Junioraufgabe 1 – Luftballons

Lösungsidee

Meine Idee war, solange eine Kombination der Speicherfächer zu finden, mit der im Ausgabefach 20 Luftballons wären, bis alle Speicherfächer leer sind.

Umsetzung

Die Lösungsidee wird in Python implementiert. Der Dateiname wird entweder als Kommandozeilen-Argument oder als Benutze-Eingabe übergeben.

Die Klasse `LVM` soll die Luftballonverpackungsmaschine simulieren. Die LVM wird mit einer Füllfolge initialisiert. Die Funktion `fach` und `verpacken` funktionieren wie in der Aufgabenstellung beschrieben. Zusätzlich geben sie jeweils zurück, wie viele Ballons im Speicherfach bzw. Ausgabefach vor der Operation waren.

Mit der Funktion `fülle_0er` wird versucht, alle leeren Speicherfächer der LVM wieder aufzufüllen. Dazu wird zunächst überprüft, ob die Füllfolge noch nicht leer ist und auch mindestens ein Speicherfach leer ist. Dann wird für jedes Speicherfach überprüft, ob es leer ist und die Füllfolge noch nicht aufgebraucht ist. Ist dies der Fall, wird es wieder aufgefüllt. Diese Schritte werden wiederholt, bis die oben zuerst genannte Bedingung nicht mehr erfüllt ist.

Die Funktion `lade_lvm_aus_datei` lädt eine Füllfolge aus einer Datei und erstellt eine LVM.

Mit der Funktion `finde_kombination` wird eine Kombination der Speicherfächer gefunden, mit der eine bestimmte Anzahl von Luftballons im Ausgabefach landen würden. Zusätzlich versucht die Funktion dieses Ziel mit möglichst wenig Speicherfächern zu erreichen, weswegen sie eventuell bei manchen Füllfolgen nicht die optimale Lösung finden kann.

In der Funktion `simuliere_lösung` werden die oben beschriebenen Funktion dazu verwendet, möglichst viele Packungen mit möglichst nur 20 Luftballons zu befüllen. Dazu wird in einer Schleife, die wiederholt wird, bis alle Speicherfächer leer sind, zunächst die Funktion `fülle_0er` aufgerufen, um eventuell leere Speicherfächer wieder aufzufüllen. Danach wird nach einer Kombination der Speicherfächer gesucht, mit der im Ausgabefach 20 Ballons wären. Falls diese gefunden wird, wird sie angewandt. Dann wird natürlich auch die Operation Verpacken() durchgeführt. Falls es keine Kombination finden konnte, wird das Speicherfach mit den wenigsten Ballons entleert. Falls danach im Ausgabefach 20 Ballons liegen, wird es verpackt. Theoretisch könnte man stattdessen versuchen, die Füllfolge und die Speicherfächer zu kombinieren, was für die Beispiele jedoch nicht nötig ist. Jede Operation an der LVM wird ausgegeben.

Zum Schluss gibt das Programm noch die Packungen und insgesamt verbrauchten Luftballons aus.

Beispiele

Die Dateien test.sh bzw. test.bat rufen das Programm jeweils mit allen Dateien nacheinander auf. Zudem wird der jeweilige Dateinamen ausgegeben. Die vollständige Ausgabe des Skriptes (mit allen Operationen) ist in der Datei ergebnis.txt gespeichert worden.

```
$ sh test.sh > ergebnis.txt
luftballons1.txt
FACH(1)
...
FACH(8)
Insgesamt wurden 4 Packungen verpackt:
[20, 20, 20, 20]
Insgesamt wurden 89 Ballons verbraucht
luftballons2.txt
FACH(1)
...
VERPACKEN()
Insgesamt wurden 4 Packungen verpackt:
[20, 20, 20, 20]
Insgesamt wurden 80 Ballons verbraucht
luftballons3.txt
FACH(1)
...
FACH(5)
Insgesamt wurden 24 Packungen verpackt:
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
20, 20, 20, 20, 20, 20, 20, 20]
Insgesamt wurden 491 Ballons verbraucht
luftballons4.txt
FACH(1)
...
FACH(6)
Insgesamt wurden 22 Packungen verpackt:
[20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
20, 20, 20, 20, 20, 20]
Insgesamt wurden 458 Ballons verbraucht
luftballons5.txt
FACH(1)
...
FACH(10)
```

```
Insgesamt wurden 17 Packungen verpackt:
[20, 20, 20, 20, 20, 20, 20, 20, 20, 30, 30, 30, 30, 30, 30, 30, 30]
Insgesamt wurden 435 Ballons verbraucht
luftballons6.txt
FACH(1)
...
FACH(10)
Insgesamt wurden 17 Packungen verpackt:
[20, 20, 20, 20, 20, 20, 20, 20, 20, 30, 30, 30, 30, 30, 30, 30, 30]
Insgesamt wurden 435 Ballons verbraucht
luftballons7.txt
FACH(1)
...
VERPACKEN()
Insgesamt wurden 5 Packungen verpackt:
[20, 20, 20, 20, 20]
Insgesamt wurden 100 Ballons verbraucht
luftballons8.txt
FACH(1)
...
FACH(9)
Insgesamt wurden 8 Packungen verpackt:
[20, 20, 20, 20, 20, 20, 20, 20]
Insgesamt wurden 175 Ballons verbraucht
```

Quelltext (gekürzt)

junior_aufgabe1,py

```
import sys
import itertools

# Luftballonverpackungsmaschine
class LVM:
    def __init__(self, füllfolge):
        [...]

    def fach(self, i):
        [...]

    def verpacken(self):
        [...]
```

```
def lade_lvm_aus_datei(filename):
    [...]
    return LVM(füllfolge)

def find_kombination(lvm, größe):
    beste_kombinations_länge = 1000
    beste_kombination = None
    # die range geht von 10 abwärts bis einschließlich 1
    for kombinations_länge in range(10, 0, -1):
        for kombination in itertools.combinations(range(10),
                                                  kombinations_länge):
            # hier wird die Anzahl der Luftballons berechnet,
            #  die mit dieser Kombination im Ausgabefach landen würden
            kombinations_größe = 0
            for speicher_index in kombination:
                kombinations_größe += lvm.speicher[speicher_index]

            # danach wird überprüft, ob die Anzahl auch der gefordeten Größe
entspricht
            #  außerdem sollte die kombination kleiner als die bisher gefundene
kombination sein
            if kombinations_größe == größe and kombinations_länge <
beste_kombinations_länge:
                beste_kombination = kombination
                beste_kombinations_länge = kombinations_länge
    return beste_kombination

# in dieser Funktion werden alle leeren Speicherfächer wieder aufgefüllt
def fülle_0er(lvm):
    [...]

def verpacken(lvm):
    kombinations_größe = lvm.ausgabe_fach
    lvm.verpacken()
    print("VERPACKEN()")
    return kombinations_größe

def fach(lvm, i):
    menge = lvm.speicher[i]
    print("FACH("+str(i+1)+")")
    lvm.fach(i+1)
    return menge

def zeige_endergebnis(lvm, verbrauchte_ballons, packungen):
    print("Insgesamt wurden", len(packungen), "Packungen verpackt:")
    print(packungen)
    print("Insgesamt wurden", verbrauchte_ballons, "Ballons verbraucht")

def simuliere_lösung(lvm):
    packungen = []
    verbrauchte_ballons = 0
    # solange noch nicht alle Speicherfächer leer sind
    while lvm.speicher.count(0) < 10:
        # leere Speicherfächer wieder auffüllen
        fülle_0er(lvm)

        # Kombination suchen, mit der im Ausgabefach exakt 20 Luftballons wären
        kombination = find_kombination(lvm, größe=20-lvm.ausgabe_fach)
        # falls eine kombination gefunden wurde,
        if kombination != None:
            for f in kombination:
                # werden die entsprechenden Speicherfächer geleert
                verbrauchte_ballons += fach(lvm, f)
            # und verpackt
            packungen.append(verpacken(lvm))
            # danach sollte der Code unten nicht ausgeführt werden,
            #  also gehen wir wieder an den Anfang der Schleife
            continue
```

```
        # Falls oben keine kombination gefunden wurde,
        #  leert das Programm das Speicherfach mit
        #  den wenigsten Luftballons in der Hoffnung, das später
        #  eine passende kombination gefunden wird.
        kleinste_mögliche_zahl = 1000
        kmz_index = None
        for i in range(0, 10):
            if 0 < lvm.speicher[i] < kleinste_mögliche_zahl:
                kmz_index = i
                kleinste_mögliche_zahl = lvm.speicher[i]
        verbrauchte_ballons += fach(lvm, kmz_index)
        # Falls danach im Ausgabefach bereits 20 Ballons sind,
        #  wird es geleert
        if lvm.ausgabe_fach >= 20:
            packungen.append(verpacken(lvm))
    return verbrauchte_ballons, packungen

[...]

lvm = lade_lvm_aus_datei(dateiname)

fülle_0er(lvm)
verbrauchte_ballons, packungen = simuliere_lösung(lvm)
zeige_endergebnis(lvm, verbrauchte_ballons, packungen)
```

Junioraufgabe 2 – LAMA

Lösungsidee/Umsetzung

Die Lösungsidee wird mit Python (tkinter) implementiert.

