42. BWInf - Aufgabe 3

Lösungsidee

Es wird mit einer priorisierenden Breitensuche nach dem Zielort *B* gesucht. Die Liste der zu besuchenden Punkte enthält nicht nur die Punkte selbst, sondern auch die Kosten. Das ist die Zeitdauer, die benötigt wird, um zu Ihnen von Punkt A aus zu gelangen. Die Priorität ergibt sich aus diesen Kosten – je kleiner sie sind, desto höher die Priorität. Der jeweils nächste zu besuchende Punkt ist also derjenige aus der Liste, dessen Kosten am niedrigsten sind.

Die Nachfolger eines Punktes ergeben sich aus den direkten Nachbarn, also die Punkte, die direkt neben oder darüber/darunter liegen. Jeder dieser Nachbarpunkte wird in die Liste der zu besuchenden Punkte aufgenommen. Die jeweils benötigte Zeitdauer ergibt sich aus der Zeitdauer zum aktuellen Punkt plus die Dauer für den Schritt, d.h. 1 in x-/y-Richtung und 3 in z-Richtung (Etagenwechsel).

Bereits besuchte Punkte können in der Folge ausgeschlossen werden. Der Grund dafür ist, dass beim ersten Besuch aufgrund der Priorität schon der schnellste Weg zu ihm verwendet worden ist.

Der kürzeste Pfad wird nach Erreichen vom Zielort berechnet. Es kann keinen noch kürzeren Pfad geben, da der Zielort in der priorisierenden Breitensuche schon früher besucht worden wäre. Um den Pfad am Ende rekonstruieren zu können, wird zu jedem besuchten Punkt auch sein jeweiliger Vorgänger gemerkt. Außerdem werden in der Liste der zu besuchenden Punkte auch die jeweiligen Vorgänger festgehalten.

Die Betrachtung der Komplexität ist wie folgt:

- Im schlimmsten Fall sind alle *n* Punkte des Labyrinths zu besuchen (*n = Breite * Höhe * 2*).
- Die Liste der zu besuchenden Punkte beinhaltet *O(n)* Punkte. Die benötigten Operationen Einfügen und Herausnehmen nach Priorität des nächsten Punktes sind in *O(log n)* erreichbar (siehe Umsetzung).
- Die Liste der bereits besuchten Punkte beinhaltet *O(n)* Punkte. Das Überprüfen, ob ein bestimmter Punkt darin enthalten ist, ist auch in *O(log n)* erreichbar (siehe Umsetzung).
- Die Rekonstruktion des Pfades führt einmalig zu *O(n)*.
- Damit ist die Gesamtkomplexität *O(n log n)*.

Umsetzung

Das Programm ist mit Python geschrieben und benutzt das Modul heapq. Folgende Python-Dateien wurden dafür erstellt:

alle_beispiele_loesen.py

- Startbares Skript
- Findet die Lösung für alle vorgegebenen Beispieldateien (im Verzeichnis Input).
- Dazu für jede Datei
  - Kreiert ein Objekt der Klasse `Labyrinth` (siehe unten), basierend auf dem Inhalt der Beispieldatei.
  - Ruft `loese` (siehe unten) mit dem Objekt aus Punkt 1 auf, um den Pfad mit der kürzesten Zeitdauer zu erhalten.

- Erhält eine Visualisierung des optimalen Pfades innerhalb des Labyrinths, indem die Funktion `zeige_pfad` des Objekts `Labyrinth` (siehe unten) aufgerufen wird.
- Kreiert Output-Datei mit demselben Namen wie die vorgegebene Beispieldatei. Darin werden die optimale Zeitdauer und Pfad sowie die Visualisierung geschrieben. Um die Bewegungsrichtungen zwischen jeweils zwei Punkten auf dem Pfad kenntlich zu machen, wird die Funktion `hole_richtung` des Objekts `Labyrinth` (siehe unten) aufgerufen.

