Dokumentation Aufgabe 2

Bundeswettbewerb Informatik 2.Runde

Aufgabenstellung:

Es soll ein Programm geschrieben werden, das Angaben für zwei gleich große Labyrinthe (m*n) einliest und dann einen Weg von A nach B für die zwei Spieler in den beiden Labyrinthen findet und ausgibt. Beide Spieler beginnen bei den Koordinaten „0, 0“ und wollen zu den Koordinaten „m, n“ gehen. Das bedeutet, sie starten beide an der linksoberen Ecke und gehen zu der rechtsunteren Ecke. Das Programm soll den kürzesten Weg finden, bei dem beide Spieler in den Labyrinthen mit denselben Anweisungen zum Ziel kommen. Wenn ein Spieler mit einer Anweisung an eine Wand stößt oder ein Weg das Ziel bereits erreicht hat, bleibt sie dort stehen. Die Anweisungen werden auf Englisch dargestellt. „D“ für runter („Down“), „U“ für hoch („Up“), „R“ für rechts („Right“) und „L“ für links („Left“).

Zusatz zu b):

Es gibt jetzt auch Gruben im Labyrinth und wenn man in das Feld reinläuft, wo es liegt, kommt man wieder zum Anfang (zu den Koordinaten „0, 0“ an).

Die Regel:

Wir müssen zuerst die Daten der Labyrinthe einlesen. Danach beginnen wir mit einer Hauptfunktion. Wir müssen immer aus einem Punkt die nächsten Möglichkeiten finden, wo man hingehen kann. Dann speichern wir das in eine Liste und wiederholen den

Vorgang mit der neuen Liste, usw. Wenn einmal beide bei den Koordinaten „m, n“ sind, hört das Programm auf und gibt den Pfad aus.

Umsetzung:

```
import sys
```

Hiermit importiere ich die Bibliothek „sys“. Ich brauche sys, um den Filenamen der Eingabedatei (Text) auf der Kommandozeile zu bekommen.

```
def hauptfunktion(p1, p2):
    list1 = [(["D"], ((0, 0), (0, 0)))]
    list2 = []
```

Das ist die Definition der Hauptfunktion. Sie ist die Hauptfunktion und ist dazu da, den ganzen Weg mit anderen Definition zu generieren. Ihre Argumente sind die Labyrinthe der Spieler eins und zwei („p1“, „p2“). In „p1“ und „p2“ stehen die Daten der jeweiligen Labyrinthe, wo die Wände sind und wo die Gruben sind (in einem Dict). Zuerst werden zwei Listen erstellt. „list1“ enthält die bereits besuchten Pfade, welche in einem „Tuple“ aus einer Liste von gegangenen Richtungen und den Endpunkten dieser Anweisung in den jeweiligen Labyrinthen besteht. Da der Anfang der beiden Wege (0, 0) ist und man eine Richtung braucht, die benutzt wurde (da es die linksobere Ecke des Labyrinthes ist, ist es egal ob man von oben kommt oder von links, daher der Platzhalter „D“, da man von hier ohnehin nicht mehr in diese Richtung hingeht),

ist dies das Anfangsstadium von „list1“. „list2“ ist dazu da, neugenerierte Pfade aus „list1“ aufzunehmen.

```
goal = (p1["size"][0] - 1, p1["size"][1] - 1)
visited = set()
while list1:
    while list1:
        lastpath = list1.pop(0)
        locations = lastpath[1]
```

Dann setzen wir das Tuple “goal” auf die Breite des Feldes minus eins und die Höhe des Labyrinthes minus eins, da wir ja von 0, 0 abzählen. Zuletzt erstellen wir noch ein Set, das „visited“ heißt, es wird die schon besuchten Punkte der zwei Spieler speichern. Dann wiederholen wir, bis „list1“ leer ist, eine innere Schleife, die sich wiederholt, bis „list1“ leer ist. Wir brauchen zwei „while“-Schleifen, weil wir später „list1“ und „list2“ tauschen wollen und wir das außerhalb der inneren „while“-Schleife machen wollen, wir aber trotzdem das Ganze noch wiederholen wollen. In der inneren Schleife nehmen wir aus „list1“ das erste Element aus, das wir dann in „lastpath“ speichern. Wir werden es später noch gebrauchen. „locations“ ist der zweite Teil von „lastpath“, d.h. die Punkte der Koordinaten, bei diesem Wegschritt. „lastdirection“ ist auch von „lastpath“ herausgenommen worden. Es ist die letzte Anweisung, die in „lastpath“ aufgezeichnet wurde.

```
possibilities = nextpossiblefields(p1, p2, locations)
for direction in possibilities:
    nextpath = lastpath[0].copy()
    nextpath.append(direction)
    nextlocations = possibilities[direction]
```

