Aufgabe 3: Zauberschule

Team-ID: ???

Team: Tobias Maurer

Bearbeiter/-innen dieser Aufgabe:
Tobias Maurer

1. October 2023

Inhaltsverzeichnis

Wenn du neue Überschriften einfügst, solltest du das Inhaltsverzeichnis mittels Rechtsklick aktualisieren.

Die gelb hinterlegten Texte, die hier bereits stehen, geben ein paar Hinweise zur Einsendung. Du solltest sie aber in deiner Einsendung wieder entfernen!

Lösungsidee

In der Aufgabe wird der schnellste Weg von einem Startpunkt zu einem Ziel in der Zauberschule „Bugwarts" gesucht. Bugwarts besteht laut der Aufgabenstellung aus Feldern.

Erreichbare Felder sind alle Felder, welche kein Wandfeld sind – somit alle freien Felder, das Startfeld und das Zielfeld.

Benachbarter Felder sind die Felder vom Feld *f*, welche erreichbar sind durch einen Stockwerkwechsel oder eine Bewegung nach links, rechts, oben und unten auf dem selben Stockwerk von *f* aus.

Man kann nun das gegebene „Bugwarts" in einen Graphen übertragen. Dazu wird jedes Feld in einen Knoten umgewandelt und alle benachbarten Felder im Graphen werden mit Kanten verbunden. Das Kantengewicht ist die Zeit, welche man bzw. Ron benötigt, um von dem aktuellem Feld a zu dem benachbarten Feld b zu gelangen. Somit ergebibt sich durch die Aufgabenstellung eine Kantengewicht von 1 für alle Kanten, die Felder auf demselben Stockwerk verbinden, und ein Kantengewicht von 3 für alle Kanten, die Felder für Stockwerkwechsel verbinden.

Nachdem der Graph fertig konstruiert ist – also alle Wege des gegebenen „Bugwarts" darstellt – werden nun die Kantengewicht, welche ursprünglich die Zeit darstellten, als Entfernung

interpretiert. Das heißt, dass die Frage von „Was ist der *schnellste* Weg von A nach B in ‚Bugwarts'" zu „Was ist der *kürzeste* Weg von A nach B in ‚Bugwarts'". Da der Graph alle möglichen Wege in „Bugwarts" represäntiert, kann man nun aus der Graphentheorie den Algorithmus „Dijkstra" verwenden, um den kürzesten Weg zu finden. Da der Algorithmus allerdings nur die kürzesten Entfernungungen von einem Startknoten zu allen anderen Knoten im Graphen berechnet, müssen noch zusätzliche Informationen gespeichert werden, damit der kürzeste Pfad rekonstruierbar ist.

Umsetzung

Bevor man den kürzesten Weg im Graphen berechnen kann, muss die Eingabedatei eingelesen und dabei in einen Graphen umgewandelt werden. Dabei wird für jeden Knoten ein Tripel, welches die x- und y-Koordinate sowie das Stockwerk (die z-Koordinate) enthält, verwendet um später bei der Ausgabe des Ergebnis noch die Position zu kennen. Zudem wird bei jedem Knoten gespeichert, ob es ein freies Feld (*Punkt*) oder ein Wandfeld (#) ist. Wandfelder werden für die später folgende Ausgabe gespeichert, aber sind nicht relevant für die Wegfindung, da diese Felder nicht erreichbar sind. Das Startfeld und das Zielfeld werden als freies Feld interpretiert. Sobald das Startfeld und das Zielfeld beim Einlesen entdeckt werden, werden die Knotenobjekte, welche dem Graph hinzu gefügt werden, als Start- bzw Zielknoten für den kürzesten Wegsuchealgorithmus gespeichert. Nachdem alle Knoten hinzugefügt wurden, werden die Kanten in den Graphen eingefügt, indem für jeden Knoten geschaut wird, welche benachbarten Felder erreichbar sind. In der von mir gegebenen Implementation ist es auch möglich eine anderen Anzahl an Stockwerken als 2 zu haben. Man muss für mehr Stockwerke nur eine leere Zeile - wie davor auch – als Trennung verwenden und das nächste Stockwerk einfügen (siehe Beispiele).

