DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Entwicklung einer Künstlichen Intelligenz für das Brett-Spiel „Mühle“

Studienarbeit

im Studiengang Angewandte Informatik

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Luisa Fuss und Julian Retzbach

05.05.2021

Bearbeitungszeitraum	24.09.2020 – 05.05.2021
Matrikelnummer, Kurs	5136331 und 9057038, TINF18AI2
Ausbildungsfirma	Roche Diagnostics GmbH
	VS Vereinigte Spezialmöbelfabriken GmbH & Co. KG
Betreuer	Prof. Dr. Karl Stroetmann

Ehrenwörtliche Erklärung

Wir versichern hiermit, dass wir die vorliegende Arbeit mit dem Thema

Entwicklung einer Künstlichen Intelligenz für das Brett-Spiel „Mühle“

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet haben.

Ort, Datum Luisa Fuss und Julian Retzbach

Anmerkung

Aus Gründen der besseren Lesbarkeit wird in dieser Arbeit für alle personenbezogenen Begriffe nur die männliche Sprachform verwendet. Sämtliche Personenbezeichnungen gelten gleichermaßen für jedes Geschlecht.

Inhaltsverzeichnis

1 Einleitung

Mühle ist ein Brettspiel für zwei Spieler, deren Ziel es ist, durch das Positionieren von jeweils neun Spielsteinen eine sogenannte "Mühle" zu bilden. Dabei befinden sich drei Steine eines Spielers in einer Reihe, um so die Steine des Gegners entfernen zu dürfen. Gewonnen hat, wer alle gegnerischen Steine bis auf zwei geschlagen hat, oder den Gegner so zubaut, dass dieser nicht mehr ziehen kann.

1.1 Aufgabenstellung

Das Ziel dieser Arbeit ist die Implementierung eines Mühle-Spiels mit Hilfe der Programmiersprache Python. Dabei soll es möglich sein, in einem Jupyter Notebook mit Hilfe einer Künstlichen Intelligenz (KI) gegen einen Computer zu spielen. Die KI verwendet dabei die Algorithmen Minimax und Alpha-Beta-Pruning und nutzt Transpositions-Tabellen. Die grafische Oberfläche soll mit Hilfe von ipywidgets und ipycanvas gestaltet werden. Die Künstliche Intelligenz sollte sinnvolle Spielzüge wählen, und es so dem menschlichen Spieler möglichst schwer machen, zu gewinnen. Die Implementierung ist an das Skript von Professor Dr. Stroetmann angelehnt.

1.2 Struktur

Das Mühle-Spiel kann über das Notebook `Muehle_Game.ipynb` gestartet werden.

Zum Lesen und Verstehen der Anwendung ist folgende Reihenfolge zu empfehlen:

1. **Einführung**
2. **Mühle** - In diesem Kapitel werden die Spielregeln und Eigenschaften des Brettspiels erläutert.
3. `Muehle_GUI.ipynb` - Hier befindet sich die Implementierung der Graphischen Nutzeroberfläche, die dem menschlichen Spieler die Kommunikation und Interaktion erleichtert, sowie die Steuerung des Spielablaufs.
4. `Muehle_Utilities.ipynb` - Die in diesem Abschnitt enthaltenen Hilfsfunktionen werden an unterschiedlichen Stellen verwendet und erleichtern die Implementierung der restlichen Komponenten.
5. `Muehle_Logic.ipynb` - In diesem Kapitel wird die Logik des Mühlespiels abgebildet. Dazu gehört die Ermittlung aller möglichen nächsten Zustände, die ausgehend von einem Zustand in einem Zug zu erreichen sind.
6. `Muehle_Heuristic.ipynb` - Eine Heuristik, wie sie in diesem Kapitel beschrieben ist, bewertet einen Zustand aus der Sicht eines Spielers und ermittelt so seine Gewinnwahrscheinlichkeit.

7. `Muehle_Algo.ipynb` - In diesem Kapitel werden die beiden Algorithmen implementiert, die die Künstliche Intelligenz darstellen.
8. **Fazit** - Abschließend werden die Ergebnisse bewertet und die Algorithmen Minimax und Alpha-Beta-Pruning hinsichtlich ihrer Spielstärke verglichen. Außerdem sollen mögliche Weiterentwicklungen diskutiert werden.

1.3 Einrichten der Entwicklungsumgebung

1.3.1 erstes Mal

Zunächst sollte das Projekt geklont werden. Das GitHub Repository ist unter https://github.com/luisafuss/muehle-studienarbeit zu finden. Darin enthalten ist die Datei `environment.yml`, die genutzt werden kann, um eine virtuelle Umgebung inklusive aller benötigen Libraries zu erhalten.

Um das Environment zu erstellen, muss im Projektordner über die Anaconda Prompt folgender Befehl aufgerufen werden:

```
conda env create -f environment.yml
```

Alternativ steht auch die Datei `requirements.txt` zur Verfügung.

1.3.2 jedes Mal

Die eingerichtete Umgebung kann mit Hilfe des folgenden Befehls aktiviert werden:

```
conda activate muehle
```

Als Nächstes kann Jupyter notebook mit folgedenm dem Befehl gestartet werden:

```
jupyter notebook
```

1.3.3 Beenden

Um den Kernel herunterzufahren, kann der Shortcut `Crtl + c` in der Anaconda Prompt genutzt werden.

Zum Verlassen des Environments dient der Befehl

```
conda deactivate
```

1.4 Mühle

Mühle ist ein sogenanntes Nullsummenspiel mit perfekter Information. Unter Nullsummenspiel versteht man in der Spieltheorie ein Spiel, bei dem der Gewinn des einen Spielers, den Verlust des anderen bedeutet (vgl. Holler, S.55). Formal bedeutet dies:

$$\forall state \in Terminalstates : utility(state, playerOne) + utility(state, playerTwo) = 0$$

wobei gilt:

$$utility(state, player) = \begin{cases} -1 & \texttt{falls player verliert bei Zustand state} \\ 0 & \texttt{falls Unentschieden} \\ 1 & \texttt{falls player gewinnt bei Zustand state.} \end{cases}$$

Ein Spiel mit perfekter Information ist laut Walter Schlee ein Spiel, bei dem jedem Spieler die getroffenen Entscheidungen des Gegners bekannt sind. Dies ist bei Mühle der Fall, da ersichtlich ist, welcher Stein von wo nach wo bewegt oder wohin er gesetzt wurde.

Quellen:

- Manfred J. Holler, Gerhard Illing: Einführung in die Spieltheorie. 7. Auflage. Springer, Berlin u. a. 2009, ISBN 978-3-540-69372-7
- Walter Schlee, Einführung in die Spieltheorie: mit Beispielen und Aufgaben, ISBN 3-528-03214-6, S. 95

1.5 Anzahl möglicher Zustände

Zunächst soll die Anzahl möglicher Zustände N berechnet werden, um die Größenordnung des Spiels besser einschätzen zu können. Ein Mühlebrett besitzt 24 Positionen, auf denen Steine platziert werden können. Jede Position kann drei Zustände annehmen:

$$ZustandPosition = \begin{cases} 0 & \texttt{falls kein Stein auf der Position ist} \\ 1 & \texttt{falls ein Stein des Spielers 1 auf der Position ist} \\ 2 & \texttt{falls ein Stein des Spielers 2 auf der Position ist.} \end{cases}$$

Damit ergibt sich

$$N = 3^{24} = 282.429.536.481$$

als obere Schranke für die Anzahl der zu betrachtenden Zustände.

Beachtet man jedoch, dass sich höchstens neun Steine eines Spielers auf dem Spielfeld befinden, lässt sich die Anzahl der Zustände N weiter einschränken. Es gilt:

- 0 bis 9 weiße Steine auf dem Feld
- 0 bis 9 schwarze Steine auf dem Feld
- (24 - Anzahl Weiß - Anzahl Schwarz) leere Felder

Daraus folgt:

1. Möglichkeiten „Weiß“ zu platzieren = $\binom{24}{\texttt{weiß}}$
2. Möglichkeiten „Schwarz“ zu platzieren = $\binom{24-\texttt{weiß}}{\texttt{schwarz}}$

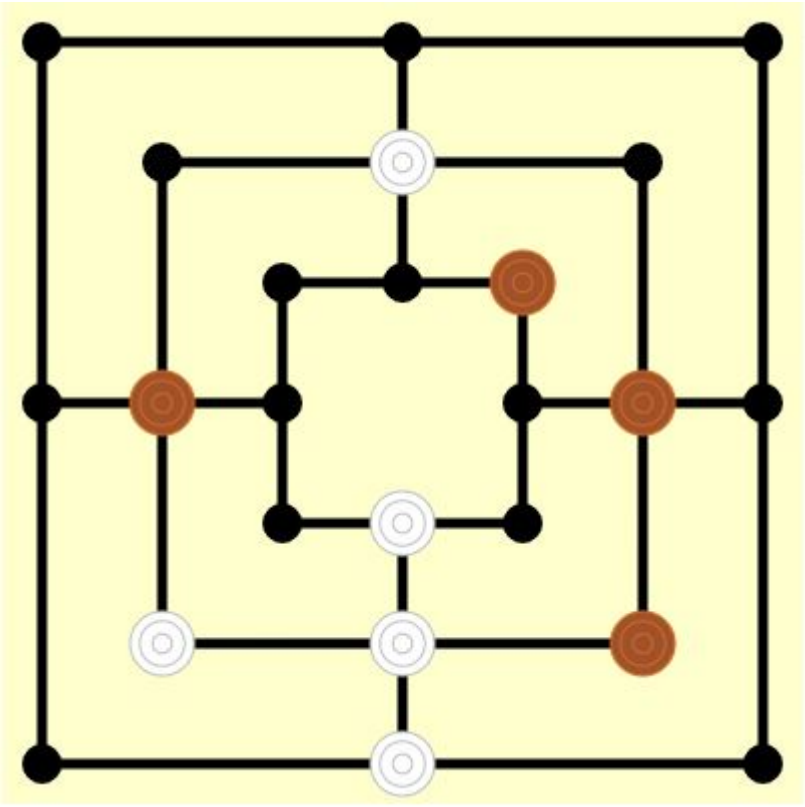
Um die Anzahl der Zustände zu erhalten müssen diese Werte multipliziert werden. Die Gesamtanzahl aller Zustände ergibt sich aus der Summe aller Zustände bei weiß $\in$ {0, 1, ..., 9} und schwarz $\in$ {0, 1, ..., 9} Markierungen. Das bedeutet:

$$\texttt{Anzahl Zustände} = \sum_{\texttt{weiß}=0}^{9} \sum_{\texttt{schwarz}=0}^{9} \binom{24}{\texttt{weiß}} \cdot \binom{24-\texttt{weiß}}{\texttt{schwarz}}$$

$$\texttt{Anzahl Zustände} = 143.122.694.691$$

Zu beachten ist hier, dass dies auch nur eine Abschätzung ist, da es praktisch zum Beispiel nicht möglich ist, dass ein Spieler neun Steine und der andere keine Steine auf dem Spielfeld hat. Noch weiter lässt sich die Anzahl der Zustände N einschränken, wenn man beachtet, dass eine Drehung des Spielfelds um 90°, 180° oder 270° im Grunde den gleichen Spielzustand darstellt. Auch die Spiegelungen an der waagrechten und senkrechten Achse sowie an den beiden Diagonalen stellen Symmetrien dar, die die Anzahl der zu betrachtenden Zustände N verringert. Auch die Inversion des Spielfeldes von innen nach außen und andersherum, kann bei korrekter Bearbeitung zu einer niedrigeren Anzahl der Zustände N führen. Diese Symmetrien werden im Rahmen der Implementierung jedoch nicht beachtet, weshalb an dieser Stelle die exakte Berechnung der Anzahl der Zustände N für diese Fälle nicht erläutert werden soll.

1.6 Spielbrett und Spielsteine



Ein typisches Mühlebrett besteht aus drei Quadraten, die durch vier Linien verbunden sind. Die Quadrate werden als Innenring, Mittelring und Außenring bezeichnet. Jeder Spieler hat neun Spielsteine, meist runde Holzscheiben, in weiß und schwarz/braun. Diese können auf eine von 24 Positionen gesetzt werde. Die Positionen ergeben sich aus Kreuzungen von zwei Linien. So entstehen 12 Eckpunkte an den Ecken der Quadrate, vier Kreuzpunkte im Mittelring, und acht T-Punkte im Innen- bzw. Außenring. Zu Beginn des Spiels befinden sich keine Spielsteine auf dem Spielbrett.

