Aufgabe 1

Lösungsidee

Bei dieser Aufgabe hatte ich die Idee, einen Algorithmus zu entwerfen, der so lange Autos verschiebt, bis entweder eine Lösung, also eine Folge von Verschiebungen, die das jeweilige Auto ausfahren lässt, oder eben keine solche Lösung gefunden werden konnte. Dieser Algorithmus würde also für eine gegebene Position jede endliche Abfolge von Verschiebungen testen, die eine Auswirkung auf diese Position hat und dann die Abfolge wählen, die die wenigsten Verschiebungen benötigt.

Umsetzung

Hierzu habe ich in Python eine Klasse `ParkingLot` entworfen, die den gesamten Parkplatz darstellt. Sie hat die Attribute `normal_cars`, welches die "normalen", also vorwärts geparkten Autos enthält, `sideways_cars`, welches die seitwärts stehenden Autos enthält, `blocked_spots`, welches die durch die seitlich stehenden Autos blockierten Stellplätze speichert und `solution`, welches die Lösung des Parkplatzes speichert. Die Objekte in `sideways_cars` und `normal_cars` sind vom Typ `SideWaysCar` bzw. `NormalCar`, welche von der (theoretisch abstrakten, jedoch in Python Zusatzaufwand durch `abc`) Basisklasse `Car` erben. All diese Objekte haben die Eigenschaften `position` und `is_movable`, welche die Position und die Möglichkeit der Verschiebung, also bei normalen Autos die Ausfahrt und bei seitlichen eine Verschiebung um einen gegebenen Wert darstellen. Außerdem hat `SideWaysCar` eine spezfische Setter-Methode, die bei Änderungen der Position auch die Eigenschaft `blocked_spots` der Elternklasse vom Typ `ParkingLot` aktualisiert. Bei der Erstellung der Lösung wird nun also über die Positionen des Parkplatzes iteriert und falls eine Position blockiert ist, wird versucht, das blockierende Auto um eine Position nach links oder rechts zu verschieben, zuerst die Variante, die weniger Schritte bedeutet. Ist keins der beiden möglich, wird die selbe Methode wieder aufgerufen, jedoch wird nun als Position die Position übergeben, die das blockierende Auto einnehmen muss, um das ursprünglich blockierte Auto freizugeben. Also wird das beschriebene Verfahren wieder auf das Auto, das die erste Verschiebung blockiert angewendet. Sollte dieses selbst blockiert sein, wird es selbst wiederum prüfen, welche Verschiebungen nötig sind, um die benötigte Position freizugeben. Dies wird so lange wiederholt, bis eine oder keine Abfolge von Schritten gefunden wurde. Sowohl die Verschiebung nach rechts als auch nach links werden so geprüft und die Abfolge mit der niedrigeren Anzahl an Verschiebungen wird als Lösung abgespeichert. Bei der Ausgabe wird neben der Lösung pro Position auch eine Visualisierung des Parkplatzes ausgegeben.

Beispiele

Zuerst einmal die offiziellen Beispiele:

```
$ python3 aufgabe1/aufgabe1.py Beispiele/a1-
Schiebeparkplatz/beispieldaten/parkplatz0.txt
A B C D E F G
    H H   I I

A:
B:
C: H 1 rechts
D: H 1 links
E:
F: H 1 links, I 2 links
G: I 1 links
```

```
$ python3 aufgabe1/aufgabe1.py Beispiele/a1-
Schiebeparkplatz/beispieldaten/parkplatz1.txt
A B C D E F G H I J K L M N
  O O P P   Q Q     R R

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:
```

```
$ python3 aufgabe1/aufgabe1.py Beispiele/a1-
Schiebeparkplatz/beispieldaten/parkplatz2.txt
A B C D E F G H I J K L M N
    O O   P P Q Q R R   S S

A:
B:
C: O 1 rechts
D: O 1 links
E:
F: O 1 links, P 2 links
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: P 1 links, Q 1 links, R 1 links, S 2 links
N: S 1 links
```

```
$ python3 aufgabe1/aufgabe1.py Beispiele/a1-
Schiebeparkplatz/beispieldaten/parkplatz3.txt
```

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

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 2 links, R 2 links
L: Q 1 links, R 1 links
M: Q 2 links, R 2 links, S 2 links
N: Q 1 links, R 1 links, S 1 links
```

```
$ python3 aufgabe1/aufgabe1.py Beispiele/a1-
Schiebeparkplatz/beispieldaten/parkplatz4.txt
A B C D E F G H I J K L M N O P
Q Q R R     S S     T T   U U

A: R 1 rechts, Q 1 rechts
B: R 2 rechts, Q 2 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:
```

```
$ python3 aufgabe1/aufgabe1.py Beispiele/a1-
Schiebeparkplatz/beispieldaten/parkplatz5.txt
A B C D E F G H I J K L M N O
    P P Q Q     R R     S S