labyrinth.py

- Definiert einen Punkttyp `punkt_typ` als `tuple[int,int,int]`. Dies wird für die Typannotationen der Punkte verwendet.
- Klasse `Labyrinth`
    - Objektattribute
        - `__ist_mauer` mit Typ `list[list[list[bool]]]` als Information, ob an einem Punkt eine Mauer anzutreffen ist. Dabei ist:
            - `List[bool]` => eine Zeile
            - `List[list[bool]]` => eine Etage
            - `List[list[list[bool]]]` => mehrere Etagen (2)
        - Punkte a und b
    - Konstruktor
        - Extrahiert aus dem Eingabetext, der ein Labyrinth darstellt, die Werte der Objektattribute.
    - Funktion `hole_nachbar_mit_kosten`
        - Listet alle möglichen Nachbarpunkte mit Kosten auf. Basierend darauf werden in einer List Comprehension alle Nachbarn aussortiert, bei denen eine Mauer ist. Um diese Information zu enthalten, wird die Funktion `ist_mauer` aufgerufen, die intern auf das Objektattribut `__ist_mauer` zurückgreift.
    - Funktion `hole_richtung`
        - Gibt eine String-Darstellung der Bewegungsrichtung für zwei gegebene Punkte zurück.
        - Dabei wird davon ausgegangen, dass die beiden Punkte benachbart sind. Das bedeutet, dass nur eine Komponente ihrer Koordinate sich unterscheidet (und zwar um 1).
        - Die String-Darstellung ist so wie in der Aufgabenstellung vorgegeben.
    - Funktion `zeige_pfad`
        - Rekonstruiert die Mauern des ursprünglichen Labyrinths basierend auf dem Objektattribut `__ist_mauer` mit # und `.` sowie die Punkte *A* und *B*.
        - Die Punkte auf dem gegebenen Pfad werden darin eingezeichnet mit der Bewegungsrichtung zum jeweils nachfolgenden Punkt aus der Funktion `hole_richtung`.
        - Gibt diese Visualisierung des Labyrinths und Pfads als String zurück.

loeser.py

- Interne Funktion `__konstruire_pfad`
    - Konstruiert einen Pfad basierend auf dessen letzten Punkt und einem Dictionary, in dem für jeden Punkt der jeweilige Vorgänger steht.

    - Zunächst werden die Punkte der Ergebnisliste angehängt (zur Sicherstellung von *O(1)*). Daher wird die Liste am Ende vor der Rückgabe umgedreht.
- Funktion `loese`
    - Setzt Lösungsidee um
    - Speichert die Liste der zu besuchenden Punkte in der Variable `geplant` mit dem Typ `list[tuple[int,punkt_typ,punkt_typ|None]]`.
        - Statt einen eigenen Typ für Heaps anzubieten, arbeitet das Modul heapq auf Listen. Allerdings sind dann `heappush` und `heappop` und nicht die Operationen wie `append` und `pop` anzuwenden.
        - Durch die Verwendung eines Heap wird *O(log n)* für das Hinzugefügen und priorisierte Entnehmen der Punkte erreicht.
        - Der Tupel besteht aus der Zeitdauer von A bis zum zu besuchenden Punkt, dem Punkt selbst und seinem Vorgängerpunkt (soweit vorhanden).
    - Hält die Menge der schon besuchten Punkte in der Variable `gesehen_mit_vorgaenger` mit dem Typ `dict[punkt_typ,punkt_typ|None]`.
        - Ein Set reicht hierfür nicht aus, da zusätzlich zur Information, ob der Punkt schon besucht worden ist, auch noch der jeweiliger Vorgängerpunkt gespeichert werden muss (zur späteren Pfadrekonstruktion).
        - Die Frage, ob ein Punkt schon besucht wurde, wird mit einem effizienten Test auf die Dictionary-Schlüssel umgesetzt.
    - Benutzt die Funktion `hole_nachbar_mit_kosten` des Objektes `Labyrinth`, um die Nachbarn und die Zeitdauer für den Schritt zu ihnen zu erhalten. Daraus werden neue Einträge in `geplant` hinzugefügt.
    - Sobald Punkt B erreicht wurde, wird die interne Funktion `__konstruire_pfad` mit Punkt B und `gesehen_mit_vorgaenger` aufgerufen und das Ergebnis zurückgegeben.

Beispiele

Die Ergebnisse sind pro Beispiel angegeben.

