Dijkstra-Lernen

March 24, 2023

1 Dijkstra: kürzeste Wege in einem Graphen

Die Suche nach kürzesten Verbindungen in Graphen hat in vielen Bereichen des täglichen Lebens praktische Anwendungen:

- **Navigationssysteme** finden den kürzesten Weg zwischen zwei Orten.
- **Lieferunternehmen** suchen die effizienteste Route für die Zustellung von Waren.
- **Stadtplaner und Verkehrsingenieure** möchten die Verkehrsflüsse optimieren, um Staus zu vermeiden.
- **Routenplanung in Netzwerken** zielt darauf, Datenpakete schnell und effizient zu transportieren.

Bereits 1959 entwickelte der niederländische Mathematiker *E. W. Dijkstra* einen Algorithmus, um in gewichteten Graphen kürzeste Wege zu finden.

Natürlich könnte man alle möglichen Wege zwischen zwei Knoten auflisten und so den kürzesten zu finden (brute-force-Ansatz), doch schon in kleinen Graphen gibt es sehr viele solcher Wege, so dass dieses Verfahren nicht wirklich effizient ist.

Der *Dijkstra-Algorithmus* löst das Problem demgegenüber sehr effizient.

In diesem Notebook werden wir diesen Algorithmus an einem Beispiel durchführen.

Dazu benutzten wir - die Programmiersprache **Python** (in der Version 3.10) - die **Jupyter-Notebook-Umgebung** - eine spezielle Python-Bibliothek `networkx`, mit der wir sehr leicht gewichtete Graphen implementieren können. - eine eigene Python-Bibliothek `nrw_graph`, die auf `networkx` basiert, die den Umgang mit Graphen methodisch-didaktisch vereinfacht.

[1]:
```
#import nrw_graph as ng
#help("nrw_graph")
```

1.1 Notwendige Bibliotheken importieren

[2]:
```
import networkx as nx
import nrw_graph as ng

# pandas ist eine Bibliothek für Python u.a. zur Verarbeitung von Daten.
import pandas as pd

# Bibliothek, z.B. um Daten graphisch darzustellen.
import matplotlib.pyplot as plt
```

1.2 Graph (Kanten, Knoten, Gewicht) aus einer Datei einlesen

Ein *gewichteter Graph* wird beschrieben durch die Angabe der zu dem Graphen gehörenden Kanten. Eine Kante ist dabei ein Objekt, in dem die Namen der beiden Endknoten sowie das Gewicht der Kante (in unserem Beispiel die Länge) enthalten sind.

Da die Knoten, die der Graph enthält, nicht explizit angegeben werden, sondern sich aus den Endknoten der Kanten ergeben, kann man auf diese Weise keine Graphen mit isolierten Knoten erzeugen. Jedoch ist das für unser Beispiel nicht tragisch, da von und zu isolierten Knoten sicherlich kein Weg führt.

Die für einen Graphen notwendigen Daten sollten sich in einer CSV-Datei befinden. Eine solche Datei enthält die Daten (also die Information über eine Kante) zeilenweise, wobei die erste Zeile ein Art Überschrift ist.

Jede Zeile enthält - in der Regel duch Kommata oder Semikolon getrennt - die Werte der jeweiligen Attribute:

- Name des Startknoten
- Name des Zielknoten
- Länge der Kante

Dabei sind in diesem Zusammenhang die Begriffe *Start* und *Ziel* ggf. missverständlich, da die Graphen, die hier benutzt werden, ungerichtet sind; gibt es also eine Kante von A nach B, die in dem Datensatz angegeben ist, gibt es automatisch auch die Kante von B nach A gleicher Länge, ohne dass sie explizit in den Datensätzen auftaucht.

[3]:
```
df_staedte = pd.read_csv("staedte.txt", sep=",")

# Hier werden aus Gründen der Übersichtlichkeit
# nur die ersten 10 Datensätze in einer Tabelle gezeigt.
display(df_staedte.head(10)) #
```

	Start	Ziel	Entfernung
0	Kiel	Schwerin	160
1	Kiel	Hamburg	97
2	Kiel	Bremen	211
3	Hamburg	Bremen	126
4	Schwerin	Hamburg	110
5	Hamm	Münster	70
6	Münster	Bremen	120

```
7     Bremen  Hannover         127
8    Hamburg  Hannover         159
9  Bielefeld  Hannover          90
```

