Das Travelling Salesman Problem

Theorie

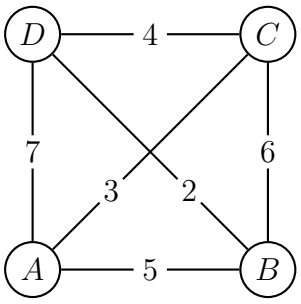
28. November 2025

Fragestellung

Ein Handelsvertreter soll n Städte besuchen und wieder in die zuerst besuchte Stadt zurückkehren. Welches ist die kürzeste Route, wenn alle paarweisen Distanzen zwischen jeweils zwei Städten bekannt sind?

Beispiel

Städtemodell ($n = 4$)



Distanztabelle

	A	B	C	D
A				
B				
C				
D				

Brute Force-Methode

Berechne die Längen aller möglichen Touren und wähle die kürzeste aus.

Die Menge aller Touren erhalten wir, indem wir alle möglichen Anordnungen der Städte bestimmen. Eine Anordnung von n Elementen auf n Plätzen wird *Permutation* von n Elementen genannt.

Aufgabe 1

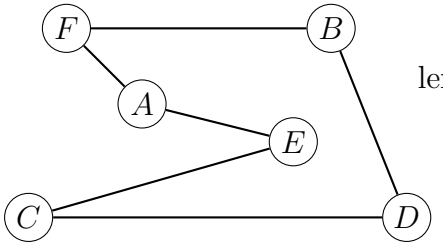
Berechne die Distanzen aller möglichen Touren in Beispiel 1, die in A beginnen. Welche ist die kürzeste?

Tour	**Länge**
$\underline{A}BCD\underline{A}$	$5 + 6 + 4 + 7 = 22$

Tour	**Länge**

Wo sollen wir starten?

Der Start der Rundreise ist für die Länge der Tour unerheblich, da eine *Rundtour* immer dieselbe Länge hat, egal, an welchem Ort man sie beginnt.



$$\text{length}(A, F, B, D, C, E, A)$$
$$= \text{length}(D, C, E, A, F, B, D)$$

Distanztabellen in Python

Als Datenstruktur für die Distanzen wählen wir eine Liste von Listen, wobei die Indizes 0, 1, 2, 3 für die Städte A, B, C, D stehen. Im Beispiel 1:

```
D = [[0, 5, 3, 7],
     [5, 0, 6, 2],
     [3, 6, 0, 4],
     [7, 2, 4, 0]]
```

Distanz zwischen den Städten B und D:

Distanz zwischen der ersten und letzten Stadt:

Eine Tour in Python

Wählen wir (willkürlich) die Stadt mit dem Index 0 ans Start- und Endpunkt der Tour, so genügt es, eine Rundtour durch n Städte als Liste mit $n-1$ Elementen darzustellen:

$ACDBA$ $\Leftrightarrow$ `[0, 2, 3, 1, 0]` $\Leftrightarrow$ `T = [2, 3, 1]`

Tourlängen berechnen

```
def tour_length(T, D):
    '''Länge einer Rundreise 0->T->0'''
    length = D[0][T[0]] # Startstrecke
    for i in range(0, len(T)-1):
        length += D[T[i]][T[i+1]]
    length += D[T[-1]][0] # Rückkehr
    return length
```

Erzeugung der Permutationen

Das Generieren aller Permutationen einer Liste ist nicht ganz einfach. Deshalb verwenden wir das Python-Modul `itertools`, das alle Permutationen einer Liste `L` mit der Methode `permutations(L)` erzeugt und als sogenannten Iterator zurückgibt, der mit einer `for`-Schleife durchlaufen werden kann.

```
from itertools import permutations

for p in permutations([1,2,3]):
    print(p)
```

Über diese Schleife lässt sich das Minimum aller Tour-Längen bestimmen.

Laufzeitkomplexität der Brute Force-Methode

- Wählt man eine beliebige Stadt als Ausgangspunkt, so sind bei n Städten $(n-1)!$ Permutationen (Touren) zu erzeugen.
- Für jede Permutation (Tour) sind n Einzeldistanzen aus der Distanzmatrix herauszulesen und zu addieren.
- Nehmen wir stark vereinfachend an, dass für die Erzeugung einer Permutation, für das Herauslesen einer Einzeldistanz und das Addieren der Distanzen jeweils ein konstanter Zeitaufwand C nötig ist, erhalten wir folgende Laufzeitkomplexität:

Eine „naive“ Implementierung in Python

```
from itertools import permutations
from tour_length import tour_length

def tsp_brute_force(D):
    '''Brute Force-Lösung des TSP'''
    n = len(D)
    opt_dist = float('inf')
    opt_tour = None
    for p in permutations(range(1, n)):
        dist = tour_length(p, D)
        if dist < opt_dist:
            opt_dist = dist
            opt_tour = p
    return opt_dist, [0]+list(opt_tour)+[0]
```

