich zu erwähnen ist, dass `halfStep` nach jedem Durchlauf geleert wird. Dies dient der Performance, da für diese Zielzustände bereits Ausgangszustände gesucht wurden. `fullStep` hingegen kann nicht geleert werden. Dies liegt darin begründet, dass bei Zuständen in denen der schwarze Spieler am Zug ist, es Folgezustände geben kann, bei welchen der weiße Spieler in einem Zug gewinnen kann, sowie Folgezustände in denen der weiße Spieler erst nach fünf Zügen gewinnen kann. Somit vergößert sich die Menge, von welchen die Folgezustände eine Teilmenge sein müssen, mit jedem berechneten Spielzug. Das bedeutet auch für den weißen Spieler, dass die Information über die Anzahl der Spielzüge bis zum Sieg immer eine Maximalangabe ist. Es sind lediglich so viele Spielzüge notwendig, wenn der schwarze Spieler *optimal* spielt.

[ ]:
```
def fillDb(states):
    fullStep = []
    stepCount = 0

    stepCount += 1

    fullStates = states.copy()
```

```
    halfStates = states.copy()
    _states = fullStates.copy()

    for state in _states:
        _, board = state
        if (findPossibleMills(board, 'w')) == set():
            continue;
        _nextStates = nextStates(state, 'w')
        for ns in _nextStates:
            if (finished(ns, 'w') and utility(ns, 'w') == 1):
                entry = { "state": state, "nextState": ns, "steps": stepCount }
                retroCol.insert_one(entry)
                fullStep.append(state)
                fullStates.remove(state)
                break

    while len(fullStates) > 0:
        enrichedFullStep = set()
        for s in fullStep:
            enrichedFullStep |= findSymmetries(s)

        _states = halfStates.copy()
        halfStep = []
        stepCount += 1
        print('current step-depth: ' + str(stepCount))

        for state in _states:
            _nextStates = nextStates(state, 'b')
            if (_nextStates.issubset(enrichedFullStep)):
                halfStep.append(state)
                halfStates.remove(state)

        if len(halfStep) == 0:
            break

        enrichedHalfStep = set()
        for s in halfStep:
            enrichedHalfStep |= findSymmetries(s)

        _states = fullStates.copy()

        stepCount += 1
        newStatesCount = 0
        print('current step-depth: ' + str(stepCount))

        for state in _states:
            _nextStates = nextStates(state, 'w')
```

```
            for ns in _nextStates:
                if (ns in enrichedHalfStep):
                    entry = { "state": state, "nextState": ns, "steps":␣
↪stepCount }
                    retroCol.insert_one(entry)
                    fullStep.append(state)
                    fullStates.remove(state)
                    newStatesCount += 1
                    break

        if (newStatesCount == 0):
            break
```

Die Funktion `createEndgameDatabase` führt alle notwendigen Funktionen aus, um die Endspieldatenbank zu generieren und zu befüllen.

Das Ausführen der Funktion ist standardmäßig auskommentiert. Wenn die Endspieldatenbank erstellt werden soll muss in Zeile 7 des folgenden Codeblocks die Kommentierung entfernt werden.

[ ]:
```
def createEndgameDatabase():
    generateBoards()
    global boards
    states = getSymmetryUnique(boards)
    fillDb(states)

# createEndgameDatabase()
```

3.4 Datenbankabfrage

Für die Verwendung der Datenbank sind zunächst einige Hilfsunktionen notwendig. Anschließend wird die Funktion `findNextStateInDb` zur Abfrage des nächsten Zuges eines Zustands in der Datenbank implementiert.

3.4.1 Hilfsfunktionen

Die Funktion `swapPlayer` invertiert einen Spieler. Die Funktion hat ein Argument

- `player` ist ein Spieler (`'w'`, `'b'` oder `' '`).

