Aufgabe 3: Zauberschule

Team-ID: 01305

Team-Name: RecursionLimitExceeded

Bearbeiter/-innen dieser Aufgabe:
Tim Himmelsbach

20. November 2023

Inhaltsverzeichnis

1 Lösungsidee

Man kann die zwei Stockwerke der Zauberschule als **Graph** $G = (V, E, w)$ betrachten, wobei ein Knoten $v \in V$ ein freies Feld und eine gewichtete Kante $e \in E$ einen möglichen Schritt zwischen zwei Feldern repräsentiert. Die Gewichtung der Kanten $w(e) \in \{1, 3\}$ entspricht der Zeit, die für den jeweiligen Schritt vom Zauberschüler aufgewandt werden muss. Gesucht wird der kürzeste Pfad zwischen dem Startknoten $a \in V$ und dem Zielknoten $b \in V$. Mit dem **A*-Algorithmus** lässt sich das Problem sehr effizient lösen. Dazu wird zunächst der Dijkstra-Algorithmus implementiert, welcher später um eine Heuristik erweitert wird. Die Funktionsweise dieser Algorithmen wird in der Umsetzung erläutert.

2 Umsetzung

2.1 Darstellung des Graphen

Zwei Knoten teilen eine Kante, wenn dessen korrespondierende Felder direkte benachbart sind. Die Position eines freien Feldes $v \in V$ lässt sich als Tripel (x, y, z) ausdrücken. Sei $u \in \{(x + 1, y, z), (x -$

$1, y, z), (x, y + 1, z), (x, y - 1, z), (x, y, 1 - z)\}$ ein benachbartes Feld, gilt $e(v, u)$ und $e(u, v)$ wenn $u \in V$. Die Gewichtung beträgt $w(e) = 3$, wenn sich die Felder auf einem unterschiedlichen Stockwerk befinden $z_v \neq z_u$, sonst $w(e) = 1$. In der Implementation ist es daher unnötig die Kanten explizit zu speicher, da die Kanten eines Knotens in konstanter $\mathcal{O}(1)$ Laufzeit berechnet werden können.

2.2 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus berechnet die Distanz, also die minimale Summe der Kantengewichte eines Pfades, zwischen einem Startknoten a und allen anderen Knoten $v \in V$ in einem positiven gewichteten Graphen. Die Laufzeit beträgt $\mathcal{O}(|V| * |E|)$ bei einer optimalen Implementiereung, wobei $|V|$ die Anzahl der Knoten und $|E|$ die Anzahl der Kanten ist. Zwei Modifikationen wurden unternommen. Um später den optimalen Pfad zu rekonstruieren, wird für jeden Knoten der Vorgänger gespeichert (Zeile 13), also der Knoten über welchen der aktuelle Knoten mit minimaler Distanz erreicht wurde. Außerdem bricht der Algorithmus ab, sobald der Zielknoten erreicht worden ist (Zeile 6), was Rechenzeit spart, wenn nur die Distanz von einem Knoten von Interesse ist.

Algorithm 1 Modifizierter Dijkstra-Algorithmus

```
 1: procedure DIJKSTRA(G, a, b)
 2:     Setze die Distanz d(v) aller Knoten v ∈ V auf ∞
 3:     Setze die Distanz d(a) des Startknotens a auf 0
 4:     while nicht alle Knoten besucht do
 5:         Wähle den unbesuchten Knoten c mit der minimalen Distanz d(c)
 6:         if c = b then
 7:             break
 8:         end if
 9:         Markiere den Knoten c als besucht
10:         for all Knoten q, die eine Kante mit c teilen do
11:             if d(q) > d(c) + w(c, q) then
12:                 Setze die Distanz d(q) auf d(c) + w(c, q)
13:                 Setze den Vorgänger p(q) auf c
14:             end if
15:         end for
16:     end while
17: end procedure
```

2.3 A*-Algorithmus

Der A*-Algorithmus benutzt eine Heuristik, um den Suchbereich zu reduzieren. Beim Auswählen wird neben der Distanz $d(c)$ auch die Heuristik $h(c)$ berücksichtigt. Während $d(c)$ die tatsächliche Distanz zum Startknoten a ist, legt die Heuristik $h(c)$ eine untere Schranke für die Distanz zum Zielknoten b fest. Da die Zauberschule ein Raster ist, ist das die Manhatten Distanz:

$$h(v) = |x_b - x_c| + |y_b - y_c| + 3 * (1 - z_c) \tag{1}$$

In der Praxis führt das dazu, dass der Algorithmus zuerst in Richtung des Ziels sucht und sich dann in die anderen Richtungen ausbreitet. Die theoretische Laufzeit bleibt jedoch gleich.

