Aufgabe 2: Vollgeladen

Team-ID: 00443

Team: !binary

Bearbeiter/-innen dieser Aufgabe:
Tobias Maurer

7. November 2021

Inhaltsverzeichnis

Lösungsidee

Die Aufgabenstellung fordert, dass Auswählen der Hotels nach deren Bewertungen. Die kleinste Bewertung soll so hoch sein wie möglich, um das Spiel zu erreichen. Dazu sortiert man die existierenden Bewertungen der Datei nach Größe absteigend (5 → 0 in 0,1er Schritten). Man probiert als Erstes nur Hotels mit der Top 1 Bewertung (z. B. 5) zu wählen. Sofern man damit das Ziel nicht erreichen kann, erlaubt man zusätzlich die Top 2 Bewertung (z. B. 4,9) bei den Hotels. Wenn damit das Zeil nicht zu erreichen ist, erlaubt man zusätzlich die Top 3 Bewertung (z. B. 4,7, da 4,8 nicht in den Bewertungen der Hotels vorkommt) und so weiter und so fort. Um jetzt aber auf jeden Fall ans Ziel zu kommen, wählt man immer das am weitesten entfernte Hotel, welches vom momentanen Hotel (oder Start) erreichbar ist. Sofern das Ziel vom zuletzt ausgewähltem Hotel erreichbar ist, gibt man diese Hotels an.

Umsetzung

Das Programm liest die gegebene Datei ein und erstellt eine Liste *h* mit allen Hotels. Jedes Hotel hat die Attribute *time* (Entfernung vom Startpunkt), *rating* (=Bewertung) und *name* (=“Hotel [Zeilennummer]“). Nun iteriert man über die Liste der Hotels und erstellt dabei eine Liste *b*, welches alle vorkommenden Bewertungen der Hotels beinhaltet und **absteigend** sortiert ist. Eine Bewertung wird nur hinzugefügt, solange diese nicht bereits in der Liste vorkommt. Als nächstes sortiert man die Liste *b* absteigend (von 5,0 zu 0,0). Als letzten Schritt der Initialisierung der nicht

mehr ändernden Werte für die gegeben Datei muss noch die gesamte Fahrzeit in einer Variable *t* gespeichert werden, um zu wissen, wann das Ziel erreicht wurde.

Der Hauptteil des Lösungsalgorithmus besteht aus einer zwei dimensionalen Schleifenstruktur. Die äußerer Schleife ist eine for-Schleife, welche die **Indexe** der Liste *b* durchgeht, und damit alle Variablen für die innere while-Schleife, welche den zu fahrenden Weg berechnet, festlegt.

In der for-Schleife wird die Position des Autos auf 0 gesetzt, die Liste *l* mit den Hotels, die befahren werden, auf eine leere Liste gesetzt und die Teilmenge der Bewertungen. Die Teilmenge der Bewertung bekommt man, indem man aus der Liste *b* von dem Index 0 an alle Elemente bis zum **Index**, der in der Zählvariable der for-Schleife steht. In Python gibt es für Listen die „slice“-Funktion. Das heißt man gibt in den eckigen Klammern zwei Indexe an, welche man mit einem Doppelpunkt trennt. Z. B. „liste[0:2]“ gibt in einer Liste die Elemente an den Indexen 0 und 1 zurück, sofern die Liste *liste* mindestens 2 Elemente hat. Das Element an index 2 wird nicht mit übergeben. Aus diesem Grund muss man den Endindex immer noch um eins erhöhen, um wirklich die Teilmenge der Bewertungen zu bekommen, die erlaubt sind (*liste*[0:0] würde nämlich eine leere Liste zurückgeben).

Als nächstes berechnet man den Weg mit der while-Schleife. In der while-Schleife berechnet man alle Hotels, die von der momentanen Position aus erreichbar sind. Als nächstes filtert man alle Hotels heraus, die die Anforderung erfüllen. Unter Anforderung wird verstanden, dass die Bewertung des Hotels in der Teilliste, die mit der „slice“-Funktion erstellt wird, vorkommt. Nun sortiert man die gefilterte Liste anhand der „time“ der Hotels, ebenfalls absteigend. Somit ist garantiert, dass das erste Hotel in der Liste das Hotel ist, welches das Auto am weitesten Richtung Ziel bringt. Nun fügt man dieses Hotel noch in die Liste *l* ein und aktualisiert die Position des Autos auf die „time“ von Hotel, da in dieser Aufgabe die Zeit äquivalent zur Strecke ist. Diese Schritte wiederholt man solange bis die Schleife abgebrochen wird. Dazu gibt es 3 Möglichkeiten:

1. Es werden keine Hotels gefunden, welche die Anforderung erfüllen und erreichbar sind, von der Position des Autos aus innerhalb des sechsstündigen Fahrintervalls. In diesem Fall bricht man die Wegsuche ab und geht in die nächsten Schleifendurchgang der for-Schleife. Dadurch werden die Forderungen immer lockerer.