Mit dem erhaltenem Graphen kann man nun den kürzesten Weg suchen. Man hat beim Einlesen den Start- und Zielknoten zwischengespeichert. Jetzt verwendet man den dijkstra-Algorithmus, um die kürzesten Entfernungen vom Startknoten aus zu berechnen. Da man am Ende aber auch am Weg interessiert ist, muss man den dijkstra-Algorithmus so modifizieren, dass der Algorithmus beim Aktualisieren auf einen kürzeren Abstand, auch speichert, von wo aus der Algorithmus „gekommen" ist. Sobald der Algorithmus alle Entfernungen berechnet hat, kann man nun mit der zusätzlichen Informationen den Pfad berechnen. Die geeignetste Datenstruktur um den Pfad darzustellen, ist eine Liste.
Um nun den Pfad in der Liste zu erstellen, erstellt man als erstes eine Liste mit dem Zielknoten als einziges Element. Nun fügt man solange den Vorgänger, welchen man beim Aktualisieren der Entfernung gespeichert hat, des **ersten** Elements der Liste als **neues** erstes Element der Liste ein, bis man den Startknoten als erstes Element der Liste hat. D. h. in der zweiten Iteration, besteht die Liste aus 2 Elementen. Als erstes den gespeicherten Vorgänger vom Zielknoten, da der Zielknoten, der nächste Knoten auf dem Pfad ist, und als zweites Element den Zielknoten.

Zum Abschluss muss man nun den Pfad in das gegebene „Bugwarts" integrieren. Da jeder Knoten über das Tripel, welches die Position wie in einem Koordinatensystem vermerkt, eindeutig identifizierbar ist, kann man mit einem 3-dimensionalen for-Schleifenkonstrukt jeden Knoten des

Graphen – einschließlich der Wandfelder – druchgehen und das Feld Zeile für Zeile bauen. Wenn der ausgewählte Knoten durch das for-Schleifenkonstrukt in der Liste des Pfades vorkommt, muss an dieser Position ein Zeichen, welches die Wegrichtung anzeigt anstelle eines Punktes oder einer Raute gesetzt werden. Das Wegrichtungszeichen ist entweder ein „B“, wenn es der Knoten das letzte Element des Pfades ist – der Zielknoten – oder es muss bestimmt werden. Dazu werden die Tripels des aktuellen Knotens und des darauf folgenden Knotens aus dem Pfad von einander abgezogen, wie in einer Vektorrechnung:

$t_1 = (x_1, x_2, x_3)$
$t_2 = (x_2, y_2, z_2)$
$t_{res} = (x_2 - x_1, y_2 - y_1, z_2 - z_1)$

t_1 steht für das Tripel des aktuellen Knotens im for-Schleifenkonstrukt
t_2 steht für das Tripel, welches im Knoten gespeichert ist, welcher auf den aktuellen Knoten im Pfad folgt
t_{res} steht für das Tripel, welches die Differenz von und bildet.

Mit dem Ergbenistripel kann man nun die Richtung in welche man gehen muss um vom aktuellen Knoten zum nachfolgenden Knoten im Pfad zu gelangen.
Als Ergebnisse für die Richtungen gibt es:

- links
- rechts
- oben
- unten
- Stockwerkwechsel nach oben
- Stockwerkwechsel nach unten

Im Programmcode sind diese Tripel als Schlüssel in einem Python-Dictionary verwendet, um auf das Zeichen, welches zum Markieren der Richtung verwendet werden soll, zu verweisen. Dies hält sich weitgehend an den Vorschlag der BWINF Seite. Da meine Implementation aber auch für mehr als 2 Stockwerke funktioniert, ist in diesem Fall ein Ausrufezeichen nicht mehr aussagekräftig genug, da es bei 3 Stockwerken auf dem mittleren Stockwerk nun sowohl ein Stockwerk nach oben als auch nach unten gehen kann. Daher wird von meinem Programm ein „D“ (=Down) verwendet, wenn zum Stockwerk nach unten (also in der Ausgabe/Datei nach unten zu scrollen ist) gewechselt wird. Ein „U“ (=Up) wird verwendet, wenn das Stockwerk nach oben (also in der Ausgabe/Datei nach oben zu scrollen ist) gewechselt wird. Wechsel werden relativ zur Reihenfolge in der Datei ausgegeben, da man somit nicht überlegen bzw. wissen muss in welcher Richtung die Stockwerke tatsächlich angeordnet sind und somit auch ein Umdenken nie notwendig ist.

