Schiebeparkplatz

Team-ID: 00940

Team: Lovely_Infestation

Bearbeiter/-innen dieser Aufgabe: Lars Noack

18. November 2020

Inhaltsverzeichnis

Lösungsidee

- Man hat eine Situation, in der 7 Autos (A, B, C, D, E, F, G) vertikal in einem Parkplatz stehen.
- Vor den 7 Autos steht eine variable Anzahl von Autos horizontal, die den 7 Autos die Ausfahrt aus dem Parkplatz versperrt.
- Jetzt ist die Aufgabe, für jedes Einzelne der 7 Autos den Weg freizubekommen, so dass diese aus der Parklücke herausfahren können.
- Dies wird erreicht, indem man die Autos horizontal verschiebt.
- Wenn man versucht die Autos horizontal zu verschieben, können sie entweder nach links, oder nach rechts geschoben werden.
    - Da man den Weg möglichst schnell (mit möglichst wenig Zügen) freibekommen soll, kann man einfach ausprobieren, ob nach links oder nach rechts schieben mehr Schritte braucht.
    - Wenn nur rechts oder links zum Ziel führt, wird die jeweilige andere Richtung genommen.
    - Wenn keine der beiden Möglichkeiten zum Ziel führt, ist die Aufgabe unlösbar.
- Um dem Ziel näherzukommen, schiebt man das Auto, das die gewünschte Ausfahrt verstellt, in die gewünschte Richtung.
    - Das Auto kann in 2 Fällen nicht in diese Richtung geschoben werden:
        - Wenn es in eine Mauer fahren würde.
            - In diesem Fall kann die Ausfahrt nicht geräumt werden, indem in diese Richtung gefahren wird.
        - Wenn es in ein anderes Auto fahren würde.
            - In diesem Fall wird das Auto, in welches der Wagen hineinfahren würde in die gleiche Richtung geschoben.
            - Natürlich muss bei diesem Verschieben genau das Gleiche beachtet werden.

    - Kann man das Auto erfolgreich verschieben, verschiebt man es, und fügt einen Schritt hinzu.
- Dies macht man so lange, bis entweder die Ausfahrt frei ist, oder ein Auto an die Wand fahren würde.

Umsetzung

- Für die Programmierung nutzte ich die Skriptsprache Python, da ich damit vergleichsweise einfach Code schreiben kann.
- Die Positionen aller Autos speicherte ich in einer 2-Dimensionalen Liste, in der jeweils das erste Element der inneren Liste das Auto, welches herauswill ist, und das zweite der Platz vor dem Auto.

```
# ein leerer String bedeutet, dass dort kein Auto steht (Ich kenne None)
cars = [
    ["A", ""],
    ["B", ""],
    ["C", "H"],
    ["D", "H"],
    ["E", ""],
    ["F", "I"],
    ["G", "I"],
]
```

- Rückblickend wäre es jedoch deutlich sauberer und einfacher gewesen, die Autos in einer eindimensionalen Liste zu speichern, da es unnötig ist die feststeckenden Autos mitzuspeichern.

```
cars = ["", "", "H", "H", "", "I", "I"]
```

- Für das Verschieben ist eine Rekursion gut geeignet. Das heißt, dass sich Funktionen selber aufrufen.

```
# eine Schleife basierend auf Rekursion
def repeat(iterations: int):
    #do something
    if iterations > 0:
        repeat(iterations-1)
```

- Als Erstes aber braucht man eine Funktion, die ein Paar von Buchstaben (ein horizontales Auto) in die gewünschte Richtung verschieben kann.
    - Dafür wird zuerst die Liste von Autos kopiert.
    - Dann geht es die originale Liste durch.
    - Wenn ein Buchstabe des Autos gefunden wird:
        - Wenn an der gleichen Stelle, um eins in die gewünschte Richtung verschoben,
            - ein Auto ist, wird die gleiche Funktion in der Funktion **rekursion** nur mit dem Buchstaben des kollidierenden Autos ausgeführt.
            - die Liste aufhört, lösche die kopierte Listen und gebe -1 zurück, was in diesem Fall bedeutet, dass das Auto in diese Richtung nicht fahren kann. `return -1`

        - setze an der gleichen Stelle in der kopierten Liste das Element auf kein Auto. *(Wichtig ist, dass je nach Fahrtrichtung in der richtigen Richtung die Liste durchschaut wird.)*
        - setze die um eins verschobene Position des Buchstabens in der kopierten Liste auf den Buchstaben.
    - Wurde die ganze Liste durchsucht, ohne dass einer dieser Fälle eingetreten ist, wird die kopierte Liste zurück gegeben `return copy_of_list` , und die Schrittzählervariable wird um eins erhöht.

- Die oben beschriebene Funktion wird sowohl in die linke, als auch rechte Richtung mit einer weiteren Funktion so lang ausgeführt, bis entweder -1 zurückgegeben wird *(das Auto kann nicht fahren wegen einer Wand)* oder eine Ausfahrt für das gewünschte Auto frei ist.
- Danach werden die Schrittzahlen, der beiden Versuche *(links und rechts)* verglichen.
    - Der Versuch mit der jeweils niedrigsten Schrittzahl ist der kürzere und wird in der Konsole ausgegeben.
    - Wenn ein Versuch keinen Weg frei machen konnte, wird der andere Weg in der Konsole ausgegeben
    - Wenn beide Richtungen keinen Weg frei machen konnten, wird eine Fehlermeldung ausgegeben, dass entweder mein Code einen Bug hat oder die Aufgabe unlösbar ist (in diesem Fall wäre das erste jedoch wahrscheinlicher)

- Dies wird mit einer einfachen Schleife auf jeden vorgegebenen Parkplatz angewendet.

Beispiele

Parkplatz 0

```
A:
B:
C: H 1 rechts
D: H 1 links
E:
F: I 1 links, H 1 links, I 1 links
G: I 1 links
```

Parkplatz 1

```
A:
B: P 1 rechts, O 1 rechts
C: O 1 links
D: P 1 rechts
E: O 1 links, P 1 links
F:
G: Q 1 rechts
H: Q 1 links
I:
J:
K: R 1 rechts
L: R 1 links
M:
N:
```

Parkplatz 2

```
A:
B:
C: O 1 rechts
D: O 1 links
E:
F: R 1 rechts, Q 1 rechts, P 1 rechts
G: P 1 links
H: R 1 rechts, Q 1 rechts
I: P 1 links, Q 1 links
J: R 1 rechts
K: P 1 links, Q 1 links, R 1 links
L:
M: S 1 links, P 1 links, Q 1 links, R 1 links, S 1 links
N: S 1 links
```

Parkplatz 3

```
A:
B: O 1 rechts
C: O 1 links
D:
E: P 1 rechts
F: P 1 links
G:
H:
I: Q 2 links
J: Q 1 links
K: Q 1 links, R 1 links, Q 1 links, R 1 links
L: Q 1 links, R 1 links
M: Q 1 links, R 1 links, S 1 links, Q 1 links, R 1 links, S 1 links
N: Q 1 links, R 1 links, S 1 links
```

Parkplatz 4

```
A: R 1 rechts, Q 1 rechts
B: R 1 rechts, Q 1 rechts, R 1 rechts, Q 1 rechts
C: R 1 rechts
D: R 2 rechts
E:
F:
G: S 1 rechts
H: S 1 links
I:
J:
K: T 1 rechts
L: T 1 links
M:
N: U 1 rechts
O: U 1 links
P:
```

```
A: R 1 rechts, Q 1 rechts
B: R 1 rechts, Q 1 rechts, R 1 rechts, Q 1 rechts
C: R 1 rechts
D: R 2 rechts
E:
F:
G: S 1 rechts
H: S 1 links
I:
J:
K: T 1 rechts
L: T 1 links
M:
N: U 1 rechts
O: U 1 links
P:
```

Quellcode

```
import sys


def read_file(path: str):
    with open(path, 'r') as park_file:
        park_str = park_file.read()
        park_list = park_str.split('\n')

        other_cars = park_list[2:-1]

        # get askii of start and stop
        start_stop = park_list[0].split(' ')
        start_char, stop_char = ord(start_stop[0]), ord(start_stop[1])

        cars = []
        while (start_char <= stop_char):
            cars.append([chr(start_char), ""])
            start_char += 1

        for other_car in other_cars:
            car_info = other_car.split(' ')
            cars[int(car_info[1])][1] = car_info[0]
            cars[int(car_info[1]) + 1][1] = car_info[0]

        return cars


def left(local_car_list_lookup_ref: iter, car: str, moves: list):
    local_car_list_lookup = list(local_car_list_lookup_ref)
    local_car_list = list(local_car_list_lookup)

    # loops through the whole list searching for the letter car
    for i, car_elem_local in enumerate(local_car_list):
        if car_elem_local[1] == car:
```

```
            # if the element left to found letter is in bounds
            if 0 <= i - 1 < len(local_car_list):

                # if element is not occupied
                if local_car_list[i - 1][1] == '':
                    # move left
                    local_car_list[i][1] = ''
                    local_car_list[i - 1][1] = car

                # if element is occupied
                else:
                    # move the car on this element left first
                    # recursive
                    result = left(local_car_list_lookup, local_car_list[i - 1][1], moves)
                    if result == -1:
                        return -1

                    # then move the actual car
                    result = left(result[0], car, result[1])
                    return result
            else:
                return -1

    moves.append(car)
    return local_car_list, moves


def right(local_car_list_lookup_ref: iter, car: str, moves: list):
    local_car_list_lookup = list(local_car_list_lookup_ref)
    local_car_list = list(local_car_list_lookup)

    # loops through the whole list searching for the letter car
    for n, car_elem_local in enumerate(reversed(local_car_list)):
        i = len(local_car_list) - n - 1

        if car_elem_local[1] == car:
            # if the element left to found letter is in bounds
            if 0 <= i + 1 < len(local_car_list):

                # if element is not occupied
                if local_car_list[i + 1][1] == '':
                    # move right
                    local_car_list[i][1] = ''
                    local_car_list[i + 1][1] = car

                # if element is occupied
                else:
                    # move the car on this element left first
                    # recursive
                    result = right(local_car_list_lookup, local_car_list[i + 1][1], moves)
                    if result == -1:
                        return -1

                    # then move the actual car
                    result = right(result[0], car, result[1])
                    return result
            else:
                return -1

    moves.append(car)
```

```
    return local_car_list, moves


def check_left(index: int, car_tuple: iter, moves=[]):
    car_list = list(car_tuple)
    if car_list[index][1] == '':
        return moves

    # tries to move the car in front once
    result = left(car_list, car_list[index][1], moves)
    if result == -1:
        return -1

    car_list, moves = result

    # if the horizontal car still cant move try again
    # recursive
    return check_left(index, car_list, moves)


def check_right(index: int, car_tuple: iter, moves=[]):
    car_list = list(car_tuple)
    if car_list[index][1] == '':
        return moves

    # tries to move the car in front once
    result = right(car_list, car_list[index][1], moves)
    if result == -1:
        return -1

    car_list, moves = result

    # if the horizontal car still cant move try again
    # recursive
    return check_right(index, car_list, moves)

def format_mov(moves: iter, label: str):
    if moves == -1:
        return '', sys.maxsize

    new_moves = []
    prev = ''
    for single_move in moves:
        if single_move != prev:
            new_moves.append([single_move, 1])
            prev = single_move
        else:
            new_moves[-1][1] += 1

    new_mov_str = []
    for new_move in new_moves:
        new_mov_str.append(f'{new_move[0]} {new_move[1]} {label}')

    return ', '.join(new_mov_str), len(moves)


# loop through all 6 parking lots
for i in range(6):
    print(f'\n\nrun parkplatz{i}.txt\n')

    # read file and convert it to 2d list
```

```
PATH = f'parkplatz{i}.txt'

CAR_LIST = read_file(PATH)
print(str(CAR_LIST) + '\n')

# iterate through every horizontally parking Car
for i, car_elem in enumerate(CAR_LIST):
    # tries and saves every necessary step if u want to go left
    CAR_LIST = read_file(PATH)
    left_moves = check_left(i, CAR_LIST, [])
    left_moves, left_weight = format_mov(left_moves, 'links')

    # tries and saves every necessary step if u want to go right
    CAR_LIST = read_file(PATH)
    right_moves = check_right(i, CAR_LIST, [])
    right_moves, right_weight = format_mov(right_moves, 'rechts')

    # compares the number of turns between left and right and displays
    # the one with less turns
    if left_weight < right_weight:
        print(f'{car_elem[0]}: {left_moves}')
    else:
        print(f'{car_elem[0]}: {right_moves}')
```