Zauberschule0

```
Dauer des schnellsten Pfades: 8

(5, 9, 0) ! (5, 9, 1) > (6, 9, 1) > (7, 9, 1) ! (7, 9, 0)

##############
#.....#.....#
#.###.#.###.#
#...#.#...#.#
###.#.###.#.#
#...#.....#.#
#.#########.#
#.....#.....#
#####.#.###.#
#....!#B..#.#
#.#########.#
#...........#
##############

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

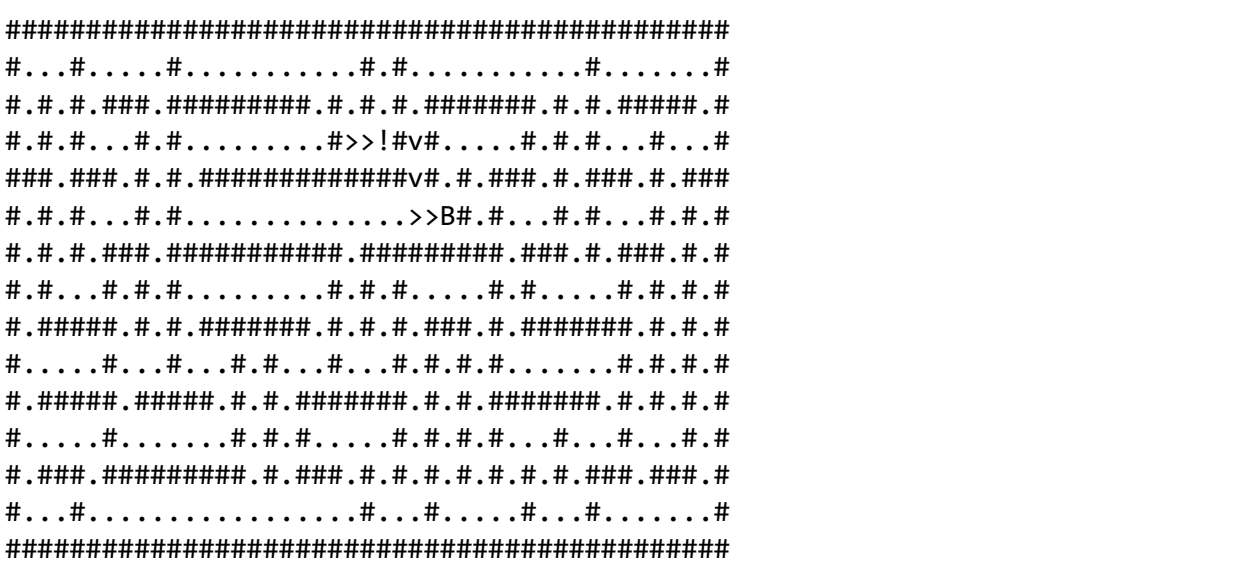
Zauberschule1

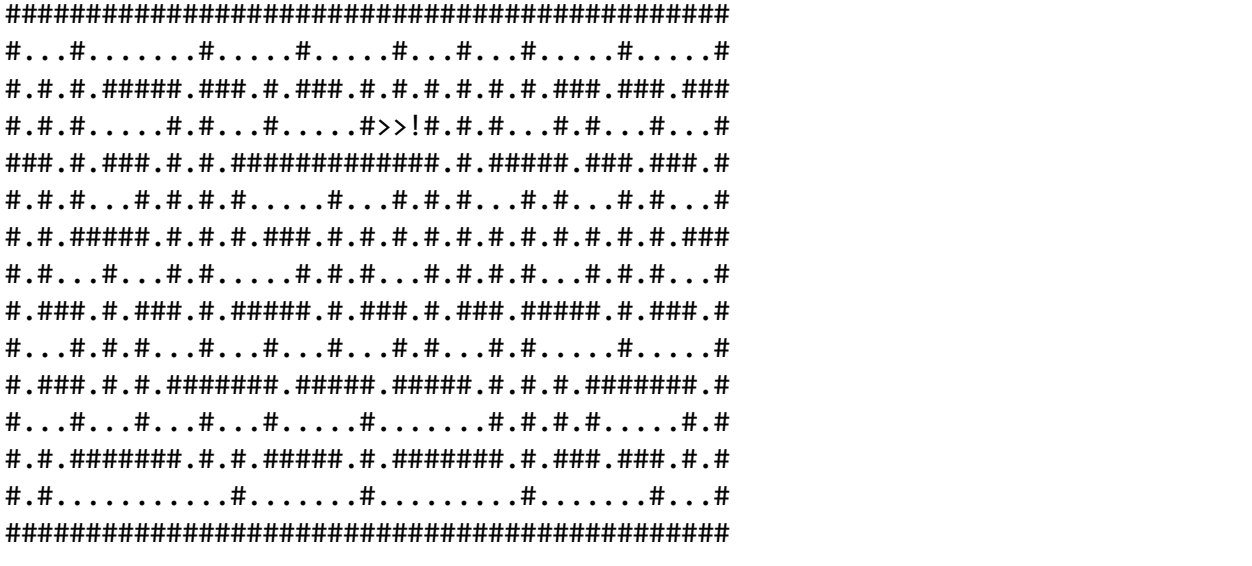
```
Dauer des schnellsten Pfades: 4

