Siemen Zielke

Dokumentation

18. November 2021
Team ID: 00108
Team: hexszeug

Die Dokumentationen für alle drei von mir bearbeiteten Aufgaben befinden sich in diesem Dokument. Ich habe aber in jedem einzelnen Aufgabenordner eine Kopie dieser Datei abgelegt.

Inhalt

1 Aufgabe 1 - Schiebeparkplatz

1.1 Lösungsidee

1.1.1 Datenstruktur

Nur die Autos, die quer vor den Parkplätzen stehen, bezeichne ich als „Autos“, während die anderen Autos in der Parkreihe nur als „Plätze“ bezeichnet werden.
Jeder Platz wird durchnummeriert, angefangen bei 0. Im Array `Autos` wird die Position der Autos gespeichert, wobei mit der Position immer der Platz gemeint ist, vor dem die linke Hälfte des Autos steht. Dieses Array hat also die Anzahl der Autos als Länge. Aus dem Array wird nun ein weiteres Array `Plätze` generiert, welches als Länge die Anzahl der Plätze hat. Dort ist der Index des Autos gespeichert, das sich vor dem jeweiligen Platz befindet, bzw. nichts, wenn sich kein Auto im Weg befindet. Die Arrays enthalten also beide einen Verweis auf das jeweils andere Array, sodass mit einem Platz das davorstehende Auto gefunden werden kann und umgekehrt. Die beiden Arrays würden für das Beispiel auf dem Aufgabenblatt so aussehen:

Autos	[2, 5]
Plätze	[None, None, 0, 0, None, 1, 1]

1.1.2 Lösungsfindung

Natürlich muss, um eine Liste mit allen Plätzen und deren Frei-Mach-Anweisungen zu erstellen, für jeden Platz der schnellste Weg, den Platz frei zu machen, gefunden werden. Es gibt, wenn man unnötiges Zu-Weit-Verschieben der Autos ausnimmt, zwei Möglichkeiten, einen Platz frei zu machen: nach links und nach rechts. Die beiden Möglichkeiten sind schwierig zu vergleichen, deswegen berechne ich immer erst beide und vergleiche sie dann, um die bessere zu finden.
Relativ einfach ist noch, zu berechnen, wie weit das Auto verschoben werden muss.

	wird nach links geschoben	**wird nach rechts geschoben**
linker Teil des Autos im Weg	2 Plätze	1 Platz
rechter Teil des Autos im Weg	1 Platz	2 Plätze

1.1.2.1 Rekursive Funktion

Mithilfe dieser drei Informationen (welches Auto, wie weit verschieben, welche Richtung) wird nun eine Funktion aufgerufen. Diese Funktion gibt erstmal einfach nur die Werte, die ihr übergeben wurden zurück. Stehen jedoch in der Bewegungsrichtung andere Autos im Weg, wird es interessant. Die Funktion führt nun sich selbst aus, nur mit dem Auto, das sich neben dem eigenen Auto befindet. An den Rückgabewert dieser Funktion hängt sie dann noch die eigenen Werte an und gibt diese Liste dann zurück. Dadurch wird gewährleistet, dass egal wie viele Autos sich direkt nebeneinander befinden, immer alle Autos zur Seite „geschoben“ werden. Da sich die Funktion unter einer

bestimmten Bedingung selbst aufruft (nur mit anderen Parametern) handelt es sich bei ihr um eine rekursive Funktion. Noch einmal zusammengefasst:

```
funktion(Auto, Entfernung, Richtung)
       wenn anderes Auto im Weg:
           Verschiebungen = funktion(anderes Auto, Entfernung, Richtung)
       hänge Auto, Entfernung und Richtung an Verschiebung an
       gebe Verschiebung zurück
```

Schema 1.1: Rekursive Funktion

1.1.2.2 Parkplatzende

Es kann natürlich vorkommen, dass die Autos in eine bestimmte Richtung nicht verschoben werden können, da der Parkplatz dort zu Ende ist. Ob dies der Fall ist, wird am Anfang der rekursiven Funktion noch überprüft, indem gezählt wird, wie viele Plätze, auf denen sich keine Autos befinden, in die jeweilige Richtung frei sind. Sind es weniger als das Auto verschoben werden soll, ist das Verschieben in die Richtung unmöglich und die Funktion gibt nichts zurück.

1.1.2.3 Seiten vergleichen

Nun müssen nur noch die beiden ermittelten Lösungen verglichen werden. Dafür wird zunächst überprüft, ob überhaupt zu beiden Seiten eine Lösung gefunden werden konnte, denn sonst ist natürlich die einzige Lösung die bessere. Gibt es zu beiden Seiten eine Lösung, wird zunächst die Anzahl der zu verschiebenden Autos verglichen. Müssen für eine Lösung weniger Autos verschoben werden, ist diese besser. Sind aber gleich viele Autos auf beiden Seiten, die verschoben werden müssen, wird die Summe der Verschiebungs-Entfernung beider Lösungen verglichen. Müssen bei einer Lösung die Autos insgesamt weniger weit verschoben werden, ist sie die bessere. Wenn selbst dort beide Lösungen gleich aufwändig sind, wird einfach die genommen, bei der die Autos nach links verschoben werden.

1.2 Implementation

Ich verwende Python, da diese Sprache gut für Algorithmen geeignet ist und sich schnell Sachen ausprobieren lassen. Das heißt natürlich aufgrund der Tatsache, dass Python eine Skript-Sprache ist, dass der gesamte Code mit dabei ist.

Ich habe weder interne noch externe Bibliotheken verwendet.

Ich habe folgende drei Module implementiert: *__main__.py*, *alphabet.py* und *reader.py*. *alphabet.py* wird nur zur Umwandlung von Zahlen in Buchstaben und umgekehrt verwendet, da sowohl Ein- und Ausgabe Buchstaben erfordern. *reader.py* fragt nach einem Dateipfad und liest, wenn sich an diesem eine Parkplatz-Datei befindet, diese ein und gibt die Breite des Parkplatzes und die Autos zurück. Der eigentliche Algorithmus befindet sich in *__main__.py*.

1.2.1 Erzeugen der Datenstruktur

In der eingelesenen Datei befinden sich die Informationen über die Autos, die die Plätze blockieren im Format vom ersten Array (Autos, siehe Idee). Es muss also zunächst dafür gesorgt werden, dass das zweite Array erzeugt wird.

```
line_spots = ""
line_cars = ""

spots = [None] * size
for i in range(size):
    line_spots += c_(i) + " "
    if i in cars:
        spots[i] = spots[i+1] = cars.index(i)
```

```
            line_cars += c_(spots[i] + size) + "-" + c_(spots[i+1] + size) + " "
        elif i - 1 not in cars:
            line_cars += "  "

    print("\n" + line_spots + "\n" + line_cars + "\n")
```

Codeblock 1.1: __main__.py Erzeugen von spots

(Die Funktion `c_()` *stammt aus* `alphabet.py` *und wandelt eine Zahl in einen Buchstaben um.)*
Der fettgedruckte Teil dient tatsächlich dem Erzeugen des zweiten Arrays `spots`. Alles andere erzeugt eine Ausgabe im Terminal, die den eingelesenen Parkplatz darstellt:

```
A B C D E F G
    H-H   I-I
```

Ausgabe 1.1: Parkplatz Visualisierung Beispiel

Es wird zunächst ein leeres Array der Länge `size` erzeugt, welche der Breite des Parkplatzes entspricht. Dann wird in einer for-Schleife jeder Platz durchgegangen und jeweils überprüft, ob sich der jeweilige Index des Platzes in `cars` befindet, also ob dort ein Auto steht. Trifft dies zu, wird die ID des Autos[1] in die aktuelle und die nächste Position in `spots` geschrieben. Dadurch verweisen sowohl der linke Teil (kleinerer Index) als auch der rechte Teil (größerer Index) des Autos auf die Auto-ID, also das Auto in `cars`.