Nun erstellen wir „possibilities“. Sie speichert die nächsten Möglichkeiten von den zwei Spielern, sich weiter zu bewegen mithilfe von der Definition „nextpossiblefields“ (siehe unten) . Dies besteht aus einem Dict und der „Schlüssel“ ist die Richtung (Anweisung) und der „Wert“ sind die Punkte von den Spielern nach der jeweiligen Anweisung. Dann wiederholt sich folgendes für jede Richtung in „possibilities“: „nextpath“ wird von „lastpath“ kopiert. Es ist der Pfad mit den Anweisungen, der kopiert wird. Da ja mit der Denifition „nextpossiblefields“ schon alle möglichen Richtungen kalkuliert wurden, können wir in dieser Schleife einfach die neue Richtung an „nextpath“ anhängen um den neuen Weg zu erhalten. Zuletzt erstellen wir noch „nextlocations“. Sie ist die Koordinaten der nächsten Felder.

```
        if nextlocations == (goal, goal):
            return nextpath[1:]
        if not nextlocations in visited:
            visited.add(nextlocations)
            list2.append((nextpath, nextlocations))
    list1, list2 = list2, list1
```

```
return []
```

Wir schauen dann bei jedem Schleifenduchgang, ob die Koordinaten „nextlocations“ mit „goal“ übereinstimmen. Wenn ja, dann sind wir am Ziel und geben den vollständigen Pfad „nextpath“ zurück, außer die erste Richtung, denn wir haben ja am Anfang als Platzhalter die Richtung „D“ an „nextpath“ angehängt. Wenn nicht, dann schauen wir ob „nextlocations“ in „visited“. Wenn ja, dann haben wir die Punkte schon besucht und wir können es verwerfen. Wenn nein, dann nehmen wir zuerst „nextlocations“ und tun es in „visited“ und dann hängen wir den Pfad mit der neuen Richtung (also „nextpath“) und die Punkte nach der Anweisung an „list2“. So gehen wir durch alle Richtungen durch. Aus der Schleife heraus, vertauschen wir „list1“ und „list2“. Das machen wir die ganze Zeit, bis die beiden Spieler am Ziel angekommen sind. Wenn aber in der äußeren „while“-Schleife „list1“ leer geworden ist (was nicht der Fall sein soll, aber nur zur Sicherheit), geben wir eine leere Liste zurück. Nun schauen wir uns mal die Definition „nextpossiblefields“ an.

```
def nextpossiblefields(p1, p2, locations):
    width = p1["size"][0]
    height = p1["size"][1]
    returndict = {}
    directions = ["R", "L", "U", "D"]
```

Das ist die Definition „nextpossiblefields“. Sie ist dazu da, die Koordinaten der nächsten möglichen Felder, wo man hingehen kann, zu generieren. Ihre Argumente sind auch die Daten der

Labyrinthe der Spieler eins und zwei („p1“, „p2“). Sie braucht aber auch die Koordinaten der Spieler (also „locations“) und die Richtung, aus der sie gekommen sind (also „lastdirection“). Zuerst wird die Breite und die Höhe aus „p1“, den Labyrinthdaten, ausgelesen und in „width“ und „height“ gespeichert. Dann wird eine leere Dict hergestellt, die „returndict“ heißt. Das ist das Dict das zurückgegeben wird und in der Definition „hauptfunktion“ verwendet wird. Dann erstellen wir eine Liste mit den vier Richtungen, die „directions“ heißt.

```
for direction in directions:
```

