# Studienarbeit zur Erstellung einer künstlichen Intelligenz zum Spielen des Brettspiels Mühle

5. Mai 2021

## 1 Einleitung

Im Rahmen dieser Studienarbeit wird eine künstliche Intelligenz entwickelt, die das Brettspiel Mühle spielen kann. Dafür werden die beiden Algorithmen *Minimax* und *α-β-Pruning* implementiert.

Die Arbeit wurde von Joost Ole Seddig, Benedikt Funke und Niclas Kaufmann im dritten Studienjahr des Bachelorstudiengangs *Angewandte Informatik* im Kurs TINF18AI2 angefertigt. Der Betreuer der Studienarbeit ist Prof. Dr. Karl Stroetmann.

### 1.1 Mühle

Mühle ist ein Brettspiel für zwei Personen. Das Spielbrett besteht aus drei Quadraten, die in den Seitenmitten verbunden sind. Jeder Spieler (meistens mit *schwarz* und *weiß* bezeichnet) startet mit 9 Steinen, die nacheinander auf das Spielbrett gelegt werden. Gewonnen ist das Spiel, wenn der Gegenspieler keinen Zug mehr spielen kann oder er weniger als drei Steine auf dem Spielbrett hat. Ein Spieler kann eine sogenannte Mühle bilden, wenn er drei seiner Spielsteine in eine Reihe ziehen kann. Daraufhin darf er dem Gegenspieler einen Spielstein vom Brett entfernen. Die exakten Spielregeln können in den offiziellen Turnierregeln des *Weltmühlespiel Dachverband* nachgelesen werden.

Die beiden Algorithmen *Minimax* und *α-β-Pruning* sind Suchalgorithmen zur Bestimmung eines optimalen Zuges bei einem Zwei-Personenspiel. Im weiteren Verlauf der Studienarbeit werden die Suchalgorithmus durch die Verwendung von Symmetrie und iterativer Tiefensuche weiter verbessert. Symmetrie bedeutet in diesem Fall, dass man es sich zu Nutze macht, dass Mühle ein quadratische Spielfeld nutzt und man durch Drehungen und Spiegelungen gleiche Zustände erhält. Iterative Tiefensuche ist ein Verfahren, mit dem man kontinuierlich die Suchtiefe des Algorithmus erhöht.

Zusätzlich zu dem Spiel soll eine graphische Oberfläche entwickelt werden, damit das Spiel auch vom Menschen gespielt werden kann.

Zum Abschluss der Arbeit wird eine Art Turnier gespielt, die die beiden Algorithmen sowie verschiedene Heuristiken gegeneinander spielen lässt. Somit soll eine quantitative Bewertung der Implementierung ermöglicht werden.

## 1.2 Inhaltsverzeichnis

Die gesamte Studienarbeit wird in Jupyter Notebooks mit der Progammiersprache Python geschrieben. So können die Implementierungen direkt erklärt werden. Zum Lesen der Studienarbeit wird folgende Lesereihenfolge empfohlen:

### 1.2.1 nmm-game-utils.ipynb

Dieses Notebook beschreibt Hilfsfunktionen, die für die allgemeine Mühle-Implementierung benötigt werden.

### 1.2.2 nmm-game.ipynb

In diesem Notebook wird das Spiel Mühle implementiert.

### 1.2.3 nmm-artificial-intelligence.ipynb

Dieses Notebook implementiert eine abstrake Superklasse `ArtificialIntelligence`, um die Implementierung von *Minimax* und *α-β-Pruning* zu vereinheitlichen.

### 1.2.4 nmm-heuristic.ipynb

Falls die maximale Suchtiefe erreicht wird und der betrachtete Zustand das Spiel nicht beendet, muss der Wert des Zustandes geschätzt werden. Diesen Wert schätzt eine Heuristik, die in diesem Notebook implementiert wird.

### 1.2.5 nmm-symmetry.ipynb

Dieses Notebook definiert die verschiedenen Symmetrien, die für ein Spielbrett erkannt werden sollen.

### 1.2.6 nmm-minimax.ipynb

In diesem Notebook wird der erste Algorithmus *Minimax* implementiert.

### 1.2.7 nmm-alpha-beta-pruning.ipynb

Der Algorithmus *α-β-Pruning* wird in diesem Notebook beschrieben und entwickelt.

### 1.2.8 nmm-gui-utils.ipynb

Dieses Notebook beschreibt Hilfsfunktionen, die für das Zeichnen der graphischen Oberfläche und Spielen in dieser benötigt werden.

### 1.2.9 nmm-gui.ipynb

Das Spielen in der graphischen Oberfläche wird in diesem Notebook implementiert.

#### 1.2.10 nmm-tournament.ipynb

Um eine Analyse der beiden Algorithmen *Minimax* und *α-β-Pruning* und verschiedener Heuristiken durchführen zu können, wird in diesem Notebook eine Art Turnier implementiert. Dieses Turnier ermöglicht es, dass verschiedene Ausführungen der Algorithmen automatisch gegeneinander spielen können und die Spielverläufe und Ergebnisse aufgezeichnet werden.

#### 1.2.11 nmm-conclusion.ipynb

Zum Abschluss der Studienarbeit wird das Turnier aus dem vorherigen Notebook gespielt und die Algorithmen und Heuristiken ausgewertet. In dem Fazit werden außerdem Verbesserungen für die Implementierung aufgezeigt.

## 2 Hilfsfunktionen für die Spielimplementierung

In dieser Dateien werden Hilfsfunktionien implementiert, die für die grundlegende Spielimplementierung benötigt werden.

### 2.1 Hilfsfunktionen für Spielsteine

Nachfolgend werden alle Hilfsfunktionen implementiert, die für das Interagieren mit den Spielsteinen benötigt werden.

Die Funktion `hasPlaceableStones` überprüft, ob ein Spieler für einen Zustand noch zusetzende Steine auf dem Stapel (*stash*) besitzt. Die Funktion hat zwei Argumente:

- `s` ∈ *States*;
- `p` ∈ *Player*.

Die Funktion gibt ein booleschen Wert zurück.

[ ]:
```
def hasPlaceableStones(s, p):
    ((cw, cb), _) = s
    return cw >= 1 if p == 'w' else cb >= 1
```

Die Funktion `countStones` zählt die Steine eines Spieler auf einem Spielbrett. Die Funktion hat zwei Argumente:

- `s` ∈ *States*;
- `p` ∈ *Player*.

Die Funktion zählt nur die Steine auf dem Brett, nicht die Steine auf dem Stapel und gibt diese als Ganzzahl zurück.

[ ]:
```
def countStones(s, p):
    (_, board) = s
    return [cell for ring in board for cell in ring].count(p)
```

Die Funktion `isAllowedToJump` überprüft, ob ein Spieler bei einem gegebenen Zustand seine Steine beliebig bewegen darf. Die Funktion hat zwei Argumente:

- `s` ∈ *States*;

- `p` $\in$ *Player*.

Ein Spieler darf mit seinen Steinen springen, g.d.w. er weniger als drei Steine auf dem Spielbrett hat und sich keine Steine mehr von dem Spieler auf dem Stapel befinden. Die zweite Bedingung wird nicht überprüft, weil davon ausgegangen wird, dass die Funktion `hasPlaceableStones` zuvor aufgerufen wird.

Die Funktion gibt einen booleschen Wert zurück.

[ ]:
```
def isAllowedToJump(s, p):
    return countStones(s, p) <= 3
```

Die Funktion `hasEnoughStones` überprüft, ob ein Spieler noch genügend Steine zum Weiterspielen übrig hat. Die Funktion hat zwei Argumente:

- `s` $\in$ *States*;
- `p` $\in$ *Player*.

Ein Spieler hat genau dann genügend Steine, wenn er noch Steine zum Setzen auf dem Stapel hat oder er mindestes drei Steine auf dem Spielbrett besitzt.

Die Funktion gibt einen booleschen Wert zurück.

[ ]:
```
def hasEnoughStones(s, p):
    return hasPlaceableStones(s, p) or countStones(s, p) >= 3
```

Die Funktion `removeFromStash` entfernt einen Stein von dem Stapel des gegebenen Spielers. Die Funktion hat zwei Argumente:

- `stash` ist ein Stapel;
- `p` $\in$ *Player*.

Die Funktion gibt den neuen Stapel zurück.

[ ]:
```
def removeFromStash(stash, p):
    return (
        stash[0] - (1 if p == 'w' else 0),
        stash[1] - (1 if p == 'b' else 0)
    )
```

## 2.2 Hilfsfunktionen für Spieler

In diesem Kapitel werden Hilfsfunktionen für die Spieler implementiert.

Die Funktion `opponent` nimmt einen Spieler und gibt den Gegenspieler zurück. Die Funktion hat ein Argument:

- `p` $\in$ *Player*.

Da Mühle ein Zwei-Personen-Spiel ist, gibt es für die Funktion nur zwei Fälle:

- bei dem weißen Spieler `'w'` wird der Gegenspieler schwarz `'b'` zurückgegeben,
- ansonsten wird standardmäßig weiß `'w'` als Gegenspieler zurückgegeben.

```
[ ]: def opponent(p):
         return 'b' if p == 'w' else 'w'
```

Die Funktion `getPlayerAt` gibt den Spieler an der gegeben Koordinate des Spielbrettes zurück. Die Funktion hat zwei Argumente:

- `board` ist ein Spielbrett;
- `coord` ist eine Koordinate, die überprüft werden soll.

Die Funktion gibt einen Spieler zurück. Falls an dieser Koordinate sich kein Spielerstein befinden sollte, wird entsprechend `' '` zurückgegeben.

```
[ ]: def getPlayerAt(board, coord):
         (r, c) = coord
         return board[r][c]
```

Die Funktion `playerPhase` berechnet für einen gegebenen Zustand und einen Spieler die aktuelle Phase des Spielers. Die Funktion hat zwei Argumente:

- `s` $\in$ *States*;
- `p` $\in$ *Player*.

Die Funktion überprüft mit den beiden Hilfsfunktionen `hasPlaceableStones` und `isAllowedToJump` die Spielerphase und gibt diese als Ganzzahl zurück:

- `1` für die *placing* Phase, g.d.w. der Spieler noch Steine auf dem Stapel hat (`hasPlaceableStones`);
- `2` für die *moving* Phase, g.d.w. der Spieler nicht in Phase 1 oder 3 ist;
- `3` für die *flying* Phase, g.d.w. der Spieler springen darf (`isAllowedToJump`).

```
[ ]: def playerPhase(s, p):
         if hasPlaceableStones(s, p):
             return 1
         elif isAllowedToJump(s, p):
             return 3
         else:
             return 2
```

### 2.3 Hilfsfunktionen für Zellen

In diesem Kapitel werden Hilfsfunktionen für die Veränderung oder Untersuchung von Zelleneigenschaften implementiert.

Die Funktion `findCellsOf` sucht auf einem Spielbrett `board` nach allen Zellen, welche durch Steine des Spielers `p` belegt sind. Die Funktion hat folgende Argumente:

- `board` ist ein Spielbrett;
- `p` $\in$ *Player*.

Zurückgegeben wird die Menge aller Zellen wo gilt `board[r][c] = p`.

[ ]:
```
def findCellsOf(board, p):
    return {(r, c) for r in range(0, 3) for c in range(0, 8) if board[r][c] == p}
```

Die Funktion `findEmptyCells` sucht auf dem Spielfeld `board` nach allen leeren Zellen. Die Funkion nimmt lediglich ein Argument:

- `board` ist ein Spielbrett.

Die Funktion ruft die Funktion `findCellsOf` mit `' '` als Spieler auf und gibt das erhaltene Ergebnis zurück. Somit ist jeder erhaltenen Zelle kein Spieler `'b'` oder `'w'` zugeordnet.

[ ]:
```
def findEmptyCells(board):
    return findCellsOf(board, ' ')
```

Die Funktion `findNeighboringEmptyCells` sucht für eine gegebene Zelle `coordinates` die freien Nachbarzellen. Die Funktion hat zwei Argumente:

- `board` ist ein Spielbrett;
- `coordinates` ist ein Tupel mit den Koordinaten der Ausgangszelle.

Die Funktion überprüft, welche der angrenzenden Zellen in der Menge von `findEmptyCells` enthalten ist und gibt diese zurück.

[ ]:
```
def findNeighboringEmptyCells(board, coordinates):
    (rootR, rootC) = coordinates
    return {
        (r, c)
        for (r, c) in findEmptyCells(board)
        if (r == rootR and (c == (rootC + 7) % 8 or c == (rootC + 1) % 8))
        or (c % 2 == 1 and c == rootC and (r == rootR - 1 or r == rootR + 1))
    }
```

Die Funktion `place` platziert den Stein eines Spielers `player` auf dem Spielbrett `board`. Folgende Argumente benötigt die Funktion:

- `board` ist ein Spielbrett;
- `coordinates` ist ein Tupel mit den Koordinaten des zu platzierenden Steins;
- `player` $\in$ *Player*.

Die Funktion platziert den Stein an den angegebenen Koordinaten und gibt das neue Spielbrett zurück.

[ ]:
```
def place(board, coordinates, player):
    (r, c) = coordinates
    return tuple(
        tuple(
            player if (c == ic) and (r == ir) else board[ir][ic]
            for ic in range(0, 8)
        ) for ir in range(0, 3)
    )
```

Die Funktion `move` bewegt einen bereits plazierten Stein auf dem Spielbrett `board`. Die Funktion erhält drei Argumente:

- `board` ist ein Spielbrett;
- `src` ist ein Tupel mit den Koordinaten der Ausgangszelle;
- `des` ist ein Tupel mit den Koordinaten der Zielzelle.

Die Funktion platziert den Wert der Ausgangszelle in die Zielzelle, entfernt den Stein aus der Ausgangszelle und gibt das neue Spielbrett zurück.

[ ]:
```
def move(board, src, des):
    src_r, src_c = src
    des_r, des_c = des
    content_src = board[src_r][src_c]
    return tuple(
        tuple(
            ' ' if (r,c) == src else (content_src if (r,c) == des else␣
↪board[r][c])
            for c in range(0, 8)
        ) for r in range(0, 3)
    )
```

### 2.4 Hilfsfunktionen für Mühlen

In diesem Kapitel werden Hilfsfunktionen implementiert, die für Mühlen nützlich sind.

Die Konstante `MILL_COORDINATES` beinhaltet alle Mühlen und deren Koordinaten, die auf dem Spielfeld möglich sind.

- Zunächst werden alle Mühlen entlang der Ringe ermittelt. Dazu werden, beginnend an jeder Ecke, die nächsten drei Koordinaten in einer Menge (bzw. `frozenset`) gespeichert.
- Für die Mühlen entlang der Verbindungslinien zwischen den Ringen, wird das gleiche Prinzip verwendet. Hier werden alle Koordinaten in der Mitte einer Seite gespeichert.

[ ]:
```
MILL_COORDINATES = {
    # Calculate all mills on the rings
    frozenset(
        (r, (c+o)%8)
        for o in range(0, 3)
    )
    for r in range(0, 3)
    for c in range(0, 8, 2)
} | {
    # Calculate all mills crossing the rings
    frozenset(
        (r, c)
        for r in range(0, 3)
    )
    for c in range(1, 8, 2)
```

```
}
```

Die Funktion `findMills` berechnet für ein Spielbrett und einen Spieler alle Mühlen, die dieser Spieler aktuell hat. Die Funktion hat zwei Argumente:

- `board` ist ein Spielbrett;
- `p` $\in$ *Players*.

Der Rückgabewert der Funktion ist eine Menge von Mühlen. Eine Mühle wird dabei als Menge (bzw. `frozenset`) dargestellt, die die Koordinaten der Steine als Tupel beinhaltet.

Zum Berechnen aller Mühlen, die ein Spieler aktuell hat, wird über die zuvor berechneten konstanten Menge `MILL_COORDINATES` iteriert und für jede Mühlenposition überprüft, ob dort der Spieler einen Stein hat.

[ ]:
```
def findMills(board, p):
    return {
        mill
        for mill in MILL_COORDINATES
        if all(
            board[r][c] == p
            for (r, c) in mill
        )
    }
```

Die Funktion `findPossibleMills` berechnet für ein Spielbrett und einen Spieler alle möglichen Mühlen. Eine mögliche Mühle ist eine Mühle, bei der eine Zelle noch leer ist, hierbei wird nicht betrachtet, ob es im nächsten Zug möglich ist, diese Mühle zu vervollständigen. Dabei wird Die Funktion hat zwei Argumente:

- `board` ist ein Spielbrett;
- `p` $\in$ *Players*.

Der Rückgabewert der Funktion ist eine Menge von möglichen Mühlen. Eine mögliche Mühle wird dabei als Menge (bzw. `frozenset`) dargestellt, die alle Koordinaten der Mühle als Tupel beinhaltet.

Zum Berechnen aller möglichen Mühlen, die ein Spieler aktuell hat, wird über die zuvor berechnete Menge `MILL_COORDINATES` iteriert und für jede Mühle überprüft, ob der Spieler dort zwei Steine hat und eine Zelle leer ist.

[ ]:
```
def findPossibleMills(board, p):
    return {
        mill
        for mill in MILL_COORDINATES
        if sorted(board[r][c] for (r, c) in mill) == [' ', p, p]
    }
```

Die Funktion `countNewMills` berechnet für ein Spielbrett, eine Menge bereits existierender Mühlen und einen Spieler die Anzahl der Mühlen, die neu entstanden sind. Die Funktion hat 3 Parameter:

- `board` ist ein Spielbrett;
- `oldMills` ist eine Menge von Mühlen (Menge von Koordinaten);
- `p` $\in$ *Players*.

Der Rückgabewert dieser Funktion ist eine Ganzzahl für die gilt *return* $\in \{0, 1, 2\}$. Indem die Menge der bereits existierenden Mühlen von der Menge der aktuellen Mühlen abgezogen wird, erhält man die Menge der aktuellen Mühlen. Von diesen kann die Anzahl berechnet werden.

[ ]:
```
def countNewMills(board, oldMills, p):
    return len(findMills(board, p) - oldMills)
```

Die Funktion `getCellsPoundable` berechnet für ein Spielbrett und einen Spieler, welche Steine des Gegeners dieser schlagen darf. Ein Stein darf geschlagen werden, wenn er sich nicht in einer Mühle befindet. Wenn sich alle Steine in einer Mühle befinden, dürfen alle Steine geschlagen werden. Die Funktion hat 2 Parameter:

- `board` ist ein Spielbrett;
- `p` $\in$ *Players*.

Der Rückgabewert ist eine Menge von Koordinaten. Die Menge aller Koordinaten der Steine des Gegeners, sowie die Menge aller Koordinaten aller Steine aus den Mühlen des Gegeners werden errechnet und voneinander abgezogen. Falls diese Menge nicht leer ist, dürfen nur an diesen Positionen Steine geschlagen werden, ansonsten dürfen alle Steine des Gegners geschlagen werden.

