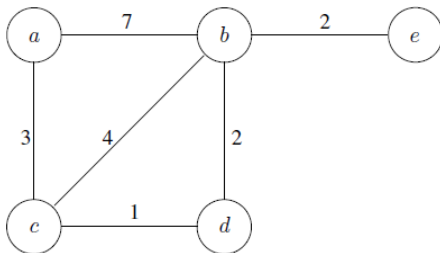
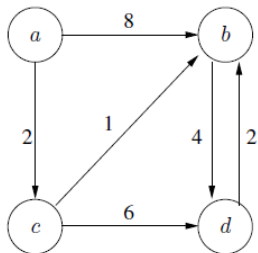


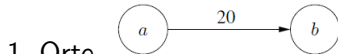
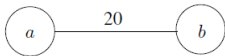
Informatik

Graphen

Ein *Graph* besteht aus Knoten und Kanten. Er kann gerichtet oder ungerichtet sein. Die Kanten können gewichtet sein (Kosten).



Mit Graphen können binäre Beziehungen zwischen Objekten modelliert werden. Die Objekte werden durch die Knoten, die Beziehungen durch die Kanten modelliert.



1. Orte

Entfernungen, Kosten, Dauer



2. Personen

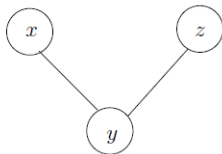
ist verheiratet mit



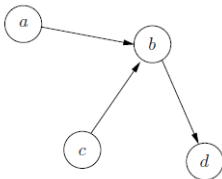
3. Ereignisse

a muss vor b geschehen

Begriffe



x ist zu y adjazent,
x und y sind Nachbarn,
x und z sind unabhängig,
der Grad von y ist 2

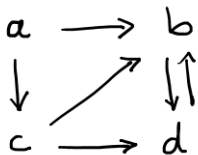


a ist Vorgänger von b,
b ist Nachfolger von a,
Eingangsgrad von b ist 2,
Ausgangsgrad von b ist 1

Ein *Weg* ist eine Folge von adjazenten Knoten.

Ein *Kreis* ist eine Weg, der zurück zum Startknoten führt.

Implementation eines ungewichteten Graphen durch eine *Adjazenzmatrix*



Wir ordnen jedem Knoten einen Index zu:

Index	0	1	2	3
Knoten	a	b	c	d

	0	1	2	3
0	0	1	1	0
1	0	0	0	1
2	0	1	0	1
3	0	1	0	0

$$G = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

gibt es Kante von a nach b?

```
>>> G[0][1] == 1
```

```
True
```

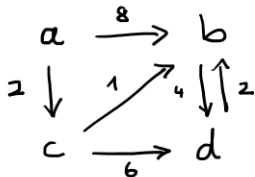
alle Nachbarn von a

```
>>> for k in [j for j in range(len(G)) if G[0][j] == 1]:  
    print(k)
```

```
1
```

```
2
```

Implementation eines gewichteten Graphen durch eine Adjazenzmatrix



Wir ordnen jedem Knoten einen Index zu:

Index	0	1	2	3
Knoten	a	b	c	d

Nicht vorhandene Kanten haben Kosten: unendlich

```
inf = float('inf')
G = [[0, 8, 2, inf],
      [inf, 0, inf, 4],
      [inf, 1, 0, 6],
      [inf, 2, inf, 0]]
```

Kosten von a nach b

```
>>> G[0][1]
```

8

alle Nachbarn von a

```
>>> for k in [j for j in range(len(G)) if j!=0 and G[0][j] < inf]:
      print(k)
```

1

2

Analyse Implementation durch Adjazenzmatrizen:

Platzbedarf = $O(|V|^2)$.

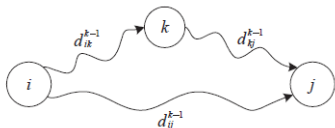
Direkter Zugriff auf Kante (i, j) in konstanter Zeit möglich.

Kein effizientes Verarbeiten der Nachbarn eines Knotens.

Sinnvoll bei dicht besetzten Graphen, d.h. Graphen mit quadratisch vielen Kanten.

Sinnvoll bei Algorithmen, die wahlfreien Zugriff auf eine Kante benötigen.

Der *Floyd-Warshall Algorithmus* löst das *all-pairs-shortest-path* Problem. Die Matrix D^k enthält die Kosten der kürzesten Wege zwischen zwei Knoten i und j , die als Zwischenknoten nur die Knoten $0, 1, \dots, k$ verwenden. Dann lässt sich $d_{i,j}^k$ errechnen durch das Minimum von $d_{i,j}^{k-1}$ und $d_{i,k}^{k-1} + d_{k,j}^{k-1}$.



Um die Kantenfolge zu rekonstruieren, wird gleichzeitig eine Folge von Matrizen P^k aufgebaut, die an Position $p_{i,j}^k$ den vorletzten Knoten auf dem kürzesten Weg von i nach j notiert, der nur über die Zwischenknoten $0, 1, \dots, k$ läuft.


```

def floyd (c):
    n = len(c)
    d = [[0]*n for j in range(n)]
    p = [[0]*n for j in range(n)]
    for i in range(n):
        for j in range(n):
            d[i][j] = c[i][j]
            p[i][j] = i

    for k in range(n):
        for i in range(n):
            for j in range(n):
                tmp = d[i][k] + d[k][j]
                if tmp < d[i][j]:
                    d[i][j] = tmp
                    p[i][j] = p[k][j]

    return d, p

def getPath(p, i, j):
    if i == j: return str(i)
    return getPath(p, i, p[i][j]) + ' - ' + str(j)

```