1.2.2 Hauptschleife

Nun kommt der Hauptteil des Programms. Das Programm soll eine Liste mit allen Lösungen für jeden Platz ausgeben, also braucht man eine Schleife die alle Plätze durchgeht.

```
result = ""

for spot in range(size):
    result += "{}: ".format(c_(spot))
    car = spots[spot]
    if car != None:
        offset = cars[car] != spot #False: left side of car is blocking the
      spot; True: right side of the car is blocking the spot
        solution = [None] * 2
        solution[0] = push_car(car, 1 + (not offset), True) #calculate
      solution to left
        solution[1] = push_car(car, 1 + offset, False) #calculate solution to
      right
        better = None
        if solution[0] == None or solution[1] == None:
            better = solution[0] == None
        elif len(solution[0][0]) == len(solution[1][0]):
            better = sum(solution[0][1]) > sum(solution[1][1])
        else:
            better = len(solution[0][0]) > len(solution[1][0])
        if solution[better] != None:
            for i in range(len(solution[better][0])):
                if i > 0:
                    result += ", "
                result += "{} {} {}".format(c_(solution[better][0][i] +
      size), solution[better][1][i], d_(better))
        else:
            result += "impossible"
    result += "\n"
```

[1] Mit der Auto-ID ist der Index des Autos in cars gemeint, also um das wievielte Auto von links es sich handelt.

```
    print(result)
```

Codeblock 1.2: __main__.py Hauptschleife

In dieser Schleife wird als erstes der Buchstabe des aktuellen Platzes an `result` (zu Anfang leerer String) angehängt und danach überprüft, ob sich überhaupt ein Auto vor dem Platz befindet. Ist dort keins, kann direkt zum nächsten Platz gesprungen werden. Wenn aber ein Auto im Weg ist, wird zunächst berechnet, ob sich nun die rechte oder linke Seite des Autos im Weg befindet und in `offset` gespeichert. Dann wird die Lösung mit der rekursiven Funktion `push_car()` in beide Richtungen berechnet[2] und jeweils in `solution[0]` bzw. `solution[1]` gespeichert.
Dann wird die bessere Lösung ermittelt und als Boolean in `better` gespeichert. Wenn das Verschieben nach Links besser ist, dann ist `better` `False`, nach rechts `True`.
Als erstes wird getestet, ob überhaupt zwei Lösungen gefunden wurden, oder ob mindestens eine Richtung nicht geht. Ist mindestens eine Lösung `None`, wird `better` auf die Rückgabe von `solution[0] == None` gesetzt. Wenn es nach links keine Lösung gibt, wird also zunächst davon ausgegangen, dass es nach rechts eine gibt. Das ist zwar logisch nicht ganz korrekt, es kann ja schließlich auch gar keine Lösung geben, aber dieser Fall kann später einfach herausgefiltert werden und der Code wird so etwas verkürzt.
Mit einem `elif` wird dann überprüft, ob beide Lösungen gleich viele Autos verschieben. Tun sie dass, wird `better` auf `sum(solution[0][1]) > sum(solution[1][1])` gesetzt. `solution[0][1]` ist dabei die Liste mit den Entfernungen, um die die Autos nach links verschoben werden müssen, währen `solution[0][0]` die Liste ist, welche Autos verschoben werden sollen. Wenn beide Summen gleich sind, ergibt der Operator „`>`“ `False`, also wird die Verschiebung nach links ausgewählt.
Sind die Anzahl der zu verschiebenden Autos nicht gleich, wird am Ende mit einem „`else`“ `better` einfach auf `len(solution[0][0]) > len(solution[1][0])` gesetzt. Hier können beide Zahlen nicht gleich sein, also ergibt „`>`“ immer ein eindeutiges Ergebnis.
Jetzt muss die bessere Lösung noch ausgegeben werden. Davor muss aber noch einmal überprüft werden, ob auch wirklich eine Lösung existiert und nicht im ersten Vergleich eine nicht existierende Lösung als die Bessere ausgewählt wurde. Dafür wird einfach der Code zur Ausgabe in `if solution[better] != None` geschrieben.
Die Ausgabe iteriert dann einfach mit einer for-Schleife über den Inhalt von `solution[better]` und fügt den Schritt in Textform[3] an `result` an.
Wenn keine Lösung gefunden wurde, wird stattdessen „impossible“ an `result` angehängt.
Egal ob nun ein Auto weggeschoben wurde oder nicht, wird nun noch eine neue Zeile in result angefangen und die Schleife beginnt wieder von vorne mit dem nächsten Platz.

1.2.3 Funktion push_car()

Der gesamte Algorithmus würde ohne diese eine Funktion nicht funktionieren. Ihr Inhalt wurde im Wesentlichen aus Schema 1.1: Rekursive Funktion übernommen und nur in Python umgesetzt. Sie besteht aus vier Teilen:

1. Überprüfen, ob die Bewegung überhaupt möglich ist.
2. Berechnen, wie weit das Auto verschoben werden kann.
3. Wenn das Auto nicht weit genug verschoben werden kann: Rekursives Aufrufen dieser Funktion für das benachbarte Auto, um für das eigentliche Auto den ausreichenden Platz zu schaffen.

[2] Wie genau die Rückgabe von push_car() aufgebaut ist, wird im zugehörigen Abschnitt beschrieben.
[3] c_() konvertiert eine Zahl in einen Buchstaben und d_() einen Boolean in eine Richtung left / right

4. Anfügen der Bewegung des Autos (welches Auto und wie weit) an die Rückgabelisten[4] (die möglicherweise durch Schritt 3 schon Inhalt haben).

```
def push_car(car, move, dir):
    spot = cars[car]
    if move > [spots[spot:], spots[:spot]][dir].count(None):
        return None
    can = 0
    for i in range(1, move + 1):
        if (dir and spots[spot - i] == None) or (not dir and spots[spot + i
+ 1] == None):
            can += 1
        else:
            break
    push = move - can
    res_cars = []
    res_moves = []
    if push > 0:
        res_cars, res_moves = push_car(car + dir * -2 + 1, push, dir)
    res_cars.append(car)
    res_moves.append(move)
    return res_cars, res_moves
```

Codeblock 1.3: push_car()