Ich hatte die Idee, zunächst in einem Konfigurationsfenster den Spieler nach der Breite und Höhe der Spielfläche zu fragen. Standardmäßig ist jeweils fünf eingestellt. Außerdem kann er die in der Aufgabenstellung beschriebene Spiel-Variante aktivieren. Als Maximalwerte nimmt mein Programm für die Höhe und Breite jeweils 10 LEDs an, wobei dieser Wert im Programm-Code beliebig geändert werden kann. Bei falschen Eingaben wird eine Fehlermeldung angezeigt.

Nachdem der Spieler das Spielfeld so konfiguriert hat wird das eigentliche Fenster mit allen LEDs erstellt. Die Größe des Fensters wird an die Anzahl der LEDs angepasst, sodass jede LED genau 50 mal 50 Pixel Platz hat. Allerdings kann der Spieler das Fenster auch beliebig vergrößern oder verkleinern. Die LEDs werden als gefärbte Buttons dargestellt (grau = aus, gelb = an). Leider ist dies auf Windows nicht möglich, auf Linux (und evtl. Mac) funktioniert es aber. Deswegen steht in jedem Button auch als Text, ob er eine angeschaltete oder ausgeschaltete LED darstellt.

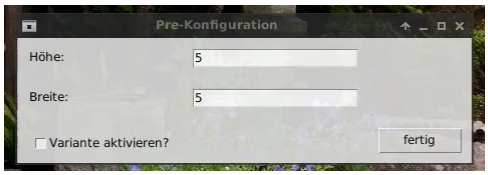
Das Drücken der Buttons wird mit der Funktion `led_gedrückt` verknüpft, welche dann anhand der x- und y-Koordinate des Buttons die LEDs (Buttons) um ihn herum umschaltet. Falls der Spieler das Spielfeld noch konfiguriert, wird nur der gedrückte Button umgeschaltet. Unter dem Spielfeld ist ein Button, mit welchem diese Konfiguration beendet werden kann. Die Variante wurde auch wie in der Aufgabenstellung beschrieben implementiert.

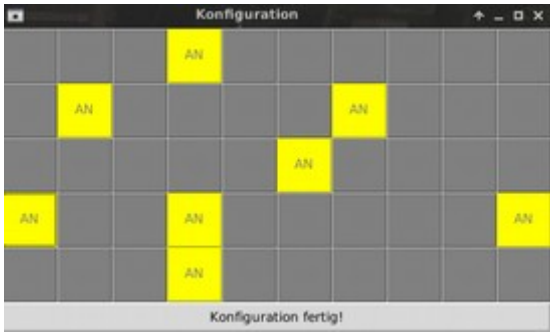
Nach jedem Zug wird überprüft, ob der Spieler gewonnen hat. Falls dies der Fall ist, werden die LEDs durch den Text „Du hast gewonnen!“ ersetzt. Danach kann man das Fenster ganz normal schließen.

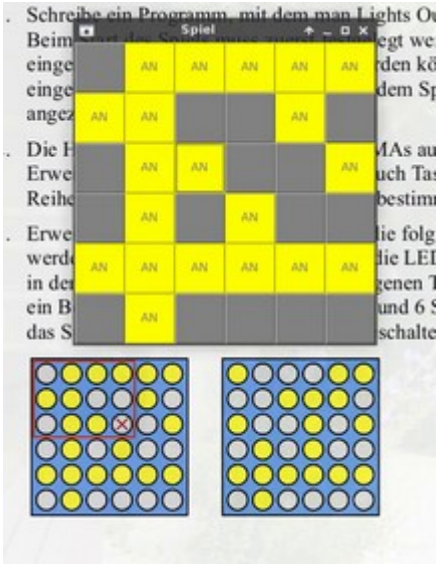
„Beispiele“

Hier ein paar (komprimierte) Screenshots des Spiels (aufgenommen unter Debian (Linux) und

XFCE 4.10). Im Verzeichnis „JuniorAufgabe2“ finden sie auch zwei Videos, die leider nicht in dieses Dokument eingefügt werden können.







Aufgabe 2 – Rhinozelfant

Lösungsidee

Meine erste Idee war, einfach für jeden Pixel des Bildes zu überprüfen, ob er Teil einer Schuppe ist. Also wird rekursiv überprüft, ob um den Pixel herum Pixel mit der gleichen Farbe vorliegen. Ab einer bestimmten Minimal-Größe (4) werden diese Pixelgruppen dann später eingefärbt.

Umsetzung

Die Lösungsidee wird in Python implementiert. Weil Python keine Bilder laden kann, wird das Bibliotheksmodul Pillow verwendet, das Bilder lesen und speichern kann.

Die Python-Funktion `pixel_gleicher_farbe` findet rekursiv alle Pixel mit der gleichen Farbe ausgehend von einem Pixel mit den Koordinaten x und y.

Im Programm wird zunächst das Bild geladen und die Pixel in die Variable `pixel_daten` eingelesen. Dann wird ein `set` erstellt, das alle bisher gefundenen Koordinaten enthalten soll, die zu einer Schuppe gehören (damit diese später nicht nochmal überprüft werden).

Danach wird in zwei `for`-Schleifen jeder Pixel besucht. Optional kann man über die Variablen `x_step` und `y_step` einstellen, um wie viele Pixel das Programm pro Schleifen-Durchgang nach rechts bzw. unten geht. Falls der Pixel bereits im oben erwähnten `set` vorhanden ist, wird er übersprungen. Dann werden zunächst alle Pixel, die die gleiche Farbe haben, in einer Liste gespeichert. Falls diese Liste größer als die Minimal-Größe ist, werden die Koordinaten der Pixel in das oben erwähnte `set` eingetragen und in einer Kopie des Original-Bilds eingefärbt.

Nachdem alle Pixel so abgearbeitet wurden, wird ausgegeben, wie viele Pixel insgesamt als Schuppen identifiziert wurden, wie das Verhältnis dieser Pixel zum gesamten Bild ist und ob das Bild vermutlich einen Rhinozelfanten enthält.

Dieser Vorgang wird dann für alle Bilder wiederholt.

Beispiele

Wir rufen das Python-Programm mit den verschiedenen Beispiel-Eingabedateien auf. Diese Dateien liegen im gleichen Ordner wie die Programmdatei. Leider werden noch nicht alle Ergebnisse innerhalb weniger Sekunden berechnet. Alle Ergebnisse werden auch in diesem Ordner unter dem Namen <originaler-Dateiname>.fertig.png gespeichert.

Die Bilder 1, 2, 4, 8 und 9 enthalten einen Rhinozelfanten.

```
$ python3 aufgabe2.py rhinozelfant1.png
Bild fertig
Insgesamt wurden 97455 Pixel gefunden
Verhältnis Schuppen/ganzes Bild: 17 %
Das Bild enthält wahrscheinlich einen Rhinozelfanten
$ python3 aufgabe2.py rhinozelfant2.png
Bild fertig
Insgesamt wurden 1351813 Pixel gefunden
Verhältnis Schuppen/ganzes Bild: 23 %
Das Bild enthält wahrscheinlich einen Rhinozelfanten
$ python3 aufgabe2.py rhinozelfant3.png
Bild fertig
Insgesamt wurden 105065 Pixel gefunden
Verhältnis Schuppen/ganzes Bild: 2 %
$ python3 aufgabe2.py rhinozelfant4.png
Bild fertig
Insgesamt wurden 1075094 Pixel gefunden
Verhältnis Schuppen/ganzes Bild: 18 %
Das Bild enthält wahrscheinlich einen Rhinozelfanten
$ python3 aufgabe2.py rhinozelfant5.png
Bild fertig
Insgesamt wurden 88867 Pixel gefunden
Verhältnis Schuppen/ganzes Bild: 1 %
$ python3 aufgabe2.py rhinozelfant6.png
Bild fertig
Insgesamt wurden 236384 Pixel gefunden
Verhältnis Schuppen/ganzes Bild: 4 %
$ python3 aufgabe2.py rhinozelfant7.png
Bild fertig
```

```
Insgesamt wurden 217996 Pixel gefunden
Verhältnis Schuppen/ganzes Bild: 4 %
$ python3 aufgabe2.py rhinozelfant8.png
Bild fertig
Insgesamt wurden 508180 Pixel gefunden
Verhältnis Schuppen/ganzes Bild: 8 %
Das Bild enthält wahrscheinlich einen Rhinozelfanten
$ python3 aufgabe2.py rhinozelfant9.png
Bild fertig
Insgesamt wurden 676510 Pixel gefunden
Verhältnis Schuppen/ganzes Bild: 11 %
Das Bild enthält wahrscheinlich einen Rhinozelfanten
```