Es gibt genau drei Möglichkeiten für die Funktion:

- `swapPlayer('w')` ⇒ `'b'`
- `swapPlayer('b')` ⇒ `'w'`
- `swapPlayer(' ')` ⇒ `' '`

[ ]:
```
def swapPlayer(player):
    if player == 'w':
        return 'b'
    if player == 'b':
        return 'w'
```

```
    return ' '
```

Die Funktion `swapPlayers` invertiert das gesamte Spielbrett eines Zustands und hat ein Argument

- `state` ist ein Zustand eines Spiels.

Mit Hilfe der Funktion `swapPlayer` tauscht sie jeden Spielstein mit der anderen Farbe, leere Felder bleiben leer. Zurückgegeben wird der neue Zustand.

[ ]:
```
def swapPlayers(state):
     return (
         state[0],
         tuple(
            tuple(
                swapPlayer(state[1][ir][ic])
                for ic in range(0, 8)
            ) for ir in range(0, 3)
         )
    )
```

Die Funktion `convertStateListToTuple` wandelt einen Zustand als Liste in einen Zustand als Tupel um. Sie hat ein Argument

- `state` ist ein Zustand.

Hintergrund ist, dass MongoDB keine Tupel kennt und deswegen den Zustand als Liste zurückgibt. Mit Hilfe der Funktion kann der Zustand aber zurück in ein Tupel konvertiert werden.

[ ]:
```
def convertStateListToTuple(state):
    return (
        (
            state[0][0],
            state[0][1]
        ),
        tuple(
            tuple(
                state[1][ir][ic]
                for ic in range(0, 8)
            ) for ir in range(0, 3)
        )
    )
```

3.4.2 Finden des nächsten Zustands

Die Funktion `findNextStateInDb` dient zum Abfragen eines nächsten Zuges für einen Spieler. Die Funktion hat zwei Argumente

- `state` ist ein Zustand;
- `player` ist ein Spieler (`'w'`, `'b'`).

Die Funktion gibt ein Zwei-Tupel zurück mit

- dem nächsten Spielzug und
- der Anzahl der benötigen Halbzüge bis zum Sieg (bei perfektem Spiel des Gegners).

Die Funktion fragt mit Hilfe der Funktion `getUniqueStateForSymmetry` einen symmetrischen Zustand ab, der den höchsten Streuwert hat. Somit ist sichergestellt, dass immer der richtige Zustand abgefragt ist (der, der auch in der Datenbank gespeichert ist). Gleichzeitig kann man die zu speichernde Anzahl an Zuständen reduzieren.

Da die Datenbank nicht den angefragten Zustand zurückgibt, muss für alle möglichen nächsten Zustände geschaut werden, ob er auch in den Symmetrien des zurückgegebenen nächsten Zustands der Datenbank ist. Wenn das der Fall ist, kann der Zustand zurückgegeben werden.

Da in der Endspieldatenbank alle Fälle für weiß gespeichert sind, muss für den schwarzen Spieler der Zustand am Anfang und Ende der Funktion invertiert werden. Dies passiert mit Hilfe der Funktion `swapPlayers`.

[ ]:
```
def findNextStateInDb(state, player):
    if player == 'b':
        state = swapPlayers(state)
    symmetryState = getUniqueStateForSymmetry(state)
    stateInDb = retroCol.find_one({ 'state': symmetryState })
    if stateInDb is None:
        return None
    symmetryNextState = convertStateListToTuple(stateInDb['nextState'])
    possibleNextStates = nextStates(state, 'w')
    nextState = None
    for pns in possibleNextStates:
        if symmetryNextState in findSymmetries(pns):
            nextState = pns
    if player == 'b':
        nextState =  swapPlayers(nextState)
    return nextState, stateInDb['steps']
```

3.5 Integration in die Studienarbeit

Damit die Endspieldatenbank auch in der Studienarbeit angewendet werden kann, wird die Klasse `AlphaBetaPruning` Im Notebook `nmm-alpha-beta-pruning.ipynb` minimal angepasst.