1.7 Spielablauf

Basierend auf den Regeln des Welt-Mühle-Dachverbands (WMD) besteht das Spiel aus drei aufeinanderfolgenden Spielphasen:

- Setzphase: Die Spieler setzen einen ihrer Steine auf eine beliebige leere Position.
- Zugphase: Die Spieler ziehen einen ihrer Spielsteine von einer Position auf eine benachbarte leere Position
- Sprungphase: Die Spieler ziehen einen ihrer Spielsteine von einer Position auf eine beliebige leere Position

1.8 Spielregeln

Die Spieler sind abwechselnd an der Reihe und befinden sich zunächst in der Setzphase, bis alle neun Steine platziert sind. Anschließend sind sie solange in der Zugphase, bis sie nur noch drei Steine besitzen und damit in die Sprungphase wechseln.

Wer eine Mühle schließt (drei Steine waagrecht oder senkrecht nebeneinander), darf einen beliebigen Stein des Gegners entfernen, solange sich dieser nicht in einer Mühle befindet. Hat der Gegner ausschließlich in einer Mühle befindliche Steine, so dürfen auch diese entfernt werden. Werden mit einem Zug zwei Mühlen geschlossen, dürfen auch zwei gegnerische Steine entfernt werden.

Kann ein Spieler in der Zugphase keinen seiner Steine bewegen, da alle benachbarten Felder belegt sind, so hat er verloren. Hat ein Spieler weniger als drei Steine, hat er ebenfalls verloren. Wurde 30 Runden lang keine Mühle geschlossen, endet das Spiel mit einem Unentschieden.

Die offiziellen Turnierregeln des WMD sind hier zu finden:

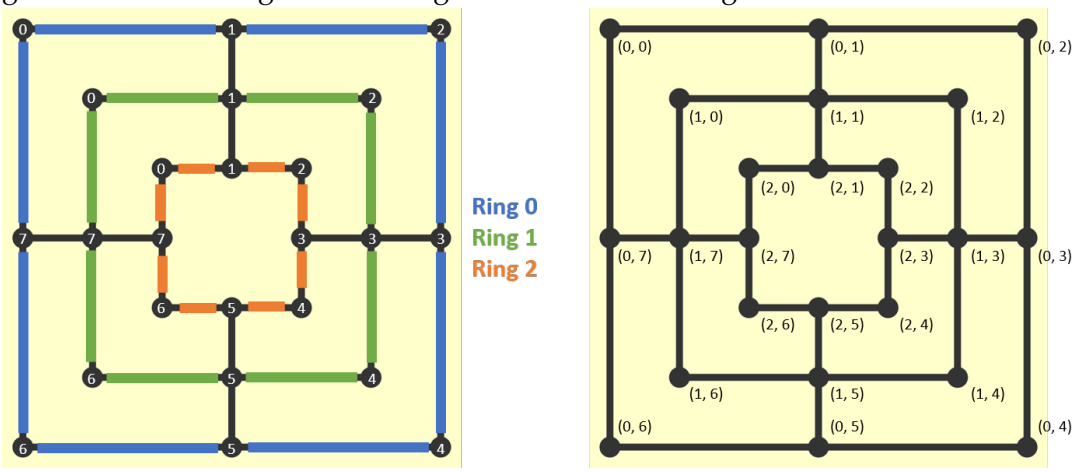
http://www.muehlespiel.eu/images/pdf/WMD_Turnierreglement.pdf.

2 GUI - Graphical User Interface / Benutzeroberfläche

In diesem Abschnitt soll die graphische Oberfläche des Spiels entwickelt werden. Dies beinhaltet die Interaktion des Spielers mit dem Spielaufbau.

2.1 Mögliche Positionen

Die Grafik unten zeigt die Notation für die Position der Spielsteine. Die Positionen werden als Tupel bestehend aus Ringnummer und Zellnummer beschrieben. Die Ringe, in der Abbildung farbig markiert, sind dabei von außen beginnend mit 0, 1 und 2 nummeriert. Für die Bezeichnung der Zellen wird in jedem Ring oben links mit 0 begonnen und im Uhrzeigersinn bis 7 weitergezählt. So ergibt sich eine eindeutige Notation der Form `(ring, cell)`.



2.2 Definitionen

Zunächst werden einige Definitionen für die gesamte Implementation festgesetzt:

- `ring` = eines der 3 Quadrate (Wertebereich: 0-2)
- `cell` = ein Punkt auf einem Ring (Wertebereich: 0-7)
- `position` = Tupel: (ring, cell)
- `canvas` = Liste aus 3 Listen, die angeben, ob, und wenn ja welcher Stein dort sitzt. Dabei symbolisiert jede Liste einen Ring. In jeder Liste sind neun Werte. Je einer für eine Zelle im Ring. Eine 0 sagt dabei aus, dass an der entsprechenden Position kein Stein ist. Eine 1 oder 2 stehen für einen Stein des zugehörigen Spielers.

- `remaining` = [Anzahl noch nicht gesetzter Steine Spieler 1, Anzahl noch nicht gesetzte Steine Spieler 2]
- `state` = [remaining, board]

2.3 Importe

Zur Visualisierung des Spielbrettes wird ipycanvas verwendet. Die Statusanzeigen sind Widgets der ipywidget-Library.

[ ]:
```
%run ./Muehle_Logic.ipynb
%run ./Muehle_Algo.ipynb
import ipycanvas
from ipycanvas import MultiCanvas

import ipywidgets as widgets
from ipywidgets import interact, interactive, fixed, interact_manual

from IPython.display import clear_output
from pathlib import Path
import time
```

2.4 Konstanten

Um den Quellcode besser les- und wartbar zu machen, werden einige Konstanten definiert. Für das Aussehen des Spielfelds:

- `BOARD_SIZE` = Größe des Spielfeldes in Pixeln
- `DOT_RADIUS` = Radius der schwarzen kleinen Punkte, die mögliche Positionen markieren (in Abhängigkeit von der Spielfeldgröße)
- `PIECE_RADIUS` = Radius der Spielsteine
- `COLOUR` = Farben der [dots, pieces_player_1, pieces_player_2]
- `COLOUR_HINT` = Farbe, in der mögliche Zielfelder markiert werden
- `COLOUR_OPPONENT` = Farbe, in der entfernbare Steine markiert werden
- `PADDING` = relativer Abstand des äußersten Quadrats zum Spielfeldrand
- `DISTANCE` = relativer Abstand zwischen den Quadraten des Spielfelds
- `TRANSPARENCY_DEFAULT` = Standardwert für die Transparenz

- `TRANSPARENCY_HINT` = Transparenz der Kreise möglicher Zielfelder und entfernbarer Steine

Für die Funktion des Spielfelds:

- `PLAYER` = Spieler
- `PLAYER` = Spielsteinfarben der Spieler
- `START_STATE` = Startzustand des Spielfelds, beide Spieler haben noch keinen Stein gesetzt und das Spielfeld ist leer
- `MAX_MOVES_WITHOUT_MILL` = Anzahl der erlaubten Züge, ohne dass eine Mühle entsteht. Ist das Limit überschritten, endet das Spiel unentschieden

[ ]:
```
# Aussehen
BOARD_SIZE      = 400
DOT_RADIUS      = BOARD_SIZE * 0.025
PIECE_RADIUS    = BOARD_SIZE * 0.04
COLOUR          = ['black', 'white', 'sienna']
COLOUR_HINT     = 'green'
COLOUR_OPPONENT = 'red'
PADDING         = 0.05
DISTANCE        = 0.15
TRANSPARENCY_DEFAULT = 1.0
TRANSPARENCY_HINT    = 0.5

# Spiel
PLAYER          = [1, 2]
PLAYER_COLOUR   = ['Weiß', 'Braun']
START_STATE     = [[9, 9], [          # Anzahl zu setzender Steine␣
 ↪(Spieler_1 (Weiß), Spieler_2 (Braun))
                 [0, 0, 0, 0, 0, 0, 0, 0], # ring 0
                 [0, 0, 0, 0, 0, 0, 0, 0], # ring 1
                 [0, 0, 0, 0, 0, 0, 0, 0]  # ring 2
                 ]]
MAX_MOVES_WITHOUT_MILL = 30
```

Die Funktion `row()` liefert einen relativen Wert für die Position der eingegebenen Reihe. `col()` macht dies für die Spalten. Der Eingabeparameter `number` gibt dabei an, die wievielte Zeile von oben bzw. die wievielte Spalte von links ausgegeben werden soll. Der Rückgabewert ist eine rationale Zahl zwischen 0,05 und 0,95. Multipliziert mit der Spielfeldbreite bzw. -höhe ergibt sich die absolute Position der Ecken und Schnittpunkte der Spielfeldlinien.

```
[ ]: #Reihe von oben nach unten (0-6)
def row(number):
    return PADDING + DISTANCE * number
#Spalte von links nach rechts (0-6)
def col(number):
    return row(number)
```

`POSITIONS` beschreibt die möglichen Punkte, an denen Steine sitzen können. Der Aufruf der Positionen erfolgt gemäß der oben vorgestellten Notation im Format `POSITIONS[ring][cell]`.

```
[ ]: POSITIONS = [([col(0), row(0)], [col(3), row(0)], [col(6), row(0)], [col(6),␣
 ↪row(3)], [col(6), row(6)], [col(3), row(6)], [col(0), row(6)], [col(0),␣
 ↪row(3)]), #ring 0
             ([col(1), row(1)], [col(3), row(1)], [col(5), row(1)], [col(5),␣
 ↪row(3)], [col(5), row(5)], [col(3), row(5)], [col(1), row(5)], [col(1),␣
 ↪row(3)]), #ring 1
             ([col(2), row(2)], [col(3), row(2)], [col(4), row(2)], [col(4),␣
 ↪row(3)], [col(4), row(4)], [col(3), row(4)], [col(2), row(4)], [col(2),␣
 ↪row(3)])] #ring 2
```

2.5 Zeichenfunktionen

`init_canvas()` erstellt den Aufbau des Spielfeldes. Dies erfolgt mit Hilfe von ipycanvas. Das Spielbrett (`canvas`) besteht aus drei übereinanderliegenden Ebenen. Ebene 0 ist der Hintergrund, ein beiges Quadrat. Ebene 1 beinhaltet die schwarzen Striche und Punkte auf dem Spielfeld. Diese setzen sich aus drei schwarzen Quadraten, vier horizontalen bzw. waagrechten Linien und 24 Punkten an den Kreuzungen der Linien, die die möglichen Positionen der Steine markieren, zusammen. Dabei wird bei der Erstellung mit relativen Positionsangaben gearbeitet, um das Spielfeld einfach skalieren zu können. In der Ebene 2 liegen schließlich die Spielsteine.

```
[ ]: def init_canvas():
    #canvas[Hintergrund, Linien, Steine]
    canvas = MultiCanvas(3, width = BOARD_SIZE, height = BOARD_SIZE)

    # Hintergrund
    canvas[0].fill_style = '#ffffcc'
    canvas[0].fill_rect(0, 0, BOARD_SIZE)

    # Strichstärke
    canvas[1].line_width = 5
```

```
    # Quadrate
    canvas[1].stroke_rect(BOARD_SIZE * col(0), BOARD_SIZE * row(0),␣
↪BOARD_SIZE * (1 - row(0) - col(0))) #ring 0
    canvas[1].stroke_rect(BOARD_SIZE * col(1), BOARD_SIZE * row(1),␣
↪BOARD_SIZE * (1 - row(1) - col(1))) #ring 1
    canvas[1].stroke_rect(BOARD_SIZE * col(2), BOARD_SIZE * row(2),␣
↪BOARD_SIZE * (1 - row(2) - col(2))) #ring 2

    # Mittelinien
    canvas[1].begin_path()
    canvas[1].move_to(BOARD_SIZE * col(3), BOARD_SIZE * row(0)) #oben
    canvas[1].line_to(BOARD_SIZE * col(3), BOARD_SIZE * row(2))
    canvas[1].move_to(BOARD_SIZE * col(6), BOARD_SIZE * row(3)) #rechts
    canvas[1].line_to(BOARD_SIZE * col(4), BOARD_SIZE * row(3))
    canvas[1].move_to(BOARD_SIZE * col(3), BOARD_SIZE * row(6)) #unten
    canvas[1].line_to(BOARD_SIZE * col(3), BOARD_SIZE * row(4))
    canvas[1].move_to(BOARD_SIZE * col(0), BOARD_SIZE * row(3)) #links
    canvas[1].line_to(BOARD_SIZE * col(2), BOARD_SIZE * row(3))
    canvas[1].stroke()

    # Punkte (außen, mitte, innen)
    for ring in POSITIONS:
        for x, y in ring:
            canvas[1].fill_arc(BOARD_SIZE * x, BOARD_SIZE * y, DOT_RADIUS,␣
↪0, 360)
    return canvas
```

Die Funktion `draw_piece()` zeichnet an einer zu übergebenen Position einen Spielstein für den eingegebenen Spieler. Dieser besteht aus einem mit der Spielerfarbe gefüllten Kreis als Hintergrund und zwei Innenringen um die Steine realistischer aussehen zu lassen.

