Team-ID: 00918, Teammitglieder: Bjarne Leichsenring, Nikolai Melnik, Oskar Schories

Aufgabe 3: Zauberschule

Lösungsidee

Wir können den kürzesten Weg von A nach B mit dem folgenden Algorithmus bestimmen: Vom Startpunkt aus gehen wir in alle Richtungen zu den benachbarten Labyrinth-Feldern und speichern den Weg zu ihnen ab. Daraufhin machen wir dasselbe Prinzip mit den benachbarten Feldern und berechnen zu dessen benachbarten Feldern die Wege. Wir wiederholen das bis wir am Zielpunkt ankommen sind und geben den Weg zu diesem Feld aus.

Einschränkungen, durch welche Felder in jeder Iteration durchgegangen werden soll, können wir mithilfe der Wegkosten wie in der Aufgabenstellung vorgegeben implementieren: Falls zu den benachbarten Feldern teleportiert werden muss, wird mit diesen in 3 Iterationen weitergearbeitet, andernfalls in der nächsten Iteration.

Als Edge-Case kann es passieren, dass zwei (oder mehr) Felder ein gleich benachbartes Feld haben und durch es insofern mehrmals iteriert wird. In diesem Fall müssen wir überprüfen, welche der Optionen der kürzeste Weg ist, und diesen speichern.

Der beschriebene Algorithmus ähnelt Pathfinding-Algorithmen wie Dijkstra, ist jedoch in der Implementierung den Vorgaben der Aufgabe angepasst.

Implementierung

Die Lösungsidee wurde in Python umgesetzt. Die Klassen `PathfindedTile` und `PathfindGrid` sind für beliebig-dimensionale Objekte nutzbar, diese Funktion wird aber in der Aufgabe nicht benötigt.

Folgende Klassen wurden implementiert:

`PathfindedTile(location: TileLocation, path: list[TileLocation], path_cost: int, maze_shape: tuple[int])`

Klasse zur Representation von einem Labyrinth-Feld mit spezifischen Informationen. Implementiert Hashing, welches für das Arbeiten mit `dict()`'s gebraucht wird , und eine Funktion `PathfindedTile.adjacent()`, welche die zu den Feld benachbarten Felder generiert, mit der Berücksichtigung der Labyrinthgröße.

`PathfindGrid(self, grid: list, cost: tuple[int], shape: tuple[int])`

Klasse, die das Labyrinth repräsentiert und in diesem auch die Wegsuche an sich durchführt.

Team-ID: 00918, Teammitglieder: Bjarne Leichsenring, Nikolai Melnik, Oskar Schories

Folgende Methoden wurden implementiert:

```
importMaze(filename: str) -> PathfindGrid
```

Importiert ein Labyrinth aus einer Datei und gibt das entsprechende PathfindGrid-Objekt zurück.

```
visualizeMazePath(pfg: PathfindGrid)
```

Gibt das Labyrinth mit dem eingezeichneten Weg mithilfe von pfeilartigen ASCII-Zeichen (`^`, `<`, `>`, `v`) für Richtung und Ausrufezeichen (`!`) für Teleportation zwischen den beiden Etagen aus. Gibt auch den Path Cost aus, bzw. wie viel Sekunden man braucht, um durch das Labyrinth zu kommen.

Beispiele:

Ausgabe von zauberschule0.txt	Ausgabe von zauberschule1.txt
Path Cost: 8 ############## #.....#.....# #.###.#.###.# #...#.#...#.# ###.#.###.#.# #...#.....#.# #.#########.# #.....#.....# #####.#.###.# #....A#B..#.# #.#########.# #...........# ############## ############## #.......#...# #...#.#.#...# #...#.#.....# #.###.#.#.### #.....#.#...# #####.###...# #.....#.....# #.#########.# #...#>>!....# #.#.#.#.###.# #.#...#...#.# ##############	Path Cost: 4 ###################### #...#.....#...#.....# #.#.#.###.#.#.#.###.# #.#.#...#.#.#...#...# ###.###.#.#.#####.### #.#.#...#.#B....#...# #.#.#.###.#^###.##### #.#...#.#.#^<A#.....# #.#####.#.#########.# #...................# ###################### ###################### #.......#.....#.....# #.###.#.#.###.#.###.# #.....#.#.#.....#.#.# #######.#.#######.#.# #.....#.#.....#...#.# #.###.#.#.###.###.#.# #.#.#...#.#...#...#.# #.#.#######.###.###.# #...........#.......# ######################

