

Aufgabe 3: Zauberschule

Team-ID: 00776

Team: BWINF0

Bearbeiter/-innen dieser Aufgabe:
Tobias Maurer

1. Oktober 2023

Inhaltsverzeichnis

Lösungsidee.....	1
Umsetzung.....	2
Beispiele.....	4
Eigene Beispiele.....	7
Quellcode.....	9

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 aktuellen Feld a zu dem benachbarten Feld b zu gelangen. Somit ergibt sich durch die Aufgabenstellung ein 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 Kantengewichte, 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“ repräsentiert, kann man nun aus der Graphentheorie den Algorithmus „Dijkstra“ verwenden, um den kürzesten Weg zu finden. Da der Algorithmus

allerdings nur die kürzesten Entfernungen 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 aber am Ende auch am Pfad und nicht nur der kürzesten Entfernung interessiert ist, fügt man eine weitere Berechnung in den Dijkstra Algorithmus ein. Dazu fügt man bei der Initialisierung ein Mapping ein zweier Felder. In dieses Mapping wird immer vom aktuellen Feld, der Schlüssel, auf das Feld, von welchem man kommt, der Wert, verwiesen. Dieses Mapping aktualisiert man immer, wenn auch ein kürzere Pfad zwischen zwei Feldern gefunden wurde. Da diese Aktualisierung nur statt findet, wenn ein Weg kürzer ist, wird am Ende der Berechnung der kürzeste Pfad, nämlich den, welchen Dijkstra gefunden hat, hinterlegt sein. Sobald der Algorithmus alle Entfernungen berechnet hat, kann man nun mit der zusätzlichen Informationen aus dem Mapping den Pfad berechnen. Die geeignetste Datenstruktur um den Pfad darzustellen, ist eine Liste.

Um nun den Pfad in der Liste aus dem Mapping zu entnehmen, 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 und als zweites Element den Zielknoten.

Zum Abschluss muss man nun den Pfad in das gegebene „Bugwards“ 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 – durchgehen 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 t_1 und t_2 bildet.

Mit dem Ergebnistripel 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.

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

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

Path length: 4

#####

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

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

zauberschule2.txt

Path length: 14

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

```
#####
#...#....#....#....#...#...#....#....#
#.#.#.#####.###.#.###.#.#.#.###.###.###
#.#.#....#.#...#....#>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 befinden sich im Ordner „data“ im Ordner „Aufgabe3“ zusammen mit den anderen BWINF Beispielen.

eigene_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):

```

32 class Graph(Generic[T, V]):
51     def find_shortest_path(self, start: Node[T, V], target: Node[T, V]) -> Tuple[List[Node[T, V]], float]:
52         # use dijkstra first to calculate the shortest path
53         distances: Dict[Node[T, V], float] = {
54             start: 0
55         }
56         previous_nodes = {}
57         reachable: List[Node[T, V]] = []
58
59         # initialize dijkstra's informations
60         for neighbour in start.destinations.keys():
61             if neighbour != start:
62                 reachable.append(neighbour)
63                 distances[neighbour] = start.destinations[neighbour]
64                 previous_nodes[neighbour] = start
65
66         # generate distances
67         while reachable:
68             current_node = min(reachable, key=lambda k: distances[k])
69             reachable.remove(current_node)
70
71             for neighbour in current_node.destinations.keys():
72                 if distances.get(neighbour, INF) == INF:
73                     # new node found
74                     reachable.append(neighbour)
75
76                 if distances.get(neighbour, INF) > distances[current_node] + current_node.destinations[neighbour]:
77                     # shorter path to this neighbour found
78                     previous_nodes[neighbour] = current_node # keep previous node for path retracing
79                     distances[neighbour] = distances[current_node] + current_node.destinations[neighbour]
80
81         # retrace path
82         path = [target]
83
84         # don't insert start to prevent KeyError since start does not have a previous node, obviously
85         while previous_nodes[path[0]] != start:
86             path.insert(0, previous_nodes[path[0]])
87
88         path.insert(0, start) # add start since it is the beginning of the path
89
90         return path, distances[target]

```

Konstanten von der Lösung:

```
7  # input chars (spaces cannot be used)
8  WALL = "#"
9  WAY = "."
10 START = "A"
11 END = "B"
12
13 # output chars for the path
14 UP = "v"
15 DOWN = "^"
16 LEFT = "<"
17 RIGHT = ">"
18 SWITCH_UP = "U"      # going floor down in a multilayered Bugwarts
19 SWITCH_DOWN = "D"    # going floor up in a multilayered Bugwarts
20 SWITCH_FLOOR = "!"   # generally for switching between floors; only used if there are exact two floors
21
22 # for which position change which output char should be used
23 DIRECTION_MAP: Dict[Tuple[int, int, int], str] = {
24     (1, 0, 0): RIGHT,
25     (-1, 0, 0): LEFT,
26     (0, 1, 0): UP,
27     (0, -1, 0): DOWN,
28     (0, 0, 1): SWITCH_DOWN,
29     (0, 0, -1): SWITCH_UP
30 }
31
32 # edge weights
33 TO_SAME_FLOOR = 1
34 TO_OTHER_FLOOR = 3
```

Einlesen der Dateien:

Auf nächster Seite