[ ]:
```
def getCellsPoundable(board, p):
    opponentCells = findCellsOf(board, opponent(p))

    opponentCellsInMills = {
        cell
        for mill in findMills(board, opponent(p))
        for cell in mill
    }

    if len(opponentCells - opponentCellsInMills) > 0:
        return opponentCells - opponentCellsInMills
    else:
        return opponentCells
```

Die Funktion `poundMills` errechnet aus einem Spielbrett, einer Anzahl an zu schlagenden Steinen und einem Spieler alle möglichen Spielbrettkonfigurationen, bei denen der Spieler die gegebene Anzahl an Steinen geschlagen hat. Die Funktion hat 3 Parameter:

- `board` ist ein Spielbrett;
- `count` $\in \{0, 1, 2\}$;
- `p` $\in$ *Players*.

Der Rückgabewert ist eine Menge an Spielbrettern. Die Funktion ist rekursiv implementiert. Wenn kein Stein mehr geschlagen werden muss, wird eine Menge mit dem aktuellen Spielbrett zurück gegeben. Ansonsten wird rekursiv die Menge aller möglichen Spielbrettkonfigurationen mit einem zu schlagenden Stein weniger errechnet. Aus allen diesen Spielbrettern wird ein weiterer Stein entfernt, der geschlagen werden darf.

[ ]:
```
def poundMills(board, count, p):
    if count <= 0:
        return { board }

    return {
        place(b, cell, ' ')
        for b in poundMills(board, count-1, p)
        for cell in getCellsPoundable(b, p)
    }
```

# 3 Spiel Definition

In diesem Kapitel werden Funktionen diskutiert, die nötig sind, um das Spiel Mühle

$$G_{NineMen'sMorris} = States, s_0, Players, nextStates, finished, utility$$

zu definieren. Es werden die offizellen Tunierregeln in der Version `3.0` vom 11. Mai 2019 des Weltmühlespiel Dachverbandes verwendet. Jedoch werden die Regeln für Unentschieden ausgelassen, da diese die Zustände zu groß machen würden. Eine Implementierung dieser Regeln ist im Kapitel über die GUI zu finden.

Mit Hilfe des Magic Command `%run` werden Hilfsmethoden geladen, die in einem eigenen Kapitel beschrieben werden.

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

## 3.1 Spieler

Zunächst soll die Menge der Spieler definiert werden. Da Mühle mit zwei Spielern gespielt wird, die jeweils weiße oder schwarze Steine legen, werden die Spieler äquivalent benannt.

- Der beginnende Spieler weiß wird als `w` dargestellt,
- der gegnerische Spieler als `b`.

Für die Implementierung der Funktionen ist diese Menge nicht nötig. Aus Gründen der Vollständigkeit soll diese hier dennoch definiert werden.

[ ]:
```
Player = {'w', 'b'}
```

## 3.2 Zustände

Die vollständige Menge aller Zustände ist aufwendig zu berechnen und wird ebenfalls nicht für die Implementierung benötigt. Da die Zustände berechnet werden, sobald diese benötigt werden. Deswegen wird hier auf eine Definition von *States* verzichtet.

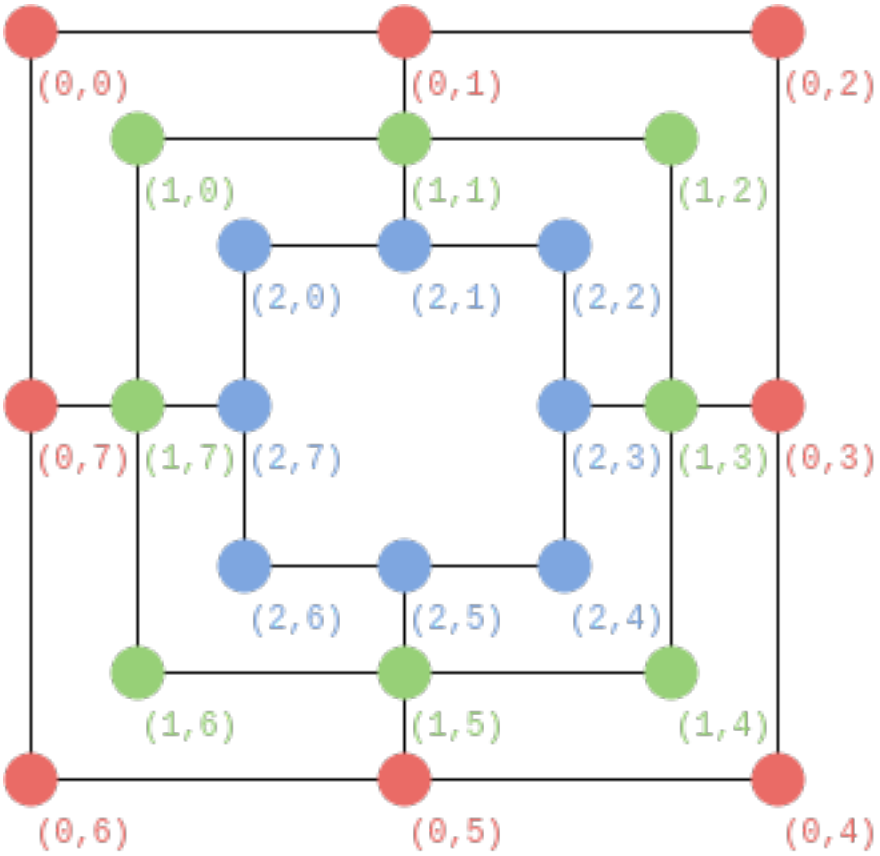
Der Startzustand $s_0$ hingegen wird zu Beginn des Spiel benötigt und ist im Folgenden definiert. Er bezeichnet ein leeres Spielbrett, bei dem beide Spieler noch alle ihre Steine legen müssen. Ein

Zustand (`state`) ist eine Tupel die 1. den Stapel (`stash`) von zu legenden Steinen und 2. das Spielbrett (`board`) selbst beinhaltet.

Der Stapel ist eine Tupel $\langle w, b \rangle$, die die Anzahl der zu legenden Steine von Weiß $w$ und Schwarz $b$ beinhaltet. Diese werden in der genannten Reihenfolge als Zahlen dargestellt.

$$w, b \in \{0...9\}$$

Das Spielbrett wird durch eine Tripel beschrieben, die die Ringe des Spielbretts beinhaltet. Alle zusammengehörigen Spielbrettpositionen, die sich auf dem gleichen Ring befinden, sind in der folgenden Abbildung durch eine Farbe markiert. Der äußere Ring ist *rot*, der mittlere *grn* und der innere *blau* dargestellt, damit gilt für einen Index $r$ der einen der Ringe bezeichnet $r \in \{0...2\}$.




Die Spielpositionen (`cells`) $c$ auf den Ringen sind beginnend mit $c = 0$ ab der oberen linken Ecke im Uhrzeigersinn durch nummeriert. Dadurch ergibt sich, dass $c \in \{0...7\}$. In der Abbilung sind die Koordinaten der Spielpositionen eingezeichnet, so liegt beispielsweise $\langle r, c \rangle = \langle 2, 4 \rangle$ auf dem inneren Ring in der unteren rechten Ecke. An diesen Spielpositionen wird der Spieler $p$ oder eine leere Spielposition gespeichert, für diese gilt $p \in \mathit{Player} \cup \{' '\}$.

So ist ein State eine Tupel $\langle \mathit{stash}, \mathit{board} \rangle$ dessen Parameter

- `stash` eine weitere Tupel, bestehend aus $\langle w, b \rangle$ und
- `board` eine Tripel, welche die Ringe $r_0$, $r_1$ und $r_2$ beinhaltet. Die Ringe selbst sind Neun-Tupel, dessen Elemente die einzelnen Spielpositionen $c \in \mathit{Player} \cup \{' '\}$ darstellen.

Der Startzustand $s_0$ wird in Python wie folgt geschrieben:

```
[ ]: s0 = ((9, 9), (
         (' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '),
         (' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '),
         (' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ')
     ))
```

### 3.3 Folgezustände

Für die Definition des Spieles Mühle $G_{NineMen'sMorris}$ wird die Funktion `nextStates` benötigt. Diese nimmt einen Zustand `s` und einen Spieler `p` entgegen und errechnet mit diesen alle Folgezustände, die entstehen, wenn der gegebene Spieler einen legalen Zug spielt.

Da das Spiel in drei Phasen aufgeteilt ist, ist auch die Implementierung aus Gründen der Übersicht in die folgenden Phasen unterteilt:

1. Die erste Phase des Spiels heißt *placing* Phase, in der die Spieler alle ihre Steine aus dem Stapel auf das Spielbrett legen müssen. Hierbei dürfen bereits Mühlen gelegt und geschlagen werden. Es muss ein Stein platziert werden.
2. Die zweite Phase heißt *moving* Phase und beginnt sobald der Stapel des Spielers leer ist. Beide Spieler kommen gleichzeitig in die zweite Phase, in der die eigenen Steine nur noch entlang der Linien auf die nächste Spielposition geschoben werden dürfen. Es muss ein Stein bewegt werden.
3. Die letzte Phase heißt *flying* Phase. Diese Phase beginnt für einen Spieler, sobald dieser nur noch 3 Steine auf dem Spielbrett liegen hat und sein Stapel leer ist. Der Eintritt in die dritte Phase ist nicht vom gegnerischen Spieler abhängig, somit können Spieler für mehrere Züge alleine in der dritten Phase sein. Hierbei können die verbleibenden Steine beliebig bewegt werden, ohne dass die Positionen direkt nebeneinander liegen müssen. Es muss ein Stein bewegt werden.

In der ersten Phase, der *placing* Phase, wird ein Stein vom Stapel genommen und auf ein freies Feld gelegt. Diese Züge werden in der Funktion `nextStatesPlace` implementiert, hierbei wird ein Spieler `p` und ein Zustand `s` erwartet, für den angenommen wird, dass sich der aktuelle Zustand in der ersten Phase befindet. Die Rückgabe dieser Funktion ist die Menge aller Zustände, die erreicht werden können, in denen der Spieler `p` einen legalen Spielzug auf dem Zustand `s` ausführt.

1. Zunächst werden die Mühlen aus dem aktuellen Zustand mit Hilfe der Funktion `findMills` gespeichert.
2. Daraufhin wird auf jede freie Spielposition ein neuer Stein des Spielers `p` gelegt.
3. Nun werden unter Verwendung von dem Ergebnis aus *1.* und der Funktion `countNewMills` alle neu entstandenen Mühlen gezählt. Für diese Anzahl gilt $a \in \{0, 1, 2\}$.
4. Um für die neu entstandenen Mühlen die entsprechende Anzahl an Steinen zu schlagen, wird die Hilfsfunktion `poundMills` verwendet.
5. Zum Schluss wird vom Stapel des Spielers `p` ein Stein weg genommen und der Stapel mit den Spielbrettern wieder zu Zuständen vereint.

```
[ ]: def nextStatesPlace(s, p):
    # Extract the count of the stones and the board
    ((cw, cb), board) = s
    # Calculate all current mills the player has
    mills = findMills(board, p)

    # Place a stone in any empty cell
    placeBoards = {
        place(board, (r, c), p)
        for (r, c) in findEmptyCells(board)
    }
    # Calculate how many new mills were created
    boardMills = {
        board: countNewMills(board, mills, p)
        for board in placeBoards
    }

    # Here all final boards will be collected
    boards = {
        result
        for (b, count) in boardMills.items()
        for result in poundMills(b, count, p)
    }

    # Remove one stone from the players stache
    (cw, cb) = (cw-1, cb) if p == 'w' else (cw, cb-1)

    # Return all possible states
    return { ((cw, cb), board) for board in boards }
```

In der *moving* Phase (2) wird ein Stein des aktuellen Spielers entlang der Linien zu einer benachbarten, leeren Spielposition geschoben. Die Definition der Funktion `nextStatesMove` wird wieder eine Zustand `s` und ein Spieler `p` erwartet, die sich in der zweiten Phase befinden. Die Rückgabe dieser Funktion ist die Menge aller Zustände, die erreicht werden können, in denen der Spieler `p` einen legalen Spielzug auf dem Zustand `s` ausführt.

1. Zunächst werden die Mühlen aus dem aktuellen Zustand mit Hilfe der Funktion `findMills` gespeichert.
2. Danach werden die Positionen aller Steine des Spielers `p` und dessen leere Nachbarpositionen mit Hilfe der Funktionen `findCellsOf` und `findNeighboringEmptyCells` errechnet. Die Steine werden dort hin bewegt.
3. Nun werden unter Verwendung von dem Ergebnis aus *1.* und der Funktion `countNewMills` alle neu entstandenen Mühlen gezählt. Für diese Anzahl gilt $a \in \{0, 1\}$.
4. Um für die neu entstandenen Mühlen die entsprechende Anzahl an Steinen zu schlagen, wird die Hilfsfunktion `poundMills` verwendet.
5. Zum Schluss wird der Stapel mit den Spielbrettern wieder zu Zuständen vereint.

```
[ ]: def nextStatesMove(s, p):
         # Extract the count of the stones and the board
         ((cw, cb), board) = s
         # Calculate all current mills the player has
         mills = findMills(board, p)

         # Choose any stone of the player and move it to an empty neighbor
         moveBoards = {
             move(board, src, des)
             for src in findCellsOf(board, p)
             for des in findNeighboringEmptyCells(board, src)
         }

         # Calculate how many new mills were created
         boardMills = {
             b: countNewMills(b, mills, p)
             for b in moveBoards
         }

         # Here all final boards will be collected
         boards = {
             result
             for (b, count) in boardMills.items()
             for result in poundMills(b, count, p)
         }

         return { ((cw, cb), board) for board in boards }
```

Die letzte Hilfsfunktion `nextStatesFly` wird in der Phase 3 verwendet und erwartet äquivalent zu den anderen Hilfsmethoden einen Zustand `s` und Spieler `p` in der Phase 3. Hier wird ein Stein des Spieler `p` an eine andere Position auf dem Spielbrett bewegt. Diese Position muss keine Nachbarposition sein und Steine können übersprungen werden. Die Rückgabe dieser Funktion ist die Menge aller Zustände, die erreicht werden können, in denen der Spieler `p` einen legalen Spielzug auf dem Zustand `s` ausführt.

1. Zunächst werden die Mühlen aus dem aktuellen Zustand mit Hilfe der Funktion `findMills` gespeichert.
2. Danach werden die Positionen aller Steine des Spielers `p` und dessen leere Nachbarpositionen mit Hilfe der Funktionen `findCellsOf` und `findNeighboringEmptyCells` errechnet. Die Steine werden dort hin bewegt.
3. Nun werden unter Verwendung von dem Ergebnis aus *1.* und der Funktion `countNewMills` alle neu entstandenen Mühlen gezählt. Für diese Anzahl gilt $a \in \{0, 1\}$.
4. Um für die neu entstandenen Mühlen die entsprechende Anzahl an Steinen zu schlagen, wird die Hilfsfunktion `poundMills` verwendet.
5. Zum Schluss wird der Stapel mit den Spielbrettern wieder zu Zuständen vereint.

[ ]:
```python
def nextStatesFly(s, p):
    # Extract the count of the stones and the board
    ((cw, cb), board) = s
    # Calculate all current mills the player has
    mills = findMills(board, p)

    # Choose any stone of the player and move it to an empty neighbor
    moveBoards = {
        move(board, src, des)
        for src in findCellsOf(board, p)
        for des in findEmptyCells(board)
    }

    # Calculate how many new mills were created
    boardMills = {
        b: countNewMills(b, mills, p)
        for b in moveBoards
    }

    # Here all final boards will be collected
    boards = {
        result
        for (b, count) in boardMills.items()
        for result in poundMills(b, count, p)
    }

    return { ((cw, cb), board) for board in boards }
```

Die Implementierung der `nextStates` Funktion führt nun die vorher definierten Funktionen zusammen. Die aktuelle Phase für den gegebenen Zustand `s` und Spieler `p` wird mit der Hilfsfunktion `playerPhase` errechnet und auf Grund dessen eine Fallunterscheidung ausgeführt, sodass gilt

$$nextStates(s,p) = \begin{cases} nextStatesPlace(s,p) & falls \;\; playerPhase(s,p) = 1 \\ nextStatesMove(s,p) & falls \;\; playerPhase(s,p) = 2 \\ nextStatesFly(s,p) & falls \;\; playerPhase(s,p) = 3 \end{cases}$$

[ ]:
```python
def nextStates(s, p):
    phase = playerPhase(s, p)
    if phase == 1:
        return nextStatesPlace(s, p)
    elif phase == 2:
        return nextStatesMove(s, p)
    else:
        return nextStatesFly(s, p)
```

### 3.4 Spielende

Für die Definition des Spieles Mühle $G_{NineMen'sMorris}$ werden zwei weitere Funktionen benötigt, die das Ende des Spiels behandeln: `finished` und `utility`.

Die Funktion `finished` errechnet für einen gegebenen Zustand `s`, ob das Spiel beendet ist, wenn Spieler `p` an der Reihe ist. Dies ist der Fall, g.d.w.

- einer der Spieler $p \in Players$ weniger als 3 Steine hat (`hasEnoughStones`) oder
- der Spieler `p` keinen legalen Zug mehr tätigen kann.

Der optionale Parameter `ns` ist lediglich eine Optimierung, die es ermöglicht, bereits berechnete Folgezustände zu verwenden, anstatt diese noch einmal errechnen lassen zu müssen.

[ ]:
```
def finished(s, p, ns=None):
    if not ns:
        ns = nextStates(s, p);
    return not hasEnoughStones(s, 'w') or \
           not hasEnoughStones(s, 'b') or \
           len(ns) == 0
```

Die Funktion `utility` errechnet für den gegebenen Zustand `s` und Spieler `p`, falls $finished(s, p) = true$ gilt, wer gewonnen hat.

$$utility(s, p) = \begin{cases} -1 & falls\ \ p\ verliert\ in\ s \\ 0 & falls\ \ Unentschieden \\ 1 & falls\ \ p\ gewinnt\ in\ s \end{cases}$$

Sobald ein Spieler zu wenig Steine hat (`hasEnoughStones`) oder keinen legalen Zug mehr tätigen kann, hat dieser verloren. Dadurch gewinnt automatisch der gegnerische Spieler. Zwar gibt es im Spiel Mühle ein Unentschieden, dies kann aber nicht anhand der Zustände erkannt werden, da hierbei die vorher gespielten Spielzüge betrachtet werden müssen.

Äquivalent zu der Funktion `finished` ist Parameter `ns` optional und stellt lediglich eine Optimierung dar, die es ermöglicht, bereits berechnete Folgezustände zu verwenden, anstatt diese noch einmal errechnen lassen zu müssen.

[ ]:
```
def utility(s, p, ns=None):
    if not ns:
        ns = nextStates(s, p);

    if not hasEnoughStones(s, p):
        return -1
    if not hasEnoughStones(s, opponent(p)):
        return 1
    if len(nextStates(s, p)) == 0:
        return -1
```

```
    # Should be impossible, as utility() will only be called if finished()␣
↪returns True
    return 0
```

# 4 Künstliche Intelligenz

Diese Studienarbeit implementiert zwei verschiedene Algorithmen als künstliche Intelligenz: Minimax und $\alpha$-$\beta$-Pruning. Damit diese Algorithmen leichter wiederverwendet und mit verschiedenen Einstellungen ausgeführt werden können, wird eine abstrakte Superklasse angelegt. Diese bescheibt welche Funktionen nötig sind, damit eine Implementierung den Ansprüchen einer künstlichen Intelligenz für Mühle entspricht.