1.3 Der Graph wird aus den Daten konstruiert

[4]:
```
# Ein neuer leerer Graph
autobahn = ng.nrw_graph()

zeilen = df_staedte.shape[0]
for i in range(zeilen):
    source = df_staedte.iloc[i]['Start']
    target = df_staedte.iloc[i]['Ziel']
    dist = float(df_staedte.iloc[i]['Entfernung'])

    autobahn.fuegeKanteHinzu(source, target, gewicht=dist)
    autobahn.deflagToKnoten(source)
    autobahn.deflagToKnoten(target)
```

1.4 Zeig mal den Graphen

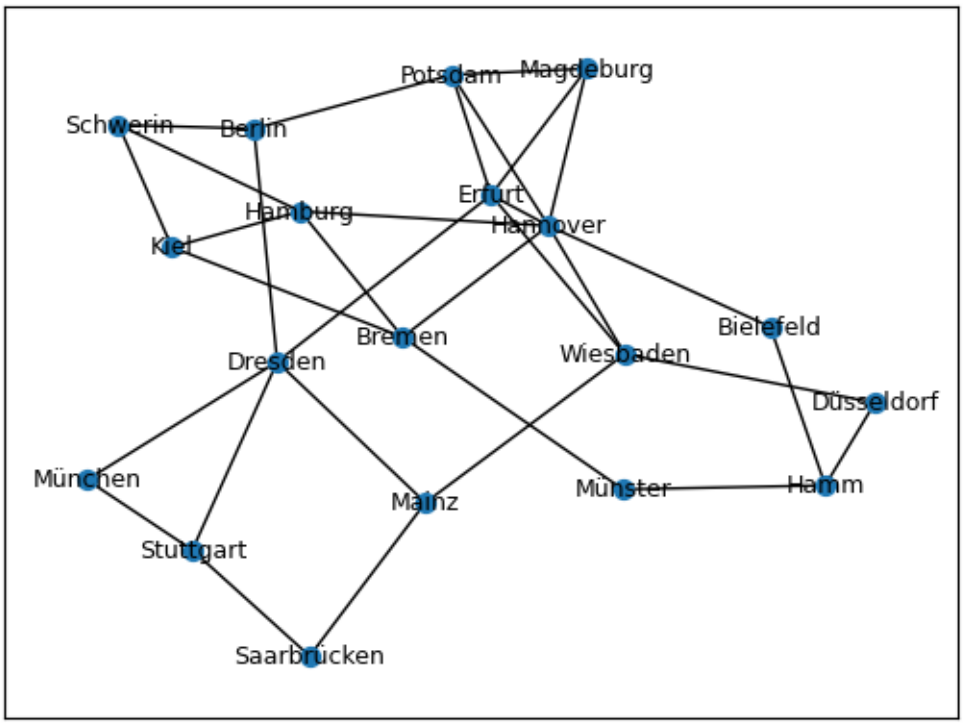
Wenn man möchte, kann man den Graphen visualisieren.

Doch **Vorsicht**:

- Die folgende Darstellung entspricht nicht den tatsächlichen geographischen Tatsachen. Denn die Daten enthalten kein Angaben über die Lage der Knoten zueinander.
- Die Länge der Verbindungen ist nicht proportional zu den in den Daten angegebenen Entfernungen.

Diese Darstellung ist also nur eine nette Spielerei, um die Fähigkeit der Bibliothek zu demonstrieren!

[5]:
```
# nx.draw(autobahn)
nx.draw_networkx(autobahn,
                 node_size=50,
                 font_size=9,
                 pos=nx.spring_layout(autobahn),
#                  pos=nx.planar_layout(autobahn),
                )
plt.draw()
```



1.5 Kontrolle: hat das Einlesen der Daten geklappt?

[6]:
```
autobahn.alleKanten()
```

[6]:
```
EdgeView([('Kiel', 'Schwerin'), ('Kiel', 'Hamburg'), ('Kiel', 'Bremen'),
('Schwerin', 'Hamburg'), ('Schwerin', 'Berlin'), ('Hamburg', 'Bremen'),
('Hamburg', 'Hannover'), ('Bremen', 'Münster'), ('Bremen', 'Hannover'), ('Hamm',
'Münster'), ('Hamm', 'Düsseldorf'), ('Hamm', 'Bielefeld'), ('Hannover',
'Bielefeld'), ('Hannover', 'Wiesbaden'), ('Hannover', 'Erfurt'), ('Hannover',
'Magdeburg'), ('Hannover', 'Potsdam'), ('Düsseldorf', 'Wiesbaden'),
('Wiesbaden', 'Mainz'), ('Wiesbaden', 'Erfurt'), ('Mainz', 'Saarbrücken'),
('Mainz', 'Dresden'), ('Saarbrücken', 'Stuttgart'), ('Erfurt', 'Magdeburg'),
('Erfurt', 'Potsdam'), ('Erfurt', 'Dresden'), ('Magdeburg', 'Potsdam'),
('Potsdam', 'Berlin'), ('Berlin', 'Dresden'), ('Dresden', 'Stuttgart'),
('Dresden', 'München'), ('Stuttgart', 'München')])
```