```

70 def parse_input(filename: str) -> Tuple[Graph, Node, Node, Tuple[int, int, int]]:
71     g: Graph[Tuple[int, int, int], str] = Graph()
72     floor = -1
73     y = 0
74     x = 0
75     start = None # start node for the later dijkstra call
76     end = None # target node for the later dijkstra call
77     prev_empty_line = True
78
79     with open(filename, 'r') as f:
80         dimensions = list(map(int, f.readline().split(" ")))
81
82         for line in f:
83             line = line.rstrip("\n")
84             if not line:
85                 # new floor
86                 if y != dimensions[0]:
87                     print("Incorrect dimensions")
88                     sys.exit(-1)
89
90                 y = 0
91                 prev_empty_line = True
92                 continue
93
94             # increase floor count when a new floor starts instead of when a floor ends to prevent miscounting
95             if prev_empty_line:
96                 floor += 1
97
98             prev_empty_line = False
99
100             # read in floor line
101             for x, char in enumerate(line):
102                 if char not in (WALL, WAY, START, END):
103                     print("Unknown field character %r" % char)
104                     sys.exit(-1)
105
106                 pos = x, y, floor
107
108                 # keep WALL position for output later. WALL nodes won't have any connections later on
109                 node = Node(pos, char)
110
111                 if char == START:
112                     start = node
113                     node.value = WAY
114
115                 if char == END:
116                     end = node
117                     node.value = WAY
118
119                 g.add_node(node)
120
121             if x != dimensions[1] - 1: # still unordered dimensions
122                 print(x)
123                 print("Incorrect dimensions in line %r" % line)
124                 sys.exit(-1)
125
126             y += 1
127
128             # ensure the right floor count
129             floor += 1
130
131             dimensions = [dimensions[1], dimensions[0], floor] # reorder dimensions to have x, y, z format
132             add_connections(g)
133
134     return g, start, end, dimensions # type: ignore

```

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

```

137 def find_direction(path: List[Node[Tuple[int, int, int], str]], k: Node[Tuple[int, int, int], str], multi_layerd: bool):
138     """
139     Returns the appropriate char for the direction Ron needs to go
140
141     Args:
142         path (List[Node[Tuple[int, int, int], str]]): the path which needs to be traveled
143         k (Node[Tuple[int, int, int], str]): the current position on the path
144         multi_layerd (bool): Should the chars for multi-layered floors be used. `U` and `D` instead of `!`.
145         This clarifies the direction if the given Bugwarts has more than two floors
146
147     Returns:
148         str: the character for that field
149     """
150     index = path.index(k)
151     if index == len(path) - 1:
152         return END
153
154     next_node = path[index + 1]
155
156     pos = k.id
157     next_pos = next_node.id
158
159     diff = (
160         next_pos[0] - pos[0],
161         next_pos[1] - pos[1],
162         next_pos[2] - pos[2]
163     )
164
165     d = DIRECTION_MAP[diff]
166     if not multi_layerd and d in (SWITCH_DOWN, SWITCH_UP):
167         return SWITCH_FLOOR
168     return d

```

Ausgabe des Ergebnisses:

```

171 def visualize(g: Graph[Tuple[int, int, int], str], path: List[Node[Tuple[int, int, int], str]], dimensions: Tuple[int, int, int]):
172     for floor in range(dimensions[2]):
173         for height in range(dimensions[1]):
174             # construct line
175             line = ""
176             for width in range(dimensions[0]):
177                 pos = (width, height, floor)
178                 k = g.nodes[pos]
179
180                 # check if has direction and use according char
181                 char = k.value
182                 if k in path:
183                     char = find_direction(path, k, dimensions[2] >= 3)
184
185                 line += char
186
187             print(line, flush=True) # move into stream
188     print()

```

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

Auf nächster Seite

```
192 def main(*input_files):
193     files = list(input_files)
194
195     if not files:
196         files.append(input("Input file path: "))
197
198     for file in files:
199         if not os.access(file, os.R_OK):
200             print("Cannot open input file %r" % file)
201
202             # reading input
203             graph, start, end, dimension = parse_input(file)
204
205             # find path
206             path, distance = graph.find_shortest_path(start, end)
207
208             # converting result to human readable data
209             print("Path length: %i" % distance)
210             visualize(graph, path, dimension)
```