Aufgabe 3: Zauberschule

Team-ID: 00879

Bearbeiter/-in dieser Aufgabe:
Karl Zschiebsch

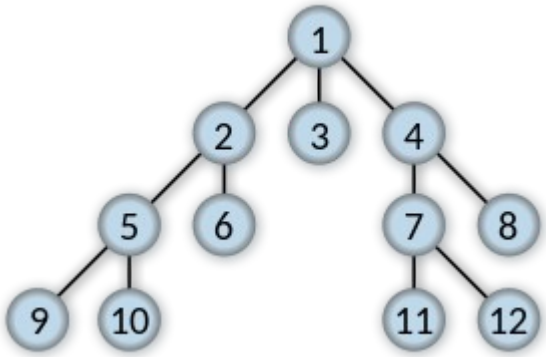
12. November 2023

Inhaltsverzeichnis

Lösungsidee

Ziel ist, den schnellsten Weg vom Startfeld zum Zielfeld zu finden.

Inspiriert wurde die Lösungsidee vom Breath-First Search Algorithmus. Bei diesem Algorithmus wird ein Labyrinth als Graph dargestellt. Die einzelnen Kreuzungen sind die Knotenpunkte des Graphen. Ausgehend vom Startfeld werden verbundenen Knoten markiert, welche wiederum ihre verbunden Knoten markieren. Dies wird wiederholt, bis ein Knoten das Zielfeld markiert.



Beispielbaum für Breath-First Search Algorithmus mit Reihenfolge, in der die Knoten markiert werden.

Bildquelle: https://en.wikipedia.org/wiki/File:Breadth-first-tree.svg

Auf dieses Problem angewendet wird die Zauberschule als Labyrinth auf einen Graphen reduziert. Um den kürzesten Weg zu finden, wird 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.

Hinzu kommt, 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

und Aktionen erreicht. Das Queue beinhaltet immer 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. Für jedes markierte Feld wird eine Aktion für jedes Nachbarfeld erstellt und in das Queue eingefügt. Es wenn die Aktion im Queue abgearbeitet wurde, wird auch das Feld markiert. Soll ein Feld auf dem selben Stockwerk markiert werden, wird eine Aktion in der ersten Liste eingefügt, ansonsten in der in der dritten Liste.

Umsetzung

`Position` ist eine hier 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, eine neue Aktion erstellt und die Methode `conquer` dort aufgerufen. Diese Methode erstellt für alle Nachbarn wie oben beschrieben eine `Action`. Diese Klasse markiert, wenn sie im Queue abgearbeitet wird, das Feld `target`. Gleichzeitig speichert es für später die Aktion `origin` ab, von der es selbst markiert wurde. Wenn eine Aktion das Zielfeld erfolgreich markiert, wird die Funktion `traceback` aufgerufen. Diese rekursive Funktion verfolgt den Wert `origin`, bis es wieder beim Startfeld angekommen ist, und speichert alle Aktionen auf diesem Weg ab. Wenn in `find_fastest` die Funktion `traceback` ausgegeben wurde, ist damit das Programm beendet.

Die Felder selbst werden durch die Klasse `Field` dargestellt. Diese Klasse hat die Funktion `neighbours`, welche eine Liste der angrenzenden freien Felder zurück gibt, sowie mehrere Funktionen, um zu bestimmen, welche Art von Feld es ist. Ein Feld zählt genau dann als frei bzw. verfügbar, wenn es ein Flur oder Zielfeld ist und noch nicht markiert wurde.

Beispiele

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

```
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 3.10. 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 = [[], []]  # Felder
        self.schedule: list[list['Action']] = [[], [], []]  # Queue mit Aktionen
        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  # Cache für Startfeld
                        self.fields[z][y].append(c)
                    reader.read(1)
                reader.read(1)  # Überspringt '\n'
        if self.start is None:  # Hier sollten wir niemals landen
            raise ValueError()

    def find_fastest(self) -> str:
        Action(building.start).conquer(building)  # Startet Aktionen vom Startfeld
        while True:  # Wiederholt so lange, bis Weg gefunden wurde
            self.schedule.append([])              # -+- Rotiert Queue
            for action in self.schedule.pop(0):  # -+
                if action.target.occupied is None:  # Markiert falls noch nicht markiert wurde
                    action.target.occupied = action.origin
                    action.conquer(building)  # Erstellt Aktionen für anliegende Felder
                if action.target.is_end():  # Beendet falls Zielfeld gefunden
                    return f'({action.get_runtime():3.0f}s) {action.traceback()}'

    def is_inside(self, p: Position) -> bool:
```

```
        # Gibt zurück, ob Position im Feld liegt
        return 0 <= p.x < self.m and 0 <= p.y < self.n and 0 <= p.z <= 1

    def is_available(self, p: Position) -> bool:
        # Gibt zurück, ob die Position noch verfügbar ist
        return self.is_inside(p) and self.get_field(p).is_available()

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


class Field:
    def __init__(self, p: Position, t: str):
        if t == '\t':
            raise AssertionError(f'Invalid type for {p}')
        self.p = p  # Aktuelle Position
        self.type = t  # Type des Feldes (Wand, Flur, Start, Ziel)
        self.occupied = None  # Ob das Feld bereits markiert wurde oder nicht

    def is_available(self) -> bool:
        # Gibt zurück, ob dieses Feld verfügbar ist
        return (self.is_floor() or self.is_end()) and self.occupied is None

    def is_floor(self) -> bool:
        # Gibt zurück, ob dieses Feld ein Flur ist
        return self.type == '.'

    def is_start(self) -> bool:
        # Gibt zurück, ob dieses Feld das Startfeld ist
        return self.type == 'A'

    def is_end(self) -> bool:
        # Gibt zurück, ob dieses Feld das Zielfeld ist
        return self.type == 'B'

    def neighbours(self, b: Building) -> list['Field']:
        # Gibt die Liste aller benachbarten Felder zurück
        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:
        # Erstellt Aktionen für die angrenzenden Felder
        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:
        # Stellt diese Aktion dar
        diff = self.target.p - self.origin.target.p
        if diff.x < 0:  # Schöner mit switch/case
            return '<'
        elif diff.x > 0:
            return '>'
        elif diff.y < 0:
            return '^'
        elif diff.y > 0:
            return 'v'
        elif diff.z != 0:
            return '!'
        else:
            raise ValueError()

    def traceback(self) -> str:
        # Rekursives bestimmen des zurückgelegten Weges
        if self.target.is_start():
            return ''
        return self.origin.traceback() + self.get_action()

    def get_runtime(self) -> int:
        # Gibt die Zeit zurück, die für den Weg benötigt wird
        time = 0
        for char in self.traceback():
            if char == '!':
                time += 3
            else:
                time += 1
        return time
```