2. Das Ziel kann nicht erreicht werden mit den Hotels, die die Anforderungen erfüllen, aber es gibt keine Nacht mehr zu vergeben. Wenn man zum Beispiel plant in 5 Tagen zu fahren, gibt es 4 Nächte. Das heißt, dass man nur 4 Hotels, 1 für jede Nacht, auswählen kann. Wenn man nun aber vom viertem Hotel nicht das Ziel erreicht, gibt es keine Lösung da es keine 5 Nacht gibt, die eingeplant ist. Zudem ist garantiert, das die Hotels immer den weitesten Weg vom Start aus zurücklegen, da Dieser Fall wird gehandhabt wie der Fall 1.

3. Man erreicht vom zuletzt hinzugefügtem Hotel das Ziel. In diesem Fall gibt man die Liste *l*, die die ausgewählten Hotels enthält, zurück als Lösung und alle Schleifen werden beendet, da eine Lösung gefunden wurde.

Es kann ebenfalls passieren, dass die for-Schleife erfolgreich beendet wird. In diesem Fall wurden alle Minima-Anforderungsmöglichkeiten überprüft und kein Weg gefunden. Somit weiß man, dass es keine Lösung geben kann.

Nun fehlt nur noch die Ausgabe, der zurückgegeben Liste in die Konsole/Datei oder was auch immer erwünscht ist.

Beispiele

hotels1.txt

```
Hotel(name='Hotel 5', time=347, rating=2.7)
diff: 340
Hotel(name='Hotel 9', time=687, rating=4.4)
diff: 320
Hotel(name='Hotel 10', time=1007, rating=2.8)
diff: 353
Hotel(name='Hotel 13', time=1360, rating=2.8)
left: 320
Total time: 1680
```

Diese Ausgabe zeigt, dass das Ziel mit den Hotels aus den Zeilen 5, 9, 10, 13 erreicht wird. Die minimal verwendete Bewertung ist 2,7. Außerdem zeigt die Differenz zwischen den Hotels, dass das Fahrinterval von 360min nie überschritten wird. Am fünftem Tag müssen Lara und co. noch 320 Minuten fahren. Dies zeigt der Wert bei „left“. Aufgrund der wenig angegebenen Hotels sinkt die bestmögliche Bewertung sehr schnell, da es tendenziell viele verschieden Bewertungen auch im niedrigen Bereich gibt.

hotels2.txt

```
Hotel(name='Hotel 5', time=341, rating=2.3)
diff: 359
Hotel(name='Hotel 12', time=700, rating=3.0)
diff: 353
Hotel(name='Hotel 17', time=1053, rating=4.8)
diff: 327
```

```
Hotel(name='Hotel 27', time=1380, rating=5.0)
left:  357
Total time: 1737
```

Da am Ende in der gegebenen Datei viele niedrige Bewertungen sind, ist es schnell klar, dass die Lösung niedrige Bewertungen beinhaltet, da es diese erlauben muss, um das Ziel zu erreichen.

hotels3.txt

```
Hotel(name='Hotel 102', time=360, rating=1.3)
diff: 357
Hotel(name='Hotel 198', time=717, rating=0.3)
diff: 359
Hotel(name='Hotel 300', time=1076, rating=3.8)
diff: 357
Hotel(name='Hotel 403', time=1433, rating=1.7)
left:  360
Total time: 1793
```

hotels4.txt

```
Hotel(name='Hotel 99', time=340, rating=4.6)
diff: 336
Hotel(name='Hotel 214', time=676, rating=4.6)
diff: 356
Hotel(name='Hotel 334', time=1032, rating=4.9)
diff: 284
Hotel(name='Hotel 436', time=1316, rating=4.9)
left:  194
Total time: 1510
```

hotels5.txt

```
Hotel(name='Hotel 287', time=317, rating=5.0)
diff: 319
Hotel(name='Hotel 583', time=636, rating=5.0)
diff: 351
Hotel(name='Hotel 915', time=987, rating=5.0)
diff: 299
Hotel(name='Hotel 1180', time=1286, rating=5.0)
left:  330
Total time: 1616
```

Bei einer großen Menge an guten Bewertungen, die gut in der Datei verteilt sind, ist es sehr wahrscheinlich, dass es eine Lösung mit einer sehr „strengen“ Anforderung gibt.