Das Programm kann vom Terminal aus wie ein Befehl verwendet werden:
python3 zauberschule.py <input_file1> <input_file2> …

Im Falle dass keine Eingabedateien im Terminal angegeben werden, wird nach einem Pfad zu einer Datei gefragt.

Beispiele

Alle hier aufgeführten Beispiele befinden sich auch im Ordner “results” in dem Ordner für “Aufgabe3”. Der Dateiname ist derselbe wie die gelb hinterlegten Abschnittsmarkierungen.

zauberschule0.txt

Path length: 8
```
#############
#.....#.....#
#.###.#.###.#
#...#.#...#.#
###.#.###.#.#
#...#.....#.#
#.#########.#
#.....#.....#
#####.#.###.#
#....!#B..#.#
#.#########.#
#...........#
#############
```

```
#############
#.......#...#
#...#.#.#...#
#...#.#.....#
#.###.#.#.###
#.....#.#...#
#####.###...#
#.....#.....#
#.#########.#
#...#>>!....#
#.#.#.#.###.#
#.#...#...#.#
#############
```

Die Länge des Pfades dabei ist 8, da Wandfelder ebenfalls eine Breite von einem Feld annehmen und somit sind 2 Felder anstatt einem Feld nach dem Stockwerkwechsel zurückzulegen (für Ron). Dies ergibt sich automatisch, da jedes Zeichen als ein Feld interpretiert wird.
Ausrufezeichen werden aus dem Grund für Stockwerkwechsel verwendet, da es nicht mehr als 2 Stockwerke in dem gegebenen „Bugwarts“ gibt.

zauberschule1.txt
Path length: 4
```
####################
```

```
#...#.....#...#.....#
#.#.#.###.#.#.#.###.#
#.#.#...#.#.#...#...#
###.###.#.#.#####.###
#.#.#...#.#B....#...#
#.#.#.###.#^###.#####
#.#...#.#.#^<<#.....#
#.#####.#.#########.#
#...................#
#####################
```

```
#####################
#.......#.....#.....#
#.###.#.#.###.#.###.#
#.....#.#.#.....#.#.#
#######.#.#######.#.#
#.....#.#.....#...#.#
#.###.#.#.###.###.#.#
#.#.#...#.#...#...#.#
#.#.#######.###.###.#
#.........#.......#
#####################
```

zauberschule2.txt

Path length: 14

```
###############################################
#...#.....#...........#.#...........#.......#
#.#.#.###.#########.#.#.#.#######.#.#.#####.#
#.#.#...#.#.........#>>!#v#.....#.#.#...#...#
###.###.#.#.##############v#.#.###.#.###.#.###
#.#.#...#.#..............>>B#.#...#.#...#.#.#
#.#.#.###.############.#########.###.#.###.#.#
#.#...#.#.#.........#.#.#.....#.#.....#.#.#.#
#.#####.#.#.#######.#.#.#.###.#.#######.#.#.#
#.....#...#...#.#...#...#.#.#.#.......#.#.#.#
#.#####.#####.#.#.#######.#.#.#######.#.#.#.#
#.....#.......#.#.#.....#.#.#.#...#...#...#.#
#.###.#########.#.###.#.#.#.#.#.#.#.###.###.#
#...#.................#...#.....#...#.......#
###############################################
```

```
###############################################
#...#.......#.....#.....#...#...#.....#.....#
#.#.#.#####.###.#.###.#.#.#.#.#.#.###.###.###
#.#.#.....#.#...#.....#>>!#.#.#...#.#...#...#
###.#.###.#.#.##############.#.#####.###.###.#
#.#.#...#.#.#.#.....#...#.#.#...#.#...#.#...#
```

```
#.#.#####.#.#.#.###.#.#.#.#.#.#.#.#.#.#.###
#.#...#...#.#.....#.#.#...#.#.#.#...#.#.#...#
#.###.#.###.#.#####.#.###.#.###.#####.#.###.#
#...#.#.#...#...#...#...#.#...#.#.....#.....#
#.###.#.#.#######.#####.#####.#.#.#.#######.#
#...#...#...#...#.....#.......#.#.#.#.....#.#
#.#.#######.#.#.#####.#.#######.#.###.###.#.#
#.#...........#.......#.........#.......#...#
###############################################
```