(13, 7, 0) < (12, 7, 0) < (11, 7, 0) ^ (11, 6, 0) ^ (11, 5, 0)

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

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

Zauberschule2

Dauer des schnellsten Pfades: 14

(21, 3, 0) > (22, 3, 0) > (23, 3, 0) ! (23, 3, 1) > (24, 3, 1) > (25, 3, 1) !
(25, 3, 0) v (25, 4, 0) v (25, 5, 0) > (26, 5, 0) > (27, 5, 0)

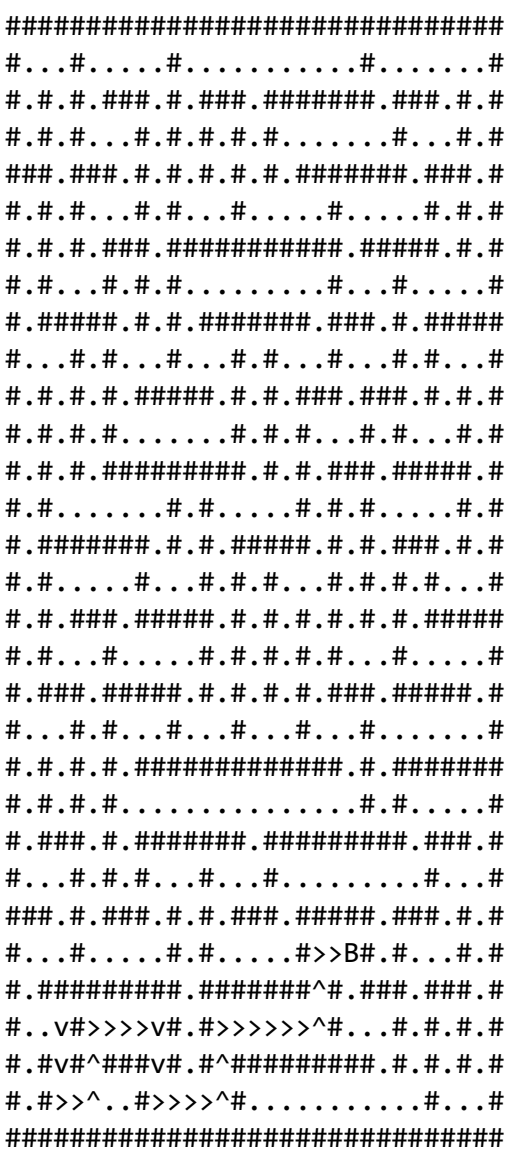


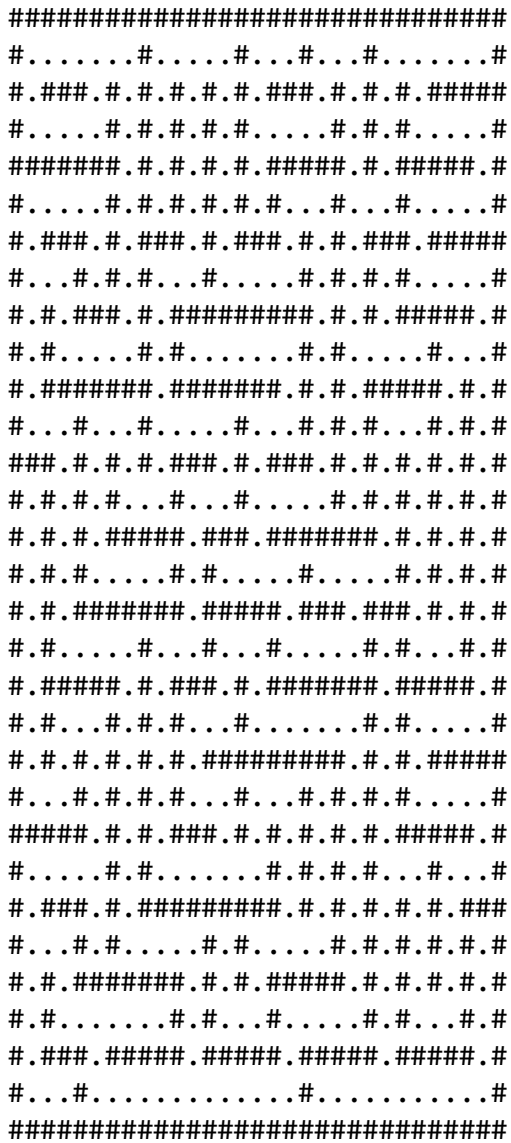


Zauberschule3

Dauer des schnellsten Pfades: 28

```
(3,27,0) v (3,28,0) v (3,29,0) > (4,29,0) > (5,29,0) ^ (5,28,0) ^ (5,27,0) >
(6,27,0) > (7,27,0) > (8,27,0) > (9,27,0) v (9,28,0) v (9,29,0) > (10,29,0) >
(11,29,0) > (12,29,0) > (13,29,0) ^ (13,28,0) ^ (13,27,0) > (14,27,0) >
(15,27,0) > (16,27,0) > (17,27,0) > (18,27,0) > (19,27,0) ^ (19,26,0) ^
(19,25,0) > (20,25,0) > (21,25,0)
```


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



Zauberschule4