Quellcode

Klasse die ein Hotel aus der Datei represäntiert

```python
class Hotel(object):

    """
    Stellt ein Hotel aus der Datei da
    """

    def __init__(self, name: str, time: int, rating: float):
        self.time = time
        self.rating = rating
        self.name = name

    def __repr__(self) -> str:
        return "Hotel(name=%r, time=%s, rating=%s)" % (self.name, self.time, self.rating)
```

Herausfiltern der Bewertungen aus der gegebenen Datei anhand der eingelesenen Hotels

```python
def get_ratings(hotels: List[Hotel]) -> List[float]:
    """Gibt eine Liste der verschiedene Bewertungen und sortiert diese von
       gut zu schlecht.

    Args:
        hotels (List[Hotel]): die Liste der Hotels auf der Strecke

    Returns:
        List[float]: die bewertungen sortiert von gut zu schlecht
    """
    ratings = []

    for h in hotels:
        if h.rating not in ratings:
            ratings.append(h.rating)

    ratings.sort(reverse=True)  # good to bad
    return ratings
```

Berechnen, welche Hotels erreicht werden können vom Standpunkt aus

```python
def get_next_hotels(hotels: List[Hotel], position: int, interval: int = 6 * 60) -> List[Hotel]:
    """Gibt die Hotels die erreichbar sind zurück und sortiert diese nach 1. deren Bewertung
       und 2. deren Zeit/Entfernung vom Startpunkt

    Args:
        hotels (List[Hotel]): Die Hotels auf der Strecke oder die in der "Zukunft" kommenden
        position (int): wie lange/weit bereits gefahren wurde
        interval (int): die maximale Länge (Zeit oder Weg) einer Fahrt

    Returns:
        List[Hotel]: die Liste der erreichbaren Hotels
    """
    return list(
        sorted(
            filter(lambda h: h.time > position and h.time <= position + interval, hotels),
            key=lambda h: (h.rating, h.time),
            reverse=True
        )
    )
```

Die Funktion, welche letzendlich alles löst

```python
def solve(hotels: List[Hotel], total_time: int, days: int = 5, interval: int = 6 * 60) -> List[Hotel]:
    """Die Implementierung der Lösung

    Args:
        hotels (List[Hotel]): die Hotels auf dem Weg
        days (int): Wie viele Fahrten gemacht werden. Z. B. 5 Tage = 5 Fahrten = 4 Nächte
        interval (int): Die Fahrzeit pro Fahrt

    Returns:
        List[Hotel]: die Lösung (4 Hotels)
    """
    ratings = get_ratings(hotels)

    for i in range(len(ratings)):
        min_requirement = ratings[:i + 1]
        position = 0
        history: List[Hotel] = [] # Bsp: nur 4 Übernachtungen bei 5 Tagen

        while len(history) < days - 1:  # Abbruchfall 2
            next_hotels = get_next_hotels(hotels, position, interval)
            next_hotels = [h for h in next_hotels if h.rating in min_requirement]
            next_hotels.sort(key=lambda h: h.time, reverse=True)  # Entfernung (weit -> klein) von position

            # Abbruchfall 1
            if not next_hotels:
                # keine Lösung gefunden, da es kein Hotel mehr gibt zu dem sie fahren können und sie können von
                # ihrem momentanem Standpunkt nicht das Ziel erreichen
                break

            history.append(next_hotels[0])
            position = history[-1].time

            # Abbruchfall 3
            if position + interval >= total_time:
                return history  # Lösung gefunden

    return []  # No solution
```

main-function/Gefüge und Ausgabe der Lösung

```python
def main():
    total_time, hotels  = read_file("a2input.txt")
    res = solve(hotels, total_time)


    if res:
        prev = None
        for h in res:

            if prev is not None:
                print("diff:", h.time - prev.time)

            print(h)
            prev = h

        print("left: ", total_time - prev.time)  # type: ignore
    else:
        print("No solution")

    print("Total time:", total_time)
```