Problem des minimalen Versorgungsnetzes

Beschreibung des Problems

Zu mehreren Orten, die untereinander mit unterschiedlich langen Kanten verbunden sind, soll ein Versorgungsnetz mit minimalem Aufwand generiert werden.

Modellierung des Problems

Das Problem wird mit einem Graphen beschrieben, dessen Ecken für die zu versorgenden Orte stehen und deren Kanten die Wege zwischen ihnen darstellen. Die Kanten sind mit einer Zahl für den jeweiligen Herstellungsaufwand bewertet.
Gesucht ist zu diesem Graphen ein minimaler aufspannender Baum [minimal spanning tree].

Entwicklung des Algorithmus

Hierfür gibt es zwei Ansätze:

Der Algorithmus von Prim

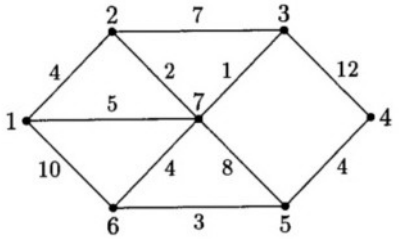
- Beginne mit einem Baum, der allein aus einer Ecke besteht.
- Ordne alle Kanten von dieser Ecke aus nach ihren Kosten. (→ Prioritätswarte-schlange nach den Bewertungen)
- Füge nun jeweils immer aus dieser jeweils die nächste Kante mit den geringsten Kosten ein, die keinen Zyklus erzeugt. Dann füge alle von der freien Ecke ausgehenden zulässigen Kanten in die Prioritätswarteschlange ein.
- Brich damit ab, wenn alle Ecken (Knoten) des Graphen zum Baum gehören.

Der Algorithmus von Kruskal

- Ordne alle Kanten nach ihren Kosten.
- Beginne mit einem Baum, der allein aus einer Kante mit minimalen Kosten besteht.
- Füge dann jeweils immer die nächste Kante mit den geringsten Kosten ein, die keinen Zyklus erzeugt.
- Brich damit ab, wenn alle Ecken (Knoten) des Graphen zum Baum gehören.

Datenmodellierung

Den Graphen wird man in Python am besten in einem Dictionary abspeichern, da man dann sehr einfach zu einem gegebenen Knoten *[key]* die von ihm ausgehenden Kanten, also die Nachfolgekanten beim Suchverfahren, auslesen kann.
Die Nachfolgekanten *[values]* selbst sind in einer Liste gespeichert. Das Muster lautet also:
{'Ort': [['Zielort', Bewertung], …], …}
Für den Turau-Graphen erhält man:




```
mustergraph={ 1: [[2,4],[6,10], [7,5]],
              2: [[1,4],[3,7],[7,2]],
              3: [[2,7],[4,12],[7,1]],
              4: [[3,12],[5,4]],
              5: [[4,4],[6,3],[7,8]],
              6: [[1,10],[5,3],[7,4],[7,4]],
              7: [[1,5],[2,2],[6,4],[3,1],[5,8],[6,4]]
    }
```

Funktionale Modellierung des Algorithmus von Prim

In der hier vorgestellten Lösung wird

- einerseits die Originalformulierung des Algorithmus nach Turau umgesetzt und
- andererseits dabei weitgehend funktional gearbeitet.

Funktionskopf und Rückgabe

Die Aufruffunktion muss den Graphen übergeben bekommen. Rückgabewert ist der minimal spanning tree. Um die Funkion einfacher zu schreiben, kann man den Startort vorgeben. Anderenfalls müsste sie sich einen der Orte des Graphen zufällig auswählen. Da die Wahl des Startortes keinen Einfluss auf die Lösung hat, kann man auch vorgeben, dass die Funktion den ersten Ort wählt.

```
def AvPrim(graph, startEcke):
    < . . . >
    return baum
```

Lokale Variable

Obwohl das sicher nicht funktional ist, stelle ich hier zunächst eine Funktion vor, die mit lokalen Variablen arbeitet.

```
    baum=[]
    enthalteneEcken=[startEcke]
    neueEcke=startEcke
    prioSchlange=[]
```

Nun wird eine Schleife ausgeführt, bis die Anzahl der Ecken mit der Anzahl der Knoten im Baum übereinstimmt.
Darin werden zunächst die Nachfolgekanten zur neuen Ecke erzeugt und nacheinander in die Prioritätswarteschlange eingefügt.
Kanten, die Zyklen erzeugen, werden entfernt. Sie sind daran zu erkennen, dass beide Ecken schon in den enthaltenen Ecken vorhanden sind.

```
    while len(enthalteneEcken)<len(graph):
        prioSchlange=entferneZyklen\
                     (ordneAlleEin\
                      (nachfolgeKanten(neueEcke, graph),\
                       prioSchlange),\
                      enthalteneEcken)
```

