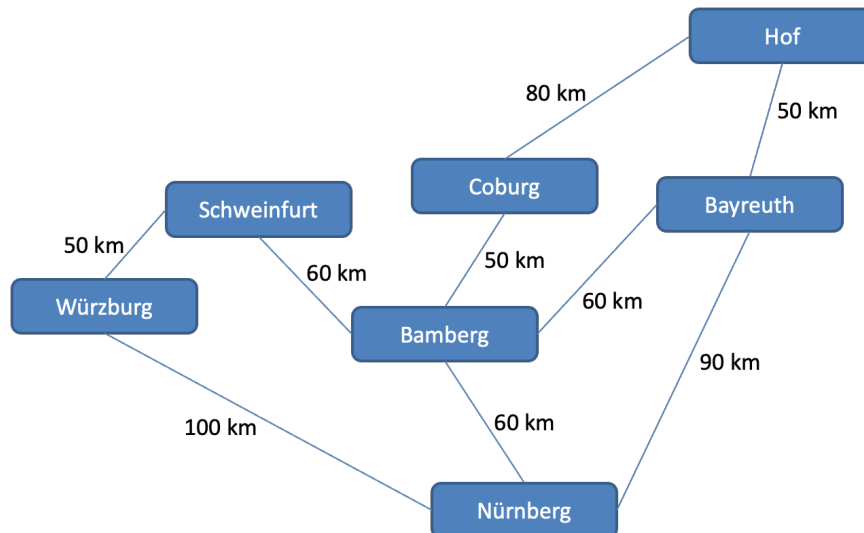


Dictionaries und Mengen

In diesem Praktikum beschäftigen wir uns mit dem berühmten "Dijkstra-Algorithmus", der den kürzesten Weg zwischen zwei Städten findet. So etwas wird z.B. in Navigationssystemen gebraucht. Der Kartenausschnitt, mit dem wir uns beschäftigen, ist dieser:



Dazu finden Sie im Moodle-Kurs eine Datei **karte.py**, in der diese Information in Form einer Liste von Tupeln vorgegeben ist:

```

connections = [('Würzburg', 'Schweinfurt', 50),
               ('Würzburg', 'Nürnberg', 100),
               ('Schweinfurt', 'Bamberg', 60),
               ('Bamberg', 'Coburg', 50),
               ('Bamberg', 'Nürnberg', 60),
               ('Bamberg', 'Bayreuth', 60),
               ('Nürnberg', 'Bayreuth', 90),
               ('Bayreuth', 'Hof', 50),
               ('Coburg', 'Hof', 80),
               ]

```

Hinweis: Diese Karte ist nur ein Beispiel. Ihre Lösung soll später auch mit beliebigen anderen Karten funktionieren, solange die Verbindungen zwischen den Städten wieder in Form einer solchen Liste vorgegeben sind.

Aufgabe 1

Öffnen Sie die Datei **karte.py** und ergänzen Sie eine Funktion

```

def allNodes(connections):
    ...

```

die alle Verkehrsknotenpunkte (Nodes) in Form einer Menge (Set!) zurückliefert, die in den übergebenen Verbindungen enthalten sind. Testen Sie die Funktion.

Aufgabe 2

Unser Dijkstra-Algorithmus benötigt die Karteninformation in Form eines Dictionaries, das für jeden Verkehrsknotenpunkt einen Eintrag besitzt. Dieser Eintrag soll wiederum ein Dictionary sein, in das wir Daten zu diesem Knotenpunkt eintragen können.

- a) Ergänzen Sie eine Funktion

```
def initMap(connections):
```

die ein entsprechendes Dictionary anlegt. Sie können dazu natürlich auch die in Aufgabe 1 entwickelte Funktion verwenden. In unserem Beispiel sollte es so aussehen (Reihenfolge der Städte ist egal):

```
{ 'Bamberg': {}, 'Hof': {}, 'Würzburg': {}, 'Coburg': {},  
  'Bayreuth': {}, 'Schweinfurt': {}, 'Nürnberg': {} }
```