[ ]:
```
def draw_piece(canvas, ring, cell, player):
    #Hintergrund
    canvas[2].fill_style = COLOUR[player]
    canvas[2].fill_arc(BOARD_SIZE * POSITIONS[ring][cell][0], BOARD_SIZE *␣
↪POSITIONS[ring][cell][1], PIECE_RADIUS, 0, 360)

    #Ringe
    canvas[2].stroke_style = 'silver' if player == 1 else 'chocolate'

    canvas[2].stroke_arc(BOARD_SIZE * POSITIONS[ring][cell][0], BOARD_SIZE *␣
↪POSITIONS[ring][cell][1], PIECE_RADIUS,       0, 360)
```

```
    canvas[2].stroke_arc(BOARD_SIZE * POSITIONS[ring][cell][0], BOARD_SIZE *␣
↪POSITIONS[ring][cell][1], PIECE_RADIUS * 0.7, 0, 360)
    canvas[2].stroke_arc(BOARD_SIZE * POSITIONS[ring][cell][0], BOARD_SIZE *␣
↪POSITIONS[ring][cell][1], PIECE_RADIUS * 0.3, 0, 360)
```

Mit Hilfe von `highlight_positions()` werden alle Positionen aus einer Liste mit einem transparenten Kreis markiert. Falls keine andere Farbe übergeben wird, wird die in den Konstanten definierte default-Farbe verwendet. Diese Funktion kann verwendet werden, um die schlagbaren Steine des Gegners oder die möglichen Zugpositionen zu bestimmen. Wenn `just_inner` True ist, dann wird nur das innere des Steins markiert. Dies wird verwendet, um den zuletzt bewegten Stein zu markieren.

[ ]:
```
def highlight_positions(canvas, highlight_positions, colour = COLOUR_HINT,␣
↪just_inner = False):
    with ipycanvas.hold_canvas(canvas):
        for (ring, cell) in highlight_positions:
            canvas[2].fill_style = colour
            canvas[2].global_alpha = TRANSPARENCY_HINT
            radius = PIECE_RADIUS if not just_inner else PIECE_RADIUS * 0.3
            canvas[2].fill_arc(BOARD_SIZE * POSITIONS[ring][cell][0],␣
↪BOARD_SIZE * POSITIONS[ring][cell][1], radius, 0, 360)
```

Die Funktion `show_start_button()` zeichnet auf dem Spielfeld ein Rechteck, dass das Wort "Start" enthält und optisch einem Button ähnelt.

[ ]:
```
def show_start_button(canvas):
    with ipycanvas.hold_canvas(canvas):
        # Hintergrund
        canvas[2].clear()
        canvas[2].fill_style = 'black'
        canvas[2].global_alpha = TRANSPARENCY_HINT
        canvas[2].fill_rect(0, 0, BOARD_SIZE, BOARD_SIZE)

        # Button
        canvas[2].global_alpha = TRANSPARENCY_DEFAULT
        canvas[2].shadow_color = 'black'
        canvas[2].shadow_offset_x = 5
        canvas[2].shadow_offset_y = 5
        canvas[2].shadow_blur = 5
        canvas[2].fill_style = 'silver'
        canvas[2].fill_rect(BOARD_SIZE/3, (BOARD_SIZE/5)*2, BOARD_SIZE/3,␣
↪height=BOARD_SIZE/5)
```

```
        canvas[2].shadow_offset_x = 0
        canvas[2].shadow_offset_y = 0
        canvas[2].shadow_blur = 0

        canvas[2].stroke_style = 'black'
        canvas[2].stroke_rect(BOARD_SIZE/3, (BOARD_SIZE/5)*2, BOARD_SIZE/3,␣
↪height=BOARD_SIZE/5)

        # Text
        canvas[2].fill_style = 'black'
        canvas[2].font = '30px Arial'
        canvas[2].fill_text('Start', BOARD_SIZE/2 *0.83, BOARD_SIZE/2*1.05)
```

`update_canvas()` zeichnet die obere Ebene des Spielbretts nach jeder Veränderung neu. So werden Positionsänderungen oder Hinweise zu möglichen Zielfeldern und schlagbaren Steinen sichtbar. Zusätzlich wird hier die Aktualisierung der Statusanzeige angestoßen. Durch den optionalen Parameter `message` ist es möglich die Statusausgabe zu überschreiben. Falls das Spiel gerade pausiert ist, wird außerdem ein Start-Button eingeblendet.

[ ]:
```
def update_canvas(game, message = ''):
    game.save_game()
    gamemode_side_by_side = widgets.HBox([gamemode_label, gamemode_toggle])
    one_below_other       = widgets.VBox([gamemode_side_by_side, turn_label,␣
↪status_label, mill_label, pieces_status_label, pieces_player_one,␣
↪pieces_player_two,\
                                          cache_label, time_label,␣
↪value_label, restart_button])
    side_by_side          = widgets.HBox([game.canvas, one_below_other])
    display(side_by_side)

    with ipycanvas.hold_canvas(game.canvas):
        game.canvas[2].clear()
        game.canvas[2].global_alpha = TRANSPARENCY_DEFAULT
        for ring in range(3):
            for cell in range(8):
                value = game.state[1][ring][cell]
                if value in PLAYER: draw_piece(game.canvas, ring, cell,␣
↪value)
        update_status_widgets(game, message)
    if game.pause == True:
        show_start_button(game.canvas)
```

2.6 Die Klasse Game - Status des Spielfelds

Der Status speichert die aktuelle Situation des Spielfeldes. Die Klasse beinhaltet mehrere Variablen. `state` gibt wie in der Definition beschrieben an, ob und wenn ja welcher Stein sich an einer Position auf dem Spielfeld befindet. Dabei gilt:

- 0 = kein Stein
- 1 = weißer Stein
- 2 = brauner Stein

Außerdem beinhaltet die Variable Angaben über die Steine, die die beiden Spieler noch setzen können. `current_player` gibt an, welcher Spieler, 1 oder 2, gerade am Zug ist. `canvas` ist ein ipycanvas-Objekt, das mittels `init_canvas()` erzeugt wird. Es handelt sich dabei um die visuelle Ausgabe des Spielfeldes. In `number_pieces_to_remove` wird gespeichert, wie viele Steine der aktuelle Spieler aufgrund von Mühlen noch entfernen darf. `selected_piece` ist ein Tupel, das die Position eines ausgewählten Steins speichert. Dieses Tupel hat die Form `(ring, cell)`. Dies ist notwendig für Zug- und Springvorgänge. Ist die Variable ungleich `None`, befindet sich der Spieler gerade in einer Zug- oder Sprungaktion. Ist die Variable belegt, so hat der Spieler bereits den zu versetzenden Stein ausgewählt und muss noch auf das Zielfeld klicken. Die Position ist in Phase 2 des Spiels relevant, um valide Zielfelder zu ermitteln. In der Phase 1 kann es vorkommen, dass kein Stein gesetzt wird, weil kein passendes Feld ausgewählt wurde, um zu verhindern, dass der Spieler trotzdem wechselt, wird die Variable `do_not_change` auf True gesetzt und diese vor Spielerwechsel abgefragt. Das Gleiche geschieht in den Phasen 2 und 3, falls kein passender Stein ausgewählt wird. `winner` ist None, solange das Spiel noch läuft. Ansonsten gilt:

- winner = 0: Unentschieden
- winner = 1: Spieler 1 hat gewonnen
- winner = 2: Spieler 2 hat gewonnen

Die Variable `pause` ist nur direkt nach der Initialisierung `True`. Beim ersten Klicken auf das Spielfeld, dem Starten des Spiels, wird sie auf `False` gesetzt. Dies ist notwendig, um die Berechnung des nächsten Zustands anzustoßen, falls der erste Zug durch einen Such-Algorithmus erfolgt. Ein Spieler kann entweder eine menschliche Person oder ein Such-Algorithmus sein. Als Algorithmen sind Minimax und Alpha-Beta-Pruning implementiert. Standardmäßig sind Spieler reale Personen. Um was es sich handelt, ist in den Variablen `mode_player_one` und `mode_player_two` gespeichert.

Die `__init__()` Funktion setzt die Werte der Game-Klasse. Dabei kann falls notwendig der aktuelle Spieler übergeben werden, falls beispielsweise Spieler 2 starten soll. Standardmäßig wird als Zustand der Startzustand übergeben, wobei davon zuvor eine Deepcopy angefertigt werden muss. Dies ist notwendig, da ansonsten die Referenz auf `START_STATE` übergeben wird und die Konstante somit im Laufe des Spiel verändert werden würde. Außerdem

kann angegeben werden, dass die Spieler statt manuell zu spielen einen Such-Algorithmus zur Bestimmung des nächsten Spiel-Zustands verwenden. Hierfür existieren die Variablen `mode_player_one` und `mode_player_two` die die gewünschte Option für Spieler 1 bzw. 2 angeben und folgende Werte annehmen können:

- None = Der Spieler spielt manuell
- 'minimax' = Der Spieler nutzt das Suchverfahren Minimax
- 'alpha-beta' = Der Spieler nutzt das Suchverfahren Alpha-Beta-Pruning

Nach der Initialisierung der Variablen wird noch festgelegt, dass bei einem Klick auf das Spielfeld die klasseneigene Funktion `play_game()` aufgerufen wird. Diese wird gleich noch definiert.

Mit Hilfe der Funktion `__str__()` kann der Status später mit dem Befehl `print(Game)` formatiert ausgegeben werden. Dies ist besonders zum Überprüfen des aktuellen Spielstandes hilfreich.

Die Variable `savegame` ist eine Liste, in der der Spielverlauf in Form der aufeinander folgenden Zustände abgelegt wird. Initiiert wird diese Liste deshalb mit dem Startzustand.

Die Variable `saved` ist ein Indikator dafür, ob das aktuelle Spiel bereits gesichert wurde.

[ ]:
```
class Game():
    def __init__(self, current_player = 1, mode_player_one = None,␣
↪mode_player_two = None):
        self.state          = copy.deepcopy(START_STATE)
        self.current_player = current_player
        self.canvas         = init_canvas()
        self.number_pieces_to_remove = 0
        self.selected_piece = (None, None)
        self.do_not_change  = False
        self.winner         = None
        self.pause          = True
        self.moves_without_mill      = 0
        self.savegame       = [copy.deepcopy(START_STATE)]
        self.saved          = False

        self.mode_player_one = mode_player_one
        self.mode_player_two = mode_player_two

        self.canvas[2].on_mouse_down(self.play_game)

    def __str__(self):
        return "state: " + str(self.state) \
```

```
            + "\nmode player 1: " + str(self.mode_player_one) \
            + "\nmode player 2: " + str(self.mode_player_two) \
            + "\ncurrent player: " + str(self.current_player) \
            + "\nnumber pieces to remove: " + str(self.
↪number_pieces_to_remove) \
            + "\nselected piece: " + str(self.selected_piece) \
            + "\ndo not change: " + str(self.do_not_change) \
            + "\nwinner: " + str(self.winner) \
            + "\npause: " + str(self.pause) \
            + "\nmoves without mill: " + str(self.moves_without_mill)
```

Die Funktion `play_game()` der Klasse `Game` steuert das Spiel. Sie wird später aufgerufen, wenn der Spieler irgendwo hin klickt. Wenn das Spiel noch nicht vorbei ist, wird überprüft, ob das Spiel gerade pausiert ist. Das ist beim ersten Klicken der Fall. Ist dem so, wird die Spielmodusauswahl deaktiviert und überprüft, ob ein Algorithmus als erstes dran ist. Ist dem so, führt dieser seinen Zug durch. Ansonsten wird das Spielbrett lediglich aktualisiert, um dem manuellen Spieler zu sagen, was er zu tun hat. Bei jedem weiteren Klick wird überprüft, ob dieser an einer Position, also im Bereich um einen schwarzen Punkt, erfolgt ist. Wenn nein, wird gegebenenfalls die Auswahl eines Steins zurückgesetzt, um einen Wechsel des zu setzenden Steins zu ermöglichen. Wenn ja, kann eine der folgenden Situationen eintreten, wobei die erste aufgelistete Situation Priorität hat:

- **Es müssen noch Steine entfernt werden** - Aufgrund von Mühlen die zuvor geschlossen wurden, dürfen noch Steine entfernt werden. Der zuvor geklickte Stein wird der Funktion `remove_piece()` übergeben, die ihn vom Spielfeld löscht.
- **Der Spieler befindet sich in Phase 1** - Auf die ausgewählte Position wird mit Hilfe von `place_piece()` ein Stein gesetzt.
- **Es wurde zuvor kein Stein ausgewählt** - Da sich der Spieler wie zuvor überprüft nicht in Phase 1 und somit in der Zug- oder Springphase befindet, und bisher kein Startfeld ausgewählt wurde, kann dies nun geschehen. Dazu wird die Funktion `select_piece()` aufgerufen.
- **Es wurde ein Stein ausgewählt** - Da sich der Spieler in der Zug- oder Springphase befindet, handelt es sich bei dem ausgewählten Feld um ein Zielfeld, auf das der zuvor ausgewählte Stein mittels `move_piece()` bewegt wird.

