

Aufgabe 5: Stadtführung

Team-ID: 00879

Bearbeiter/-in dieser Aufgabe:
Karl Zschiebsch

16. November 2023

Inhaltsverzeichnis

Lösungsidee.....	1
Umsetzung.....	1
Beispiele.....	2
Quellcode.....	2

Lösungsidee

Ziel ist, die Tourlänge zu kürzen und dabei essentielle Tourpunkte zu behalten und gleichzeitig die Tour streng chronologisch zu belassen. Für die Lösung dieses Problems nehme ich dadurch an, dass die Anzahl der Tourpunkte nicht zwingend aufs Minimum reduziert werden muss, sondern nur die Länge der Tour optimiert werden soll.

Statt die Tour als ganzes zu optimieren, wird die Tour in Teilsequenzen unterteilt. Eine Teilsequenz wird darüber definiert, dass sie immer mit mindestens einem essentiellen Tourpunkt beginnt oder endet. Dies liegt daran, dass essentielle Tourpunkte nicht weggestrichen werden können und es somit nicht Sinnvoll ist, diese zu optimieren.

Zu Beginn wird der optimale Start- und Endpunkt gewählt. Falls in der ersten oder letzten Sequenz sich jedoch nur ein einziges Element befindet, wird dieser Schritt übersprungen, da bereits der optimale Start- oder Endpunkt vorhanden ist. Ansonsten werden zuerst alle Paare bestimmt, wo der Ort sowohl in der ersten als auch in der letzten Sequenz vorkommt. Die Distanz zwischen all diesen Orten wird bestimmt. Das Paar, wo die Differenz der Tourlänge bis zu diesem Punkt am geringsten ist, ist der optimale Start- und Endpunkt. Alle Tourpunkte, die vor dem neuen Startpunkt bzw. nach dem neuen Endpunkt liegen, werden umgehend entfernt.

Danach werden alle einzelnen Sequenzen optimiert. Dafür wird untersucht, ob es für jeden Ort einen anderen gibt, der unmittelbar an diesen angrenzt. Ist dies der Fall, werden alle Orte dazwischen aus der Tour entfernt.

Umsetzung

Die Klasse `TourPoint` stellt einen einzelnen Punkt in einer Tour dar. Sie hat die Funktion `is_local_equ`, die überprüft, ob zwei Tourpunkte lokal gleich sind, dh. den gleichen Ort repräsentieren.

Die `Tour`-Klasse repräsentiert die Gesamttour und enthält Methoden zur Optimierung der Tourlänge. Wenn sie erstellt wird, liest sie eine Datei ein um daraus die Tourpunkte zu bekommen. Die Tourpunkte werden alle in `points` gespeichert. Zusätzlich wird der Punkt der letzten Sequenz in `sequences` hinzugefügt. Sollte der Tourpunkt essentiell sein, startet er darüber hinaus eine neue Sequenz, dh. eine neue Liste mit dem essentiellen Tourpunkt wird den Sequenzen hinzu gefügt. Hinzu kommt, dass die Länge zwischen den Orten parallel berechnet wird. Die Länge wird in ein Dictionary eingetragen. Die Länge zwischen zwei Orten A und B berechnet, indem die gesamte Distanz der Tour, die bis dahin bei A und B zurück gelegt wurde, voneinander Abgezogen wird. Die Distanz wird sowohl bei A als auch bei B eingetragen. Konkret weist dieses Dictionary jedem `str` ein weiteres Dictionary zu (Name des Ortes zu verbundenen Orten), indem jedem `str` einem `int` zugewiesen wird (Verbundener Ort zu Distanz). `optimise` ist die Hauptmethode zur Optimierung der Tourlänge. Sie ruft `find_start` Methoden auf, um den optimalen Startpunkt zu finden und dann für jede Teilsequenz `shorter`, um diese Sequenz zu kürzen.