Nun ist sicher, dass die erste Kante in der **`prioSchlange`** den Baum minimal fortsetzt. Sie wird in den Baum aufgenommen, die freie Ecke ist die neueEcke, die auch den enthaltenen Ecken hinzugefügt wird.

```
        neueEcke, baum, enthalteneEcken, prioSchlange=\
                  kanteEinbauen(baum, enthalteneEcken, prioSchlange)
```

In diesem Programmabschnitt wird ausgenutzt, dass Python Mehrfachzuweisungen zulässt. Die Hilfsfunktion **`kanteEinbauen`** [dazu s.u.] muss also die Ergebnis-Werte für **`neueEcke`**, **`baum`**, **`enthalteneEcken`** und **`prioSchlange`** zurückgeben, damit im zweiten Durchlauf damit weiter gearbeitet werden kann.

Nach Abbruch der Schleife wird der Baum ausgegeben.

```
    return baum
```

Hilfsfunktionen

Auffällig war der Zugriff auf mehrere Hilfsfunktionen. Sie werden im folgenden Teil beschrieben.

----- ordneAlleEin -----

Die Funktion ordnet eine Liste von Kanten in die die Prioritätswarteschlange ein. Grundsätzlich entsteht hier eine Funktion zum Einfügen in eine sortierte Liste. Sie benötigt wiederum eine eigene Hilfsfunktion zum Einfügen einer einzelnen Kante.

```
def ordneAlleEin(kanten, sortierte):
    for kante in kanten:
        sortierte=ordneEin(kante, sortierte)
    return sortierte
```

----- ordneEin -----

Die Funktion ordnet eine einzelne Kante ein. Auch sie benötigt eine weitere Hilfsfunktion, **kuerzer**, zum Vergleich von zwei Kanten.

```
def ordneEin(kante, sortierte):
    for index in xrange(len(sortierte)):
        if kuerzer(kante, sortierte[index]):
            sortierte[index:index]=[kante]
            return sortierte
    sortierte[len(sortierte):len(sortierte)]=[kante]
    return sortierte
```

----- kuerzer -----

Die Funktion **kuerzer** ist also das Prädikat zum Vergleich von Kanten.

```
def kuerzer(kante1, kante2):
    return  (kante1[2] < kante2[2])
```

----- nachfolgeKanten -----

Die Funktion bestimmt zu einer Ecke die Nachfolgekanten.

```
def nachfolgeKanten(ecke, graph):
    kanten=[]
    nachfolger=graph[ecke]
    for nach in nachfolger:
        kanten+=[[ecke]+nach]
    return kanten
```

----- entferneZyklen -----

Die Funktion entfernt Kanten, die einen Zyklus erzeugen.

```
def entferneZyklen(prioSchlange, enthalteneEcken):
    temp=[]
    for kante in prioSchlange:
        if not (kante[0] in enthalteneEcken):
            temp.append(kante)
        elif not (kante[1] in enthalteneEcken):
            temp.append(kante)
```

```
    return temp
```

----- kanteEinbauen -----

Die Funktion baut die Kante ein und aktualisiert die anderen Daten.

```
def kanteEinbauen(baum, enthalteneEcken, prioSchlange):
    baum.append(prioSchlange[0])
    if prioSchlange[0][0] in enthalteneEcken:
        enthalteneEcken.append(prioSchlange[0][1])
        neueEcke=prioSchlange[0][1]
    else:
        enthalteneEcken.append(prioSchlange[0][0])
        neueEcke=prioSchlange[0][0]
    prioSchlange.remove(prioSchlange[0])
    return neueEcke, baum, enthalteneEcken, prioSchlange
```

===== Testaufruf =====

Mögliche Testaufrufe:

```
AvPrim(mustergraph, 1)
```

liefert:

```
[[1, 2, 4], [2, 7, 2], [7, 3, 1], [7, 6, 4], [6, 5, 3], [5, 4, 4]]
```

und

```
import gallenbacher_graph
AvPrim(gallenbacher_graph.gallenbacherGraph, 'I')
```

liefert:

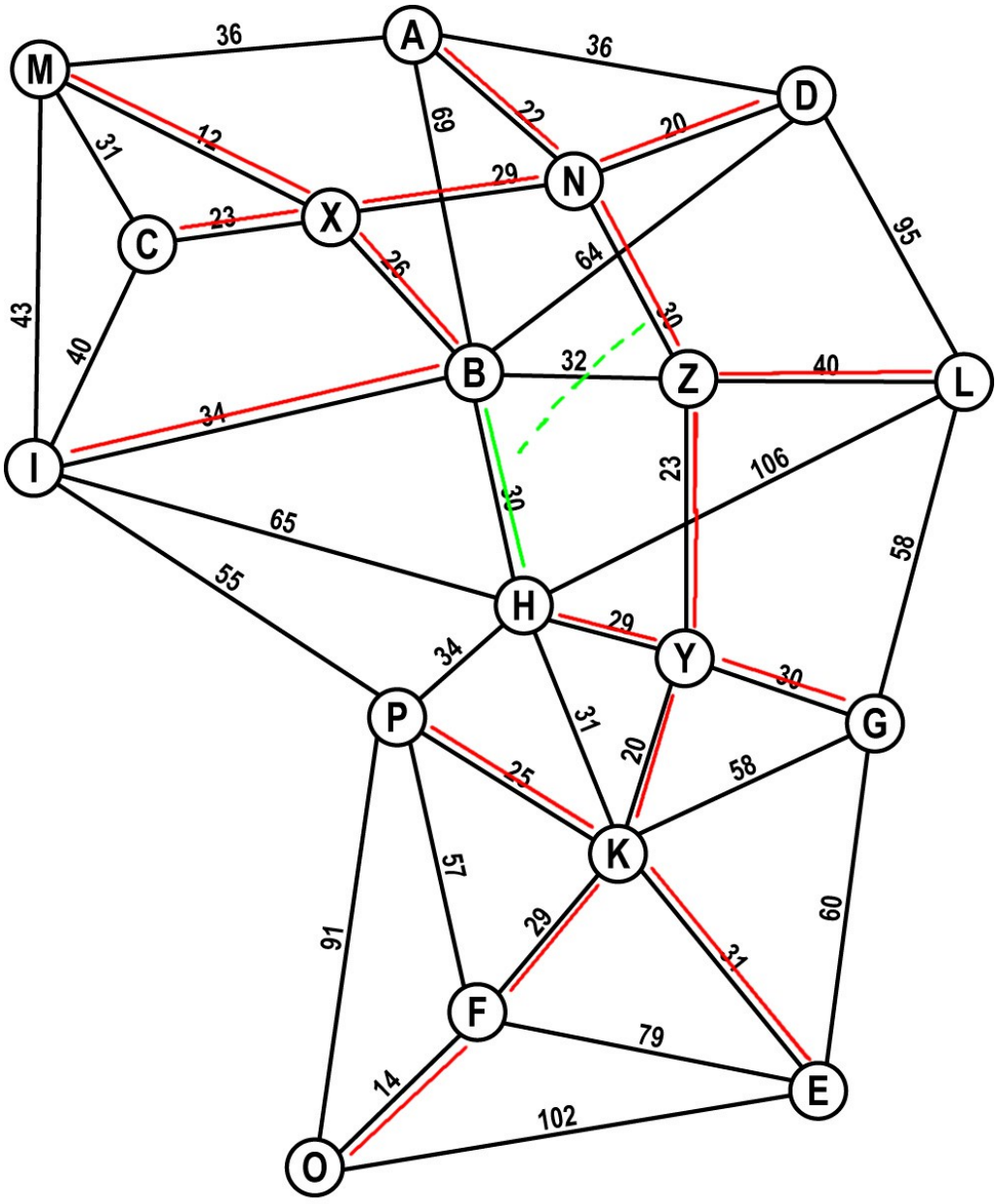
```
[['I', 'B', 34], ['B', 'X', 26], ['X', 'M', 12], ['X', 'C', 23], ['X',
'N', 29], ['N', 'D', 20], ['N', 'A', 22], ['B', 'H', 30], ['H', 'Y', 29],
['Y', 'K', 20], ['Y', 'Z', 23], ['K', 'P', 25], ['K', 'F', 29], ['F',
'O', 14], ['Y', 'G', 30], ['K', 'E', 31], ['Z', 'L', 40]]
```

Nicht funktional!

Die eigentliche Suchfunktion ist nicht funktional geschrieben. Eine (fast) vollständig funktionale Variante ist hier ohne Erläuterung angegeben.

```
# ----- Schritt -----
# Der eigentliche Algorithmus
def Schritt(neueEcke, baum, enthalteneEcken, prioSchlange, graph):
    if len(enthalteneEcken)<len(graph):
        prioSchlange=entferneZyklen\
                     (ordneAlleEin\
                      (nachfolgeKanten(neueEcke, graph),\
                       prioSchlange),\
                      enthalteneEcken)
        neueEcke, baum, enthalteneEcken, prioSchlange=\
                  kanteEinbauen(baum, enthalteneEcken, prioSchlange)
        return Schritt(neueEcke, baum, enthalteneEcken, prioSchlange, graph)
    else:
        return baum

# ----- AvPrim -----
# Aufrufhülle für die Rekursion
def AvPrim(graph, startEcke):
    return Schritt(startEcke, [], [startEcke], [], graph)
```



Aus: Gallenbacher, *Abenteuer Informatik, 2. Aufl.*
© Spektrum Akademischer Verlag GmbH 2008