Als erstes erhält der Konstruktor der Klasse einen Parameter `useEndgameDatabase`, der standardmäßig auf `False` gesetzt ist. Dieser Parameter wird im Konstruktur auf die Variable `useEndgameDatabase` gesetzt.

Zum anderen wurde die Funktion `bestMoves` folgendermaßen erweitert

```
def bestMoves(self, state, player):
    if self.useEndgameDatabase:
        if countStones(state, 'w') == 3 and countStones(state, 'b') == 3:
            newState = findNextStateInDb(state, player)
            if newState is not None:
                newState, steps = newState
                return BestMoves(
```

```
                [newState],
                1,
                {
                    'win_in(database)': steps
                }
            )

    # ... nicht veränderte Implementierung ...
```

Bevor Alpha-Beta-Pruning sich selbst die besten Züge berechnet, wird überprüft ob die Endspieldatenbank benutzt werden soll und ob beide Spieler nur noch drei Steine auf dem Brett haben. Ist dies der Fall, wird mit Hilfe der oben implementierten Funktion `findNextStateInDb` überprüft, ob ein Datenbankeintrag für den Zustand existiert.

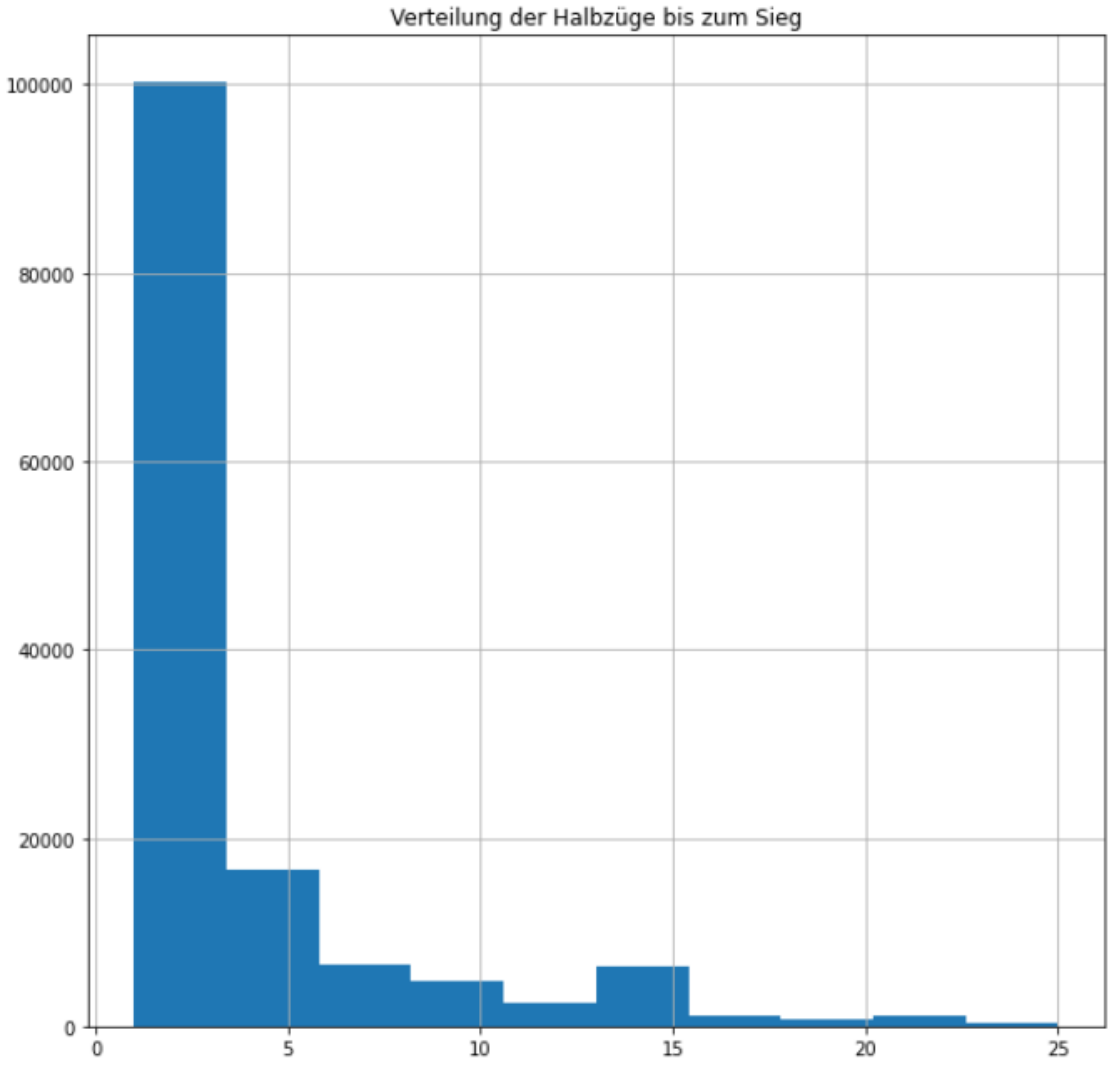
4 Retrograde Analyse

Im Folgenden soll die retrograde Analyse für das 3-3 Endspiel von Mühle durchgeführt werden. Zu erst sollen allgemeine Beobachtungen über das Endspiel aufgestellt werde. Anschließend soll die Verwendung der Endspieldatenbank getestet werden.

4.1 Beobachtungen