## 4.1 BestMoves

Die Klasse `BestMoves` beschreibt das Ergebnis, welches eine künstliche Intelligenz für Mühle erzeugen soll. Diese Klasse beschreibt den Rückgabewert der später definierten Funktion `bestMoves`. Sie hat drei Attribute:

- `states` $\subset$ *States*;
- `value` $\in [-1.0, 1.0]$;
- `debugInformation` ist ein `dict`, welches weitere Informationen, wie beispielsweise die erreichte Rekursionstiefe oder die besuchten Zustände, beinhalten kann.

[ ]:
```
class BestMoves():
    def __init__(self, states, value, debugInformation):
        self.states = states
        self.value = value
        self.debugInformation = debugInformation
```

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

[ ]:
```
def __repr__(self):
    return f"BestMoves(states={self.states}, value={self.value},␣
↪debugInformation={self.debugInformation})"

BestMoves.__repr__ = __repr__
del __repr__
```

Die Funktion `choice` wählt zufällig einen der möglichen besten Züge aus einer `BestMoves` Instanz aus. Dadurch wird sichergestellt, dass nicht immer der gleiche Zug durch die künstliche Intelligenz gespielt wird.

[ ]:
```
import random
def choice(self):
    return random.choice(self.states)
```

```
BestMoves.choice = choice
del choice
```

## 4.2 ArtificialIntelligence

Für die Definition der abstrakten Superklasse müssen zunächst aus dem Paket `abc` *Abstract Base Classes* Hilfsklassen und -funktionen importiert werden. Diese werden benötigt um eine abstrakte Klassen in Python darstellen zu können.

[ ]:
```
from abc import ABC, abstractmethod
```

Die abstrakte Superklasse `ArtificialIntelligence` ist selbst eine Unterklasse von `ABC`, dadurch wird die Klasse als abstrakt markiert. `ArtificialIntelligence` hat eine abstrakte Funktion `bestMoves`, die für einen Zustand und einen Spieler alle besten Züge errechnen soll und diese in Form einer `BestMoves` Instanz zurückgeben soll. Sie hat zwei Argumente:

- `states` $\in$ *States*;
- `player` $\in$ *Players*.

[ ]:
```
class ArtificialIntelligence(ABC):
    @abstractmethod
    def bestMoves(self, state, player) -> BestMoves:
        pass
```

## 4.3 Heuristik

Da die Rechenleistung nicht ausreicht um vor jedem Zug den gesamten Spielbaum abzusuchen, gibt es eine maximale Rekursionstiefe, bei der die Suche abgebrochen wird. Wenn diese Rekusionstiefe erreicht wird, muss der Wert des aktuellen Zustands geschätzt werden. Hierfür wird die in diesem Kapitel beschriebene Heuristik verwendet.

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

Die Klasse `HeuristicWeights` ist eine Hilfsklasse, dessen Instanzen beim Aufruf von der später definierten Funktion `heuristic` übergeben werden können. In dieser Klasse werden alle Gewichtungen für die einzelnen Eigenschaften eines Zustandes gespeichert. Dadurch wird es ermöglicht künstliche Intelligenzen mit verschiedenen Heuristiken gegeneinander antreten zu lassen, um herauszufinden welche Heuristik den Wert eines Zustandes am genausten abbildet.

Eine Instanz der Klasse `HeuristicWeights` besteht aus vier Gewichtungen, eine für jede Eigenschaft:

- `stones`: Diese Eigenschaft zählt, wie viele Steine der Spieler auf dem Spielbrett hat.
- `stash`: Die Steine auf dem Stapel werden ebenfalls gezählt.
- `mills`: Die Anzahl der Mühlen, die ein Spieler auf dem Spielbrett hat, wird durch diese Eigenschaft gezählt.

- `possible_mills`: Die Anzahl der möglichen Mühlen, dh. Mühlen bei denen eine Zelle noch frei ist, werden ebenfalls gezählt. (Siehe `findPossibleMills` im Kapitel *Hilfsfunktionen für die Spielimplementierung* für eine genauere Beschreibung einer möglichen Mühle.)

[ ]:
```
class HeuristicWeights():
    def __init__(self, stones=1, stash=1, mills=4, possible_mills=2):
        self.stones = stones
        self.stash = stash
        self.mills = mills
        self.possible_mills = possible_mills
```

Für Debuggingzwecke wird eine `__repr__` Funktion implementiert, die eine String-Repräsentation aus einer `HeuristicWeights` Instanz mit allen Einstelungen erstellt.

[ ]:
```
def __repr__(self: HeuristicWeights):
    return f"HeuristicWeights(stones={self.stones}, stash={self.stash}, " + \
           f"mills={self.mills}, possible_mills={self.possible_mills})"

HeuristicWeights.__repr__ = __repr__
del __repr__
```

Die Funktion `heuristic` berechnet für einen Spieler den geschätzten Wert eines Zuststandes anhand der oben aufgeführten Eigenschaften. Die Funktion hat drei Argumente:

- `state` $\in$ *States*;
- `player` $\in$ *Players*;
- optional: `weights` ist eine Instanz der Klasse `HeuristicWeights`.

Für die Implementierung werden alle gewichteten Werte der Eigenschaften für die Spieler weiß `w` und schwarz `b`, sowie der maximale Wert für die Eigenschaft errechnet. Damit die Schätzung des Wertes nicht außerhalb des Wertebereiches, gegeben durch die tatsächlichen Werte für gewinnende (`1.0`) und verlierende (`-1.0`) Zustände, liegt, wird das Maximum um eins erhöht und der errechnete Wert durch das Maximum skaliert. Zum Schluss wird das Vorzeichen angepasst, damit der gegebene Spieler berücksichtigt wird.

[ ]:
```
def heuristic(state, player, weights=HeuristicWeights()):
    ((stash_white, stash_black), board) = state

    # Count the stones on the board
    white = weights.stones * countStones(state, 'w')
    black = weights.stones * countStones(state, 'b')
    # Count the stones in the stash
    white += weights.stash * stash_white
    black += weights.stash * stash_black
    # There can be at maximum 9 stones per player, so the maximum is the maximum␣
 ↪of the weights times the stones
    maximum = 9 * max(weights.stones, weights.stash)

    # Count the mills the player currently has
```

```
    white += weights.mills * len(findMills(board, 'w'))
    black += weights.mills * len(findMills(board, 'b'))
    # There can be at maximum 4 mills for each player
    maximum += 4 * weights.mills

    # Count the possible mills the player currently has
    white += weights.possible_mills * len(findPossibleMills(board, 'w'))
    black += weights.possible_mills * len(findPossibleMills(board, 'b'))
    # There can be at maximum 8 possible mills for each player
    maximum += 8 * weights.possible_mills

    # Substract the player scores and clamp them into (-1;+1)
    score = (white - black) / (maximum + 1)

    # Select the correct player
    return score if player == 'w' else -score
```

# 5 Symmetrie

Um das Caching noch effektiver zu gestalten, sollen neben Transpositionen auch Symmetrien erkannt werden. In diesem Kapitel werden alle Funktionen, die für die Symmetrieerkennung nötig sind, vorgestellt und implementiert.

Zunächst werden Hilfsfunktion definiert, die auf den gegebenen Spielbrettern (`boards`) eine bestimmte Symmetrie anwenden und alle resultierenden Spielbretter in einer Menge (`Set`) zurück geben. Schlussendlich werden alle Symmetrien nacheinander angewandt, damit auch zusammengesetzte Symetrien wie beispielsweise `Rotation um 90°` dann `Spiegelung an der horizontalen Achse` errechnet werden.

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

## 5.1 Rotation

Ein Spielbrett kann um 90°, 180° oder 270° gedreht werden, die resultierenden Spielbretter sind rotationssymmetrisch.

Die Eingabe besteht aus einer Menge von Spielbrettern (`boards`), die Ausgabe ist ebenfalls eine Menge, die alle Spielbretter enthält, die rotationssymmetrisch zu der Eingabe sind. Berechnet wird die Ausgabe indem alle Ringe um $k \in 2, 4, 6$ Zellen rotiert werden. Durch Aneinanderreihung der letzten $8 - k$ Zellen und der ersten $k$ Zellen kommt die Rotation zustande.

[ ]:
```
def symmetryRotation(boards):
    return {
        tuple(
            board[ring][rotation:] + board[ring][:rotation]
            for ring in range(3)
        )
        for rotation in range(2, 6+1, 2)
```

```
        for board in boards
    }
```

## 5.2 Spiegelung

Bei den Spiegelungen wird an vier Achsen gespiegelt:

- die *horizontale* und *vertikale* Achse, sowie
- die Diagonale von oben links nach unten rechts (*negative Diagonale*) und die Diagonale von unten links nach oben rechts (*positive Diagnonale*).

Diese Spiegelungen können einzelnd pro Ring vorgenommen werden, da der äußere Ring bleibt nach der Spiegelung weiterhin der äußere Ring. Gleiches gilt für die anderen Ringe. Alle Spiegelungen lassen sich durch eine Invertierung der Ringe und eine Rotation von $k \in 0, 2, 4, 6$ darstellen.

[ ]:
```
def symmetryHorizontal(boards):
    return {
        tuple(
            tuple(
                board[ring][(8-(cell+2))%8]
                for cell in range(8)
            )
            for ring in range(3)
        )
        for board in boards
    }
```

[ ]:
```
def symmetryVertical(boards):
    return {
        tuple(
            tuple(
                board[ring][(8-(cell+6))%8]
                for cell in range(8)
            )
            for ring in range(3)
        )
        for board in boards
    }
```

[ ]:
```
def symmetryDiagonalPositive(boards):
    return {
        tuple(
            tuple(
                board[ring][(8-(cell+4))%8]
                for cell in range(8)
            )
            for ring in range(3)
```

```
        )
        for board in boards
    }
```

[ ]:
```
def symmetryDiagnoalNegative(boards):
    return {
        tuple(
            tuple(
                board[ring][(8-cell)%8]
                for cell in range(8)
            )
            for ring in range(3)
        )
        for board in boards
    }
```

### 5.3 Ring-Tausch

Da der innere und der äußere Ring über symmetrische Kanten mit dem mittleren Ring verbunden ist, können der äußere und der innere Ringe getauscht werden. Dies funktioniert indem rückwärts über die Ringe iteriert wird.

[ ]:
```
def symmetryRing(boards):
    return {
        tuple(
            board[ring]
            for ring in reversed(range(3))
        )
        for board in boards
    }
```

### 5.4 Zusammenführung

Damit alle möglichen Symmetrien gefunden werden, wird jede Hilfsfunktion einzelnd auf alle vorherigen Spielbretter (`boards`) oder Zustände (`states`) angewandt. Dadurch sind auch zusammengesetzte Symmetrien wie beispielsweise `Rotation um 90°` dann `Spiegelung an der horizontalen Achse` möglich. Mit Hilfe einer Menge wird sichergestellt, dass keine Duplikate zurück gegeben werden.

[ ]:
```
def findSymmetries(state):
    stash, board = state

    boards = { board }
    boards |= symmetryRotation(boards)
    boards |= symmetryHorizontal(boards)
    boards |= symmetryVertical(boards)
    boards |= symmetryDiagonalPositive(boards)
```

```
    boards |= symmetryDiagnoalNegative(boards)
    boards |= symmetryRing(boards)

    return {
        (stash, board)
        for board in boards
    }
```

[ ]:
```
%run ./nmm-game.ipynb
%run ./nmm-artificial-intelligence.ipynb
%run ./nmm-heuristic.ipynb
import time
```

## 6 Minimax

Der *Minimax* Algorithmus bietet eine einfache Möglichkeit, um den perfekten Spielzug in einem Nullsummenspiel zu berechnen. Um diesen zu finden, wird der komplette Spielbaum vollständig per Tiefensuche durchsucht. Für einen Spielbaum mit der Tiefe `h` und dem Verzweigungsgrad `b` bedeutet das für die Zeit `t` und den Speicher `m`:

$$t_{Minimax} \in \mathcal{O}(b^h)$$

$$m_{Minimax} \in \mathcal{O}(b \cdot h)$$

Logischerweise ist *Minimax* somit nicht für die komplette Berechnung des Spiels geeignet, weil dies die üblicherweise zur Verfügung stehenden Ressourcen überschreitet. Aus diesem Grund wird die Tiefensuche auf eine maximale Tiefe `limit` beschränkt. Dies hat jedoch zur Folge, dass bei der maximalen Tiefe häufig noch kein eindeutiges Ergebnis *Mini* dem Wert `-1` (sichere Niederlage für `player`) oder *Max* mit dem Wert `1` (sicherer Sieg für `player`) erkannt werden konnte. Somit ist die Funktion `heuristic(state, player)` notwendig die in einem solchen Fall eine einfache heuristische Bewertung des `state` für den `player` durchführt und einen Wert $ -1 < value < 1 $ berechnet.

Für die Implementierung des *Minimax* Algorithmus wird nun eine Klasse `Minimax` implementiert, die von der zuvor definierten Klasse `ArtificialIntelligence` erbt. Hierzu wird die Funktion `bestMoves` überschrieben, die später genauer definiert wird, und der Konstruktor `__init__` implementiert. Der Konstruktor besitzt zwei optionale Parameter, die den Algorithmus konfigurieren können:

- Die `limit` Einstellung setzt die maximale Rekursionstiefe;
- Durch den `weights` Parameter können die Gewichtungen der zuvor definierten Heuristik bestimmt werden.

Zusätzlich initialisiert der Konstruktor das Attribut `cache` mit einem leeren `dict`. Dieses wird später als Transpositionstabelle verwendet.

[ ]:
```
class Minimax(ArtificialIntelligence):
    def __init__(self, limit=2, weights=None):
        self.cache = {}
```

```
        self.limit = limit
        self.weights = weights
        if self.weights is None:
            self.weights = HeuristicWeights()

    def bestMoves(self, state, player):
        pass
```

Für Debuggingzwecke wird eine `__repr__` Funktion implementiert, die eine String-Repräsentation aus einer `Minimax` Instanz mit allen Einstelungen erstellt.

[ ]:
```
def __repr__(self: Minimax):
    return f"Minimax(limit={self.limit}, weights={self.weights})"

Minimax.__repr__ = __repr__
del __repr__
```

Die Funktion `memoize` erwartet eine Funktion `f` als Parameter und überprüft, ob diese bereits mit den gleichen Parametern aufgerufen wurde. Falls ja, wird der gespeicherte Wert zurückgegeben. Falls nicht, werden die Parameter und das Ergebnis nach der Errechnung in die Transpositionstabelle gespeichert.

[ ]:
```
def memoize(f):
    def f_memoized(self, state, player, limit):
        key = (state, player, limit)

        if key in self.cache:
            self.cache_hit += 1
            return self.cache[key]

        result = f(self, state, player, limit)
        self.cache[key] = result

        self.cache_miss += 1
        return result

    return f_memoized
```

Die Funktion `value` nimmt einen Spielzustand `state` und liefert bei Ende des Spiels den Wert `value` mit Hilfe der `utility` Funktion für den Spieler `player`. Mit `limit` wird die Rekursionstiefe begrenzt. Beim Erreichen dieser wird die Funktion `heuristic` aufgerufen. In allen weiteren Fällen wird rekursiv in den nächsten möglichen Schritten nach dem maximal zu erreichenden Wert `value` gesucht. Zusammengefasst bedeutet das für die Funktion:

$$value(state, player, limit) = \begin{cases} utility(state, player) & falls\ \ finished(state, player) = true \\ heuristic(state, player) & falls\ \ limit = 0 \\ max(-value(ns, \neg player, limit)) & \\ \forall ns \in nextStates(state, player) & sonst \end{cases}$$

[ ]:
```
@memoize
def value(self, state, player, limit):
    if finished(state, player):
        return utility(state, player)
    if limit == 0:
        return heuristic(state, player, self.weights)
    return max([-self.value(ns, opponent(player), limit-1) for ns in␣
 ↪nextStates(state, player)])

Minimax.value = value
del value
```

Die Funktion `bestMoves` berechnet den geschätzten Wert aller möglichen Züge für einen Spieler `player` und wählt die Züge mit dem besten Wert aus. Dieser Wert wird innerhalb der maximalen Rekursionstiefe `limit` mit dem gegebenen Ausgangszustand `state` gesucht.

Zusätzlich werden Informationen gesammelt, die die genaueren Abläufe im Algorithmus abbilden:

- `runtime` ist die Rechendauer der Funktion in Sekunden;
- `limit` beschreibt die verwendete maximale Rekursionstiefe;
- `cache_hit` ist die Anzahl der Berechnungen, die durch die Transpositionstabelle (`cache`) eingespart werden konnten;
- `cache_miss` hingegen ist die Anzahl der Berechnungen, die trotz der Transpositionstablle durchgeführt werden mussten.

[ ]:
```
def bestMoves(self, state, player) -> BestMoves:
    # Start clock
    start = time.time()

    # Reset debug counter
    self.cache_hit = 0
    self.cache_miss = 0

    # Compute all expected values
    moves = [
        (-self.value(state, opponent(player), self.limit), state)
        for state in nextStates(state, player)
    ]
```

```
    maximum = max(value for (value, state) in moves)
    bestMoves = [state for (value, state) in moves if value == maximum]

    end = time.time()
    return BestMoves(
        bestMoves,
        maximum,
        # Collect debug information
        {   "runtime": end - start,
            "limit": self.limit,
            "cache_hit": self.cache_hit,
            "cache_miss": self.cache_miss,   }
    )

Minimax.bestMoves = bestMoves
del bestMoves
```

## 7 $\alpha$-$\beta$-Pruning

[ ]:
```
%run ./nmm-game.ipynb
%run ./nmm-artificial-intelligence.ipynb
%run ./nmm-heuristic.ipynb
%run ./nmm-symmetry.ipynb
import time
```

Für die Implementierung des *α-β-Pruning* Algorithmus wird eine Klasse `AlphaBetaPruning` implementiert, die ebenfalls von der zuvor definierten Klasse `ArtificialIntelligence` erbt. Auch hier wird die Funktion `bestMoves` überschrieben, die später genauer definiert wird, und den Konstruktor `__init__` implementiert. Der Konstruktor besitzt drei optionale Parameter, die den Algorithmus konfigurieren können:

- die `max_states` Einstellung setzt die maximale Anzahl an Zusänden, die betrachtet werden sollen;
- die `symmetry` Einstellung legt fest, ob alle symmetrischen Spielfelder zu einem Zustand berechnet werden sollen und dann ebenfalls in der Transpositionstabelle abgelegt werden sollen;
- durch den `weights` Parameter können die Gewichtungen der zuvor definierten Heuristik bestimmt werden.

