

Aufgabe 4: Nandu

Team-ID: 00879

Bearbeiter/-in dieser Aufgabe:
Karl Zschiebsch

12. November 2023

Inhaltsverzeichnis

Lösungsidee.....	1
Umsetzung.....	1
Beispiele.....	2
Quellcode.....	2

Lösungsidee

Um die Zustände der LEDs zu bestimmen, wird die Konstruktion als einen Stapel von Ebenen betrachtet, wobei das Licht von der obersten zur untersten Ebene durchgegeben wird. Dieser Stapel ist gerichtet, dh. Licht geht nur von oben nach unten durch, und nicht von unten nach oben.

Das Licht wird von den Taschenlampen über die Bausteine zu der Ausgabe durchgegeben. Hierbei werden alle Bausteine als Blöcke betrachtet, die in der Ebene liegen. Die Blöcke sind jeweils mit einem anderen Block verknüpft, der in genau der selben Ebene liegt. Die Taschenlampen in der obersten Ebene werden als Eingabe betrachtet, während die LEDs in der letzten Ebene als Ausgabe betrachtet werden.

Es werden alle Permutationen berechnet, die alle möglichen Zustände für die Eingabereihe darstellt. Für jede Permutation wird die Eingabereihe entsprechend der Permutation gesetzt. Von der Eingabereihe ausgehend wird für jeden Block bestimmt, ob er Licht zur weiter unten liegenden Ebene weitergibt oder nicht. Ob ein Block Licht weitergibt, hängt von einer Aktivierungsfunktion ab. Diese überprüft den eigenen Zustand bzw. den Zustand des verknüpften Blockes. Diese Aktivierungsfunktion lässt sich eineindeutig aus den eingelesenen Zeichen herleiten. Falls er Licht weitergibt, wird ein Wert für den Sensor des Blockes in der darunter liegenden Reihe entsprechend gesetzt. Dieses Vorgehen wird wiederholt, bis die letzte Ausgabereihe erreicht ist. Alle Zustände der Eingabe und Ausgabequellen werden ausgelesen und ausgegeben. Danach werden die Zustände des Feldes zurückgesetzt. Dieses Verfahren wird für alle verbleibenden Permutationen wiederholt.

Umsetzung

Die Utility-Klasse `Position` speichert schlichtweg die Koordinaten der Blöcke ab.

In `Environment` wird die Konstruktion aus einer Datei eingelesen. `deactivate_all` deaktiviert alle Blöcke. `reactivate_all` hingegen gibt die Aktivierung von der Eingabereihe über alle Ebenen zur Ausgabereihe durch. `get_state` gibt den Wert der Eingabe- und Ausgabereihe aus.

Für die Aktivierungsfunktionen gibt es die folgende Tabelle:

Zeichen	Funktionsname	Erklärung
W	<code>white_activation</code>	Aktiviert, wenn nicht die Sensoren des eigenen Blockes und des verbundenen Blockes aktiviert sind
R	<code>red_major_activation</code>	Aktiviert, wenn der eigene Sensor nicht aktiviert ist.
r	<code>red_minor_activation</code>	Aktiviert, wenn der Sensor des verbundenen Blockes nicht aktiviert ist. Ignoriert den eigenen Sensor, hat effektiv somit keinen Sensor.
B	<code>blue_activation</code>	Aktiviert, wenn der eigene Sensor aktiviert ist

Die Klasse `Block` stellt jeden Block/LED/Sensor dar. Diese Klasse bekommt als Attribute unter anderem `pos`, um die eigene Position in den Ebenen bestimmen zu können, sowie `con`, welches den verbundenen Block festlegt. Das Attribut `activated` legt dabei fest, ob Licht auf den Sensor gefallen ist oder nicht. In dem Attribut `type` wird abgespeichert, welches Zeichen diesen Block darstellt. Dies wird benötigt, um die Aktivierungsfunktion festzustellen (s. Tabelle oben). Die Methode `activates` bestimmt, ob der Block den darunter liegenden Block aktiviert oder nicht. Dafür wird die Aktivierungsfunktion bestimmt. Anmerkung: Eingabefelder, Ausgabefelder und leere Felder haben keine Funktion, wie man in der Tabelle erkennen kann. Diese werden vorher abgefragt, um dadurch verursachte Fehler zu vermeiden. Die Funktion `process_activation` gibt das Licht an der unteren Ebene weiter. Falls es ein Startfeld ist, wird Licht abhängig von der Permutation weitergegeben. Leere Felder geben hier niemals Licht weiter. Dieser Punkt geht nicht ganz klar aus der Aufgabenstellung heraus, es gibt kein Beispiel aus den Aufgaben, womit ich selber diese Annahme getroffen habe. Ist es nicht Leer, also zwingend weiß, blau oder rot, wird dann Licht weiter gegeben, wenn die Aktivierungsfunktion über `activates true` zurück gibt. Alle anderen Funktionen bestimmen den Typ des Blockes aus `type`.