Die Datenbank enthält für die das 3-3 Endspiel exakt 140.621 Einträge. Diese Einträge sind alles gewonnene Zustände. Diese Anzahl stimmt genau mit der von Lemmerich und Späth in ihrer Studienarbeit überein [LS05]. Somit kann davon ausgegangen werden, dass alle gewonnen Zustände im Endspiel abgebildet sind. Damit können bei einem perfekten Spiel $82,9\%$ der eindeutigen Zustände (im Sinne der Symmetrien) gewonnen werden. Insgesamt hat Mühle auf 3-3 Brett 2.691.920 Zustände, die von insgesamt 169.626 eindeutigen Zuständen abgebildet werden können.

Die höchste Anzahl an Halbzügen in dem 3-3 Endspiel, die garantiert zu einem Sieg führen, sind 25 Halbzüge. Also kann der Spieler in 13 Zügen gewinnen, selbst wenn der Gegner perfekt spielt. Bei der Verteilung der Halbzüge fällt auf, dass es mehr Zustände mit 15 Zügen gibt, als mit 13 Zügen zum Sieg.



4.2 Turnier

Um die Endspieldatenbank unter realen Umständen testen zu lassen, soll sie in einem Turnier mit verschiedenen Konfigurationen von Alpha-Beta-Pruning antreten. Dafür wird die Implementierung des Turniers der Studienarbeit (implementiert in `nmm-tournament.ipynb`) verwendet.

Insgesamt sollen drei künstliche Intelligenzen gegen eine KI mit der Endspieldatenbank antreten:

- [**MM**]: Minimax mit einer maximalen Suchtiefe von zwei und der Standardheuristik;
- [**AB5**]: Alpha-Beta-Pruning mit maximal 5000 Zuständen und
- [**AB25**]: Alpha-Beta-Pruning mit maximal 25000 Zuständen.

Die KI mit der Verwendung der Endspieldatenbank hat folgende Konfiguration:

- [**DB**]: Alpha-Beta-Pruning mit maximal 25000 Zuständen und der Verwendung der Endspieldatenbank.

Alle drei Alpha-Beta-Pruning Algorithmen verwenden zum besseren Vergleichen die selbe Heuristik. Diese Heuristik wurde als beste Heuristik in der Studienarbeit befunden.

