Aufgabe 4: Würfelglück

Team-ID: 00960

Team: Egge Q2

Bearbeiter/-innen dieser Aufgabe:
Benedikt Biedermann, Kevin Bah, Leonie-Sophie Ilyuk, Nils Kettler, Ole Kettler, Dominik Pfeiffer, Shalina Rohdenburg, Jule Schlifke, Rebekka Schmidt

16. November 2021

Inhaltsverzeichnis

Lösungsidee

Um den Freunden zu helfen, die besten Würfel zu finden, muss „Mensch ärgere dich nicht” so simuliert werden, dass beliebig oft wiederholbar zwei Computerspieler mit ihren Würfeln gegeneinander antreten können.

Für die Simulation muss ein Model des Spielbretts erstellt werden (Start-, Lauf- und Zielfelder) auf dem dann zwei Spieler abwechselnd (unter Beachtung aller Spielregeln) ihre jeweils vier Spielfiguren bewegen. Strategisches Ziehen der Figuren findet nahezu nicht statt, da mit der vorgegebenen Regeländerung stets die vorderste Figur auf den Lauffeldern gezogen werden muss (lediglich innerhalb der Zielfelder kann strategisch gezogen werden).

Die Simulation, Spieler und Figuren müssen bei jeder Wiederholung wieder korrekt auf den gleichen Anfangszustand zurückgesetzt werden.

Jeder einzelne Spieler soll dann 200-mal gegen jeden anderen antreten. Gezählt werden nur die gewonnenen Spiele. Geteilt durch die Zahl aller jeweils gespielten Spiele ergibt sich daraus eine vergleichbare Gewinn-Rate für jeden Würfeltyp.

Umsetzung

Objektorientierte Programmierung

Spieler und Figuren müssen jeweils eindeutig identifizierbar sein. Wir haben uns deshalb entschieden die Klassen *Player* und *Pawn* anzulegen, da so jede Instanz einmalig und von anderen Instanzen unterscheidbar ist. Die vier Spielfiguren sind über *player.pawn_list* bzw. *pawn.owner* miteinander verbunden und somit abfragbar. Weiterer Vorteil der objektorientierten Programmierung ist, dass eine beliebige Anzahl Kopien eines Objekts hergestellt werden können. Das nutzen wir für die Klasse *Game*, die mit ihren vielen Methoden die Spielregeln abbildet und als Kern der Simulation beliebig oft instanziert werden kann.

Modellierung des Spielbretts

Wir haben uns bei der Modellierung des Spielbretts dafür entschieden, möglichst nah an der realen Vorlage zu bleiben, um auftretende Fehler besser nachvollziehen zu können. Die Startfelder werden als allgemeines Attribut *game.base* geführt, das die Spielfiguren aller Spieler beinhaltet. Es macht keinen Unterschied, an welcher Position die Spielfiguren stehen – nur, ob sie noch in der Liste *game.base* sind, ist relevant.

Die Lauffelder sind eine Liste von 40 Feldern (entspricht Feld Index 0 – 39), wobei den beiden Spielern dann jeweils unterschiedliche Start- und Endpositionen zugewiesen werden:

```
self.board = [None] * 40
self.base = []
self.p1.startpos = 0
self.p1.endpos = 39
self.p1.goal = [None] * 4
self.p2.startpos = 20
self.p2.endpos = 19
self.p2.goal = [None] * 4
```

In diesem Auszug aus der *__init__()* Methode der Klasse *Game* werden diese Start- und Endpositionen ersichtlich.

Zusätzlich sieht man auch, dass die Zielfelder (*game.player.goal*) den Spielerinstanzen zugeordnet werden. Im Gegensatz zu den Startfeldern (*game.base*) ist hier die Reihenfolge der vier Felder wichtig.

Die Funktion *print_board()* gibt eine grafische Darstellung des gesamten Spielbretts aus.

