10.4 KÜRZESTER WEG, 3 KRÜGE

EINFÜHRUNG

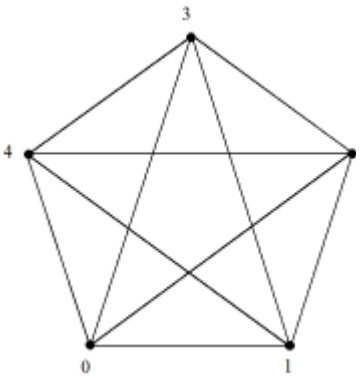
Viele wichtige algorithmische Lösungsverfahren haben ihren Ursprung in der Praxis des täglichen Leben so auch das Backtracking. Wie du bereits erfahren hast, wählt der Computer einen Spielzug aus de Möglichkeiten der erlaubten Züge und verfolgt ihn konsequent weiter. Gerät er in eine Sackgasse, s nimmt er die vorhergehenden Züge systematisch zurück. Diese Lösungsstrategie heisst im täglichen Lebe auch ***Versuch und Irrtum*** (trial and error). Es ist bekannt, dass sie nicht immer optimal ist. Besser wäre e den nächsten Zug möglichst günstig im Bezug auf die Zielsetzung zu wählen [mehr...].

PROGRAMMIERKONZEPTE: *Versuch und Irrtum, Graph mit Zyklen, Backtracking*

GRAPH MIT ZYKLEN

Beim Durchlaufen eines Graphen kann es sein, dass du nach einigen Schritten wieder im gleichen Zustan ankommst. Denke dabei zum Beispiel an das U-Bahn-Netz einer grossen Stadt, wo die Stationen ein verflochtene Struktur aufweisen. Willst du in dieser Stadt von A nach B fahren, so gibt es mehrer Möglichkeiten, und du kannst leicht auch in Zyklen herumfahren. Als Vorbereitung auf solch Navigationsaufgaben betrachtest du einen Graphen mit 5 Knoten, bei dem jeder Knoten mit jedem andere verbunden ist.

Die Knoten werden mit einer Knotennummer 0..4 identifiziert. Wie du bereits gesehen hast, findet der einfache Backtracking-Algorithmus den Weg von einem bestimmten Knoten zu einem anderen unter Umständen nicht, weil er in einem Zyklus hängen bleibt. Du benötigst aber nur eine kleine Ergänzung, um dies zu vermeiden: Bevor du den rekursiven Aufruf von *search()* machst, prüfst du in der Liste *visited*, ob der betreffende Nachbarknoten bereits besucht war. Wenn ja, dann überspringst du diesen Nachbarn. Im Programm werden nun alle 16 Wege zwischen den Knoten ausgeschrieben.




```
def getNeighbours(node):
    return range(0, node) + range(node + 1, 5)

def search(node):
    global nbSolution
    visited.append(node)  # node marked as visited

    # Check for solution
    if node == targetNode:
        nbSolution += 1
        print nbSolution, ". Route:", visited
        # don't stop to get all solutions

    for neighbour in getNeighbours(node):
        if neighbour not in visited: # Check if already visited
            search(neighbour) # recursive call
    visited.pop()

startNode = 0
targetNode = 4
```

```
nbSolution = 0
visited = []
search(startNode)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Durch Prüfen, ob ein Nachbar bereits besucht wurde, kann man das Backtracking auch auf Graphen m Zyklen anwenden. Vergisst du diese Prüfung, so wird sich dein Programm "aufhängen", was zu einer bösen Laufzeitfehler führt, da der Funktionsaufrufspeicher überläuft.

KÜRZESTER WEG, NAVIGATIONSSYSTEME

Das Auffinden eines Weges von einem Startort A zu einem Zielort B ist im täglichen Leben omnipräsent. D oft viele Wege von A nach B (nicht nur nach Rom) führen, besteht meist noch die zusätzliche Aufgabe, de bezüglich eines bestimmten Kriteriums (Weglänge, Fahrzeit, Strassenqualität, Sehenswürdigkeiten, Koste usw.) optimalen Weg zu finden [mehr...].

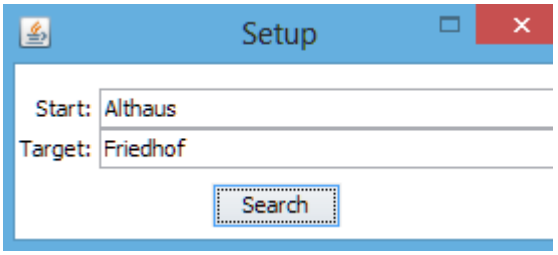
In deinem Programm geht es nur um das Grundsätzliche. Darum wählst du ein Ortsnetz mit nur 6 Orte die du als U-Bahn-Stationen in einer fiktiven Stadt auffassen kannst. Die Knoten des Graphen werden m dem Namen der Stationen identifiziert. (Die Anfangsbuchstaben sind A, B, C, D, E, F, die Stationen könnte auch mit diesen Buchstaben oder mit Knotennummern identifiziert werden.) Die Angabe de Nachbarstationen ist eine Zuordnung eines Stationsnamens zu einer Liste von Namen, wozu sich ei Dictionary hervorragend eignet, das als Schlüssel (key) den Stationsnamen und als Wert (value) die List der Nachbarstationen hat. Statt einer Funktion *getNeighbours()* wird direkt das Dictionary *neighbour* verwendet.

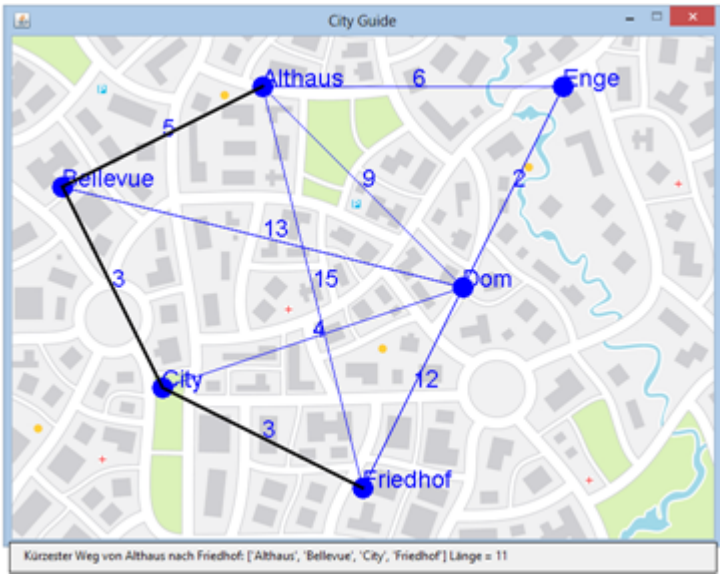
Analog werden die Distanzen zwischen den Stationen ebenfalls in einem Dictionary *distance* abgespeichert, das als Schlüssel die zwei verbundenen Stationen und als Wert die Distanz hat.

Um dies Stationen in einer GameGrid einzutragen, gibt es zudem noch ein Dictionary *locations* mit de Zellenkoordinaten (Locations) der Stationen.

Der zentrale Teil deines Programms ist eine genaue Wiederholung des oben verwendeten Backtracking Algorithmus. Darüber hinaus benötigst du einige Hilfsfunktionen zur grafischen Darstellung.

Für die Benutzereingabe verwendest du einen Eingabedialog und Ausgaben schreibst du in eine Statusbar. Überdies zeichnest du den optimalen Weg im Stationengraphen ein.