Insgesamt spielt jeder Algorithmus sechs mal gegen den Algorithmus mit der Endspieldatenbank, dreimal als weiß und dreimal als schwarz.

4.2.1 Ergebnisse

Die Ergebnisse der Turniere sind in der folgenden Tabelle notiert. Die Logs der Spielveräufes sind im Ordner `jupyter-rounds-retrograde-analysis` zu finden. Dabei bezeichnen die Siege/Niederlagen immer aus Sicht von [DB].

	[MM]	[AB5]	[AB25]
[DB] gewonnen	3	2	3
[DB] verloren	0	2	3
unentschieden	3	2	0

Die Ergebnisse zeigen, dass die Endspieldatenbank keinen wirklichen Einfluss auf die Spielverläufe gehabt haben. Die Ergebnisse sind ähnlich denen aus der Studienarbeit. Dies ist damit zu eklären, dass die Datenbank nur die 3-3 Spiele beinhaltet. Ein Spiel ist aber meistens schon davor beendet, es kommt also nie zu der Anwendung der Datenbank. Dieses Verhalten wird durch die Turniere bestätigt. Alle Spiele sind vorher beendet worden.

5 Fazit

Als Ergebnis der Hausarbeit wurde eine Endspieldatenbank implementiert, die alle gewonnen Spielsituationen bei 3-3 Steinen auf dem Brett enthält. Die Datenbank wurde erfolgreich in die vorangegangene Studienarbeit eingebunden. Dabei wurde die Implementierung der Datenbankerzeugung und -nutzung ausführlich beschrieben.

In der retrograden Analyse wurde festgestellt, dass in dem 3-3 Endspiel von Mühle Spieler schon 25 Halbzüge im Vorraus ihren Sieg sichern können. Dann kann selbst das perfekte Spiel des Gegners den Sieg nicht mehr verhindern. Außerdem wurde gezeigt, dass bei einem perfekten Spiel $82,9\%$ der Spiele gewonnen werden können.

Die Turniere haben gezeigt, dass die Endspieldatenbank keinen wirklichen Nutzen im Spiel hat, weil die Spiele in den seltesten Fällen zu einem 3-3 Spiel werden. Dieses Verhalten kann man entgegenwirken, indem man die Datenbank auf mehr Spielvarianten erweitert.

Falls man die Endspieldatenbank in Aktion sehen möchte, kann man folgenden Code in dem Notebook `nmm-gui.ipynb` ausführen. Dieser erstellt ein Spiel gegen Alpha-Beta-Pruning, der in maximal 25 Halbzügen verloren ist.

```
s1 = ((0, 0),
 (('b', 'b', ' ', 'w', ' ', 'b', 'w', ' '),
  (' ', ' ', ' ', ' ', ' ', 'w', ' ', ' '),
  (' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')))

gameState = GameState(state=s1,algorithm1 = AlphaBetaPruning(useEndgameDatabase=True))

gameState.canvas
```

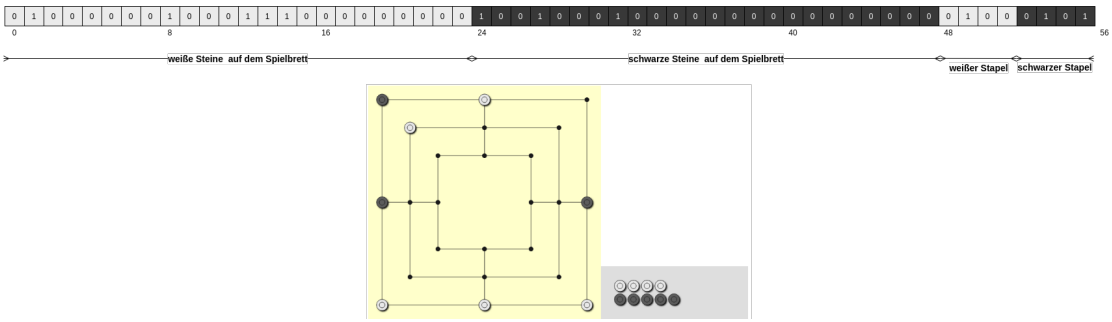
In der Hausarbeit wurde auf die Speicherung von Remi- und Verluststellungen verzichtet. Dies hat zum einen den Grund des Speicher- und Rechenbedarfs der Tupelschreibweise. Zum Anderen wird dies in der Studienarbeit nicht benötigt. Wenn sich der Spieler in einer verlorenen Situation befindet, ist es irrelevant, in wie vielen Zügen er verloren hat.