[7]:
```
for kante in autobahn.alleKanten():
    start = kante[0]
    ziel = kante[1]
    print(start.ljust(12), " -- ", ziel.ljust(12), ":", autobahn.
 ↪kantenGewicht(start, ziel))
```

```
Kiel          --  Schwerin     : 160.0
```

```
Kiel          --  Hamburg      : 97.0
Kiel          --  Bremen       : 211.0
Schwerin      --  Hamburg      : 110.0
Schwerin      --  Berlin       : 224.0
Hamburg       --  Bremen       : 126.0
Hamburg       --  Hannover     : 159.0
Bremen        --  Münster      : 120.0
Bremen        --  Hannover     : 127.0
Hamm          --  Münster      : 70.0
Hamm          --  Düsseldorf   : 110.0
Hamm          --  Bielefeld    : 80.0
Hannover      --  Bielefeld    : 90.0
Hannover      --  Wiesbaden    : 374.0
Hannover      --  Erfurt       : 249.0
Hannover      --  Magdeburg    : 146.0
Hannover      --  Potsdam      : 264.0
Düsseldorf    --  Wiesbaden    : 212.0
Wiesbaden     --  Mainz        : 8.0
Wiesbaden     --  Erfurt       : 283.0
Mainz         --  Saarbrücken  : 151.0
Mainz         --  Dresden      : 498.0
Saarbrücken   --  Stuttgart    : 222.0
Erfurt        --  Magdeburg    : 170.0
Erfurt        --  Potsdam      : 281.0
Erfurt        --  Dresden      : 215.0
Magdeburg     --  Potsdam      : 134.0
Potsdam       --  Berlin       : 35.0
Berlin        --  Dresden      : 193.0
Dresden       --  Stuttgart    : 507.0
Dresden       --  München      : 460.0
Stuttgart     --  München      : 223.0
```

1.6 Einige Hilfsfunktionen

1.6.1 Informationen über Knoten; Markieren von Knoten

Wir betrachten einmal einen Knoten K des Graphen. In vielen Situationen ist es wichtig zu wissen, von welchem Knoten V (der Vorgängerknoten von K) man zu K kommt und wie weit es dabei ist. - Manchmal möchte man wissen, wie weit der es von V zu K ist, - in anderen Fällen ist die Entfernung vom Start über V zu K interessant.

Insgesamt kann man die drei Informationen - Name von K - Name des Vorgängerknoten - Entfernung zu K

als 3-elementige Liste verwalten, die als Information genutzt wird.

1.6.2 Knoten können *besucht* werden

Knoten können ein boolsches Flag (`True` oder `False`) haben.

Ein Knoten K nennen wir *besucht*, wenn bekannt ist, wie lang der kürzeste Weg vom Start zu K ist. In einem Graphen sind zunächst alle Knoten unbesucht, haben also die Flagge `False`.

Ein besuchter Knoten hat die Flagge `True`. Wenn Knoten besucht sind, haben sie eine Marke in Form einer 3-elementigen Liste (s.o.).

Die folgende Funktion erzeugt eine Liste aller besuchten Knoten (genauer der zugehörigen Knotenmarken):

[8]:
```
def alleBesuchtenKnoten():
    alle = []
    for knoten in autobahn.alleKnoten():
        if autobahn.knotenHatFlag(knoten):
            (ueber, lang) = autobahn.knotenMarke(knoten)
            alle.append([knoten,ueber, lang])
    return alle
```

[9]:
```
# Eine Hilfsfunktion, damit man die später die Liste von Kanten (s.o.) sortieren␣
↪kann.
# Die einträge in einer solchen Liste sind ebenfalls Listen der Form␣
↪[über,ziel,entfernung]
def entfernung (liste):
    return liste[2]
```

1.6.3 Wir finden *Folgeknoten*

Für jeden Knoten K im Graphen ist es wichtig zu wissen, welche Knoten F von K aus direkt zu erreichen sind. Dabei werden die drei Informationen - Name von K - Name von F - Länge der Kante K-F

in Form einer 3-elementigen Liste (s.o.) verwaltet.