zauberschule3.txt
Path length: 28

```
################################
#...#.....#...........#.......#
#.#.#.###.#.###.########.###.#.#
#.#.#...#.#.#.#.#.......#...#.#
###.###.#.#.#.#.#.########.###.#
#.#.#...#.#...#.....#.....#.#.#
#.#.#.###.############.#####.#.#
#.#...#.#.#.........#...#.....#
#.#####.#.#.########.###.#.#####
#...#.#...#...#.#...#...#.#...#
#.#.#.#.#####.#.#.###.###.#.#.#
#.#.#.#.......#.#.#...#.#...#.#
#.#.#.##########.#.#.###.######.#
#.#.......#.#.....#.#.#.....#.#
#.#########.#.#.#####.#.#.###.#.#
#.#.....#...#.#.#...#.#.#.#.#...#
#.#.###.#####.#.#.#.#.#.#.#####
#.#...#.....#.#.#.#.#...#.....#
#.###.#####.#.#.#.#.###.######.#
#...#.#...#...#...#...#.......#
#.#.#.#.###############.#.########
#.#.#.#..............#.#.....#
#.###.#.########.##########.###.#
#...#.#.#...#...#...#.........#...#
###.#.###.#.#.###.######.###.#.#
#...#.....#.#.....#>>B#.#...#.#
#.##########.########^#.###.###.#
#..v#>>>>v#.#>>>>>>^#...#.#.#.#
#.#v#^###v#.#^##########.#.#.#.#
#.#>>^..#>>>>^#...........#...#
################################

################################
#.......#.....#...#...#.......#
#.###.#.#.#.#.###.#.#.#.######
#.....#.#.#.#.#.....#.#.#.....#
```

```
#######.#.#.#.#.#####.#.#####.#
#.....#.#.#.#.#.#...#...#.....#
#.###.#.###.#.###.#.#.###.#####
#...#.#.#...#.....#.#.#.#.....#
#.#.###.#.#########.#.#.#####.#
#.#.....#.#.......#.#.....#...#
#.#######.#######.#.#.#####.#.#
#...#...#.....#...#.#.#...#.#.#
###.#.#.#.###.#.###.#.#.#.#.#.#
#.#.#.#...#...#.....#.#.#.#.#.#
#.#.#.#####.###.#######.#.#.#.#
#.#.#.....#.#.....#.....#.#.#.#
#.#.#######.#####.###.###.#.#.#
#.#.....#...#...#.....#.#...#.#
#.#####.#.###.#.#######.#####.#
#.#...#.#.#...#.......#.#.....#
#.#.#.#.#.#.#########.#.#.#####
#...#.#.#.#...#...#.#.#.#.....#
#####.#.#.###.#.#.#.#.#.#####.#
#.....#.#.......#.#.#.#...#...#
#.###.#.#########.#.#.#.#.#.###
#...#.#.....#.#.....#.#.#.#.#.#
#.#.#######.#.#.#####.#.#.#.#.#
#.#.......#.#...#.....#.#...#.#
#.###.#####.#####.#####.#####.#
#...#.............#..........#
################################
```

zauberschule5.txt
Die Ausgabe hierbei befindet sich ausschließlich im Ordner „results“ im Ordner „Aufgabe3“ aufgrund der ca. 400 Zeilen Ausgabe.

Eigene Beispiele

Diese Dateien befindeen sich im Ordner „data“ im Ordner „Aufgabe3“ zusammen mit den anderen BWINF Beispielen.

eiegene_zauberschule0.txt

Inhalt des Beispiels:

```
4 4
.###
..##
####
####

A###
```

#.#.
###.
###B

####
#.#.
#...
####

Ausgabe von eigene_zauberschule0.txt
Path length: 18
v###
>D##
####
####

U###
#D#.
###v
###B

####
#v#.
#>>U
####

Hier sieht man das statt einem Ausrufezeichen „U“ oder „D“ als Symbole verwendet wurden. Wie oben bereits erklärt, steht „D“ für den Stockwerkwechsel nach unten relativ gesehen zur Ausgaube und „U“ für den Stockwerkwechsel nach oben.

eigene_zauberschule1.txt

Inhalt des Beispiels:
4 4
A...
###.
###.
B##.