Zusätzlich initialisiert der Konstruktor das Attribute `cache` mit einem leeren `dict`. Dieses wird später als Transpositionstabelle, bzw. als Sortierungsgrundlage in der Funktion `orderMoves` für *Iterative Deepening* verwendet.

[ ]:
```
class AlphaBetaPruning(ArtificialIntelligence):
    def __init__(self, max_states=25_000, symmetry=True, weights=None):
        self.cache = {}
```

```
        self.max_states = max_states
        self.symmetry = symmetry
        self.weights = weights
        if self.weights is None:
            self.weights = HeuristicWeights()

    def bestMoves(self, state, player):
        pass
```

Für Debuggingzwecke wird eine `__repr__` Funktion implementiert, die eine String-Repräsentation aus einer `AlphaBetaPruning` Instanz mit allen Einstelungen erstellt.

[ ]:
```
def __repr__(self: AlphaBetaPruning):
    return f"AlphaBetaPruning(max_states={self.max_states}, symmetry={self.
 ↪symmetry}, weights={self.weights})"

AlphaBetaPruning.__repr__ = __repr__
del __repr__
```

### 7.0.1 Iterative Tiefensuche

Die Iterative Tiefensuche bietet die Möglichkeit eine konstante Anzahl an Zuständen betrachten zu können. Dieses bietet den Vorteil gegenüber fest definierten Tiefenlimits, dass bei weniger komplexen Bäumen die Suche weiter fortgesetzt werden kann und die Antwortzeit an den Nutzer durch die gleiche Anzahl an Zuständen nahezu konstant bleibt. Dies sorgt gerade in Phase 2 des Mühle-Spiels dafür, dass die Spielzüge um einiges weiter im Voraus berechnet werden können. Andererseits verhindert es jedoch auch bei sehr komplexes Suchbäumen eine verlängerte Antwortzeit.

In der Umsetzung bedeutet das, dass die maximale Rekursionstiefe bei einem Aufruf der `bestMoves()`-Funktion zunächst immer bei `limit=1` liegt. Wenn nach der Betrachung aller Zustände mit Verwendung der Rekursionstiefe noch keine `IterativeMaxCountException()` geworfen wurde, wird das Maximum um eins erhöht.

Die gewünschte maximale Anzahl an Zuständen `maxCount` kann im `AlphaBetaPruning`-Konstruktor mit Hilfe der Option `max_states` festgelegt werden.

**Exception für das Erreichen der maximalen Zuständen** `IterativeMaxCountException`

Die Exception `IterativeMaxCountException` wird aufgerufen, wenn ein Fehler aufgrund des Erreichens der maximal zu betrachtenden Zustände bei der iterativen Tiefensuche erzeugt werden soll. Der Parameter `maxCount` im Konstruktor der Fehlermeldung nimmt dabei die erreichte maximale Anzahl an besuchten Zuständen an, um diese in der Fehlermeldung wiedergeben zu können.

[ ]:
```
class IterativeMaxCountException(Exception):
    def __init__(self, maxCount):
        super().__init__(f"Reached max count ({maxCount}) of iterative deepeing")
```

**Hilfsfunktion zum Überprüfen der maximal zu besuchenden Zustände** `checkMaxStates`

Die Hilfsfunktion `checkMaxStates` zählt die errechneten und besuchten Zustände für die Implementierung der *iterativen Tiefensuche*. Ist die Anzahl der maximal erlaubten Zustände überschritten, wird ein Fehler ausgelöst, damit der Algorithmus abbricht und ein Ergebnis zurück gegeben werden kann. Diese Ausnahmebetrachtung ist in der Funktion `bestMoves` implementiert.

[ ]:
```
def checkMaxStates(self):
    self.visited += 1
    if (self.visited >= self.max_states):
        raise IterativeMaxCountException(self.max_states)

AlphaBetaPruning.checkMaxStates = checkMaxStates
del checkMaxStates
```

**Sortieren der Zustände**

Die Funktion `orderMoves` sortiert eine Liste von Zuständen `states` für einen Spieler `player` entsprechend der für diese Zustände erreichten Werte aus der vorherigen Iteration der *Iterativen Tiefensuche*. So soll erreicht werden, dass vielversprechende Zustände früher betrachtet werden und sich somit Teilbäume eventuell früher abschneiden lassen.

[ ]:
```
def orderMoves(self, states, player, limit):
    return sorted(
        states,
        key     = lambda state: self.cache.get((state, player, limit-1), (0, -1,␣
 ↪1))[0],
        reverse = True
    )

AlphaBetaPruning.orderMoves = orderMoves
del orderMoves
```

### 7.0.2 Implementierung *α-β-Pruning*

**Hilfsfunktion zum Füllen der Transpositionstabelle** `writeCache`

Die Hilfsfunktion `writeCache` legt den gegebenen Zustand `state` und Spieler `player` mit samt der errechneten Werte `value`, `alpha` und `beta` in der Transpositionstabelle (`cache`) ab. Ist zusätzlich die Einstellung `symmetry` aktiviert, werden auch die errechneten Spielfelder in der Transpositionstabelle gespeichert.

[ ]:
```
def writeCache(self, state, player, value, alpha, beta, limit):
    if self.symmetry:
        for symmetricState in findSymmetries(state):
            self.cache[(symmetricState, player, limit)] = (value, alpha, beta)
    else:
        self.cache[(state, player, limit)] = (value, alpha, beta)
```

```
AlphaBetaPruning.writeCache = writeCache
del writeCache
```

**Hilfsfunktion zum Abgleich mit der Transpositionstabelle** `value`

Die `value` Funktion ist ein Wrapper für die eigentliche Implementierung des *α-β-Pruning* in der Funktion `alphaBeta`. Dieser Wrapper überprüft die Transpositionstabelle *cache* auf das Vorhandensein eines bereits berechneten Wertes für eine Kombination aus `state`, `player`, und `limit`. Ist ein Wert vorhanden, wird dieser auf seine Validität überprüft. Wenn diese Überprüfung fehlschlägt, oder kein Wert vorhanden ist, wird dieser mithilfe der `alphaBeta`-Funktion berechnet und in der Transpositionstabelle *cache* abgespeichert.

Ein Ergebnis aus dem Cache ist valide, solange das Intervall `alpha` und `beta` aus den Parametern innerhalb des im Cache verwendeten Intervalls `a` und `b` liegt. Also das Intervall des Caches muss genereller sein, als das Intervall aus den Parametern.

Für Auswertungszwecke wird außerdem bei jedem der drei möglichen Fälle ein Zähler erhöht.

[ ]:
```
def value(self, state, player, alpha, beta, limit):
    if (state, player, limit) in self.cache:
        (val, a, b) = self.cache[(state, player, limit)]
        if a <= alpha and beta <= b:
            self.cache_hit += 1
            return val
        else:
            alpha = min(alpha, a)
            beta  = max(beta , b)
            val   = self.alphaBeta(state, player, alpha, beta, limit)
            self.writeCache(state, player, val, alpha, beta, limit)
            self.cache_invalid += 1
            return val
    else:
        val = self.alphaBeta(state, player, alpha, beta, limit)
        self.writeCache(state, player, val, alpha, beta, limit)
        self.cache_miss += 1
        return val

AlphaBetaPruning.value = value
del value
```

**Funktion zur Berechnung des Wertes eines Spielzustands** `alphaBeta`

Die Funktion `alphaBeta` beinhaltet nun die eigentliche Implementierung des *α-β-Pruning*.

- Wie zuvor beim *Minimax-Algorithmus*, wird der `utility` Wert zurückgegeben, falls das Spiel in dem State `s` beendet (`finished`) ist.
- Ebenfalls äquivalent wird der `heuristic` Wert verwendet, sobald das Rekursionslimit (`limit`) erreicht wird.
- Zusätzlich wird mithilfe der Funktion `checkMaxStates` überprüft, ob die maximal zu betrachtende Anzahl an Zuständen erreicht wurde.

- Der eigentliche $\alpha$-$\beta$-Pruning Alogrithmus errechnet rekursiv mit Hilfe des Caches (`value`) den Wert eines Zuges. Hierbei wird der erste Wert der nächsten States verwendet, der größer oder gleich der oberen Grenze `beta` ist.

```
[ ]: def alphaBeta(self, state, player, alpha, beta, limit):
         self.checkMaxStates()

         if limit == 0:
             return heuristic(state, player, self.weights)

         states = nextStates(state, player)
         if finished(state, player, ns=states):
             return utility(state, player, ns=states)

         val = alpha
         for ns in self.orderMoves(states, player, limit):
             val = max(val, -self.value(ns, opponent(player), -beta, -alpha, limit-1))
             if val >= beta:
                 return val
             alpha = max(val, alpha)
         return val

     AlphaBetaPruning.alphaBeta = alphaBeta
     del alphaBeta
```

**Funktion zur Auswahl des bestmöglichen Zuges** `bestMoves`

Die Funktion `bestMoves` berechnet rekursiv den geschätzten Wert aller möglichen Züge für einen Spieler `player` und wählt die Züge mit dem besten Wert aus. Diese rekursive Suche wird solange ausgeführt, bis die maximale Anzahl der zu betrachtenden Zustände erreicht ist. Erreicht wird dies durch das Abfangen einer Exception, welche beim Erreichen dieser Grenze ausgelöst wird und somit die Endlosschleife beendet. Wenn für eine Rekursionstiefe alle Zustände betrachtet wurden, wird diese automatisch erhöht. Abschließend werden aus den erhaltenen Zuständen die Zustände mit dem besten Wert ausgewählt und zurückgegeben.

Zusätzlich werden Informationen gesammelt, die die genaueren Abläufe im Algorithmus abbilden:

- `runtime` ist die Rechendauer der Funktion in Sekunden;
- `limit` ist die erreichte maximale Rekursionstiefe;
- `visited` ist die Anzahl der besuchten Zustände;
- `max_states` ist die Anzahl der maximal zu besuchenden Zustände;
- `cache_hit` ist die Anzahl der Berechenungen, die durch die Tanspositionstabelle (`cache`) eingespart werden konnten;
- `cache_invalid` ist die Anzahl der Berechnungen, die durchgeführt werden mussten, da der in der Transpositionstabelle vorhandene Wert nicht zu verwenden war;

- `cache_miss` ist die Anzahl der Berechnungen, die trotz der Transpositionstabelle durchgeführt werden mussten, da kein Eintrag gefunden wurde.

[ ]:
```python
def bestMoves(self, state, player):
    # Start clock
    start = time.time()

    # Reset counter
    self.visited = 0
    self.cache_hit = 0
    self.cache_invalid = 0
    self.cache_miss = 0

    states = nextStates(state, player)
    moves = [(0, s) for s in states]

    limit = 1
    while True:
        try:
            moves = [
                (-self.value(s, opponent(player), -1, 1, limit), s)
                for s in states
            ]
            limit += 1
        except IterativeMaxCountException:
            break

    maximum = max(v for (v, s) in moves)
    bestMoves = [s for (v, s) in moves if v == maximum]

    end = time.time()
    return BestMoves(
        bestMoves,
        maximum,
        # Collect debug information
        {   "runtime": end - start,
            "limit": limit,
            "visited": self.visited,
            "max_states": self.max_states,
            "cache_hit": self.cache_hit,
            "cache_invalid": self.cache_invalid,
            "cache_miss": self.cache_miss,   }
    )

AlphaBetaPruning.bestMoves = bestMoves
del bestMoves
```

# 8 Hilfsfunktionen für die grafische Oberfläche

In diesem Notebook werden Hilfsfunktionen und Konstanten definiert, die für das Zeichnen und Spielen auf der grafischen Oberfläche benötigt werden.

[ ]:
```
import math
```

Da das Spiel in der GUI in einem seperatem Thread läuft, werden Fehler oder Warnungen nicht in der Jupyter-Notebook geloggt. Deswegen wird zu Debugzwecken ein Datei-Logger implementiert. Dieser wird im Rahmen der Arbeit aber nicht weiter erläutert.

[ ]:
```
import logging

logger = logging.getLogger('GUI')
logger.setLevel(logging.DEBUG)
fh = logging.FileHandler('log.txt')
fh.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
fh.setFormatter(formatter)
logger.addHandler(fh)
```

## 8.1 Konstanten

Um eine einheitliche GUI zur Verfügung zu stellen, die leicht zu warten ist, werden im Folgenden einige Konstanten definiert.

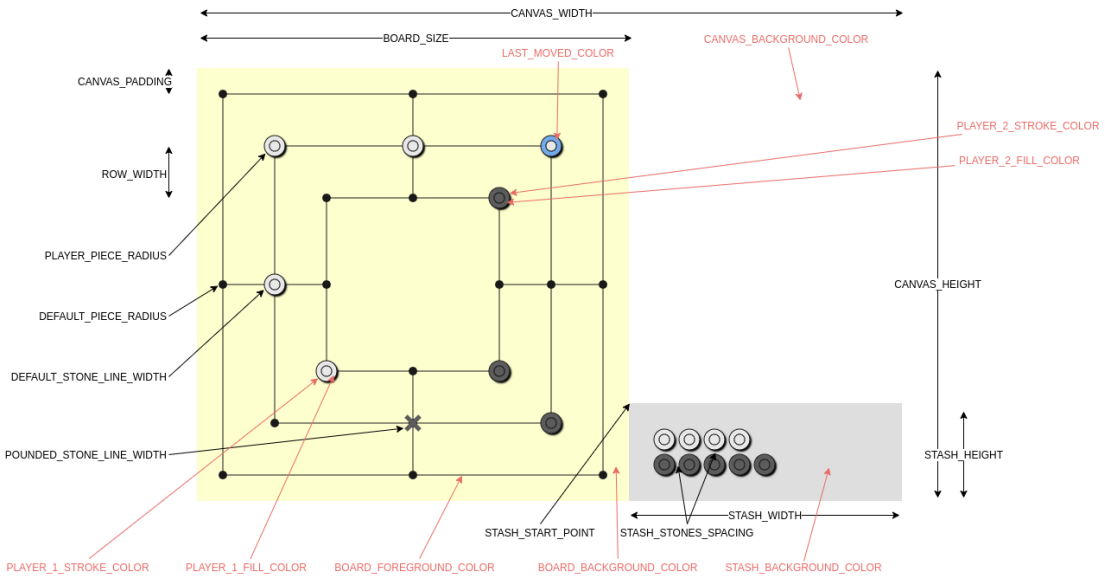
### 8.1.1 Spieler

Die Konstanten dienen dazu, die Strings der Spieler bzw. des leeren Feldes zu definieren.

[ ]:
```
NO_PLAYER = ' '
PLAYER_1 = 'w'
PLAYER_2 = 'b'
```

### 8.1.2 Brett

In der Abbildung sind alle Konstanten für die Zeichenfläche angegeben, um die Bedeutung der einzelnen Konstanten besser zeigen zu können. Zusätzlich sind in der Abbildung alle Farbkonstanten rotmarkiert.




[ ]:
```
BOARD_SIZE = 500
CANVAS_PADDING = 30
ROW_WIDTH = 60
PLAYER_PIECE_RADIUS = 12
DEFAULT_PIECE_RADIUS = 5

DEFAULT_STONE_LINE_WIDTH = 1
SELECTED_STONE_LINE_WIDTH = 3
POUNDED_STONE_LINE_WIDTH = 5

STASH_STONES_SPACING = 5

STASH_HEIGHT = ((PLAYER_PIECE_RADIUS * 2 ) * 2) + (STASH_STONES_SPACING) +␣
 ↪(CANVAS_PADDING * 2)
STASH_WIDTH = ((PLAYER_PIECE_RADIUS * 2 ) * 9) + (STASH_STONES_SPACING * 8) +␣
 ↪(CANVAS_PADDING * 2)

CANVAS_HEIGHT = BOARD_SIZE
CANVAS_WIDTH = 1500

STASH_STARTING_POINT_X = BOARD_SIZE
STASH_STARTING_POINT_Y = CANVAS_HEIGHT - STASH_HEIGHT
```

### 8.1.3 Farben

Die Farben für das Spiel werden im Folgenden definiert. In der obigen Abbildung sind diese genauer erklärt.

[ ]:
```
CANVAS_BACKGROUND_COLOR = '#ffffff'
BOARD_FOREGROUND_COLOR = '#191919'
```

```
BOARD_BACKGROUND_COLOR = '#ffffcb'
STASH_BACKGROUND_COLOR = '#dedede'
PLAYER_1_FILL_COLOR = '#E8E8E8'
PLAYER_1_STROKE_COLOR = '#191919'
PLAYER_2_FILL_COLOR = '#5c5c5c'
PLAYER_2_STROKE_COLOR = '#191919'
LAST_MOVED_COLOR = '#61aced'
```

#### 8.1.4 Text

In der GUI gibt es drei verschiedene Textarten:

- Spielnachricht (wer spielt gerade oder ob das Spiel beendet ist),
- Hinweise (falls zum Beispiel eine ungültige Aktion ausgeführt worden ist) und - Informationen über den letzten Zug der künstlichen Intelligenz.

Diese Textarten haben eine eigene Schriftart, -größe und -farbe.

[ ]:
```
TEXT_X = BOARD_SIZE + CANVAS_PADDING
TEXT_Y = CANVAS_PADDING
TEXT_MAX_WIDTH = CANVAS_WIDTH - BOARD_SIZE - 2 * CANVAS_PADDING
TEXT_VERTICAL_PADDING = 30
TEXT_MSG_FONT = '18px sans-serif'
TEXT_MSG_COLOR = '#333333'
TEXT_HINT_FONT = '14px sans-serif'
TEXT_HINT_COLOR = '#c75528'
TEXT_INFO_FONT = '14px mono'
TEXT_INFO_COLOR = '#333333'
```

#### 8.1.5 Schatten

Um das Spiel dynamischer zu gestalten, haben Spielsteine einen Schatten.