Falls nach abarbeiten der Situationen noch Steine zu entfernen sind, werden diese markiert. Außerdem wird überprüft, ob das Spiel zu Ende ist. Falls sich der Spieler nicht in einem Zwischenzustand von Phase 2 oder 3 (Startfeld ausgewählt, Zielfeld noch nicht bekannt) befindet, keine Steine mehr entfernt werden müssen, das Spiel nicht zu Ende ist und vorher alle korrekt gelaufen ist, ist der nächste Spieler an der Reihe und das Spielfeld wird aktualisiert. `algo_step()` wird aufgerufen, um zu überprüfen, ob der nächste Zug ein Zug der KI ist, der gegebenenfalls ausgeführt werden muss.

[ ]:
```
def play_game(self, x, y):
    self.check_if_finished()
    if self.winner != None: return

    if self.pause:
        self.pause = False
        gamemode_toggle.disabled = True
        update_canvas(self)
        if (self.current_player == 1 and self.mode_player_one != None) or␣
↪(self.current_player == 2 and self.mode_player_two != None):
            self.algo_step()
        return

    old_state = copy.deepcopy(self.state)
    phase = get_player_phase(self.state, self.current_player)
    piece = self.get_clicked_piece(x, y)

    # Situationen überprüfen
    if piece == None:
        if self.number_pieces_to_remove == 0:
            self.selected_piece = (None, None) # Steinauswahl zurücksetzen
            update_canvas(self)
        return
    if self.number_pieces_to_remove > 0:
        self.remove_piece(piece)
    elif phase == 1:
        self.place_piece(piece)
    elif self.selected_piece != (None, None):
        ring, cell = piece
        if self.state[1][ring][cell] == self.current_player: # Steinauswahl␣
↪wechseln
            self.select_piece(piece)
            return
        else:
            self.move_piece(piece)
    elif phase == 2 or phase == 3:
        self.select_piece(piece)
    else:
        change_status_label('Du befindest Dich anscheinend in keiner␣
↪gültigen Spielphase. Irgendwas ist schief gegangen... Sorry!')

    if self.number_pieces_to_remove > 0:
```

```
        update_canvas(self)
        highlight_positions(self.canvas, get_opponent_beatable_pieces(self.
↪state[1], self.current_player), colour = COLOUR_OPPONENT)

    self.check_if_finished()

    if self.selected_piece == (None, None) and not self.
↪number_pieces_to_remove > 0 and self.winner == None and not self.
↪do_not_change:
        moved_piece = get_moved_piece(old_state[1], self.state[1], self.
↪current_player)
        self.current_player = opponent(self.current_player) #␣
↪Spielerwechsel, wenn nicht noch eine Aktion ausgeführt werden muss
        self.moves_without_mill += 1
        self.check_if_finished()
        update_canvas(self)
        highlight_positions(self.canvas, [moved_piece], "blue", just_inner =␣
↪True)

    self.algo_step()

Game.play_game = play_game
del play_game
```

Die Funktion `get_clicked_piece()` der Klasse `Game` überprüft, ob die übergebenen x- und y-Koordinaten auf einer validen Position, also einem Spielfeldpunkt liegen. Wenn ja, wird diese Position in der Form `(ring, cell)` zurückgegeben, ansonsten `None`. Zur Ermittlung wird über alle möglichen Positionen iteriert und geprüft, ob die Klick-Koordinaten in einem Pufferbereich rund um den Mittelpunkt der Position liegen. Ist dies für beide Koordinaten der Fall, wurde der angeklickte Punkt gefunden.

[ ]:
```
def get_clicked_piece(self, x, y):
    piece = None
    for ring in POSITIONS:
        for pos in ring:
            check_x = pos[0] * BOARD_SIZE - PIECE_RADIUS < x < pos[0] *␣
↪BOARD_SIZE + PIECE_RADIUS
            check_y = pos[1] * BOARD_SIZE - PIECE_RADIUS < y < pos[1] *␣
↪BOARD_SIZE + PIECE_RADIUS
            if check_x and check_y:
                piece = (POSITIONS.index(ring), ring.index(pos))
    return piece
```

```
Game.get_clicked_piece = get_clicked_piece
del get_clicked_piece
```

In Phase 1 wird durch `place_piece()` der Klasse `Game` überprüft, ob das angeklickte Feld noch frei ist. Wenn ja, wird dort ein Stein des Spielers positioniert und die eigene Steinanzahl in `remaining` um 1 reduziert. Um zu überprüfen, ob bei der Aktion neue Mühlen entstanden sind, wird vor und nach dem Setzen des Steins alle Mühlen gesucht. Anschließend wird geprüft, ob und wenn ja, wie viele, neue Mühlen sich in der Menge der Mühlen nach dem Setzen des Steins befinden. Dies ist notwendig, um zu ermitteln, wie viele Steine des Gegners entfernt werden dürfen.

[ ]:
```
def place_piece(self, piece):
    remaining, board = self.state
    if piece in empty_positions(board):
        self.do_not_change = False
        ring, cell = piece
        mills_before = find_mills(board, self.current_player)
        board[ring][cell] = self.current_player
        remaining[self.current_player - 1] -= 1
        mills_after = find_mills(board, self.current_player)
        self.number_pieces_to_remove = count_new_mills(mills_before,␣
↪mills_after)
        update_canvas(self)
    else:
        self.do_not_change = True
        change_status_label('Du kannst Deinen Stein nur auf ein leeres Feld␣
↪setzen. Bitte probiere es erneut!')
Game.place_piece = place_piece
del place_piece
```

`remove_piece()` der Klasse `Game` entfernt den übergebenen Stein vom Spielfeld, falls es erlaubt ist, diesen zu entfernen. Dazu werden alle schlagbaren Steine des Gegners ermittelt und überprüft, ob der angeklickte Stein dazu gehört. Ist dies der Fall, wird der Stein entfernt, indem an dieser Stelle eine `0` eingetragen wird. Andernfalls wird eine Fehlermeldung geworfen und der Spieler kann einen anderen Stein zum Entfernen auswählen. Falls noch mehr Steine entfernt werden müssen, werden die möglichen Positionen am Ende markiert.

[ ]:
```
def remove_piece(self, piece):
    _, board = self.state
    ring, cell = piece
    beatable_pieces = get_opponent_beatable_pieces(board, self.
↪current_player)
    if piece in beatable_pieces:
```

```
            board[ring][cell] = 0
            self.number_pieces_to_remove -= 1
            update_canvas(self)
        else:
            change_status_label('Du kannst diesen Stein nicht entfernen. Bitte␣
↪probiere es erneut!')
        if self.number_pieces_to_remove > 0:
            highlight_positions(self.canvas, get_opponent_beatable_pieces(board,␣
↪self.current_player), colour = COLOUR_OPPONENT)

        self.moves_without_mill = -1
Game.remove_piece = remove_piece
del remove_piece
```

`move_piece()` der Klasse `Game` bewegt den zuvor ausgewählten Stein, falls zulässig, auf die übergebene Position. Das Startfeld ist dabei in der Variable `selected_piece` gespeichert. Damit das Zielfeld valide ist, muss es in der Liste sein, die durch `next_positions()` ausgehend vom ausgewählten Stein ermittelt werden. Ist dies der Fall, wird das alte Feld auf `0` gesetzt und das Zielfeld mit dem Wert des aktuellen Spielers belegt. Um zu überprüfen, ob bei der Aktion neue Mühlen entstanden sind, wird vor und nach dem Versetzen des Steins alle Mühlen gesucht. Anschließend wird geprüft, ob und wenn ja, wie viele, neue Mühlen sich in der Menge der Mühlen nach dem Bewegen des Steins befinden. Dies ist notwendig, um zu ermitteln, wie viele Steine des Gegners entfernt werden dürfen.

[ ]:
```
def move_piece(self, piece):
    _, board = self.state
    start_ring, start_cell = self.selected_piece
    goal_ring,  goal_cell  = piece
    possible_positions = next_positions(self.state, self.current_player,␣
↪start_ring, start_cell)
    if piece in possible_positions:
        mills_before = find_mills(board, self.current_player)
        board[start_ring][start_cell] = 0
        board[goal_ring] [goal_cell]  = self.current_player
        mills_after = find_mills(board, self.current_player)
        self.number_pieces_to_remove = count_new_mills(mills_before,␣
↪mills_after)
        self.selected_piece = (None, None)
        update_canvas(self)
    else:
        change_status_label('Du kannst Deinen Stein nicht auf dieses Feld␣
↪setzen. Bitte probiere es erneut!')
```

```
Game.move_piece = move_piece
del move_piece
```

Falls sich der Spieler in der Zug- oder Springphase befindet, und noch kein Startfeld ausgewählt wurde, wird `select_piece()` der Klasse `Game` aufgerufen. Diese Funktion überprüft, ob der übergebene Stein ein eigener Stein ist. Wenn ja wird er in der Variable `selected_piece` gespeichert und ausgehend davon mögliche Zielfelder berechnet, um diese hervorheben zu können. Nach erfolgreicher Durchführung befindet sich der Spieler in einem Zwischenzustand, da das Zielfeld noch nicht bekannt ist.

[ ]:
```
def select_piece(self, piece):
    if piece in player_pieces(self.state[1], self.current_player):
        self.do_not_change = False
        self.selected_piece = piece
        change_status_label('Zielort für den eigenen Stein auswählen.')
        ring, cell = piece
        possible_positions = next_positions(self.state, self.current_player,␣
 ↪ring, cell)
        update_canvas(self)
        highlight_positions(self.canvas, possible_positions)
    else:
        self.do_not_change = True
        change_status_label('Das ist nicht Dein Stein. Bitte probiere es␣
 ↪erneut!')
Game.select_piece = select_piece
del select_piece
```

Die Funktion `algo_step()` der Klasse `Game` führt einen Zug aus, falls der aktuelle Spieler ein Algorithmus ist. Als Algorithmen stehen Alpha-Beta-Pruning und Minimax zur Verfügung. Dazu wird zunächst überprüft, ob nicht vielleicht das Spiel schon beendet ist. In diesem Fall würde gar nichts geschehen. Andernfalls wird geprüft, um welchen Algorithmus es sich handelt. Anschließend wird die entsprechende Funktion aufgerufen, die den nächsten Zustand berechnet und ausgibt. Dies ist entweder `alpha_beta_pruning()` oder `minimax()`. Der Zustand des Spiels wird auf den berechneten Zustand gesetzt und es wird geprüft, ob das Spiel mit dem Zug beendet wurde. Der Spieler wechselt und das Spielbrett wird aktualisiert. Falls das Spiel noch nicht beendet ist, wird die Funktion nochmal aufgerufen, da der nächste Spieler ebenfalls ein Such-Algorithmus sein könnte.

[ ]:
```
def algo_step(self):
    start = time.time()
    if self.winner != None: return
    number_opponent_stones_before = count_player_pieces(self.state[1],␣
 ↪opponent(self.current_player))
```

```
    old_state = copy.deepcopy(self.state)
    if (self.current_player == 1 and self.mode_player_one == "alpha-beta")␣
↪or (self.current_player == 2 and self.mode_player_two == "alpha-beta"):
        change_status_label("Der Computer rechnet noch... Ein Moment bitte.")
        value, self.state = alpha_beta_pruning(self.state, self.
↪current_player)
    elif (self.current_player == 1 and self.mode_player_one == "minimax") or␣
↪(self.current_player == 2 and self.mode_player_two == "minimax"):
        change_status_label("Der Computer rechnet noch... Ein Moment bitte.")
        value, self.state = minimax(self.state, self.current_player)
    else:
        return
    number_opponent_stones_after = count_player_pieces(self.state[1],␣
↪opponent(self.current_player))
    if number_opponent_stones_before > number_opponent_stones_after:
        self.moves_without_mill = -1
    self.moves_without_mill += 1

    moved_piece = get_moved_piece(old_state[1], self.state[1], self.
↪current_player)

    self.check_if_finished()
    end = time.time()
    change_time_label(end-start)
    change_value_label(value)
    self.current_player = opponent(self.current_player)
    update_canvas(self)
    highlight_positions(self.canvas, [moved_piece], "blue", just_inner =␣
↪True)
    if self.winner == None: self.algo_step()
Game.algo_step = algo_step
del algo_step
```

`check_if_finished()`prüft, ob das Spiel zu Ende ist und setzt gegebenenfalls den Gewinner. Falls die Maximalanzahl der Züge ohne Mühle erreicht wurde, endet das Spiel unentschieden. Mit Hilfe von `finished()` wird überprüft, ob es sich um einen Terminal-Zustand handelt. Wenn ja, ermittelt `utility()`, ob der aktuelle Spieler oder der Gegner der Gewinner ist.