Aufgabe 2

Schreibe ein Python-Modul `tsp_benchmark.py`, das den Zeitaufwand für die Brute Force-Lösung für $n = 4, 5, \ldots, 11, 12$ Städte mit der `time()`-Funktion aus dem gleichnamigen Modul berechnet. Der folgende Code erzeugt zufällige Distanzmatrizen.

```
from random import randint, sample

def random_dist(n, symmetric=True, a=10, b=99):
    '''Zufällige Distanzmatrix für n Städte.'''
    D = [[0 for i in range(n)] for j in range(n)]
    for i in range(0, n-1):
        for j in range(i+1, n):
            D[i][j] = randint(a, b)
            if symmetric:
                D[j][i] = D[i][j]
            else:
                D[j][i] = randint(a, b)

```

```
14      return D
15
16  def random_dist_diff(n, scale=1):
17      '''Symm. Distanzmatrix für n Städte mit verschiedenen Distanzen'''
18      D = [[0 for i in range(n)] for j in range(n)]
19      k = n*(n-1)//2
20      items = sample(range(1, scale*k+1), k)
21      for i in range(0, n-1):
22          for j in range(i+1, n):
23              D[i][j] = items.pop()
24              D[j][i] = D[i][j]
25
26      return D
27
28
29  if __name__ == '__main__':
30
31      D = random_dist_diff(5, 3)
32      for row in D:
33          print(row)
```

Das Problem

Die Lösung von Aufgabe 2 zeigt, dass der Zeitaufwand für das Testen aller möglichen Pfade mit unserer Implementierung faktoriell in Abhängigkeit der Städtezahl n wächst. Dies bedeutet, dass schon bei relativ bescheidenen Städtezahlen Wartezeiten entstehen, die nicht mehr tolerierbar sind.

Zwar gibt es Algorithmen, welche das TSP „nur“ mit exponentiellem Zeitaufwand exakt lösen können aber auch dort tritt früher oder später das Problem auf, dass die Wartezeiten inakzeptabel gross werden.

Nearest Neighbor-Heuristik

Der Begriff *Heuristik* bezeichnet ein Verfahren, das trotz begrenztem Wissen eine mögliche Lösung eines Problems findet. Diese Lösung kann jedoch von der optimalen Lösung abweichen.

Bei der Nearest Neighbor-Heuristik (NNH) für das TSP wählen wir einen Startknoten aus und bestimmten eine Rundtour, in dem wir, im Startknoten beginnend, jeweils zu einer der am nächsten liegenden Städte gehen und dann auf die gleiche Weise die folgenden Städte aufsuchen. Bei der Rückkehr zum Ausgangspunkt können wir nicht mehr auswählen und müssen die vorgegebene Distanz verwenden.

Diese Strategie wird in der Informatik als *greedy* (gierig) bezeichnet und führt beim TSP im allgemeinen nicht zu einer optimalen Lösung.

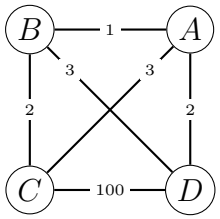
Aufgabe 3

Berechne für das TSP mit der folgenden Distanztabelle die Länge der Rundtouren mit der NNH für alle möglichen Startpunkte und vergleiche mit der kürzesten Tour ($ACBDA$). Welche Schlussfolgerungen lassen sich aus den Resultaten ziehen?

	A	B	C	D
A	0	1	1	1
B	1	0	4	1
C	1	4	0	7
D	1	1	7	0

Aufgabe 4

Gegeben ist folgendes TSP



Zeige, dass man mit der Nearest-Neighbor-Heuristik für jeden Startknoten dieselbe Tourlänge erhält. Gibt es eine bessere Lösung? Wenn ja, welche?

Bisher waren die Kantengewichte mehr oder weniger zufällig gewählte positive Zahlen. Bei realen Anwendungen im Bereich des Transportwesens, tritt jedoch eine wichtige Einschränkung auf, die z. B. von TSP in Aufgabe 4 nicht erfüllt wird. Betrachten wir dort den von den Knoten A, C und D induzierten Teilgraphen, stellen wir fest, dass der direkte Weg von C nach D länger ist als der Umweg von C über A nach D, was unserer Vorstellung von Distanzen im „realen Raum“ widerspricht. Wir wollen nun die Bedingung(en) formulieren, für die das oben genannte Problem nicht auftreten kann.

Metrische Räume

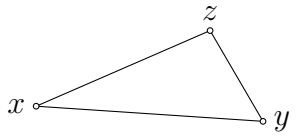
Es sei M eine beliebige Menge und d eine Funktion, die jedem Paar $(x, y) \in M \times M$ eine nichtnegative Zahl $d(x, y)$ zuordnet. Wenn für alle x, y, $z \in M$ die Eigenschaften

(1) $d(x, x) = 0$

(2) $d(x, y) = d(y, x)$

(3) $d(x, y) \leq d(x, z) + d(z, y)$

erfüllt sind, dann wird das Paar (M, d) *metrischer Raum* genannt. Bedingung (3) besagt, dass in einem Dreieck jede Seite nie länger als die Summer der beiden anderen sein darf. Daher wird (3) auch *Dreiecksungleichung* genannt.