```
from gamegrid import *

neighbours = {
    'Althaus':['Bellevue', 'Dom', 'Enge', 'Friedhof'],
    'Bellevue':['Althaus', 'City', 'Dom'],
    'City':['Bellevue', 'Dom', 'Friedhof'],
    'Dom':['Althaus', 'Bellevue', 'City', 'Enge', 'Friedhof'],
    'Enge':['Althaus', 'Dom'],
    'Friedhof':['Althaus', 'City', 'Dom']}

distances = {
   ('Althaus', 'Bellevue'):5, ('Althaus', 'Dom'):9,
   ('Althaus', 'Enge'):6, ('Althaus', 'Friedhof'):15,
   ('Bellevue', 'City'):3, ('Bellevue', 'Dom'):13,
   ('City', 'Dom'):4, ('City', 'Friedhof'):3,
   ('Dom', 'Enge'):2, ('Dom', 'Friedhof'):12}

locations = {
    'Althaus':Location(2, 0),
    'Bellevue':Location(0, 1),
    'City':Location(1, 3),
    'Dom':Location(4, 2),
    'Enge':Location(5, 0),
    'Friedhof':Location(3, 4)}

def getNeighbourDistance(station1, station2):
    if station1 < station2:
        return distances[(station1, station2)]
    return distances[(station2, station1)]

def totalDistance(li):
    count = 0
    for i in range(len(li) - 1):
        count += getNeighbourDistance(li[i], li[i + 1])
    return count

def drawGraph():
    getBg().clear()
    getBg().setPaintColor(Color.blue)
    for station in locations:
        location = locations[station]
        getBg().fillCircle(toPoint(location), 10)
        startPoint = toPoint(location)
```

```
        getBg().drawText(station, startPoint)
        for s in neighbours[station]:
            drawConnection(station, s)
            if s < station:
                distance = distances[(s, station)]
            else:
                distance = distances[(station, s)]
            endPoint = toPoint(locations[s])
            getBg().drawText(str(distance),
                   getDividingPoint(startPoint, endPoint, 0.5))
    refresh()

def drawConnection(startStation, endStation):
    startPoint = toPoint(locations[startStation])
    endPoint = toPoint(locations[endStation])
    getBg().drawLine(startPoint, endPoint)

def search(station):
    global trackToTarget, trackLength
    visited.append(station)  # station marked as visited

    # Check for solution
    if station == targetStation:
        currentDistance = totalDistance(visited)
        if currentDistance < trackLength:
            trackLength = currentDistance
            trackToTarget = visited[:]

    for s in neighbours[station]:
        if s not in visited:  # if all are visited, recursion returns
            search(s) # recursive call
    visited.pop() # station may be visited by another path

def init():
    global visited, trackToTarget, trackLength
    visited = []
    trackToTarget = []
    trackLength = 1000
    drawGraph()