[ ]:
```
def check_if_finished(self):
    if self.moves_without_mill >= MAX_MOVES_WITHOUT_MILL:
        self.winner = 0
        update_canvas(self)
```

```
    elif finished(self.state, self.current_player):
        util = utility(self.state, self.current_player)
        if   util ==  0:
            self.winner = 0
        elif util ==  1:
            self.winner = self.current_player
        elif util == -1:
            self.winner = opponent(self.current_player)
        update_canvas(self)
Game.check_if_finished = check_if_finished
del check_if_finished
```

Die Methode `save_game()` wird von `update_canvas()` aufgerufen und speichert die Zugfolge in einer Liste. Nach Ende des Spiels wird dieser Spielverlauf als Text-Datei im Ordner "savegames" abgelegt. Falls dieser noch nicht existiert, wird er zunächst erstellt.

[ ]:
```
def save_game(self):
    if self.state != self.savegame[-1]:
        self.savegame.append(copy.deepcopy(self.state))
    if self.winner != None and self.saved == False:
        Path("savegames").mkdir(parents=True, exist_ok=True)
        filename = 'savegames/savegame_' + str(time.time()) + '.txt'
        f = open(filename,'w')
        f.write('Game: ' + str(gamemode_toggle.value)+' | Winner = ' +␣
 ↪str(self.winner)+'\n')
        for state in self.savegame:
            f.write(str(state)+'\n')
        f.close()
        self.saved = True

Game.save_game= save_game
del save_game
```

2.7 Die Statusanzeige - Initialisierung und Funktionen der Status-Widgets und des Restart-Button

Die Methode `restart()` wird durch den restart_button aufgerufen und setzt den Status des Spieles wieder zurück, indem eine neue Instanz der Klasse `Game` erzeugt und das Spielfeld anschließend aktualisiert wird. Außerdem wird die Spielmodusauswahl aktiviert.

```
[ ]: def restart(b):
         gamemode_toggle.disabled = False
         gamemode_toggle.value = 'Spieler vs Spieler'
         global game
         game = Game()
         clear_output(wait = True)
         update_canvas(game)
```

Die Methode `change_gamemode()` ist der Handler für das game_toggle-Widget und ändert den Spielmodus nach dem aktuellen Wert des gamemode_toggle-Widget.

```
[ ]: def change_gamemode(change):
         global game
         gamemode = gamemode_toggle.value
         if gamemode == 'Spieler vs Spieler':
             game.mode_player_one = None
             game.mode_player_two = None
         elif gamemode == 'Spieler vs alpha-beta':
             game.mode_player_one = None
             game.mode_player_two = 'alpha-beta'
         elif gamemode == 'alpha-beta vs Minimax':
             game.mode_player_one = 'alpha-beta'
             game.mode_player_two = 'minimax'
```

An dieser Stelle findet die Initialisierung der unterschiedlichen Statuselemente statt.

- Das `turn_label` ist ein Label, das den Spieler anzeigt, der aktuell an der Reihe ist
- Das `status_label` ist für weitere Statusanzeigen vorgesehen, wie die Aufforderung an den aktuellen Spieler, im Fall einer Mühle, einen Stein des Gegners zu entfernen.
- Das `mill_label` zeigt an, wie viele Züge ohne Mühlen noch möglich sind.
- Das `pieces_status_label` zeigt je nach Spielphase entweder die Anzahl der Steine, die noch gesetzt werden können (Phase 1), oder die noch übrig sind (Phase 2 und 3).
- Das `time_label` zeigt die Rechenzeit des Computers für den letzten Zug an.
- Das `value_label` wird die Bewertung des vom Computer ermittelten Zustands wiedergeben.
- Der `restart_button` ruft die Methode `restart()` auf und setzt somit das Spiel auf den Startzustand zurück
- `pieces_player_one` und `pieces_player_two` zeigen jeweils einen horizontalen Balken, der in Phase 1 nach jedem gesetzten Stein kleiner wird, bis keiner mehr vorhanden ist. In Phase 2 und 3 ist die Auslenkung des Balkens von der Anzahl der Steine des jeweiligen

Spielers abhängig. Somit ist schnell erkennbar, welcher Spieler gerade vorne liegt.

- Die ToggleButtons `gamemode_toggle` ermöglichen die Auswahl des Spielmodus (Player versus Player / Player versus Algorithmus / Algorithmus versus Algorithmus)
- Das `gamemode_label` ist das Label für die ToggleButtons

[ ]:
```
turn_label          = widgets.Label(value = 'X ist an der Reihe')
status_label        = widgets.Label(value = 'Status:')
mill_label          = widgets.Label(value = 'Verbleibende Züge ohne Mühle: ')
pieces_status_label = widgets.Label(value = 'Setzbare Steine:')
cache_label         = widgets.Label(value = 'Zustände im Cache:')
time_label          = widgets.Label(value = 'Rechenzeit Computer:')
value_label         = widgets.Label(value = 'Wert:')

restart_button      = widgets.Button(description = 'Neustart')
restart_button.on_click(restart)

pieces_player_one   = widgets.FloatProgress(value = 9, max = 9, min = 0,␣
 ↪description = PLAYER_COLOUR[0], style={'bar_color': 'silver'})
pieces_player_two   = widgets.FloatProgress(value = 9, max = 9, min = 0,␣
 ↪description = PLAYER_COLOUR[1], style={'bar_color': COLOUR[2]})

gamemode_toggle     = widgets.ToggleButtons(
                        options     = ['Spieler vs Spieler', 'Spieler vs␣
 ↪alpha-beta', 'alpha-beta vs Minimax'],
                        disabled    = False,
                        tooltips    = ['Spieler versus Player', 'Spieler␣
 ↪versus Alpha-Beta-Algorithm', 'Alpha-Beta-Algorithm versus␣
 ↪Minimax-Algorithm']
                    )
gamemode_toggle.observe(change_gamemode, 'value')
gamemode_label      = widgets.Label(value = 'Spielmodus: ')
```

Die Methode `change_status_label()` nimmt einen beliebigen Status-Text entgegen und setzt diesen mit dem Präfix 'Status:' als Text des `status_label`.

[ ]:
```
def change_status_label(message):
    status_label.value = 'Status: ' + message
```

Die Methode `change_time_label()` nimmt die Rechenzeit des Computers entgegen und setzt diesen mit einem Präfix und Suffix als Text des `time_label`.

[ ]:
```
def change_time_label(time):
    time_label.value = 'Rechenzeit Computer: ' + str(round(time, 2)) + '␣
↪Sekunden'
```

Die Methode `change_value_label()` nimmt den von der Heuristik ermittelten Wert des Computers entgegen und setzt diesen mit einem Präfix und Suffix als Text des `value_label`.

[ ]:
```
def change_value_label(value):
    value_label.value = 'Wert: ' + str(round(value, 3))
```

Die Methode `update_status_widgets()` aktualisiert die unterschiedlichen Statusanzeigen passend zur aktuellen Spielphase und den zuletzt getätigten bzw. erwarteten Aktionen.

[ ]:
```
def update_status_widgets(game, message):
    remaining, board = game.state
    player = game.current_player
    turn_label.value  = PLAYER_COLOUR[player - 1] + ' ist an der Reihe.'
    cache_label.value = 'Zustände im Cache: Alpha-Beta: ' +␣
↪str(len(Cache_AB)) + ", Minimax: " + str(len(Cache_Memoize))
    mill_label.value  = 'Verbleibende Züge ohne Mühle: ' +␣
↪str(MAX_MOVES_WITHOUT_MILL - game.moves_without_mill)

    phase = get_player_phase(game.state, player)
    if phase == 1: change_status_label('Bitte einen Stein auf dem Spielfeld␣
↪platzieren.')
    if phase == 2: change_status_label('Bitte einen Stein zum Ziehen␣
↪auswählen.')
    if phase == 3: change_status_label('Bitte einen Stein zum Springen␣
↪auswählen.')

    if game.pause: change_status_label('Bitte oben den Spielmodus auswählen␣
↪und anschließend auf "Start" klicken.')

    if remaining[0] == remaining[1] == 0:
        pieces_status_label.value = 'Verbleibende Steine:'
        pieces_player_one.value = count_player_pieces(board, 1)
        pieces_player_two.value = count_player_pieces(board, 2)
    else:
        pieces_status_label.value = 'Setzbare Steine:'
        pieces_player_one.value = remaining[0]
        pieces_player_two.value = remaining[1]

    if message != '': change_status_label(message)
```

```
    if game.winner in PLAYER: change_status_label(PLAYER_COLOUR[game.winner␣
↪- 1] + ' hat gewonnen!')
    elif game.winner == 0: change_status_label('Unentschieden!')
    if game.number_pieces_to_remove == 1: change_status_label('Mühle! Bitte␣
↪noch 1 Stein des Gegners entfernen.')
    elif game.number_pieces_to_remove > 1:  change_status_label('Mühle!␣
↪Bitte noch ' + str(game.number_pieces_to_remove) + ' Steine des Gegners␣
↪entfernen.')
```

2.8 Spielen

Schließlich kann das Spielfeld erzeugt und ein Spiel gestartet werden. Dabei wird bei jedem Mausklick auf das Spielfeld `play_game()` mit den Koordinaten des Klicks aufgerufen.

[ ]:
```
game = Game()
update_canvas(game)
```

3 Hilfsfunktionen

Im Folgenden sollen Funktionen implementiert werden, die an verschiedenen Stellen in der Entwicklung des Mühlespiels genutzt werden können. Dazu sind sie so einfach wie möglich gehalten.

`count_player_pieces()` zählt die Steine, die ein anzugebender Spieler gerade auf dem Spielfeld hat. Dafür wird über alle möglichen Positionen iteriert und gezählt, bei wie vielen Positionen der Wert der Position mit dem Wert des Spielers übereinstimmt.

[ ]:
```
def count_player_pieces(board, player):
    return [position for ring in board for position in ring].count(player)
```

`empty_positions()` gibt eine Liste mit Tupeln zurück. Jedes Tupel der Form `(ring, cell)` beschreibt eine Stelle auf dem Spielfeld, auf der kein Spielstein steht, also der Wert 0 ist. So erhält man alle leeren Positionen.

[ ]:
```
def empty_positions(board):
    return player_pieces(board, 0)
```

`neighboring_positions()` liefert für eine gegebene Position auf dem Spielfeld alle benachbarten Felder. Die Position wird dabei in der der Form `(ring, cell)` übergeben. Je nachdem wo sich die übergebene Stelle befindet, hat sie 2, 3 oder 4 Nachbarfelder, die es zu ermitteln gilt. Zunächst werden der linke und der rechte Nachbar ermittelt, den jede Position besitzt. Da die Zellen von 0 bis 7 durchnummeriert sind, muss 1 zu dem Zellenwert dazu addiert oder abgezogen werden. Um von 0 zu 7 und zurück zu gelangen, wird das Ergebnis noch Modulo 8 genommen. Die Zellen 1, 3, 5, 7 liegen in der Mitte und haben deshalb noch einen weiteren Nachbar, falls sie in Ring 0 oder 2 sind. Ansonsten haben sie sogar noch zwei weitere Nachbarpositionen.

[ ]:
```
def neighboring_positions(position):
    ring, cell = position

    left_neighbor  = (ring, (cell - 1) % 8)
    right_neighbor = (ring, (cell + 1) % 8)

    positions = [left_neighbor, right_neighbor]

    if cell % 2 == 1:
        if ring in [0, 2] :
            positions.append((1, cell))
        else:
```

```
            positions.append((0, cell))
            positions.append((2, cell))
    return positions
```

`empty_neighbors()` berechnet ausgehend von einem gewählten Stein alle benachbarten leeren Felder. Hierfür wird ein Spielbrett, sowie die Position des ausgewählten Steines mit ring und cell übergeben. Zurückgegeben wird eine Liste der Position.

[ ]:
```
def empty_neighbors(board, ring, cell):
    neighboring = set(neighboring_positions((ring, cell)))
    empty = set(empty_positions(board))
    return neighboring.intersection(empty)
```

`player_pieces()` gibt eine Liste mit allen Positionen eines zu übergebenden Spielers zurück. Jeder Eintrag der Form `(ring, cell)` beschreibt eine Stelle auf dem Spielfeld, auf der ein Spielstein des Spielers steht, also der Wert gleich dem Wert des Spielers ist.