Datei	**Dijkstra**	**A***
zauberschule1.txt	6	5
zauberschule2.txt	71	17
zauberschule3.txt	243	56
zauberschule4.txt	5023	1210
zauberschule5.txt	7510	3275

Tabelle 1: Anzahl der besuchten Knoten

2.4 Priority Queue

Eine Priority Queue ist ein abstraker Datentyp, welcher in der Funktionsweise der Queue ähnlich ist. Jedoch wird Ordnung nicht nach dem LIFO Prinzip (Last-In-First-Out) bestimmt, sondern nach der Priorität eines Elements. Das kann, wie im Fall des Dijkstra Algorithmus, eine Zahl (die Distanz zum Startknoten) sein. Das Element mit der höchsten Priorität (in diesem Fall mit der niedrigsten Distanz) wird entnommen. Eine naive Implementation würde das minimale Element in linearer Laufzeit $\mathcal{O}(|E|)$ suchen. Doch es geht noch schneller. Mit einem Binary Heap kann das minimale Element in logarithmischer Laufzeit $\mathcal{O}(\log|E|)$ gefunden werden. Ein Binary Heap ist ein Binärbaum. Die Wurzel des Baumes ist das minimale Element. Die Kinder eines Knotens sind größer oder gleich dem Elternknoten. Diese Eigenschaften ermöglichen es das minimale Element in logarithmischer Laufzeit zu entfernen und ein neues Element in logarithmischer Laufzeit einzufügen. Fibonacci Heaps sind noch effizienter, dies würde aber den Rahmen sprengen.

2.5 Implementation

Der Algorithmus wurde in Python implementiert. Beim Ausführen der Datei muss ein Pfad zur gewünschten Beispieleingabe als Argument übergeben werden. Die Implementation benutz das 'heapq' Modul aus der Python-Standart-Bibliothek. Die theoretische Laufzet beträgt $\mathcal{O}(|V| * \log(|V|))$.

3 Beispiele

3.1 zauberschule0.txt

```
Kuerzester Pfad: !>>!
Laenge: 8
Berechnungszeit: 0.0004131793975830078s
```

3.2 zauberschule1.txt

```
Kuerzester Pfad: <<^^
Laenge: 4
Berechnungszeit: 0.0003981590270996094s
```

3.3 zauberschule2.txt

```
Kuerzester Pfad: >>!>>!vv>>
Laenge: 14
Berechnungszeit: 0.0008580684661865234s
```

3.4 zauberschule3.txt

```
Kuerzester Pfad: vv>>^^>>>>vv>>>>^^>>>>>>>^^>>
Laenge: 28
Berechnungszeit: 0.0012390613555908203s
```

3.5 zauberschule4.txt

```
Kuerzester Pfad: ^^<<!^^<<<<^^!<<<<<<<<vv<<^^<<<<<<<<!vv<<^^<<!^^<<^^<<<<^^<<!vv<<<<<<^^!
Laenge: 84
Berechnungszeit: 0.01666879653930664s
```

3.6 zauberschule5.txt

```
Kuerzester Pfad: !vv>>vv>>>>>>!>>>>>>>>vv
    !>>>>>>>>^^^^>>>>>>!^^^^^^^^!>>>>!>>>>^^^^>>^^^^>>!>>>>!vvvv>>>>>>!>>>>!>>>>^^<<
Laenge: 124
Berechnungszeit: 0.040593862533569336s
```

4 Quellcode

4.1 A*-Algorithmus

```
# Priority Queue, die die Punkte nach ihrer heuristischen Distanz sortiert
queue = [(0,0) + start]
# Dictionary, das fuer jeden Punkt den Vorgaenger und den Schritt dorthin speichert
source = {}
while queue:
    _, g, x, y, z = heapq.heappop(queue)
    # Wurde der Punkt schon mal besucht, wird er uebersprungen
    if g > dist[x][y][z]: continue
    if (x, y, z) == end:
        break
    # Berechnung der Kanten des Knotens anhand der Koodinaten
    for tg, tx, ty, tz, tc in [(g+1, x+1, y, z, '>'), (g+1, x-1, y, z, '<'), (g+1, x, y
    +1, z, 'v'), (g+1, x, y-1, z, '^'), (g+3, x, y, not z, '!')]:
        # Das freie Feld kann ueber einen kuerzeren Weg erreicht werden
        if tg < dist[tx][ty][tz]:
            dist[tx][ty][tz] = tg
            # Die Manhatten Distanz beschreibt eine untere Schranke fuer die Distanz zum
    Zielknoten
            h = abs(end[0]-tx) + abs(end[1]-ty) + abs(end[2]-tz) * 3
            # Ein Punkt kann mehrmals in der Priority Queue vorkommen, wenn er spaeter
    ueber einen kuerzeren Weg erreicht wird
            # Deswegen wird der Punkt mit der realen Distanz in die Priority Queue
    eingefuegt
            heapq.heappush(queue, (tg + h, tg, tx, ty, tz))
            source[(tx, ty, tz)] = (x, y, z, tc)
```

4.2 Rekonstruktion des Pfades

```
# Rekonstruktion des Pfades
current = end
path = ""
while current != start:
    x, y, z, c = source[current]
    path = c + path
    current = (x, y, z)
```