....
###.
###.
….

Ausgabe von eiegen_zauberschule1.txt
Path length: 15

```
>>>v
###v
###v
B##!
```

```
....
###.
###.
!<<<
```

Hier sieht man das der Algorithmus das Stockwerk erst so spät wie möglich wechselt. In der Theorie ist es möglich, dass man als erstes das Stockwerk wechselt und danach im unterem Stockwerk bis zu unteren linken Ecke geht, um dann das Stockwerk zum Zielfeld zu wechseln. Dies wird vom Dijkstra Algorithmus allerdings verhindert, da Abstände nur dann aktualisiert werden, wenn diese kleiner sind als der bereits bekannte Abstand. Somit wird der Vorgänger ebenfalls nicht aktualisiert und es wird solange wie möglich immer der kleinste Abstand zum benachbarten Weg genommen. Dies ist kein Hindernis für Ron, da somit mindestens ein Weg gefunden wird, welcher unter den schnellsten, sofern mehrere gleich schnelle existieren, vorzufinden ist.

Quellcode

Python-Module die verwendet wurden: sys (für erzwungenes Programmende), os (um den Dateipfad zu überprüfen) und typing für Python type hints.

Eigenes Pythonmodul, welches verwendet wurde: graph (gibt einfache Klassen für die Knoten und den Graph)

Der Algorithmus um den kürzesten Weg von Startknoten zum Zielknoten zu finden (befindet sich als Methode in der Graphenklasse wie man oben sieht):

```python
class Graph(Generic[T, V]):
    def find_shortest_path(self, start: Node[T, V], target: Node[T, V]) -> Tuple[List[Node[T, V]], float]:
        # use dijkstra first to calculate the shortest path
        distances: Dict[Node[T, V], float] = {
            start: 0
        }
        previous_nodes = {}
        reachable: List[Node[T, V]] = []

        # initialize dijkstra's informations
        for neighbour in start.destinations.keys():
            if neighbour != start:
                reachable.append(neighbour)
                distances[neighbour] = start.destinations[neighbour]
                previous_nodes[neighbour] = start

        # generate distances
        while reachable:
            current_node = min(reachable, key=lambda k: distances[k])
            reachable.remove(current_node)

            for neighbour in current_node.destinations.keys():
                if distances.get(neighbour, INF) == INF:
                    # new node found
                    reachable.append(neighbour)

                if distances.get(neighbour, INF) > distances[current_node] + current_node.destinations[neighbour]:
                    # shorter path to this neighbour found
                    previous_nodes[neighbour] = current_node  # keep previous node for path retracing
                    distances[neighbour] = distances[current_node] + current_node.destinations[neighbour]

        # retrace path
        path = [target]

        # don't insert start to prevent KeyError since start does not have a previous node, obviously
        while previous_nodes[path[0]] != start:
            path.insert(0, previous_nodes[path[0]])

        path.insert(0, start)  # add start since it is the beginning of the path

        return path, distances[target]
```

Konstanten von der Lösung:

```python
# input chars (spaces cannot be used)
WALL = "#"
WAY = "."
START = "A"
END = "B"

# output chars for the path
UP = "v"
DOWN = "^"
LEFT = "<"
RIGHT = ">"
SWITCH_UP = "U"      # going floor down in a multilayered Bugwarts
SWITCH_DOWN = "D"    # going floor up in a multilayered Bugwarts
SWITCH_FLOOR = "!"   # generally for switching between floors; only used if there are exact two floors

# for which position change which output char should be used
DIRECTION_MAP: Dict[Tuple[int, int, int], str] = {
    (1, 0, 0): RIGHT,
    (-1, 0, 0): LEFT,
    (0, 1, 0): UP,
    (0, -1, 0): DOWN,
    (0, 0, 1): SWITCH_DOWN,
    (0, 0, -1): SWITCH_UP
}

# edge weights
TO_SAME_FLOOR = 1
TO_OTHER_FLOOR = 3
```