[ ]:
```
def player_pieces(board, player):
    return [(ring, cell) for ring in range(3) for cell in range(8) if␣
 ↪board[ring][cell] == player]
```

`opponent()` gibt den Gegner des übergebenen Spielers zurück. Falls 1 übergeben wird, kommt 2 zurück, ansonsten andersrum.

[ ]:
```
def opponent(player):
    return 3 - player
```

`get_moved_piece` gibt den Stein eines Spielers zurück, der zwischen zwei zu übergebenden Zuständen bewegt wurde.

[ ]:
```
def get_moved_piece(old_board, new_board, player):
    for ring in range(3):
        for cell in range(8):
            if old_board[ring][cell] == 0 and new_board[ring][cell] ==␣
 ↪player:
                return (ring, cell)
    return (None, None)
```

Die Funktion `to_tuple()` wandelt einen übergebenen Spielstatus in Tupelform um.

[ ]:
```
def to_tuple(state):
    return (tuple(state[0]), tuple(tuple(ring) for ring in state[1]))
```

Die Funktion `to_list()` wandelt einen übergebenen Spielstatus in Listenform um.

[ ]:
```
def to_list(state):
    return [list(state[0]), [list(ring) for ring in state[1]]]
```

3.1 Mühlen

`find_mills()` gibt alle Mühlen zurück, die ein Spieler auf dem übergebenen Spielbrett hat. Hierzu werden in der ersten Schleife alle Mühlen ermittelt, die sich auf einem einzigen Ringen befinden. Die zweite Schleife erkennt alle Mühlen, die sich über alle drei Ringe erstrecken. Die Eigenschaft einer Mühle ist, dass drei Steine in gerader Linie direkt nebeneinander liegen. Mühlen werden dabei als Tupel der Form `((ring 1, cell 1), (ring 2, cell 2), (ring 3, cell 3))` gespeichert.

[ ]:
```
def find_mills(board, player):
    mills = set()

    for ring in range (0,3):
        for cell in [0,2,4,6]:
            if(board[ring][cell] == board[ring][cell + 1] ==␣
↪board[ring][(cell + 2) % 8] == player):
                mills.add(((ring, cell), (ring, cell + 1), (ring, (cell + 2)␣
↪% 8)))
    for cell in [1,3,5,7]:
        if(board[0][cell] == board[1][cell] == board[2][cell] == player):
            mills.add(((0, cell), (1, cell), (2, cell)))
    return mills
```

`count_new_mills()` nimmt zwei Mengen von Mühlen und berechnet die Anzahl der Unterschiede zwischen `old_mills` und `new_mills`.

[ ]:
```
def count_new_mills(old_mills, new_mills):
    return len(new_mills.difference(old_mills))
```

`get_opponent_beatable_pieces()` gibt eine Liste mit entfernbaren Steinen des Gegners zurück. Dies ist notwendig, wenn auf Grund des Bildens einer Mühle Steine des Gegners entfernt werden dürfen. Dazu werden alle Steine sowie alle Mühlsteine des Gegners ermittelt. Die schlagbaren Steine sind die Steine des Spielers ohne dessen Mühlsteine. Ist das Ergebnis jedoch leer, werden alle Steine des Gegners zurückgegeben, da laut den offiziellen Regeln in diesem Fall auch Steine aus Mühlen entfernt werden können. Die Arbeit mit Mengen bringt hier keinen Performancevorteil.

[ ]:
```
def get_opponent_beatable_pieces(board, current_player):
    opponent_player = opponent(current_player)
```

```
    opponent_mill_pieces = [(ring, cell) for mill in find_mills(board,␣
↪opponent_player) for (ring, cell) in mill]
    opponent_pieces      = player_pieces(board, opponent_player)
    beatable_pieces      = [piece for piece in opponent_pieces if piece not␣
↪in opponent_mill_pieces]

    return beatable_pieces if len(beatable_pieces) != 0 else opponent_pieces
```

`beat_pieces()` gibt alle möglichen Spielfelder zurück, nachdem eine bestimmte Anzahl an schlagbaren gegnerischen Steinen (`count`) entfernt, also durch den Wert 0 ersetzt, wurde. Die Eingabeparameter sind dazu ein Spielbrett, also der zweite Teile von `state`, eine natürliche Zahl, die der Anzahl der zu entfernenden Steine entspricht, und ein Spieler, der die Steine entfernt. Die Methode funktioniert rekursiv. Als Rekursionsanker dient `count <= 0`. Das Duplikat entfernen wäre schöner, wenn man statt den Listen Mengen verwenden würde. Dazu müsste aber State auf Tupel umgestellt werden, da Lists mutable sind. Wenn noch umgeformt werden muss, bringt das keinen Performancevorteil.

[ ]:
```
def beat_pieces(board, count, player):
    if count <= 0: return [board]

    boards = []
    beatable_pieces = get_opponent_beatable_pieces(board, player)
    for ring, cell in beatable_pieces:
        new_board = copy.deepcopy(board)
        new_board[ring][cell] = 0
        boards.extend(beat_pieces(new_board, count - 1, player))

    # Duplikate entfernen
    result = []
    for board in boards:
        if board not in result:
            result.append(board)
    return result
```

[ ]:
```
%%HTML
<style>
.container { width:100% }
</style>
```

4 Logik

[ ]:
```
%run ./Muehle_Utilities.ipynb
import copy
```

4.1 Spielphasen

Die Funktion `get_player_phase()` prüft in welcher Phase sich der übergebenen Spieler befindet. Hat er noch setzbare Steine, ist also sein Wert in `remaining` größer null, befindet er sich in Phase eins. Ist dies nicht der Fall und auch auf dem Feld befinden sich nur noch drei Steine, so befindet der Spieler sich in der Springphase. Ist dem nicht so, so ist der Spieler in Phase zwei, der Zugphase. Die Eingabeparameter der Funktion sind ein Spielstatus, sowie ein Spieler in der Form 1 für Spieler eins und 2 für Spieler zwei. Zurückgegeben wird die Spielphase des übergebenen Spielers in dem übergebenen Spiel.

[ ]:
```
def get_player_phase(state, player):
    [remaining, board] = state
    if remaining[player - 1] >= 1:
        return 1
    elif count_player_pieces(board, player) == 3:
        return 3
    else:
        return 2
```

`next_positions()` berechnet in welcher Phase sich der Spieler befindet und dementsprechend, wo er als nächstes klicken kann. Falls der Spieler in Phase 1 (Setzphase) oder Phase 3 (Springphase) ist, sind dies unabhängig vom aktuellen Spieler oder in Phase 3 auch unabhängig vom ausgewählten Stein alle leeren Felder. Für Phase 2 wird die zuvor definierte Funktion `empty_neighbors()` aufgerufen. Übergeben wird der Funktion `next_positions()` hierfür ein Spielstatus, ein Spieler, und die Position des ausgewählten Steines (ring und cell).

[ ]:
```
def next_positions(state, player, ring, cell):
    _, board = state
    phase = get_player_phase(state, player)
    return empty_positions(board) if (phase == 1 or phase == 3) else␣
↪empty_neighbors(board, ring, cell)
```

4.2 next states

Im Folgenden werden Funktionen implementiert, die je nach Spielphase alle möglichen nächsten Zustände für einen Spieler bei gegebenem Zustand ausrechnet. Es existieren dementsprechend die Methoden

- next_states_phase_one()
- next_states_phase_two()
- next_states_phase_three()

Alle drei nehmen einen Zustand und einen Spieler und geben eine List möglicher nächster Zustände zurück. Der Ablauf aller drei Methoden ist im Allgemeinen folgendermaßen: 1. Aktuelle Mühlen suchen 2. Steine positionieren/verschieben und entstehende Bretter in einer Liste speichern 3. Für jedes Board überprüfen, ob neue Mühlen dazu gekommen sind und ggf. Steine des Gegners entfernen 4. next_states zusammenbauen und zurückgeben

Für **Schritt 1** wird die Funktion `find_mills()` genutzt. Die Phasen unterscheiden sich im Wesentlichen in **Schritt 2**, weshalb dieser für die einzelnen Phasen gleich genauer betrachtet werden soll. In **Schritt 3** wird über die zuvor entwickelten Zustände iteriert und für jedes Brett zunächst erneut `find_mills()` und damit zusammen mit den zu Beginn ermittelten Mühlen `count_new_mills()` aufgerufen. Letztere Funktion liefert eine natürliche Zahl, die angibt, wie viele neue Mühlen durch den Wechsel in den Zustand entstehen. Falls das Ergebnis 0 ist, geschieht nichts. Falls die Zahl größer 0 ist, wird pro entstandene Mühle einmal die Funktion `beat_pieces()` aufgerufen, die jeweils einen schlagbaren Stein des Gegners entfernt. In diesem Fall wird das berechnete Board also nochmal manipuliert. In **Schritt 4** wird der Zustand, der neben dem Board auch aus der Variable `remaining` besteht, zusammengesetzt.

4.2.1 Phase 1

In Phase 1 wird ein Stein auf ein leeres Feld gesetzt. Die nächsten Zustände entstehen also, in dem über alle leeren Felder iteriert und dort jeweils ein Stein positioniert wird. Dies geschieht als Schritt 2 in `next_states_phase_one()`. Die Besonderheit in der ersten Phase ist, dass in Schritt 4 auch die Variable `remaining` angepasst werden muss, in dem Spieler ein Stein abgezogen wird.

[ ]:
```
def next_states_phase_one(state, player):
    [remaining, board] = state
    boards = []
    # Schritt 1
    mills = find_mills(board, player)

    # Schritt 2
    empty = empty_positions(board)
```

```
    boards_after_placing = []
    for ring, cell in empty:
        new_board = copy.deepcopy(board)
        new_board[ring][cell] = player
        boards_after_placing.append(new_board)

    # Schritt 3
    for new_board in boards_after_placing:
        number_new_mills = count_new_mills(mills, find_mills(new_board,␣
↪player))
        if number_new_mills > 0:
            boards.extend(beat_pieces(new_board, number_new_mills, player))
        else:
            boards.append(new_board)

    # Schritt 4
    remaining = [remaining[0] - 1, remaining[1]] if player == 1 else␣
↪[remaining[0], remaining[1] - 1]

    return [[remaining, new_board] for new_board in boards]
```

4.2.2 Phase 2

Die nächsten Zustände entstehen in der Zugphase durch das Verschieben eines eigenen Steins auf ein benachbartes freies Feld. Im Schritt 2 von `next_states_phase_two()` werden also zunächst alle Spielsteine eines Spielers ermittelt, um anschließend über die iterieren zu können. Für jeden dieser Steine werden dann mit Hilfe von `empty_neighbors()` alle leeren benachbarten Felder ausgerechnet. Pro Stein und freie Nachbar-Position entsteht ein möglicher neuer Zustand indem der Wert der Startposition auf 0 und der Wert der freien Nachbar-Position, also der Zielposition, auf den Wert des Spielers gesetzt wird.

[ ]:
```
def next_states_phase_two(state, player):
    [remaining, board] = state
    boards = []
    # Schritt 1
    mills = find_mills(board, player)

    # Schritt 2
    pieces = player_pieces(board, player)
    boards_after_placing = []
    for ring_start, cell_start in pieces:
        positions = empty_neighbors(board, ring_start, cell_start)
```

```
        for ring_goal, cell_goal in positions:
            new_board = copy.deepcopy(board)
            new_board[ring_start][cell_start] = 0
            new_board[ring_goal][cell_goal] = player
            boards_after_placing.append(new_board)

    # Schritt 3
    for new_board in boards_after_placing:
        number_new_mills = count_new_mills(mills, find_mills(new_board,␣
↪player))
        if number_new_mills > 0:
            boards.extend(beat_pieces(new_board, number_new_mills, player))
        else:
            boards.append(new_board)

    # Schritt 4
    return [[remaining, new_board] for new_board in boards]
```

4.2.3 Phase 3

In Phase 3 kann ein beliebiger eigener Stein auf ein beliebiges freies Feld versetzt werden. Dazu wird in Schritt 2 von `next_states_phase_three()` zunächst nach eigenen Steinen (Startfelder) und leeren Positionen (Zielfelder) gesucht. Anschließende entstehen mögliche Spielbretter aus allen möglichen Kombinationen von Start- und Zielfeldern. Das Startfeld erhält den Wert 0, das Zielfeld den Wert des Spielers.

