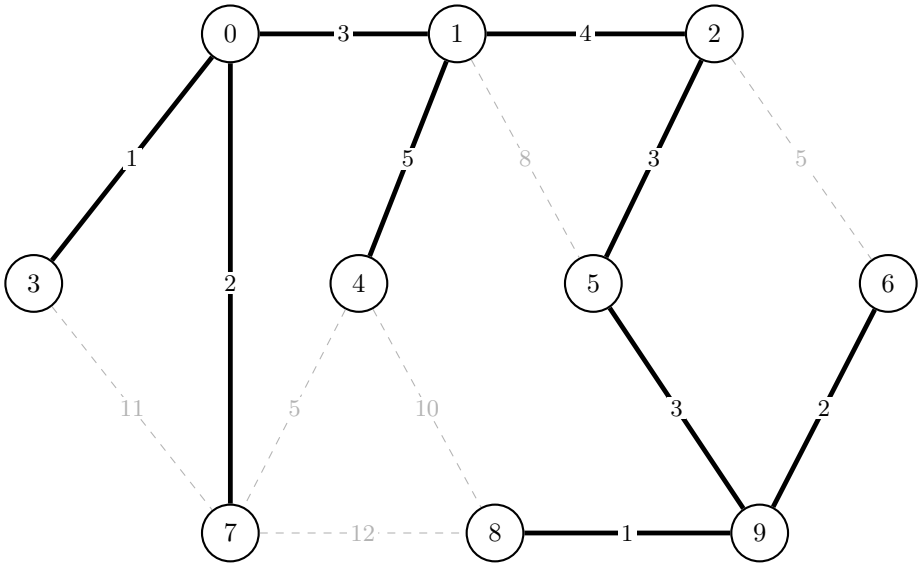
ARCHENHOLD

GYMNASIUM

Graphen

Begleitskript zum Informatikunterricht

Suchen · kürzeste Wege · Spannbäume · Zuordnungen · Reihenfolgen



Inhaltsverzeichnis

1 Der Begriff des Graphen **2**

1.1 Einführung 2

1.2 Erreichbarkeit: die Algorithmen von Warshall und Floyd 4

1.3 Breitensuche und Tiefensuche 7

1.4 Der Algorithmus von Dijkstra 9

1.5 Der Algorithmus von Kruskal 11

1.6 Der Algorithmus von Prim 13

2 Ausgewählte Probleme **16**

2.1 Das Heiratsproblem 16

2.2 Der Algorithmus von Gale und Shapley 16

3 Topologische Sortierung **18**

4 Überblick: Welches Verfahren wofür? **19**

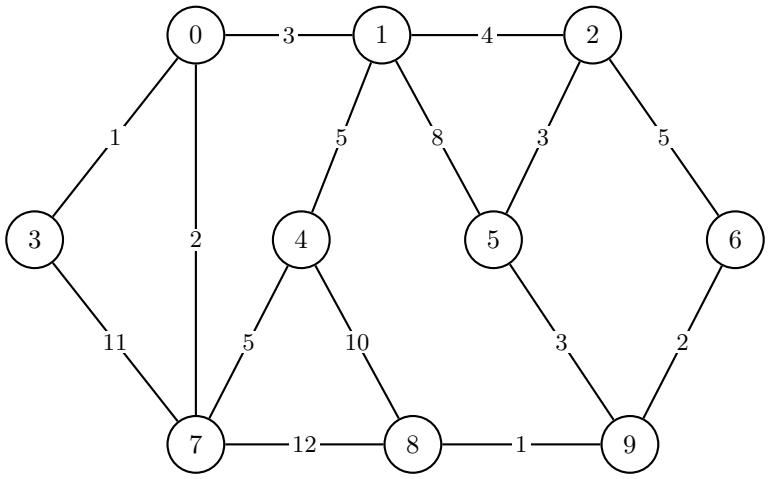
5 Übungen **21**

6 Lösungen **23**

So arbeiten Sie mit dem Skript: Jedes Verfahren wird erklärt, als Python-Quelltext gezeigt und an einem Beispiel Schritt für Schritt durchgerechnet. Die Kästen *„auf einen Blick“* fassen das Wichtigste zusammen. Abschnitt 5 enthält Übungen zum Selbstrechnen, Abschnitt 6 die vollständigen Lösungen – erst selbst rechnen, dann vergleichen. Alle Tabellen lassen sich auch mit der Graphen-App Schritt für Schritt nachvollziehen.