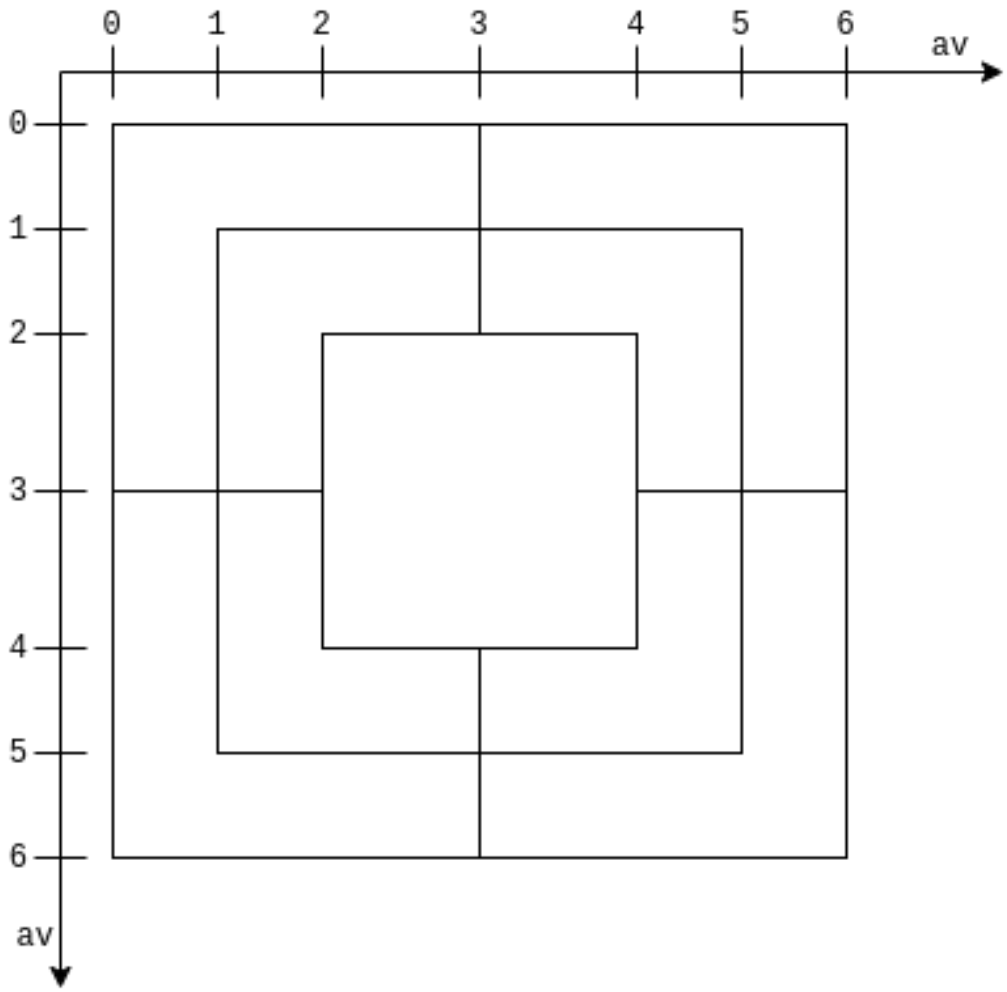
[ ]:
```
SHADOW_COLOR_ENABLED = '#000000'
SHADOW_OFFSET_X_ENABLED = 2
SHADOW_OFFSET_Y_ENABLED = 2
SHADOW_BLUR_ENABLED = 2

SHADOW_COLOR_DISABLED = 'rgba(0, 0, 0, 0)'
SHADOW_OFFSET_X_DISABLED = 0
SHADOW_OFFSET_Y_DISABLED = 0
SHADOW_BLUR_DISABLED = 0
```

#### 8.1.6 Berechnete Werte

Die Koordinaten für die Knoten lassen sich aus den obrigen Konstanten berechnen. Da das Mühlespielbrett horizontal und vertikal identisch ist, werden für die Koordinaten auf der x- und y-

Achse die gleichen Werte benötigt, die mit *av* (für *available values*) bezeichnet werden. Es werden insgesamt sieben Werte $av_0$ bis $av_6$ benötigt, die in der folgenden Abbildung dargestellt werden.




Die Werte lassen sich wie folgt berechnen.

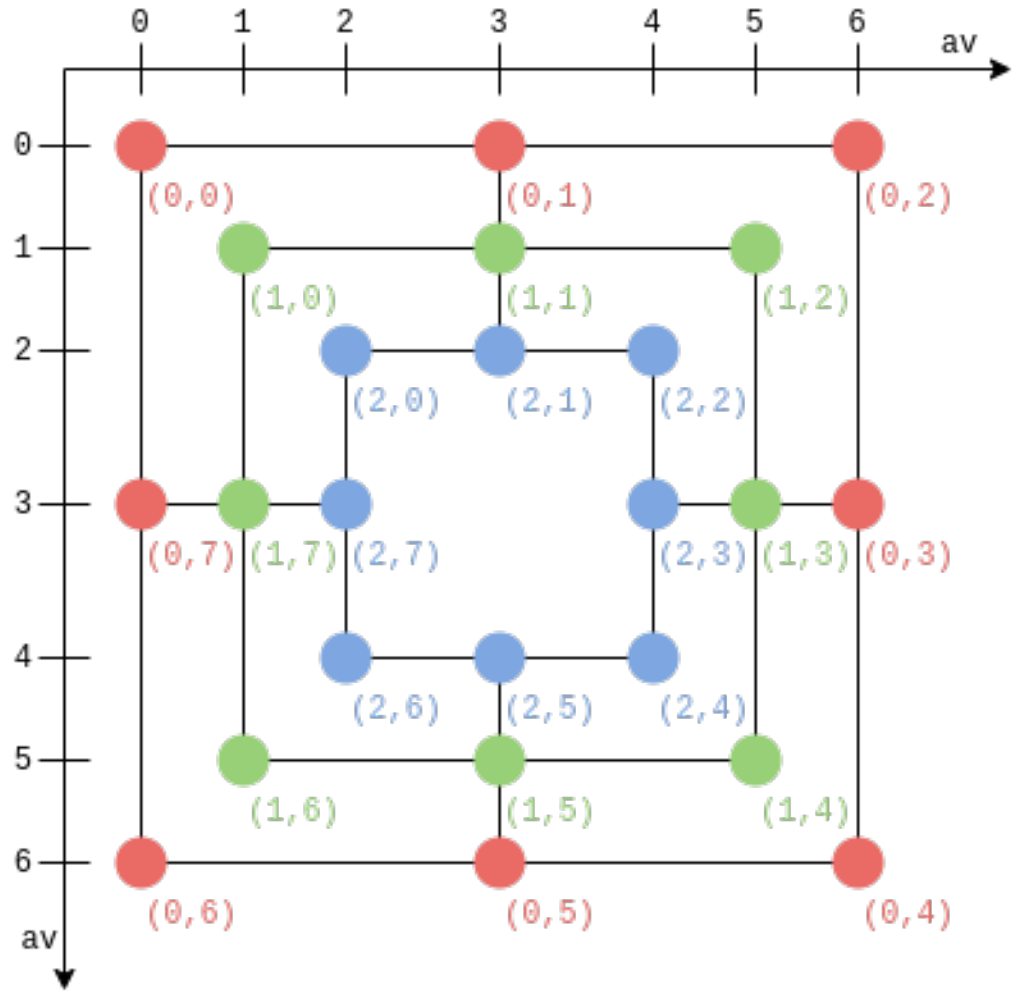
$$av_0 = CANVAS\_PADDING$$
$$av_1 = CANVAS\_PADDING + ROW\_WIDTH$$
$$av_2 = CANVAS\_PADDING + 2 \cdot ROW\_WIDTH$$
$$av_3 = \frac{BOARD\_SIZE}{2}$$
$$av_4 = BOARD\_SIZE - (CANVAS\_PADDING + 2 \cdot ROW\_WIDTH)$$
$$av_5 = BOARD\_SIZE - (CANVAS\_PADDING + ROW\_WIDTH)$$
$$av_6 = BOARD\_SIZE - CANVAS\_PADDING$$

[ ]:
```
av = (
    math.floor(CANVAS_PADDING),
```

```
    math.floor(CANVAS_PADDING + ROW_WIDTH),
    math.floor(CANVAS_PADDING + 2 * ROW_WIDTH),
    math.floor(BOARD_SIZE / 2),
    math.floor(BOARD_SIZE - (CANVAS_PADDING + 2 * ROW_WIDTH)),
    math.floor(BOARD_SIZE - (CANVAS_PADDING + ROW_WIDTH)),
    math.floor(BOARD_SIZE - CANVAS_PADDING)
)
```

### 8.1.7 Koordinaten

Die Koordinaten der Knoten sind in dem zweidimensionalen Tupel `coords` definiert. Zuerst wird der Ring definiert, von außen nach innen. Danach die Position im Ring, beginnend von oben links und dann im Uhrzeigersinn. In der Abbildung sind die Knoten mit den Koordinaten dargestellt. Die Werte von den Koordinaten sind die x- und y-Werte auf der Zeichenfläche, definiert in `av`.




[ ]:
```
coords = (
    (
```

```
        (av[0], av[0]),
        (av[3], av[0]),
        (av[6], av[0]),
        (av[6], av[3]),
        (av[6], av[6]),
        (av[3], av[6]),
        (av[0], av[6]),
        (av[0], av[3])
    ),
    (
        (av[1], av[1]),
        (av[3], av[1]),
        (av[5], av[1]),
        (av[5], av[3]),
        (av[5], av[5]),
        (av[3], av[5]),
        (av[1], av[5]),
        (av[1], av[3])
    ),
    (
        (av[2], av[2]),
        (av[3], av[2]),
        (av[4], av[2]),
        (av[4], av[3]),
        (av[4], av[4]),
        (av[3], av[4]),
        (av[2], av[4]),
        (av[2], av[3])
    )
)
```

## 8.2 Funktionen zum Zeichnen

Die grafische Oberfläche (englisch *graphical user interface*, GUI) wird mit dem Python-Modul ipycanvas aufgebaut. Dieses Modul ermöglicht die Verwendung einer interaktiven Zeichenfläche zum Zeichnen von 2D-Objekten in IPython. Es bringt eine Reihe von Funktionen mit, um einfache Formen zeichnen zu können. Gezeichnet wird auf einem 2D-Canvas-Objekt mit den Startkoordinaten `(0,0)` oben links.

[ ]:
```
import ipycanvas
```

Die Funktion `toggleShadow` schaltet den Schatten auf einem gegebenen Canvas ein und aus. Sie hat folgende Eingabeparameter:

- `c` ist eine Referenz auf ein Canvas-Objekt.
- `enable` ist ein boolischer Wert, der angibt, ob Schatten auf dem Canvas `c` ein oder ausgeschaltet werden soll.

Wird der Schatten eingeschaltet, werden die Schatteneigenschaften des Canvas mit den oben definierten Konstanten gesetzt. Andernfalls werden die Standardwerte von *ipycanvas* gesetzt, was bedeutet, der Schatten wird deaktiviert.

[ ]:
```
def toggleShadow(c, enable):
    c.shadow_color    = SHADOW_COLOR_ENABLED    if enable else␣
 ↪SHADOW_COLOR_DISABLED
    c.shadow_offset_x = SHADOW_OFFSET_X_ENABLED if enable else␣
 ↪SHADOW_OFFSET_X_DISABLED
    c.shadow_offset_y = SHADOW_OFFSET_Y_ENABLED if enable else␣
 ↪SHADOW_OFFSET_Y_DISABLED
    c.shadow_blur     = SHADOW_BLUR_ENABLED     if enable else␣
 ↪SHADOW_BLUR_DISABLED
```

Die Funktion `drawCircle` dient zum Zeichnen eines Kreises auf einem Zeichenfeld. Die Funktion hat vier Argumente und drei optionale Parameter:

- `c` ist eine Referenz auf ein Canvas-Objekt, auf dem der Kreis gezeichnet werden soll.
- `coords` ist die Koordinate des Mittelpunktes des Kreies.
- `radius` ist der Radius des Kreises.
- `color` gibt die Farbe des Kreises an.
- `strokeColor` ist ein optionaler Parameter, der die Farbe der Umrandung angibt. Der Standardwert ist `None`. In dem Fall wird der Kreis nicht umrandet.
- `lineWidth` ist ein optionaler Parameter, der die Liniendicke angibt. Der Standardwert ist in der Kontante `DEFAULT_STONE_LINE_WIDTH` definiert.
- `useShadow` ist ein optionaler, boolischer Wert. Wenn er gesetzt ist, wird ein Schatten von dem Kreis gemalt. Standardmäßig ist der Wert `False`.

[ ]:
```
def drawCircle(
        c,
        coords,
        radius,
        color,
        strokeColor = None,
        lineWidth = DEFAULT_STONE_LINE_WIDTH,
        useShadow = False):
    if useShadow:
        toggleShadow(c, True)
    c.fill_style = color
    c.fill_arc(coords[0], coords[1], radius, 0, 2 * math.pi)
    if useShadow:
        toggleShadow(c, False)
    if strokeColor is not None:
        c.line_width = lineWidth
        c.stroke_style = strokeColor
        c.stroke_arc(coords[0], coords[1], radius, 0, 2 * math.pi)
```

Die Funktion `drawStone` dient zum Zeichnen eines Steines auf einem Zeichenfeld mit Hilfe der

Funktion `drawCircle`. Ein Spielstein besteht aus zwei Kreisen und ein leerer Knoten (also wo sich kein Spieler befinden) aus einem Kreis.

Die Funktion hat drei Argumente und zwei optionale Parameter:

- `c` ist eine Referenz auf ein Canvas-Objekt, auf dem der Stein gezeichnet werden soll;
- `coords` ist die Koordinate des Mittelpunktes des Steines;
- `player` gibt den Spieler an;
- `selected` ist ein optionaler, boolischer Wert, der angibt, ob ein Spielerstein ausgewählt ist oder nicht. Der Standardwert ist `False`;
- `lastMoved` ist ein optionaler, boolischer Wert, der angibt ob der zu zeichnende Spielerstein zuletzt bewegt worden ist. Der Standardwert ist `False`.

```
[ ]: def drawStone(c, coords, player, selected = False, lastMoved = False):
         if player == NO_PLAYER:
             drawCircle(c, coords, DEFAULT_PIECE_RADIUS, BOARD_FOREGROUND_COLOR)
         else:
             color       = PLAYER_1_FILL_COLOR   if player == PLAYER_1 else␣
      ↪PLAYER_2_FILL_COLOR
             strokeColor = PLAYER_1_STROKE_COLOR if player == PLAYER_1 else␣
      ↪PLAYER_2_STROKE_COLOR

             lineWidth = SELECTED_STONE_LINE_WIDTH if selected else␣
      ↪DEFAULT_STONE_LINE_WIDTH
             drawCircle(
                 c,
                 coords,
                 PLAYER_PIECE_RADIUS,
                 LAST_MOVED_COLOR if lastMoved else color,
                 strokeColor,
                 lineWidth,
                 useShadow = True)
             drawCircle(
                 c,
                 coords,
                 math.floor(PLAYER_PIECE_RADIUS / 2),
                 color,
                 strokeColor,
                 lineWidth)
```

Die Funktion `drawPoundedStone` dient zum Zeichnen eines geschlagenden Steines auf einem Zeichenfeld. Ein geschlagender Stein wird durch ein Kreuz in der GUI dargestellt.

Die Funktion hat drei Argumente:

- `c` ist eine Referenz auf ein Canvas-Objekt, auf dem der geschlagende Stein gezeichnet werden soll;
- `coords` ist die Koordinate des Mittelpunktes des geschlagenden Steines;
- `player` gibt den Spieler des Steines an.

```
[ ]: def drawPoundedStone(c, coords, player):
         x, y = coords
         offset = int(math.sin(0.25*math.pi) * PLAYER_PIECE_RADIUS)
         c.line_width = POUNDED_STONE_LINE_WIDTH
         c.stroke_style = PLAYER_1_FILL_COLOR   if player == PLAYER_1 else␣
     ↪PLAYER_2_FILL_COLOR
         with ipycanvas.hold_canvas(c):
             c.stroke_line(x - offset, y - offset, x + offset, y + offset)
             c.stroke_line(x - offset, y + offset, x + offset, y - offset)
```

Die Funktion `drawText` zeichnet einen Text auf einer Zeichenfläche. Die Funktion hat zwei Argumente und zwei optionale Parameter:

- `c` ist eine Referenz auf ein Canvas-Objekt, auf dem der Text gezeichnet werden soll.
- `msg` ist die Nachricht, die auf dem Canvas geschrieben werden soll.
- `hint` ist ein optionaler String, der ein Hinweis oder eine Warnung ist. Standardmäßig ist die Variable auf `None` gesetzt.
- `information` ist ein optionales Dictionary. Alle Einträge des Dictionaries werden als Information auf dem Zeichenfeld ausgegeben.

Bei jedem Funktionsaufruf wird am Anfang der Inhalt der Zeichenfläche gelöscht, sodass sich immer nur eine Version der Texte auf der Zeichenfläche befindet.

```
[ ]: def drawText(c, msg, hint = None, information = None):
         with ipycanvas.hold_canvas(c):
             c.clear()
             y = TEXT_Y
             c.font = TEXT_MSG_FONT
             c.fill_style = TEXT_MSG_COLOR
             c.fill_text(msg, TEXT_X, y, max_width = TEXT_MAX_WIDTH)
             y += TEXT_VERTICAL_PADDING
             if hint is not None:
                 c.font = TEXT_HINT_FONT
                 c.fill_style = TEXT_HINT_COLOR
                 c.fill_text('Hint: ' + hint, TEXT_X, y, max_width = TEXT_MAX_WIDTH)
                 y += TEXT_VERTICAL_PADDING
             if information is not None:
                 c.font = TEXT_INFO_FONT
                 c.fill_style = TEXT_INFO_COLOR
                 c.fill_text('Information from last move:', TEXT_X, y, max_width =␣
     ↪TEXT_MAX_WIDTH)
                 y += TEXT_VERTICAL_PADDING
                 for (key, value) in information.items():
                     c.fill_text(f'   {key.ljust(16)} {value}', TEXT_X, y, max_width␣
     ↪= TEXT_MAX_WIDTH)
                     y += TEXT_VERTICAL_PADDING
```

Die Funktion `constructLine` dient zum Konstruieren einer Linie auf einem Zeichenfeld. Die

Funktion hat drei Eingabeparameter:

- `c` ist eine Referenz auf ein Canvas-Objekt, auf dem die Linie gezeichnet werden soll.
- `start` ist die Koordinate des Startpunktes der Linie.
- `end` ist die Koordinate des Endpunktes der Linie.

Die Funktion aktualisiert einen Pfad auf dem Zeichenfeld, aber sie zeichnet noch nicht den aktualisierten Pfad.

[ ]:
```
def constructLine(c, start, end):
    c.move_to(start[0], start[1])
    c.line_to(end[0], end[1])
```

Die Funktion `constructSquare` konstruiert ein Quadrat auf einer Zeichenfläche und lässt es mit Hilfe der Funktion `constructLine` zeichnen. Die Funktion hat zwei Eingabeargumente:

- `c` ist eine Referenz auf ein Canvas-Objekt, auf dem das Quadrat gezeichnet werden soll.
- `ring` ist ein Acht-Tupel, das die Koordinaten eines Ringes enthält.

[ ]:
```
def constructSquare(c, ring):
    for i in range(4):
        start = i * 2
        end = (i * 2 + 2) if (i * 2 + 2 <= 6) else 0
        constructLine(c, ring[start], ring[end])
```

Die Funktion `constructCrossLines` konstruiert die Querlinien des Mühlespiels auf einer Zeichenfläche und lässt es mit Hilfe der Funktion `constructLine` zeichnen. Die Funktion hat zwei Eingabeargumente:

- `c` ist eine Referenz auf ein Canvas-Objekt, auf dem die Querlinien gezeichnet werden sollen.
- `coords` ist ein zweidimensionales Tupel, welches alle Koordinaten des Spielbrettes enthält (vgl. das Kapitel *Koordinaten* in der GUI).