Quelltext (gekürzt)

aufgabe2.py

```
from PIL import Image
import sys

interaktiv = False

# standardmäßig nur 1000
sys.setrecursionlimit(10000)

# wie viele Pixel das Programm pro Schritt geht
y_step = 2
x_step = 2

# diese Funktion findet Pixel, die die gleiche Farbe haben
# ausgehend von dem Punkt (x, y)
def pixel_gleicher_farbe(bild, x, y, gefundene_pixel=set()):
    # falls der Pixel bereits in einem bekannten Fund ist,
    # ist es nicht nötig weiterzusuchen
    if (x, y) in schuppen_koordinaten:
        return set()
    farbe = pixel_daten[x][y]
    gefundene_pixel.add((x, y))
    # Nun wird überprüft ob der Pixel eins weiter rechts
    # - nicht am Rand ist
    # - dieselbe Farbe hat wie der erste
    # - noch nicht gefunden wurde
    if x < bild.size[0]-1 and (x+1, y) not in gefundene_pixel:
        if pixel_daten[x+1][y] == farbe:
            gefundene_pixel.add((x+1, y))
            # Nun wird ausgehend vom Pixel eins weiter rechts
            # weiter gesucht
            pixel_gleicher_farbe(bild, x+1, y, gefundene_pixel=gefundene_pixel)

    # dieselbe Methode wird auch in die
    # drei anderen Richtungen angewandt
    [...]

    return gefundene_pixel

# diese Funktion lädt alle Pixel eines Bildes
# in einen (zwei-dimensionalen) Tupel
```

```
def lade_pixel(bild):
    [...]
    return tuple(pixel_daten)

try:
    bild = Image.open(sys.argv[1])
    pixel_daten = lade_pixel(bild)
    neues_bild = bild.copy()
    # in schuppen_koordinaten werden alle bisher
    # gefundenen Pixel gespeichert, um schnell
    # bestimmen zu können, ob ein Pixel bereits
    # gefunden wurde
    schuppen_koordinaten = set()

    mindestgröße = 4

    for y in range(0, bild.size[1], y_step):
        for x in range(0, bild.size[0], x_step):
            if (x, y) in schuppen_koordinaten:
                continue
            gleiche = pixel_gleicher_farbe(bild, x, y, gefundene_pixel=set())
            if len(gleiche) >= mindestgröße:
                schuppen_koordinaten |= gleiche
                for koordinate in gleiche:
                    neues_bild.putpixel(koordinate, (255, 255, 255))
                    #print(".", end="")
    print("Bild fertig")
    print("Insgesamt wurden", len(schuppen_koordinaten), "Pixel gefunden")
    schuppen_prozent= round(len(schuppen_koordinaten)/
(bild.size[0]*bild.size[1])*100)
    print("Verhältnis Schuppen/ganzes Bild:", schuppen_prozent, "%")
    if schuppen_prozent >= 8:
        print("Das Bild enthält wahrscheinlich einen Rhinozelfanten")
    if not interaktiv or input("speichern?")[0].lower() != "n":
        neues_bild.save(sys.argv[1]+".fertig.png")
except KeyboardInterrupt:
    pass
```

Lösungsidee 2

Anscheinend ist jede Hautschuppe des Rhinozelfanten exakt 2x2 Pixel groß. Also kam ich auf die Idee, einfach nur nach 2x2 Pixel-Gruppen zu suchen. Weil der Rhinozelfant meist einen sehr großen Teil des Bildes einnimmt, habe ich mich entschieden, zunächst nur etwa 100 gleichmäßig verteilte Stichproben pro Bild zu machen um danach, falls die Stichprobe erfolgreich war, um sie herum nach weiteren Hautschuppen zu suchen.

Umsetzung 2

Diese Lösungsidee wird in Rust implementiert, weil Python einfach zu langsam ist. Weil Rust keine Bilder laden kann, wird die „Crate“ `image` verwendet, die Bilder lesen und speichern kann.

Die Funktion `ist_schuppe` gibt anhand einer x- und y-Koordinate an, ob die Pixelgruppe [(x, y), (x+1, y), (x+1, y+1), (x, y+1)] eine Schuppe ist (also dieselbe Farbe hat).

Im Programm wird zunächst das Bild geladen und ein Vektor `erfolgreiche_proben` erstellt, der erfolgreiche Stichproben enthalten soll. Anschließend werden die Variablen `x_step` und `y_step` berechnet, die angeben, wie weit die Stichproben voneinander entfernt sein sollen. Falls die Bildbreite kleiner 768 Pixel ist, wird `x_step` verkleinert, damit auch im ersten Beispiel der Rhinozelfant entdeckt wird. Deswegen sollte man eigentlich beide Methoden auf die Bilder anwenden: eine zum schnellen Überprüfen, die vielleicht manche Rhinozelfanten übersieht und die weiter oben beschriebene Methode, um wirklich alle zu finden.

Danach werden in einer Schleife alle Orte besucht, an denen eine Stichprobe gemacht werden soll. Dann wird für jeden der Pixel in (x, y), (x+1, y), (x+1, y+1), (x, y+1) überprüft, ob er die obere linke Ecke einer Schuppe ist. Falls dies der Fall ist, wird der Pixel an den Vektor `erfolgreiche_proben` angehängt.

Anschließend wird ausgegeben, wie viele Stichproben erfolgreich waren.

Dann werden die zwei (Hash-)Sets `gefundene_schuppen` und `besuchte_punkte` erstellt. `gefundene_schuppen` soll bisher gefundene Schuppen zur späteren Einfärbung enthalten. In `besuchte_punkte` werden alle bisher besuchten Punkte gespeichert, damit das Programm nicht zweimal denselben Punkt bearbeitet.

Danach wird in einer Schleife, die wiederholt wird, bis der Vektor `erfolgreiche_proben` leer ist, immer zunächst eine Koordinate aus diesem Vektor entfernt. Dieser Punkt wird in `besuchte_punkte` eingefügt. Nun wird ausgehend von diesem Punkt überprüft, ob die 2x2 Pixelgruppen um ihn herum auch Schuppen sind. Falls dies der Fall ist, werden sie an den Vektor `gefundene_schuppen` angehängt. Falls sie auch noch nicht in `besuchte_punkte` sind, werden sie auch an den Vektor `erfolgreiche_proben` angehängt, damit sie später auch in dieser Schleife abgearbeitet werden.

Nachdem diese Schleife beendet ist, wird ausgegeben, wie viele Punkte besucht und Schuppen gefunden wurden.

Danach werden in einer Kopie des Original-Bildes alle Pixel eingefärbt. Schließlich wird diese Kopie unter dem Dateinamen <originaler-dateiname>.fertig.png gespeichert.

Beispiele 2

Wir rufen das Rust-Programm mit cargo, dem Packet-Manager von Rust auf. Nun werden auch alle Ergebnisse innerhalb weniger Sekunden berechnet. Rechts sind jeweils kleinere Version der Bilder, die einen Rhinozelfanten enthalten. Die Bilder sind in der Einsendung aus Platzgründen als jpg-Bilder gespeichert.

```
$ cargo run --release rhinozelfant1.png
Erfolgreiche Stichproben: 80
Gefundene Schuppen: 19898
Besuchte Punkte: 19910
$ cargo run --release rhinozelfant2.png
Erfolgreiche Stichproben: 21
Gefundene Schuppen: 332907
Besuchte Punkte: 332907
$ cargo run --release rhinozelfant3.png
Erfolgreiche Stichproben: 0
Gefundene Schuppen: 0
```

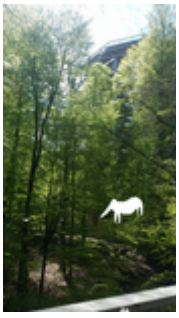


Bild 1

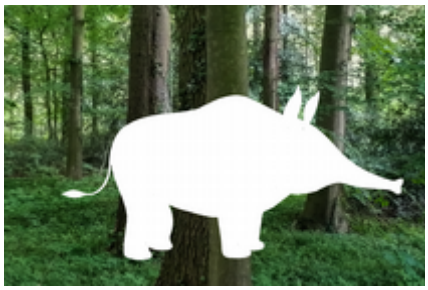


Bild 2

```
Besuchte Punkte: 0
$ cargo run --release rhinozelfant4.png
Erfolgreiche Stichproben: 16
Gefundene Schuppen: 254751
Besuchte Punkte: 254751
$ cargo run --release rhinozelfant5.png
Erfolgreiche Stichproben: 0
Gefundene Schuppen: 0
Besuchte Punkte: 0
$ cargo run --release rhinozelfant6.png
Erfolgreiche Stichproben: 3
Gefundene Schuppen: 0
Besuchte Punkte: 3
$ cargo run --release rhinozelfant7.png
Erfolgreiche Stichproben: 13
Gefundene Schuppen: 11483
Besuchte Punkte: 11484
$ cargo run --release rhinozelfant8.png
Erfolgreiche Stichproben: 11
Gefundene Schuppen: 113732
Besuchte Punkte: 113734
$ cargo run --release rhinozelfant9.png
Erfolgreiche Stichproben: 9
Gefundene Schuppen: 150080
Besuchte Punkte: 150080
```