1.2.3.1 Schritt 1

Schritt 1 ist mit einer Zeile gelöst, die aber relativ komplex ist:
`if move > [spots[spot:], spots[:spot]][dir].count(None)`
`spots[spot:]` erzeugt eine Unterliste von `spots`, in der alle Elemente nach dem Element `spot` sind, also der zu betrachtende Bereich, wenn man nach rechts verschieben will. `spots[:spot]` ist dementsprechend für die andere Seite zuständig. Diese beiden Unterlisten werden mit `[rechte-Liste, linke-Liste]` in eine zweidimensionale Liste eingefügt. In Index 0 dieser Liste befindet sich also der Bereich, in dem sich bewegt werden darf, wenn man das Auto nach rechts verschieben soll und in 1 wenn es nach links soll. Dadurch kann man mit dem übergebenen Boolean `dir` die Seite auswählen (0 / False für rechts, 1 / True für links).
Mit `.count(None)` wird nun noch gezählt, wie viele freie Plätze es in der ausgewählten Richtung gibt. Wenn es weniger freie Plätze gibt, als sich bewegt werden soll (`move`), dann ist es auch durch Wegschieben anderer Autos unmöglich, genug Platz zu machen. Das Auto kann nicht verschoben werden und die Funktion wird mit `return None` beendet.

1.2.3.2 Schritt 2

Schritt 2 besteht aus einer for-Schleife, die nur ein oder zweimal ausgeführt wird, je nachdem wie weit das Auto verschoben werden soll. Beim ersten Durchlauf wird der Platz direkt neben dem Auto und beim zweiten Durchlauf der Platz dahinter darauf überprüft, ob sich dort ein Auto befindet. Ist dort frei, wird die Variable `can`, die zu Anfang auf `0` gesetzt wurde, um `1` erhöht. Falls dort ein Auto steht wird die Schleife abgebrochen.

1.2.3.3 Schritt 3

In Schritt 3 wird zunächst `push` auf `move - can` gesetzt. Push enthält also, wie weit das nächste Auto, das sich im Weg befindet, verschoben werden muss. Dann werden noch die leeren Arrays `res_cars`

[4] Zwei separate Listen für Autos und Entfernungen, da dann die Auswertung einfacher wird.

und *res_moves* erzeugt. Falls *push* größer als *0* ist, wird nun rekursiv die aktuelle Funktion *push_cars()* aufgerufen und *res_cars* und *res_moves* mit der Rückgabe dieser überschrieben.

1.2.3.4 Schritt 4

Nun werden an die Listen *res_cars* und *res_moves* noch jeweils das Auto bzw. die Entfernung, die als Parameter der Funktion übergeben wurden, angehängt.

1.2.3.5 Rückgabe

Die Rückgabe der Funktion erfolgt als zwei getrennte Listen mit derselben Länge. Das dient alleinig dazu, dass die beiden Lösungen nach links und rechts einfacher verglichen werden können, da einfach die Funktionen *len()* und *sum()* verwendet werden können. (Siehe Hauptschleife)

1.3 Beispiele

Hier sind die Ausgaben die von park_and_push.zip zu den Dateien *parkplatz0-5.txt* ausgegeben werden.

```
C: ~\>python3 park_and_push.py.zip
Enter parking meta file path: parkplatz0.txt

A B C D E F G
    H-H   I-I

A:
B:
C: H 1 right
D: H 1 left
E:
F: H 1 left, I 2 left
G: I 1 left

C: ~\>python3 park_and_push.py.zip
Enter parking meta file path: parkplatz1.txt

A B C D E F G H I J K L M N
  O-O P-P   Q-Q     R-R

A:
B: P 1 right, O 1 right
C: O 1 left
D: P 1 right
E: O 1 left, P 1 left
F:
G: Q 1 right
H: Q 1 left
I:
J:
K: R 1 right
L: R 1 left
M:
N:

C: ~\>python3 park_and_push.py.zip
Enter parking meta file path: parkplatz2.txt

A B C D E F G H I J K L M N
    O-O   P-P Q-Q R-R   S-S
```

```
A:
B:
C: O 1 right
D: O 1 left
E:
F: O 1 left, P 2 left
G: P 1 left
H: R 1 right, Q 1 right
I: P 1 left, Q 1 left
J: R 1 right
K: P 1 left, Q 1 left, R 1 left
L:
M: P 1 left, Q 1 left, R 1 left, S 2 left
N: S 1 left

C: ~\>python3 park_and_push.py.zip
Enter parking meta file path: parkplatz3.txt

A B C D E F G H I J K L M N
  O-O   P-P     Q-Q R-R S-S

A:
B: O 1 right
C: O 1 left
D:
E: P 1 right
F: P 1 left
G:
H:
I: Q 2 left
J: Q 1 left
K: Q 2 left, R 2 left
L: Q 1 left, R 1 left
M: Q 2 left, R 2 left, S 2 left
N: Q 1 left, R 1 left, S 1 left

C: ~\>python3 park_and_push.py.zip
Enter parking meta file path: parkplatz4.txt

A B C D E F G H I J K L M N O P
Q-Q R-R     S-S     T-T   U-U

A: R 1 right, Q 1 right
B: R 2 right, Q 2 right
C: R 1 right
D: R 2 right
E:
F:
G: S 1 right
H: S 1 left
I:
J:
K: T 1 right
L: T 1 left
M:
N: U 1 right
O: U 1 left
```

```
P:


C: ~\>python3 park_and_push.py.zip
Enter parking meta file path: parkplatz5.txt

A B C D E F G H I J K L M N O
    P-P Q-Q     R-R     S-S

A:
B:
C: P 2 left
D: P 1 left
E: Q 1 right
F: Q 2 right
G:
H:
I: R 1 right
J: R 1 left
K:
L:
M: S 1 right
N: S 1 left
O:
```

Ausgabe 1.2: Anwendung auf Beispiele parkplatz0-5.txt

2 Aufgabe 2 – Vollgeladen

2.1 Lösungsidee

2.1.1 Idee 1

Meine erste Idee war, am ersten Tag das beste erreichbare Hotel zu suchen. Dabei muss man aufpassen, dass man noch genug Zeit für den Rest des Wegs hat. Das wird dann für alle fünf Tage wiederholt. Dabei gibt es nur einen kleinen Denkfehler. Es kann nämlich vorkommen, dass man, indem man in der ersten Nacht in einem bspw. 2.5er Hotel schläft, in der nächsten Nacht nur noch Hotels mit Bewertung 1.0 zur Auswahl hat. Wenn man sich stattdessen in der ersten Nacht für ein 2.4er Hotel entschieden hätte, das weiter vom Start entfernt ist, könnte man am zweiten Tag vielleicht den Weg zum nächsten 3.0er Hotel schaffen. Deswegen gibt diese Idee nicht den in der Aufgabe definierten optimalen Weg[5] zurück.

2.1.2 Idee 2

Die zweite Idee fängt damit an, den Bereich zu berechnen, zu dem man vom Start fahren kann. Dann wird für jedes einzelne Hotel in diesem Bereich rekursiv berechnet, welchen Bereich man von diesem Hotel aus am zweiten Tag erreichen kann. Das wird insgesamt fünfmal durchgeführt (einmal pro Tag). Alle Wege, die zum Ziel führen, werden in eine Liste eingefügt. Nun schaut man sich nur die schlechtesten Hotels in jedem Weg an. Der Weg mit dem besten Hotel ist der optimale Weg.

