Aufgabe 5: Stadtführung

Team-ID: 01305

Team-Name: RecursionLimitExceeded

Bearbeiter/-innen dieser Aufgabe:
Tim Himmelsbach

20. November 2023

Inhaltsverzeichnis

1 Lösungsidee **1**

2 Umsetzung **2**
2.1 Identifizieren der Teiltouren . . . 2
2.2 Weighted Interval Scheduling . . . 3
2.3 Implementation . . . 3

3 Beispiele **3**
3.1 tour1.txt . . . 3
3.2 tour2.txt . . . 3
3.3 tour3.txt . . . 3
3.4 tour4.txt . . . 4
3.5 tour5.txt . . . 4

4 Quellcode **4**
4.1 Identifizieren der Teiltouren . . . 4
4.2 Ermitteln der optimalen Lösung . . . 5
4.3 Kürzen der Tour . . . 5

1 Lösungsidee

Gesucht wird eine Stadttour **minimaler Streckenlänge**, die durch das Kürzen von geschlossenen Teiltouren aus einer gegebenen Tour erreicht werden kann. Das Kernproblem der Aufgabe besteht im Umgang mit Teiltouren bei welchen das Kürzen den Start- oder Endpunkt einer anderen Teiltour entfernt. Das Prolem der Teiltouren kann auf das **Maximum Weight Independent Set** (MWIS) Problem reduziert werden. Dabei wird eine Teiltour als gewichteter Knoten und ein Konflikt zwischen zwei Teiltouren als Kante zwischen den entsprechenden Knoten modelliert. Das Gewicht eines Knotens entspricht der geographischen Länge der Teiltour. Das MWIS kann mithilfe eines **Branch and Bound** Algorithmus in nicht-polynomieller Zeit ermittelt werden. Jedoch handelt es sich bei jedem Teiltourkonfliktgraphen um einen **Kreisbogengraphen** für welchen das MWIS in polynomieller Zeit ermittelt werden kann. Bei weiterer Betrachtung fallen parallelen zum **Weighted Intervall Scheduling** Probem auf, welches mit bereit sortiertem Input in lineare Laufzeit gelöst werden kann. Das Problem ist äquivalent zum Ermitteln des MWIS auf einem Intervallgraphen. Ein Teiltourkonfliktgraphen ist ein Intervallgraph, wenn es mindestens ein essenzieller Tourpunkt existiert.

Eine Stadttour A der Länge n wird als Folge von Tourpunkten $A = p_1, p_2, ..., p_n$ definiert. Jedem Tourpunkt wird ein Ort, der kumulierte Abstand zum Startpunkt der Tour, sowie ein Zeitpunkt zugeordnet:

$p_i = (o_i, d_i, t_i)$. Die Tour verläuft streng chronologisch: $t_1 < t_2 < ... < t_n$ und beginnt an dem Ort an dem sie endet: $o_1 = o_n$. E ist die Menge aller essenziellen Tourpunkte.

Eine geschlossene Teiltour wird als offenes Intervall von Start- bis Endpunkt dargestellt $(p_i, p_j) \neq \emptyset$ dargestellt, vorausgesetzt $i < j$. Der Ort des Start- und Endpunktes sind identisch: $o_i = o_j$. Zwischen dem Start- und Endpunkt darf es keinen essenziellen Tourpunkt geben: $\forall k \in (p_i, p_j) : k \notin E$.

Bisher wird eine geschlossene Teiltour zwischen zwei willkürlichen Zeitpunkten des selben Ortes definiert. Dies ist aus mehreren Gründen problematisch. Wenn ein Tourpunkt Startpunkt (oder Endpunkt) von mehreren geschlossenen Touren ist, entstehen Redundanzen. So beschreibt die Teiltour (p_a, p_c) im Grunde das gleiche wie $(p_a, p_b) \cup (p_b, p_c)$, wenn $t_a < t_b < t_c$ und $o_a = o_b = o_c$. Beim Kürzen der Teiltour (p_a, p_c) würde außerdem der Punkt p_b entfernt werden, obwohl dies für eine maximale Kürzung nicht notwendig gewesen wäre. Somit wäre (p_a, p_c) nach meiner Auffassung nicht im Sinne der Aufgabenstellung. Somit wird zusätzlich festgelgt, dass $\forall k \in (i, j) : o_k \neq o_i$. Die Streckenlänge einer geschlossenen Teiltour ist $d(p_i, p_j)$.