[ ]:
```
def constructCrossLines(c, coords):
    for i in range(4):
        k = i * 2 + 1
        constructLine(c, coords[0][k], coords[2][k])
```

Die Funktion `setupCanvas` erstellt das Canvas-Objekt und zeichnet den Hintergrund des Spielfeldes. Die Funktion hat keine Eingabeparameter und gibt eine Referenz auf das erstellte Canvas-Objekt zurück.

Die Zeichenfläche besteht aus einem MultiCanvas-Objekt mit drei Ebenen:

- Der Hintergrund, der das Spielbrett mit den Linien darstellt;
- Auf der zweiten Ebene wird der Text für das Spiel geschrieben;
- Die Spielsteine werden auf der obersten Ebene gezeichnet.

[ ]:
```
def setupCanvas():
    canvas = ipycanvas.MultiCanvas(3, width = CANVAS_WIDTH, height =␣
 ↪CANVAS_HEIGHT)
```

```
    with ipycanvas.hold_canvas(canvas[0]):

        canvas[0].fill_style = CANVAS_BACKGROUND_COLOR
        canvas[0].fill_rect(0, 0, CANVAS_WIDTH, CANVAS_HEIGHT)

        canvas[0].fill_style = BOARD_BACKGROUND_COLOR
        canvas[0].fill_rect(0, 0, BOARD_SIZE, BOARD_SIZE)

        canvas[0].fill_style = STASH_BACKGROUND_COLOR
        canvas[0].fill_rect(STASH_STARTING_POINT_X, STASH_STARTING_POINT_Y,␣
↪STASH_WIDTH, STASH_HEIGHT)


        canvas[0].stroke_style = BOARD_FOREGROUND_COLOR
        canvas[0].begin_path()

        for i in range(3):
            constructSquare(canvas[0], coords[i])

        constructCrossLines(canvas[0], coords)
        canvas[0].stroke()

    return canvas
```

Die Funktion `updateGui` dient zum Aktualisieren der Zeichenfläche für einen gegebenen Spielzustand. Die Funktion hat zwei Argumente und drei optionale Parameter:

- `c` ist eine Referenz auf ein Canvas-Objekt;
- `state` ist der Spielzustand, der in der GUI angezeigt werden soll;
- `selectedStone` ist eine Koordinate von einem selektierten Stein. Der Standardwert ist `None`. Ist in der Phase 2 oder 3 ein Stein auswählt, kann er mit diesem optionalen Parameter auf der Zeichenfläche hervorgehoben werden;
- `movedStone` ist die Koordinate des zuletzt bewegten Steins, um diesen in der GUI entsprechend zu markieren. Der Standardwert ist `None`, was bedeutet, das kein Stein bewegt worden ist;
- `poundedStones` ist eine Menge von den geschlagenden Steinen des Gegenspielers. Diese werden in der GUI als ein Kreuz dargestellt. Standardmäßig ist `poundedStones` eine leere Menge, was bedeutet, das kein Stein geschlagen worden ist.

[ ]:
```
def updateGui(c, state, selectedStone = None, movedStone = None, poundedStones =␣
↪set()):
    with ipycanvas.hold_canvas(c):
        c.clear()
        ((stashP1, stashP2), squares) = state

        # update pieces on the board
        for i in range(len(squares)):
            for j in range(len(squares[i])):
```

```
                drawStone(
                    c,
                    coords[i][j],
                    squares[i][j],
                    selected = selectedStone == (i, j),
                    lastMoved = movedStone == (i, j))

        for (player, (i, j)) in poundedStones:
            drawPoundedStone(c, coords[i][j], player)

        # update pieces on the stash

        # player 1

        x = STASH_STARTING_POINT_X + PLAYER_PIECE_RADIUS
        y = STASH_STARTING_POINT_Y + CANVAS_PADDING + PLAYER_PIECE_RADIUS

        for i in range(stashP1):
            x += 2 * PLAYER_PIECE_RADIUS + STASH_STONES_SPACING
            drawStone(c, (x, y), PLAYER_1)

        # player 2

        x = STASH_STARTING_POINT_X + PLAYER_PIECE_RADIUS
        y = STASH_STARTING_POINT_Y + CANVAS_PADDING + 3 * PLAYER_PIECE_RADIUS +␣
↪STASH_STONES_SPACING

        for i in range(stashP2):
            x += 2 * PLAYER_PIECE_RADIUS + STASH_STONES_SPACING
            drawStone(c, (x, y), PLAYER_2)
```

## 8.3 Hilfsfunktionen für das Spielen in der GUI

In diesem Kapitel werden Funktionen deklariert, die Hilfsfunktionen für die GUI darstellen, aber unabhängig von dem eigentlichen Spielzustand sind.

Zusätzlich werden die Jupyter-Notebooks von dem Minimax- und Alpha-Beta-Pruning-Algorithmus benötigt und hier ausführt.

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

Die Funktion `getClickedStone` dient zum Ermitteln, ob auf der Zeichenfläche eine Ecke angeklickt worden ist, auf dem ein Stein stehen kann. Die Funktion hat zwei Argumente:

- `x` für den Wert auf der horizontalen Achse;
- `y` für den Wert auf der vertikalen Achse.

Es müssen nicht die genauen Werte für x und y angeklickt werden, sondern es gibt einen Puffer in Höhe des Radius von einem Spielerstein. Falls eine Position für einen Stein angeklickt worden ist, für die jeweilige Koordinate aus dem `coords`-Tupel zurückgegeben. Falls keine Position gefunden worden ist, wird `None` zurückgegeben.

[ ]:
```
def getClickedStone(x, y):
    for value in av:
        if value - PLAYER_PIECE_RADIUS <= x <= value + PLAYER_PIECE_RADIUS:
            x = value
        if value - PLAYER_PIECE_RADIUS <= y <= value + PLAYER_PIECE_RADIUS:
            y = value

    for i in range(len(coords)):
        for j in range(len(coords[i])):
            if coords[i][j] == (x, y):
                return (i, j)
    return None
```

Die Funktion `getChangedStones` ermittelt den zuletzt bewegten Stein eines Spielers und alle geschlagenden Steine des Gegenspielers zwischen zwei Zuständen. Die Funktion hat drei Argumente:

- `oldState` ist der Ausgangszustand;
- `newState` ist der neue Zustand;
- `player` ist der Spieler, den den Zug gespielt hat.

Die Funktion gibt ein Zwei-Tupel der Form `<movedStone, poundedStones>` mit

1. `movedStone` ist die Koordinate des bewegten Steins des Spielers;
2. `poundedStones` ist eine Menge von Zwei-Tupeln der Form `<op, coord>` mit
3. `op` ist der Gegenspieler;
4. `coord` ist die Koordinate des geschlagenden Steines;

zurück.

[ ]:
```
def getChangedStones(oldState, newState, player):
    (_, oldBoard) = oldState
    (_, newBoard) = newState
    op = opponent(player)
    movedStone = None;
    poundedStones = set()

    for i in range(len(oldBoard)):
        for j in range(len(oldBoard[i])):
            if oldBoard[i][j] != newBoard[i][j]:
                if newBoard[i][j] == player:
                    movedStone = (i, j)
                if oldBoard[i][j] == op:
```

```
                    poundedStones |= { (op, (i, j)) }
    return (movedStone, poundedStones)
```

# 9 Grafische Oberfläche

In diesem Notebook ist das Spielen in der GUI implementiert. Die Hilfsfunktionen sind in dem Notebook nmm-game-utils definiert.

[ ]:
```
%run ./nmm-gui-utils.ipynb
```

## 9.1 Klasse GameState

Die Klasse `GameState` dient zum Spielen und Verwalten von einem Mühle-Spiel in der GUI.

Der Konstruktur der Klasse hat elf Eingabeparameter, die alle optional sind:

- `state` ist der Startzustand für das Spiel. Standardmäßig wird das Spiel mit `s0` gestartet, welches ein leeres Spielfeld darstellt;
- `player` definiert den Spieler, der den ersten Zug spielt. Standardmäßig wird das Spiel mit dem weißen Spieler (`w`) gestartet;
- `algorithm1` definiert, ob und welcher Algorithmus für den weißen Spieler spielt. Standardmäßig wird der Spieler von einem Menschen gespielt (`None`). Für *α-β-Pruning* ist eine Instanz der Klasse `AlphaBetaPruning` zu übergeben und für *Minimax* eine Instanz der Klasse `Minimax`.
- `algorithm2` definiert, ob und welcher Algorithmus für den schwarzen Spieler spielt. Standardmäßig wird der Spieler von einem Menschen gespielt (`None`). Für *α-β-Pruning* ist eine Instanz der Klasse `AlphaBetaPruning` zu übergeben und für *Minimax* eine Instanz der Klasse `Minimax`.
- `timeout` definiert einen Timeout in Sekunden, der nach einem Computer-Zug gesetzt wird. Dies dient dazu, die Übersichtlichkeit zu erhöhen, wenn Computer gegen Computer spielt. Standardmäßig gibt es keinen Timeout (`None`);
- `stepwise` ist ein boolischer Wert und gibt an, ob das Spiel bei Computer gegen Computer im Einzelschrittmodus gespielt wird. Das bedeutet, nach jedem Computerzug muss der nächste Computerzug manuell begonnen werden. Standardmäßig ist der Einzelschrittmodus deaktiviert (`False`);
- `limitMovesWithoutMill` ist eine Ganzzahl, die angibt, wie viele Züge ohne geschlagende Mühle gespielt werden können. Ist das Limit überschritten, endet das Spiel in einem Unentschieden. Der Standardwert ist `30`. Um dieses Limit auszuschalten, muss der Wert auf `None` gesetzt werden;
- `limitStatesCounter` ist eine Ganzzahl, die angibt, wie oft ein gleicher Zustand gespielt werden kann. Ist das Limit überschritten, endet das Spiel in einem Unentschieden. Der Standardwert ist `5`. Um dieses Limit auszuschalten, muss der Wert auf `None` gesetzt werden;

```
[ ]: from collections import defaultdict

class GameState:
    def __init__(self,
                 state = s0,
                 player = PLAYER_1,
                 algorithm1 = None,
                 algorithm2 = None,
                 timeout = None,
                 stepwise = False,
                 limitMovesWithoutMill = 30,
                 limitStatesCounter = 5):

        self.algorithm1 = algorithm1
        self.algorithm2 = algorithm2

        self.state = state
        self.player = player
        self.canvas = setupCanvas()
        self.winner = None
        self.information = None
        self.resetStateVariables()

        self.timeout = timeout
        self.stepwise = stepwise
        self.limitMovesWithoutMill = limitMovesWithoutMill
        self.movesWithoutMill = 0
        self.limitStatesCounter = limitStatesCounter
        self.statesCounter = defaultdict(int)

        self.pause = (self.player == PLAYER_1 and self.algorithm1) or (self.
 ↪player == PLAYER_2 and self.algorithm2)
        if self.pause:
            self.hint = 'Please click to start the game.'

        self.canvas[2].on_mouse_up(self.handleGame)

        updateGui(self.canvas[2], self.state)
        logger.info('game state initalized')
        self.updateText()
```

Die Funktion `resetStateVariables` dient zum Zurücksetzen der temporären Hilfsvariablen der Klasse `GameState`. Diese Funktion hat weder Ein- noch Ausgabe.

```
[ ]: def resetStateVariables(self):
    self.stateTemp = None
    self.millsToPound = 0
```

```
    self.selectedStone = None
    self.hint = None

GameState.resetStateVariables = resetStateVariables
del resetStateVariables
```

Die Funktion `handleGame` steuert den Ablauf des Spiels. Die Funktion wird bei jedem Mausklick auf das Canvas-Objekt von dem Event `on_mouse_up` aufgerufen. Die Funktion hat zwei Argumente:

- `x` ist relative Wert der Maus zu dem Canvas-Objekt auf der horizontalen Achse;
- `y` ist relative Wert der Maus zu dem Canvas-Objekt auf der vertikalen Achse.

[ ]:
```
def handleGame(self, x, y):
    if self.winner is not None:
        logger.warning('Game has ended!')
        return

    if self.pause:
        self.pause = False
        self.hint = None

        self.updateText()
        self.checkForComputerStep()

        return

    phase = playerPhase(self.state, self.player)
    logger.info(f'player phase: {phase}')

    stone = getClickedStone(x, y)

    if stone is None:
        logger.warning('No stone was clicked!')
        if self.selectedStone is not None and self.millsToPound <= 0:
            self.cancelStep()
    elif self.millsToPound > 0:
        self.poundMillInGui(stone)
    elif self.selectedStone is not None:
        self.moveStone(stone)
    elif phase == 1:
        self.placeStone(stone)
    elif phase == 2 or phase == 3:
        self.selectStone(stone)

    self.information = None
    self.updateText()
```

```
    self.checkForComputerStep()

GameState.handleGame = handleGame
del handleGame
```

Die Funktion `togglePlayer` tauscht den Spieler, der den nächsten Zug spielt.

[ ]:
```
def togglePlayer(self):
    self.player = opponent(self.player)

GameState.togglePlayer = togglePlayer
del togglePlayer
```

Die Funktion `playNewState` spielt einen vollständigen Zug in der GUI. Dafür hat sie ein Argument:

- `newState` ist der neue Zustand, der gespielt werden soll.

Die Funktion aktualisiert alle benötigten Hilfsvariablen in der Klasse `GameState` und ruft diverse Funktionen auf, um das Spiel für den neuen Zug vorzubereiten.

Sind die Regeln für ein Untenschieden `limitStatesCounter` und `limitMovesWithoutMill` aktiviert (also nicht `None`), werden die entsprechenden Zähler, um die Regeln zu kontrollieren, aktualisiert. Die Regeln werden jedoch nur aktualisiert, wenn sich die Spieler nicht mehr in der Setzphase befinden.

[ ]:
```
def playNewState(self, newState):
    movedStone, poundedStones = getChangedStones(self.state, newState, self.
 ↪player)
    phase = playerPhase(self.state, self.player)
    if phase != 1:
        if self.limitStatesCounter is not None:
            self.statesCounter[newState] += 1
            logger.info(f'the state {newState} was played {self.
 ↪statesCounter[newState]} times.')

        if self.limitMovesWithoutMill is not None:
            if (len(poundedStones) == 0):
                self.movesWithoutMill += 1
            else:
                self.movesWithoutMill = 0
            logger.info(f'moves without mill: {self.movesWithoutMill}')

    self.state = newState
    logger.info(f'New State was played:\n{newState}')
    self.resetStateVariables()
    self.togglePlayer()
    self.checkIfFinished()
```

```
    if phase != 1:
        if  (self.limitStatesCounter is not None) and \
            (self.statesCounter[self.state] + 1 >= self.limitStatesCounter):
                self.hint = f'The state has already been played {self.
↪statesCounter[newState]} times. ' \
                            'If it is played once more, the game will end in a␣
↪remis!'

        if  (self.limitMovesWithoutMill is not None) and \
            (self.movesWithoutMill + 5 >= self.limitMovesWithoutMill):
                self.hint = f'No mill was pound in the last {self.
↪movesWithoutMill} moves. ' \
                            f'In {self.limitMovesWithoutMill - self.
↪movesWithoutMill} moves the game will end in a remis!'

    updateGui(self.canvas[2], self.state, movedStone = movedStone, poundedStones␣
↪= poundedStones)
    self.updateText()

GameState.playNewState = playNewState
del playNewState
```

Die Funktion `checkForComputerStep` überprüft, ob der nächste Zug von einem Algorithmus gespielt werden soll. Falls dies der Fall ist, wird der jeweilige Algorithmus ausgeführt und der Spielzustand aktualisiert.

[ ]:
```
from time import sleep

def checkForComputerStep(self):
    if self.winner is not None:
        logger.warning('Game has ended!')
        return

    if self.pause:
        logger.warning('Pause!')
        self.hint = 'The game has paused. Please click to continue!'
        self.updateText()
        return

    if (self.player == PLAYER_1 and self.algorithm1) or (self.player == PLAYER_2␣
↪and self.algorithm2):
        algorithmName = self.algorithm1.__class__.__name__ \
                        if self.player == PLAYER_1 \
                        else self.algorithm2.__class__.__name__
        logger.info(f'Computer calculating for player {self.player} with␣
↪algorithm {algorithmName}')
```

```
        if self.player == PLAYER_1:
            moves = self.algorithm1.bestMoves(self.state, self.player)
        else:
            moves = self.algorithm2.bestMoves(self.state, self.player)

        nextState = moves.choice()
        self.information = { 'score': moves.value }
        if moves.debugInformation:
            self.information.update(moves.debugInformation)

        logger.info(f'Algorithm {algorithmName} calulated best state with score␣
↪{moves.value}')
        logger.info(f'Debug Information:\n{moves.debugInformation}')
        self.playNewState(nextState)

        if self.timeout:
            self.hint = f'Timeout ({self.timeout} seconds). Please wait!'
            self.updateText()
            sleep(self.timeout)

        self.pause = self.stepwise
        self.checkForComputerStep()

GameState.checkForComputerStep = checkForComputerStep
del checkForComputerStep
```

Die Funktion `placeStone` dient zum Platzieren eines Spielersteins in der Spielphase 1. Die Funktion hat ein Argument:

- `coord` ist die Koordinate aus dem Tupel `coords` an dem der Stein des Spielers gesetzt werden soll.

```
[ ]: def placeStone(self, coord):
    if getPlayerAt(self.state[1], coord) != NO_PLAYER:
        logger.warning(f'{coord} is not free')
        self.hint = f'The slot at {coord} is not free!'
        return

    newState = (removeFromStash(self.state[0], self.player), place(self.
↪state[1], coord, self.player))

    if self.validateNewState(newState):
        logging.info('stone placed')
    else:
        logger.info('NewState not in allAvailableStates, checking for new Mills .
↪..')
        self.checkForNewMills(newState)
```

```
GameState.placeStone = placeStone
del placeStone
```

Die Funktion `selectStone` dient zum Selektieren des Steines, der in der Phase 2 verschoben bzw. in Phase 3 springen soll. Die Funktion hat ein Argument:

- `stone` ist die Koordinate des Steines, der bewegt werden soll.

Die Funktion erzeugt bei erfolgreicher Validierung einen Hilfszustand in der GUI mit dem markierten Stein.

[ ]:
```
def selectStone(self, stone):
    if getPlayerAt(self.state[1], stone) != self.player:
        logger.warning(f'{stone} is not the own stone')
        self.hint = 'Please select your own stone!'
        return
    self.selectedStone = stone
    self.hint = None
    updateGui(self.canvas[2], self.state, selectedStone = self.selectedStone)


GameState.selectStone = selectStone
del selectStone
```

Die Funktion `moveStone` dient zum Bewegen des selektierten Steins in der Zug- und Endphase. Die Funktion hat ein Argument:

- `coord` ist die Koordinate, wohin der Stein bewegt werden soll.

Der zu bewegende Stein wurde in dem vorherigen Hilfszug in der Funktion `selectStone` ausgewählt und in der Hilfsvariablen `selectedStone` gespeichert.