Die folgende Funktion liefert eine Liste aller möglichen Kanten von K aus zu Folgeknoten (bzw. deren Kanteninfos):

[10]:
```
def kantenVon(von):
    kanten = []
    for (s,z) in autobahn.alleKanten():
        if s == von:
            l = autobahn.kantenGewicht(s, z)
            kanten.append([s, z, l])
        elif z == von:
            l = autobahn.kantenGewicht(s, z)
            kanten.append([z, s, l])

    return kanten
```

1.6.4 Wir finden *lokale Schnittkanten*

Betrachten wir jetzt einen bereits besuchten Knoten K.

Im Gegensatz zur vorigen Funktion `kantenVon` inetressieren wir uns jetzt nur für solche Folgeknoten F, die noch nicht besucht sind. Eine Kante von K zu dem unbesuchten Folgeknoten F nennen wir eine *lokale Schnittkante*.

Zu dem Folgeknoten F haben wir also (erneut in Form einer 3-elementigen Liste) die Informationen: - Name von K - Name von F - Länge der Kante K-F

Die folgende Funktion liefert eine Liste aller möglichen Schnittkanten von K aus zu Folgeknoten (bzw. deren Kanteninfos):

[11]:
```
def lokaleSchnittkantenVon(von):
    kanten = []
    for (s,z) in autobahn.alleKanten():
        if s == von and not autobahn.knotenHatFlag(z):
            l = autobahn.kantenGewicht(s, z)
            kanten.append([s, z, l])
        elif z == von and not autobahn.knotenHatFlag(s):
            l = autobahn.kantenGewicht(s, z)
            kanten.append ([z,s, l])

    return kanten
```

1.6.5 Wir finden *Schnittkanten*

Die Schnittkanteninformationen, die wir in der vorigen Funktion erzeugt haben, sind jedoch für unsere Zwecke nicht zielführend, da zu einem Folgeknoten F von K nicht die Kantenlänge K-F, sondern die Weglänge vom Start über K zu F wichtig ist.

Eine solche Kante nennen wir *Schnittkante*.

Also erzeugen wir zu jedem unbesuchten Folgeknoten F die Informationen

- Name von K
- Name von F
- Länge des Weges vom Start über K zu F

Die folgende Funktion erzeugt eine Liste aller Schnittkanteninformationen im Graph. Es werden also zu allen besuchten Knoten K die Schnittkanteninfos erzeugt:

[12]:
```
def alleSchnittkanten():
    kanten = []
    for (s,z) in autobahn.alleKanten():
        if autobahn.knotenHatFlag(s) and not autobahn.knotenHatFlag(z):
            l = autobahn.kantenGewicht(s, z)
            (ueber, weit) = autobahn.knotenMarke(s)
            l += weit
            kanten.append ([s, z, l])
        elif autobahn.knotenHatFlag(z) and not autobahn.knotenHatFlag(s):
            l = autobahn.kantenGewicht(z, s)
            (ueber, weit) = autobahn.knotenMarke(z)
            l += weit
```

```
            kanten.append([z, s, l])

    return kanten
```

1.6.6 Markierung von besuchten Knoten

Wenn ein Knoten K besucht ist, möchte man den kürzesten Weg vom Start zu K kennen.

Kennt man zu jedem besuchten Knoten den Vorgänger auf dem kürzesten Weg, kann der kürzeste Weg rückwärts rekonstruiert werden.

Also markieren wir jeden besuchten Knoten K mit einem Tupel, bestehend aus:

- Name des Vorgängers V
- Länge des kürzesten Weges vom Start über V zu K

1.7 Dijkstra "Zu Fuß" lösen: Von Berlin nach München

[13]:
```
startknoten = "Berlin"
zielknoten = "München"
```

1.7.1 Berlin ist bereits besucht!

Zunächst eine Trivialität: - Möchte man von Berlin nach Berlin reisen, so ist die kürzeste Verbindung über Berlin mit einer Länge von 0.0

Also wird Berlin mit einer Flagge und eine Marke der Form (ueber, laenge) versehen:

[14]:
```
autobahn.flagToKnoten("Berlin")
autobahn.markiereKnoten("Berlin", ("Berlin", 0.0))
```

Jetzt kann man sich alle geflaggten Knoten mit ihren Marken ansehen.