Eine Roundtour hat wie ein Kreis weder Anfang noch Ende. Man kann aufgrunddessen interpretieren, dass auch geschlossene Teiltouren, die den Start- oder Endpunkt beinhalten möglich sind. Man kann eine solche Teiltour als Vereinigung von zwei halboffenen Intervallen darstellen: $(p_c, p_n] \cup [p_1, p_a)$. Beim Kürzen einer solchen Teiltour würden sich der Start- und Endpunkt der gesanteb Stadtour verändern. In diesem Beispiel wird der Startpunkt zu p_a und der Endpunkt zu p_c.

Um eine Stadttour minimaler Länge zu erreichen, muss die Summe der Länge der gekürzten Touren maximal sein unter der Beachtung, dass das Kürzen einer geschlossenen Teiltour eventuell den Start- oder Endpunkt einer anderen Teiltour entfernt, diese somit nicht gekürzt werden kann. Sei der Graph $G = (V, E, w)$ ein Teiltourenkonfliktgraph. Ein Knoten $v \in V$ repräsentiert eine Teiltour. Zwei Knoten teilen eine Kante, wenn dessen korrespondierende Teiltouren eine nicht-leere Schnittmenge haben. Die Gewichtung eines Knotens entspricht der Länge der korrespondierenden Teiltour. Gesucht wird die Menge von Knoten $\mathcal{I}$, sodass die Summe der Gewichte aller Knoten in $\mathcal{I}$ maximal ist und keine zwei Knoten eine Kante teilen.

$$\arg\max_{\mathcal{I} \subseteq V} \sum_{i \in \mathcal{I}} w(i) \quad \text{sodass} \quad \forall u, v \in \mathcal{I} : e(u, v), e(v, u) \notin E \tag{1}$$

Das Problem ist formal als *Maximum Weight Independent Set* bekannt. Es gibt keinen bekannten Algorithmus, der das Problem für alle Graphen in polynomieller Zeit Lösen kann, somit fällt es unter die Kategorie der NP-schweren Probleme. Man könnte das MWIS für einen Teiltourenkonfliktgraphen in exponetieller Laufzeit $\mathcal{O}(2^n)$ mit einerm Branch and Bound Algorithmus ermitteln. Das Problem wird in kleinere Teilprobleme geteilt. Ein Knoten $v \in V$ kann zwei Zustände annehmen, entweder ist er Teil der optimalen Lösung $v \in \mathcal{I}^*$ oder nicht $v \notin \mathcal{I}^*$. Ist der Knoten Teil der Lösung, sind alle Knoten mit welchen der Knoten eine Kante teilt nicht Teil der Lösung, sonst wäre die Menge nicht *independent* (unabhängig). Es handelt sich hier um einen Spezialfall des MWIS Problem. Bei jedem Teiltourenkonfliktgraphen handelt es sich um einen Kreisbogengraphen. Jeder Kreisbogengraph ist wiederum auch ein Intervallgraph wenn es einen Punkt auf dem Bogenmodell gibt, der sich nicht zwischen dem Start- und Endpunkt eines Kreisbogens befindet. Zerschneidet man den Kreisbogen an einem solchen Punkt, erhält man eine Linie mit Intervallen. Diese Linie ist ein Intervallgraph. Somit ist dieser Kreisbogengraph ein Intervallgraph. Ein essezieller Tourpunkt erfüllt diese Charakteristik. Das Problem des MWIS auf einem Intervallgraphen ist äquivalent zu dem **Weighted Interval Scheduling Problem**, welches in linearer Laufzeit $\mathcal{O}(n)$ gelöst werden kann. Ein Algorithmus wurde erstmals von *Kleinberg* und *Tardos* im Jahr 2006 vorgestellt. [1] Auf die Funktionsweise dieses Algorithmus wird im nächsten Abschnitt eingegangen.

2 Umsetzung

2.1 Identifizieren der Teiltouren

Die Teiltouren werden mittels eines linearen Verfahrens ermittelt. Dazu wird die Tour ab einem essenziellen Punkt abgelaufen. Ist kein esentieller Punkt vorhanden, ist dies der erste Punkt der Tour. Der letzte Punkt eines Ortes wird in einer HashMap gespeichert. Diese wird während des passieren eines esssenziellen Punktes geleert, somit wird verhindert das später essenziellen Punkte gekürzt werden. Gibt es für den Ort des aktuellen Punktes schon einen Wert in der HashMap, wird eine Teiltour gespeichert. Außerdem wird in der HashMap gespeichert wie viele Teiltouren schon vor dem Starpunkt beendet wurden, was für das Ermitteln der optimalen Lösung später wichtig ist. Dieses Verfahren wird wiederholt bis alle Punkte der Tour abgelaufen sind.

[1] J. Kleinberg, E. Tardos. Algorithm Design. 2006

2.2 Weighted Interval Scheduling

Voraussetzung für die lineare Laufzeit ist, dass die Teiltouren schon nach ihrem Endpunkt sortiert sind. Dies ist durch das vorherige Verfahren gegeben. Sei t_i eine Teiltour. Dann ist s_i die Einsparung, also die geographische Länge der Teiltour. p_i ist die Anzahl der Touren, deren Endpunkt vor dem Starpunkt dieser Teiltour liegen, also nicht mit ihr überlappen. p_i ist somit auch der Index, der letzten Teiltour die nicht mit t_i überlappt. t_0 ist aus Berechnugszwecken ein Platzhalter: $p_0 = 0$ und $s_0 = 0$. $M(i)$ ist die optimale Lösung für die Teiltouren $t_1, t_2, ..., t_i$. $M(i)$ kann rekursiv wie folgt berechnet werden: $M(i) = \max\{s_i + M(p_i), M(i-1)\}$ (Ähnlich wie zu den Branch and Bound Algorithmus). Die Teiltouren werden von $i = 1$ bis $i = n$ durchiteriert dabei wird $M(i)$ berechnet und gespeichert, dabei ist $M(0) = 0$. Um schlussendlich die Lösung zurückverfolgen zu können wird von $i = n$ bis $i = 1$ iteriert. Eine Teiltour gehört dann zur optimalen Lösung, wenn $s_i + M(p_i) > M(i-1)$.

2.3 Implementation

Der Algorithmus wurde in Python implementiert. Beim Aussführen muss der Pfad zur gewünschten Datei angegeben werden.

3 Beispiele

Bemerkung: Sonderzeichen ä, ü, ß werden in der Ausgabe als ae, ue, ss dargestellt.

3.1 tour1.txt

```
Brauerei,1613,X,0
Karzer,1665,X,80
Rathaus,1678,X,150
Rathaus,1739,X,150
Euler-Bruecke,1768, ,330
Fibonacci-Gaststaette,1820,X,360
Schiefes Haus,1823, ,480
Theater,1880, ,610
Emmy-Noether-Campus,1912,X,740
Emmy-Noether-Campus,1998,X,740
Euler-Bruecke,1999, ,870
Brauerei,2012, ,1020
Laenge der alten Tour: 2060
Laenge der neuen Tour: 1020
Zeit: 0.0004286766052246094s
```

3.2 tour2.txt

```
Brauerei,1613, ,0
Karzer,1665,X,80
Rathaus,1678, ,150
Rathaus,1739, ,150
Euler-Bruecke,1768, ,330
Fibonacci-Gaststaette,1820,X,360
Schiefes Haus,1823, ,480
Theater,1880, ,610
Emmy-Noether-Campus,1912,X,740
Emmy-Noether-Campus,1998,X,740
Euler-Bruecke,1999, ,870
Brauerei,2012, ,1020
Laenge der alten Tour: 2060
Laenge der neuen Tour: 1020
Zeit: 0.0005230903625488281s
```

3.3 tour3.txt

```
Talstation,1768, ,0
Waeldle,1805, ,520
Mittlere Alp,1823, ,1160
Observatorium,1833, ,1450
Observatorium,1874,X,1450
Piz Spitz,1898, ,1920
```

```
Panoramasteg,1912,X,2140
Panoramasteg,1952, ,2140
Ziegenbruecke,1979,X,2390
Talstation,2005, ,2670
Laenge der alten Tour: 4560
Laenge der neuen Tour: 2670
```

3.4 tour4.txt

```
Marktplatz,1549, ,0
Marktplatz,1562, ,0
Springbrunnen,1571, ,80
Dom,1596,X,150
Bogenschuetze,1610, ,270
Bogenschuetze,1683, ,270
Schnecke,1698,X,420
Fischweiher,1710, ,600
Reiterhof,1728,X,720
Schnecke,1742, ,860
Schmiede,1765, ,1030
Grosse Gabel,1794, ,1140
Grosse Gabel,1874, ,1140
Fingerhut,1917,X,1210
Stadion,1934, ,1330
Marktplatz,1962, ,1420
Laenge der alten Tour: 3200
Laenge der neuen Tour: 1420
Zeit: 0.0005469322204589844s
```

3.5 tour5.txt

```
Gabelhaus,1638, ,0
Gabelhaus,1699, ,0
Hexentanzplatz,1703,X,160
Eselsbruecke,1711, ,280
Dreibannstein,1724, ,390
Dreibannstein,1752, ,390
Schmetterling,1760,X,540
Dreibannstein,1781, ,620
Maerchenwald,1793,X,700
Maerchenwald,1840, ,700
Eselsbruecke,1855, ,780
Eselsbruecke,1877, ,780
Reiterdenkmal,1880, ,920
Riesenrad,1881, ,1100
Riesenrad,1902, ,1100
Dreibannstein,1911,X,1230
Olympisches Dorf,1924, ,1390
Haus der Zukunft,1927,X,1520
Stellwerk,1931, ,1640
Stellwerk,1942, ,1640
Labyrinth,1955, ,1850
Gauklerstadl,1961, ,1930
Planetarium,1971,X,2010
Kaenguruhfarm,1976, ,2060
Balzplatz,1978, ,2140
Dreibannstein,1998,X,2230
Labyrinth,2013, ,2360
CO2-Speicher,2022, ,2550
Gabelhaus,2023, ,2620
Laenge der alten Tour: 5000
Laenge der neuen Tour: 2620
Zeit: 0.0005822181701660156s
```

4 Quellcode

4.1 Identifizieren der Teiltouren

```python
# Identifizieren der Teiltouren
q = {} # Speichert das letzte Vorkommen eines Ortes nach einem essentiellen Punkt
teiltouren = [Teiltour(0,0,0,0)]
for index in range(first_essentiel,first_essentiel+n+1):
```

```
    t = tour[index%n]
    # Wenn der Ort schon einmal vorkam, dann ist er der Endpunkt einer Teiltour
    if t.ort in q:
        last_occurence, n_intervals_before = q[t.ort]
        start = tour[last_occurence]
        # Berechnen der Ersparnis
        saving = t.position - start.position if t.position >= start.position else tour[-1].position - start.position + t.position
        if saving != 0: teiltouren.append(Teiltour(last_occurence, index%n, saving, n_intervals_before))
    if t.essentiel: q = {}
    # Speichern des letzten Vorkommens eines Ortes und die Anzahl der Teiltouren die vor dem Starpunkt enden
    q[t.ort] = (index%n, len(teiltouren) - 1)
```

4.2 Ermitteln der optimalen Lösung

```
# Berechnen der optimalen Loesung
for i in range(1, len(teiltouren)):
    t = teiltouren[i]
    t.m = max(teiltouren[i-1].m, teiltouren[t.before].m + teiltouren[i].weight)

# Rueckverfolgen der optimalen Loesung
L = [] # Liste der Teiltouren, die in der optimalen Loesung enthalten sind
i = len(teiltouren) - 1
while i > 0:
    t = teiltouren[i]
    if t.weight + teiltouren[t.before].m >= teiltouren[i-1].m:
        L.append(t)
        i = t.before
    else:
        i -= 1
```

4.3 Kürzen der Tour

```
# Kuerzen der Tour
in_tour = [True] * n # Speichert, ob ein Punkt in der Tour enthalten ist
saving = [0] * n # Speichert die Ersparnis beim Endpunkt einer Teiltour
for t in L:
    if t.end > t.start:
        in_tour[t.start+1:t.end] = [False] * (t.end - (t.start + 1))
        saving[t.end] = t.weight
    else:
        in_tour[t.start+1:] = [False] * (n - (t.start + 1))
        in_tour[:t.end] = [False] * t.end
        saving[t.end] = tour[t.end].position

# Ausgabe der neuen Tour
s = 0
l = 0
for i in range(n):
    if in_tour[i]:
        s += saving[i]
        l = tour[i].position - s
        print(f'{tour[i].ort},{tour[i].zeit},{"X" if tour[i].essentiel else " "},{tour[i].position-s}')
```