Suchprobleme

In diesem Abschnitt werden nur einige Suchprobleme und Suchverfahren exemplarisch beschrieben und die Eignung von Python zu ihrer Behandlung untersucht.

Routenplaner

Eine gute Möglichkeit der Einführung in die Problemstellung bietet das Material von Gallenbacher[1] zum Routenplaner. Seine Intention ist, Informatik und Algorithmen ohne Computerunterstützung kennenzulernen [*"Informatik ohne Computer"*]. Selbst wenn man darauf nicht verzichten möchte, bietet das Buch eine interessante Möglichkeit des Einstiegs in viele Themenbereiche der Informatik.
Gallenbacher gibt im Abschnitt 1 *"Sag' mir wohin"* eine Miniwelt[2] durch eine Landkarte vor und führt schrittweise Modellierung und Abstraktion durch. Er nutzt das Modell eines Ameisenstammes für das Suchen der kürzesten Verbindung zu einem Zielort und entwickelt damit eine Anwendung des Algorithmus von Dijkstra.

Erarbeiten einer Lösung

Als Einstieg kann man die Aufgabe stellen, eine Lösung selbst herauszufinden. Eine Reflexionsphase mit der Frage *"Wie sind Sie vorgegangen?"* führt vermutlich auf eine Variante eines brute-force – Verfahrens [*ich untersuche alle möglichen Wege*] oder eines greedy Verfahrens [*ich gehe einen für gut gehaltenen Weg zu Ende*] und gibt die Möglichkeit an ihr die Probleme von brute-force oder greedy Verfahren zu untersuchen. Beim Reduzieren der Informationen auf das für das Problem Wesentliche kann man die Fachbegriffe Graph, Knoten [Ecken] und Kanten einführen.
Mit dem Modell des Ameisenstammes kann man nun schrittweise zu einer optimalen Lösung gelangen und kommt durch Reflexion der durchgeführten Schritte zu einer Formulierung des Algorithmus.
In der Reflexionsphase sollte man darauf eingehen, dass der Algorithmus von Dijkstra immer die kürzesten Wege von einem Startpunkt zu vielen anderen Punkten bestimmt. So kann man untersuchen, wie das Navigationssystem es nach der erstmaligen Berechnung des Weges schafft, in kurzer Zeit ohne eine Neuberechnung auf ein Verhalten des Fahrers zu reagieren, der sich nicht an den vorgegebenen optimalen Weg gehalten hat

Realisierung mit Python

Will man die Lösung auch mit Python umsetzen, dann muss man zunächst ein Datenmodell für das Problem finden. In der hier vorliegenden Beispiellösung wurde ein Dictionary verwendet. Die Anwendung des Typs Dictionary zur Datenmodellierung des Graphen entspricht dem Verwenden einer Assoziationsliste bei Scheme. Weiter wäre eine Berechnung des Aufwandes bei brute-force (exponentiell) und Dijkstra (quadratisch) im Vergleich möglich.

Datenmodell

```
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]],
```

1 Buch und CD: Gallenbacher; Abenteuer Informatik
2 Zu den bildlichen Darstellungen sehe man in seinem Buch nach.

```
    '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', 30]],
    'Z' : [['B', 32], ['L', 40], ['N', 30], ['Y', 23]]
}
```

Zugriffe auf die Daten:

```
>>> gallenbacherGraph['C']
[['I', 40], ['M', 31], ['X', 23]]
```

Programm

Das Programm arbeitet mit einer Prioritätswarteschlange zur Verwaltung der bisher gegangenen Wege.

```
start='I'
ziel='O'

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

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

    def fuegeEin(self, bisher, dazu):
        neuerWeg=[dazu]
        for kante in bisher:
            neuerWeg.append(kante)
        if len(self.wege)==0:
            self.wege=[neuerWeg]
            return
        # zunaechst muss getestet werden, ob der Ort schon enthalten ist.
        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)

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

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

Ein Testaufruf:

```
print dijkstraa([start], prioSchlange([[start,0]]),
gallenbacherGraph[start], ziel)
```

liefert:

```
[['O', 14], ['F', 29], ['K', 25], ['P', 55], ['I', 0]]
```

Container-Lade-Problem

Das Container-Lade-Problem kann einen zusammenhängenden Kontext für die Untersuchung von Suchverfahren bilden. Es gehört einerseits zu den am schwierigsten zu lösenden np-vollständigen Problemen und bietet daher die Möglichkeit, sich mit deren Problemen und intelligenten Verfahren zu ihrer Bearbeitung beschäftigen. Andererseits lässt es sich aber zum Einstieg auf das einfache Rucksackproblem vereinfachen, an dem die Eignung der grundlegenden Verfahren untersucht werden kann. Das einfache Rucksackproblem lässt sich so beschreiben: Packe von einer Anzahl Stücke, die sich nur in einer Größe [Gewicht, Länge, Volumen, …] unterscheiden möglichst viel in einen Behälter, dessen Fassungsvermögen allein durch diese Größe bestimmt ist.

Das Rucksackproblem, greedy und vollständig mit Tiefensuche

Greedy ist sehr schnell und einfach

Wir geben die Stücke als Liste von Zahlen vor, der Container [Rucksack, …] ist durch einen Zahlenwert vorgegeben. Die Stückeliste umfasse die Teile [30, 30, 30, 20, 20, 20, 20] und der Container habe ein Fassungsvermögen von 80.
Die einfüllende Person wählt vermutlich nacheinander Stücke zum Einfüllen aus und füllt sie in den Container ein, wenn sie noch hinein passen. Kann der Container vollständig gefüllt werden?

```
# Globale Variable für die Stücke:
stuecke=[30,30,30,20,20,20,20]
container=[][1]

# Prädikat zum Testen, ob der Container voll ist.
def istVoll(container, fassungsvermoegen):
    return fassungsvermoegen==sum(container)
```

1 Siehe Bemerkung weiter unten!

```
def istZuVoll(container, fassungsvermoegen):
    return fassungsvermoegen<sum(container)

def fuelle(stuecke, container, fassungsvermoegen):
    while len(stuecke)!=0:
        container.append(stuecke[0])
        stuecke=stuecke[1:]
        if istVoll(container, fassungsvermoegen):
            return container
        elif istZuVoll(container, fassungsvermoegen):
            container=container[:len(container)-1]
    return False
```

Der Aufruf von:

```
>>> fuelle(stuecke, container, 80)
[30, 30, 20]
```

liefert eine vollständige Füllung.

Eine wichtige Anmerkung zum Arbeiten mit Listen bei Python: Ein erneuter Aufruf liefert nun:

```
>>> fuelle(stuecke, container, 80)
False
```

Woran liegt das? Das Programm arbeitet mit einer globalen Variablen vom Typ Liste. Jeder schreibende Zugriff darauf verändert aber diese Liste dauerhaft. Man sollte also nicht mit einer globalen Variablen für den Container arbeiten, sondern den leeren Container jeweils neu übergeben:

```
>>> fuelle(stuecke, [], 80)
[30, 30, 20]
>>> fuelle(stuecke, [], 80)
[30, 30, 20]
```

War das jetzt schon die Lösung? Für dieses Beispiel hat unser Verfahren eine Lösung ergeben. Es war auch nicht verlangt, alle möglichen Lösungen zu finden. Tatsächlich gibt es ja bei diesen Stücken und dem Container weitere. Die Frage muss man anders stellen: Finden wir nach diesem Algorithmus immer eine Lösung, also auch bei anderen Stücken und/oder anderen Fassungsvermögen?

Die Aufgabe, nach diesem Algorithmus die Füllung bei einem Fassungsvermögen von 100 zu versuchen, macht das Problem deutlich.

```
>>> fuelle(stuecke, [], 100)
False
```

Das Programm füllt nämlich das dritte Stück 30-er-Stück ein und damit kann eine Füllung von 100 nicht mehr erreicht werden, wenn kein 10-er-Stück vorhanden ist, obwohl mit den Stücken [30,30,20,20] sehr wohl eine Lösung möglich wäre.

Von greedy zu vollständigen Suchverfahren

Der Schritt zu einem vollständigen Suchverfahren, nämlich der Tiefensuche mit backtracking ist nun sehr einfach. Es muss eine Möglichkeit geschaffen werden, das Einfüllen eines Stückes auch dann rückgängig zu machen, wenn es prinzipiell zulässig war und das Programm dazu zu bringen, zu dem Bearbeitungsschritt zurück zu gehen, an dem diese Entscheidung gefällt wurde.

Hier die Erweiterung auf die Tiefensuche mit backtracking, bei der man zu einer rekursiven Lösung wechseln sollte.

```
def fuelle(stuecke, container, fassungsvermoegen):
```

```
    if istVoll(container, fassungsvermoegen):
        return container
    elif len(stuecke)==0:
        return False
    elif istZuVoll(container, fassungsvermoegen):
        return False
    else:
        ergebnis=fuelle(stuecke[1:], container+stuecke[:1], fassungsvermoegen)
        if ergebnis:
            return ergebnis
        else:
            return fuelle(stuecke[1:], container, fassungsvermoegen)
```

Die Testaufrufe ergeben befriedigende Ergebnisse.

```
>>> fuelle(stuecke, [], 80)
[30, 30, 20]
>>> fuelle(stuecke, [], 100)
[30, 30, 20, 20]
```

minimal spanning tree

Eine Betrachtung des Aufwandes zeigt, dass man viele Probleme wegen zu großer Datenmengen und/oder zu großer Laufzeit mit vollständigen Suchverfahren nicht bewältigen kann [s.u.]. Am anderen "Ende" der Skala des Aufwandes kann man Probleme wie das Problem des minimalen Versorgungsnetzes betrachten, die man mit greedy Verfahren wirklich lösen kann.
Für den Aufbau eines solchen Versorgungsnetzes fallen Kosten an, die in der Regel zumindest von der Länge der Teilverbindungen abhängig sind. Bilden wir einen Graphen zur Beschreibung dieses Netzes, dann sind den Kanten diese Kosten zuzuordnen. Allgemein wird für die Kantenbewertung in Graphen daher häufig der Begriff Kosten verwendet. Das Ziel in vielen Anwendungsfällen wird sein, die durch das Netz entstehenden Gesamtkosten zu minimieren. Ein minimales Versorgungsnetz stellt immer einen Baum[1] dar.
Man kann zur Bestimmung eines minimal spanning tree den Algorithmus von Prim oder den von Kruskal betrachten und beide Algorithmen können auch gut von den Schülerinnen und Schülern selbst entwickelt werden. Beispielhaft ist hier nur der Algorithmus von Kruskal betrachtet.

Der Algorithmus von Kruskal[2]

- 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.

Man kann nun sehr leicht zeigen, dass der hier beschriebene Algorithmus eine Lösung findet, wenn es überhaupt Lösungen geben kann. Beim Aufbau des Baumes können Teilgraphen entstehen, die nicht zusammenhängend sind, letztlich müssen aber alle

1 Ein Graph ist ein Baum, wenn es zwischen je zwei seiner Ecken genau einen Weg gibt, in ihm also keine Zyklen auftreten.
Ein aufspannender Baum verbindet alle Knoten des Graphen.
2 Siehe dazu auch Turau: Algorithmische Graphentheorie

Ecken mit dem Graphen verbunden sein. Anderenfalls gäbe es keine Kante von dieser Ecke zu irgendeiner Ecke des Baumes, was der Forderung nach einem zusammenhängenden Ausgangsgraphen widerspricht.

Wird eine Kante hinzugefügt, dann hat sie entweder

- keine Ecke mit dem derzeitigen Teilgraphen gemeinsam, die Kante ist dann zur Zeit noch isoliert

oder sie hat

- eine Ecke mit dem derzeitigen Teilgraphen gemeinsam, dann setzt sie einen Ast fort oder bildet einen neuen

oder sie

- verbindet zwei bis dahin noch isolierte Teile des Teilgraphen mit einander, nur dann hat sie zwei Ecken mit dem derzeitigen Teilgraphen gemeinsam.

Weil anderenfalls ein Zyklus auftritt, ist es nicht möglich, dass zwei Ecken gemeinsam zu einem zusammenhängenden Abschnitt gehören. Das angegebene Programm löst die gestellte Aufgabe. Es ist allerdings stark an der entsprechenden Lösung mit Scheme orientiert und nicht optimiert auf Python [was hier kein Nachteil ist].

Programm Kruskal

```
# Graph von Turau:
mustergraph=[[1,2,4],[2,3,7],[3,4,12],[4,5,4],[5,6,3],[6,1,10],[1,7,5],[2,7,2],
[6,7,4],[3,7,1],[5,7,8],[6,7,4]]
# umkehrung=[[2,1,4],[3,2,7],[4,3,12],[5,4,4],[6,5,3],[1,6,10],[7,1,5],[7,2,2],
[7,6,4],[7,3,1],[7,5,8],[7,6,4]]
## Kanten: [<ecke-1>, <ecke-2>, <bewertung>]

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

# ----- leer -----
# Hilfsfunktion für leere Liste
def leer(liste): return len(liste)==0

# ----- rest -----
# Hilfsfunktion fuer Restliste
def rest(liste):
    return liste[1:]

# ----- ordneKanten -----
def ordneKanten(kanten):
    sortierte=[]
    for kante in kanten:
        sortierte=ordneEin(kante, sortierte)
    return sortierte

# ----- ordneEin -----
# sortiert durch einfuegen
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

# ----- eckeEnthalten -----
#  prüft, ob eine Ecke in der Kantenliste enthalten ist.
def eckeEnthalten(ecke, kanten):
    for kante in kanten:
        if ecke==kante[0]: return True
        elif ecke==kante[1]: return True
    return False

# ----- alle-ecken-enthalten -----
# prüft, ob alle Ecken im ersten Teilbaum in der Liste der Baeume enthalten
sind.
def alleEckenEnthalten(ecken, baeume):
    for ecke in ecken:
        if not eckeEnthalten(ecke, baeume[0]): return False
    else: return True

# ----- alleEcken -----
# extrahiert aus einem Baum alle Ecken
def alleEcken (baum):
    m=set([])
    for kante in baum:
        m.add(kante[0])
        m.add(kante[1])
    return list(m)

# ----- inTeilbaeumen -----
# gibt die Teilbaeume zurück, in denen die Kante enthalten ist.
# Sind beide Ecken im selben Baum, wird zweimal derselbe zurueckgegeben.
def inTeilbaeumen(kante, baeume):
    teilbaeume=[]
    for ecke in kante[:2]:
        for baum in baeume:
            if eckeEnthalten(ecke, baum):
                teilbaeume+=[baum]
    return teilbaeume

# ----- verbinde -----
# verbindet die beiden Teilbäume durch die Kante.
def verbinde(kante, zweiTeilbaeume, alleBaeume):
    alleBaeume.remove(zweiTeilbaeume[0])
    alleBaeume.remove(zweiTeilbaeume[1])
    neuerTeilbaum=[kante]+zweiTeilbaeume[0]+zweiTeilbaeume[1]
    return alleBaeume+[neuerTeilbaum]

# ----- hinzufuegen -----
# fügt die neue Kante ihrem Teilbaum hinzu
def hinzufuegen(kante, teilbaum, baeume):
    baeume.remove(teilbaum)
    teilbaum+=[kante]
    return baeume+[teilbaum]