makeGameGrid(7, 5, 100, None, "sprites/city.png", False)
setTitle("City Guide")
addStatusBar(30)
show()
init()
startStation = ""
while not startStation in locations:
    startStation = inputString("Start station")
targetStation = ""
while not targetStation in locations:
    targetStation = inputString("Target station")
search(startStation)
setStatusText("Shortest way from " + startStation + " to " + targetStation
    + ": " + str(trackToTarget) + " Length = " + str(trackLength))
for i in range(len(trackToTarget) - 1):
    s1 = trackToTarget[i]
    s2 = trackToTarget[i + 1]
    getBg().setPaintColor(Color.black)
    getBg().setLineWidth(3)
    drawConnection(s1, s2)
refresh()
```

MEMO

Die Suche nach dem kürzesten Weg in einem Graphen gehört zu den Grundaufgaben der Informatik. Di hier gezeigte Lösung mit Backtracking führt zwar zum Ziel, ist aber sehr rechenintensiv. Es gibt viel besser Lösungsalgorithmen, die auch davon Gebrauch machen, dass man nicht alle Wege systematisch absuche muss. (Der berühmteste heisst Dijkstra-Algorithmus.) Es ist beispielsweise wenig wahrscheinlich, dass de kürzeste Weg von einem nördlich gelegenen Startort zu einem südlich gelegenen Zielort über eine we entfernte Station nördlich des Startorts führt.

DREIKRÜGEPROBLEM

Seit vielen Jahrhunderten findet man in Kinderbüchern und Zeitschriften Denksportaufgaben, bei dene eine vorgegebene Menge durch Abmessen (Umgiessen, Wägen, usw.) in bestimmte Teilmengen aufgete werden soll. Das seit dem 17. Jahrhundert bekannte Dreikrügeproblem wird dem französische Mathematiker Bachet de Méziriac zugeschrieben und lautet wie folgt:

Zwei Freunde haben beschlossen, sich durch Umgiessen 8 Liter Wein zu teilen, der sich in einem 8-Liter-Krug befindet. Sie besitzen dazu neben dem 8-Liter-Krug noch einen 5-Liter und einen 3-Liter-Krug. Die Krüge habe keine Inhaltsmarkierung. Wie müssen sie vorgehen und wie viele Umgiessvorgänge sind im Minimum nötig?

Gemäss der Aufgabenstellung geht es also nicht nur darum, eine Lösung zu finden, was du wahrscheinlic auch mit ein bisschen Gedankenspielerei zu Stande bringst, sondern alle Lösungen zu suchen, um darau die kürzeste zu bestimmen. Ohne Computer ist dies sehr anstrengend. Man nennt die Suche nach alle Lösungen auch **Erschöpfende Suche**.

Wieder gehst du gemäss der vorher erprobtem Lösungsstrategie mit Backtracking vor. Zuerst erfindest d eine geeignete Datenstruktur für die Spielzustände. Du verwendest dazu eine Liste mit einem Zahlentripe das den aktuellen Füllzustand der drei Krüge beschreibt. [1, 4, 3] soll heissen, dass der 8-Liter-Kru gegenwärtig 1 Liter, der 5-Liter-Krug 4 Liter und der 3-Liter-Krug 3 Liter enthält.

Du kannst die Spielzustände wiederum als Knoten in einem Graphen modellieren und das Umgiessen a Übergang von einem Knoten zu einem seiner Nachbarknoten auffassen. Es ist hier, wie in vielen andere Beispielen, nicht sinnvoll, den ganzen Spielbaum zu Beginn aufzubauen. Vielmehr bestimmst du di Nachbarknoten eines Knotens *node* in der Funktion *getNeighbours(node)* erst dann, wenn du sie im Lauf des Spiels tatsächlich benötigst. Dabei gehst du von folgender Überlegung aus:

Unabhängig davon, welche Menge sich in den Krügen befindet, gibt es für das Umgiessen grundsätzlich Möglichkeiten: Man nimmt einen der Krüge und giesst seinen ganzen Inhalt oder soviel wie Pla vorhanden ist in einen der zwei anderen Krüge. In *getNeighbours()* sammelst du daher in der List *neighbours* die Nachbarknoten für diese 6 Fälle. Die Funktion *transfer(state, source, target)* hilft dir dab herauszufinden, welches der Nachbarzustand zu einem bestimmten Zustand *state* und gegebene Krugnummern *source* und *target* beim Umgiessen von *source* nach *target* ist. Dabei werden d Kruggrössen (maximaler Inhalt) und die darin bereits enthaltenen Mengen berücksichtigt.
In der rekursiven Funktion *search()* verwendest du wieder den Backtracking-Algorithmus, so wie er d bereits bekannt ist.

```
def transfer(state, source, target):
    # Assumption: source, target 0..2, source != target
    s = state[:] # clone
    if s[source] == 0 or s[target] == capacity[target]:
        return s  # source empty or target full
    free = capacity[target] - s[target]
```

```
    if s[source] <= free: # source has enough space in target
        s[target] += s[source]
        s[source] = 0
    else:  # target is filled-up
        s[target] = capacity[target]
        s[source] -= free
    return s

def getNeighbours(node):
# returns list of neighbours
    neighbours = []
    t = transfer(node, 0, 1) # from 0 to 1
    if t not in neighbours:
       neighbours.append(t)
    t = transfer(node, 0, 2) # from 0 to 2
    if t not in neighbours:
        neighbours.append(t)
    t = transfer(node, 1, 0) # from 1 to 0
    if  t not in neighbours:
        neighbours.append(t)
    t = transfer(node, 1, 2) # from 1 to 2
    if t not in neighbours:
        neighbours.append(t)
    t = transfer(node, 2, 0) # from 2 to 0
    if t not in neighbours:
        neighbours.append(t)
    t = transfer(node, 2, 1) # from 2 to 1
    if t not in neighbours:
        neighbours.append(t)
    return neighbours

def search(node):
    global nbSolution
    visited.append(node)

    # Check for solution
    if node == targetNode:
        nbSolution += 1
        print nbSolution, ". Route:", visited, ". Length:", len(visited)

    for s in getNeighbours(node):
        if s not in visited:
            search(s)
    visited.pop()

capacity = [8, 5, 3]
startNode = [8, 0, 0]
targetNode = [4, 4, 0]
nbSolution = 0
visited = []
search(startNode)
print "Done. Find the best  solution!"
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die Lösungen werden in das Ausgabefenster ausgeschrieben. Damit du unvoreingenommen an da Problem heran gehst und vielleicht versuchst, zuerst eine eigene Lösung mit Papier und Bleistift zu finde werden sie hier nicht gezeigt. Immerhin sei verraten, dass es 16 Lösungen gibt und für den l längsten 1 Umgiessvorgänge nötig sind.

AUFGABEN

1. Vereinfache das Navigationsprogramm so, dass die Knoten mit Zahlen 0, 1, 2, 3, 4, 5 identifizie werden und *neighbours* eine Liste mit Teillisten ist.

2. Mit einem 3-Liter und einem 5-Liter-Krug, sollst du genau 4 Liter Wasser aus einem See schöpfe Beschreibe, wie du vorgehen würdest und gib den kürzesten Umgiessvorgang an. Beachte, dass d das Wasser auch wieder in den See zurückgiessen kannst.