A:
B:
C: P 2 links
D: P 1 links
E: Q 1 rechts
F: Q 2 rechts
G:
H:
I: R 1 rechts
J: R 1 links
K:
```

```
L:
M: S 1 rechts
N: S 1 links
O:
```

Ein viel einfacheres Beispiel, wie ich finde, ist jedoch dieses:

```
A E
2
F 0
G 2
```

```
$ python3 aufgabe1/aufgabe1.py Beispiele/a1-
Schiebeparkplatz/beispieldaten/parkplatz6.txt
A B C D E
F F G G

A: G 1 rechts, F 1 rechts
B: nicht lösbar
C: G 1 rechts
D: nicht lösbar
E:
```

Es zeigt die verkettete Verschiebung und auch den Fall, dass keine Lösung vorhanden sind sowie dass eine Verschiebung ausreicht oder gar keine benötigt wird. Das kann man dann natürlich beliebig komplex machen, z.B.:

```
A G
3
H 0
I 2
J 4
```

```
$ python3 aufgabe1/aufgabe1.py Beispiele/a1-
Schiebeparkplatz/beispieldaten/parkplatz7.txt
A B C D E F G
H H I I J J

A: J 1 rechts, I 1 rechts, H 1 rechts
B: nicht lösbar
C: J 1 rechts, I 1 rechts
D: nicht lösbar
E: J 1 rechts
F: nicht lösbar
G:
```

Quelltext

```
class ParkingLot:
    """Ein Parkplatzszenario"""

    def __init__(self, path: str) -> None:
```

```
        # Initialisierung aller Attribute
        self.normal_cars = []
        self.sideways_cars = []
        self._blocked_spots = {}
        with open(path, "r") as file:
            max_letter = file.readline()[2]
            for normal_car_letter in letters[:get_index_of_letter(max_letter) +
1]:
                self.normal_cars.append(NormalCar(self, normal_car_letter))
            for sideways_car in range(int(file.readline())):
                sideways_car_letter, position = file.readline().split()
                self.sideways_cars.append(SideWaysCar(self, sideways_car_letter,
int(position)))

    def __str__(self) -> str:
        # Formatiert die Lösung anschaulich
        solution = self.solution
        output = self.visual()
        for slot in solution:
            if solution[slot] is False:
                output += slot + ": nicht lösbar\n"
            else:
                output += slot + ": "
                for index, item in enumerate(reversed(solution[slot].items())):
                    blocking_car, move = item
                    if move > 0:
                        direction = "rechts"
                    else:
                        direction = "links"
                    output += blocking_car + " " + str(abs(move)) + " " +
direction
                    if index != len(solution[slot]) - 1:
                        output += ", "
                output += "\n"
        return output

    @property
    def blocked_spots(self) -> list:
        """Getter-Methode für `_blocked_spots`, fasst alle blockierten Positionen
in einer `list` zusammen

        :return: Liste aller blockierten Positionen
        """
        return sum(list(self._blocked_spots.values()), [])

    def update_blocked_spots(self, car=None) -> None:
        """Aktualisiert die interne Repräsentation der seitwärts gerichteten
Autos

        :param car: das Auto, das aktualisiert werden soll
        """
        if car is None:
            for blocking_car in self._blocked_spots:
                self._blocked_spots[blocking_car] = [blocking_car.position,
blocking_car.position + 1]
        elif car.position in self.blocked_spots and self.find_blocking_car(
                car.position) != car or car.position + 1 in self.blocked_spots
and self.find_blocking_car(
```

```
            car.position + 1) != car:
                raise ValueError("Dieser Platz ist bereits belegt.")
        else:
            self._blocked_spots[car] = [car.position, car.position + 1]

    def find_blocking_car(self, position: int):
        """Findet zu einer gegebenen Position das dort stehende seitwärts
gerichtete Auto

        :param position: die Position
        :return: ein `SidewaysCar`
        """
        return self.sideways_cars[int(self.blocked_spots.index(position) / 2)]

    @property
    def solution(self) -> dict:
        """Getter-Methode, die die Lösung generiert

        :return: `dict` mit Lösungsschritten
        """
        solution_dict = {}
        for normal_car in self.normal_cars:
            if not normal_car.is_movable:
                solution_dict[normal_car.letter] =
self.find_solution(normal_car.position)
            else:
                solution_dict[normal_car.letter] = {}
        return solution_dict

    def find_solution(self, position: int, steps: ChainMap = None) -> Union[dict,
bool]:
        """Rekursive Methode, die für eine gegebene Position so lange Autos
verschiebt, bis eine Lösung gefunden wurde