Bild 4

Bild 8

Bild 9

Quelltext 2 (gekürzt)

Cargo.toml

```
[package]
[...]
[dependencies]
image = "0.10"
```

src/main.rs

```
#![feature(step_by)]

// zum Bilder einlesen
extern crate image;
use image::*;

[...]
```

```
struct Punkt {
	x: u32,
	y: u32
}

fn ist_schuppe(bild: &DynamicImage, punkt: &Punkt) -> bool {
	let x = punkt.x;
	let y = punkt.y;
	let farbe = bild.get_pixel(x, y);
	return farbe == bild.get_pixel(x+1, y)
	&& farbe == bild.get_pixel(x, y+1)
	&& farbe == bild.get_pixel(x+1, y+1)
}

fn main() {
	[...]
	// Kommandozeilenargument lesen
	if let Some(arg1) = env::args().nth(1) {
		// datei öffnen
		let bild = image::open(&arg1);
		[...]

		// hier werden mögliche Schuppen zur
		// weiteren Verarbeitung gespeichert
		let mut erfolgreiche_proben = Vec::new();

		[...]
		macro_rules! check {
			($punkt:expr) => (
				if ist_schuppe(&bild, &$punkt) {
					erfolgreiche_proben.push($punkt);
				}
			)
		}

		for y in (0..bild.height()).step_by(y_step) {
			for x in (0..bild.width()).step_by(x_step) {
				check!(Punkt { x: x, y: y });
				check!(Punkt { x: x+1, y: y });
				check!(Punkt { x: x+1, y: y+1 });
				check!(Punkt { x: x, y: y+1 });
			}
		}
		[...]
		let mut gefundene_schuppen = HashSet::new();
		// insg. besuchte Punkte
		let mut besuchte_punkte = HashSet::new();

		macro_rules! check {
				($punkt:expr) => (
					if ist_schuppe(&bild, &$punkt) {
						gefundene_schuppen.insert($punkt);
						if !besuchte_punkte.contains(&$punkt) {
							erfolgreiche_proben.push($punkt);
						}
					}
				)
			}

		// an erfolgreice_proben werden in der Schleife
		// noch mehr mögliche Schuppen angefügt
		while !erfolgreiche_proben.is_empty() {
			let pos = erfolgreiche_proben.pop().unwrap();
			besuchte_punkte.insert(pos);
			let x = pos.x;
			let y = pos.y;

			// um die Schuppe herum werden ebenfalls
			// 2x2 pixelgruppen überprüft
			// (falls diese noch auf dem Bild sind)
			if x+3 < bild.width() {
```

```
                        check!(Punkt { x: x+2, y: y });
                  }
                  [...]
            }

            println!("Gefundene Schuppen: {}", gefundene_schuppen.len());
            println!("Besuchte Punkte: {}", besuchte_punkte.len());

            let mut neues_bild = bild.clone();

            // gefunde schuppen einfärben
            [...]

            // speichern
            match save_buffer(
                  arg1+".fertig.png",
                  [...]
            ) {
                  Ok(_) => (),
                  Err(e) => println!("Datei konnte leider nicht gespeichert
werden:\n {}", e)
            }
      } else {
            println!("Bitte so aufrufen: cargo run --release -- <dateiname>");
      }
}
```

Aufgabe 3 – Rotation

Lösungsidee

Ich hatte die Idee mithilfe einer Breitensuche alle Entscheidungswege zu untersuchen. Einen Entscheidungsweg stelle ich so dar: [US, GUS] bzw. [↻, ↺]. US steht für „UhrzeigerSinn“, GUS für „Gegen den UhrzeigerSinn“.

Da mir schnell klar wurde, dass es mit dieser Methode unmöglich ist zu bestimmen, ob ein Puzzle nicht lösbar ist, entschied ich mich stattdessen dazu, auch alle möglichen Raster-Zustände zu finden. Wenn alle möglichen Zustände ein nicht gelöstes Puzzle darstellen, ist es unlösbar.

Um das zu erreichen werden während der Suche alle Raster-Zustände abgespeichert. Falls mein Programm nochmal denselben Zustand erreicht, wird dieser „Ast“ des Entscheidungsbaumes aufgegeben, weil er ja schon einmal erreicht und weitergeführt wurde. Dies implementiere ich durch zwei Listen/Vektoren: `base` und `nächste_schicht`. In `base` werden alle Entscheidungswege gespeichert, welche dann später die Basis für die nächste Ebene des Entscheidungsbaumes darstellen. In `nächste_schicht` liegen alle Entscheidungswege, die aus base gebildet wurden.

In einer unendlichen Schleife wird nun zunächst `nächste_schicht` aus `base` berechnet. Dazu wird jeder Entscheidungsweg in base verdoppelt, damit diese danach jeweils um eine Drehung im Uhrzeigersinn und um eine Drehung gegen den Uhrzeigersinn erweitert werden. Beispiel: [[US]] → [[US, US], [US, GUS]]. Danach wird `base` durch einen neuen Vektor ersetzt. Nun werden die Entscheidungswege in `nächste_schicht` auf das Start-Puzzle (also das Puzzle, das aus der Datei ausgelesen wurde) angewandt. Falls der entstandene Zustand schon einmal in der Suche erreicht wurde, wird der Entscheidungsweg nicht an `base` angefügt, ansonsten schon. Falls der Zustand ein gelöstes Puzzle darstellt, wird der Lösungsweg zurückgegeben.

Umsetzung

Der oben beschriebene Such-Algorithmus wird in der Funktion `optimierte_breiten_suche` implementiert.

Um das Puzzle zunächst aus einer Datei zu laden, wird davor die Funktion `lade_puzzle` aufgerufen. Leider kann die Funktion nur Puzzle lesen, deren Stäbe ohne Unterbrechung aufsteigend von null an nummeriert sind. Ein Puzzle wird als Vektor von Vektoren, die Zahlen enthalten, dargestellt. Die Funktion überprüft auch, ob das Puzzle richtig aufgebaut ist, also z. B. die Anzahl der Ausgänge.

Innerhalb der Suche werden die Entscheidungswege dann als Vektoren eines Aufzählungstypen (DrehRichtung) verarbeitet. Die Funktion `wende_entscheidungen_an` wendet solch einen Weg dann auf ein Puzzle an (mithilfe eines Caches wird vermieden, dass bereits berechnete Entscheidungswege-Teile nochmal berechnet werden).

Dazu ruft diese Funktion die Funktionen `rotate` und `wende_gravitation_an` auf. Beide modifizieren das Puzzle ohne es zu kopieren. In der zweiten Funktion werden die Stäbe von unten nach oben jeweils nur einmal (nach unten) bewegt.

Nach jedem verarbeiteten Entscheidungsweg wird mit der Funktion `puzzle_gelöst` überprüft, ob das berechnete Puzzle gelöst ist. Falls in der untersten Zeile des Rasters mindestens ein Teil eines Stabes liegt, ist dieser Stab wohl durch (in) den Ausgang gefallen.

Schließlich werden wie oben beschrieben die Entscheidungswege der nächsten Ebene mit der Funktion `erstelle_nächste_schicht` erzeugt.

Zusätzlich schrieb ich noch ein paar Tests und Benchmarks (Aufruf jeweils mit „cargo test“ und „cargo bench“).

Beispiele

Das Rust-Programm wird von der Kommandozeile aufgerufen. Alle Ergebnisse (ihrer Beispiele) werden innerhalb weniger Sekunden berechnet. Zunächst sollte man das Programm mit cargo kompilieren. Das Programm wird im Verzeichnis „target/debug/“ gespeichert. „[...]“ steht für unwichtige Programm-Ausgaben. Für Windows habe ich das Programm bereits kompiliert, statt „target/debug/Agabe3uf“ könnte man auf Windows also „Aufgabe3_windows64bit.exe“ benutzen. Auf der rechten Seite ist das Puzzle immer auch in Form eines Bildes dargestellt.

