# Aufgabe 4: Streichholzrätsel

Team-ID: 00087

Team-Name: One Man Army

Bearbeiter dieser Aufgabe:
Merlin Moelter

8. September 2020

## Inhaltsverzeichnis



## Lösungsidee

### Format

Mein Format macht sich die Eigenschaft der Modelle zunutze, dass alle Streichhölzer miteinander verbunden sein müssen; es kann keines geben, das frei „herumliegt“. Des Weiteren basiert mein Format auf den Regeln, dass alle Winkel der Streichhölzer relativ zur X-Achse ein Vielfaches von 30° sind, und man sie deshalb mühelos ablesen kann, und dass alle Streichhölzer gleich lang sind.

Man kann sich dieses Format wie einen Pfad vorstellen, der bei (0, 0) beginnt. Ein Streichholz wird mit einem Ende auf diese Position gelegt und um einen gegebenen Winkel gedreht. Nun bestehen die Möglichkeiten an das andere Ende dieses Streichholzes oder wieder bei (0, 0) ein neues Streichholz anzulegen und zu drehen. Danach sind es drei verschiedene Positionen, an denen ein neues angelegt werden kann, und so weiter. Mit Hilfe dieser Methode kann jede beliebige Anordnung an Streichhölzern definiert werden.

### Modelle erstellen

Zuerst wird eine, in meinem Format vorliegende, Textdatei eingelesen und nach Zeilen getrennt. Aus jeder dieser Zeilen wird dann ein Objekt erstellt, welches den Winkel, also den Zahlenwert der Zeile, und die Anzahl der Tabs („Tiefe“ genannt) enthält. Diese Objekte werden „Elemente“ genannt. Anschließend wird eine Tiefe – Position Map erstellt die angeben wird, wo ein neues Streichholz angesetzt werden soll, und begonnen über die Elemente zu iterieren. Die Startposition eines Elementes, also die Position, an die es „angelegt“ wird, ist (0, 0), sofern seine Tiefe 0 ist, ansonsten entspricht sie dem der Tiefe zugehörigen Wert in der Tiefe – Position Map.

Um die Endposition eines Elementes zu bestimmen, wird zuerst der Ortsvektor zur Startposition erstellt und zu diesem (1, 0) hinzugefügt. Anschließend wird er um den gegebenen Winkel gedreht und seine Werte ausgelesen. Diese sind die Koordinaten des Endpunktes. Der Wert der Tiefe – Position Map für die Tiefe dieses Elementes wird auf seine Endposition gesetzt.

## Rätsel lösen

Die Streichholzmodelle (eines für „Vorher“ und eines für „Nachher“), die in meinem Format vorliegen, werden eingelesen und mit etwas Vektorgeometrie so formatiert, dass alle Streichhölzer in der Form „von“ ⇔ „zu“ vorliegen. „von“ und „zu“ sind die Koordinaten der beiden Endpunkte eines jeden Streichholzes. Für jedes der beiden Modelle wird dann anhand dieser Koordinaten ein Bild erstellt, wobei das Bild des „Vorher“-Modells aus roten und das des „Nachher“-Modells aus grünen Elementen besteht. Beide Bilder werden mit etwas Transparenz versehen und übereinandergelegt. In dem resultierenden Bild sind rote, grüne und bräunliche Streichhölzer abgebildet. Die roten Streichhölzer bedeuten, dass diese auf die grünen Flächen bewegt werden müssen, um vom „Vorher“- zum „Nachher“-Modell zu gelangen. Die bräunlichen bleiben unverändert.

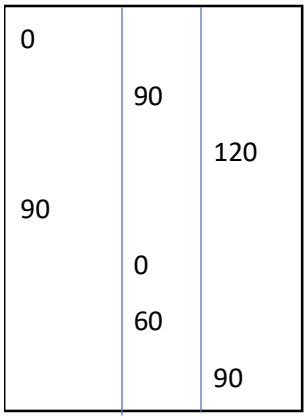
Ein Rätsel gilt als nicht lösbar, wenn die Anzahl der Streichhölzer des „Nachher“-Modelles größer ist als die des „Vorher“-Modelles.

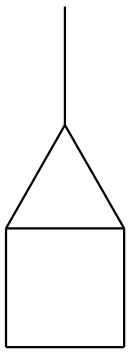
# Umsetzung

## Format

Modelle in meinem Format werden nur mit Tabs (4 Leerzeichen) und Zahlen deklariert. Ein Tab bedeutet, dass das Streichholz an das Ende des Vorherigen angelegt wird, und die Zahl gibt den Winkel des Streichholzes zur X-Achse an.

### Beispiel



Die blauen Striche stellen einen Tab (4 Leerzeichen) dar.

Der Pfad lässt sich so lesen: Setze an der Position (0, 0) ein Streichholz an und drehe es um 0°. Gehe an das Ende dieses Streichholzes und lege ein weiteres an, welches du um 90° drehst. Gehe an das Ende dieses Streichholzes und lege an weiteres an, welches du um 120° drehst. Gehe zurück zur Position (0, 0) und lege ein Streichholz an, welches du um 90° drehst. Gehe an das Ende dieses Streichholzes und lege ein weiteres an, welches du um 0° drehst. Lege an der gleichen Position ein Streichholz an und drehe es um 60°. Gehe an sein Ende und lege ein Streichholz an, welches du um 90° drehst.