[ ]:
```
def moveStone(self, coord):
    if getPlayerAt(self.state[1], coord) != NO_PLAYER:
        logger.warning(f'{coord} is not free')
        self.hint = f'The slot at {coord} is not free!'
        return

    canJump = isAllowedToJump(self.state, self.player)

    if canJump or coord in findNeighboringEmptyCells(self.state[1], self.
 ↪selectedStone):
        newState = (self.state[0], place(self.state[1], self.selectedStone,␣
 ↪NO_PLAYER))
        newState = (newState[0], place(newState[1], coord, self.player))

        if self.validateNewState(newState):
            movement = 'jumped' if canJump else 'moved'
```

```
            logger.info(f'Stone successfully {movement}!')
        else:
            logger.info('Round not finished, checking for new mills...')
            self.checkForNewMills(newState)
    else:
        logger.warning(f'{coord} is not a (free) neighbor of {self.
↪selectedStone}!')
        self.hint = f'The slot at {coord} is not a (free) neighbor of {self.
↪selectedStone}!'

GameState.moveStone = moveStone
del moveStone
```

Die Funktion `checkForNewMills` überprüft, ob ein gegebener Zustand neue Mühlen enthält. Die Funktion hat ein Argument:

- `newState` ist der neue Zustand, der überprüft werden soll.

Falls neue Mühlen gefunden worden sind, wird ein temporärer Zustand erstellt, der den menschlichen Spieler auffordert, einen gegnerischen Stein von dem Spiellbrett zu entfernen. Im Englischen wird dies als *pounding* bezeichnet.

[ ]:
```
def checkForNewMills(self, newState):
    oldMills = findMills(self.state[1], self.player)
    newMills = countNewMills(newState[1], oldMills, self.player)

    if newMills > 0:
        self.stateTemp = newState
        self.millsToPound = newMills
        self.hint = None
        movedStone, poundedStones = getChangedStones(self.state, self.stateTemp,␣
↪self.player)
        updateGui(self.canvas[2], self.stateTemp, movedStone = movedStone,␣
↪poundedStones = poundedStones)

GameState.checkForNewMills = checkForNewMills
del checkForNewMills
```

Die Funktion `poundMillInGui` entfernt einen gegnerischen Spielerstein und beendet somit einen Mühlenzug. Die Funktion hat dabei ein Argument:

- `stone` ist die Koordinate des gegnerischen Spielersteins, der entfernt werden soll.

Ob ein Spielerstein entfernt werden kann, wird mit der Funktion `validateNewState` validiert.

In der Setzphase kann es vorkommen, dass ein Spieler zwei Mühlen schlagen kann. In diesem Fall kann das nicht von der Funktion `validateNewState` ausführt werden, weil der Zug noch nicht abgeschlossen ist und somit nicht in der Menge `nextStates` auftritt. In diesem Fall muss die Validierung von der Funktion selber durchgeführt werden.

```
[ ]: def poundMillInGui(self, stone):
    if self.millsToPound  <= 0:
        logger.warning('Player has no Mills to pound!')
        return
    if getPlayerAt(self.state[1], stone) != opponent(self.player):
        logger.warning(f'{stone} is not the opponent!')
        self.hint = 'Please select an opponent stone!'

        return
    # if the player has only one mill left to pound, it uses the place function␣
 ↪and afterwards
    # validates the newState
    if self.millsToPound == 1:
        newState = (self.stateTemp[0], place(self.stateTemp[1], stone,␣
 ↪NO_PLAYER))
        if self.validateNewState(newState):
            logger.info('success')
        else:
            logger.warning('Mills could not be pounded! The new state could not␣
 ↪be validated by the game logic.')
            self.hint = 'Please do not select an opponent stone that is in a␣
 ↪mill!'

        return

    # otherwise the gui has to validate the mill manually,
    # as the intermediate step cannot be checked by the game logic
    if stone in getCellsPoundable(self.stateTemp[1], self.player):
        self.stateTemp = (self.stateTemp[0], place(self.stateTemp[1], stone,␣
 ↪NO_PLAYER))
        self.millsToPound -= 1
        self.hint = None
        movedStone, poundedStones = getChangedStones(self.state, self.stateTemp,␣
 ↪self.player)
        updateGui(self.canvas[2], self.stateTemp, movedStone = movedStone,␣
 ↪poundedStones = poundedStones)
    else:
        logger.warning('Mills could not be pounded! The new state could not be␣
 ↪validated by the gui.')
        self.hint = 'Please do not select an opponent stone that is in a mill!'

GameState.poundMillInGui = poundMillInGui
del poundMillInGui
```

Die Funktion `validateNewState` überprüft, ob ein gegebener Zustand in der Menge der `nextStates` vorhanden ist. Die Funktion hat ein Argument:

- `newState` ist der neue Zustand, der validert werden soll.

Falls sich der neue Zustand `newState` in der Menge `nextStates` von dem aktuellen Zustand `state` befindet, ist ein Zug von dem Spieler abgeschlossen. Die Hilfsvariablen werden zurückgesetzt und es wird der Spieler getauscht.

[ ]:
```
def validateNewState(self, newState):
    allAvailableStates = nextStates(self.state, self.player)
    if newState in allAvailableStates:
        self.playNewState(newState)
        return True
    return False

GameState.validateNewState = validateNewState
del validateNewState
```

Die Funktion `cancelStep` dient zum Deselektieren eines Steines in der Zug- oder Endphase.

[ ]:
```
def cancelStep(self):
    logger.warn('step is canceled.')

    self.resetStateVariables()
    updateGui(self.canvas[2], self.state)

GameState.cancelStep = cancelStep
del cancelStep
```

Die Funktion `updateText` dient zum Aktualisieren des Textes und des Hinweises auf dem Spielbrett. Die Funktion ermittelt dabei selbständig den aktuellen Zustand des Spieles anhand der Variablen innerhalb der Klasse.

[ ]:
```
def updateText(self):
    phase = playerPhase(self.state, self.player)
    if self.winner is None:
        message = f'Player {self.player}: '

        if self.player == PLAYER_1 and self.algorithm1:
            message += f'Computer\'s turn with {self.algorithm1.__class__.
 ↪__name__}. Please wait.'
        elif self.player == PLAYER_2 and self.algorithm2:
            message += f'Computer\'s turn with {self.algorithm2.__class__.
 ↪__name__}. Please wait.'
        elif self.millsToPound == 1:
            message += 'Pound your mill.'
        elif self.millsToPound > 1:
            message += f'You have {self.millsToPound} mills left to pound.␣
 ↪Please pound your next mill.'
        elif self.selectedStone is not None:
```

```
            movement = 'Move' if phase == 2 else 'Jump'
            message += f'{movement} your selected stone.'
        elif phase == 1:
            message += 'Place your stone.'
        elif phase == 2 or phase == 3:
            movement = 'move' if phase == 2 else 'jump'
            message += f'select your stone you want to {movement}.'
    else:
        message = 'The game has ended: '
        self.hint = None
        if self.winner == NO_PLAYER:
            message += 'Tie.'
        else:
            message += f'{self.winner} has won!'
    logger.info(message)
    drawText(self.canvas[1], message, hint = self.hint, information = self.
 ↪information)

GameState.updateText = updateText
del updateText
```

Die Funktion `checkIfFinished` überprüft, ob ein Spiel beendet worden ist. Dabei benutzt es die Funktionen `finished` und `utility` aus dem Jupyter-Notebookt `nmm-game`.

[ ]:
```
def checkIfFinished(self):
    if (self.limitMovesWithoutMill is not None) and (self.movesWithoutMill >=␣
 ↪self.limitMovesWithoutMill):
        self.winner = NO_PLAYER
    elif (self.limitStatesCounter is not None) and (self.statesCounter[self.
 ↪state] >= self.limitStatesCounter):
        self.winner = NO_PLAYER
    elif finished(self.state, self.player):
        status = utility(self.state, self.player)
        if status == 0:
            self.winner = NO_PLAYER
        else:
            self.winner = self.player if status == 1 else opponent(self.player)

GameState.checkIfFinished = checkIfFinished
del checkIfFinished
```

## 9.2 Spielen

Um das Spiel in der GUI zu spielen, muss zuerst ein Objekt der Klasse `GameState` initialisiert werden. Die Argumente für die Klasse bestimmen die Spieloptionen. Im Folgenden seien vier Beispiele für die Ausführungsoptionen gegeben.

### 9.2.1 Beispiel 1

Der weiße Spieler wird von einem Menschen gespielt und der schwarze Spieler von $\alpha$-$\beta$-Pruning mit der Standardkonfiguration.

[ ]:
```
# gameState = GameState(algorithm2 = AlphaBetaPruning())

# gameState.canvas
```

### 9.2.2 Beispiel 2

Der weiße Spieler wird von einem Menschen gespielt und der schwarze Spieler von $\alpha$-$\beta$-Pruning mit maximal 1000 Zuständen pro Zug.

[ ]:
```
# gameState = GameState(algorithm2 = AlphaBetaPruning(max_states=1000))

# gameState.canvas
```

### 9.2.3 Beispiel 3

In diesem Spiel spielt $\alpha$-$\beta$-Pruning für weiß gegen Minimax für schwarz. Beide KI's spielen mit der Standardkonfiguration. Nach jedem Zug gibt es eine automatische Pause von 5 Sekunden.

[ ]:
```
# gameState = GameState(algorithm1 = AlphaBetaPruning(), algorithm2 = Minimax(),␣
 ↪timeout=5)

# gameState.canvas
```

### 9.2.4 Beispiel 4

In diesem Spiel spielt $\alpha$-$\beta$-Pruning für weiß gegen Minimax für schwarz. $\alpha$-$\beta$-Pruning spielt mit einer geänderten Konfiguration, die auch eine andere Gewichtung der Heuristik verwendet. Das Spiel wird im Einzelschrittmodus gespielt.

[ ]:
```
# customWeights = HeuristicWeights(stones = 2, stash = 2, mills = 2,␣
 ↪possible_mills = 2)

# gameState = GameState(
#                 algorithm1 = AlphaBetaPruning(max_states = 5_000, weights =␣
 ↪customWeights),
#                 algorithm2 = AlphaBetaPruning(),
#                 stepwise = True)

# gameState.canvas
```

# 10 Tunier

Damit unterschiedliche Algorithmen und deren Einstellungen vergleichen werden können, wird eine virtuelles Tunier implementiert. Bei diesem Tunier nehmen die Algorithmen in verschiedenen Ausführungen teil und spielen mehrmals gegeneinander, damit sicher gestellt wird, dass es sich nicht um Zufall handelt.

Innerhalb eines Tuniers (eng. `Tournament`) spielen alle Teilnehmer gegen jeden anderen Teilnehmer, sodass jeder Teilnehmer einmal als Spieler weiß beginnt. Diese Teilnehmerbegegnungen nennen sich Runden (eng. `Round`) in denen mehrere Spiele (eng. `Match`) gespielt werden.

Am Ende des Tuniers kann dann anhand der Anzahl der Gewinne ausgewertet werden, welcher Algorithmus mit welcher Einstellung am besten abschneidet.

Zunächst werden beide implementierten Algorithmen geladen: `AlphaBetaPruning` und `Minimax`.

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

Um eine übersichtlichere Entwicklung zu ermöglichen, werden Typdefinitionen geladen, welche später im Code verwendet werden.

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

## 10.1 Spiel

Ein Spiel wird durch die Klasse `Match` implementiert, welche ein einziges Spiel zwischen zwei Teilnehmern darstellt. Der Konstruktor der Klasse erwartet zwei verpflichtende Argumente und sechs optionale Argumente:

Verpflichtend:

- `white` ist eine Instanz einer `ArtificialIntelligence`, die den weißen Spieler spielen wird;
- `black` ist eine Instanz einer `ArtificialIntelligence`, die den schwarzen Spieler spielen wird;

Optional:

- `start_state` ist der Startzustand, der verwendet werden soll, standardmäßig `s0`;
- `start_player` ist der Spieler, der das Spiel beginnen soll, standardmäig `w`;
- `max_turns` ist die maximale Anzahl an Zügen, die das Spiel dauern darf, bevor es in einem Remis endet, standardmäßig `250`;
- `max_state_replayed` ist die maximale Anzahl die ein Zug nochmal gespielt werden darf, bevor das Spiel in einem Remis endet, standardmäßig `5`;
- `max_states_without_mill` ist die maximale Anzahl von aufeinanderfolgenden Zügen in denen kein Stein geschlagen wurde, bevor das Spiel in einem Remis endet, standardmäßig `30`;
- `name` ein optionaler Name für das Spiel.

Des Weiteren werden zwei Attribute initialisiert, die für den Spielverlauf nötig sind:

- `log` der Verlauf aller Zustände;
- `no_mill_played` die Anzahl der letzten Züge ohne eine neue Mühle.

[ ]:
```
class Match():
    def __init__(
        self,
        white: ArtificialIntelligence, black: ArtificialIntelligence,
        start_state = s0, start_player = 'w',
        max_turns: int = 250, max_state_replayed: int = 5,␣
 ↪max_states_without_mill: int = 30,
        name: str = ""
    ):
        self.white = white
        self.black = black

        self.state = start_state
        self.player = start_player
        self.max_turns = max_turns
        self.max_state_replayed = max_state_replayed
        self.max_states_without_mill = max_states_without_mill
        self.name = name

        self.log = [start_state]
        self.no_mill_played = 0
```

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

[ ]:
```
def __repr__(self: Match):
    return f"Match(name='{self.name}', white={type(self.white).__name__}, " + \
           f"black={type(self.black).__name__}, max_turns={self.max_turns}, " + \
           f"max_state_replayed={self.max_state_replayed},␣
 ↪max_states_without_mill={max_states_without_mill})"

Match.__repr__ = __repr__
del __repr__
```

Das Ergebnis eines Spiels wird in der `MatchResult` Klasse gespeichert. Durch die Aufteilung in die Klassen `Match` und `MatchResult` kann der Garbage Collector die `Match` Instanz löschen, sobald das Spiel beendet ist und die Variable nicht mehr verwendet wird. So können möglicherweise große Transpositionstabellen gelöscht werden und der RAM wieder frei gegeben werden.

Ein `MatchResult` besteht aus drei Attributen, die im Konstruktor gesetzt werden müssen:

- `winner` $\in$ *Player* $\cup \{' '\}$, der Gewinner (oder Remis) des Spieles;
- `log` ist die chronologische Liste aller gespielen Zustände;
- `reason` eine Zeichenkette die genauer beschreibt warum das Spiel endete.

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

[ ]:
```
class MatchResult():
    def __init__(self, winner: str, log: List, reason: str):
        self.winner = winner
        self.log    = log
        self.reason = reason

    def __repr__(self):
        return f"MatchResult(winner='{self.winner}', log={len(self.log): >3},␣
↪reason='{self.reason}')"
```

Die Hilfsfunktion `current_ai` gibt für das aktuelle Spiel die KI-Instanz zurück, die gerade am Zug ist.

[ ]:
```
def current_ai(self: Match) -> ArtificialIntelligence:
    if self.player == 'w':
        return self.white
    return self.black

Match.current_ai = current_ai
del current_ai
```

Die Hilfsfunktion `check_remis` überprüft, ob der aktuelle Zustand und Spieler zu einem Remis führen. Ist dies der Fall, wird ein entsprechendes `MatchResult` mit Begründung, ansonsten `None`, zurückgegeben. Gründe für ein Remis sind:

- die maximale Anzahl an Zügen wurde gespielt - `max_turns`;
- ein Zug wurde öfter als maximal erlaubt gespielt - `max_state_replayed`;
- die maximale Anzahl an Zügen ohne eine neue Mühle wurde gespielt - `max_states_without_mill`.

[ ]:
```
def check_remis(self: Match) -> Optional[MatchResult]:
    if len(self.log) >= self.max_turns:
        return MatchResult(
            winner = ' ',
            log    = self.log,
            reason = f"Reached max_turns after {self.max_turns} turns"
        )

    if self.log.count(self.state) >= self.max_state_replayed:
        return MatchResult(
            winner = ' ',
            log    = self.log,
            reason = f"State has been replayed for {self.log.count(self.state)}␣
↪turns"
```

```
        )

    if self.no_mill_played >= self.max_states_without_mill:
        return MatchResult(
            winner = ' ',
            log    = self.log,
            reason = f"No mill has been played for {self.no_mill_played} turns"
        )
    return None

Match.check_remis = check_remis
del check_remis
```

Um ein Spiel zu spielen wird die Funktion `player` implementiert, diese bedient sich der im `Match` gespeicherten Einstellungen und Daten.

Prinzipiell unendlich lang sind die Spieler abwechselnd am Zug und spielen ihren am besten berechneten Zug. Ein Spiel endet sobald per `finished` Funktion ein Gewinner ermittelt wurde oder per `check_remis` Hilfsfunktion ein Remis beschlossen wurde. Ist dies nicht der Fall, wird der aktuelle Teilnehmer nach dem nächsten (besten) Zug befragt, dieser wird gespeichert und der Gegner ist am Zug.

Da nach einer undefinierten Anzahl an Zügen entweder `finished` oder `check_remis` ein Ergebnis liefern, wird immer eine `MatchResult` Instanz zurückgegeben.

[ ]:
```
def play(self: Match) -> MatchResult:
    while True:
        remis = self.check_remis()
        if remis is not None:
            return remis

        if finished(self.state, self.player):
            # Remis was already checked
            winner = self.player if utility(self.state, self.player) == 1 else␣
 ↪opponent(self.player)
            return MatchResult(
                winner = winner,
                log    = self.log,
                reason = f"A player won the match"
            )

        mills_before = findMills(self.state[1], self.player)

        bestMoves = self.current_ai().bestMoves(self.state, self.player)
        self.state  = bestMoves.choice()
        self.player = opponent(self.player)

        self.log.append(self.state)
```

```
        if playerPhase(self.state, self.player) != 1 and \
           countNewMills(self.state[1], mills_before, self.player) <= 0:
            self.no_mill_played += 1
        else:
            self.no_mill_played = 0

Match.play = play
del play
```

## 10.2 Runde

Damit zufällige Gewinne möglichst ausgeschlossen werden, können mehrere Spiele zwischen den gleichen Teilnehmern gleichzeitig in einer Runde gespielt werden. Hierzu wird die Python Bibliothek `multiprocessing` verwendet, die für jedes Spiel einen eigenen Prozess startet.

[ ]:
```
from multiprocessing import Pool
```

Die Klasse `Round` spiegelt solch eine Runde wieder und besitzt drei Parameter im Konstruktor:

- `white` ist eine Instanz oder eine Funktion die eine `ArtificialIntelligence` produziert, die den weißen Spieler spielt;
- `black` ist eine Instanz oder eine Funktion die eine `ArtificialIntelligence` produziert, die den schwarzen Spieler spielt;
- `instances` ist die Anzahl der Spiele die gleichzeitig gestartet werden sollen;
- `seed_offset` ist die Zahl, die auf den Seed addiert werden soll.

Die Parameter `white` und `black` können auch Funktionen akzeptieren, damit sichergestellt werden kann, dass die Teilnehmer innerhalb der gestarteten Spiele sich keine Transpositionstabllen teilen und somit einen Vorteil erhalten könnten.