3. Erfinde ein lösbares Umgiessproblem und stelle es als Denksportaufgabe für andere Personen i deiner Umgebung.

ZUSATZSTOFF

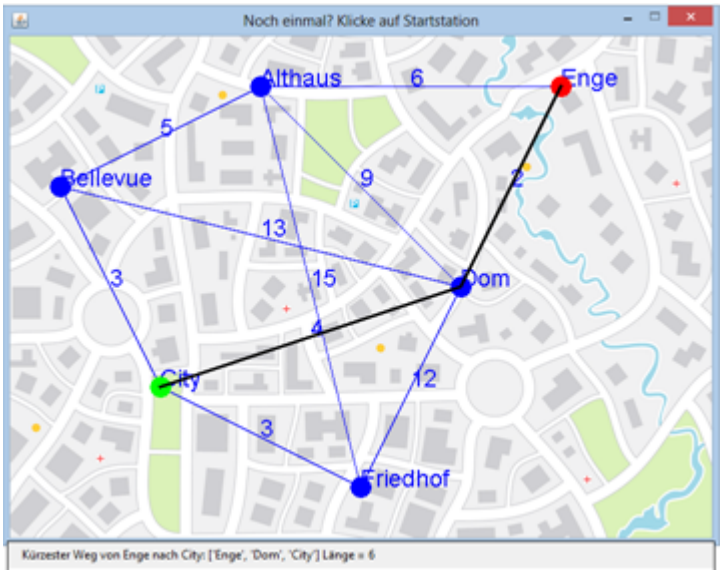
CITY-NAVIGATION MIT MAUSUNTERSTÜTZUNG

In einem professionellen Programm spielt die Benutzeroberfläche eine zentrale Rolle. Dabei muss sich de Programmierer weniger von programmtechnischen Überlegungen leiten lassen, sondern sich vielmehr i die Situation des unvoreingenommenen Anwenders versetzen, der mit möglichst wenig Aufwand und m natürlicher menschlicher Logik das Programm verwendet. Man spricht dabei auch vom **Benutzerinterfac** das heutzutage meist eine grafische Benutzeroberfläche mit Mausbedienung umfasst. Der Aufwand für di Entwicklung des Benutzerinterfaces kann ein beträchtlicher Teil des Gesamtaufwands eine Informatikprojekts ausmachen.

Für Navigationssysteme werden immer mehr berührungsempfindliche Bildschirme (touch screens verwendet, die sich aber bezüglich der Programmlogik wenig von einer Mausbedienung unterscheide Daher wirst du hier die City-Navigation auf Mausbedienung umstellen, so dass der Benutzer Start- un Zielort mit einem Mausklick auswählen kann. Vom Programm abgegebene Informationen werden einersei in die Titelzeile und andererseits in die Statusbar geschrieben.

Der Mausklick löst einen Event aus, der im Callback *pressEvent()* abgearbeitet wird. Diese registrierst du i *makeGameGrid()* mit dem benannten Parameter *mousePressed*. Dabei musst du berücksichtigen, das sich das Programm in zwei unterschiedlichen Zuständen befinden kann, je nachdem ob der Benutzer a nächste Aktion die Startstation anklickt oder ob er dies bereits getan hat und als nächstes die Zielstatio wählen muss. Für diese Zustandsumschaltung genügt eine boolesche Variable (ein Flag) *isStart*, die dan *True* ist, wenn als nächstes die Startstation zu wählen ist.