Das Resultat ist das Haus rechts neben dem Pfad aus Beispiel 0.

## Modell

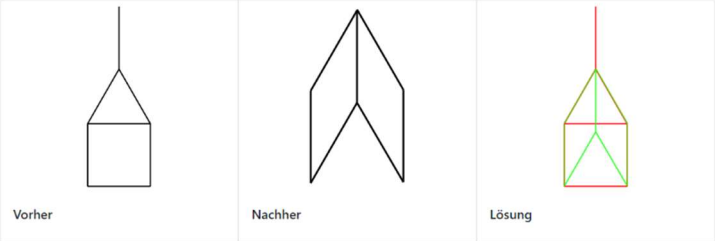
Ein Modell ist ein Objekt der Klasse „Model“, welches mit den Zeilen einer Textdatei im korrekten Format instanziiert wird. Bei der Initialisierung eines solchen Objektes, werden seine Elemente berechnet, also die „Streichhölzer“ mit ihren „von“ und „zu“ Positionen. Alle Positionen werden als Ortsvektoren zu diesen Punkten gespeichert, wobei diese Vektoren Objekte der Klasse „Vector2d“ (zweidimensionaler Vektor) sind.

Die Zahlenwerte der „von“ und „zu“ Positionen werden auf 2 Nachkommastellen abgerundet, da Javascript teilweise ungenau mit Gleitkommaarithmetik ist und dies nur zu Komplikationen führt.

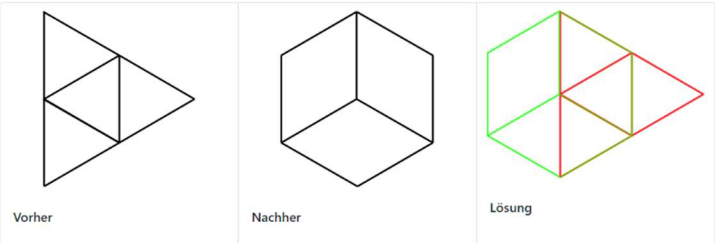
Aus einem Modell wird ein Bild in Form eines SVGs generiert. Diese bieten den Vorteil, durch einfache Text-Anweisungen erstellt zu werden (ähnlich wie HTML) und sind somit für dynamische Generierung, zumindest in diesem Fall, bestens geeignet. Die Elemente zweier Modelle können mit diesem Format ganz einfach übereinandergelegt werden.

## BWINF Aufgabe 4 (Streichholzrätsel)

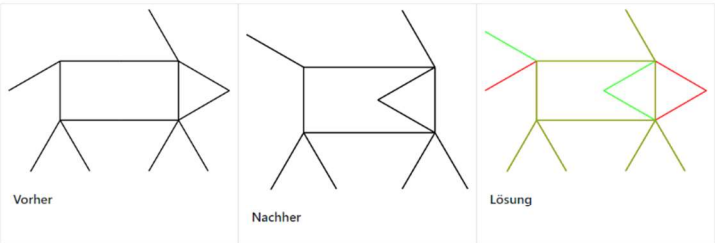
### Beispiel 0



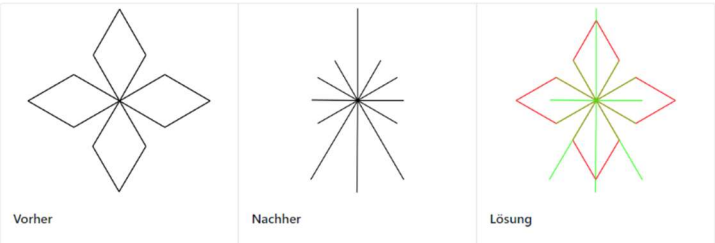

### Beispiel 1



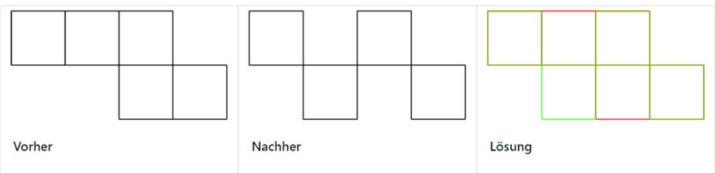

### Beispiel 2



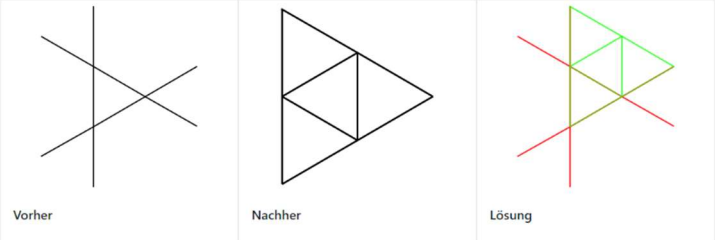

### Beispiel 3




### Beispiel 4




### Beispiel 5



# Beispiele

Das Skript nimmt die Nummer des Beispiel-Ordners als Parameter, generiert die Bilder, und speichert sie in dem Ordner („from.svg“, „to.svg“ und „solution.svg“).