Schau dir dazu die entsprechende Hilfsfunktion weiter oben an!

[15]:
```
alleBesuchtenKnoten()
```

[15]:
```
[['Berlin', 'Berlin', 0.0]]
```

1.7.2 Welcher Ort ist Berlin am nächsten?

Schau dir auch dazu die entsprechende Hilfsfunktion weiter oben an!

[16]:
```
kantenVon("Berlin")
```

[16]:
```
[['Berlin', 'Schwerin', 224.0],
 ['Berlin', 'Potsdam', 35.0],
 ['Berlin', 'Dresden', 193.0]]
```

Also ist Potsdam derjenige Ort, der von Berlin am nächsten liegt, so dass wir Potsdam als besucht betrachten können.

Will man also von Berlin nach Potsdam, dann (mal wieder eine Trivialität) fährt man über Berlin; die Strecke hat eine Länge von 35.0

Diese Informationen trägt man ein:

[17]:
```
autobahn.flagToKnoten("Potsdam")
autobahn.markiereKnoten("Potsdam", ("Berlin", 35.0))
```

Zur Kontrolle:

[18]:
```
alleBesuchtenKnoten()
```

[18]:
```
[['Potsdam', 'Berlin', 35.0], ['Berlin', 'Berlin', 0.0]]
```

1.7.3 Jetzt geht's weiter: von Berlin oder von Potsdam?

Man kann jetzt entweder - von Berlin aus direkt - oder von Berlin über Potsdam

weiterfahren zu einem Ort, der möglichst nahe ist.

Definition Kanten, die einen besuchten mit einem unbesuchten Ort verbinden, nennt man **Schnittkanten**

Also suchen wir zunächst alle Orte (mitsamt Entfernungen), die von Berlin direkt erreichbar sind. Dabei lassen wir natürlich den bereits besuchten Ort Potsdam aus:

[19]:
```
lokaleSchnittkantenVon("Berlin")
```

[19]:
```
[['Berlin', 'Schwerin', 224.0], ['Berlin', 'Dresden', 193.0]]
```

Jedoch müssen wir auch Schnittkanten - ausgehend von Potsdam - betrachten. Dabei ist aber zu beachten, dass ein Ort X, der von Potsdam direkt erreichbar ist, eine Gesamtroute der Form

- Berlin - Potsdam - X

hat, so dass die Weglänge sich dann zusammensetzt aus der Länge von (Berlin - Potsdam) und der Länge (Potsdam - X).

[20]:
```
alleSchnittkanten()
```

[20]:
```
[['Berlin', 'Schwerin', 224.0],
 ['Potsdam', 'Hannover', 299.0],
 ['Potsdam', 'Erfurt', 316.0],
 ['Potsdam', 'Magdeburg', 169.0],
 ['Berlin', 'Dresden', 193.0]]
```

Das ergibt also insgesamt 5 Schnittkanten:

1. 'Berlin' - 'Schwerin', 224.0,
2. 'Potsdam' - 'Hannover', 299.0,
3. 'Potsdam' - 'Erfurt', 316.0,
4. 'Potsdam' - 'Magdeburg', 169.0,
5. 'Berlin' - 'Dresden', 193.0

und damit 5 Routen von Berlin aus:

1. 'Berlin', 'Schwerin', 224.0,
2. 'Berlin' - 'Potsdam - 'Hannover', 299.0,
3. 'Berlin' - 'Potsdam' - 'Erfurt', 316.0,
4. 'Berlin' - 'Potsdam' - 'Magdeburg', 169.0,
5. 'Berlin', 'Dresden', 193.0

Die Route nach Magdeburg (über Potsdam) ist also die kürzeste. Das müssen wir jetzt eintragen:

[21]:
```
autobahn.flagToKnoten("Magdeburg")
autobahn.markiereKnoten("Magdeburg", ("Potsdam", 169.0))
```

Auch hier die Kontrolle:

[22]:
```
alleBesuchtenKnoten()
```

[22]:
```
[['Magdeburg', 'Potsdam', 169.0],
 ['Potsdam', 'Berlin', 35.0],
 ['Berlin', 'Berlin', 0.0]]
```

1.7.4 Jetzt ist alles klar!?

Aufgabe:

Setze das Verfahren fort.