Zum Abschluss der Hausarbeit soll noch eine Verbesserungsmöglichkeit besprochen werden, die den Nutzen der Datenbank wesentlich verbessern würde.

5.1 Verbesserungsmöglichkeit

Das größte Problem dieser Implementierung ist wahrscheinlich, wie Zustände gespeichert werden. In der Hausarbeit (und auch in der Studienarbeit) werden Zustände als Tupel gespeichert. Das Problem hier ist, das die Tupelschreibweise sehr viel Speicher und auch mehr Performance beim Berechnen benötigt. So kommt man bereits bei der 3-3-Endspieldatenbank an Grenzen moderner Heimcomputer (vorallem wegem dem Arbeitsspeicher). Speichert man die Zustände anders, beispielsweise als Bitboards wie Gasser in seiner Lösung des Mühlespiels, müsste man das Spiel mit der heutigen Technik problemlos lösen können (Gasser hat es bereits 1996 erreicht).

Als Alternative für die Speicherung als Bits wird von Lemmerich und Späth [LS05] folgende Möglichkeit genannt. Man speichert die einzelnen Positionen der Steine von weiß und schwarz von links oben entlang des Rings und dann den zweiten und dritten Ring. Hat der Spieler auf der Position einen Stein, so ist der Bit auf `1`, andernfalls `0`. Dafür bräuchte man für jeden Spieler 24 Bit und nochmal je vier Bit für den Stapel. In der folgenden Abbildung ist diese Variante dargestellt und zur Veranschaulichung noch ein passendes Spielfeld.



Diese Variante würde theoretisch 56 Bits für jeden Zustand benötigen, also 7 Byte. In Python würde diese große Zahl aber 32 Byte benötigen:

[ ]:
```
import sys
print(sys.getsizeof(0b01000000100011100000000010010001000000000000000000100),␣
 ↪'Bytes')
```

Dagegen benötigt ein Zustand in der Tupelnotation 424 Bytes (`s0` ist der Startzustand, definiert in dem Notebook *nmm-game.ipynb*):

[ ]:
```
print(sys.getsizeof(s0) + sys.getsizeof(s0[0]) + sum([sys.getsizeof(ring) for␣
 ↪ring in s0[1]]), 'Bytes')
```

Somit verbraucht ein Zustand in der Tupelschreibweise fast 400 Bytes mehr als in der Bitschreibweise. Dies wirkt sich ins Besondere bei sehr vielen Zuständen aus, die bei der Generierung der

Endspieldatenbank auch benötigt werden. Auch performance-technisch sollte eine Implementierung mit Bits wesentlich schneller sein als die verwendete Implementierung. So müssen die Funktionen bei fast jeder Operation über die Tupel in Tupel in Tupel iterieren.

Als Nachteil für die Bitschreibweise könnte man die Lesbarkeit anmerken. Es ist wesentlich Entwicklerfreundlicher mit den Tupeln zu arbeiten als Bitoperationen durchzuführen.

Eine Neuimplementierung mit Bits wäre für die Hausarbeit sicherlich sinnvoll, da man so auch weitere Endspiele betrachten kann. Aber für die Neuimplementierung müsste man jede (Hilfs)funktion neu implementieren, was den Rahmen der Hausarbeit gesprengt hätte.