Kürzester Weg

Entwicklung
eines Programms
zum
Gallenbacherbeispiel

entsprechend der Algorithmusformulierung
von Gallenbacher

Kürzester Weg

Datenstruktur und Zugrifssfunktion

- Wahl der Datenstruktur, Alternativen sind
  - Kantenliste
  - Knotenliste
- Hier gewählt: Knotenliste, allerdings nicht in Form einer Liste, sondern als ***Dictionary***

Kürzester Weg

```
gallenbacherGraph={
    'A' : [['B', 69], ['D', 36], ['M', 36], ['N', 22]],
    'B' : [['A', 69], ['D', 64], ['Z', 32], ['H', 30], ['I', 34],
['X', 26]],
    'C' : [['I', 40], ['M', 31], ['X', 23]],
    'D' : [['A', 36], ['B', 64], ['L', 95], ['N', 20]],
    'E' : [['F', 79], ['G', 60], ['K', 31], ['O', 102]],
    'F' : [['E', 79], ['K', 29], ['O', 14], ['P', 57]],
    'G' : [['E', 60], ['K', 58], ['L', 58], ['Y', 30]],
    'H' : [['B', 30], ['I', 65], ['K', 31], ['L', 106], ['P', 34],
['Y', 29]],
    'I' : [['B', 34], ['C', 40], ['H', 65], ['M', 43], ['P', 55]],
    'K' : [['E', 31], ['F', 29], ['G', 58], ['H', 31], ['P', 25],
['Y', 20]],
    'L' : [['D', 95], ['G', 58], ['H', 106], ['Z', 40]],
    'M' : [['A', 36], ['C', 31], ['I', 43], ['X', 12]],
    'N' : [['A', 22], ['D', 20], ['X', 29], ['Z', 30]],
    'O' : [['E', 102], ['F', 14], ['P', 91]],
    'P' : [['F', 57], ['H', 34], ['I', 55], ['K', 25], ['O', 91]],
    'X' : [['B', 26], ['C', 23], ['M', 12], ['N', 29]],
    'Y' : [['G', 30], ['H', 29], ['K', 20], ['Z', 23]],
    'Z' : [['B', 32], ['L', 40], ['N', 30], ['Y', 23]]
```

Kürzester Weg

- Durch die Wahl des Dictionary ist die Teilaufgabe
  Bestimmung der Nachfolgekanten zu einem Knoten
  besonders einfach:
- Der Aufruf `gallenbacherGraph['A']` liefert beispielsweise

  `[['B', 69], ['D', 36], ['M', 36], ['N', 22]]`

  und damit direkt eine Beschreibung der Nachfolgekanten zum Knoten 'A'.

Kürzester Weg

- Wichtige grundlegende Entscheidung ist der Einsatz einer Prioritätswarteschlange als weitere Datenstruktur.
- Sie kann - wie hier beschrieben – objektorientiert realisiert werden oder durch passende Anwendung auf einer (*normalen*) Liste, wie in der gesonderten Präsentation ***Prioritaetswarteschlange-Py*** beschrieben.

- Klassenkopf und Zugriffsmethoden:

Kürzester Weg

```
class prioSchlange():
    def __init__(self, initWeg):
        self.wege=[initWeg]

    def gibWege(self):
        return self.wege

    def wegLaenge(self, weg):
        laenge=0
        for kante in weg:
            laenge+=kante[1]
        return laenge

    def ersterRaus(self):
        temp = self.wege[0]
        self.wege=self.wege[1:]
        return temp
```

Kürzester Weg

```
def fuegeEin(self, bisher, dazu):
    neuerWeg=[dazu]
    for kante in bisher:
        neuerWeg.append(kante)
    if len(self.wege)==0:
        self.wege=[neuerWeg]
        return
    #  Ort schon enthalten ?
    for weg in self.wege:
        if neuerWeg[0]==weg[0]:
            # es gibt schon einen Weg zum Knoten
            if self.wegLaenge(neuerWeg)>=self.wegLaenge(weg):
                # dann ist nichts zu tun
                return
            else:
                # der alte Weg muss entfernt werden
                self.wege.remove(weg)
    position=len(self.wege)
    for weg in self.wege:
        if self.wegLaenge(weg)>self.wegLaenge(neuerWeg):
            position=self.wege.index(weg)
            break
    self.wege.insert(position,neuerWeg)
```

Kürzester Weg

- Die hier beschriebene Methode fuegeEin gewährleistet, dass der neue Weg in die Prioritätswarteschlange an der richtigen Position eingebaut wird, wenn es denn keinen kürzeren Weg zu dem aktuellen Knoten gibt.
- Nachfolgende längere Wege werden nicht gesucht, da sie gegebenenfalls bei nachfolgenden Schritten entfernt werden.

- Dijkstra selbst ist dann überschaubar:

Kürzester Weg

```
def dijkstra(rot, wege, nachfolgeKanten, ziel):
    while len(wege.gibWege())>0:
        aktuellerWeg=wege.ersterRaus()
        if ziel==aktuellerWeg[0][0]: return aktuellerWeg
        for kante in nachfolgeKanten:
            if not (kante[0] in rot):
                wege.fuegeEin(aktuellerWeg,kante)
        neuer = wege.gibWege()[0][0][0]
        if not neuer in rot:
            rot.append(neuer)
        nachfolgeKanten=gallenbacherGraph[neuer]
    return False
```

Kürzester Weg

- Die Funktion arbeitet intern vollständig iterativ.
- In der for-Schleife werden zu allen nachfolgenden Kanten die fortsetzenden Wege, deren aktueller Endpunkt nicht bereits (*und damit kürzer*) erreicht wurde, in die Prioritätswarteschlange (wege) eingefügt.
- Dies wird (*while-Schleife*) solange gemacht, bis das gesuchte Ziel erreicht wurde oder (*Misserfolg*) keine Wege mehr zur Verfügung stehen.