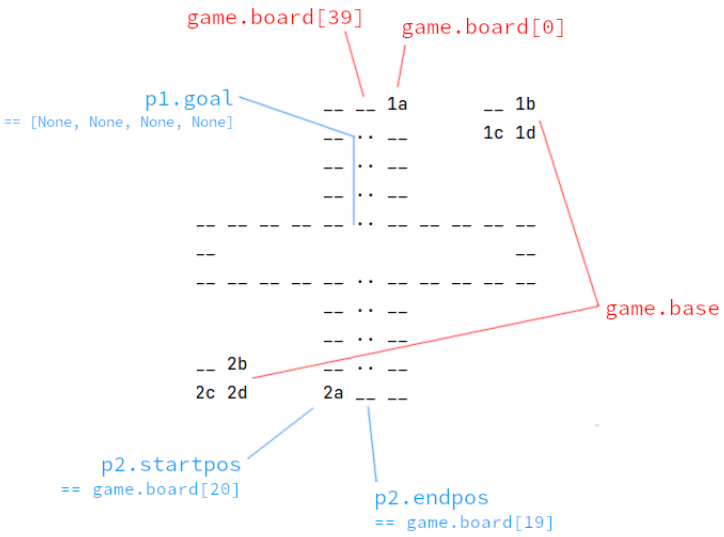

Figure 1: print_board() bei Spielstart

Ein Problem bei dieser Modellierung ist, dass *Player 2* (*p2*) die Grenzen des Spielbretts überqueren muss, um zum Ziel zu gelangen. Dies kann mit einem Test in der Methode *move_pawn_on_board()* überprüft werden. Die Spieligur wird dann entsprechend am Anfang des Bretts wieder eingesetzt:

```
idx = self.board.index(pawn)
new_idx = idx + roll
# Hat das Ende des modellierten Spielfelds erreicht
if self.is_end_of_board(new_idx):
    new_idx = new_idx - len(self.board) - 1
```

Um dennoch stets mit dem „vordersten Stein" des jeweiligen Spielers zu ziehen, hat jedes *Pawn*-Objekt das Attribut *pawn.moves_to_goal,* das beim Einsetzen mit 39 initialisiert wird, bei jedem Zug um die Augenzahl des Wurfes reduziert wird und auf dem letzten Feld vor den Zielfeldern entsprechend 0 erreicht.

Sonderfall „Lauffeld ist blockiert": Lösung mit Rekursion

Ein Sonderfall, der gelöst werden muss, ist die Blockade von Lauffeldern entweder beim Einsetzen eines neuen Spielsteins (blockierte Startposition) oder beim einfachen Ziehen. Im einfacheren Fall handelt es sich um eine gegnerische Spielfigur: Diese wird vom Feld entfernt (auf die *game.base* Felder zurückgesetzt) und wir ziehen unsere Figur.

Wenn es sich um unsere eigene Figur handelt, die blockierende Instanz von *Pawn* also die Kondition *pawn.owner is player* erfüllt, bewegen wir mit unserem Wurf stattdessen erst diese blockierende Spielfigur:

```
def clear_position(self, player, roll, position):
    pawn = self.board[position]
    if pawn.owner is player:
        self.move_pawn_on_board(player, roll, pawn=pawn)
    else:
        self.throw_out_pawn(pawn)
        if position == player.startpos:
            self.activate_pawn(player)
        else:
            self.move_pawn_on_board(player, roll)
```

Die Methode *move_pawn_on_board()* beinhaltet selbst wieder zunächst die Abfrage *is_position_blocked()* und führt dann gegebenenfalls erneut *clear_position()* aus. Dieses rekursive Aufrufen der Funktionen kann problematisch werden, wenn Spielfiguren sich gegenseitig blockieren, z.B. bei Würfeln die wenige Zahlen im Bereich [1, 2, 3] enthalten und entsprechend die Spielfiguren auf den Zielfeldern nicht gut verschieben können. Für diese Fälle zählen wir Rekursion mit dem Attribut *recursion_terminator* und brechen nach fünf Wiederholungen ab.

Beispiele

Wie oben beschrieben tritt in jedem Durchlauf jeder Würfel in jeweils 200 Partien gegen jeden anderen Würfel an. Die folgenden Werte variieren bei jedem Durchlauf leicht (im Bereich +/- 10) und sind in ihrer Aussage, welcher Würfel in der jeweiligen Sammlung der Stärkste ist, stabil.

wuerfel0.txt

Der Würfel #2 setzt sich deutlich gegen alle anderen durch: Er kann bei jeder 6 erneut würfeln, bringt so schnell viele Figuren auf die Lauffelder und kann dann dennoch mit 50% Chance eine 1 zu würfeln die Zielfelder schnell füllen.