Beispiele

Lösung der ersten drei Beispielaufgaben von der Website. Alle Lösungen der Beispielaufgaben sind in `task.log` dokumentiert.

```
tour1.txt
Brauerei,1613,X,0
Karzer,1665,X,80
Rathaus,1678,X,150
Rathaus,1739,X,150
Euler-Brücke,1768, ,330
Fibonacci-Gaststätte,1820,X,360
Emmy-Noether-Campus,1912,X,450
Emmy-Noether-Campus,1998,X,450
Euler-Brücke,1999, ,580
Brauerei,2012, ,730

tour2.txt
Brauerei,1613, ,0
Karzer,1665,X,80
Rathaus,1739, ,150
Euler-Brücke,1768, ,330
Fibonacci-Gaststätte,1820,X,360
Emmy-Noether-Campus,1912,X,450
Emmy-Noether-Campus,1998,X,450
Euler-Brücke,1999, ,580
Brauerei,2012, ,730

tour3.txt
```

```
Talstation,1768,,0
Wäldle,1841,,520
Observatorium,1874,X,770
Piz Spitz,1898,,1240
Panoramasteg,1912,X,1460
Ziegenbrücke,1979,X,1710
Talstation,2005,,1990
```

Quellcode

Dies ist der Quellcode in Python 3.10. Es werden keine Bibliotheken benötigt.

```
class TourPoint:
    def __init__(self, name: str, year: int, essential: bool,
distance: int):
        self.name = name
        self.year = year
        self.essential = essential
        self.distance = distance

    def is_local_equ(self, another: 'TourPoint'):
        return self.name == another.name

class Tour:
    def __init__(self, path: str):
        with open(path, 'r') as reader:
            self.m = int(reader.readline())
            self.points = []
            self.sequences: list[list[TourPoint]] = [[]]
            self.distances: dict[str, dict[str, int]] = {}
            for i in range(self.m):
                args = reader.readline().split(',')
                name, year, essential, distance = args[0],
int(args[1]), args[2] == 'X', int(args[3])
                point = TourPoint(name, year, essential, distance)
                self.sequences[len(self.sequences) -
1].append(point)
                if name not in self.distances.keys():
                    self.distances[name] = {}
                if i > 0:
                    previous = self.points[i - 1].name
                    difference = distance - self.points[i -
1].distance
                    self.distances[name][previous] = difference
                    self.distances[previous][name] = difference
                if essential:
                    self.sequences.append([point])
                    self.points.append(point)

    def optimise(self):
        self.find_start()
        for sequence in self.sequences:
            self.shorter(sequence)
```

```

        self.reset_distances()

    def get_pairs(self) -> list[tuple[TourPoint, TourPoint]]:
        pairs = []
        for i in self.sequences[0]:
            for j in self.sequences[-1]:
                if i.is_local_equ(j):
                    pairs.append((i, j))
        if len(pairs) == 0:
            raise ValueError("No pairs")
        return pairs

    def find_start(self) -> None:
        pairs = self.get_pairs()
        index: int = ...
        distance: int = ...
        for i in range(len(pairs)):
            diff = pairs[i][1].distance - pairs[i][0].distance
            if distance is Ellipsis or diff < distance:
                distance = diff
                index = i
        if index is Ellipsis:
            raise IndexError("Index is Ellipsis")
        print(pairs[index])
        for v in self.sequences[0]:
            if v.index(pairs[index][0]):
                self.points.remove(v)
        for v in self.sequences[-1][self.sequences[-1].index(pairs[index][1]) + 1:]:
            if v.index(pairs[index][1]):
                self.points.remove(v)

    def shorter(self, sequence: list[TourPoint]):
        size = len(sequence)
        for i in range(0, size, 1):
            if sequence[i] in self.points:
                for j in range(i + 1, size, 1):
                    if sequence[j] in self.points and
                        self.are_connected(sequence[i],
sequence[j]):
                        for e in sequence[i+1:j]:
                            if e in self.points:
                                self.points.remove(e)

    def are_connected(self, left: TourPoint, right: TourPoint) ->
bool:
        if left.is_local_equ(right):
            return True
        return left.name in self.distances[right.name].keys()

    def get_distance(self, left: TourPoint, right: TourPoint) ->
int:
        if left.is_local_equ(right):
            return 0
        try:

```

```
        return self.distances[left.name][right.name]
    except KeyError:
        raise ValueError(f"{left} <-> {right}"
({self.are_connected(left, right)})")

    def reset_distances(self):
        self.points[0].distance = 0
        for i in range(1, len(self.points)):
            self.points[i].distance = self.points[i-1].distance +
                self.get_distance(self.points[i-1],
self.points[i])
```