Ich habe diese Idee implementiert. Beim Testen mit den Beispielen hat sich aber herausgestellt, dass dieser Algorithmus sehr langsam ist. Bei den letzten beiden Beispielen musste ich das Programm abbrechen, da nach etwa 15min immer noch kein Ende in Sicht war. Ich entschied mich also, eine

[5] Mit Weg wird eine Liste aus vier Hotels bezeichnet, die in aufsteigender Reihenfolge vom Start zum Ziel nacheinander angefahren werden.

neue Idee zu entwickeln. Trotzdem ist diese Implementation als *hotels_slow.py.zip* in der Abgabe zu finden.

2.1.3 Idee 3

Die dritte und finale Idee beruht auf einer Funktion, die einen beliebigen Weg auf einer Liste von Hotels berechnet. Diese Funktion wird auf die ursprüngliche Liste von Hotels, die am Anfang eingelesen wird, angewendet. Gibt es einen Weg, wird das Hotel mit der schlechtesten Bewertung entfernt. Das wird so lange wiederholt, bis es keinen Weg mehr gibt. Der Weg, der als letztes gefunden werden konnte ist der optimale Weg. Diese dritte Idee ist zwar schneller als die zweite Idee, kann dafür aber auch nur für genau die Bedingung aus der Aufgabe genutzt werden, während Idee 2 auch bspw. den Weg mit dem besten Durchschnitt berechnen könnte.

2.2 Implementation

Ich habe wieder Python verwendet. Das vollständige Skript (mit allen Modulen) befindet sich in *hotels.py.zip*. (Der Python-Interpreter kann die Zip-Datei direkt ausführen.)

Ich habe keine Bibliotheken verwendet.

2.2.1 Variablen und Konstanten

Der Start und das Ziel werden als „Pseudohotels“ an die Liste angefügt, um die Speicherung und Berechnung zu erleichtern. Dabei erhalten beide eine Bewertung, die 1.0 über der maximalen Bewertung in der eingelesenen Liste liegt, um diese unter keinen Umständen aus den verfügbaren Hotels zu entfernen.

Variablen:

`hotel_distances`	Liste aller Entfernungen der Hotels, die noch nicht raus sind
`hotel_distances_backup`	Gesamte Liste aller Entfernungen der Hotels
`hotel_ratings`	Liste aller Hotelbewertungen, die noch nicht raus sind
`hotel_ratings_backup`	Gesamte Liste aller Hotelbewertungen
`hotel_ids`	Liste, in der die Indexe der jeweiligen Hotels in den _backup Listen stehen. Dient nach dem Aussortieren mehrerer Hotels zum Wiederherstellen der eigentlichen Position des Hotels in der Originalliste.

Konstanten:

`DISTANCEPERDAY`	Minutenzahl, die pro Tag zurückgelegt werden kann. (Gesetzt auf 360)
`DAYS`	Tage, die die Fahrt maximal dauern darf. (Gesetzt auf 5)
`BEST`	Erst nach Einlesen der Liste. Enthält die Bewertung für das beste Hotel.
`TOTALDISTANCE`	Erst nach Einlesen der Liste. Entfernung, in der das Ziel liegt.

2.2.2 Die „Wegfindungs-Funktion“

```
def getOneWay():
    way = [0]
    hotelID = 0
    for day in range(DAYS):
        find = hotel_distances[hotelID] + DISTANCEPERDAY
        end = len(hotel_distances) - 1
        while hotelID <= end and hotel_distances[hotelID] <= find:
            hotelID += 1
        hotelID -= 1
        way.append(hotelID)
        if hotelID == end:
```

```
            return way
    return None
```

Codeblock 2.1: getOneWay()

Die Funktion funktioniert sehr einfach. Sie „fährt“ einfach jeden Tag zu dem Hotel, das am weitesten weg, aber noch in den 6 Stunden, die pro Tag zur Verfügung stehen, liegt.

2.2.3 Einlesen

```
BEST, TOTALDISTANCE, hotel_distances, hotel_ratings = reader.readHotels()

hotel_distances.insert(0, 0)
hotel_ratings.insert(0, BEST + 1.0)
hotel_distances.append(TOTALDISTANCE)
hotel_ratings.append(BEST + 1.0)

hotel_distances_backup = hotel_distances[:]
hotel_ratings_backup = hotel_ratings[:]
hotel_ids = list(range(len(hotel_distances)))
```

Codeblock 2.2: Einlesen

Das Einlesen aus der Datei selbst wird durch reader.py realisiert.
Im zweiten Abschnitt werden der Start und das Ziel der Reise als „Pseudohotels“ eingefügt. (Siehe Variablen und Konstanten.)
Im dritten Absatz werden die *backup* Listen und *hotel_ids* erzeugt.

2.2.4 Hauptschleife

```
removed_position = None
removed_distance = None
removed_valuation = None
removed_id = None

while getOneWay() != None:
    removed_position = hotel_ratings.index(min(hotel_ratings))
    removed_distance = hotel_distances.pop(removed_position)
    removed_valuation = hotel_ratings.pop(removed_position)
    removed_id = hotel_ids.pop(removed_position)
```

Codeblock 2.3: Hauptschleife

Zunächst werden die Variablen initialisiert, in denen das zuletzt entfernte Hotel und die Position (Index), an der es entfernt wurde festgehalten werden. Das wird benötigt um nach der Schleife einen Schritt rückgängig zu machen, da sonst kein Weg möglich ist.

Die Schleife wird als while-Schleife solange ausgeführt, bis es keinen Weg mehr gibt. In der Schleife wird zunächst die Position des schlechtesten Hotels ermittelt und diese dann entfernt.

2.2.5 Auswertung

```
if removed_position != None:
    hotel_distances.insert(removed_position, removed_distance)
    hotel_ratings.insert(removed_position, removed_valuation)
    hotel_ids.insert(removed_position, removed_id)

    bestWay = []

    for hotelID in getOneWay():
        bestWay.append(hotel_ids[hotelID])
```

```
        print(printer.getWayDisplay(bestWay, hotel_distances_backup,
           hotel_ratings_backup))
    else:
        print("Its not possible to reach the destination by driving only {}
           minutes on {} days and sleeping at the given
           hotels.".format(DISTANCEPERDAY, DAYS))
```

Codeblock 2.4: Auswertung

Zunächst wird ausgeschlossen, dass selbst im Ausgangszustand kein Weg möglich ist. Ist dort keiner möglich wurde auch kein Weg entfernt, da die Hauptschleife nie ausgeführt wurde. Es wird eine Meldung (im else-Block) ausgegeben.
Gibt es also mindestens einen Weg, wird die letzte Hotelliste, bei der es noch einen Weg gab, wiederhergestellt.
Nun wird mithilfe von *`getOneWay()`* ein Weg berechnet, der automatisch auch optimal ist – wie in der Lösungsidee erläutert. Dieser Weg wird dann mithilfe von *`hotel_ids`* auf die Ausgangsliste von Hotels umgerechnet (damit in der Ausgabe die Indexe stimmen). Dieser Weg wird dann durch printer.py in einen lesbaren Text mit Entfernungsangaben zwischen den Hotels usw. umgewandelt und in der Konsole ausgegeben.