1 Der Begriff des Graphen

1.1 Einführung



In der Graphentheorie ist ein Graph eine Menge von Punkten (*Knoten*), die durch Linien (*Kanten*) miteinander verbunden sind. Man unterscheidet *gerichtete* Graphen, deren Kanten eine Orientierung haben (meist durch Pfeile dargestellt), und *ungerichtete* Graphen. Gelangt man auf einem Weg durch den Graphen wieder an den Ausgangspunkt, ist ein *Zyklus* entstanden; entsprechend unterscheidet man zyklische und azyklische Graphen.

Ein Graph $G = (V, E)$ besteht aus einer Menge V von Knoten (*vertices*) und einer Menge E von Kanten (*edges*). Jede Kante $e = \{v_1, v_2\}$ verbindet zwei Knoten. Ist e ein ungeordnetes Paar, ist der Graph ungerichtet (wie oben); ist $e = (v_1, v_2)$ ein geordnetes Paar, hat jede Kante einen Anfangs- und einen Endknoten. Trägt jede Kante eine Zahl (Länge, Kosten, Latenz), spricht man von einem *gewichteten* (bewerteten) Graphen.

Zwei Knoten heißen *benachbart* (adjazent), wenn sie durch eine Kante verbunden sind. Der *Grad* $d(v)$ eines Knotens v ist die Anzahl der Kanten, die an ihm enden. Daraus ergeben sich die beiden typischen Darstellungsformen *Adjazenzmatrix* und *Adjazenzliste*, hier für den Graphen oben (in der Matrix steht das Gewicht, 0 bedeutet „keine Kante“).

Adjazenzmatrix

x	0	1	2	3	4	5	6	7	8	9
0	0	3	0	1	0	0	0	2	0	0
1	3	0	4	0	5	8	0	0	0	0
2	0	4	0	0	0	3	5	0	0	0
3	1	0	0	0	0	0	0	11	0	0
4	0	5	0	0	0	0	0	5	10	0
5	0	8	3	0	0	0	0	0	0	3
6	0	0	5	0	0	0	0	0	0	2
7	2	0	0	11	5	0	0	0	12	0
8	0	0	0	0	10	0	0	12	0	1
9	0	0	0	0	0	3	2	0	1	0

Adjazenzliste

$0 \mapsto 1 \mapsto 3 \mapsto 7$
$1 \mapsto 0 \mapsto 2 \mapsto 4 \mapsto 5$
$2 \mapsto 1 \mapsto 5 \mapsto 6$
$3 \mapsto 0 \mapsto 7$
$4 \mapsto 1 \mapsto 7 \mapsto 8$
$5 \mapsto 1 \mapsto 2 \mapsto 9$
$6 \mapsto 2 \mapsto 9$
$7 \mapsto 0 \mapsto 3 \mapsto 4 \mapsto 8$
$8 \mapsto 4 \mapsto 7 \mapsto 9$
$9 \mapsto 5 \mapsto 6 \mapsto 8$

Graphen finden sich in fast allen Gebieten der Informatik: bei Wegeproblemen wie dem *kürzesten Weg* oder der günstigsten Vernetzung aller Knoten, bei Abhängigkeiten (Reihenfolgeplanung) – und jeder Baum ist ein besonderer Graph (zusammenhängend und ohne Zyklus).

Für die Implementierung bieten sich Python-Listen an. Die Matrix ist übersichtlich, die Liste spart bei wenigen Kanten Platz und liefert die Nachbarn eines Knotens direkt; viele Verfahren nutzen deshalb beide Strukturen und eine Umwandlung zwischen ihnen.