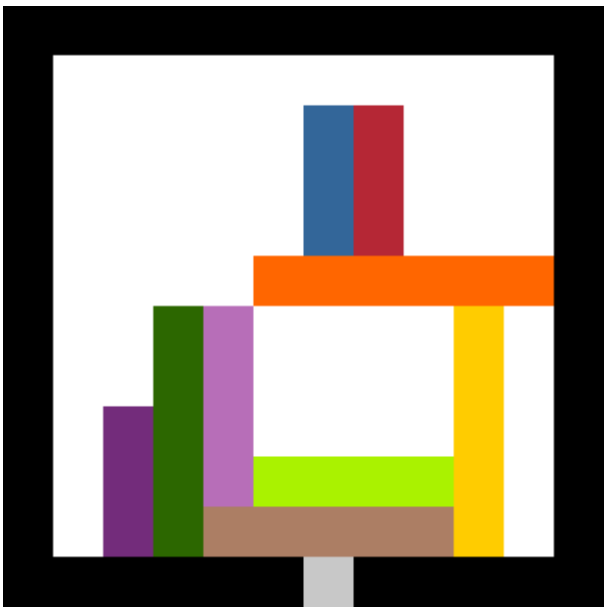
```
$ cargo rustc -- -C opt-level=3
[...]
```

```
$ target/debug/Aufgabe3 rotation1_03.txt
```

```
-3 -3 -3 -3 -3 -3 -3 -3
-3 -2 -2 -2 -2 -2 00 -3
-3 -2 -2 -2 -2 -2 00 -3
-3 01 01 02 02 02 02 -3
-3 03 03 -2 -2 -2 04 -3
-3 05 05 -2 -2 -2 04 -3
-3 06 06 06 -2 -2 04 -3
-3 -3 -3 -1 -3 -3 -3 -3
```

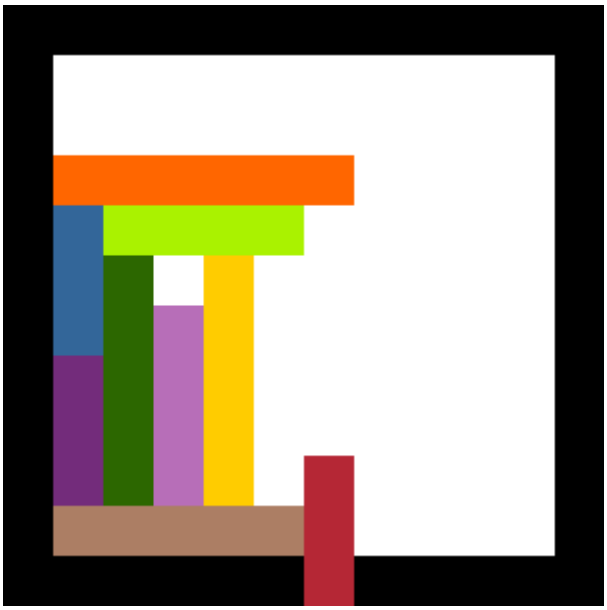
```
[...] 0 sekunden
Lösung: [↺, ↺, ↺, ↺, ↻, ↺]
-3 -3 -3 -3 -3 -3 -3 -3
-3 -2 -2 02 02 02 02 -3
-3 -2 -2 -2 -2 -2 04 -3
-3 -2 -2 -2 01 01 04 -3
-3 -2 -2 -2 03 03 04 -3
-3 -2 -2 -2 -2 05 05 -3
-3 -2 -2 00 06 06 06 -3
-3 -3 -3 00 -3 -3 -3 -3
```

```
$ target/debug/Aufgabe3 rotation2_03.txt
-3 -3 -3 -3 -3 -3 -3 -3 -3 -3 -3 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 00 01 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 00 01 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 00 01 -2 -2 -2 -3
-3 -2 -2 -2 -2 02 02 02 02 02 02 -3
-3 -2 -2 03 04 -2 -2 -2 -2 05 -2 -3
-3 -2 -2 03 04 -2 -2 -2 -2 05 -2 -3
-3 -2 06 03 04 -2 -2 -2 -2 05 -2 -3
-3 -2 06 03 04 07 07 07 07 05 -2 -3
-3 -2 06 03 08 08 08 08 08 05 -2 -3
-3 -3 -3 -3 -3 -3 -1 -3 -3 -3 -3 -3
```

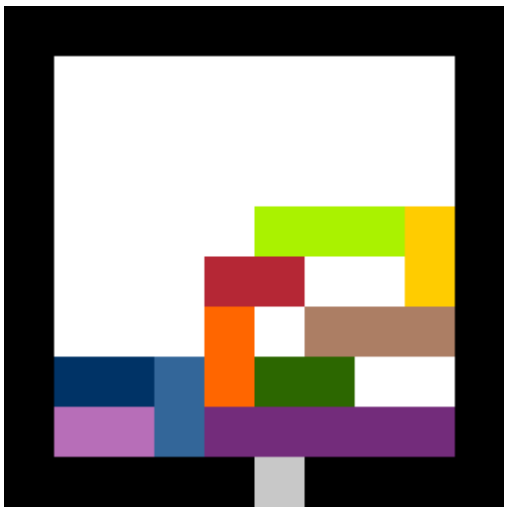


```
[...] 0 sekunden
Lösung: [↻, ↻, ↻, ↻, ↺, ↻, ↻, ↺, ↺, ↺, ↺, ↺, ↺, ↺, ↻, ↻, ↻, ↻, ↻,
↻, ↺, ↻]
```

```
-3 -3 -3 -3 -3 -3 -3 -3 -3 -3 -3 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 02 02 02 02 02 02 -2 -2 -2 -2 -3
-3 00 07 07 07 07 -2 -2 -2 -2 -2 -3
-3 00 03 -2 05 -2 -2 -2 -2 -2 -2 -3
-3 00 03 04 05 -2 -2 -2 -2 -2 -2 -3
-3 06 03 04 05 -2 -2 -2 -2 -2 -2 -3
-3 06 03 04 05 -2 -2 -2 -2 -2 -2 -3
-3 06 03 04 05 -2 01 -2 -2 -2 -2 -3
-3 08 08 08 08 08 01 -2 -2 -2 -2 -3
-3 -3 -3 -3 -3 -3 01 -3 -3 -3 -3 -3
```



```
$ target/debug/Aufgabe3 rotation3_03.txt
-3 -3 -3 -3 -3 -3 -3 -3 -3 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 07 07 07 05 -3
-3 -2 -2 -2 01 01 -2 -2 05 -3
-3 -2 -2 -2 02 -2 08 08 08 -3
-3 09 09 00 02 03 03 -2 -2 -3
-3 04 04 00 06 06 06 06 06 -3
-3 -3 -3 -3 -3 -1 -3 -3 -3 -3
```

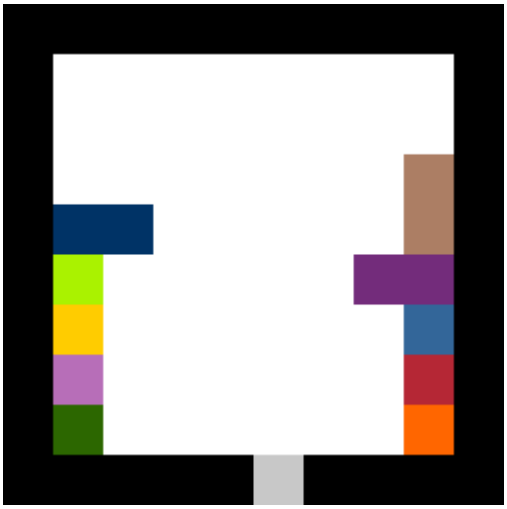


```
[...] 4 sekunden
Lösung: [↺, ↻, ↻, ↻, ↻, ↺, ↺, ↺, ↻, ↻, ↻, ↺, ↺, ↺, ↺, ↺, ↺, ↺, ↻,
↻, ↺, ↺, ↻, ↻, ↻, ↻, ↻, ↻, ↻, ↺, ↻, ↻, ↻, ↻, ↻, ↻, ↻, ↻, ↻, ↻,
↺, ↺, ↺, ↻, ↻, ↻, ↻, ↻, ↺, ↺, ↺, ↺, ↺, ↻, ↻, ↻, ↻, ↺, ↺, ↻, ↻, ↻,
↻, ↻, ↻, ↻, ↺, ↻, ↺, ↺, ↺, ↺, ↺, ↺, ↺, ↺, ↺, ↺, ↺, ↻, ↺, ↺, ↻, ↻,
↻, ↻, ↻, ↺, ↻]
-3 -3 -3 -3 -3 -3 -3 -3 -3 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 07 07 07 09 09 -2 -2 -2 -3
-3 06 06 06 06 06 08 08 08 -3
```

```
-3 04 04 -2 -2 -2 02 00 -2 -3
-3 01 01 03 03 05 02 00 -2 -3
-3 -3 -3 -3 -3 05 -3 -3 -3 -3
```