        :param position: die Position auf dem Parkplatz
        :param steps: die bisher gesammelten Schritte
        :return: `dict` falls es eine Lösung gibt, sonst `False`
        """
        # Initalisierung der Schritte
        if steps is None:
            steps = ChainMap()
        # Falls die gesuchte Position außerhalb der vorhandenen liegt oder das
blockierende Auto bereits verschoben
        # wurde, kann keine Lösung gefunden werden
        if position not in range(len(self.normal_cars)) or position in
self.blocked_spots and self.find_blocking_car(
                position).letter in steps.keys():
            return False
        else:
            # Kopie des Parkplatzes, um die ursprünglichen Positionen nicht zu
ändern
            parking_lot_copy = dc(self)
            blocking_car = parking_lot_copy.find_blocking_car(position)
            position_difference = position - blocking_car.position
            # Zuerst wird versucht, das blockierende Auto um die kürzeste Distanz
zu verschieben
            if position_difference == 1:
                disposition = -1
```

```
            else:
                disposition = 1
            # Kann es dorthin nicht verschoben werden, wird versucht, es in die
andere Richtung zu verschieben
            if not blocking_car.is_movable(disposition):
                if disposition == -1:
                    disposition = 2
                else:
                    disposition = -2
            # Kann es in eine der beiden Richtungen nun verschoben werden, kann
diese Verschiebung als Lösung
            # zurückgegeben werden
            if blocking_car.is_movable(disposition):
                blocking_car.position += disposition
                return dict(steps.new_child({blocking_car.letter: disposition}))
            # Ist das nicht der Fall, wird rekursiv die kürzeste Abfolge von
Verschiebungen gesucht, die die gewünschte
            # ursprüngliche Verschiebung erlaubt
            else:
                steps = steps.new_child({blocking_car.letter: 0})
                # Dazu ruft sich die Methode innerhalb des kopierten Parkplatzes
mit dem Argument der Verschiebung
                # um eine Position nach rechts und links selbst auf
                steps_right =
parking_lot_copy.find_solution(blocking_car.position + 2 + position_difference,
dc(steps))
                steps_left = parking_lot_copy.find_solution(blocking_car.position
- 2 + position_difference, dc(steps))
                # Geben beide keine Lösung zurück, gibt es keine
                if not steps_left and not steps_right:
                    return False
                # Geben beide eine zurück, wird die Anzahl der Schritte
verglichen und anhand dessen entschieden
                elif steps_left and steps_right:
                    amount_steps_left = sum(steps_left.values(), -2 +
position_difference)
                    amount_steps_right = sum(steps_right.values(), 1 +
position_difference)
                    if abs(amount_steps_left) <= amount_steps_right:
                        steps = steps_left
                        steps[blocking_car.letter] = - 2 + position_difference
                    else:
                        steps = steps_right
                        steps[blocking_car.letter] = 1 + position_difference
                # Gibt nur eine der beiden eine Lösung zurück, muss diese gewählt
werden
                elif steps_left:
                    steps = steps_left
                    steps[blocking_car.letter] = -2 + position_difference
                else:
                    steps = steps_right
                    steps[blocking_car.letter] = 1 + position_difference
                return dict(steps)

class Car:
    """Basisklasse für die Autos auf dem Parkplatz"""
    def __init__(self, parent_lot: ParkingLot, letter: str, position: int) ->
None:
```

```
        self.letter = letter
        self.parent_lot = parent_lot
        self._position = position

    @property
    def position(self) -> int:
        return self._position


class NormalCar(Car):
    """Klasse für normal geparkte Autos, erbt von `Car`"""
    def __init__(self, parent_lot: ParkingLot, letter: str) -> None:
        super().__init__(parent_lot, letter, get_index_of_letter(letter))

    @property
    def is_movable(self) -> bool:
        if self.position in self.parent_lot.blocked_spots:
            return False
        else:
            return True


class SideWaysCar(Car):
    """Klasse für seitwärts geparkte Autos, erbt von `Car`"""
    def __init__(self, parent_lot: ParkingLot, letter: str, position: int) ->
None:
        super().__init__(parent_lot, letter, position)
        self.parent_lot.update_blocked_spots(self)

    def is_movable(self, delta: int) -> bool:
        # Da die Position der seitlichen Autos lediglich die linke der beiden
eingenommenen Positionen beschreibt,
        # muss dafür bei Verschiebungen nach rechts korrigiert werden
        if delta > 0:
            delta += 1
        if self.position + delta in self.parent_lot.blocked_spots or
self.position + delta \
                not in range(0, len(self.parent_lot.normal_cars)):
            return False
        else:
            return True

    @Car.position.setter
    def position(self, position: int) -> None:
        self._position = position
        # Bei jeder Positionsänderung wird auch die Repräsentation des
Parkplatzes geändert
        self.parent_lot.update_blocked_spots(self)
```