Überraschend ist auf den ersten Blick Würfel #3, der kein einziges Spiel gewinnen kann. Dies erklärt sich durch einen Defekt des Würfels: Er kann ohne 6 seine Spielfiguren nicht auf die Lauffelder bringen.

ID	Seiten	Gewonnen	Unentschieden	Gespielt	Gewinn-Rate
1	1 2 3 4 5 6	633	0	1000	63 %
2	1 1 1 6 6 6	981	0	1000	98 %
3	1 2 3 4	0	0	1000	0 %
4	0 1 2 3 4 5 6 7 8 9	547	0	1000	55 %
5	1 2 3 4 5 6 7 8 9 10 11 12	523	0	1000	52 %
6	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	316	0	1000	32 %

wuerfel1.txt

Würfel #2 setzt sich jedesmal relativ knapp vor Würfel #3 durch. Interessant ist bei dieser Würfel-Sammlung, dass hier Partien auftauchen, die nicht entschieden werden können. Die Kondition für Unentschieden haben wir bei 500 Runden ohne Entscheidung festgelegt.

Der Grund dafür findet sich jeweils in der kleinsten Zahl der Würfel: Wenn die kleinste mögliche Zahl 3 ist (Würfel #3), gibt es Situationen in denen nur ein Zielfeld frei ist und es für die letzte Spielfigur (die nicht rückwärts gehen kann) unmöglich wird, auf das Zielffeld vorzurücken. Bei den Würfeln #4 bis #6 wird es entsprechend zunnehmend wahrscheinlich, dass eine solche Situation eintritt, zumal diese Würfel die Spielfiguren auf den Zielfeldern gar nicht mehr bewegen können.

ID	Seiten	Gewonnen	Unentschieden	Gespielt	Gewinn-Rate
1	1 2 3 4 5 6	513	0	1000	51 %
2	2 3 4 5 6 7	705	0	1000	70 %
3	3 4 5 6 7 8	664	42	1000	66 %
4	4 5 6 7 8 9	435	172	1000	43 %
5	5 6 7 8 9 10	216	211	1000	22 %
6	6 7 8 9 10 11	141	227	1000	14 %

wuerfel2.txt

Würfel Variante #5 hat eine erstaunlich hohe Gewinnrate, besonders weil die Konkurrenz in dieser Gruppe ebenfalls recht stark ist. Die Erklärung liegt offensichtlich in der hohen Wahrscheinlichkeit eine 6 zu würfeln begründet und der Spielregel, dass man bei einer 6 abermals würfeln und ziehen darf. Je mehr 6en ein Würfel hat, desto erfolgreicher ist er.

ID	Seiten	Gewonnen	Unentschieden	Gespielt	Gewinn-Rate
1	1 1 1 1 1 6	2	0	800	0 %
2	1 1 1 1 6 6	201	0	800	25 %
3	1 1 1 6 6 6	404	0	800	51 %
4	1 1 6 6 6 6	620	0	800	78 %
5	1 6 6 6 6 6	773	0	800	97 %

wuerfel3.txt

In dieser Sammlung setzt sich der pyramidenförmige Würfel #1 mit vier Seiten knapp durch, offensichtlich, weil hier die Wahrscheinlichkeit, eine 6 zu würfeln bei 25% liegt, im Gegensatz zu Würfel #2 (17%) und Würfel #3 (13%). Würfel #3 scheint hier zu den Würfeln #4 bis #6 einen guten Kompromis aus hohen Zahlen und der Chance, eine 6 zu Würfeln darzustellen.

Würfel #1 bietet zudem einen guten Ausgleich zwischen eher hohen Zahlen (gut für die Lauffelder), sehr niedrigen Zahlen (gut für die Zielfelder) und mit einer 6 der Möglichkeit Figuren auf das Lauffeld zu bringen und nochmals zu ziehen.