Dauer des schnellsten Pfades: 84

```
(73,67,0) ^ (73,66,0) ^ (73,65,0) < (72,65,0) < (71,65,0) ! (71,65,1) ^
(71,64,1) ^ (71,63,1) < (70,63,1) < (69,63,1) < (68,63,1) < (67,63,1) ^
(67,62,1) ^ (67,61,1) ! (67,61,0) < (66,61,0) < (65,61,0) < (64,61,0) <
(63,61,0) < (62,61,0) < (61,61,0) < (60,61,0) < (59,61,0) v (59,62,0) v
(59,63,0) < (58,63,0) < (57,63,0) ^ (57,62,0) ^ (57,61,0) < (56,61,0) <
(55,61,0) < (54,61,0) < (53,61,0) < (52,61,0) < (51,61,0) < (50,61,0) <
(49,61,0) ! (49,61,1) v (49,62,1) v (49,63,1) < (48,63,1) < (47,63,1) ^
(47,62,1) ^ (47,61,1) < (46,61,1) < (45,61,1) ! (45,61,0) ^ (45,60,0) ^
(45,59,0) < (44,59,0) < (43,59,0) ^ (43,58,0) ^ (43,57,0) < (42,57,0) <
(41,57,0) < (40,57,0) < (39,57,0) ^ (39,56,0) ^ (39,55,0) < (38,55,0) <
(37,55,0) ! (37,55,1) v (37,56,1) v (37,57,1) < (36,57,1) < (35,57,1) <
(34,57,1) < (33,57,1) < (32,57,1) < (31,57,1) ^ (31,56,1) ^ (31,55,1) !
(31,55,0)
```