Einlesen der Dateien:

Auf nächster Seite

```python
def parse_input(filename: str) -> Tuple[Graph, Node, Node, Tuple[int, int, int]]:
    g: Graph[Tuple[int, int, int], str] = Graph()
    floor = -1
    y = 0
    x = 0
    start = None  # start node for the later dijkstra call
    end = None    # target node for the later dijkstra call
    prev_empty_line = True

    with open(filename, 'r') as f:
        dimensions = list(map(int, f.readline().split(" ")))

        for line in f:
            line = line.rstrip(" \n")
            if not line:
                # new floor
                if y != dimensions[0]:
                    print("Incorrect dimensions")
                    sys.exit(-1)

                y = 0
                prev_empty_line = True
                continue

            # increase floor count when a new floor starts instead of when a floor ends to prevent miscounting
            if prev_empty_line:
                floor += 1

            prev_empty_line = False

            # read in floor line
            for x, char in enumerate(line):
                if char not in (WALL, WAY, START, END):
                    print("Unknown field character %r" % char)
                    sys.exit(-1)

                pos = x, y, floor

                # keep WALL position for output later. WALL nodes won't have any connections later on
                node = Node(pos, char)

                if char == START:
                    start = node
                    node.value = WAY

                if char == END:
                    end = node
                    node.value = WAY

                g.add_node(node)

            if x != dimensions[1] - 1:  # still unordered dimensions
                print(x)
                print("Incorrect dimensions in line %r" % line)
                sys.exit(-1)

            y += 1

    # ensure the right floor count
    floor += 1

    dimensions = [dimensions[1], dimensions[0], floor]  # reorder dimensions to have x, y, z format
    add_connections(g)

    return g, start, end, dimensions  # type: ignore
```

Bestimmt das Zeichen, welches verwendet werden sollte, um die Richtung des Weges anzugeben:

```python
def find_direction(path: List[Node[Tuple[int, int, int], str]], k: Node[Tuple[int, int, int], str], multi_layerd: bool):
    """
    Returns the appropriate char for the direction Ron needs to go

    Args:
        path (List[Node[Tuple[int, int, int], str]]): the path which needs to be traveled
        k (Node[Tuple[int, int, int], str]): the current position on the path
        multi_layered (bool): Should the chars for multi-layered floors be used. `U` and `D` instead of `!`.
            This clarifies the direction if the given Bugwarts has more than two floors

    Returns:
        str: the character for that field
    """
    index = path.index(k)
    if index == len(path) - 1:
        return END

    next_node = path[index + 1]

    pos = k.id
    next_pos = next_node.id

    diff = (
        next_pos[0] - pos[0],
        next_pos[1] - pos[1],
        next_pos[2] - pos[2]
    )

    d = DIRECTION_MAP[diff]
    if not multi_layerd and d in (SWITCH_DOWN, SWITCH_UP):
        return SWITCH_FLOOR
    return d
```

Ausgabe des Ergebnisses:

```python
def visualize(g: Graph[Tuple[int, int, int], str], path: List[Node[Tuple[int, int, int], str]], dimensions: Tuple[int, int, int]):
    for floor in range(dimensions[2]):
        for height in range(dimensions[1]):
            # construct line
            line = ""
            for width in range(dimensions[0]):
                pos = (width, height, floor)
                k = g.nodes[pos]

                # check if has direction and use according char
                char = k.value
                if k in path:
                    char = find_direction(path, k, dimensions[2] >= 3)

                line += char

            print(line, flush=True)  # move into stream
        print()
```

Bindefunktion der drei Elemente (einlesen, schnellsten Weg finden, ausgeben) bzw. main-function:

Auf nächster Seite

```python
def main(*input_files):
    files = list(input_files)

    if not files:
        files.append(input("Input file path: "))

    for file in files:
        if not os.access(file, os.R_OK):
            print("Cannot open input file %r" % file)

        # reading input
        graph, start, end, dimension = parse_input(file)

        # find path
        path, distance = graph.find_shortest_path(start, end)

        # converting result to human readable data
        print("Path length: %i" % distance)
        visualize(graph, path, dimension)
```