Hier die schrittweise Lösung:

[ ]:
```
alleSchnittkanten()
```

[ ]:
```
autobahn.flagToKnoten("Dresden")
autobahn.markiereKnoten("Dresden", ("Berlin", 193.0))
```

[ ]:
```
alleBesuchtenKnoten()
```

[ ]:
```
alleSchnittkanten()
```

[ ]:
```
autobahn.flagToKnoten("Schwerin")
autobahn.markiereKnoten("Schwerin", ("Berlin", 224.0))
```

[ ]:
```
alleBesuchtenKnoten()
```

[ ]:
```
alleSchnittkanten()
```

[ ]:
```
autobahn.flagToKnoten("Hannover")
autobahn.markiereKnoten("Hannover", ("Potsdam", 299.0))
```

[ ]:
```
alleBesuchtenKnoten()
```

[ ]:
```
alleSchnittkanten()
```

[ ]:
```
autobahn.flagToKnoten("Erfurt")
autobahn.markiereKnoten("Erfurt", ("Potsdam", 316.0))
```

[ ]:
```
alleBesuchtenKnoten()
```

[ ]:
```
alleSchnittkanten()
```

[ ]:
```
autobahn.flagToKnoten("Hamburg")
autobahn.markiereKnoten("Hamburg", ("Schwerin", 334.0))
```

[ ]:
```
alleBesuchtenKnoten()
```

[ ]:
```
alleSchnittkanten()
```

[ ]:
```
autobahn.flagToKnoten("Kiel")
autobahn.markiereKnoten("Kiel", ("Schwerin", 384.0))
```

[ ]:
```
alleBesuchtenKnoten()
```

[ ]:
```
alleSchnittkanten()
```

[ ]:
```
autobahn.flagToKnoten("Bielefeld")
autobahn.markiereKnoten("Bielefeld", ("Hannover", 389.0))
```

[ ]:
```
alleBesuchtenKnoten()
```

[ ]:
```
alleSchnittkanten()
```

[ ]:
```
autobahn.flagToKnoten("Bremen")
autobahn.markiereKnoten("Bremen", ("Hannover", 426.0))
```

[ ]:
```
alleBesuchtenKnoten()
```

[ ]:
```
alleSchnittkanten()
```

[ ]:
```
autobahn.flagToKnoten("Hamm")
autobahn.markiereKnoten("Hamm", ("Bielefeld", 469.0))
```

[ ]:
```
alleBesuchtenKnoten()
```

[ ]:
```
alleSchnittkanten()
```

[ ]:
```
autobahn.flagToKnoten("Münster")
autobahn.markiereKnoten("Münster", ("Hamm", 539.0))
```

[ ]:
```
alleBesuchtenKnoten()
```

[ ]:
```
alleSchnittkanten()
```

[ ]:
```
autobahn.flagToKnoten("Düsseldorf")
autobahn.markiereKnoten("Düsseldorf", ("Hamm", 579.0))
```

[ ]:
```
alleBesuchtenKnoten()
```

[ ]:
```
alleSchnittkanten()
```

[ ]:
```
autobahn.flagToKnoten("Wiesbaden")
autobahn.markiereKnoten("Wiesbaden", ("Erfurt", 599.0))
```

[ ]:
```
alleBesuchtenKnoten()
```

[ ]:
```
alleSchnittkanten()
```

[ ]:
```
autobahn.flagToKnoten("Mainz")
autobahn.markiereKnoten("Mainz", ("Wiesbaden", 607.0))
```

[ ]:
```
alleBesuchtenKnoten()
```

[ ]:
```
alleSchnittkanten()
```

[ ]:
```
autobahn.flagToKnoten("München")
autobahn.markiereKnoten("München", ("Dresden", 653.0))
```

[ ]:
```
alleBesuchtenKnoten()
```

1.7.5 Wir sind in München angekommen!

Wir können jetzt den Weg von Berlin nach München erkennen, indem wir quasi rückwärts laufen:

- von Dresden nach München
- von Berlin nach Dresden

Insgesamt hat der kürzeste Weg Berlin - Dresden - München eine Länge von 653.0 km

1.7.6 Wir können auch den besten Weg von Berlin nach Münster finden:

- von Hamm nach Münster
- von Bielefeld nach Hamm
- von Hannover nach Bielefeld
- von Potsdfam nach Hannover
- von Berlin nach Postdam

Nach Münster sind es also 539 km:

Berlin - Potsdam - Hannover - Bielefeld - Hamm - Münster

[ ]: