Aufgabe 5: Stadtführung

Team-ID: 00250

pip install BWINF_42

Bearbeiter dieser Aufgabe: Irfan Topal (Teilnahme-ID: 71833)

20.11.2023

Inhaltsverzeichnis

1. Lösungsidee

Diese Aufgabe ist eine spezielle Variante des „Travelling Salesman Problems“. Das klassische TSP erfordert, dass der Geschäftsmann alle Städte auf der kürzest möglichen Route besucht und schließlich zum Ausgangspunkt zurückkehrt. Im vorliegenden Fall handelt es sich um eine Modifikation, bei der nicht alle Orte besucht werden müssen. Stattdessen soll eine optimale Teilroute gefunden werden, die die essentiellen Orte in chronologischer Reihenfolge abdeckt und gleichzeitig die Gesamtlänge minimiert.

Die Optimierung erfolgt, indem die ausgewählten Umwege aus der ursprünglichen Route entfernt werden. Man spart sich also den Umweg und besucht einen Ort nur einmal statt mehrmals. Dadurch verringert sich die Gesamtdistanz und die optimierte Route wird erstellt.
Die eigentliche Lösungsidee des Codes besteht darin, zunächst alle möglichen Umwege zu identifizieren, indem Orte gefunden werden, die während der Tour mehrfach besucht werden.

Diese Umwege werden nach der eingesparten Entfernung sortiert und diejenigen ausgewählt, die die Gesamtentfernung minimieren.

2. Umsetzung

Das Programm ist in 5 einfache Schritte unterteilt.

2.1 Schritt 1: Auslesen

Der erste Schritt ist das Auslesen der Tourdaten und ihre Speicherung in einem zweidimensionalen Array.

2.2 Schritt 2: Umwege finden (s. Quellcode 4.1)

In diesem Schritt geht es darum, alle möglichen Umwege in der Tour zu identifizieren. Der Algorithmus durchläuft die Liste der Tourpunkte und sucht nach Orten, die während der Tour mehrfach besucht werden. Dabei wird nicht berücksichtigt, ob sich die Umwege überschneiden, so dass z.B. zwei Umwege teilweise die gleichen Orte abdecken können. Abschließend wird die mögliche Distanzersparnis berechnet und zusammen mit dem Umweg gespeichert.

Der eigentliche Algorithmus beginnt z.B. am ersten Ort und durchläuft dann jeden Ort, der in der Zukunft liegt, bis eines der folgenden Ereignisse eintritt:

a) derselbe Ort kommt, oder
b) ein essentieller Tourenort (X) wird erreicht.

Wenn Ereignis a) eintritt, bedeutet dies, dass ein möglicher Umweg gefunden wurde (derselbe Ort wird mehrmals besucht). Es wird aber nicht an dieser Stelle angehalten und zum zweiten Ort gewechselt, sondern es wird weiter nach Umwegen gesucht, daher können hier z.B. Überlappungen auftreten.
Wenn Ereignis b) eintritt, bedeutet dies, dass der Ort nicht mehrfach besucht wird, bis ein essentieller Ort erreicht ist. Dann wird z.B. zum zweiten Ort gewechselt und der Algorithmus beginnt von vorne.

2.3 Schritt 3: Umwege filtern (s. Quellcode 4.2)

In diesem Schritt werden die Umwege gefiltert, um die Umwege auf mögliche Überlappungen zu prüfen und ggf. die sparsamste Route zu wählen.

Zuerst werden die Umwege nach der eingesparten Gesamtentfernung sortiert, so dass der sparsamste an erster Stelle steht. Dann wird eine Liste der bereits „besuchten“ Orte erstellt. Diese ist natürlich zunächst leer und wird mit dem folgenden Algorithmus gefüllt: Es wird jeder Umweg durchlaufen und geprüft, ob die jeweiligen Orte aus dem Umweg schon besucht wurden. Ist dies nicht der Fall, wird der Umweg „akzeptiert“ und zur weiteren Verarbeitung gespeichert. Die Orte aus dem Umweg werden dann bei den „bereits besuchten“ Orten gespeichert.

Da man die Umwege am Anfang nach Ersparnis sortiert hat, kann man bei Überschneidungen sicher sein, den effizienteren zuerst genommen zu haben.

2.4 Schritt 4: Neue Tour erstellen (s. Quellcode 4.3)

In diesem Schritt werden die im Schritt 3 filtrierte Umwege aus der ursprünglichen Tour entfernt. Dabei werden einfach die ganzen Orte, die in einem Umweg (zwischen Start- und Endpunkt) sind, in eine Liste gepackt. Dann wird der Ursprungstour durchgelaufen und die Distanzersparnisse abgezogen, dass die Entfernungen stimmen, wenn man die Umwege aus der Tour entfernt. Abschließend entfernt man wiegesagt die filtrierten Umwege aus der Ursprungstour und erhält die neue, effizientere Tour.

2.5 Schritt 5: Tour ausgeben

Als letzter Schritt wird der Tour in eine Datei ausgegeben. Ausgabe erfolgt in gleichem Stil wie der Eingangsdateien. Zuerst Anzahl Orte n und dann die jeweiligen Orte. Falls man einen Schlenker weggelassen hat, wird der Start- und Endpunkt direkt hintereinander ausgegeben (also gleicher Ort) mit der Entfernung von 0. Dabei wird der gesamt gesparten Distanz auch ausgegeben.

2.6 Verbesserungsmöglichkeiten

Bei meiner Umsetzung habe ich nicht auf mögliche Einsparungen durch die Wahl anderer Startorte geachtet, da ich der Meinung bin, dass man dadurch auf mögliche Umwege, die mehr Einsparungen bringen, verzichten muss. Außerdem wäre es im realen Leben interessant, den „ältesten“ Ort zu besuchen und nicht darauf zu verzichten. Aber natürlich könnte man einen Algorithmus schreiben, der die jeweiligen Ersparnisse durch Umwege oder andere Startorte vergleicht.

3. Beispiele

In den Beispielen wird zuerst die Konsolenausgabe und dann die Datei output.txt gezeigt.

3.1 tour1.txt

```
Success: Most efficient tour has been written to output.txt
Total saved distance: 1040
```

```
12
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
Schiefes Haus,1823, ,480
Theater,1880, ,610
Emmy-Noether-Campus,1912,X,740
Emmy-Noether-Campus,1998,X,740
Euler-Brücke,1999, ,870
Brauerei,2012, ,1020
```

3.2 tour2.txt

```
Success: Most efficient tour has been written to output.txt
Total saved distance: 1040
```

```
12
Brauerei,1613, ,0
Karzer,1665,X,80
Rathaus,1678, ,150
Rathaus,1739, ,150
Euler-Brücke,1768, ,330
Fibonacci-Gaststätte,1820,X,360
Schiefes Haus,1823, ,480
Theater,1880, ,610
Emmy-Noether-Campus,1912,X,740
Emmy-Noether-Campus,1998,X,740
Euler-Brücke,1999, ,870
Brauerei,2012, ,1020
```

3.3 tour3.txt

```
Success: Most efficient tour has been written to output.txt
Total saved distance: 1890
```

```
10
Talstation,1768, ,0
Wäldle,1805, ,520
Mittlere Alp,1823, ,1160
Observatorium,1833, ,1450
Observatorium,1874,X,1450
Piz Spitz,1898, ,1920
Panoramasteg,1912,X,2140
Panoramasteg,1952, ,2140
Ziegenbrücke,1979,X,2390
Talstation,2005, ,2670
```

3.4 tour4.txt

```
Success: Most efficient tour has been written to output.txt
Total saved distance: 1200
```

```
21
Blaues Pferd,1523, ,0
Alte Mühle,1544, ,110
Marktplatz,1549, ,210
Marktplatz,1562, ,210
Springbrunnen,1571, ,290
Dom,1596,X,360
Bogenschütze,1610, ,480
Bogenschütze,1683, ,480
Schnecke,1698,X,630
Fischweiher,1710, ,810
Reiterhof,1728,X,930
Schnecke,1742, ,1070
Schmiede,1765, ,1240
Große Gabel,1794, ,1350
Große Gabel,1874, ,1350
Fingerhut,1917,X,1420
Stadion,1934, ,1540
Marktplatz,1962, ,1630
Baumschule,1974, ,1710
Polizeipräsidium,1991, ,1850
Blaues Pferd,2004, ,2000
```

3.5 tour5.txt

```
Success: Most efficient tour has been written to output.txt
Total saved distance: 2330
```

```
28
Gabelhaus,1638, ,0
Gabelhaus,1699, ,0
Hexentanzplatz,1703,X,160
Eselsbrücke,1711, ,280
Dreibannstein,1724, ,390
Dreibannstein,1752, ,390
Schmetterling,1760,X,540
Dreibannstein,1781, ,620
Märchenwald,1793,X,700
Fuchsbau,1811, ,780
Torfmoor,1817, ,890
Gartenschau,1825, ,1030
Gartenschau,1898, ,1030
Riesenrad,1902, ,1150
Dreibannstein,1911,X,1280
Olympisches Dorf,1924, ,1440
Haus der Zukunft,1927,X,1570
Stellwerk,1931, ,1690
Stellwerk,1942, ,1690
Labyrinth,1955, ,1900
Gauklerstadl,1961, ,1980
Planetarium,1971,X,2060
Känguruhfarm,1976, ,2110
Balzplatz,1978, ,2190
Dreibannstein,1998,X,2280
Labyrinth,2013, ,2410
CO2-Speicher,2022, ,2600
Gabelhaus,2023, ,2670
```

3.6 Eigenes Beispiel

Mit folgendem Input

```
8
Bahnhof,1500, ,0
Marktplatz,1550, ,150
Schnecke,1600, ,200
Bahnhof,1650, ,225
Reiterhof,1700, , 375
Marktplatz,1750, ,450
Dom,1800,X,500
Bahnhof,1850, ,550
```

gibt es zwei mögliche, sich überlappende Umwege, nämlich

I. Bahnhof,1500 - Bahnhof,1650

II. Marktplatz,1550 - Marktplatz,1750

Der erste Umweg spart nur 225 Einheiten ein, der zweite dagegen 300. Es wird erwartet, dass das Programm den zweiten Umweg wählt und diese entfernt, weil man dann schneller zum Ziel kommt.

Ausgabe des Programms:

```
Success: Most efficient tour has been written to output.txt
Total saved distance: 300
```

```
5
Bahnhof,1500, ,0
Marktplatz,1550, ,150
Marktplatz,1750, ,150
Dom,1800,X,200
Bahnhof,1850, ,250
```

4. Wichtige Teile des Codes

4.1 find_detours(n, places)

```
# find every possible detour
def find_detours(n, places):
    possible_detours = []
    for i, current_place in enumerate(places):
        for j in range(i + 1, n):
            other_place = places[j]

            # compare every place coming after the current one until the place
name is same
            if other_place[0] == current_place[0]:
                # add the possible detour to an array and save the start, end
and possible saved distance
                possible_detours.append([i, j, other_place[3] -
current_place[3]])

                # if the end of the detour is an essential place then stop
searching for more detours
                if other_place[2] == "X":
                    break

            # if there is an essential place coming up before a possible
detour, then break
            elif other_place[2] == "X":
                break

    return possible_detours
```

4.2 filter_detours(possible_detours)

```
# filter out detours with overlapping time periods to get the largest saved
distance
def filter_detours(possible_detours):
    filtered_detours = []

    # sort from biggest saved distance to lowest
    sorted_detours = sorted(possible_detours, key=lambda x: x[2],
reverse=True)

    visited_places = set()
    for detour in sorted_detours:
        # check if there is no overlap with previously visited places
        # (respectively places that will be skipped due to detour getting
removed)
        # because we first sorted the detours with largest saved distance, we
can
```

```
        # be sure that the most efficient detours get removed and the rest is
left on route
        # because the places are (possibly) overlapping
        if all(p not in visited_places for p in range(detour[0], detour[1])):
            visited_places.update(range(detour[0], detour[1]))
            filtered_detours.append(detour)

    return filtered_detours
```

4.3 create_efficient_route(detours, places)

```
# remove the given detours from the route to create the most efficient one
def create_efficient_route(detours, places):
    indices_to_remove = set()
    saved_distances = {}

    # fill the set with places that will get skipped due to detour
    for detour in detours:
        start_index, end_index, saved_distance = detour
        indices_to_remove.update(range(start_index + 1, end_index))
        saved_distances[end_index] = saved_distance

    # update the distances by subtracting saved distances from the places in
the final route
    for detour_end, saved_distance in saved_distances.items():
        for i in range(detour_end, len(places)):
            if i not in indices_to_remove:
                places[i][3] = places[i][3] - saved_distance

    # create a new list with the original places, not including those in a
detour
    route = [list(place) for i, place in enumerate(places) if i not in
indices_to_remove]

    return route, sum(saved_distances.values())
```