# ----- kruskal -----
# Aufrufhülle für die Rekursion
def kruskal(graph):
    graph=ordneKanten(graph)
```

```
    baeume=[]
    enthalten=[]
    for kante in graph:
        enthalten=inTeilbaeumen(kante, baeume)
        if len(enthalten)==0:
            # neuer Teilbaum
            baeume+=[[kante]]
        elif len(enthalten)==1:
            baeume=hinzufuegen(kante, enthalten[0], baeume)
        elif enthalten[0]!=enthalten[1]:
            baeume=verbinde(kante, enthalten, baeume)
    return baeume[0]


### ===== Testaufruf =====
print kruskal(mustergraph)
```

Der Algorithmus von Prim

- Beginne mit einem Baum, der allein aus einer Ecke besteht.
- Ordne alle Kanten von dieser Ecke aus nach ihren Kosten. (Prioritätswarteschlange 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.

Auch hier kann man sehr leicht zeigen, dass der hier beschriebene Algorithmus eine Lösung findet, wenn es überhaupt Lösungen geben kann. In jedem Fall sollte man sich hier ebenso wie beim Algorithmus von Kruskal einmal den Aufbau des Baumes für einen Beispiel-Graphen ansehen. Beim Algorithmus von Prim sind die beim Aufbau des Baumes entstehenden Teilgraphen stets zusammenhängend. Wird eine Kante hinzugefügt, dann hat sie stets eine Ecke mit dem derzeitigen Teilgraphen gemeinsam, setzt also einen Ast fort oder bildet einen neuen.
Bemerkenswert für unsere Betrachtung verschiedener Suchverfahren ist die Feststellung:

Die Datenstruktur ändert sich!

Mit dem völlig neuen Ansatz zur Lösung des Problems benötigt man auch eine völlig neue Datenstruktur: Es wird eine Datenstruktur aufgebaut, in der die Elemente nicht wie bei stack (->Tiefensuche) und queue (Warteschlange ; ->Breitensuche) nach dem Zeitpunkt ihres Einfügens in die Datenstruktur geordnet werden. Statt dessen werden sie nach dem ihnen zugehörenden Wert geordnet eingefügt (oder es wird zumindest die Zugriffsfunktion so realisiert, dass der Zugriff so funktioniert, als wäre das so). Die angemessene Datenstruktur zur Lösung des Problems ist die ***Prioritäts - Warteschlange***.
Das Programm ist zwar wegen der Behandlung der Prioritätswarteschlange in dem Abschnitt etwas umfangreicher als das zum Algorithmus von Kruskal, muss aber nicht die verschiedenen Fälle für die Teilbäume realisieren und ist daher insgesamt einfacher.

Weitere Untersuchungen mit anderen Problemen

Will man die Probleme vollständiger Suchverfahren deutlich machen, muss man zu Problemen wechseln, bei denen der Suchraum einen exponentiellen Anstieg abhängig

von der Suchtiefe aufweist. Dafür bietet sich beispielsweise die Verallgemeinerung des Problems der acht Damen an. An diesem Problem kann man auch zeigen, dass eine Verbesserung der Tiefensuche, bei der man möglichst früh prüft, ob der Pfad überhaupt noch zu einer Lösung führen kann oder ob schon einschränkende Bedingungen verletzt wurden, zwar die Laufzeit verbessern kann, aber nicht prinzipiell das Problem des exponentiellen Anstiegs des Aufwands verhindert.
Ähnliche Untersuchungen bieten sich beim Travelling-Salesman-Problem an, das zusätzlich ein Fall der np-vollständigen Probleme ist, von denen man kein Verfahren zur Bestimmung einer optimalen Lösung kennt und von denen die Informatiker überzeugt sind, dass es solche Lösungen auch nicht gibt. Hier bietet sich dann die Anwendung intelligenter Suchverfahren wie Genetischer Algorithmen an, die zumindest akzeptable suboptimale Lösungen liefern. Eine interessante Alternative ist auch die Betrachtung des A*-Verfahrens.