[ ]:
```
def next_states_phase_three(state, player):
    [remaining, board] = state
    boards = []
    # Schritt 1
    mills = find_mills(board, player)

    # Schritt 2
    pieces = player_pieces(board, player)
    empty = empty_positions(board)
    boards_after_placing = []
    for ring_start, cell_start in pieces:
        for ring_goal, cell_goal in empty:
            new_board = copy.deepcopy(board)
            new_board[ring_start][cell_start] = 0
            new_board[ring_goal][cell_goal] = player
            boards_after_placing.append(new_board)
```

```
    # Schritt 3
    for new_board in boards_after_placing:
        number_new_mills = count_new_mills(mills, find_mills(new_board,␣
↪player))
        if number_new_mills > 0:
            boards.extend(beat_pieces(new_board, number_new_mills, player))
        else:
            boards.append(new_board)

    # Schritt 4
    return [[remaining, new_board] for new_board in boards]
```

`next_states()` nimmt einen Zustand und einen Spieler, berechnet die aktuelle Spielphase und ruft die entsprechende Funktion zur Berechnung der nächsten Zustände, wie sie zuvor definiert wurden, auf.

[ ]:
```
def next_states(state, player):
    phase =  get_player_phase(state, player)
    if phase == 1:
        return next_states_phase_one(state, player)
    elif phase == 2:
        return next_states_phase_two(state, player)
    else:
        return next_states_phase_three(state, player)
```

4.3 Spielende

Die Funktion `finished()` prüft, ob das Spiel für den übergebenen Zustand zu Ende ist, wenn der der übergebene Spieler an der Reihe ist.

Solange ein Spieler noch Steine setzen kann, ist das Spiel noch nicht entschieden. Ansonsten gibt es hier zwei Möglichkeiten, bei denen das Spiel zu Ende ist. Die erste Situation tritt ein, wenn ein Spieler weniger als drei Steine hat. Er hat dann verloren. Bei der zweiten Option verliert ein Spieler, wenn er sich in Phase 2 befindet, aber nicht mehr ziehen kann. Dies kommt vor, wenn alle benachbarten Felder seiner Steine vom Gegenspieler besetzt sind. Prüfen lässt sich dies auch, in dem man berechnet, ob keine validen Folgezustände existieren. Tritt keine der beiden Situationen ein, ist das Spiel laut dieser Funktion noch nicht beendet. Praktisch existieren noch weitere Situationen, in denen ein Spiel beendet ist. Dies ist zum Beispiel der Fall, wenn über zu viele Züge hinweg keine Mühle geschlossen wurde. Dies wird jedoch separat im Notebook `Muehle_GUI.ipynb` ausgewertet.

```
[ ]: def finished(state, current_player):
         [remaining, board] = state
         if any(number_player_pieces > 0 for number_player_pieces in remaining):␣
     ↪return False
         if any(count_player_pieces(board, player) < 3 for player in [1, 2]):␣
     ↪return True
         if len(next_states(state, current_player)) == 0: return True
         return False
```

Die Funktion `utility()` nimmt einen Zustand für den das Spiel beendet ist, also `finished(state, current_player)==True` und einen Spieler und gibt einen numerischen Wert zurück, der folgendermaßen zu interpretieren ist:

- -1 = Der Spieler hat verloren
- 0 = Unentschieden
- 1 = Der Spieler hat gewonnen

Da die Funktion nur mit einem Endzustand aufgerufen wird, muss die Spielerphase nicht beachtet werden. Denn es ist nur möglich in der Zug- und in der Springphase zu gewinnen. Für diese gelten alle unten genannten Regeln.

Falls der Spieler sich nicht in Phase 3 befindet, aber weniger als drei Steine auf dem Brett hat, wird -1 zurückgegeben, falls der Gegner weniger als drei Steine hat wird 1 zurückgegeben. Danach befindet man sich in einer der folgenden Situationen:

- utility = -1, der Spieler hat weniger als 3 Steine → Ergebnis zurückgegeben
- utility = 0, beide Spieler haben mehr als 3 Steine → Prüfen, ob Spieler noch ziehen kann
- utility = 1, der Gegner hat weniger als 3 Steine → Ergebnis zurückgeben

Als nächstes wird überprüft, ob der aktuelle Spieler noch ziehen kann, also ein möglicher Folgezustand existiert. Wenn nicht, wird -1 zurückgegeben, da der Spieler verloren hat.

- utility = -1, der Spieler kann nicht mehr ziehen → Spieler hat verloren

Ansonsten wird theoretisch 0 zurückgegeben, was bedeutet, dass das Spiel unentschieden ist. Allerdings wird dies praktisch nicht passieren, da `utility()` nur mit einem Zustand aufgerufen wird, der ein sicherer Endzustand ist, also entweder der Spieler oder der Gegner gewonnen hat. Ein Unentschieden existiert zwar, tritt aber auf, wenn beispielsweise die maximale Anzahl an Zügen ohne eine neue Mühle erreicht wurde. Dies wird jedoch nicht durch `utility()` überprüft.

```
[ ]: def utility(terminal_state, player):
         if count_player_pieces(terminal_state[1], player)           < 3: return␣
     ↪-1
```

```
    if count_player_pieces(terminal_state[1], opponent(player)) < 3: return 1

    if len(next_states(terminal_state, player)) == 0: return -1
    return 0
```

[ ]:
```
%run ./Muehle_Utilities.ipynb
```

5 Heuristik

Die Funktion `heuristic()` dient dazu die Siegwahrscheinlichkeit für einen Spieler bei einem gegebenen Zustand zu berechnen. Je höher das Ergebnis, desto wahrscheinlicher gewinnt der Spieler.

Der Wert setzt sich aus verschiedenen Einzelwerten zusammen:

- Steine auf dem Spielfeld und noch setzbare Steine zählen je **1 Punkt**
- Jede eigene Mühle gibt **3 Punkte**

Die Punkte werden für jeden Spieler aufsummiert, der Wert des Gegners vom eigenen Wert abgezogen und schließlich normiert (in den Wertebereich [-1, +1] verschoben).

[ ]:
```
def heuristic(state, player):
    remaining, board = state
    opponent_player  = opponent(player)
    maximum_score    = 0

    # Steine auf Spielfeld und in Remaining
    maximum_score   += 9
    score_player     = count_player_pieces(board, player)          +␣
↪remaining[player-1]
    score_opponent   = count_player_pieces(board, opponent_player) +␣
↪remaining[opponent_player-1]

    # Mühlen
    maximum_score   += 4 * 3 # Ein Spieler kann maximal 4 Mühlen␣
↪gleichzeitig haben
    score_player    += len(find_mills(board, player)) * 3
    score_opponent  += len(find_mills(board, opponent_player)) * 3

    return (score_player - score_opponent) / maximum_score
```

6 Algorithmen

[ ]:
```
%run ./Muehle_Logic.ipynb
%run ./Muehle_Heuristic.ipynb
```

[ ]:
```
Cache_Memoize = {}
```

6.1 Memoisierung

Die nachfolgende Memoisierungsfunktion `memoize(f)` wird im Rahmen des Minimax-Algorithmus verwendet, um zu `value_minimax(state, player, depth)` einen Cache hinzuzufügen. Als Schlüssel zur Zwischenspeicherung der berechneten Werte wird ein Tupel aus Status, Spieler und Suchtiefe verwendet (Argumente von `value_minimax()`).

[ ]:
```
def memoize(f):
    global Cache_Memoize

    def f_memoized(*args):
        key = (args[0], args[1], args[2])
        if key in Cache_Memoize:
            return Cache_Memoize[key]
        result = f(*args)
        Cache_Memoize[key] = result
        return result

    return f_memoized
```

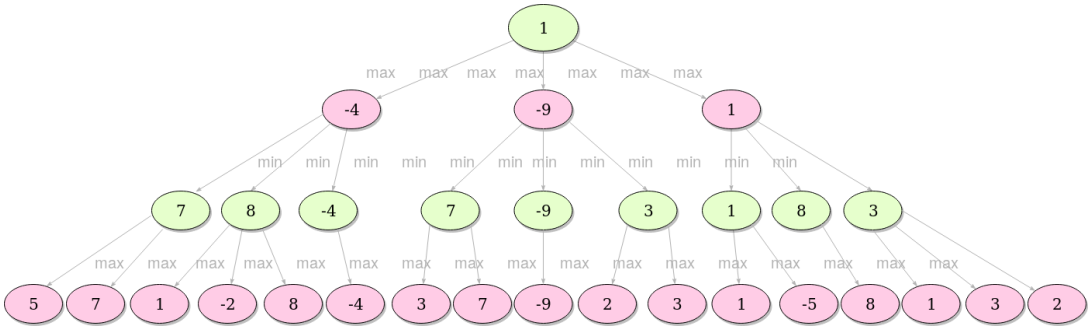
6.2 Minimax-Algorithmus

Der Minimax-Algorithmus wird im Rahmen dieser Studienarbeit für die Ermittlung der optimalen Strategie für das Nullsummenspiel Mühle verwendet. Bei einem Nullsummenspiel wie Mühle ist der Gewinn eines Spielers gleichzeitig der Verlust des anderen Spielers. Die Summe ist somit null. Der Minimax-Algorithmus ist hierbei ein relativ einfacher Algorithmus für Nullsummenspiele, der durch die spätere Verwendung von Alpha-Beta-Pruning deutlich verbessert werden kann.

Der Minimax-Algorithmus beruht auf einer Bewertungsfunktion, der Heuristik und systematischer Suche. Grundlegend werden bei Minimax alle auf den aktuellen Spielstatus folgenden Zustände berechnet und bewertet. Dies ist vergleichbar mit einer Baumstruktur, bei der bis zu den Blättern alle Zustände ausgewertet werden. Da dies aus Gründen der Rechenzeit und

des Speichers nicht möglich ist, werden die Folgezustände nur bis zu einer gewissen Tiefe berechnet und ausgewertet. Um nur bis zu einer geringen Baumtiefe suchen zu müssen wird allerdings eine geeignete Heuristik benötigt, da wir mit der übergebenen Suchtiefe nicht immer einen Endzustand erreichen und den Zustand somit nicht eindeutig bewerten können. Mit der Nutzung dieser Heuristik verlieren wir also die Sicherheit den optimalen Zug zu wählen, erhalten aber immerhin eine gute Einschätzung und eine akzeptable Rechenzeit. Ohne eine Heuristik müssten wir alle Spielverläufe im Vorfeld ausrechnen, was bei nicht trivialen Nullsummenspielen äußerst aufwendig, beziehungsweise nicht möglich ist.

Minimax-Baumstruktur-Beispiel: Die Werte innerhalb der Knoten/Blätter entsprechen der Bewertung des jeweiligen Zustandes



Quelle: https://stackabuse.com/minimax-and-alpha-beta-pruning-in-python

Die Funktion `value_minimax(state, player, depth)` erhält drei Argumente. Einen Spielzustand, einen Spieler und die Suchtiefe. Die Funktion gibt dabei den Wert zurück, den der übergebene Spielzustand für den übergeben Spieler hat. Dieser Wert wird, für den Fall, dass das Spiel mit dem übergebenen Zustand beendet ist, von der Funktion `utility()` berechnet. Wenn die maximale Suchtiefe erreicht wurde wird der Wert allerdings von der Funktion `heuristic()` berechnet. Ist die maximale Suchtiefe noch nicht erreicht wird rekursiv nach dem besten Folgezustand gesucht. Um Werte für Zustände nicht mehrfach berechnen zu müssen werden diese durch Memoisation `@memoize` zwischengespeichert.

[ ]:
```
@memoize
def value_minimax(state, player, depth):
    if finished(to_list(state), player):
        return utility(to_list(state), player)
    if depth == 0:
        return heuristic(state, player)
    o = opponent(player)
    depth -= 1
    return max([ -value_minimax(to_tuple(ns), o, depth) for ns in␣
 ↪next_states(to_list(state), player) ])
```

Die Funktion `best_move_minimax(state, player, depth)` erhält drei Argumente. Einen Spielzustand, einen Spieler und die Suchtiefe. Die Rückgabewerte sind, der von der Funktion

ermittelte, beste Folgezustand und dessen Bewertung. Gibt es mehrere beste Folgezustände wird der Folgezustand zufällig ausgewählt.

Die besten Züge werden von der Funktion durch das wiederholte negierte Aufrufen der Funktion `value_minimax()` bestimmt, da für den Folgezustand mit dem Gegner als Spieler gesucht werden muss. Genauer gesagt müssen wir, da wir uns in einem Nullsummenspiel befinden, die Bewertungen für alle berechneten Züge negieren, um für den, der Funktion `best_move_minimax()`, übergebenen Spieler den besten Zug zu wählen.