Da alle Lösungen in Form von Bildern vorliegen, habe ich als übersichtliche Zusammenfassung die Datei „Lösung.html“ erstellt, die sich im Stammverzeichnis der Aufgabe befindet. Ein Screenshot dieser Webseite ist auf Seite 4 abgebildet

# Quellcode

## index.js

```
const fs = require("fs")
const path = require("path")
const Model = require("./Model.js")

const inputFileNumber = process.argv[2] || 0

// Konvertiert eine Textdatei zu einem Modell
function convertTXTToModel(path) {
    const content = fs.readFileSync(path, "utf-8")
    const lines = content.replace(/\r/g, "").split("\n")
    return new Model(lines)
}

const DATA_DIR = path.join(__dirname, "beispieldaten", "" +
inputFileNumber)

// "Von"-Modell
const fromModel = convertTXTToModel(path.join(DATA_DIR, "from.txt"))
// "Zu"-Modell
const toModel = convertTXTToModel(path.join(DATA_DIR, "to.txt"))

// Prüfen, ob das Rätsel lösbar ist
if (fromModel.elements.length < toModel.elements.length) {
    throw new Error("Das Rätsel ist nicht lösbar, da im 'Nachher' Modell
mehr Streichhölzer vorhanden sind als im 'Vorher' Modell.")
}

// Ausgabe erstellen
fs.writeFileSync(path.join(DATA_DIR, "from.svg"), fromModel.generateSVG())
fs.writeFileSync(path.join(DATA_DIR, "to.svg"), toModel.generateSVG())
fs.writeFileSync(path.join(DATA_DIR, "solution.svg"),
Model.generateSolution(fromModel, toModel))
```

## Model.js

```
const Vector2d = require("../lib/Vector2d.js")

const STROKE_WIDTH = .02
const FLOATING_POINT_PRECISION = 100
const SVG_MARGIN = 0.1
const SVG_SIZE_FACTOR = 250

// Grad zu Radianten konvertieren
function degToRad(degrees) {
```

```
    return degrees * Math.PI / 180
}

function generateSVGTag(elements) {
    // Hier wird ein SVG Wurzel-Element mit passenden Attributen erstellt,
    // damit das Modell gut sichtbar ist.
}

class Element {
    constructor(from, to) {
        this.from = from
        this.to = to
    }
}

class Model {
    static fromElements(elements) {
        const model = new Model([])
        model.elements = elements
        return model
    }

    static generateSolution(fromModel, toModel) {
        function renderElements(elements, color) {
            return elements.map(({ from, to }) => (
                `<path d="M ${from.value[0]} ${from.value[1]} L
${to.value[0]} ${to.value[1]}" stroke="${color}" stroke-
width="${STROKE_WIDTH}"/>`
            )).join("\n")
        }

        // Alle Pfade werden hier drei mal eingefügt, damit die Farben
nicht zu hell sind
        return `
            ${generateSVGTag([...fromModel.elements, ...toModel.elements])}
                ${renderElements(fromModel.elements, "rgba(255, 0, 0,
.5)")}
                ${renderElements(toModel.elements, "rgba(0, 255, 0, .5)")}

                ${renderElements(fromModel.elements, "rgba(255, 0, 0,
.5)")}
                ${renderElements(toModel.elements, "rgba(0, 255, 0, .5)")}

                ${renderElements(fromModel.elements, "rgba(255, 0, 0,
.5)")}
                ${renderElements(toModel.elements, "rgba(0, 255, 0, .5)")}
            </svg>
        `
    }

    constructor(lines) {
        this.lines = lines

        this.createElements()
    }

    createElements() {
        const moves = this.lines.map(line => ({
            angle: parseInt(line.match(/-?\d+/)[0]),
            depth: (line.match(/\s/g) || []).length / 4
        }))
```

```
        const depthPositionMap = new Map()
        this.elements = []

        for (let move of moves) {
            const startPosition = move.depth > 0 ?
depthPositionMap.get(move.depth - 1) : new Vector2d([0, 0])

            // Das Vorzeichen des Winkels wird hier gewechselt, damit es im
SVG richtig herum dargestellt wird
            const endPosition = startPosition.clone().add(new Vector2d([1,
0]).rotate(degToRad(-move.angle)))

            startPosition.roundTo(FLOATING_POINT_PRECISION)
            endPosition.roundTo(FLOATING_POINT_PRECISION)

            this.elements.push(new Element(startPosition, endPosition))

            depthPositionMap.set(move.depth, endPosition)
        }
    }

    generateSVG() {
        return `
            ${generateSVGTag(this.elements)}
                ${this.elements.map(({ from, to }) => (
                    `<path d="M ${from.value[0]} ${from.value[1]} L
${to.value[0]} ${to.value[1]}" stroke="black" stroke-
width="${STROKE_WIDTH}"/>`
                )).join("\n")}
            </svg>
        `
    }
}

module.exports = Model
```