2.3 Beispiele

Hier sind die Ausgaben von *`hotels.py.zip`* auf die Beispiele *`hotels1-5.txt`*. Die Ausgaben sind nach folgendem Schema aufgebaut:
`|`*`Hotel-Index`*`:`*`Entfernung zum Start`*`(`*`Bewertung`*`)| --`*`Entfernung zum nächsten Hotel`*`-- |` …
Dabei ist `|S|` der Start und `|E|` das Ziel (Ende).

```
C: ~\>python3 hotels.py.zip
Please enter hotel meta file path: hotels1.txt
|S| --347.0-- |3:347.0(2.7)| --340.0-- |7:687.0(4.4)| --320.0--
       |8:1007.0(2.8)| --353.0-- |11:1360.0(2.8)| --320.0-- |E|

C: ~\>python3 hotels.py.zip
Please enter hotel meta file path: hotels2.txt
|S| --341.0-- |3:341.0(2.3)| --359.0-- |10:700.0(3.0)| --353.0--
       |15:1053.0(4.8)| --327.0-- |25:1380.0(5.0)| --357.0-- |E|

C: ~\>python3 hotels.py.zip
Please enter hotel meta file path: hotels3.txt
|S| --360.0-- |101:360.0(1.0)| --357.0-- |196:717.0(0.3)| --359.0--
       |298:1076.0(3.8)| --357.0-- |401:1433.0(1.7)| --360.0-- |E|

C: ~\>python3 hotels.py.zip
Please enter hotel meta file path: hotels4.txt
|S| --340.0-- |97:340.0(4.6)| --336.0-- |212:676.0(4.6)| --356.0--
       |332:1032.0(4.9)| --284.0-- |434:1316.0(4.9)| --194.0-- |E|

C: ~\>python3 hotels.py.zip
Please enter hotel meta file path: hotels5.txt
|S| --317.0-- |285:317.0(5.0)| --319.0-- |581:636.0(5.0)| --351.0--
       |913:987.0(5.0)| --299.0-- |1178:1286.0(5.0)| --330.0-- |E|
```

Ausgabe 2.1: Anwendung auf Beispiele hotels1-5.txt

4 Aufgabe 4 - Würfelglück

4.1 Lösungsidee

4.1.1 Würfel

Die Würfel werden eingelesen und in Arrays gespeichert. Der normale Spielwürfel (1 - 6) würde z. B. so aussehen: `[1, 2, 3, 4, 5, 6]`. Beim Würfeln wird einfach eine von einem Pseudozufallsgenerator[6] generiete Zahl als Index auf das Würfel-Array angewendet und so das Würfelergebnis ermittelt.

4.1.2 Figuren und Spielbrett

Die Positionen der Figuren auf dem Spielbrett werden als Zahlen angegeben. Da die Figuren der beiden Spieler so gut wie nicht miteinander interagieren - außer beim Schlagen - speichere ich die Positionen der Figuren aus Sicht des jeweiligen Spielers. Daraus ergibt sich folgende Feldverteilung:

B (Haus)	-1
A (Startfeld)	0
Normale Felder	1 – 39
a – d (Zielfelder)	40 – 43

Die Positionen eines Spielers werden in einem Array gespeichert, dass die Zahlen nach Größe sortiert. Das Array eines Spielers könnte also beispielsweise so aussehen: `[-1, 0, 13, 41]`.

4.1.3 Spielzug

Grundsätzlich ist ein Spielzug ganz einfach: Nehmen wir an, Spieler A ist am Zug. Zunächst wird ein Würfelergebnis wie oben beschrieben ermittelt und in einer Variablen gespeichert. Nun wird das letzte Element[7] im Positions-Array von Spieler A um den Würfelwert erhöht. Fertig; Spieler B ist dran. Natürlich gibt es einige Ausnahmen. Um besser mit diesen umgehen zu können, berechne ich zunächst immer, welche Figur wohin ziehen muss und speichere diese Werte in zwei Variablen. Ich teile also den Zug in zwei Teile auf: Die Berechnung und die Ausführung des Zugs.

4.1.3.1 Berechnung

Zunächst filtere ich die einzige Besonderheit heraus, bei einer Figur nicht um den Würfelwert gezogen werden soll, sondern aus dem Haus (-1) auf das Startfeld (0) gezogen wird: bei einer Sechs. Dies kann aber auch nur geschehen, wenn sich noch mindestens eine Figur im Haus befindet und das Startfeld frei ist. Ist eines davon nicht der Fall, wird normal eine Figur um den Würfelwert gezogen.

Nun wird also normal gezogen. Die Frage ist nur, welche Figur gezogen werden soll. Dazu geht eine Schleife einfach die Figuren von hinten nach vorne durch und überprüft ob sie jeweils ziehen kann. Eine Figur kann nicht ziehen, wenn

a) sich eine andere Figur desselben Spielers auf dem Zielfeld (Position der Figur + Würfelwert) befindet
b) das Zielfeld nicht existiert, da es außerhalb des Spielfelds liegt (Zahlen über 43) oder
c) sich die Figur noch im Haus bzw. auf Feld -1 befindet.

Ist keins dieser drei Dinge der Fall, wird die Figur ausgewählt d.h. die „zu-bewegende-Figur-Variable“ wird mit ihr überschrieben. So wird automatisch die Figur ausgewählt, die sich am weitesten vorne befindet und ziehen kann, da die weiter hinten liegenden überschrieben werden. Das gleiche passiert

[6] Ein Pseudozufallsgenerator ist denke ich für dieses Projekt ausreichend.
[7] Gemäß Regeländerung in den Informationen zur Aufgabe auf der Webseite soll immer die vorderste Figur gezogen werden.

mit dem Zielfeld und der Zielfeld-Variablen.
In den Regeln steht noch, dass eine Figur auf dem Startfeld Platz machen muss, wenn sich noch Figuren im Haus befinden. (Natürlich nur wenn sie kann.) Darum wird in der Schleife überprüft, ob die aktuelle Figur Platz machen muss. Dann wird die Schleife abgebrochen, sodass sichergestellt wird, dass keine andere Figur mehr ausgewählt wird.

4.1.3.2 Ausführung

Es wird zunächst überprüft ob eine Figur ausgewählt ist. Kann keine Figur ziehen ist dies nämlich nicht der Fall.
Falls also eine Figur ausgewählt wurde, muss diese ziehen. Dafür wird einfach die Position der Figur auf das ausgewählte Feld gesetzt.

Wichtig ist jetzt noch das Schlagen. Das kann natürlich nur geschehen, wenn sich der Gegner theoretisch auf dem Zielfeld befinden könnte, es also nicht Feld a-d oder das Haus ist. Kann also potentiell ein Gegner geschlagen werden, wird das Zielfeld in die Perspektive des Gegners umgerechnet[8] und dann mit den Figuren des Gegners abgeglichen. Befindet sich eine Figur des Gegners auf dem Feld, wird ihre Position auf -1 gesetzt (sie kommt ins Haus zurück).

4.1.3.3 Gewinner erkennen

Nun wird noch überprüft, ob der gerade ausgeführte Zug vielleicht zu einem Gewinn geführt hat. Wenn sich im Positions-Array keine Zahl kleiner 40 befindet, hat also der aktuelle Spieler gewonnen und das Spiel wird beendet. (Felder über 40 sind die Zielfelder.)

4.1.3.4 Anderen Spieler auswählen

Wenn das Spiel noch nicht vorbei ist und der Spieler keine Sechs gewürfelt hat (also nochmal dran ist), wird jetzt der andere Spieler ausgewählt und der Zug beginnt von vorne.

4.2 Implementation

Wieder habe ich mich für die Verwendung der Sprache Python entschieden, da diese wie gesagt perfekt dafür geeignet ist, Algorithmen zu schreiben. Der Code ist also auch wieder komplett mit dabei.

4.2.1 Bibliotheken

Ich habe die internen Bibliotheken `Random`, `Sys`, `Os`, `Time` und `Multiprocessing` verwendet. Letztere dient zur Aufteilung der Spiele auf die CPU-Kerne des Computers, um die Rechengeschwindigkeit zu erhöhen. Die Verwendung der anderen Bibliotheken ist denke ich klar.

4.2.2 main.py

Es befinden sich im Programm sowohl eine `__main__.py` Datei als auch eine `main.py`. Dies ist notwendig, da die verschiedenen Prozessor-Kerne sonst nicht nur das Spiel selbst ausführen würden, sondern Instanzen des gesamten Programms. Das liegt daran, dass die einzelnen Threads der Kerne das Programm, dass sie gestartet hat, importieren, damit sie die Funktionen in dem Programm aufrufen können. Da in Python Importieren dasselbe ist wie Ausführen (nur in derselben Interpreter-Instanz), würde das gesamte Programm rekursiv aufgerufen werden, bis man es abbreche.[9]

[8] Dazu mehr in der Implementation.
[9] Es gibt zwar in Python die Klausel `if __name__ == "__main__"`, um zu überprüfen, ob es sich um das vom User ausgeführte Programm handelt, das funktioniert aber in diesem Fall nicht, da `__main__.py` immer und nicht nur wenn es das Hauptprogramm ist `__main__` als Namen hat, aufgrund des Dateinamens.

In main.py befindet sich die Funktion *run()*. In dieser Funktion wird zunächst optional die Anzahl der Simulationen pro Würfelpaar eingelesen (Standard 1 Mio.). Dann wird mithilfe von *reader.py* die Würfelliste vom Nutzer erfragt und eingelesen. Dabei werden die Würfel im zweidimensionalen Array *dices* gespeichert. Es wird nun mit zwei ineinander verschachtelten for-Schleifen jede Möglichkeit zwei Würfel zu kombinieren durchgegangen, wobei doppelte Kombinationen mit der Bedingung *if i < j* herausgefiltert werden. Die Würfelpaare werden dann als Tupel dem vorher erstelltem Array *games* angehängt. Das dient dem Zweck, vor dem Simulieren die Anzahl der unterschiedlichen Würfelpaar-Spiele, die simuliert werden müssen, zu wissen. Nun werden die Spiele von vorne nach hinten durchgegangen, mithilfe von printer.py werden Informationen über das Spiel (vorher und Ergebnis nachher) in der Konsole angezeigt, und dann mit *simulate_games()* simuliert.

```
def run():
    global simulations_per_dice_pair
    if len(argv) > 1:
        try:
            simulations_per_dice_pair = int(argv[1])
            if(simulations_per_dice_pair < 1000):
                simulations_per_dice_pair = 1000
            print("Running with different simulation count:",
  simulations_per_dice_pair)
        except Exception as e:
            pass
    dices = reader.load_dices()
    games = []
    for i in range(len(dices)):
        for j in range(len(dices)):
            if i < j:
                games.append((i, j))
    for i in games:
        a = i[0]
        b = i[1]
        dice_a = dices[a]
        dice_b = dices[b]
        printer.print_simulation_start(dice_a, dice_b, games.index(i) + 1,
  len(games), SIMULATIONS_PER_DICE_PAIR)
        results = simulate_games(a, b, dices)
        wins_a = results.count(0)
        wins_b = results.count(1)
        ties = results.count(-1)
        printer.print_results(dice_a, dice_b, wins_a, wins_b, ties)
```

Codeblock 4.1: run() in main.py

In *simulate_game()* werden hauptsächlich die einzelnen „ausreichenden“ Spiele, deren Anzahl (1 Mio.) in der Konstante *simulations_per_dice_pair* gespeichert ist, auf die unterschiedlichen Prozessor-Kerne des PCs verteilt. Außerdem wird während den Simulationen mithilfe von *pgbar.py* ein Ladebalken in der Konsole angezeigt. Am Ende werden die Ergebnisse aller Spiele in einem großen Array zurückgegeben. Die jeweiligen Kerne starten ein Spiel, indem sie *game_thread()* aufrufen.

In *game_thread()* werden zunächst die Argumente, die sich, bedingt durch Multiprocessing, in einem Tupel befinden, entpackt. Dann wird überprüft, ob beide Würfel überhaupt mindestens eine Sechs haben. Sonst wird aus Performance-Gründen das Spiel gar nicht erst simuliert.
Nun wird das Spiel mit *run()* in *game.py* simuliert. Dabei wurde in *frame_output* vorher gespeichert, ob das Spiel eins aus zehn Spielen ist, bei dem die Züge des Spiels in einer Datei zu Überprüfungszwecken ausgegeben werden sollen.

```
def game_thread(arg):
    index = arg[0]
    dice_a = arg[1]
    dice_b = arg[2]
    id = arg[3]
    result = None
    a_can_win = 6 in dice_a
    b_can_win = 6 in dice_b
    if a_can_win and not b_can_win:
        result = 0
    if b_can_win and not a_can_win:
        result = 1
    if not a_can_win and not b_can_win:
        result = -1
    if result == None:
        frame_output = (index % (SIMULATIONS_PER_DICE_PAIR / 10)) == 0
        result = game.run(dice_a, dice_b, id=id, frame_output=frame_output)
    return result
```

Codeblock 4.2: game_thread() in main.py

4.2.3 game.py

In *game.py* befinden sich zunächst die beiden Funktionen *roll(dice)*, die ein zufälliges Element aus dem übergebenen Array *dice* zurückgibt, und *opp_view(pos)*, die die übergebene Position in die Perspektive des Gegners umrechnet (wichtig beim Schlagen).

Der wichtigste Teil ist aber natürlich *run()*[10]. Die Positionen wie auch die Würfel der beiden Spieler werden in den zweidimensionalen Arrays *pos* bzw. *d* gespeichert. Zuerst wird ausgewürfelt, wer beginnt. Diesen Wert speichere ich in Boolean *p*. So kann ich später einfach mit *d[p]* auf den Würfel des aktuellen Spielers zugreifen.
Der Game-Loop ist das Herzstück des gesamten Programms. Er ist begrenzt auf *4096* Durchläufe[11]. In ihm wird, wie schon in der Lösungsidee beschrieben, das Spiel ausgeführt. Noch einmal kurz zusammengefasst:
Das Positionen-Array wird aufsteigend sortiert. Der Würfel wird geworfen. Die zu bewegende Figur und das Zielfeld (hier: *piece* und *move_to*) werden berechnet. Der Zug wird ausgeführt und dabei möglicherweise ein Gegner geschlagen. Es wird überprüft, ob der aktuelle Spieler gewonnen hat und dann das Ergebnis zurückgegeben. Wenn nicht, wird, nur wenn keine Sechs gewürfelt wurde, der andere Spieler ausgewählt. Der Game-Loop beginnt wieder von vorne.

```
def run(dice_a, dice_b, frame_display=False, frame_output=False, id='0'):
    ###Prepare output file
    ...
    ###Prepare game
    pos = [[0, -1, -1, -1], [0, -1, -1, -1]]
    d = [dice_a, dice_b]
    p = None
    ###Roll out starting player
    while p == None:
        a = roll(d[0])
        b = roll(d[1])
        if a != b:
```

[10] Die Teile, die zur Ausgabe des Spiels als Datei dienen, sind hier weggelassen, da diese nicht wichtig für die Funktion des Programms sind.
[11] Es kann vorkommen, dass die Figuren vor dem Ziel stehen bleiben und nicht weiter gehen können, weil sie keine 1 würfeln (z.B., wenn keine 1 auf dem Würfel ist). Aus Performance-Gründen wird das Spiel nach 2^{12} = 4096 Zügen (ein normales dauert ca. 1000 Züge) abgebrochen und als Ergebnis Unentschieden zurückgegeben.

```
            p = a < b
        ###GAME LOOP
        for move_index in range(max_moves_per_game):
            pos[p].sort() #sort pieces
            dice_top = roll(d[p]) #roll dice
            ###Precalculate move
            piece = None
            move_to = None
            if dice_top == 6 and -1 in pos[p] and 0 not in pos[p]: #rolled a 6,
          piece on -1 and no piece on 0 -> get piece away from -1
                piece = pos[p].index(-1)
                move_to = 0
            else:
                for i in pos[p]: #select first piece which can move
                    if i + dice_top not in pos[p] and i + dice_top < field_size +
          4 and i != -1:
                        piece = pos[p].index(i)
                        move_to = i + dice_top
                        if i == 0 and -1 in pos[p]:
                            break
            ###Execute precalculated move
            if piece != None: #test if move can be made
                pos[p][piece] = move_to #actual move
                if opp_view(move_to) in pos[not p] and move_to < field_size:
          #moved to a field the opponent stands on -> capture its piece
                    pos[not p][pos[not p].index(opp_view(move_to))] = -1 #capture
          opponents piece
            ###Print frame
            ...
            ###Win detection
            if min(pos[p]) >= field_size:
                return int(p)
            ###Select other player
            if dice_top != 6:
                p = not p
        return -1
```

Codeblock 4.3: run() in game.py (gekürtzt)

4.2.4 Andere Module

Es gibt noch die Dateien *pgbar.py*, *reader.py* und *printer.py*. In *pgbar.py* wird ein Ladebalken für die Simulationen generiert. *reader.py* liest die Würfelliste aus einer Textdatei ein. *printer.py* gibt weitere Informationen wie das Ergebnis und optional auch die Spielzüge der Spiele aus. Es werden automatisch die Spielzüge von jeweils 10 Spielen pro Würfelpaar als Datei ausgegeben.

4.3 Beispiele

Die Simulation von 1 Mio. Spielen dauert bei mir (8 Kerne, 4,9GHz, Dualthreading) etwa 20 Sekunden. Man kann aber auch einfach beim Start des Programms die gewünschte Anzahl an Simulationen[12] übergeben. Will man beispielsweise eine Simulation mit 10 000 Spielen pro Würfelpaar starten, gibt man *python3 ~\ludo.py.zip 10000* in die Konsole ein.

4.3.1 Ausgaben

Hier ist die Ausgabe des Beispiels *wuerfel1.txt*. Diese und alle weiteren Ausgaben befinden sich in *wuerfel0-3_ausgabe.txt*.

[12] Ist diese kleiner als 1000 wird sie der Akkuratheit des Ergebnisses wegen auf 1000 gesetzt.

```
C: ~\>python3 ludo.py.zip
Enter dice meta file path: wuerfel1.txt
Decoding wuerfel1.txt...
Sucessfully loaded 6 dices from wuerfel1.txt:
[1, 2, 3, 4, 5, 6]
[2, 3, 4, 5, 6, 7]
[3, 4, 5, 6, 7, 8]
[4, 5, 6, 7, 8, 9]
[5, 6, 7, 8, 9, 10]
[6, 7, 8, 9, 10, 11]


Simulating game 1/15 1000000 times ([1, 2, 3, 4, 5, 6] vs. [2, 3, 4, 5, 6,
       7])
████████████████████████████████████ Done 1000000/1000000 00:22
[1, 2, 3, 4, 5, 6]  434897
[2, 3, 4, 5, 6, 7]  565103
game timeout / tie       0


Simulating game 2/15 1000000 times ([1, 2, 3, 4, 5, 6] vs. [3, 4, 5, 6, 7,
       8])
████████████████████████████████████ Done 1000000/1000000 00:21
[1, 2, 3, 4, 5, 6]  500653
[3, 4, 5, 6, 7, 8]  499347
game timeout / tie       0


Simulating game 3/15 1000000 times ([1, 2, 3, 4, 5, 6] vs. [4, 5, 6, 7, 8,
       9])
████████████████████████████████████ Done 1000000/1000000 00:21
[1, 2, 3, 4, 5, 6]  542505
[4, 5, 6, 7, 8, 9]  457495
game timeout / tie       0


Simulating game 4/15 1000000 times ([1, 2, 3, 4, 5, 6] vs. [5, 6, 7, 8, 9,
       10])
████████████████████████████████████ Done 1000000/1000000 00:22
[1, 2, 3, 4, 5, 6]   692163
[5, 6, 7, 8, 9, 10]  307837
game timeout / tie        0


Simulating game 5/15 1000000 times ([1, 2, 3, 4, 5, 6] vs. [6, 7, 8, 9, 10,
       11])
████████████████████████████████████ Done 1000000/1000000 00:22
[1, 2, 3, 4, 5, 6]    808300
[6, 7, 8, 9, 10, 11]  191700
game timeout / tie         0


Simulating game 6/15 1000000 times ([2, 3, 4, 5, 6, 7] vs. [3, 4, 5, 6, 7,
       8])
████████████████████████████████████ Done 1000000/1000000 01:15
[2, 3, 4, 5, 6, 7]  464622
[3, 4, 5, 6, 7, 8]  428534
game timeout / tie  106844
```

```
Simulating game 7/15 1000000 times ([2, 3, 4, 5, 6, 7] vs. [4, 5, 6, 7, 8,
       9])
████████████████████████████████████████ Done 1000000/1000000 01:26
[2, 3, 4, 5, 6, 7]  477861
[4, 5, 6, 7, 8, 9]  394270
game timeout / tie  127869


Simulating game 8/15 1000000 times ([2, 3, 4, 5, 6, 7] vs. [5, 6, 7, 8, 9,
       10])
████████████████████████████████████████ Done 1000000/1000000 01:53
[2, 3, 4, 5, 6, 7]   569584
[5, 6, 7, 8, 9, 10]  254500
game timeout / tie   175916


Simulating game 9/15 1000000 times ([2, 3, 4, 5, 6, 7] vs. [6, 7, 8, 9, 10,
       11])
████████████████████████████████████████ Done 1000000/1000000 02:11
[2, 3, 4, 5, 6, 7]    639412
[6, 7, 8, 9, 10, 11]  151734
game timeout / tie    208854


Simulating game 10/15 1000000 times ([3, 4, 5, 6, 7, 8] vs. [4, 5, 6, 7, 8,
       9])
████████████████████████████████████████ Done 1000000/1000000 02:21
[3, 4, 5, 6, 7, 8]  399119
[4, 5, 6, 7, 8, 9]  371571
game timeout / tie  229310


Simulating game 11/15 1000000 times ([3, 4, 5, 6, 7, 8] vs. [5, 6, 7, 8, 9,
       10])
████████████████████████████████████████ Done 1000000/1000000 03:09
[3, 4, 5, 6, 7, 8]   454180
[5, 6, 7, 8, 9, 10]  230467
game timeout / tie   315353


Simulating game 12/15 1000000 times ([3, 4, 5, 6, 7, 8] vs. [6, 7, 8, 9, 10,
       11])
████████████████████████████████████████ Done 1000000/1000000 03:40
[3, 4, 5, 6, 7, 8]    497820
[6, 7, 8, 9, 10, 11]  131664
game timeout / tie    370516


Simulating game 13/15 1000000 times ([4, 5, 6, 7, 8, 9] vs. [5, 6, 7, 8, 9,
       10])
████████████████████████████████████████ Done 1000000/1000000 03:54
[4, 5, 6, 7, 8, 9]   397179
[5, 6, 7, 8, 9, 10]  211707
game timeout / tie   391114


Simulating game 14/15 1000000 times ([4, 5, 6, 7, 8, 9] vs. [6, 7, 8, 9, 10,
       11])
████████████████████████████████████████ Done 1000000/1000000 04:28
[4, 5, 6, 7, 8, 9]    430391
```

```
[6, 7, 8, 9, 10, 11]  118951
game timeout / tie    450658


Simulating game 15/15 1000000 times ([5, 6, 7, 8, 9, 10] vs. [6, 7, 8, 9, 10,
       11])
████████████████████████████████████████ Done 1000000/1000000 06:00
[5, 6, 7, 8, 9, 10]   249528
[6, 7, 8, 9, 10, 11]  127135
game timeout / tie    623337
```

Ausgabe 4.1: Anwendung des Programms auf wuerfel1.txt

4.3.2 Einzelne Spiele

Das Programm gibt wie gesagt automatisch zehn Spiele pro Würfelpaar als Textdatei[13] aus. In den Dateien *spiel_beispiel0-9.txt* befinden sich Beispiele für Spiele mit zwei normalen Würfeln.

```
+++++  *****
+ A +  * 2 *
+++++  *****

AAA
a  b  c  d

#####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####
#   #  #   #  # A #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #
#####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####
#   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  # b #
#####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####

a  b  c  d

bbb

------------
+++++  *****
+ B +  * 4 *
+++++  *****

aaa
a  b  c  d

#####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####
#   #  #   #  # a #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #
#####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####
#   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  # B #  #   #  #   #  #   #  #   #
#####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####

a  b  c  d

BBB

------------
+++++  *****
+ A +  * 6 *
+++++  *****

AA
a  b  c  d

#####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####
# A #  #   #  # A #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #
#####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####
#   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  # b #  #   #  #   #  #   #  #   #
#####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####

a  b  c  d

bbb

------------
+++++  *****
+ A +  * 5 *
+++++  *****

AA
a  b  c  d

#####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####
#   #  #   #  # A #  #   #  #   #  # A #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #
#####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####
#   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  # b #  #   #  #   #  #   #  #   #
#####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####

a  b  c  d

bbb

------------
+++++  *****
+ B +  * 5 *
+++++  *****

aa
a  b  c  d

#####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####
#   #  #   #  # a #  #   #  #   #  # a #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #
#####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####
#   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  # B #  #   #  #   #  #   #  #   #  #   #  #   #  #   #  #   #
#####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####  #####

a  b  c  d

BBB

------------
```

Ausgabe 4.2: Spiel-Beispiel 1 (Zug 1-5)

[13] Diese heißen z.B. *1vs3_100000.txt* oder *3vs4_0.txt*

In den Dateien befinden sich die einzelnen Spielzüge als ASCII-Art (zu sehen in Ausgabe 4.2). Dabei ist das herkömmliche Mensch-ärgere-dich-nicht-Brett so modifiziert, dass die Felder in zwei Reihen übereinander liegen. Spieler A startet oben links und Spieler B unten rechts. Figuren werden durch Groß- und Kleinbuchstaben dargestellt, wobei Großbuchstaben für den Spieler am Zug stehen. Über dem Spielfeld findet sich nochmal der aktuelle Spieler im +-Kasten und das Würfelergebnis im *-Kasten. Darunter findet man die Figuren im Haus von A. Dann folgen die Zielfelder von A, die nicht wie die anderen Felder als ASCII-Art dargestellt sind, sondern einfach als Figuren unter der Beschriftung a-d. Unter dem Feld findet man die Zielfelder von B gefolgt vom Haus von B.

```
+ Spieler +  * Würfel *

HAUS
a  b  c  d
Zielfelder

SPIELFELD

a  b  c  d
Zielfelder
HAUS
```

Schema 4.1: Aufbau Spiel-Dateien

4.4 Erkenntnisse

Durch die hohe Wiederholungsrate von 1 Mio. Spielen pro Würfelpaar sind die Ergebnisse nach jedem Durchlauf ziemlich ähnlich. Es gibt kaum Abweichungen, also sind die Ergebnisse auch ziemlich genau.

In jedem Beispiel gab es einen Würfel, der gegen alle anderen Würfel in dem Beispiel gewonnen hat. Diese sind hier als Gewinner aufgelistet.

Beispiel	Gewinner-Würfel
`wuerfel0.txt`	`[1, 1, 1, 6, 6, 6]`
`wuerfel1.txt`	`[2, 3, 4, 5, 6, 7]`
`wuerfel2.txt`	`[1, 6, 6, 6, 6, 6]`
`wuerfel3.txt`	`[1, 2, 5, 6]`

Tabelle 4.1: Gewinner der Beispiele

Je höher die Wahrscheinlichkeit für eine Sechs ist, umso besser ist der Würfel. Fehlen kleine Zahlen wie 1, 2 oder 3 verschlechtert sich seine Gewinnchance, da man oft vor dem Ziel stehen bleibt und nicht mehr weiter gehen kann, weil die Zahlen, die man würfelt, zu hoch sind. Diese Schwäche ist beim Gewinner von *`wuerfel1.txt`* aber durch die zusätzliche *7* ausgeglichen.