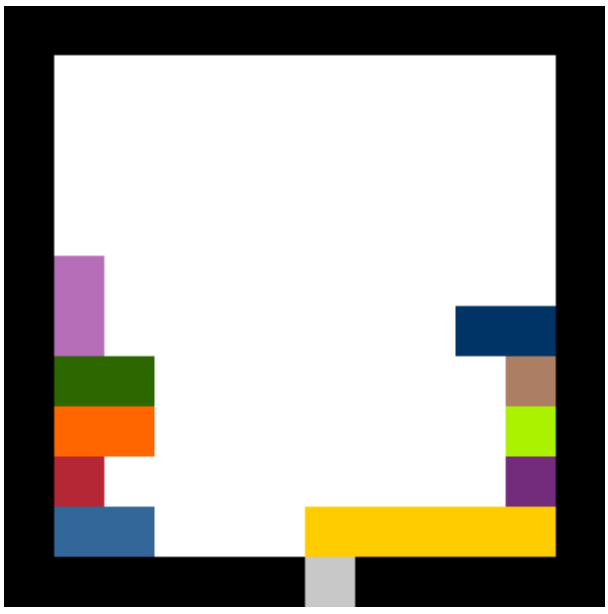
rotation4_n.txt ist ein kleiner Härtetest für mein Programm, weil sehr viele Würfel (Stäbe mit einer Länge von eins) vorhanden sind.

```
$ target/debug/Aufgabe3 rotation4_n.txt
-3 -3 -3 -3 -3 -3 -3 -3 -3 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 -2 -2 08 -3
-3 09 09 -2 -2 -2 -2 -2 08 -3
-3 07 -2 -2 -2 -2 -2 06 06 -3
-3 05 -2 -2 -2 -2 -2 -2 00 -3
-3 04 -2 -2 -2 -2 -2 -2 01 -3
-3 03 -2 -2 -2 -2 -2 -2 02 -3
-3 -3 -3 -3 -3 -1 -3 -3 -3 -3
[...] 28 sekunden
keine lösung gefunden!
```



rotation6_n.txt ist ein etwas einfacherer Härtetest, wobei mein Programm unter anderem über 200 Elemente lange Entscheidungswege überprüft.

```
$ target/debug/Aufgabe3 rotation6_n.txt
-3 -3 -3 -3 -3 -3 -3 -3 -3 -3 -3 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 04 -2 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 04 -2 -2 -2 -2 -2 -2 -2 09 09 -3
-3 03 03 -2 -2 -2 -2 -2 -2 -2 08 -3
-3 02 02 -2 -2 -2 -2 -2 -2 -2 07 -3
-3 01 -2 -2 -2 -2 -2 -2 -2 -2 06 -3
-3 00 00 -2 -2 -2 05 05 05 05 05 -3
-3 -3 -3 -3 -3 -3 -1 -3 -3 -3 -3 -3
[...] 18 sekunden
keine lösung gefunden!
```



rotation8_n.txt ist ein ähnlicher Test, nur mit mehr (und längeren) Stäbchen. Leider kann mein Programm für dieses Puzzle nicht (einigermaßen schnell) beweisen, dass es keine Lösung gibt.

```
$ target/debug/Aufgabe3 rotation8_n.txt
-3 -3 -3 -3 -3 -3 -3 -3 -3 -3 -3 -3 -3 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -3
-3 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 04 -3
-3 08 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 04 -3
-3 08 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 03 -3
-3 08 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 03 -3
-3 09 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 01 -3
-3 09 -2 -2 -2 -2 -2 -2 -2 -2 -2 02 01 -3
-3 09 -2 11 11 -2 -2 -2 -2 -2 05 02 00 -3
-3 09 10 10 -2 -2 -2 -2 -2 -2 05 02 00 -3
-3 06 06 07 07 -2 12 12 12 12 12 12 12 -3
-3 -3 -3 -3 -3 -3 -3 -1 -3 -3 -3 -3 -3 -3
103 tiefe [...] 160 sekunden
```

rotation9_n.txt demonstriert, dass das Programm auch ungerade Seitenlängen verarbeiten kann (die Beispiele auf der Website haben alle gerade Seitenlängen).

```
$ target/debug/Aufgabe3 rotation9_n.txt
-3 -3 -3 -3 -3
-3 -2 -2 -2 -3
-3 01 -2 00 -3
-3 01 -2 00 -3
-3 -3 -1 -3 -3
[...] 0 sekunden
Lösung: [↻, ↺]
-3 -3 -3 -3 -3
-3 -2 -2 -2 -3
-3 -2 -2 00 -3
-3 -2 01 00 -3
-3 -3 01 -3 -3
```

rotation10_n.txt demonstriert, dass das Programm auch Ausgänge an den Seiten verarbeiten kann (die Beispiele auf der Website haben alle den Ausgang unten).

```
$ target/debug/Aufgabe3 rotation10_n.txt
```

```
-3 -3 -3 -3 -3 -3 -3
-3 -2 -2 03 03 03 -3
-1 -2 -2 -2 -2 02 -3
-3 01 -2 -2 -2 02 -3
-3 01 -2 -2 -2 02 -3
-3 01 00 00 -2 02 -3
-3 -3 -3 -3 -3 -3 -3
[...] 0 sekunden
Lösung: [↺, ↻, ↺]
```

```
-3 -3 -3 -3 -3 -3 -3
-3 -2 -2 -2 -2 -2 -3
-3 -2 02 02 02 02 -3
-3 -2 -2 -2 -2 00 -3
-3 -2 03 -2 -2 00 -3
-3 -2 03 01 01 01 -3
-3 -3 03 -3 -3 -3 -3
```

rotation11_n.txt demonstriert, dass das Programm auch mit mehreren Ausgänge funktioniert (laut der Aufgabenstellung ist aber nur einer erlaubt). Dieses Verhalten kann man im Programm mit einer Konstante (maximale Ausgänge) ändern. Außerdem wird vor der Lösungssuche noch einmal die Gravitation angewandt, falls der Puzzle-Ersteller dies vergessen hatte..

```
$ target/debug/Aufgabe3 rotation11_n.txt
-3 -3 -3 -3 -3 -3 -3
-3 -2 -2 -2 -2 -2 -3
-1 -2 -2 02 02 02 -1
-3 03 -2 -2 -2 01 -3
-3 03 -2 -2 -2 01 -3
-3 03 00 00 -2 01 -3
-3 -3 -3 -3 -3 -3 -3
Lösung: [↻]
```

```
-3 -3 -3 -3 -1 -3 -3
-3 -2 -2 -2 -2 -2 -3
-3 03 03 03 -2 -2 -3
-3 00 -2 -2 -2 -2 -3
-3 00 -2 -2 02 -2 -3
-3 01 01 01 02 -2 -3
-3 -3 -3 -3 02 -3 -3
```

rotation12_n.txt demonstriert, dass mein Programm auch mit gar keinen Ausgängen funktioniert. Etwas sinnlos, aber warum nicht? Dieses Verhalten kann man auch im Programm mit einer Konstante (minimale Ausgänge) verändern.

```
$ target/debug/Aufgabe3 rotation12_n.txt
-3 -3 -3 -3 -3 -3 -3
-3 08 -2 05 05 05 -3
-3 06 -2 -2 -2 07 -3
-3 04 09 -2 -2 03 -3
-3 04 01 -2 -2 03 -3
-3 04 02 02 00 03 -3
-3 -3 -3 -3 -3 -3 -3
[...] 6 sekunden
keine lösung gefunden!
```

rotation13_n.txt demonstriert, dass das Programm auch mit größeren Ausgängen funktioniert. Dies kann man auch einfach im Programm mit einer Konstante (maximale Ausgänge) ändern.

```
$ target/debug/Aufgabe3 rotation13_n.txt
-3 -3 -3 -3 -3 -3 -3
-3 -2 -2 -2 05 05 -3
-3 -2 -2 -2 04 04 -3
-3 -2 -2 -2 03 03 -3
-3 -2 -2 -2 02 02 -3
-3 00 -2 -2 01 01 -3
-3 -3 -1 -1 -3 -3 -3
[...] 0 sekunden
Lösung: [ʊ, ʊ]
-3 -3 -3 -3 -3 -3 -3
-3 -2 -2 -2 05 05 -3
-3 -2 -2 -2 04 04 -3
-3 -2 -2 -2 03 03 -3
-3 -2 -2 -2 02 02 -3
-3 -2 -2 -2 01 01 -3
-3 -3 -1 00 -3 -3 -3
```

rotation14_c.txt und rotation15_c.txt demonstrieren, dass auch falsch geschriebene Dateien erkannt werden. Die Puzzle sind rechts dargestellt.

```
$ target/debug/Aufgabe3 rotation14_c.txt

Fehler beim Einlesen: "Stäbchen ergeben keinen Sinn"

$ target/debug/Aufgabe3 rotation15_c.txt

Fehler beim Einlesen: "Rahmen des Puzzles ist nicht
korrekt aufgebaut"
```

