Aufgabe 3: Zauberschule

Team-ID: 00879

Bearbeiter/-in dieser Aufgabe:
Karl Zschiebsch

12. November 2023

Inhaltsverzeichnis

Lösungsidee

Inspiriert wurde die Lösungsidee vom Breath-First Search Algorithmus. Dabei wird um den kürzesten Weg zu finden, vom Startfeld ausgehend alle Nachbarn markiert. Die markierten Felder markieren wiederum ihre Nachbarn. Dies wird solange wiederholt, bis ein Feld das Zielfeld markiert. Dann wird vom Zielfeld zurückverfolgt, von welchen Feldern diese jeweils selbst markiert wurden. Der Weg, der darüber zurückverfolgt wurde, ist der schnellste.

Es wird noch beachtet, dass nicht alle Felder “gleichzeitig” ihre Nachbarn markieren, sondern es eine Zeitverzögerung gibt von 1 bzw. 3 Sekunden gibt. Diese Zeitverzögerung wird durch einen Queue erreicht. Das Queue beinhaltet immer aus drei Listen. Wenn neue Felder markiert werden sollen, wird die vorderste Liste entfernt und hinten eine neue Liste eingefügt. Dadurch rotieren die Listen und somit auch ihre Elemente. Dann wird die gerade entfernte Liste abgearbeitet. Wie oben beschrieben, werden die Felder markiert. Alle markierten Felder werden in das Queue eingefügt. Wurde ein Feld auf dem selben Stockwerk markiert, wird es in der ersten Liste eingefügt, ansonsten in der in der dritten Liste.

Umsetzung

`Position` ist eine einfache Utility-Klasse, die nur die Positionen der Felder bzw. Aktionen als Koordinaten darstellt.

`Building` liest beim erstellen eine Datei ein, um daraus das Gebäude mit ihren Feldern zu erstellen. Dabei werden alle Felder in `fields` gespeichert. Während des Einlesens wird das Startfeld in `start` zwischengespeichert. `n` und `m` sind die Höhe bzw. Breite der Stockwerke. Es wird angenommen, dass es immer nur zwei Stockwerke gibt. `schedule` ist dabei das Queue. Das Queue wurde über eine Python Liste ermöglicht. Die Hauptmethode ist hierbei `find_fastest`.

Dabei wird zunächst beim Startfeld, welches zwischengespeichert wurde, die Methode `conquer` aufgerufen. Diese Methode erstellt für alle Nachbarn wie oben beschrieben eine `Action`. Diese Klasse markiert, wenn sie im Queue abgearbeitet wird, das Zielfeld. Gleichzeitig speichert es für später die Aktion ab, von der es selbst markiert wurde.

Beispiele

Unten angefügt sind alle Ergebnisse für die jeweilige Dateien. Die Symbole, um den Weg darzustellen, sind die selben wie in der `README.txt` beschrieben. Vor dem Weg steht jeweils noch die Zeit, die man für den Weg brauchen würde.

```
zauberschule0.txt
(  8s) !>>!

zauberschule1.txt
(  4s) <<^^

zauberschule2.txt
( 14s) >>!>>!vv>>

zauberschule3.txt
( 28s) vv>>^^>>>>vv>>>>^^>>>>>>^^>>

zauberschule4.txt
( 84s) ^!^^^!<<<<<<^^<<<<<<<<<vv<<^^<<<<<<<<<!^^^^!<<!<<vv<<^^<<vv!<<<<^^<<^^<<<<

zauberschule5.txt
(124s) !vv>>vv>>>>>>!>>>>>>>>!vv>>>>>>>>^^^^!^^>>!>>>>>>!^^^^>>^^>>>>^^^^!
^^>>^^>>>>>>!vv!vv>>>>>>>>!>>>>>>^^<<
```

Quellcode

Dies ist der Quellcode in Python. Es werden keine Bibliotheken benötigt.

```python
class Position:

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

    def __sub__(self, other: 'Position') -> 'Position':
        return Position(self.x - other.x, self.y - other.y, self.z - other.z)


class Building:
    def __init__(self, path: str):
        self.fields = [[], []]
        self.schedule: list[list['Action']] = [[], [], []]
        with open(path, 'r') as reader:
            self.n, self.m = [int(v) for v in reader.readline().split(' ')]
            for z in range(len(self.fields)):
                for y in range(self.n):
                    self.fields[z].append([])
                    for x in range(self.m):
                        c = Field(Position(x, y, z), reader.read(1))
```

```
                        if c.is_start():
                            self.start = c
                        self.fields[z][y].append(c)
                    reader.read(1)
                reader.read(1)
        if self.start is None:
            raise ValueError()

    def find_fastest(self) -> str:
        Action(building.start).conquer(building)
        while True:
            self.schedule.append([])
            for action in self.schedule.pop(0):
                if action.target.occupied is None:
                    action.target.occupied = action.origin
                    action.conquer(building)
                if action.target.is_end():
                    return f'({action.get_runtime():3.0f}s)
{action.traceback()}'

    def is_inside(self, p: Position) -> bool:
        return 0 <= p.x < self.m and 0 <= p.y < self.n and 0 <= p.z <= 1

    def is_available(self, p: Position) -> bool:
        return self.is_inside(p) and self.get_field(p).is_available()

    def get_field(self, p: Position) -> 'Field':
        if self.is_inside(p):
            return self.fields[p.z][p.y][p.x]
        else:
            raise IndexError(f'{p} is not in field!')

    def __str__(self) -> str:
        build = ''
        for z in range(len(self.fields)):
            for y in range(self.n):
                for x in range(self.m):
                    build += self.fields[z][y][x].type
                build += '\n'
            build += '\n'
        return build


class Field:
    def __init__(self, p: Position, t: str):
        if t == '\t':
            raise AssertionError(f'Invalid type for {p}')
        self.p = p
        self.type = t
        self.occupied = None

    def is_available(self) -> bool:
        return (self.is_floor() or self.is_end()) and self.occupied is None

    def is_floor(self) -> bool:
        return self.type == '.'

    def is_start(self) -> bool:
        return self.type == 'A'

    def is_end(self) -> bool:
        return self.type == 'B'

    def neighbours(self, b: Building) -> list['Field']:
```

```
        neighbours = []
        for i in range(-1, 2, 2):
            p_0 = Position(self.p.x + i, self.p.y, self.p.z)
            if b.is_available(p_0):
                neighbours.append(b.get_field(p_0))
            p_1 = Position(self.p.x, self.p.y + i, self.p.z)
            if b.is_available(p_1):
                neighbours.append(b.get_field(p_1))
        return neighbours


class Action:

    def __init__(self, target: Field, origin: 'Action' = None):
        self.target = target
        self.origin = origin

    def conquer(self, b: Building) -> None:
        t = self.target  # Shortcut
        for f in t.neighbours(b):
            b.schedule[0].append(Action(f, origin=self))
        pos = Position(t.p.x, t.p.y, (1, 0)[t.p.z])
        if b.is_available(pos):
            f = b.get_field(pos)
            b.schedule[2].append(Action(f, origin=self))

    def get_action(self) -> str:
        diff = self.target.p - self.origin.target.p
        if diff.x < 0:
            return '<'
        if diff.x > 0:
            return '>'
        if diff.y < 0:
            return '^'
        if diff.y > 0:
            return 'v'
        if diff.z != 0:
            return '!'

    def traceback(self) -> str:
        if self.target.is_start():
            return ''
        return self.origin.traceback() + self.get_action()

    def get_runtime(self) -> int:
        time = 0
        for char in self.traceback():
            if char == '!':
                time += 3
            else:
                time += 1
        return time
```