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 die von ihm ausgehenden Kanten, also die Nachfolgekanten beim Suchverfahren, auslesen kann. Die Nachfolgekanten selbst sind in einer Liste gespeichert.
{'Ort': [['Zielort', Bewertung], …], …}

Funktionale Modellierung des Algorithmus

Prim

Die Aufruffunktion muss den Graphen übergeben bekommen. Rückgabewert ist der minimal spanning tree. Um die Funkion einfacher zu schreiben, kann man den Startort vorgeben.

```
def AvPrim(graph, startEcke):
    …
```

Die Funktion muss zunächst die lokalen Variablen definieren:

```
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.
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.
Nach Abbruch der Schleife wird der Baum ausgegeben.

```
# Graph von Turau:
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]]}

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

# ----- ordneEin -----
# sortiert durch Einfügen
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

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

# ----- Der eigentliche Algorithmus -----
# Aufrufhülle für die Rekursion
def AvPrim(graph, startEcke):
    baum=[]
    enthalteneEcken=[startEcke]
    neueEcke=startEcke
    prioSchlange=[]
    while len(enthalteneEcken)<len(graph):
        for kante in nachfolgeKanten(neueEcke, graph):
            prioSchlange=ordneEin(kante, prioSchlange)
        # Entferne alle Zyklen:
        temp=[]
        for kante in prioSchlange:
            if not (kante[0] in enthalteneEcken):
                temp.append(kante)
            elif not (kante[1] in enthalteneEcken):
                temp.append(kante)
        prioSchlange=temp
        # Die erste Kante setzt den Baum minimal fort
        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 baum
```