Das Programm soll so aufgebaut sein, dass der Benutzer die Wegsuche mehrmals durchführen kann, ohn dass er das Programm neu starten muss. Das Programm muss daher programmintern wieder in eine wohldefinierten Anfangszustand zurückgesetzt werden. Man spricht von der **Initialisierung**, die am beste mit einer Funktion *init()* durchgeführt wird. Da bei Programmstart gewisse Initialisierungen automatisc durchgeführt werden, ist es keineswegs trivial, das laufende Programm durch eine eigene Funktion imme wieder in einen wohldefinierten Anfangszustand zurückzusetzen. **Initialisierungsfehler** sind darum we verbreitete, gefährliche und schwierig aufzufindende Programmierfehler, da sich oft das Programm bei de Erprobung richtig und erst später im Einsatz falsch verhält.




```
from gamegrid import *

neighbours = {
    'Althaus':['Bellevue', 'Dom', 'Enge', 'Friedhof'],
    'Bellevue':['Althaus', 'City', 'Dom'],
    'City':['Bellevue', 'Dom', 'Friedhof'],
    'Dom':['Althaus', 'Bellevue', 'City', 'Enge', 'Friedhof'],
    'Enge':['Althaus', 'Dom'],
    'Friedhof':['Althaus', 'City', 'Dom']}

distances = {
   ('Althaus', 'Bellevue'):5, ('Althaus', 'Dom'):9,
   ('Althaus', 'Enge'):6, ('Althaus', 'Friedhof'):15,
   ('Bellevue', 'City'):3, ('Bellevue', 'Dom'):13,
   ('City', 'Dom'):4, ('City', 'Friedhof'):3,
   ('Dom', 'Enge'):2, ('Dom', 'Friedhof'):12}

locations = {
    'Althaus':Location(2, 0),
    'Bellevue':Location(0, 1),
    'City':Location(1, 3),
    'Dom':Location(4, 2),
    'Enge':Location(5, 0),
    'Friedhof':Location(3, 4)}

def getNeighbourDistance(station1, station2):
    if station1 < station2:
        return distances[(station1, station2)]
    return distances[(station2, station1)]

def totalDistance(li):
    count = 0
    for i in range(len(li) - 1):
        count += getNeighbourDistance(li[i], li[i + 1])
    return count

def drawGraph():
    getBg().clear()
    getBg().setPaintColor(Color.blue)
    for station in locations:
        location = locations[station]
        getBg().fillCircle(toPoint(location), 10)
        startPoint = toPoint(location)
```

```
        getBg().drawText(station, startPoint)
        for s in neighbours[station]:
            drawConnection(station, s)
            if s < station:
                distance = distances[(s, station)]
            else:
                distance = distances[(station, s)]
            endPoint = toPoint(locations[s])
            getBg().drawText(str(distance),
                 getDividingPoint(startPoint, endPoint, 0.5))
    refresh()

def drawConnection(startStation, endStation):
    startPoint = toPoint(locations[startStation])
    endPoint = toPoint(locations[endStation])
    getBg().drawLine(startPoint, endPoint)

def search(station):
    global trackToTarget, trackLength
    visited.append(station)  # station marked as visited

    # Check for solution
    if station == targetStation:
        currentDistance = totalDistance(visited)
        if currentDistance < trackLength:
            trackLength = currentDistance
            trackToTarget = visited[:]

    for s in neighbours[station]:
        if s not in visited:  # if all are visited, recursion returns
            search(s) # recursive call
    visited.pop() # station may be visited by another path

def getStation(location):
    for station in locations:
        if locations[station] == location:
            return station
    return None # station not found

def init():
    global visited, trackToTarget, trackLength
    visited = []
    trackToTarget = []
    trackLength = 1000
    drawGraph()

def pressEvent(e):
    global isStart, startStation, targetStation
    mouseLoc = toLocationInGrid(e.getX(), e.getY())
    mouseStation = getStation(mouseLoc)
    if mouseStation == None:
        return
    if isStart:
        isStart = False
        init()
        setTitle("Click on destination  station")
        startStation = mouseStation
        getBg().setPaintColor(Color.red)
        getBg().fillCircle(toPoint(mouseLoc), 10)
    else:
        isStart = True
        setTitle("Once again? Click on  starting station.")
        targetStation = mouseStation
```

```
        getBg().setPaintColor(Color.green)
        getBg().fillCircle(toPoint(mouseLoc), 10)
        search(startStation)
        setStatusText("Shortest route from " + startStation + " to "
            + targetStation + ": " + str(trackToTarget) + " Length = "
            + str(trackLength))
        for i in range(len(trackToTarget) - 1):
            s1 = trackToTarget[i]
            s2 = trackToTarget[i + 1]
            getBg().setPaintColor(Color.black)
            getBg().setLineWidth(3)
            drawConnection(s1, s2)
            getBg().setLineWidth(1)
    refresh()

isStart = True
makeGameGrid(7, 5, 100, None, "sprites/city.png", False,
             mousePressed = pressEvent)
setTitle("City Guide. Click on starting  station.")
addStatusBar(30)
show()
init()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Der algorithmische Teil mit dem Backtracking bleibt praktisch unverändert. Das Benutzerinterface mit de Maussteuerung ist trotz guter Unterstützung durch Callbacks ziemlich aufwendig.

Die Verwendung von **global** führt in Python leicht zu Initialisierungsfehlern, da globale Variablen i Funktionen erzeugt werden können und man später vergisst, ihren Wert zurückzusetzen.

ROUTING-ALGORITHMUS VON DIJKSTRA

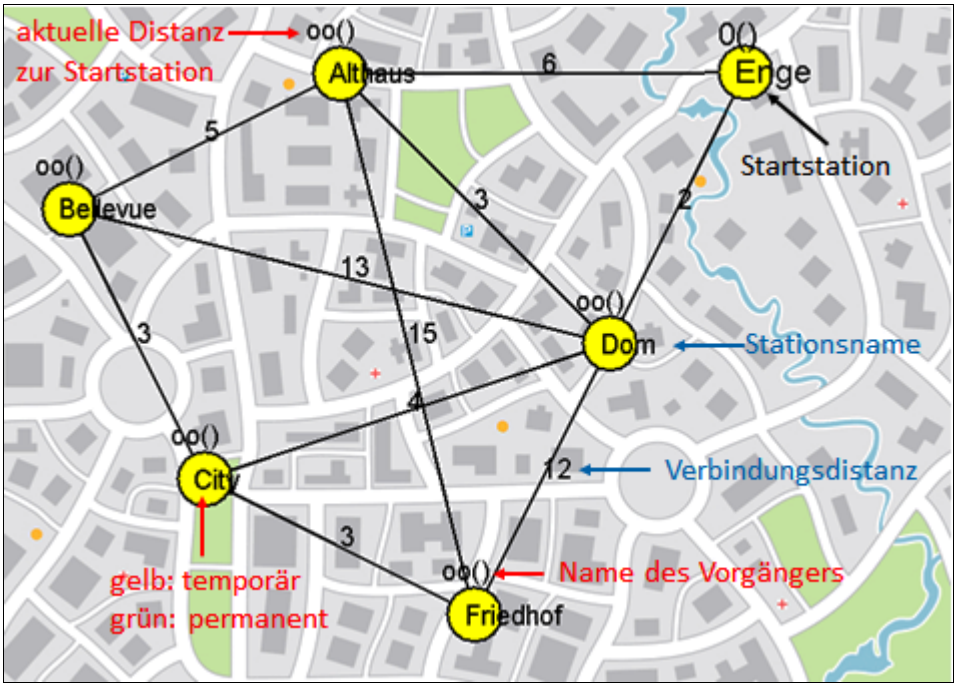
Bereits 1959 veröffentlichte der berühmte niederländische Informatiker Edsger W. Disjkstra in einer dreiseitigen Artikel einen Algorithmus, um in einem Graphen sehr effizient und elegant den kürzesten We von einem Anfangsknoten A zu einem Endknoten B zu finden. Die Strategie ist es, nicht einfach all möglichen Wege von A zu B zu suchen und davon den kürzesten zu bestimmen, sondern gezielt z versuchen, von A nach B zu gelangen, indem man Wege absucht, die von einem bestimmten Knoten z demjenigen weiterführen, der am nächsten liegt. (Einen solchen Algorithmus nennt man auch "gierig" (eng **greedy**), da man bereits während der Ausführung noch einer optimalen Lösung sucht.)

Am einfachsten lässt sich der Algorithmus am bereits oben verwendeten Beispiel der Bahnstatione erläutern. Die Stationen haben wie oben einen Namen und eine Position und das Bahnnetz wird durch di direkten Verbindungen und ihre Distanzen beschrieben, d.h. jede Station hat eine Anzahl vo Nachbarstationen mit bekannten Verbindungsdistanzen. Diese Angaben verändern sich während de Programmablaufs nicht.

Für den Routing-Algorithmus von Dijkstra benötigt jede Station drei Statusinformationen, die während de Programmablaufs angepasst werden.

a. Die Station kann "temporär" oder "permanent" sein (ein String "ttemporary" bzw. "permanent"
b. Die Station hat eine aktuelle Distanz zur Startstation (eine Float Zahl >= 0)
c. Die Station kennt den Namen der Vorgängerstation (ein String mit einem Namen oder ein leere String, wenn es noch keinen Vorgänger gibt)

Zu Beginn sind alle Stationen temporär, die aktuelle Distanz wir auf einen sehr grossen Wert gesetzt (z.B 1000, eingetragen als oo (unendlich)) und der Name des Vorgängers ist leer. Dann fragt man nach der Startstation und setzt dessen Distanz auf 0.



Algorithmus von Dijkstra:

Solange nicht alle Stationen permanent sind, mache Folgendes:

- Suche unter allen temporären Stationen diejenige mit der kleinsten Distanz. Sie heisse S
- Mache S permanent
- Für alle temporären Nachbarstationen N von S mache Folgendes: Berechne die Länge des Wege zu N, der über S führt. Falls diese kleiner ist als N bereits eingetragen hat, so ersetze sie durc diesen neuen Wert und setze die Vorgängerstation von N auf S.

Nach Ende des Durchlaufs haben alle Stationen einen Vorgängerknoten (ausser eine Station ist gar nicl von der Startstation aus erreichbar). Den kürzesten Weg zu einer beliebig wählbaren Endstation find man, indem man rückwärts von der Endstation die Vorgängerstationen durchläuft.

In der Implementierung mit Python werden Nachbarn, Verbindungsdistanzen und Locations in Dictionarie *neighbours, distances, locations* gespeichert. Für den sich ändernden Stationszustand wird ebenfalls ei Dictionary *stations* mit dem Stationsnamen als Key definiert. Der Value ist eine Liste mit den Daten gemäs a,b,c, auf die man mit den Indizes 0, 1, 2 zugreifen kann.

Zur Demonstration musst du nach Eingabe der Startstation eine beliebige Taste drücken, um de Algorithmus schrittweise auszuführen. Beobachte, wie die Werte der Stationen gemäss a, b, c laufen angepasst werden. Zuletzt wirst du nach einer Endstation gefragt und die kürzeste Route von d Startstation zur Endstation wird eingezeichnet und ausgeschrieben.

Du nimmst es in Kauf, dass das Programm durch die grafische Ausgabe etwas aufwendiger wird. Dafi könnte das Programm mit wenig Mehraufwand bereits als professionelle Anwendung verwendet werden.

```
from gpanel import *
from sys import exit

neighbours = {
    'Althaus':('Bellevue', 'Dom', 'Enge', 'Friedhof'),
    'Bellevue':('Althaus', 'City', 'Dom'),
    'City':('Bellevue', 'Dom', 'Friedhof'),
    'Dom':('Althaus', 'Bellevue', 'City', 'Enge', 'Friedhof'),
    'Enge':('Althaus', 'Dom'),
    'Friedhof':('Althaus', 'City', 'Dom')}

distances = {  # keys are ordered alphabetically
   ('Althaus', 'Bellevue'):5, ('Althaus', 'Dom'):3,
```

```
   ('Althaus', 'Enge'):6, ('Althaus', 'Friedhof'):15,
   ('Bellevue', 'City'):3, ('Bellevue', 'Dom'):13,
   ('City', 'Dom'):4, ('City', 'Friedhof'):3,
   ('Dom', 'Enge'):2, ('Dom', 'Friedhof'):12}

locations = {
    'Althaus':(5, 9),
    'Bellevue':(1, 7),
    'City':(3, 3),
    'Dom':(9, 5),
    'Enge':(11, 9),
    'Friedhof':(7, 1)}

stations = {
    # temporary or permanent,
    # distance to start station (1000 if infinite),
    # previous station (emtpy if not yet set)
    'Althaus':['temporary', 1000, ''],
    'Bellevue':['temporary', 1000, ''],
    'City':['temporary', 1000, ''],
    'Dom':['temporary', 1000, ''],
    'Enge':['temporary', 1000, ''],
    'Friedhof':['temporary', 1000, '']}

def getNeighbourDistance(station1, station2):
    if station1 < station2:  # compare alphabetically
        return distances[(station1, station2)]
    return distances[(station2, station1)]

def drawGraph():
    image("sprites/city.png", 0, 0)
    lineWidth(2)
    setColor("black")
    for station in locations:
        move(locations[station])
        circle(0.4) # draw station as circle
        for neighbour in neighbours[station]:
            drawConnection(station, neighbour) # draw connecting lines

    # fill with color to indicate temporary or permanent
    for station in locations:
        if stations[station][0] == "temporary":
            setColor("yellow")
        elif stations[station][0] == "permanent":
            setColor("green")
        pt = locations[station]
        move(pt)
        fillCircle(0.38)