- b) Nun sollen die Entfernungsangaben bei den einzelnen Verkehrsknotenpunkten hinterlegt werden. Implementieren Sie hierzu eine Funktion

```
def addConnection(map, connection):
```

so dass nach Ablauf des folgenden Programmcodes

```
map = initMap(connections)  
for con in connections:  
    addConnection(map, con)
```

das Map-Dictionary in unserem Beispiel folgenden Aufbau besitzt (Reihenfolge wieder egal):

```
{ 'Bamberg': {'connections':  
    [('Schweinfurt', 60), ('Coburg', 50), ('Nürnberg', 60),  
    ('Bayreuth', 60)] },  
  'Hof': {'connections':  
    [('Bayreuth', 50), ('Coburg', 80)] },  
  'Würzburg': {'connections':  
    [('Schweinfurt', 50), ('Nürnberg', 100)] },  
  'Coburg': {'connections':  
    [('Bamberg', 50), ('Hof', 80)] },  
  'Bayreuth': {'connections':  
    [('Bamberg', 60), ('Nürnberg', 90), ('Hof', 50)] },  
  'Schweinfurt': {'connections':  
    [('Würzburg', 50), ('Bamberg', 60)] },  
  'Nürnberg': {'connections':  
    [('Würzburg', 100), ('Bamberg', 60), ('Bayreuth', 90)] }  
}
```

Aufgabe 3

Beim Dijkstra-Algorithmus werden beginnend von einem Startpunkt wellenförmig die Entfernungen zu den anderen Knoten sukzessive eingetragen. Außerdem sollte bei jedem Knoten vermerkt werden, von welchem Vorgängerknoten man kommt, so dass am Ende der gesamte Pfad ermittelt werden kann. Diese Informationen (Entfernung vom Startpunkt, Vorgängerknoten und ein Kennzeichen, ob die Entfernung schon endgültig ist) sollen ebenfalls in unserem Dictionary eingepflegt werden. Anders als die Entfernungen, die immer gleich bleiben, müssen diese drei Werte aber vor jeder Suche nach einem kürzesten Pfad wieder zurückgesetzt werden. Wir sehen daher eine neue Funktion

```
def resetMap(map):
```

vor, die bei jedem Knotenpunkt drei Dictionaryeinträge ("distance", "predecessor", "finalized") ergänzt bzw. zurücksetzt. Der Eintrag für Nürnberg sieht anschließend in unserem Beispiel wie folgt aus:

```
'Nürnberg': {
  'connections':
    [ ('Würzburg', 100), ('Bamberg', 60), ('Bayreuth', 90)],
  'distance': None,
  'predecessor': None,
  'finalized': False }
```

Aufgabe 4

Vor der Suche nach einem kürzesten Pfad ist der Startknoten festzulegen. Die soll mit der Funktion

```
def setStart(map, start):
```

erfolgen, die das Dictionary zurücksetzt (s. Aufgabe 3) und lediglich beim Startknoten die Entfernung 0 und als Vorgänger sich selbst einträgt. Also sollte z.B. nach

```
setStart(map, "Würzburg")
```

der Eintrag für Würzburg so aussehen:

```
'Würzburg': {
  'connections': [('Schweinfurt', 50), ('Nürnberg', 100)],
  'distance': 0,
  'predecessor': 'Würzburg',
  'finalized': False
},
```

Aufgabe 5

Der Dijkstra-Algorithmus berechnet nun die kürzesten Entfernungen zu jedem Knoten wie folgt:

1. Es wird der Knoten mit der kürzesten Entfernung zum Start ausgewählt, der noch nicht finalisiert ist (dies wird beim ersten Durchlauf der Startknoten sein – alle anderen haben noch keine Entfernungsangabe).
2. Dieser gewählte Knoten wird auf "finalisiert" gesetzt, d.h. für diesen Knoten ist die kürzeste Strecke gefunden.
3. Alle Zielknoten, die vom gewählten Knoten erreicht werden können werden überprüft: Ist der Zielknoten über den aktuell gewählten Knoten schneller zu erreichen, als ggf. bisher berechnet? In diesem Fall wird die Entfernung im Zielknoten angepasst und der aktuelle Knoten als Vorgänger beim Zielknoten eingetragen.
4. Sofern es weitere noch nicht finalisierte Knoten gibt, wird mit Schritt 1 fortgefahren.

Implementieren Sie eine Funktion

```
def calculateAllDistances(map):
```

die den Dijkstra-Algorithmus in unserem Dictionary ausführt.

Aufgabe 6

Nachdem für alle Knoten die kürzesten Entfernungen berechnet sind, kann der kürzeste Pfad ermittelt werden, indem man – beginnend beim Endpunkt – jeweils den Vorgänger ermittelt und dies solange wiederholt, bis man beim Startpunkt angekommen ist (zu erkennen daran, dass der Startpunkt sich selbst als Vorgänger hat). Implementieren Sie eine Funktion

```
def pathFromTo(map, start, destination):
```

die eine Liste der Stationen vom Start bis zum Ziel entlang des kürzesten Pfads zurückgibt. Im Beispiel:

```
pathFromTo(map, "Würzburg", "Hof")
```

mit dem Ergebnis

```
['Würzburg', 'Schweinfurt', 'Bamberg', 'Bayreuth', 'Hof']
```