ID	Seiten	Gewonnen	Unentschieden	Gespielt	Gewinn-Rate
1	1 2 5 6	762	0	1000	76 %
2	1 2 3 4 5 6	610	0	1000	61 %
3	1 2 3 4 5 6 7 8	636	0	1000	64 %
4	0 1 2 3 4 5 6 7 8 9	438	0	1000	44 %
5	1 2 3 4 5 6 7 8 9 10 11 12	408	0	1000	41 %
6	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	146	0	1000	15 %

wuerfel-gewinner.txt

Abschließend haben wir die jeweiligen Gewinnerwürfel gegeneinander antreten lassen. Würfel #4 gewinnt mit einer absoluten Gewinn-Rate von 100%, weil hier die Konkurrenz verglichen mit der 1. Würfelsammlung schwächer ist. Der zweidimensionale Würfel #2 wurde zur Kontrolle eingeführt und liefert erwartungsgemäß nahezu das selbe Ergebnis wie Würfel #1.

ID	Seiten	Gewonnen	Unentschieden	Gespielt	Gewinn-Rate
1	1 1 1 6 6 6	479	0	800	60 %
2	1 6	466	0	800	58 %
3	2 3 4 5 6 7	135	0	800	17 %
4	1 6 6 6 6 6	800	0	800	100 %
5	1 2 5 6	120	0	800	15 %

Quellcode

Datei 1: program.py

```
from pathlib import Path
from models import Player, Game, print_board

def read_input(filename='wuerfel0.txt'):
    """ Beispieldatei einlesen
    Die Zeilen in List umwandeln, Zeilenumbrüche mit .strip() entfernen.
    Anschließend die Zahlen von String->Integer konvertieren.
    Default ist das Aufgabenbeispiel wuerfel0.txt.
    """
    file = Path('beispieldaten', filename)
    with open(file, 'r') as file_in:
        dice_total = file_in.readline().strip()
        dice_list = [line.strip().split() for line in file_in.readlines()]
        for num, dice in enumerate(dice_list):
            dice_list[num] = [int(element) for element in dice]

    return dice_list


def setup(dice_list):
    """ Spieler anlegen
    Lege ein Player-Objekt für jeden gegebenen Würfel an.
    """
    players_list = []
    for dice in dice_list:
        faces = dice.pop(0)
        new_player = Player(dice=dice)
        players_list.append(new_player)

    return players_list


def do_simulation(player1, player2):
    """ Führe ein Spiel zwischen zwei Spielern durch
    """
    game = Game(player1, player2)

    while not game.winner and game.round < 500:
        game.play_round()

    if game.round == 500:
        print("Played 500 rounds, stuck here:")
        print_board(game)

    return game.winner


def play_pair_matches(player1, player2, games_per_pair):
    """ Simuliere eine gegebene Anzahl an Spielen
    Startspieler wechseln ab der Hälfte
    """
    switch = int(games_per_pair / 2)

    for count in range(games_per_pair):
        if count <= switch:
            winner = do_simulation(player1, player2)
        else:
            winner = do_simulation(player2, player1)
```

```
        player1.games_played += 1
        player2.games_played += 1
        if winner:
            winner.wins += 1
        else:
            player1.draws += 1
            player2.draws += 1


if __name__ == '__main__':
    players_list = setup(read_input())

    games_played = 0
    games_per_pair = 200
    games_per_player = games_per_pair * (len(players_list) - 1)

    # Jeder Spieler (Würfel) tritt gegen jeden anderen Würfel an
    # 1 vs 2/3/4/5, 2 vs 3/4/5, 3 vs 4/5, 4 vs 5
    for idx, player in enumerate(players_list):
        idx += 1
        while idx in range(len(players_list)):
            play_pair_matches(player, players_list[idx], games_per_pair)
            games_played += games_per_pair
            idx += 1

    print(f'Played {games_played} games.')
    for player in players_list:
        print(f'{player} with dice {player.dice} has won {player.wins}, drew {player.draws} out of
{player.games_played} games. ({(player.wins / player.games_played):.2f} win rate)')
```

Datei 2: models.py

```
import random
import string
from copy import deepcopy
from itertools import chain
from operator import attrgetter


class Player:
    """ Erzeugt für jeden Würfel ein eindeutig identifizierbares Spieler-Objekt
    dem vier Spielfiguren zugeordnet werden.
    """
    counter = 1

    def __init__(self, dice):
        self.id = Player.counter
        self.dice = dice
        self.games_played = 0
        self.wins = 0
        self.pawn_list = []
        for _ in range(4):
            new_pawn = Pawn(owner=self)
            self.pawn_list.append(new_pawn)
        Player.counter += 1

    def roll_dice(self):
        return random.choice(self.dice)

    def __repr__(self):
        return f'\"Player {self.id} (d{len(self.dice)})\"'
```

```
class Pawn:
    """ Erzeugt für jedes Spieler-Objekt vier eindeutig identifizierbare Spielfiguren-Objekte
    (z.B. 1a, 1b, 1c, 1d)
    """
    counter = 1
    previous_owner = None

    def __init__(self, owner):
        self.id = ''
        self.owner = owner
        self.moves_to_goal = 100  # wird durch activate_pawn() auf 39 gesetzt
        self.position = None
        self.set_id()

    def set_id(self):
        if self.owner is Pawn.previous_owner:
            Pawn.counter += 1
        else:
            Pawn.counter = 1
        Pawn.previous_owner = self.owner
        self.id = string.ascii_lowercase[Pawn.counter - 1]

    def __repr__(self):
        return f'{self.owner.id}{self.id}'


class Game:
    """ Erzeugt ein 'Mensch-ärgere-dich-nicht' Spiel für zwei Spieler.
    Das Model des Spielfelds besteht aus Start/B-Feldern (base), den Lauffeldern (board)
    und den Zielfeldern (goal).
    """
    def __init__(self, p1, p2):
        self.p1 = p1
        self.p1.startpos = 0
        self.p1.endpos = 39
        self.p1.goal = [None] * 4
        self.p2 = p2
        self.p2.startpos = 20
        self.p2.endpos = 19
        self.p2.goal = [None] * 4
        self.base = []
        self.board = [None] * 40
        self.round = 1
        self.winner = None
        self.recursion_terminator = dict()

        for pawn in chain(self.p1.pawn_list, self.p2.pawn_list):
            self.base.append(pawn)
        self.activate_pawn(self.p1)
        self.activate_pawn(self.p2)

    def play_round(self):
        for player in [self.p1, self.p2]:
            self.one_turn(player)
            if self.winner:
                return
        self.round += 1

    def one_turn(self, player):
        roll = player.roll_dice()

        while roll == 6:
            if self.has_pawn_in_base(player):
                if self.is_position_blocked(player.startpos):
                    self.clear_position(player, roll, player.startpos)
```

```
                else:
                    self.activate_pawn(player)
            else:
                self.move_pawn_on_board(player, roll)
            roll = player.roll_dice()

        self.move_pawn_on_board(player, roll)

        if None not in player.goal:
            self.winner = player

    def is_position_blocked(self, position):
        if self.board[position]:
            return True
        return False

    def clear_position(self, player, roll, position):
        # Bei manchen Würfeln bleiben die letzten beiden Spielfiguren vor dem vollen Ziel stecken
        if self.recursion_terminator.get(self.round):
            if self.recursion_terminator[self.round] > 5:
                return
            self.recursion_terminator[self.round] += 1
        else:
            self.recursion_terminator[self.round] = 1

        pawn = self.board[position]
        if pawn.owner is player:
            self.move_pawn_on_board(player, roll, pawn=pawn)
        else:
            self.throw_out_pawn(pawn)
            if position == player.startpos:
                self.activate_pawn(player)
            else:
                self.move_pawn_on_board(player, roll)

    def select_pawn_to_move(self, pawn_list):
        return

    def move_pawn_on_board(self, player, roll, pawn=None):
        if pawn in self.base:
            return
        pawns_on_board = [p for p in player.pawn_list if p not in self.base]
        if pawns_on_board:
            if not pawn:
                pawn = min(pawns_on_board, key=attrgetter('moves_to_goal'))
            if pawn in player.goal:
                self.move_pawn_within_goal(player, roll)
                return
        else:
            return
        idx = self.board.index(pawn)
        new_idx = idx + roll
        # Hat das Ende des modellierten Spielfelds erreicht
        if self.is_end_of_board(new_idx):
            new_idx = new_idx - len(self.board)
        # Hat die Zielfelder erreicht
        if roll > pawn.moves_to_goal:  # Feld vor dem Zielfeld == 0
            success = self.move_pawn_into_goal(pawn, roll, player)
            if not success:
                alt_pawn = sorted(player.pawn_list, key=attrgetter('moves_to_goal'))[1]
                if pawn is alt_pawn or alt_pawn in player.goal:
                    return
                self.move_pawn_on_board(player, roll, pawn=alt_pawn)
        else:
            if self.is_position_blocked(new_idx):
                self.clear_position(player, roll, new_idx)
```

```
            else:
                self.board[idx] = None
                self.board[new_idx] = pawn
                pawn.moves_to_goal -= roll

    def throw_out_pawn(self, pawn):
        idx = self.board.index(pawn)
        self.board[idx] = None
        self.base.append(pawn)
        pawn.moves_to_goal = 100

    def has_pawn_in_base(self, player):
        for pawn in player.pawn_list:
            if pawn in self.base:
                return True
        return False

    def activate_pawn(self, player):
        for pawn in player.pawn_list:
            if pawn in self.base:
                self.base.remove(pawn)
                pawn.moves_to_goal = 39
                self.board[player.startpos] = pawn
                return

    def is_end_of_board(self, position):
        if position > (len(self.board) - 1):
            return True
        return False

    def move_pawn_into_goal(self, pawn, roll, player):
        moves_in_goal = roll - pawn.moves_to_goal
        if moves_in_goal > 4:
            return False
        idx_board = self.board.index(pawn)
        idx_goal = moves_in_goal - 1
        if player.goal[idx_goal]:
            success = self.move_pawn_within_goal(player, roll)
            if success:
                return True
            return False
        self.board[idx_board] = None
        pawn.moves_to_goal = 100 + idx_goal
        player.goal[idx_goal] = pawn
        return True

    def move_pawn_within_goal(self, player, roll):
        if roll > 3:
            return False
        for idx, element in enumerate(player.goal):
            if element is not None:
                # element ist Spielfigur
                if idx in [0, 1, 2]:
                    try:
                        if not player.goal[idx+roll]:
                            player.goal[idx] = None
                            player.goal[idx+roll] = element
                            return True
                    except IndexError:
                        pass
                if idx in [1, 2, 3]:
                    try:
                        if not player.goal[idx-roll]:
                            player.goal[idx] = None
                            player.goal[idx-roll] = element
                            return True
```

```
                    except IndexError:
                        pass
        return False


def print_board(game):
    p1 = ['__'] * 4
    p2 = ['__'] * 4
    for idx, p in zip(range(4), game.p1.pawn_list):
        p1[idx] = p if p in game.base else '__'
    for idx, p in zip(range(4), game.p2.pawn_list):
        p2[idx] = p if p in game.base else '__'

    g1 = ['..'] * 4
    g2 = ['..'] * 4
    for idx, p in enumerate(game.p1.goal):
        g1[idx] = p if p is not None else '..'
    for idx, p in enumerate(game.p2.goal):
        g2[idx] = p if p is not None else '..'

    b = deepcopy(game.board)
    for idx, dot in enumerate(b):
        if dot is None:
            b[idx] = '__'
        else:
            b[idx] = dot

    print(f'            {b[38]} {b[39]} {b[0]}       {p1[0]} {p1[1]}')
    print(f'            {b[37]} {g1[0]} {b[1]}       {p1[2]} {p1[3]}')
    print(f'            {b[36]} {g1[1]} {b[2]}')
    print(f'            {b[35]} {g1[2]} {b[3]}')
    print(f'{b[30]} {b[31]} {b[32]} {b[33]} {b[34]} {g1[3]} {b[4]} {b[5]} {b[6]} {b[7]} {b[8]}')
    print(f'{b[29]}                           {b[9]}')
    print(f'{b[28]} {b[27]} {b[26]} {b[25]} {b[24]} {g2[3]} {b[14]} {b[13]} {b[12]} {b[11]}
{b[10]}')
    print(f'            {b[23]} {g2[2]} {b[15]}')
    print(f'            {b[22]} {g2[1]} {b[16]}')
    print(f'{p2[0]} {p2[1]}       {b[21]} {g2[0]} {b[17]}')
    print(f'{p2[2]} {p2[3]}       {b[20]} {b[19]} {b[18]}')
    print()
```