Für die Ausführung wird in einer Schleife über alle möglichen Permutationen der Zustände der LEDs drüber iteriert. Die Zustände werden dabei auf die Werte 0 (Aus) und 1 (An) reduziert. Vor jedem Iterationsdurchgang werden zunächst die Zustände der Blöcke mit der Funktion `deactivate_all` zurück gesetzt. Für jede Permutation wird der Wert der Eingabereihe entsprechend gesetzt. Dann wird das Licht von oben nach unten durch jede Ebene durchgegeben, und am Schluss werden alle Zustände der Eingabe- und Ausgabereihe ausgegeben.

Beispiele

Dies sind die Lösungen zu den ersten drei Beispielen der Website. `nandu0.txt` ist einfach nur das Beispiel aus dem Aufgabenblatt, Abbildung Links. Da die letzten Tabellen relativ groß sind, sind diese in der Datei `task.log` ausgelagert.

```
nandu0.txt
Q1  Q2  |  L1  L2
-----+-----
0   0   |   0   0
1   0   |   0   0
1   1   |   1   1
0   1   |   0   0

nandu1.txt
Q1  Q2  |  L1  L2
-----+-----
0   0   |   1   1
1   0   |   1   1
1   1   |   0   0
0   1   |   1   1

nandu2.txt
Q1  Q2  |  L1  L2
-----+-----
0   0   |   0   1
0   1   |   0   1
1   0   |   0   1
1   1   |   1   0

nandu3.txt
Q1  Q2  Q3  |  L1  L2  L3  L4
-----+-----
0   0   0   |   1   0   0   1
0   0   1   |   1   0   0   0
0   1   0   |   1   0   1   1
0   1   1   |   1   0   1   0
1   0   0   |   0   1   0   1
1   0   1   |   0   1   0   0
1   1   0   |   0   1   1   1
1   1   1   |   0   1   1   0
```

Quellcode

Dies ist der Quellcode, geschrieben in Python. Es wird `itertools` importiert, um alle Permutationen der Zustände der Taschenlampen zu berechnen.

```
import itertools

class Position:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

    def __add__(self, other) -> 'Position':
```

```

    return Position(self.x + other.x, self.y + other.y)

class Environment:
    def __init__(self, path: str):
        self.fields: list[list['Block']] = []
        self.sources = []
        self.results = []
        with open(path, 'r') as reader:
            self.n, self.m = [int(v) for v in reader.readline().replace('\n', '').split(' ')]
            for y in range(self.m):
                self.fields.append([])
                types = reader.readline().replace('\n', '').replace(' ', '').split(' ')
                last = None
                for x in range(self.n):
                    block = Block(self, Position(x, y), types[x].strip())
                    if block.requires_connection():
                        if last is None:
                            last = block
                        else:
                            block.con = last
                            last.con = block
                            last = None
                    elif block.is_source():
                        self.sources.append(block)
                    elif block.is_result():
                        self.results.append(block)
                    self.fields[y].append(block)

    def deactivate_all(self) -> None:
        for array in self.fields[1:]:
            for field in array:
                field.activated = False

    def reactivate_all(self) -> None:
        for array in self.fields[:-1]:
            for field in array:
                field.process_activation()

    def get_state(self) -> str:
        build = ""
        for source in self.sources:
            build += f'{source.activated:<5}'
        build += ' | '
        for result in self.results:
            build += f'{result.activated:<5}'
        return build

    def white_activation(white: 'Block') -> bool:
        return not (white.is_activated() and white.con.is_activated())

    def red_major_activation(red_major: 'Block') -> bool:
        return not red_major.is_activated()

```

```
def red_minor_activation(red_minor: 'Block') -> bool:
    return not red_minor.con.is_activated()

def blue_activation(blue: 'Block') -> bool:
    return blue.is_activated()

class Block:
    activation_map = {
        'W': white_activation,
        'R': red_major_activation,
        'r': red_minor_activation,
        'B': blue_activation
    }

    def __init__(self, env: Environment, pos: Position, t: str, con: 'Block' = None):
        self.env = env
        self.pos = pos
        self.con = con
        self.type = t
        self.activated = False

    def is_activated(self) -> bool:
        return self.activated

    def activates(self) -> bool:
        func = Block.activation_map.get(self.type)
        if func is None:
            raise ValueError(f'Type : {self.type} ? {self.is_source()}')
        return func(self)

    def process_activation(self):
        if self.is_empty():
            return
        block = self.env.fields[self.pos.y + 1][self.pos.x]
        if self.is_source():
            block.activated = self.activated
        elif self.requires_connection():
            if self.activated():
                block.activated = True
        else:
            raise ValueError(self.type)

    def requires_connection(self) -> bool:
        return self.is_white() or self.is_blue() or self.is_red()

    def is_source(self) -> bool:
        return self.type[0] == 'Q'

    def is_result(self) -> bool:
        return self.type[0] == 'L'
```

```
def is_white(self) -> bool:
    return self.type[0] == 'W'

def is_blue(self) -> bool:
    return self.type[0] == 'B'

def is_red(self) -> bool:
    return self.type.capitalize()[0] == 'R'

def is_empty(self) -> bool:
    return self.type[0] == 'X'

for i in itertools.product(range(2), repeat=len(environment.sources)):
    environment.deactivate_all()
    for j in range(len(environment.sources)):
        environment.sources[j].activated = i[j] == 1
    environment.reactivate_all()
    print(environment.get_state())
```