Quelltext (gekürzt)

src/main.rs

```
[...]
/* Generelle Informationen
 * - die Makros set und get setten und getten
 *    jeweils einen Punkt im Raster
 * - farbige Ausgabe des Puzzles nur mit Linux und evtl. Mac
 * - Groß- und Kleinschreibung (in kommentaren) eher weniger
 * - index eines Stabes == die Zahl in der Datei (0-9 und A-Z bzw.
0-36)
 */

const RAHMEN: i8 = -3;
const LEER: i8 = -2;
const AUSGANG: i8 = -1;

const MINIMALE_AUSGÄNGE: usize = 0; // problemlos veränderbar
(Aufgabenstellung -> 1)
const MAXIMALE_AUSGÄNGE: usize = 1000; // problemlos veränderbar
(Aufgabenstellung -> 1)

[...]

#[derive(Clone, Eq, Hash, PartialEq)]
pub struct Puzzle {
     raster: Vec<Vec<i8>>
}

// meistens nur als Vec<Punkt> == stab verwendet
#[derive(Clone, Copy, Debug, PartialEq)]
pub struct Punkt {
     x: usize,
     y: usize
}

[...]

// lädt ein Puzzle aus einer Datei und kann Fehler zurückgeben
pub fn lade_puzzle(dateiname: &str) -> Result<Puzzle, String> {
     [...]
}
```

```
fn verifiziere_rahmen(raster: &[Vec<i8>]) -> bool {
     [...]
}

// überprüft, dass alle stäbchen eindeutig vertikal
// oder horizontal sind
fn verifiziere_stäbchen(stäbchen: &[Vec<Punkt>]) -> bool {
     [...]
}
// überprüft, ob ein Stab eindeutig vertikal
// oder horizontal ist
fn verifiziere_stab(stab: &[Punkt]) -> bool {
     [...]
}

pub fn drehe(mut puzzle: Puzzle, dir: &DrehRichtung) -> Puzzle {
     [...]
          // mit 'x' markierte Punkte werden unten in der Schleife
          // unten bearbeitet
          /* - - - - - -
           * - x x     -
           * - x x     -
           * -         -
           * -         -
           * - - -   - - */
     [...]
                    // Ergebnis: (nur das Innere des Rahmens)
                    // Punkt B erhält den Wert von A (ol)
                    // C von B (or)
                    // D von C (ur)
                    // A von D (ul)
                    /* - A - -
                     * - - - B
                     * D - - -
                     * - - C - */
     [...]
     puzzle
}

// findet alle Stäbe in einem Raster bzw. Puzzle
pub fn finde_stäbchen(raster: &[Vec<i8>]) -> Vec<Vec<Punkt>> {
     [...]
}

fn get_stäbchen_richtung(stab: &[Punkt]) -> StabRichtung {
     // falls der Stab kein Würfel ist und die x-Koordinaten der
     // ersten beiden elemente gleich sind,
     if stab.len() > 1 && stab[0].x == stab[1].x {
          // ist er vertikal
          StabRichtung::Vertikal
     } else {
          StabRichtung::Horizontal
     }
```

```
}
[...]

pub fn wende_gravitation_an(mut puzzle: Puzzle) -> Puzzle {
     [...]
     // erst alle Stäbchen finden
     let stäbchen = finde_stäbchen(&puzzle.raster);

     [...]
     let mut sortiert = unterste_teile_der_stäbe.clone();
     sortiert.sort_by_key(|&x| x.y);
     sortiert.reverse();

     // diese Schleife beginnt also mit dem untersten Stab
     for punkt_aus_stab in sortiert {
          [...]
          // die Schritte, die der Stab nach unten fallen soll
          let mut schritte = 0;

          if richtung == StabRichtung::Vertikal {
               let unterster_punkt = punkt_aus_stab;
               // hier wird berechnet, wie viele der Punkte bis zum
Rahmen (einschließlich dem Rahmen)
               // hintereinander leer sind
               [...]
          } else {
               // die y-Koordinate der horizontalen Stäbe ist für
jeden teil gleich
               let y_level = stab[0].y;

               // hier wird berechnet, wie viele der Punkte bis zum
Rahmen (einschließlich dem Rahmen)
               // hintereinander leer sind
               [...]
          }

          // falls schritte == 0 ist, passiert nichts
          for offset in 0..schritte {
               // erste vorherige Position mit LEER ersetzen
               for p in stab {
                    set!(p.x, p.y+offset, LEER);
               }
               // dann neue Position mit stab_index füllen
               for p in stab {
                    set!(p.x, p.y+offset+1, stab_index as i8);
               }
          }
     }
     puzzle
}

// gibt an, ob das Puzzle gelöst ist
fn puzzle_gelöst(puzzle: &Puzzle) -> bool {
```

```
     puzzle.raster
     // unterer Teil des Rahmens
     .last().expect("Code 4-un").iter()
     .fold(
          false, |acc, &x|
          // falls dieses Element zu einem Stab gehört,
          //  wird acc auf true gesetzt
          if x >= 0 { true }
          else { acc })
}

// wendet einen Entscheidungsweg an (mithilfe eines caches)
fn wende_entscheidungen_an(puzzle: Puzzle, ew: &[DrehRichtung],
cache: &mut HashMap<Vec<DrehRichtung>, Puzzle>) -> Puzzle {
     // ew == Entscheidungsweg
     if cache.contains_key(ew) {
          cache.get(ew).expect("Code 5-un").clone()
     } else if ew.len() == 1 { // z.b. [US] oder [GUS] (cache lohnt
nicht)s
          wende_gravitation_an(drehe(puzzle, &ew[0]))
     } else {
          // Bsp. ew == [US, GUS, US]
          let mut ew = Vec::from(ew);
          // Bsp. US
          let letzte_drehung = ew.pop().unwrap();
          // Bsp. Ergebnis des EWs [US, GUS]
          let puzzle_vor_letzter_drehung =
wende_entscheidungen_an(puzzle, &ew, cache);
          // Bsp. Ergebnis der Drehung US auf vorige variable
          let endzustand =
wende_gravitation_an(drehe(puzzle_vor_letzter_drehung,
&letzte_drehung));
          ew.push(letzte_drehung);
          cache.insert(ew, endzustand.clone());
          endzustand
     }
}

pub fn erstelle_nächste_schicht(base: &[Vec<DrehRichtung>]) ->
Vec<Vec<DrehRichtung>> {
     let mut nächste_schicht = Vec::new();
     // Bsp. base == [[US]]
     // -> nächste_schicht == [[US, US], [US, GUS]]
     for path in base {
          let mut c1 = path.clone();
          let mut c2 = path.clone();
          c1.push(DrehRichtung::US);
          c2.push(DrehRichtung::GUS);
          nächste_schicht.push(c1);
          nächste_schicht.push(c2);
     }
     nächste_schicht
}
```

```
// findet einen EW, der das Puzzle löst oder stellt
// fest, dass es keine Lösung gibt
pub fn optimierte_breiten_suche(puzzle: Puzzle) ->
Option<Vec<DrehRichtung>> {
     // EW == Entscheidungsweg, z.b. [US, GUS, US]
     // cache wie in der Dokumentation beschriben
     let mut cache = HashMap::new();
     // wie in der Dokumentation
     let mut bekannte_zustände = HashSet::new();
     // ein leerer EW == vec![]
     let mut base = vec![vec![]];
     let mut tiefe = 0; // für Fortschritts-Anzeige
     let start_zeit = Instant::now(); // ^
     loop {
          let paths: Vec<Vec<DrehRichtung>> =
cache.keys().cloned().collect();
          for path in paths {
               // falls der Entscheidungsweg kleiner als die Tiefe
ist,
               // wird er eh nie mehr aus dem Cache geholt,
               // weil der Cache immer zuerst versucht, die
Entscheidungswege
               // der letzten Ebene zu benutzen
               // (betrifft hier also die ebene vor der letzten)
               if path.len() < tiefe {
                    cache.remove(&path); // spart RAM
               }
          }
          tiefe += 1;

          // aus den EWs der letzten Ebene werden
          // die neuen EWs gebaut
          let nächste_schicht = erstelle_nächste_schicht(&base);

          // falls diese leer ist, gibt es keine lösung
          if nächste_schicht.is_empty() {
               return None;
          }
          // eine neue Basis für die nächste runde wird erstellt
          base = Vec::new();
          for ew in &nächste_schicht {
               // EW auf Puzzle anwenden
               let tmp_puzzle =
wende_entscheidungen_an(puzzle.clone(), ew, &mut cache);
               // falls noch nicht bekannt
               if !bekannte_zustände.contains(&tmp_puzzle) {
                    // an base für nächste runde anfügen
                    base.push(ew.clone());
               }
               if puzzle_gelöst(&tmp_puzzle) {
                    // puzzle gelöst: EW zurückgeben
                    return Some(ew.clone());
```

```
                } else {
                    // sonst: zustand merken
                    bekannte_zustände.insert(tmp_puzzle);
                }
            }
            // "Fortschrittsanzeige"
            println!("{:03} tiefe, {:06} nächste_schicht, {:07}
bekannt, {:07} gecached, {:04} sekunden", tiefe,
nächste_schicht.len(), bekannte_zustände.len(), cache.len(),
start_zeit.elapsed().as_secs());
        }
}

fn main() {
    if let Some(dateiname) = env::args().nth(1) {
        let puzzle = lade_puzzle(&dateiname);
        [...]
        // Start-Zustand anzeigen
        let puzzle = wende_gravitation_an(puzzle.unwrap());
        println!("{}", puzzle);

        if puzzle_gelöst(&puzzle) {
            println!("Puzzle ist bereits gelöst?!");
            return
        }

        // hoffentlich eine Lösung finden
        let solution = optimierte_breiten_suche(puzzle.clone());
        if let Some(solution) = solution {
            print!("Lösung: [");
            for (index, drehung) in solution.iter().enumerate()
{
                if index < solution.len()-1 {
                    print!("{}, ", drehung);
                } else {
                    println!("{}]", drehung);
                }
            }
            // gelöstes Puzzle anzeigen
            println!("{}", wende_entscheidungen_an(puzzle,
&solution, &mut HashMap::new()));
        } else {
            println!("keine lösung gefunden!");
        }
    } else {
        println!("Bitte so aufrufen: ./target/debug/Aufgabe3
<dateiname>");
    }
}


// ein paar tests
#[cfg(test)]
```

```
mod tests {
     use super::*;

     #[test]
     fn nächste_schicht() {
          [...]
     }

     #[test]
     fn t1() {
          [...]
     }
     #[test]
     fn t2() {
          [...]
     }
     #[test]
     fn t3() {
          [...]
     }

     [...]
}

// paar benchmarks
#[cfg(test)]
mod benchs {
     [...]
}
```