Ausgabe von zauberschule2.txt

```
Path Cost: 14
###############################################
#...#.....#..........#.#..........#.......#
#.#.#.###.#########.#.#.#.#######.#.#.#####.#
#.#.#...#.#.........#A>!#v#.....#.#.#...#...#
###.###.#.#.##############v#.#.###.#.###.#.###
#.#.#...#.#..............>>B#.#...#.#...#.#.#
#.#.#.###.############.#########.###.#.###.#.#
#.#...#.#.#.........#.#.#.....#.#.....#.#.#.#
#.#####.#.#.#######.#.#.#.###.#.#######.#.#.#
#.....#...#...#.#...#...#.#.#.#.......#.#.#.#
#.#####.#####.#.#.#######.#.#.#######.#.#.#.#
#.....#.......#.#.#.....#.#.#.#...#...#...#.#
#.###.#########.#.###.#.#.#.#.#.#.#.###.###.#
#...#.................#...#.....#...#.......#
###############################################

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

Ausgabe von zauberschule4.txt

```
Path Cost: 28
################################
#...#.....#..........#.......#
#.#.#.###.#.###.#######.###.#.#
#.#.#...#.#.#.#.#.......#...#.#
###.###.#.#.#.#.#.#######.###.#
#.#.#...#.#...#.....#.....#.#.#
#.#.#.###.###########.#####.#.#
#.#...#.#.#.........#...#.....#
#.#####.#.#.#######.###.#.#####
#...#.#...#...#.#...#...#.#...#
#.#.#.#.#####.#.#.###.###.#.#.#
#.#.#.#.......#.#.#...#.#...#.#
#.#.#.#########.#.#.###.#####.#
#.#.......#.#.....#.#.#.....#.#
#.#######.#.#.#####.#.#.###.#.#
#.#.....#...#.#.#...#.#.#.#...#
#.#.###.#####.#.#.#.#.#.#.#####
#.#...#.....#.#.#.#.#...#.....#
#.###.#####.#.#.#.#.###.#####.#
#...#.#...#...#...#...#.......#
#.#.#.#.##############.#.#######
#.#.#.#..............#.#.....#
#.###.#.#######.#########.###.#
#...#.#.#...#...#.........#...#
###.#.###.#.#.###.#####.###.#.#
#...#.....#.#.....#>>B#.#...#.#
#.#########.#######^#.###.###.#
#..A#>>>>v#.#>>>>>>^#...#.#.#.#
#.#v#^###v#.#^#########.#.#.#.#
#.#>>^..#>>>>^#...........#...#
################################

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

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

Für Beispiele 4 und 5 wurden jeweils die Dauer gefragt wegen der :

```
Path Cost (zauberschule4.txt): 84
Path Cost (zauberschule5.txt): 124
```

Team-ID: 00918, Teammitglieder: Bjarne Leichsenring, Nikolai Melnik, Oskar Schories

