Aufgabe 2

Lösungsidee

Hier hatte ich die Idee, alle (möglichen) Kombinationen durchzuspielen und aus diesen dann die herauszupicken, die die höchste minimale Bewertung hat.

Umsetzung

Das habe ich erreicht, indem ich eine Klasse `Route` angelegt habe, welche alle Hotels in der Eigenschaft `hotels` abspeichert. Außerdem hat sie die Attribute `min_rating`, welches später relevant wird und `total_distance`, welche die Gesamtdistanz enthält. Die Hotels werden als Objekte vom Typ `Hotel` in der Liste gespeichert, welche die Attribute `distance`, `rating` und `parent_route` enthalten. Sie repräsentieren die Entfernung vom Start, die Bewertung und eine Referenz zum Elternobjekt vom Typ `Route`. Außerdem enthält jedes `Hotel` die `daily_range`, welche die Liste an Hotels darstellt, die innerhalb eines Tages (also 360 Minuten) erreichbar sind. Um nun die optimale Folge von Hotels zu ermitteln, wird jede Kombination von Hotels so lange verfolgt, bis das Ziel entweder schon vorher nicht mehr innerhalb der vorgegebenen Zeit erreichbar sein wird (da mehr Strecke zurückgelegt werden müsste als an den verbleibenden Tagen möglich) oder die Route nicht besser sein kann als die bisher als beste Lösung festgestellte, da ein Hotel vorkommt, dessen Bewertung niedriger ist als die niedrigste der bisherigen Lösung (`min_rating`), oder bis eben eine Lösung innerhalb von vier Übernachtungen gefunden wurde. Ist bisher keine Lösung vorhanden oder ist die neue Lösung besser als die bisherige, wird die alte Lösung überschrieben. Zu Beginn hatte ich aufgrund der hohen Menge an Hotels in den Beispielen 4 und 5 extrem lange Berechnungszeiten, da ich jede einzelne Kombination vier Tage tief verfolgt habe und meine Methode `range_per_day` `list comprehension` verwendet hat. Die Ersetzung der `list comprehension` durch ein `filter`-Objekt hat bei der Profilierung mit `cProfile` zu unglaublichen Verbesserungen der Berechnungszeit geführt, von minutenlangen Prozessen zu Bruchteilen einer Sekunde. Weitere Schritte zur Beschleunigung waren die Speicherung der `daily_range` in den `Hotel`-Objekten statt Neuberechnung bei jedem Aufruf und die Einführung des Vergleichs mit `min_rating`.

Beispiele

Hier die Anwendung auf die Beispiele

```
$ python3 aufgabe2/aufgabe2.py Beispiele/a2-Vollgeladen/beispieldaten/hotels1.txt
1. Position: 347, Bewertung: 2.7
2. Position: 687, Bewertung: 4.4
3. Position: 1007, Bewertung: 2.8
4. Position: 1360, Bewertung: 2.8
```

```
$ python3 aufgabe2/aufgabe2.py Beispiele/a2-Vollgeladen/beispieldaten/hotels2.txt
1. Position: 340, Bewertung: 1.6
2. Position: 700, Bewertung: 3.0
3. Position: 1051, Bewertung: 2.3
4. Position: 1377, Bewertung: 1.8
```

```
$ python3 aufgabe2/aufgabe2.py Beispiele/a2-Vollgeladen/beispieldaten/hotels3.txt
1. Position: 358, Bewertung: 2.5
2. Position: 717, Bewertung: 0.3
3. Position: 1075, Bewertung: 0.8
4. Position: 1433, Bewertung: 1.7
```

```
$ python3 aufgabe2/aufgabe2.py Beispiele/a2-Vollgeladen/beispieldaten/hotels4.txt
1. Position: 340, Bewertung: 4.6
2. Position: 658, Bewertung: 4.6
3. Position: 979, Bewertung: 4.7
4. Position: 1301, Bewertung: 5.0
```

```
$ python3 aufgabe2/aufgabe2.py Beispiele/a2-Vollgeladen/beispieldaten/hotels5.txt
1. Position: 280, Bewertung: 5.0
2. Position: 636, Bewertung: 5.0
3. Position: 987, Bewertung: 5.0
4. Position: 1271, Bewertung: 5.0
```