[ ]:
```
def best_move_minimax(state, player, depth):
    ns          = next_states(state, player)
    best_value  = value_minimax(to_tuple(state), player, depth)
    best_moves  = [s for s in ns if -value_minimax(to_tuple(s),␣
 ↪opponent(player), depth - 1 ) == best_value]
    best_state  = random.choice(best_moves)
    return best_value, best_state
```

Die Funktion `minimax(state, player, depth)` wurde erstellt, um die Funktion `best_move_minimax()` nach außen hin eindeutiger von `best_move_ab()` abzugrenzen. Die übergebenen Argumente und Rückgabewerte entsprechen somit denen der Funktion `best_move_minimax()`.

[ ]:
```
def minimax(state, player, depth = 5):
    return(best_move_minimax(state, player, depth))
```

6.3 Alpha-Beta-Pruning

Das Alpha-Beta-Pruning ist, wie im Rahmen des Minimax-Algorithmus schon erwähnt eine Verbesserung von Minimax. Beim Alpha-Beta-Pruning wird die Auswertung von Teilbäumen abgebrochen, sobald klar ist, dass keine Verbesserung erwartbar ist. Durch diese Technik wird die Rechenzeit bei steigender Suchtiefe gegenüber Minimax erheblich reduziert. Das Hauptkonzept von Alpha-Beta-Pruning besteht darin, die Werte Alpha und Beta über die gesamte Suche hinweg mitzunehmen. Alpha enthält dabei den bestmöglichen Wert der erkundeten Optionen für den maximierenden Spieler und Beta das gleiche für den minimierenden Spieler, wobei Alpha und Beta initial auf dem für Alpha und Beta schlechtesten Wert starten. (Alpha = -1, Beta = 1)

[ ]:
```
Cache_AB = {}
```

Der Funktion `value_ab(state, player, alpha=-1, beta=1)` werden vier Argumente übergeben. Einen Spielzustand, einen Spieler, Alpha, Beta und die Suchtiefe. Dabei gibt value_ab wie value_minimax, den ermittelten Wert für den übergebenen Spielzustand und den übergebenen Spieler zurück.

[ ]:
```python
def value_ab(state, player, alpha=-1, beta=1, depth=6):
    global Cache_AB
    state = to_tuple(state)
    if (state, player, depth) in Cache_AB:
        value, a, b = Cache_AB[(state, player, depth)]
        if a <= alpha and beta <= b:
            return value
        else:
            alpha = min(alpha, a)
            beta  = max(beta , b)
            value   = alpha_beta(state, player, alpha, beta, depth)
            Cache_AB[(state, player, depth)] = value, alpha, beta
            return value
    else:
        value = alpha_beta(state, player, alpha, beta, depth)
        Cache_AB[(state, player, depth)] = value, alpha, beta
        return value
```

Die Funktion `alpha_beta(state, player, alpha, beta)` erhält die gleichen Werte wie die Funktion `value_ab()` und gibt den Wert für den übergebenen Zustand zurück. Dabei arbeitet `alpha_beta()` nach den im Skript vorgestellten Regeln:

- $\alpha \leq \texttt{value}(state, player) \leq \beta \rightarrow \texttt{alpha\_beta}(state, player, \alpha, \beta) = \texttt{value}(state, player)$
- $\texttt{value}(state, player) < \alpha \rightarrow \texttt{alpha\_beta}(state, player, \alpha, \beta) \leq \alpha$
- $\beta < \texttt{value}(state, player) \rightarrow \beta \leq \texttt{alpha\_beta}(state, player, \alpha, \beta)$

[ ]:
```python
def alpha_beta(state, player, alpha, beta, depth):
    state = to_list(state)
    if finished(state, player):
        return utility(state, player)
    if depth == 0:
        return heuristic(state, player)
    value = alpha
    for ns in next_states(state, player):
        value = max(value, -value_ab(ns, opponent(player), -beta, -alpha,␣
↪depth-1))
        if value >= beta:
            return value
        alpha = max(value, alpha)
    return value
```

Die Funktion `best_move_ab(state, player, depth = 5)` erhält drei Argumente. Einen Spielzustand, einen Spieler und die Suchtiefe. Die Rückgabewerte sind, der von der Funkti-

on ermittelte, beste Folgezustand und dessen Bewertung. Ermittelt wird diese Zustand durch die Bewertung aller Folgezustände durch die Funktion `value_ab()` und die darauf folgende Auswahl des besten Wertes. Verbessert wird die Effizienz von Alpha-Beta-Pruning an dieser Stelle durch den Einsatz von iterative deepening. Dabei wird die Suchtiefe immer weiter, bis zum Erreichen des Werts von `depth`, gesteigert.

[ ]:
```
def best_move_ab(state, player, depth):
    ns     = next_states(state, player)
    moves  = []
    values = []
    ### iterative deepening
    for s in ns:
        distance = 1
        while distance <= (depth - 1):
            value = -value_ab(s, opponent(player), depth = distance)
            distance += 1
        values = values + [value]
        moves = moves + [s]
    ###
    best_value = max(values)
    best_moves = []
    for i in range(0,len(values)):
        if values[i] == best_value:
            best_moves = best_moves + [moves[i]]
    best_state = random.choice(best_moves)
    return best_value, best_state
```

Die Funktion `alpha_beta_pruning(state, player, depth)` wurde erstellt, um die Funktion `best_move_ab(state, player, depth)` nach außen hin eindeutiger von `best_move_minimax (state, player, depth)` abzugrenzen.

[ ]:
```
def alpha_beta_pruning(state, player, depth = 6):
    return(best_move_ab(state, player, depth))
```

7 Fazit

Abschließend soll ein Fazit zum entwickelten Mühle-Spiel gezogen werden. Dazu wird die Performance der implementierten Algorithmen Alpha-Beta und Minimax verglichen und beurteilt. Des Weiteren soll analysiert werden, mit welchen Maßnahmen die Künstliche Intelligenz weiter verbessert werden könnte.

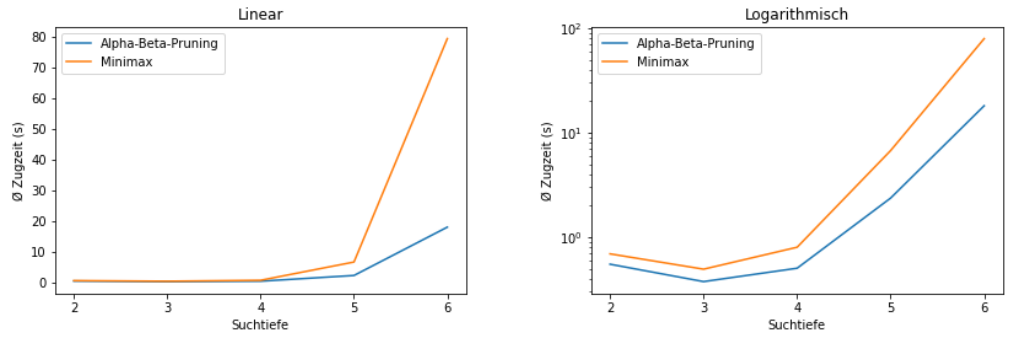
7.1 Alpha-Beta vs. Minimax

Für die folgende Auswertung ist zu beachten, dass die Spiele auf einem Rechner mit folgender Spezifikation durchgeführt wurden:

- Prozessor: AMD Ryzen 5 3600x
- RAM: 16Gb DDR4

Zur Bewertung der zeitlichen Performance wurden Spiele durchgeführt, bei denen die beiden Algorithmen Alpha-Beta-Pruning und Minimax mit der gleichen Suchtiefe gegeneinander gespielt haben. Die Ergebnisse sind der unten aufgeführten Tabelle sowie den Grafiken zu entnehmen.

Suchtiefe	Ø Zugzeit Alpha-Beta (s)	Ø Zugzeit Minimax (s)	MM/AB
2	0,551338714	0,692698593	126%
3	0,37587081	0,494166458	131%
4	0,504812208	0,801580364	159%
5	2,36854893	6,745960603	285%
6	18,09719203	79,44205967	439%



Zu sehen ist, dass sowohl der absolute als auch der relative Abstand mit zunehmender Suchtiefe exponentiell steigen. Das heißt, dass gerade bei einer großen Suchtiefe Alpha-Beta-Pruning wesentlich schneller ist. Doch eine höhere Suchtiefe führt zu einem besseren Spiel der künstlichen Intelligenz und somit zu einer höheren Gewinnwahrscheinlichkeit. Deshalb ist für diesen

Anwendungsfall als Such-Algorithmus Alpha-Beta-Pruning die bessere Wahl.

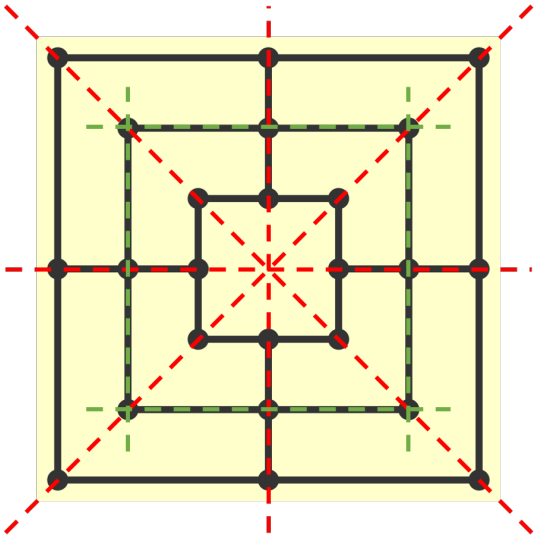
7.2 Ausblick

Das entwickelte Mühle-Spiel ist in der Lage, verschiedene Spielmodi anzubieten. Das Spielen gegen eine KI, sowie das Spiel KI gegen KI ist möglich, sodass das grundlegende Ziel dieser Arbeit erfüllt wurde. Auch die Verwendung von Transpositionstabellen als Speicher für die bereits bewerteten Zustände wurde umgesetzt. Eine Bewertung der Ergebnisse wurde im vorangehenden Kapitel vorgenommen. Nun soll auf Möglichkeiten eingegangen werden, die die Performance optimieren oder die Stärke der KI steigern.

Die erste Möglichkeit hier ist die Verbesserung der Heuristik. Da die Heuristik die Bewertung der Zustände vornimmt und somit über den nächsten Zustand entscheidet, ist sie der Ansatzpunkt für Leistungssteigerungen. Aktuell umfasst die Heuristik die Aspekte Anzahl der Spielsteine sowie die Anzahl der geschlossenen Mühlen. Überlegungen für weitere Aspekte sind die Betrachtung der Anzahl der Zugmöglichkeiten und Anzahl der offenen Mühlen sowie die Bewertung von bestimmten Stellungen. Kriterien für die Entscheidung sind die Veränderung der Rechenzeit und des Gewinnanteils. Ist Erhöhung der Rechenzeit möglichst niedrig, während der Anteil der Gewinne gesteigert werden kann, sollte der Aspekt in die Heuristik mit aufgenommen werden. Auch die Gewichtung der unterschiedlichen Aspekte ist ein Punkt, an dem noch eingegriffen und optimiert werden kann.

Eine weitere Verbesserungsmöglichkeit ist die Einführung einer Spieldatenbank zur Ermittlung besserer Züge. Prinzipiell ist so eine Datenbank für das gesamte Spiel nutzbar, aufgrund des hohen Speicherbedarfs wird häufig jedoch nur eine Endspieldatenbank umgesetzt. Gerade in der dritten Spielphase, wenn die Spieler viele Zugmöglichkeiten haben und der Spielbaum so stark verzweigt ist, kann dies hilfreich sein. Ein Grundstein für die Erstellung und Verwendung einer solchen Spieldatenbank wurde mit dem Abspeichern der Spiele schon gelegt. Eine richtige Verwendung der Datenbank lag jedoch nicht im Anforderungsrahmen dieser Studienarbeit.

Eine weitere Optimierungsmöglichkeit, um schneller mehr Zustände abzudecken und den Speicherbedarf zu senken ist die Beachtung von Symmetrien und Drehungen. Die Grafik unten veranschaulicht die Symmetrien eines Spielfeldes. Die roten Achsen sind einfache Spiegelungen. Aber auch das Tauschen des inneren und äußeren Rings ist möglich. Dies wird durch die grünen Linien veranschaulicht.



Auch eine Drehung des Spielfelds um 90, 180 oder 270 Grad verändert den eigentlichen Zustand nicht, weshalb auch hier eine Vereinfachung stattfinden kann.

Bei der Überlegung, ob Symmetrien und Drehungen betrachtet werden sollen, ist zu beachten, dass zwar der Speicheraufwand und die Anzahl der möglichen Zustände reduziert werden, die Rechenzeit jedoch stark zunimmt.

Abschließend ist zu sagen, dass noch weitere Verbesserungen möglich sind, dies aber immer unter Abwägung der Vor- und Nachteile und vor allem hinsichtlich der benötigten Rechenzeit geschehen sollte.