Aufgabe 4 – Radfahrspaß

Lösungsidee

Ich hatte die Idee, die minimal und maximal mögliche Geschwindigkeit über die Strecke zu bestimmen. Dazu werden alle Buchstaben der Strecke analysiert. Für einen Abschnitt bergab bzw. bergauf werden die minimal und maximal mögliche Geschwindigkeit jeweils um eins erhöht bzw. verkleinert. Falls der Abschnitt flach ist und die zurzeit minimal mögliche Geschwindigkeit null ist, wird die minimal und maximal mögliche Geschwindigkeit um eins erhöht (der Fahrer beschleunigt). Falls die minimal mögliche Geschwindigkeit nicht null (über null) ist, wird sie stattdessen um eins verkleinert (der Fahrer bremst ab).

Falls die maximale Geschwindigkeit in einem Abschnitt unter null ist, würde der Fahrer auf der Strecke zurück rollen, also auf keinen Fall im Ziel ankommen. Es kann verkommen, dass die minimal mögliche Geschwindigkeit unterwegs unter null liegt. Ist dies der Fall wird sie um zwei erhöht, um praktisch eine früher simulierte Abbremsung des Fahrers durch eine Beschleunigung zu ersetzen.

Falls die minimal Geschwindigkeit im Ziel über null liegt, würde der Fahrer zu schnell im Ziel ankommen.

Um zu berechnen, wie oft der Fahrer beschleunigen und abbremsen sollte, um zu gewinnen, wird zunächst die Anzahl aller Abschnitte bestimmt. Die Hälfte davon minus die Anzahl der Abschnitte abwärts ergibt die Anzahl der Beschleunigungen des Fahrers (so beschleunigt das Fahrrad in genau der Hälfte der Abschnitte). Um zu bestimmen, wie oft der Fahrer abbremsen sollte, wird dementsprechend die Hälfte minus die Anzahl der Abschnitte nach oben berechnet.

Umsetzung

Die Lösungsidee wird in Rust implementiert.

Die Struktur `Ergebnis` soll das Ergebnis der Analyse der Strecke enthalten: ob sie befahrbar ist, wie viele Abschnitte bergab gehen, wie viele geradeaus und wie viele bergauf. Die Funktion `befahrbar` durchläuft eine Strecke, die zuvor aus einer Datei geladen wurde.

Im Programm wird zunächst der Dateiname von der Kommandozeile übernommen. Diese Datei wird dann mit der Funktion `befahrbar` untersucht.

In der Funktion werden zunächst Variablen für die minimal mögliche Geschwindigkeit, die maximal mögliche Geschwindigkeit und die Anzahl der Abschnitte angelegt. In einer Schleife wird jeder Buchstabe der Datei eingelesen. Danach arbeitet das Programm wie oben beschrieben, wobei es Buchstaben ignoriert, die es nicht zuordnen kann.

Außerdem hielt ich es für eine gute Idee, statt 49.992.957 mal „+“ auszugeben, einfach nur anzugeben, wie oft der Fahrer beschleunigen bzw. abbremsen sollte (dieses Verhalten kann im Programm mit der (im Programm groß geschriebenen) Konstante `aufgabenstellung_befolgen` ändern). Der Fahrer sollte immer zuerst (auf den flachen Abschnitten) sooft beschleunigen, wie das Programm ausgibt und danach sooft abbremsen, wie das Programm ausgibt.

Beispiele

Wir rufen das Rust-Programm von der Kommandozeile auf (zunächst wird es kompiliert). Alle Ergebnisse werden innerhalb weniger Sekunden berechnet. Die Parcour-Dateien liegen im selben Verzeichnis wie das Programm.

```
$ rustc -C lto -C opt-level=3 aufgabe4rust.rs
$ ./aufgabe4rust parcours1.txt
Nicht befahrbar!
$ ./aufgabe4rust parcours2.txt
Nicht befahrbar!
$ ./aufgabe4rust parcours3.txt
Nicht befahrbar!
$ ./aufgabe4rust parcours4.txt
Befahrbar!
Der Fahrer sollte
```

```
 1001577 mal beschleunigen und
 83 mal abbremsen.
$ ./aufgabe4rust parcours5.txt
Nicht befahrbar!
$ ./aufgabe4rust parcours6.txt
Nicht befahrbar!
$ ./aufgabe4rust parcours7.txt
Nicht befahrbar!
$ ./aufgabe4rust parcours8.txt
Nicht befahrbar!
$ ./aufgabe4rust parcours9.txt
Befahrbar!
Der Fahrer sollte
 49992957 mal beschleunigen und
 252 mal abbremsen.
```

Quelltext (gekürzt)

aufgabe4rust.rs

```
[...]
const AUFGABENSTELLUNG_BEFOLGEN: bool = false; // ;)

struct Ergebnis {
      befahrbar: bool,
      runter: u64,
      gerade: u64,
      hoch: u64
}

fn befahrbar(dateiname: &str) -> Ergebnis {
      let mut runter = 0;
      let mut gerade = 0;
      let mut hoch = 0;

      let mut min_geschwindigkeit = 0;
      let mut max_geschwindigkeit = 0;

      [...]
      for c in reader.chars() {
            let c = c.expect("Datei enthält keinen Text");
            match c {
                  '\\' => {
                        min_geschwindigkeit += 1;
                        max_geschwindigkeit += 1;
                        runter += 1;
                  }
                  '_' => {
                        if min_geschwindigkeit == 0 {
                              min_geschwindigkeit += 1;
                        } else {
                              min_geschwindigkeit -= 1;
                        }
                        max_geschwindigkeit += 1;
```

```
                        gerade += 1;
                  },
                  '/' => {
                        min_geschwindigkeit -= 1;
                        max_geschwindigkeit -= 1;
                        hoch += 1;
                  },
                  _ => {}
            }
            if max_geschwindigkeit < 0 {
                  return Ergebnis { befahrbar: false, runter: runter, gerade:
gerade, hoch: hoch };
            }

            if min_geschwindigkeit < 0 {
                  min_geschwindigkeit += 2;
            }
      }

      if min_geschwindigkeit > 0 {
            return Ergebnis { befahrbar: false, runter: runter, gerade: gerade,
hoch: hoch };
      } else {
            return Ergebnis { befahrbar: true, runter: runter, gerade: gerade,
hoch: hoch };
      }
}

fn main() {
      if let Some(arg1) = env::args().nth(1) {
            let erstes_ergebnis = befahrbar(&arg1);
            match erstes_ergebnis.befahrbar {
                  true => {
                        println!("Befahrbar!");
                        [...]
                        let beschleunigen = (alle/2)-runter;
                        let abbremsen =  (alle/2)-hoch;
                        if !AUFGABENSTELLUNG_BEFOLGEN {
                              println!("Der Fahrer sollte\n {} mal beschleunigen
und\n {} mal abbremsen.",
                                          beschleunigen, abbremsen);
                        } else {
                              for _ in 0..beschleunigen {
                                    print!("+");
                              }
                              for _ in 0..abbremsen {
                                    print!("-");
                              }
                              println!();
                        }
                  },
                  false => println!("Nicht befahrbar!")
            }
      } else {
            println!("Bitte so aufrufen: ./aufgabe4rust <dateiname>");
      }
}
```