Zauberschule5

Dauer des schnellsten Pfades: 124

```
(13,167,0) ! (13,167,1) v (13,168,1) v (13,169,1) > (14,169,1) > (15,169,1) v
(15,170,1) v (15,171,1) > (16,171,1) > (17,171,1) > (18,171,1) > (19,171,1) >
(20,171,1) > (21,171,1) ! (21,171,0) > (22,171,0) > (23,171,0) > (24,171,0) >
(25,171,0) > (26,171,0) > (27,171,0) > (28,171,0) > (29,171,0) ! (29,171,1) v
(29,172,1) v (29,173,1) > (30,173,1) > (31,173,1) > (32,173,1) > (33,173,1) >
(34,173,1) > (35,173,1) > (36,173,1) > (37,173,1) ^ (37,172,1) ^ (37,171,1) ^
(37,170,1) ^ (37,169,1) > (38,169,1) > (39,169,1) > (40,169,1) > (41,169,1) >
(42,169,1) > (43,169,1) ! (43,169,0) ^ (43,168,0) ^ (43,167,0) ^ (43,166,0) ^
(43,165,0) ^ (43,164,0) ^ (43,163,0) ^ (43,162,0) ^ (43,161,0) ! (43,161,1) >
(44,161,1) > (45,161,1) ! (45,161,0) > (46,161,0) > (47,161,0) > (48,161,0) >
(49,161,0) > (50,161,0) > (51,161,0) ^ (51,160,0) ^ (51,159,0) ^ (51,158,0) ^
(51,157,0) > (52,157,0) > (53,157,0) ^ (53,156,0) ^ (53,155,0) ^ (53,154,0) ^
(53,153,0) ! (53,153,1) > (54,153,1) > (55,153,1) > (56,153,1) > (57,153,1) >
(58,153,1) > (59,153,1) ! (59,153,0) v (59,154,0) v (59,155,0) ! (59,155,1) v
(59,156,1) v (59,157,1) > (60,157,1) > (61,157,1) > (62,157,1) > (63,157,1) >
(64,157,1) > (65,157,1) > (66,157,1) > (67,157,1) ! (67,157,0) > (68,157,0) >
(69,157,0) > (70,157,0) > (71,157,0) > (72,157,0) > (73,157,0) ^ (73,156,0) ^
(73,155,0) < (72,155,0) < (71,155,0)
```

Quelltext: Auszüge

labyrinth.py

```
class Labyrinth:
    …
    def hole_nachbar_mit_kosten(self, c: punkt_typ) \
            -> list[tuple[int, punkt_typ]]:
        x, y, z = c
        kandidaten = [
            (1, (x, y - 1, z)),
            (1, (x - 1, y, z)),
            (1, (x, y + 1, z)),
            (1, (x + 1, y, z)),
            (3, (x, y, 1 - z))
        ]
        # k sind die Kosten, c der Kandidat
        # - ist c eine Mauer? dann aussortieren
        return [(k, c) for k, c in kandidaten if not self.ist_mauer(c)]
    def hole_richtung(self, von: punkt_typ, bis: punkt_typ) -> str:
        vx, vy, vz = von
        bx, by, bz = bis
        # irrelevant, in welche Richtung die Etage gewechselt wurde
        # -> nur der Betrag relevant
        bewegung = (bx - vx, by - vy, abs(bz - vz))
        match bewegung:
            case -1, 0, 0:
                return '<'
            case 1, 0, 0:
                return '>'
            case 0, -1, 0:
                return '^'
            case 0, 1, 0:
                return 'v'
            case 0, 0, 1:
                return '!'
        # Anfrage einer Bewegung nicht zum Nachbarn -> Fehler
        assert False
    …
```

loeser.py

```python
def __konstruire_pfad(letzter_punkt: punkt_typ,
        vorgaenger_fuer_punkt: dict[punkt_typ, punkt_typ | None]) -> list[punkt_typ]:
    ergebnis = []
    p = letzter_punkt
    while p:
        ergebnis.append(p)
        p = vorgaenger_fuer_punkt[p]
    # Punkte des Pfades sind von hinten nach vorne -> Umkehrung notwendig
    ergebnis.reverse()
    return ergebnis

def loese(laby: Labyrinth) -> tuple[int, list[punkt_typ]]:
    gesehen_mit_vorgaenger: dict[punkt_typ, punkt_typ | None] = {}
    # Start mit Punkt A - hat keinen Vorgaenger
    geplant: list[tuple[int, punkt_typ, punkt_typ | None]] \
        = [(0, laby.a, None)]
    while geplant:
        # O(log n) - Liste geplant bleibt nach pop wie Heap strukturiert
        kosten, c, vorgaenger = heapq.heappop(geplant)
        if not c in gesehen_mit_vorgaenger:
            gesehen_mit_vorgaenger[c] = vorgaenger
            if c == laby.b:
                # Abbruchkriterium der Schleife - Ziel gefunden
                # einmalig O(n)
                return (kosten, __konstruire_pfad(laby.b, gesehen_mit_vorgaenger))
            for naechste_kosten, naechstes_c in laby.hole_nachbar_mit_kosten(c):
                # O(log n) - Liste geplant bleibt nach push wie Heap strukturiert
                heapq.heappush(geplant,
                   (naechste_kosten + kosten, naechstes_c, c)
                )
    # Kein Weg von A nach B -> Aufgabe unloesbar
    assert False
```