Quelltext

Die Kommentare wurden aus Gewohnheit in Englisch geschrieben.

```python
import itertools

# types purely for signature readability
TileLocation = tuple[int]

class PathfindedTile:
    # A tile that has been pathfinded to and contains information about the path to it

    def __init__(self, location: TileLocation, path: list[TileLocation], path_cost: int, maze_shape: tuple[int]):
        self.location = location
        self.path = path
        self.path_cost = path_cost
        self.maze_shape = maze_shape

    def adjacent(self):
        # generates tiles that are adjacent to the current tile, by iterating over every dimension and getting the tile coordinates that are offset by +1/-1
        # doesn't go out of bounds using the maze shape
        for inx in range(len(self.location)):
            if self.location[inx] + 1 < self.maze_shape[inx]:
                adj = list(self.location)
                adj[inx] += 1
                yield tuple(adj), inx
            if self.location[inx] - 1 >= 0:
                adj = list(self.location)
                adj[inx] -= 1
                yield tuple(adj), inx

    def __hash__(self):
        # hash function to work properly with sets
        return self.location.__hash__()

    def __eq__(self, other):
        # equality function to work properly with sets
        if isinstance(other, PathfindedTile):
            return self.location == other.location
        return self.location == other

class PathfindGrid:

    WALL: str = "#"

    def __init__(self, grid: list, cost: tuple[int], shape: tuple[int]):
        self.grid = grid
        self.movement_cost = cost
        self.shape = shape

    def pathfind(self, begin_tile: TileLocation, end_tile: TileLocation) -> tuple[list[TileLocation], int]:
```

```
        # pathfinds from begin_tile to end_tile. returns the path cost and the path itself.


        # first 'edge' is the starter tile. we use a dictionary due to needing to access the
PathfindedTile-instances out of here later
        edges: dict[TileLocation,  PathfindedTile] = {
            begin_tile : PathfindedTile(
                location = begin_tile,
                path = [begin_tile],
                path_cost = 0,
                maze_shape = self.shape
            )
        }

        visited: set[TileLocation] = set()


        # the path cost that we are currently working with. is used to work with the tiles that we need
        # the amount of loops that is required to finish this algorithm is consequently the cost of the path
itself.
        path_cost = 0

        while True:

            if not edges: # aquivalent to when edges is empty
                raise Exception("Out of options, cannot continue pathfinding! Does the maze have a solution?")

            # edges that are going to be used in the next iteration of the while loop
            next_edges: dict[TileLocation, PathfindedTile] = {}

            for tile_location, tile in edges.items():

                # ignoring tiles to which our current path cost cannot reach
                if path_cost < tile.path_cost:
                    # just readding it to the dictionary
                    next_edges[tile_location] = tile
                    continue

                # we're done, good job!
                if tile == end_tile:
                    return tile.path, tile.path_cost

                for adjacent_tile_location, direction in tile.adjacent():

                    tile_value: str = self.getTileValue(adjacent_tile_location)

                    if tile_value == self.WALL or adjacent_tile_location in visited:
                        continue

                    adjacent_tile_path: list[TileLocation] = tile.path + [adjacent_tile_location]
                    adjacent_tile_cost: int = tile.path_cost + self.movement_cost[direction]
```

```
                    # specific edge case where 2 tiles have the same adjacent, and one of the paths is longer
                    # (due to the next_tile of the other iteration using a direction with bigger path cost)
                    # we can't update the visited on the fly because we have to check the path lenght and cannot
simply skip over
                    if adjacent_tile_location in next_edges.keys():
                        duplicate_tile: PathfindedTile = next_edges[adjacent_tile_location]
                        if duplicate_tile.path_cost <= adjacent_tile_cost:
                            continue

                    adjacent_tile = PathfindedTile(
                        location = adjacent_tile_location,
                        path = adjacent_tile_path,
                        path_cost = adjacent_tile_cost,
                        maze_shape = self.shape
                    )

                    next_edges[adjacent_tile_location] = adjacent_tile

            edges = next_edges.copy()

            # mark the visited locations as actually visited to not go through them again
            visited = visited.union(next_edges.keys())

            path_cost += 1

    def getTileValue(self, loc: TileLocation) -> str:
        # acquires a tile value from any-dimensional grids
        item = self.grid
        for _loc in loc:
            item = item[_loc]
        return item


def importMaze(filename: str) -> PathfindGrid:
    with open(filename, "r") as f:
        data = f.read().split("\n")
    height, width = map(int, data[0].split())
    grid = [list(k) for k in data[1:height+1]], [list(k) for k in data[height+2:2*height+2]]
    return PathfindGrid(
        grid = grid,
        cost = (3, 1, 1),
        shape = (2, height, width)
    )

def visualizeMazePath(pfg: PathfindGrid):
    def searchTileValue(value, grid: PathfindGrid):
        # itertools.product generates the cartesian product, equivalent to a nested for-loop
        # (gives us all the possible coordinates that we have to go through in a singular for-loop
        # any amount of dimensions)
```

```
        for loc in itertools.product(*[range(k) for k in grid.shape]):
            if grid.getTileValue(loc) == value:
                return loc
    begin_tile = searchTileValue("A", pfg)
    end_tile = searchTileValue("B", pfg)

    path, cost = pfg.pathfind(begin_tile, end_tile)
    grid = pfg.grid

    for inx, current_tile in enumerate(path[:-1]):

        next_tile = path[inx+1]

        # in the 2 cases below, due to implementation, each value represents the folllowing:
        #   - x/dx describes the top/bottom level,
        #   - y/dy the row, and
        #   - z/dz the column
        dx, dy, dz = tuple([nxt - cur for nxt, cur in zip(next_tile, current_tile)])

        x, y, z = current_tile

        grid[x][y][z] = {
            (1, 0, 0) : "!",
            (-1, 0, 0) : "!",
            (0, 1, 0) : "v",
            (0, -1, 0) : "^",
            (0, 0, 1) : ">",
            (0, 0, -1) : "<"
        }[dx, dy, dz]

    grid[begin_tile[0]][begin_tile[1]][begin_tile[2]] = "A"

    # (an unholy) way to string up the maze in a singular row instead of looping 3 times.
    string = "\n\n".join("\n".join("".join(row) for row in level) for level in grid)

    return f"Path Cost: {cost}\n" + string


if __name__ == "__main__":
    for i in range(6):
        res = visualizeMazePath(importMaze(f"files/zauberschule{i}.txt"))
        with open(f"output/zauberschule{i}.txt","w") as f:
            f.write(res)
```