Von Dijkstra zu Prim

minimal spanning tree

ausgehend von einem
Einstieg (*bzw vorheriger Bearbeitung*)
über den Algorithmus von Dijkstra

© Claus Albowski

Von Dijkstra zu Prim

- Voraussetzung:
  - Algorithmus von Dijkstra ist bekannt und verstanden
- Problemstellung
  - Minimales Versorgungsnetz
- Vorgehen
  - Erarbeitung und
  - Formulierung des Algorithmus von Prim

© Claus Albowski

Von Dijkstra zu Prim

Einstieg:

- Beginne mit einem Baum, der allein aus einem Knoten besteht (*egal, welcher Knoten*).
- Ordne alle Kanten von diesem Knoten aus nach ihren Kosten. (→ Prioritätswarteschlange nach den Bewertungen)

© Claus Albowski

Von Dijkstra zu Prim

Wiederholungschritt

- Füge nun aus dieser Prioritätswarteschlange jeweils immer die Kante mit den geringsten Kosten (*also die kürzeste*) ein, die zu einem noch nicht erreichten Knoten führt (*keinen Zyklus erzeugt*).
- Dann füge alle vom neuen freien Knoten dieser Kante ausgehenden zulässigen Kanten in die Prioritätswarteschlange ein.
- Brich damit ab, wenn alle Knoten des Graphen zum Baum gehören.

© Claus Albowski

Von Dijkstra zu Prim

Aufgaben:

- Einordnen der Kanten vom aktuellen Knoten aus in die Prioritätswarteschlange nach ihren Bewertungen
- Bestimmung der Kanten, die von dem neuen Knoten ausgehen
- Prüfen, ob der freie Knoten (*Zielknoten der Kante*) in der Besucht-Liste enthalten ist
- Prüfen, ob alle Knoten im Baum enthalten sind
- die eigentliche Algorithmussteuerung

© Claus Albowski

Von Dijkstra zu Prim

Daten als „Knotenliste“, besser Dictionary in *gallenbacher-graph.py*

- Die Datenstruktur benötigt keine besondere Zugrifssfunktion

```
gallenbacherGraph['A']
```

liefert beispielsweise

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

© Claus Albowski

Von Dijkstra zu Prim

Prioritätswarteschlange

- Die Bearbeitung zum Dijkstra-Programm liefert die Verwaltung der OO Prioritätswarteschlange aus gallenbacher_dijkstra_iterativ.py
  - auf
    `fuegeAlleEin(alle, vor)`
    wird von außen zugegriffen
  - intern wird
    `fuegeEineEin(eine, vor)`
    verwendet

© Claus Albowski

Von Dijkstra zu Prim

- Die Funktion **`vor`** (das Prädikat), konkret **`kuerzer`**

```
def kuerzer( kante_1 , kante_2 )
    return kante_1[2] < kante_2[2]
```

Die Kanten werden also verwendet in der Form:
(<Knoten-1> <Knoten-2> <Bewertung>)

© Claus Albowski

Von Dijkstra zu Prim

- Alternativ: **`kuerzer`** mit Hilfsfunktion **`laenge`**

```
def kuerzer( kante_1 , kante_2 )
    return laenge(kante_1) < laenge(kante_2)

def laenge( kante )
    return kante[2]
```

Die Kanten werden also verwendet in der Form:
(<Knoten-1> <Knoten-2> <Bewertung>)

© Claus Albowski

Von Dijkstra zu Prim

Erarbeitung der Funktion **`AvPrim`**

- Kopf und interne Definitionen

```
def AvPrim(startknoten):
    baum=[]
    enthalteneKnoten=[startknoten]
    neuerKnoten=startknoten
    prioSchlange=PrioSchlange()
    . . .
```

© Claus Albowski

Von Dijkstra zu Prim

Erarbeitung der Funktion **`AvPrim`** weiter

```
while len(enthalteneKnoten)<len(graph):
    prioSchlange.fuegeAlleEin(
                 nachfolgeKanten(neuerKnoten),
                 kuerzer)
    prioSchlange.entferneZyklen(enthalteneKnoten)
    # Die erste Kante setzt den Baum minimal fort
    erste=prioSchlange.ersteRaus()
    baum.append(erste)
    if erste[0] in enthalteneKnoten:
        enthalteneKnoten.append(erste[1])
        neuerKnoten=erste[1]
    else:
        enthalteneKnoten.append(erste[0])
        neuerKnoten=erste[0]
return baum
```

© Claus Albowski

Von Dijkstra zu Prim

Hilfsfunktion **`nachfolgeKanten`**

- Vervollständigt zusätzlich die Kanten zu *[<Knoten 1>, <Knoten 2>, <Laenge>]*

```
def nachfolgeKanten(knoten):
    kanten=[]
    nachfolger=graph[knoten]
    for nach in nachfolger:
        kanten.append([knoten]+nach)
    return kanten
```

© Claus Albowski

Von Dijkstra zu Prim

Hilfsfunktion `laengeAllerKanten`

```
def laengeAllerKanten(kanten):
    laenge=0
    for kante in kanten:
        laenge+=kante[2]
    return laenge
```

© Claus Albowski