Wir wiederholen jetzt folgendes für jede Richtung in “directions”, um zu schauen, welche Koordinaten wir mit jeder Richtung bekommen.

```
if has_pit(p1, locations[0], direction) or has_pit(p2, locations[1], direction):
    newlocation1 = locations[0]
    newlocation2 = locations[1]
else:
```

Dieser Teil ist Zusatz zu Aufgabe 2b). Hier schauen wir mit der Definition „has_pit“ (siehe unten), ob an der Stelle wo Spieler eins und Spieler zwei sind, nachdem sie in die Richtung gegangen sind, eine Grube ist. Wenn einer von denen eine Grube ist, dann bleiben sie an derselben Stelle stehen und gehen nicht in die Richtung. „newlocation1“ und „newlocation2“ sind die neuen Koordinaten der zwei Spieler und werden zu den jetzigen Koordinaten, die

schon vorgegeben wurden, da die Spieler sich nicht bewegen. Das heißt wir verwerfen diese Richtung. Wenn aber keine Grube ist, dann passiert folgendes:

```
if has_barrier(p1, locations[0], direction) or locations[0] == (width – 1, height – 1):
    newlocation1 = locations[0]
else:
    newlocation1 = nextlocation(locations[0], direction)
```

Wir schauen jetzt bei Spieler eins mithilfe von der Definition „has_barrier“ (siehe unten), ob in dieser Richtung eine Wand ist oder ob der Spieler am Ziel ist. Wenn „True“ rauskommt, dann heißt es, es gibt eine Wand oder man ist am Ziel, deswegen bewegen wir uns nicht, „newlocation1“ ändert sich nicht und „newlocation1“ speichert die angegebenen Koordinaten („locations“). Wenn „False“ rauskommt, dann ist da keine Wand, und wir können mithilfe von der Definition „nextlocation“ (siehe unten) die Koordinaten der nächsten Punkte herausfinden und in „newlocation1“ speichern.

```
if has_barrier(p2, locations[1], direction) or locations[1] == (width – 1, height – 1):
    newlocation2 = locations[1]
else:
```

```
    newlocation2 = nextlocation(locations[1], direction)
```

Dasselbe machen wir mit Spieler zwei.

```
    if (newlocation1, newlocation2) != locations:
        returndict[direction] = (newlocation1, newlocation2)
return returndict
```

Wir haben jetzt bei Spieler eins und zwei separat geschaut, ob da eine Wand ist. Jetzt müssen wir die Richtungen verwerfen, wo beide auf eine Wand zustoßen (Oder andersherum gesagt, die speichern, wo einer oder beide sich auf ein anderes Feld bewegt haben.) Wir schauen ob, “newlocation1” und “newlocation2” sich verändert haben. Wenn sie nicht wie die schon angegebenen Koordinaten (“locations”) sind, heißt es, dass sie sich bewegt haben und wir speichern die Richtung (“direction”) mit den neuen Koordinaten von Spieler eins und zwei (“newlocation1”, “newlocation2”) in das Dict “returndict”. Wenn wir alle Richtungen haben und aus der Schleife raus sind, geben wir “returndict” zurück.

```
def nextlocation(location, direction):
    if direction == "D":
        return (location[0], location[1] + 1)
    if direction == "U":
        return (location[0], location[1] – 1)
```

```
    if direction == "L":
        return (location[0] - 1, location[1])
    if direction == "R":
        return (location[0] + 1, location[1])
```

Das ist die Definition “nextlocation”. Sie nimmt als Argumente zwei Koordinaten (“location”) und eine Richtung (“direction”) ein und gibt die Koordinaten des Feldes, nachdem der Spieler in die Richtung gegangen ist, zurück. Das geschieht mit einem kleinem Programm, wo man nach der Richtung fragt und dann für die Richtung eine Koordinate ändert und sie dann zurückgibt.

```
def has_barrier(labyrinth, location, direction):
    if direction == "D":
        if location[1] >= height - 1:
            return True
        return
        labyrinth["horizontal_barriers"][locati
        on[1]][location[0]] == 1
    if direction == "L":
        if location[0] <= 0:
            return True
        return
        labyrinth["vertical_barriers"][location
        [1]][location[0] - 1] == 1
```

```
    if direction == "R":
        if location[0] >= width - 1:
            return True
        return
        labyrinth["vertical_barriers"][location
        [1]][location[0]] == 1
    if direction == "U":
        if location[1] <= 0:
            return True
        return
        labyrinth["horizontal_barriers"][locati
        on[1] - 1][location[0]] == 1
```

Dies ist die Definition “has_barrier”. Sie ist dazu da, um für einen Punkt zu schauen, ob man, wenn man in die Richtung geht, die angegeben ist, auf eine Wand stößt. Wenn da eine Wand ist, gibt sie “True” aus und wenn da keine Wand ist, gibt sie “False” aus. Ihre Argumente sind die Daten eines Labyrinthes (“labyrinth”), die Koordinaten des Spielers (“location”) und die Richtung (“direction”). Wir schauen dann zuerst, welche Richtung “direction” ist. Bei jeder Richtung schauen wir, ob wir an der Grenze (oder Umriss) des Labyrinthes stoßen, denn sie stehen ja nicht in den Daten der Labyrinthe. Wenn ja, geben wir “True” aus. Wenn nicht, schauen wir (für jede Richtung ist es anders) in den Daten, ob in der Richtung, eine Wand ist und geben “True” oder “False” aus.

```
def has_pit(labyrinth, location, direction):
```

```
pits = labyrinth["pits"]
if direction == "D":
    newlocation = (location[0], location[1] + 1)
    for pit in pits:
        if newlocation == pit:
            return True
    return False
if direction == "U":
    newlocation = (location[0], location[1] - 1)
    for pit in pits:
        if newlocation == pit:
            return True
    return False
if direction == "R":
    newlocation = (location[0] + 1, location[1])
    for pit in pits:
        if newlocation == pit:
            return True
    return False
```

```
    if direction == "L":
        newlocation = (location[0] - 1,
        location[1])
        for pit in pits:
            if newlocation == pit:
                return True
        return False
```

Dieser Teil ist Zusatz zu Aufgabe 2b). Das ist die Definition „has_pit“. Sie ist dazu da, zu schauen, ob man von einem Punkt im Feld, wenn man in die angegebene Richtung geht, auf eine Grube stößt. Wenn es eine Grube dort gibt, gibt „has_pit“ „True“ zurück, und wenn da keine Grube ist, gibt „has_pit“ „False“ zurück. Ihre Argumente sind auch wie „has_barrier“ die Daten des Labyrinthes („labyrinth“), den jetzigen Punkt („location“) und die Richtung („direction“). Zuerst nehmen wir aus den Daten des Labyrinthes die Koordinaten der Gruben heraus und speichern es in „pits“. Dann schauen wir, welche Richtung „direction“ ist. Danach genieren wir die nächsten Koordinaten, wo wir nach der Grube schauen wollen (bei jeder Richtung ist es anders) und speichern es in „newlocation“. Dann schauen wir für jede Grube in „pits“, ob die Koordinaten der Grube mit „newlocation“ übereinstimmen. Wenn ja, dann ist da eine Grube und wir geben „True“ zurück. Wenn nein, dann geben wir „False“ zurück.

```
file = open(sys.argv[1], "r")
grubenort1 = []
```

```
grubenort2 = []
lines = [l.rstrip() for l in file.readlines()]
(width, height) = [int(l) for l in lines[0].split(" ")]
lines = lines[1:]
```

Hier ist das Einlesen und Aufteilen der Daten der Labyrinthe, wo wir den Grundbaustein aufbauen. Zuerst lesen wir die Datei ein und speichern es in “file”. Dann erstellen wir zwei leere Listen: “grubenort1” und “grubenort2”. Sie werden später die Gruben der zwei Labyrinthe aufzeichnen. Dann nehmen wir “file” und teilen es in Zeilen auf und speichern es in “lines”. Danach nehmen wir die erste Zeile (das ist die Größe des Labyrinthes) und separaten die Breite und die Höhe, wandeln die beiden in “integer” und speichern sie in “width” und “height”. Zuletzt verkürzen wir “lines” um eine Zeile, da wir die erste Zeile herausgenommen haben.

```
lines = [[int(width) for width in line.split(" ")]for line in lines]
vertical1 = lines[0:height]
lines = lines[height:]
horizontal1 = lines[0:height-1]
lines = lines[height-1:]
gruben1 = int(lines[0][0])
lines = lines[1:]
```

Jetzt verwandeln wir alle Zeilen von “lines” in Listen um und konvertieren die Zahlen in den Zeilen (die gerade “strings” sind) zu “integers”. Jetzt müssen wir nur die Zeilen zu unterschiedlichen Variablen ordnen. Wir ordnen die ersten “height” Zeilen zu “vertical1”. Das sind die vertikalen Wänder. Wir regenieren “lines” wieder, aber ohne die “vertical1”-Zeilen. Danach nehmen wir die nächsten “height” minus eins Zeilen und tun sie in “horizontal1”. Das sind die horizontalen Wände. Dann regenieren wir “lines”. Jetzt nehmen wir die erste Zeile (das ist die Anzahl von Gruben) und setzen es in “gruben1”. Zuletzt regenieren wir “lines”.

```
if gruben1 > 0:
    for z in range(gruben1):
        grubenort1.append((lines[z][0],
    lines[z][1]))
    lines = lines[z+1:]
else:
    lines = lines[:]
```

Nun schauen wir ob “gruben1” größer als null ist. Wenn ja, dann gibt es in den nächsten Reihen noch die Koordinaten der Gruben. Also wiederholen wir Folgendes “gruben1”-mal: Wir hängen an “grubenort1”, die Liste, die wir schon erstellt haben, die Zeilen wo die Gruben sind als Tuple. Aus der Schleife heraus, regenerieren wir “lines”. Wenn aber “gruben1” nicht größer als null ist (also null ist), können wir einfach “lines” so lassen, wie es gerade ist. Das war alles zum ersten Labyrinth. Jetzt machen wir das Gleiche beim zweiten Labyrinth.

```
vertical2 = lines[0:height]
lines = lines[height:]
horizontal2 = lines[0:height-1]
lines = lines[height-1:]
gruben2 = int(lines[0][0])
lines = lines[1:]
if gruben2 > 0:
    for z in range(gruben2):
        grubenort2.append((lines[z][0], lines[z][1]))
    lines = lines[z+1:]
else:
    lines = lines[:]
```

So haben wir alle Daten der zwei Labyrinthe.

```
labyrinth1 = {"size": (width, height), "vertical_barriers": vertical1, "horizontal_barriers": horizontal1, "pits": grubenort1}
labyrinth2 = {"size": (width, height), "vertical_barriers": vertical2, "horizontal_barriers": horizontal2, "pits": grubenort2}
```

```
print("Der Pfad lautet: " + ", ".join(hauptfunktion(labyrinth1, labyrinth2)))
```

Hier bauen wir die Daten der Labyrinthe zusammen. Dafür benutzen wir zwei Dicts, die als “size” die Breite und Höhe haben, als “vertical_barriers” die vertikalen Wänder, als “horizontal_barriers” die horizontalen Wänder und als “pits” die Koordinaten der Gruben. Jetzt haben wir die Daten aufgebaut. Zuletzt geben wir “Der Pfad lautet: “ und angehängt mit dem Pfad der Ausgabe von der “hauptfunktion” zusammen aus.

ENDE UMSETZUNG

Beispiele:

labyrinthe0.txt

3 3

1 0

1 1

0 1

0 0 0

0 0 0

0

1 0

0 1

0 0

0 0 0

1 1 0

1

0 2

Output: a) Der Pfad lautet: D, D, R, U, U, R, D, D

b) Der Pfad lautet: D, D, R, U, U, R, D, D

labyrinthe1.txt

5 4

1 0 0 0

1 0 0 1

1 0 1 1

0 1 0 1

0 0 1 1 0

0 1 1 0 0

0 0 0 0 0

0

0 0 0 0

0 1 0 1

1 0 1 0

0 0 0 0

1 1 1 1 0

0 0 0 0 0

0 1 1 1 1

0

Output: a) Der Pfad lautet: R, R, R, R, D, D, L, U, L, D, L, U, L, D, D, R, U, R, D, R, U, U, L, L, U, R, R, R, D, D, D

b) Der Pfad lautet: R, R, R, R, D, D, L, U, L, D, L, U, L, D, D, R, U, R, D, R, U, U, L, L, U, R, R, R, D, D, D

labyrinthe2.txt

10 10

```
001010000
011101001
000001100
001100001
100000101
011010001
011111101
000000101
100011110
011110001
0100101100
1100101011
1111110100
0100111100
0101010010
0111100011
1000011110
0101101000
0100000010
3
02
95
39
010011101
111101110
000000100
111110001
000100000
100111100
001010101
001001010
101000100
010110011
0011000000
0100100101
1100011100
0001111011
1110011110
```

```
0011000101
0110110010
0010111001
1000001100
9
11
21
03
13
96
27
57
79
89
```

Output: a) Der Pfad lautet: R, R, L, D, R, D, D, R, R, R, U, U, R, R, R, D, L, D, R, D, R, D, L, D, L, U, L, L, L, L, U, L, L, D, D, L, D, R, D, R, R, R, R, D, D, R, R, U, U, U, R, U, U, R, R, R, R, R, D, L, D, L, D, D, R, R, D

b) Der Pfad lautet: R, R, L, D, R, D, D, R, R, R, U, U, R, R, R, D, L, D, R, D, R, D, L, D, L, U, L, L, L, L, U, L, L, D, D, L, D, R, D, R, R, R, R, D, D, R, R, U, U, U, R, U, U, R, R, R, R, R, D, L, D, L, D, D, R, R, D

labyrinthe3.txt (Um Platz zu sparen, haben wir die Datei ausgelassen)

Output: a) Der Pfad lautet: D, D, D, D, R, D, D, R, U, R, R, R, D, D, D, D, R, R, R, R, D, L, D, L, D, D, D, D, D, L, L, D, L, D, L, D, D, D, D, R, R, D, D, R, R, D, D, R, L, D, D, D, L, D, R, D, L, L, D, L, U, L, D, D, R, R, D, D, R, R, D, L, D, D, R, R, R, R, D, D, R, R, D, R, R, D, R, D, D, L, L, D, D, L, D, L, D, D, R, U, R, D, D, L, D, L, D, L, U, L, U, L, U, L, D, L, D, R, R, D, D, D, R, R, U, R, R, U, R, U, R, D, D, L, D, D, L, L, L, U, R, U, L, D, D, D, D, L, D, D, D, R, U, R, D, R, D, R, D, R, R, R, R, R

b) Der Pfad lautet: D, D, D, D, R, D, D, R, U, R, R, R, D, D, D, D, R, R, R, R, D, L, D, L, D, D, D, D, D, L, L, D, L, D, L, D, D, D, D, R, R, D, D, R, R, D, D, R, L, D, D, D, L, D, R, D, L, L, D, L, U, L, D, D, R, R, D, D, R, R, D, L, D, D, R, R, R, R, D, D, R, R, D, R, R, D, R, D, D, L, L, D, D, L, D, L, D, D, R, U, R, D, D, L, D, L, D, L, U, L, U, L, U, L, D, L, D, R, R, D, D, D, R, R, U, R, R, U, R, U, R, D, D, L, D, D, L, L, L, U, R, U, L, D, D, D, D, L, D, D, D, R, U, R, D, R, D, R, D, R, R, R, R, R

<u>labyrinthe4.txt (Achtung! Die Datei ist zu groß um sie hier darzustellen)</u>

Zu groß! Musste abgebrochen werden!

<u>labyrinthe5.txt (Achtung! Die Datei ist zu groß um sie hier darzustellen)</u>

Zu groß! Musste abgebrochen werden!

<u>labyrinthe6.txt (Achtung! Die Datei ist zu groß um sie hier darzustellen)</u>

Output: a) Der Pfad lautet: R, D, D, D, D, D, D, D, D, D, D, D, D, D, D, D, D, D, D, D, D, D, D, D, D,

D, D

b) Der Pfad lautet: R, R, R, D, D, D, D, D, D, D, D, R, R, D, R, D, D, R, R, R, R, R, D, R, R, R, R, R, D, D, R, D, R, R, R, D, R, D, R, R, R, R, D, R, R, D, R, D, R, R, R, R, D, D, D, D, R, R, D, D, D, R, D, R, D, R, R, R, R, R, R, R, D, D, R, D, R, R, D, R, R, R, R, D, R, R, D, R, R, R, D, D, R, R, R, R, D, R, R, R, R, R, R, R, R, R, R, R, R, D, R, D, D, D, D, R, R, R, R, R, R, R, D, R, R, D, R, D, D, R, R, R, R, R, R, R, R, R, R, R, D, R, R, D, R, D, D, R, D, R, D, R, R, R, D, R, R, R, D, R, D, R, D, R, R, R, R, R, R, R, D, D, R, R, R, R, D, D, R, R, R, D, R, R, D, R, D, R, D, R, R, R, R, R, R, R, R, R, R, D, R, D, D, R, R, R, R, D, R, R, R, D, R, R, D, R, R, R, D, D, R, D, D, D, D, R, R, R, R, R, R, R, D, R, R, D, D, R, R, R, R, R, R, R, R, R, R, R, R, R, D, R, D, D, R, R, D, D, D, D, R, R, D, D, R, R, D, D, R, R, R, R, D, D, D, D, D, R, R, D, D, D, D, D, D, D, D, D, R, D, D, D, R, D, D, D, D, D, R, R, R, D, R, D, R, D, D, D, D, D, D, R, D, R, D, D, D, R, R, R, D, D, R, D, D, R, D, R, R, D, D, D, D, D, R, D, D, D, D, D, D, R, D, D, D, D, D, D, D, R, D, D, R, R, D, D, D, D, D, D, D, D, R, D, D, D, D, D, D, R, D, R, D, R, R, R, D, D, D, D, D, D, R, R, R, R, D, D, D, D, R, D, D, D, D, D, D, D, D, D, R, D, D, D, R, D, D, D, R, D, D, R, D, D, R, D, R, R, D, D, R, D, D, R, R, D, D, D, R, D, R, R, R, R, R, R, D, D, D, D, R, D, R, D, D, D, D, D, D, D, D, R, R, D, D, D, D, D, D, D, D, R, R, R, R, D, D, D, R, D, D, D, D, D, D, D, R, R, D, R, R, D, D, R, D, D, D, D, D, D, R, D, R, D, D, R, D, D

labyrinthe7.txt (Um Platz zu sparen, haben wir die Datei ausgelassen)

Output: a) Der Pfad lautet: R, R, R, D, L, L, D, D, L, D, D, D, R, R, D, R, U, R, D, R, R, U, R, U, U, R, R, R, R, R, R, D, D, L, D, L, L, D, D, R, R, R, D, D, R, D, R, U, R, U, R, R, U, R, R, U, R, U, U, L, U, U, L, U, R, R, D, R, U, R, D, R, D, L, L, D, R, D, R, R, U, R, R, U, R, R, R, D, D, R, R, U, U, R, D, R, R, R, D, R, D, D, L, L, D, L, L, D, D, R, D, R, R, R, R

b) R, R, R, D, L, L, D, D, L, D, D, D, R, R, D, R, U, R, D, R, R, U, R, U, U, R, R, R, R, R, R, D, D, L, D, L, L, D, D, R, R, R, D, D, R, D, R, U, R, U, R, R, U, R, R, U, R, U, U, L, U, U, L, U, R, R, D, R, U, R, D, R, D, L, L, D, R, D, R, R, U, R, R, U, R, R, R, D, D, R, R, U, U, R, D, R, R, R, D, R, D, D, L, L, D, L, L, D, D, R, D, R, R, R, R

labyrinthe8.txt (Achtung! Die Datei ist zu groß um sie hier darzustellen)

Zu groß! Musste abgebrochen werden!

Labyrinthe9.txt (Achtung! Die Datei ist zu groß um sie hier darzustellen)

Zu groß! Musste abgebrochen werden!