Metrisches TSP

Fordern wir, dass die Menge der Städte und ihre paarweisen Distanzen einen metrischen Raum bilden, so handelt es sich um ein *metrisches TSP*.

Dies führt unter anderem dazu, dass „pathologische“ Beispiele wie in Aufgabe 4 nicht mehr so einfach zu konstruieren sind.

Die MST-Heuristik

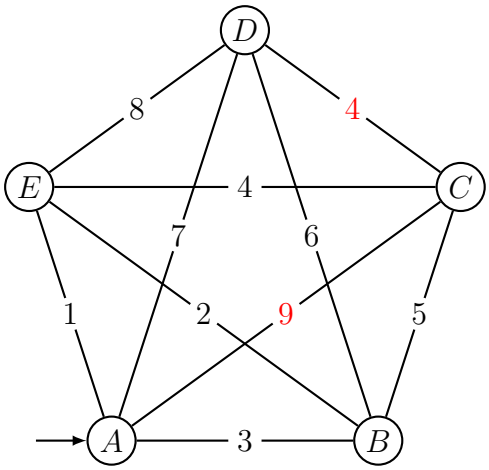
Ist ein TSP metrisch, so garantiert uns der folgende Algorithmus eine Rundreise H, die nicht länger als das Doppelte der optimalen Lösung H_{opt} ist.

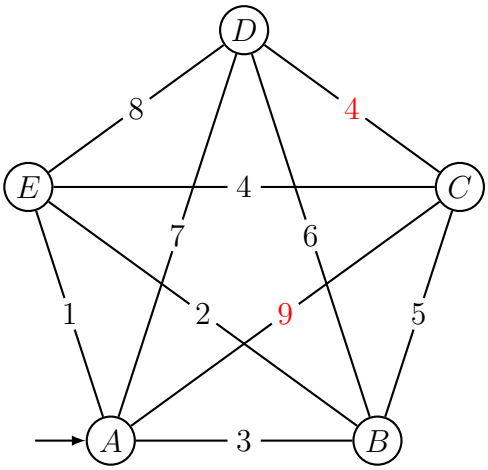
(1) Bestimme einen minimalen Spannbaum (MST) T im Städtegraphen ($\rightarrow$ Algorithmus von Prim oder Kruskal).

(2) Wähle v_0 als Startknoten, führe auf T eine Tiefensuche durch und verbinde den zuletzt erreichten Knoten mit v_0.

Die Folge von Knoten aus (2) ist die gesuchte Lösung H.

Beispiel





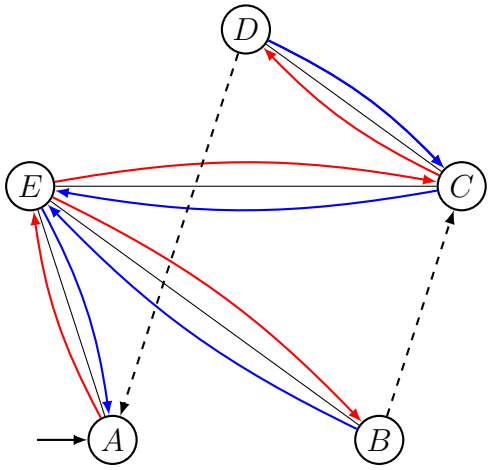
Beweis von $|H| \leq 2 \cdot |H_{\mathbf{OPT}}|$

Es seien H_{OPT} eine optimale Rundtour und T_{MST} ein minimaler Spannbaum des Graphen G. Mit Betragszeichen bezeichnen wir jeweils die Längen von Rundtouren oder Bäumen.

Entfernen wir aus H_{OPT} irgend eine Kante, so erhalten wir einen Spannbaum T und es gilt $|H_{\text{OPT}}| > |T|$.

Da ein minimaler Spannbaum nicht länger als irgend ein Spannbaum des Graphen ist, gilt $|H_{\text{OPT}}| > |T| \geq |T_{\text{MST}}|$.

Wenn wir den Baum T_{MST} mit einer Tiefensuche durchlaufen und wieder zum Startknoten zurückkehren, haben wir jede seiner Kanten doppelt verwendet.



Da wir in der gesuchte Rundtour H keine Kante doppelt durchlaufen dürfen, müssen wir Knoten überspringen (unterbrochene Pfeile), welche die Tiefensuche erneut besucht, um aus einer „Sackgasse“ zurückzukehren (blaue Pfeile).

Weil das TSP metrisch ist, ist jede „Abkürzung“ (unterbrochene Pfeile) auch tatsächlich kürzer als der Weg über die bereits besuchten Knoten. Daher ist im Beispiel $|BC|$ kürzer als $|BE| + |EC|$ und $|DA|$ ist kürzer als $|DC| + |CE| + |EA|$.

Wir fassen die Ergebnisse zusammen:

$$|H_{\text{OPT}}| > |T_{\text{MST}}| \quad \Rightarrow \quad 2 \cdot |H_{\text{OPT}}| > 2 \cdot |T_{\text{MST}}|$$

$$2 \cdot |T_{\text{MST}}| > H \quad \Rightarrow \quad 2 \cdot |H_{\text{OPT}}| > H$$

□