[ ]:
```
class Round():
    def __init__(
        self,
        white: Union[ArtificialIntelligence, Callable],
        black: Union[ArtificialIntelligence, Callable],
        instances: int,
        seed_offset: int
    ):
        self.white = white
        self.black = black
        self.instances = instances
        self.seed_offset = seed_offset
```

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

```
[ ]: def __repr__(self: Round):
         return f"Round(white={type(self.white).__name__}, black={type(self.black).
     ↪__name__}, instances={self.instances})"

     Round.__repr__ = __repr__
     del __repr__
```

Die Hilfsfunktion `execute` wird durch die Python Bibliothek `multiprocessing` innerhalb des neu erzeugten Prozesses ausgeführt. Sie setzt den Seed der Random-Funktion auf den Index der Runde damit verschiedene Spiele stattfinden und startet das Spiel.

```
[ ]: def execute(match):
         seed, rnd = match
         random.seed(seed)
         return rnd.play()
```

Die Funktion `play` erzeugt eine Liste von `instances` Spielen und erstellt gegebenenfalls die Instanzen der KIs aus den gespeicherten Erstellungsfunktionen. Daraufhin werden die Spiele per `multiprocessing` und `execute` Hilfsfunktion in neuen Prozessen gestartet.

Sobald jedes Spiel beendet wurde, werden die Ergebnisse wieder als Liste zurück gegeben.

```
[ ]: def play(self: Round) -> List[MatchResult]:
         matches = [
             Match(
                 self.white() if callable(self.white) else self.white,
                 self.black() if callable(self.black) else self.black,
                 name = f"r={i: <2}"
             )
             for i in range(self.instances)
         ]
         with Pool(self.instances) as pool:
             return pool.map(execute, ((seed+self.seed_offset, match) for seed, match␣
     ↪in enumerate(matches)))

     Round.play = play
     del play
```

### 10.3 Tunier

Damit nun verschiedene Teilnehmer in unterschiedlichen Konstellationen verglichen werden können, wurde ein Tunier implementiert. Die Klasse `Tournament` erwartet einen verflichtenden und zwei optionale Parameter:

Verpflichtend:

- `participants` ist eine Liste an Teilnehmer `ArtificialIntelligence` Instanzen oder Funktionen, die diese erzeugen;

Optional:

- `instances_per_round` ist die Anzahl der Spiele die pro Runde und pro Teilnehmer Konstellation gespielt werden soll;
- `name` ist ein Name, der den Log-Dateien angehängt wird;
- `skip` ist die Anzahl der Runden, die übersprungen werden sollen. Dies ist Hilfreich falls die Berechnungen abbrechen und fortgesetzt werden sollen;
- `seed_offset` ist die Zahl, die auf den Seed einer Runde addiert werden soll. So können Runden in Teilen ausgerechnet werden.

Pro Spiel können durch die Transpositionstabelle bis zu 6 Gigabyte an RAM benötigt werden. Dementsprechend wird empfohlen die `instances_per_round` Einstellung zu bearbeiten.

[ ]:
```python
class Tournament():
    def __init__(
        self,
        participants: List[Union[ArtificialIntelligence, Callable]],
        instances_per_round: int = 4,
        name: str = "unnamed",
        skip: int = 0,
        seed_offset: int = 0
    ):
        self.participants = participants
        self.instances_per_round = instances_per_round
        self.name = name
        self.skip = skip
        self.seed_offset = seed_offset
```

Die Hilfsfunktion `save` speichert die Ergebnisse einer Runde in einer menschenlesbaren Datei, die nach dem Schema `round-NAME-ID.txt` benannt ist. Die Funktion erwartet drei Parameter:

- `idx` ist die Nummer der aktuellen Runde (wird um eins erhöht, um eine natürliche Zählung zu verwenden);
- `rnd` ist die aktuelle Runde;
- `results` ist die Liste der Ergebnisse, die gespeichert werden soll.

[ ]:
```python
def save(self: Tournament, idx: int, rnd: Round, results: List[MatchResult]):
    path = f"round-{self.name}-{idx+1}.txt"
    with open(path, "w") as file:
        file.write(f"Round: {idx+1}\n")

        file.write(f"\nPlayer:\n")
        white = rnd.white() if callable(rnd.white) else rnd.white
        black = rnd.black() if callable(rnd.black) else rnd.black
        file.write(f"  white: {white}\n")
        file.write(f"  black: {black}\n")
```

```
        file.write(f"\nResult:\n")
        file.write(f"  remis: {sum(result.winner==' ' for result in results)}\n")
        file.write(f"  white: {sum(result.winner=='w' for result in results)}\n")
        file.write(f"  black: {sum(result.winner=='b' for result in results)}\n")

        for midx, result in enumerate(results):
            file.write(f"\nMatch {midx+1}:\n")
            file.write(f"  Winner: '{result.winner}'\n")
            file.write(f"  Reason: {result.reason}\n")
            file.write(f"  Log:\n")
            for sidx, state in enumerate(result.log):
                file.write(f"    {sidx+1: >3}. {state}\n")
    return path

Tournament.save = save
del save
```

Die Funktion `play` startet das Tunier und speichert per Hilfsfunktion `save` die Ergebnisse in einer Textdatei ab. Die Laufzeit kann gegebenenfalls mehrere Stunden lang sein. Mit Hilfe der Bibliotheken `time` und `tqdm` wird der Fortschitt angezeigt.

Hierzu werden alle möglichen Konstellationen der Teilnehmer erzeugt, sodass jeder Teilnehmer gegen jeden anderen Teilnehmer, sowohl als *weiß* als auch als *schwarz*, spielt. Spiele gegen sich selbst werden nicht durchgeführt.

[ ]:
```
import time, tqdm
def play(self: Tournament):
    rounds = list(enumerate(
        Round(a, b, self.instances_per_round, self.seed_offset)
        for a in self.participants
        for b in self.participants
        if a != b
    ))

    for idx, rnd in tqdm.tqdm(rounds):
        if idx < self.skip:
            print(f"Round {idx+1: >2}/{len(rounds)} was skipped")
            continue

        start = time.time()
        results = rnd.play()
        end = time.time()
        print(f"Round {idx+1: >2}/{len(rounds)} took {end-start}")
        path = self.save(idx, rnd, results)
        print(f" > Saved to {path}")

Tournament.play = play
del play
```

# 11 Fazit

Das Ziel der Studienarbeit, eine künstliche Intelligenz für das Brettspiel Mühle zu entwickeln, wurde erreicht. Es wurden zwei Algorithmen mit verschiedenen Verbesserungen implementiert, die ein interessantes Spiel gegen Menschen ermöglichen.

## 11.1 Bewertung

Um eine möglichst objektive Bewertung der Algorithmen durchzuführen, sollen in diesem Kapitel die Algorithmen *Minimax* und *α-β-Pruning*, sowie mehrere Heuristiken ausprobiert und miteinander verglichen werden. Die dazu benötigten Implementierungen wurden bereits im Kapitel *Turnier* vorgenommen.

Jede gespielte Runde ist als Textdatei mit genauem Spielverlauf aufgezeichnet und in dem Ordner rounds zu finden.

Das zuvor beschriebene Modul *Turnier* muss geladen werden. Durch das Laden des Moduls werden bereits alle anderen nötigen Module wie *Minimax*, *α-β-Pruning* oder die *Game*-Definition geladen.

[ ]:
```
%run nmm-tournament.ipynb
```

### 11.1.1 Minimax vs. α-β-Pruning

In dem ersten Experiment sollen die Algorithmen *Minimax* und *α-β-Pruning* gegeneinander antreten. Beide verwenden die selbe Heuristik, welche zufällig ausgewählt wurde. Insgesamt werden zwei Runden à vier Spiele gespielt. Jeder Algorithmus beginnt einmal als der *weiße* Spieler.

- *α-β-Pruning* betrachtet 25.000 Zustande pro Zug. Dies dauert in jeder Phase des Spiels ca. 10 Sekunden.
- *Minimax* hingegen durchsucht einen Baum mit der Tiefe drei und schaut somit drei Spielzüge in die Zukunft. Dies dauert zu Beginn des Spiels ca. 35 Sekunden, in der zweiten Phase hingegen ca. 1-2 Sekunden.

Zu erwarten ist, dass *α-β-Pruning* besser als *Minimax* abschneidet, da es die Möglichkeit besitzt, Teilbäume abzuschneiden, welche nicht mehr in Frage kommen würden. Außerdem kann *α-β-Pruning* in der zweiten Phase des Spiels aufgrund der *iterativen Tiefensuche* eine größere Rekursionstiefe erreichen.

[ ]:
```
Tournament(
    [
        lambda: Minimax(
            weights    = HeuristicWeights(stones=1, stash=1, mills=4,␣
↪possible_mills=2),
            limit      = 3
        ),
        lambda: AlphaBetaPruning(
            weights    = HeuristicWeights(stones=1, stash=1, mills=4,␣
↪possible_mills=2),
```

```
            max_states = 25_000
        )
    ],
    instances_per_round = 4,
    name                = "mm-vs-ab"
).play()
```

| | Minimax | | | α-β-Pruning | | |
|---|---|---|---|---|---|---|
| Minimax | | | | 2 | 1 | 1 |
| α-β-Pruning | 1 | 1 | 2 | | | |
| g r v | 2 | 2 | 4 | 4 | 2 | 2 |
| Punkte | | -2 | | | 2 | |

Die vorherige Vermutung lässt sich mit diesem Ergebnis bestätigen: *α-β-Pruning* schneidet besser ab als *Minimax*. Dennoch ist es überraschend, dass zwei der Spiele im Remis enden und weitere zwei Spiele sogar von Minimax gewonnen werden.

Zu dem Gesamtsieg von *α-β-Pruning* hat einerseits der Algorithmus selbst beigetragen, da dadurch weite Teile des Suchbaumes übersprungen werden konnten. Die Verwendung der *iterativen Tiefensuche* hat mit einer festen Suchgröße dazu beigetragen, dass bei gleichbleibender Rechenzeit dynamisch die richtige Rekursionstiefe gewählt werden konnte. Die Erkennung von symmetrischen Zuständen hilft besonders in den ersten Zügen eine große Rekursionstiefe zu erreichen, da hier viele Symmetrien auftauchen. Auch in der letzten fliegenden Phase können ein paar Rechnungen damit eingespaart werden.

### 11.1.2 Heuristik vs. Heuristik (α-β-Pruning)

In dem zweiten Experiment soll unter Verwendung des *α-β*-Pruning Algorithmus herausgefunden werden, welche Heuristik am besten geeignet ist. Die Anzahl der möglichen Heuristiken ist unendlich groß, da jeder der vier Parameter eine reelle Zahl ist. Aus diesem Grund soll nur ein kleines Experiment durchgeführt werden. Es treten sechs Konfigurationen in 3 Spielen pro Runde an. Dadurch ergibt sich eine Rundenanzahl von 30.

Bei diesem Experiment soll die Wichtigkeit der Parameter festgestellt werden, indem alle möglichen Heuristiken mit den Permutationen der Zahlen $1, 2, 3$ gegeneinander antreten. Die Parameter `stones` und `stash` erhalten jedoch immer den gleichen Wert, da sich während der Entwicklung gezeigt hat, dass die Algorithmen besonders in der ersten Phase falsche Entscheidungen treffen, sollten sich diese Parameter unterscheiden. Als positiver Nebeneffekt verringert sich die Größe des Experimentes.

```
[ ]: Tournament(
        [
            lambda: AlphaBetaPruning(weights=HeuristicWeights(stones=1, stash=1,␣
     ↪mills=2, possible_mills=3)),
            lambda: AlphaBetaPruning(weights=HeuristicWeights(stones=2, stash=2,␣
     ↪mills=1, possible_mills=3)),
            lambda: AlphaBetaPruning(weights=HeuristicWeights(stones=3, stash=3,␣
     ↪mills=2, possible_mills=1)),
            lambda: AlphaBetaPruning(weights=HeuristicWeights(stones=1, stash=1,␣
     ↪mills=3, possible_mills=2)),
            lambda: AlphaBetaPruning(weights=HeuristicWeights(stones=2, stash=2,␣
     ↪mills=3, possible_mills=1)),
            lambda: AlphaBetaPruning(weights=HeuristicWeights(stones=3, stash=3,␣
     ↪mills=1, possible_mills=2)),
        ],
        instances_per_round = 3,
        name                = "hr-vs-hr",
    ).play()
```

Die Rechenzeit für dieses Experiment betrug ca. 12 Stunden. Dabei reichte zwischenzeitlich der Arbeitsspeicher von 32GB nicht aus und das Experiment brach kurzzeitig ab.

Die aggregierten Ergebnisse sind in der folgenden Tabelle dargestellt. Die detaillierten Ergebnisse sind in den Dateien `round-hr-vs-hr-{1-30}.txt` zu finden. Die Spalten stellen die *weißen* Spieler dar und die Zeilen die *schwarzen* Spieler. Jede Heuristik hat einmal als *weiß* und einmal als *schwarz* gegen jede andere Heuristik gespielt. Gegen sich selbst wurde nicht gespielt (siehe Diagonale ohne Daten).

Der Bezeichner für jede Heurisik ist als Tupel aufgeschrieben, die folgender Definition entspricht:

$$\langle stones, stash, mills, possible\_mills \rangle$$

Eine Zelle in der Tabelle stellt eine Runde dar und zählt die Anzahl der Spiele in drei Kategorien:

$$Wei\ gewinnt \mid Remis \mid Schwarz\ gewinnt$$

| | (1,1,2,3) | | | (2,2,1,3) | | | (3,3,2,1) | | | (1,1,3,2) | | | (2,2,3,1) | | | (3,3,1,2) | | | w | r | s |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **(1,1,2,3)** | | | | 1 | 2 | 0 | 2 | 1 | 0 | 2 | 1 | 0 | 3 | 0 | 0 | 2 | 1 | 0 | 10 | 5 | 0 |
| **(2,2,1,3)** | 1 | 2 | 0 | | | | 3 | 0 | 0 | 3 | 0 | 0 | 3 | 0 | 0 | 2 | 1 | 0 | 12 | 3 | 0 |
| **(3,3,2,1)** | 0 | 1 | 2 | 1 | 0 | 2 | | | | 1 | 0 | 2 | 0 | 2 | 1 | 1 | 1 | 1 | 3 | 4 | 8 |
| **(1,1,3,2)** | 1 | 0 | 2 | 0 | 1 | 2 | 1 | 1 | 1 | | | | 1 | 0 | 2 | 0 | 3 | 0 | 3 | 5 | 7 |
| **(2,2,3,1)** | 0 | 1 | 2 | 0 | 1 | 2 | 1 | 1 | 1 | 1 | 0 | 2 | | | | 1 | 1 | 1 | 3 | 4 | 8 |
| **(3,3,1,2)** | 1 | 0 | 2 | 1 | 0 | 2 | 1 | 1 | 1 | 0 | 2 | 1 | 1 | 1 | 1 | | | | 4 | 4 | 7 |
| **w r s** | 3 | 4 | 8 | 3 | 4 | 8 | 8 | 4 | 3 | 7 | 3 | 5 | 8 | 3 | 4 | 6 | 7 | 2 | | | |

Aggregiert ergibt sich aus den Rundenergebnissen folgende Tabelle. Alle Spiele der Heuristiken wurden aufgeschlüsselt in

$$Gewonnen \mid Remis \mid Verloren$$

Für die Punkteberechnung wurde jedes Gewinnen mit $+1$ und jedes Verlieren mit $-1$ bewertet. Ein Remis hat eine Wertung von 0.

| | (1,1,2,3) | | | (2,2,1,3) | | | (3,3,2,1) | | | (1,1,3,2) | | | (2,2,3,1) | | | (3,3,1,2) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **weiß** | 3 | 4 | 8 | 3 | 4 | 8 | 8 | 4 | 3 | 7 | 3 | 5 | 8 | 3 | 4 | 6 | 7 | 2 |
| **schwarz** | 0 | 5 | 10 | 0 | 3 | 12 | 8 | 4 | 3 | 7 | 5 | 3 | 8 | 4 | 3 | 7 | 4 | 4 |
| **g r v** | 3 | 9 | 18 | 3 | 7 | 20 | 16 | 8 | 6 | 14 | 8 | 8 | 16 | 7 | 7 | 13 | 11 | 6 |
| **Punkte** | -15 | | | -17 | | | 10 | | | 6 | | | 9 | | | 7 | | |

Als klare Verlierer sind die Heuristiken zu bewerten, welche den Parameter *possible_mills* größer als die anderen Parameter gewählt haben. Beide haben mehr als die Hälfte der 30 gespielten Spiele verloren und haben somit auch eine sehr negative Punktwertung.

Mit einem Punkt Vorsprung gewinnt die Heuristik $\langle 3, 3, 2, 1 \rangle$ in der Gesamtwertung. Diese gewichtet *stones* und *stash* höher als *mills*, welche wiederum höher als *possible_mills* gewichtet werden. Der zweite Platz $\langle 2, 2, 3, 1 \rangle$ verliert nur mit einem Punkt. Auch im direkten Vergleich gewann der erste Platz nur einmal öfter gegen den zweiten Platz.

Da diese Werte sehr nah aneinander liegen, können diese kleinen Abweichungen auch durch Zufall entstanden sein. Bei gleich bewerteten Zügen wählt der Algorithmus einen Spielzug zufällig aus.

## 11.2 Verbesserungsmöglichkeiten

Die Implementierung benötigt sehr viel Arbeitsspeicher, was jedoch beim normalen Spielen kein Problem darstellen sollte. Erst wenn zu Testzwecken mehrere Spiele gleichzeitig gespielt werden, macht sich der große Arbeitsspeicherverbrauch bemerkbar. Durch die Verwendung einer Bitmaske könnte der Arbeitsspeicher um ein Vielfaches verringert werden. Die Änderung hätte aber eine komplette Neuimplementierung erfordert, welche den Rahmen der Studienarbeit gesprengt hätte.

Die Rechenzeit der Algorithmen ist sehr hoch. Dies hängt stark mit dem Arbeitsspeicherverbrauch zusammen, da hierdurch mehr Daten kopiert werden müssen und Berechnungen auf Grund der größeren Datenmengen länger brauchen. Eine Verringerung der Rechenzeit könnte die Algorithmen aus Sicht des Anwenders besser werden lassen, da in der gleichen Zeit mehr Zustände in einer größeren Rekursionstiefe, und damit weitere Züge in der Zukunft, betrachtet werden.

Das Lösen dieser Probleme würde eine komplette Neuimplementierung der Algorithmen nötig machen. Im gleichen Zuge könnte dann jedoch eine schnellere, kompilierte Sprache für die Implementierung gewählt werden, beispielsweise *C*, *C++* oder *Rust*.

# 12 Literatur


- *An Introduction to Artificial Intelligence*, Prof. Dr. Karl Stroetmann, 2020-2021
- *Depth-First Iterative-Deepening: An Optimal Admissible Tree Search*, Richard E. Korf, 1985
- *Turnierreglement*, WELTMÜHLESPIEL DACHVERBAND, 2019
- Mühlespiel, WELTMÜHLESPIEL DACHVERBAND, 2018