        # show station indentifier
        setColor("black")
        text(pt[0] - 0.15, pt[1] - 0.15, station)

        # show distance to start station and previous station
        dist = stations[station][1]
        dist = "oo" if dist == 1000 else str(dist)  # 1000->infinite sign
        text(pt[0] - 0.5, pt[1] + 0.5,
            dist + "(" + stations[station][2] + ")")
    repaint()

def drawConnection(start, end):
    ptStart = locations[start]
    ptEnd = locations[end]
```

```
    line(ptStart, ptEnd)
    distance = getNeighbourDistance(start, end)
    text(getDividingPoint(ptStart, ptEnd, 0.5), str(distance))

def getTemporaryNeighbours(station):
    li = []
    for n in neighbours[station]:
        if stations[n][0] == "temporary":
            li.append(n)
    return li

def getNearestTemporaryStation():
    minstation = None
    min = 1000
    for n in stations:
        if stations[n][0] == "temporary" and stations[n][1] <= min:
            min = stations[n][1]
            minstation = n
    return minstation

def updateStations(station):
    stations[station][0] = "permanent"  # make station permanent
    currentDist = stations[station][1]
    for n in getTemporaryNeighbours(station):
        newDist = currentDist + getNeighbourDistance(station, n)
        dist = stations[n][1]
        if newDist < dist:
            stations[n][1] = newDist
            stations[n][2]= station

makeGPanel(Size(700, 500))
enableRepaint(False)
window(0, 14, 0, 10)
title("Shortest Path (Dijkstra Algorithm)")
font(Font("Arial", Font.PLAIN, 20))
addStatusBar(30)
drawGraph()
startStation = ""
setStatusText("Select start station")
while not startStation in locations:
    startStation = inputString("Start station", False)
    if startStation == None:
        dispose()
        exit()

stations[startStation][1] = 0
drawGraph()
steps = 0
setStatusText("Press any key to step through algorithm...")

# Perform Dijkstra algorithm
while True:
   if getKeyWait():
        currentStation = getNearestTemporaryStation()
        if currentStation == None:
            break
        updateStations(currentStation)
        drawGraph()
        steps += 1
        setStatusText("Step #" + str(steps) + " - continue...")

setStatusText("Algorithm finished.")
targetStation = ""
```

```
while not targetStation in locations:
    targetStation = inputString("Target station", False)
    if targetStation == None:
        dispose()
        exit()

# Build track
station = targetStation
path = [station]
while station != startStation:
    station = stations[station][2]
    path.append(station)
track = path[::-1] # reverse
lineWidth(4)
setColor("red")
trackLength = 0
for i in range(len(path) - 1):
    n1 = track[i]
    n2 = track[i + 1]
    trackLength += getNeighbourDistance(n1, n2)
    line(locations[n1], locations[n2])
setStatusText("Shortest route from " + startStation
    + " to " + targetStation  + ": " + str(track)
    + " Length = " + str(trackLength))
repaint()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

AUFGABEN

1. Lass den Algorithmus ausführen, ohne auf Tasteneingaben zu warten.

2. Wähle andere Verbindungsdistanzen und führe den Algorithmus einige Male durch.

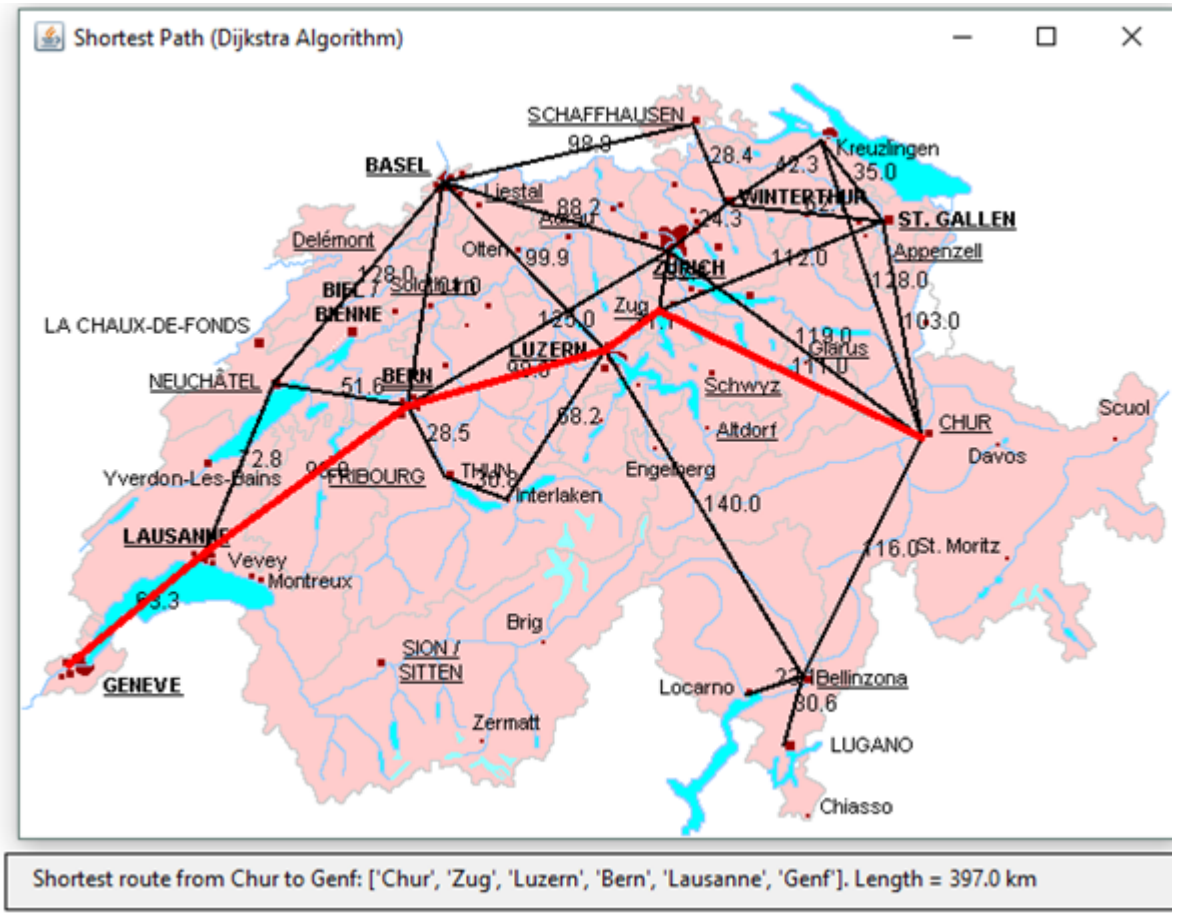
3. Wähle ein Stationsnetz mit zwei Teilnetzen, die nicht miteinander verbunden sind. Verbessere da Programm, dass es "No route from A to B" ausschreibt, falls A und B in verschiedenen Teilnetzen lieger