Quelltext

```
class Hotel:
    """Ein Hotel mit den Attributen `distance`, `rating` und dem zugehörigen
`Route`-Objekt"""

    def __init__(self, distance: int, rating: float, parent_route):
        self.distance = distance
        self.rating = rating
        self.parent_route = parent_route
        self._daily_range = None

    def __repr__(self) -> str:
        return "Hotel({}, {})".format(self.distance, self.rating)

    def __str__(self) -> str:
        return "Position: {}, Bewertung: {}".format(self.distance, self.rating)

    def __lt__(self, other) -> bool:
        # Implementierung von __lt__ ermöglicht den "kleiner-als" Vergleich
zwischen zwei Hotel-Objekten
        assert type(other) == Hotel
        return self.rating < other.rating

    @property
    def daily_range(self) -> list:
        """Stellt die Liste von Hotels dar, die innerhalb eines Tages erreicht
werden können"""
```

```
        if not self._daily_range:
            self._daily_range = self.parent_route.range_per_day(self)
        return self._daily_range


class Route:

    def __init__(self, path: str) -> None:
        # Initialisierung aller Attribute
        self.min_rating = None
        self.hotels = []
        with open(path, "r") as file:
            hotel_amount = int(file.readline())
            self.total_distance = int(file.readline())
            for hotel in range(hotel_amount):
                distance, rating = file.readline().split(" ")
                self.hotels.append(Hotel(int(distance), float(rating), self))

    def __str__(self) -> str:
        # Formatiert die Lösung anschaulich
        possible_routes = self.find_route()
        if possible_routes:
            output = ""
            for index, hotel in enumerate(possible_routes):
                output += "{}. {}\n".format(index + 1, hotel)
        else:
            output = "Es wurde keine Lösung gefunden."
        return output

    def range_per_day(self, hotel: Hotel = None) -> filter:
        """Gibt für ein gegebenes Hotel `hotel` die verfügbaren Hotels zurück,
die innerhalb eines Tages erreicht
        werden können

        :param hotel: das Hotel
        :return: die Liste von Hotels als `filter`-Objekt
        """
        if not hotel:
            start_distance = 0
        else:
            start_distance = hotel.distance
        return filter(lambda h: start_distance < h.distance <= start_distance +
360, self.hotels)

    def find_route(self, hotel: Hotel = None, counter: int = 0, stops:
list[Hotel] = []) -> Union[bool, list[Hotel]]:
        """Findet rekursiv die optimale Route

        :param hotel: das Ausgangshotel
        :param counter: die Etappe der Route
        :param stops: die bisherigen Stopps
        :return: `False` falls keine Lösung vorhanden, sonst eine Liste von
Hotels
        """
        stops = stops.copy()
        # Falls kein Hotel übergeben wird, befinden wir uns am Anfang der Route
        if hotel is None:
            daily_range = self.range_per_day()
```

```
        # sonst kann dieser Pfad in bestimmten Fällen bereits abgebrochen oder
zurückgegeben werden
        else:
            # zB falls das eigene Rating niedriger ist als das niedrigste der
aktuellen Lösung
            if self.min_rating and hotel.rating <= self.min_rating:
                return False
            daily_range = hotel.daily_range
            stops.append(hotel)
            # Haben wir innerhalb von 5 Tagen das Ziel erreicht, kann die Abfolge
von Hotels zurückgegeben werden
            if counter <= 4 and self.total_distance - 360 <= hotel.distance:
                return stops
            # Ist es von diesem Hotel aus nicht mehr möglich, das Ziel innerhalb
der Zeit zu erreichen,wird abgebrochen
            elif counter > 4 or counter <= 4 and self.total_distance - 360 * (5 -
counter) > hotel.distance:
                return False
        solution = []
        # Jedes Hotel, das innerhalb eines Tages erreichbar ist, wird als
nächster Schritt geprüft
        for stop in daily_range:
            possible_route = self.find_route(stop, counter + 1, stops)
            if possible_route:
                # Wird eine Lösung gefunden und es gibt bereits eine vorherige,
überprüfe die beiden auf ihre minimale
                # Bewertung
                if solution:
                    solution = max(solution, possible_route.copy(), key=lambda r:
min(r))
                    self.min_rating = min(solution).rating
                # Wird eine Lösung gefunden und es gibt bisher noch keine, setze
sie als Lösung
                if not solution:
                    solution = possible_route.copy()
                    self.min_rating = min(solution).rating
        # Gib die Lösung aus, sofern es eine gibt
        if solution:
            return solution
        else:
            return False
```