4. Nimm 10 - 20 grössere Städte in deinem Land, die über Bahnverbindungen miteinander verbunde sind. Erstelle ein Programm, das bei Wahl des Abfahrts- und Zielortes die kürzeste Route ausschreil (ohne Grafik).

5*. Als bereits professionelle Applikation gehst du vom Streckennetz der Schweizer Bahnen (SBB/CFF) m 18 Stationen aus. Der Anwender kann den Abfahrts- und Zielort angeben, und das Programm zeichne die kürzeste Route in einer Schweizer Karte ein. Zudem schreibt es die Route im Statusfeld aus. Geh von unten angegebenen Daten aus, die du noch beliebig erweitern kannst.

   Hinweis: Die Schweizer Karte *swissmap.gif* mit der Dimension (618, 412) befindet sich in der Sprite Distribution von TigerJython. Zur Darstellung wählst du ein GPanel mit dieser Grösse und wählst ein (y)-Koordinatensystem, das genau diesen Bereich überdeckt.

   ```
   makeGPanel(Size(612, 412))
   window(0, 618, 0, 412)
   image("sprites/swissmap.gif", 0, 0)
   ```



Du kannst in deinem Programm folgende Daten verwenden:

```
neighbours = {
    'Bern':['Basel', 'Lausanne', 'Luzern', 'Neuenburg', 'Thun', 'Zurich'],
    'Chur':['Bellinzona', 'Kreuzlingen', 'St.Gallen', 'Zug', 'Zurich'],
    'Winterthur':['Kreuzlingen', 'St.Gallen', 'Schaffhausen', 'Zurich'],
    'Lausanne':['Bern', 'Genf', 'Neuenburg'],
    'Luzern':['Basel', 'Bellinzona', 'Bern', 'Interlaken', 'Zug'],
    'Thun':['Bern', 'Interlaken'],
    'Zurich':['Basel', 'Bern', 'Chur', 'Winterthur', 'Zug'],
    'Zug':['Chur', 'Luzern', 'St.Gallen', 'Zurich'],
    'Schaffhausen':['Basel', 'Winterthur'],
    'Neuenburg':['Basel', 'Bern', 'Lausanne'],
    'St.Gallen':['Chur', 'Kreuzlingen', 'Winterthur', 'Zug'],
    'Bellinzona':['Chur', 'Locarno', 'Lugano', 'Luzern'],
    'Basel':['Bern', 'Luzern', 'Neuenburg', 'Schaffhausen', 'Zurich'],
    'Lugano':['Bellinzona'],
    'Locarno':['Bellinzona'],
    'Genf':['Lausanne'],
    'Interlaken':['Luzern', 'Thun'],
    'Kreuzlingen':['Chur', 'St.Gallen', 'Winterthur']}

distances = {  # keys are ordered alphabetically
    ('Bern', 'Lausanne'):93.0,('Bern', 'Luzern'):98.6,
    ('Bern', 'Neuenburg'):51.6, ('Bern', 'Thun'):28.5,
    ('Bern', 'Zurich'):125.0,('Chur', 'Kreuzlingen'):128.0,
    ('Chur', 'St.Gallen'):103.0, ('Chur', 'Zug'):111.0,
    ('Chur', 'Zurich'):119.0, ('Winterthur', 'Zurich'):24.3,
    ('Lausanne', 'Neuenburg'):72.8,('Luzern', 'Zug'):31.1,
    ('Zug', 'Zurich'):32.4, ('Schaffhausen', 'Winterthur'):28.4,
    ('St.Gallen', 'Winterthur'):62.1, ('St.Gallen', 'Zug'):112.0,
```

```
    ('Bellinzona', 'Chur'):116.0, ('Bellinzona', 'Locarno'):23.1,
    ('Bellinzona', 'Lugano'):30.6, ('Bellinzona', 'Luzern'):140.0,
    ('Basel', 'Bern'):101.0, ('Basel', 'Luzern'):99.9,
    ('Basel', 'Neuenburg'):128.0, ('Basel', 'Schaffhausen'):98.8,
    ('Basel', 'Zurich'):88.2, ('Genf', 'Lausanne'):63.3,
    ('Interlaken', 'Luzern'):68.2, ('Interlaken', 'Thun'):30.8,
    ('Kreuzlingen','St.Gallen'):35.0,('Kreuzlingen','Winterthur'):42.3}

locations = {
    'Bern':(207, 231),
    'Chur':(484, 214),
    'Winterthur':(379, 339),
    'Lausanne':(98, 151),
    'Luzern':(314, 261),
    'Thun':(227, 192),
    'Zurich':(348, 314),
    'Zug':(343, 282),
    'Schaffhausen':(361, 382),
    'Neuenburg':(135, 243),
    'Bellinzona':(420, 85),
    'Basel':(226, 351),
    'Lugano':(410, 48),
    'Locarno':(390, 75),
    'Genf':(25, 91),
    'St.Gallen':(463, 330),
    'Interlaken':(260, 180),
    'Kreuzlingen':(430, 373)}
```