Quelltext 1: Graph mit Adjazenzmatrix und Adjazenzliste

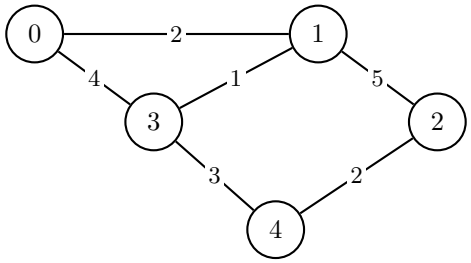
```
class Graph:
    def __init__(self, n):
        self.n = n                                  # Anzahl der Knoten
        self.matrix = []
        for i in range(n):
            self.matrix.append([0] * n)             # 0 = keine Kante
        self.ad_liste = []

    def kante_einfuegen(self, u, v, gewicht=1):     # ungerichtet
        self.matrix[u][v] = gewicht
        self.matrix[v][u] = gewicht

    def matrix2liste(self):
        self.ad_liste = []
        for i in range(self.n):
            nachbarn = []
            for j in range(self.n):
                if self.matrix[i][j] != 0:
                    nachbarn.append(j)
            self.ad_liste.append(nachbarn)          # Index = Knoten

    def zeige_liste(self):
        for i in range(self.n):
            print(i, "->", self.ad_liste[i])
```

1.2 Erreichbarkeit: die Algorithmen von Warshall und Floyd



Multipliziert man die (ungewichtete) Adjazenzmatrix A mit sich selbst, gibt der Eintrag $(A^2)_{ij}$ die Anzahl der Wege der *Länge 2* von i nach j an, allgemein $(A^m)_{ij}$ die Anzahl der Wege der Länge m. Beispiel: Es gibt eine direkte Kante von 0 nach 3; das Produkt zeigt zusätzlich, dass es genau einen Weg der Länge 2 gibt, nämlich über Knoten 1 (Zeile 0 und Spalte 3 fett):

x	0	1	2	3	4
0	0	**1**	0	**1**	0
1	1	0	1	**1**	0
2	0	1	0	0	1
3	1	1	0	0	1
4	0	0	1	1	0

$$(A^2)_{03} = \sum_{i=0}^{4} a_{0i}\, a_{i3} = a_{01}\, a_{13} = 1 \cdot 1 = 1$$

Auf dieser Idee bauen zwei Verfahren auf, die beide drei geschachtelte Schleifen über alle Knoten haben und deshalb die Laufzeit $O(n^3)$:

- Der Algorithmus von **Warshall** bestimmt die *transitive Hülle*: Gibt es von i nach k und von k nach j einen Weg, wird auch $i \to j$ als erreichbar eingetragen.
- Der Algorithmus von **Floyd** (Floyd-Warshall) überträgt das auf gewichtete Graphen und bestimmt die *kürzesten Wege* zwischen allen Knotenpaaren.

Quelltext 2: Algorithmus von Warshall (transitive Hülle)

```
def warshall(self):
    r = []                                          # Kopie der Matrix: 1 = erreichbar
    for i in range(self.n):
        zeile = []
        for j in range(self.n):
            zeile.append(1 if self.matrix[i][j] != 0 else 0)
        r.append(zeile)
    for k in range(self.n):                         # Zwischenknoten
        for i in range(self.n):
            for j in range(self.n):
                if r[i][k] == 1 and r[k][j] == 1:
                    r[i][j] = 1
    return r
```

Enthält die Erreichbarkeitsmatrix am Ende nur Einsen, ist jeder Knoten von jedem anderen aus (eventuell über Umwege) erreichbar: Der Graph ist *zusammenhängend*. (Nicht zu verwechseln mit einem *vollständigen* Graphen, in dem jeder Knoten mit jedem *direkt* verbunden ist.)

Beim Algorithmus von Floyd steht in der Matrix die bisher kürzeste bekannte Länge; fehlende Kanten werden mit ∞ belegt. Für jeden Zwischenknoten k wird geprüft, ob der Umweg über k kürzer ist.

Quelltext 3: Algorithmus von Floyd (kürzeste Wege zwischen allen Paaren)

```
def floyd(self):
    d = []
    for i in range(self.n):
        zeile = []
        for j in range(self.n):
            w = self.matrix[i][j]
            zeile.append(w if w != 0 else float("inf"))
        d.append(zeile)
    for k in range(self.n):
        for i in range(self.n):
            for j in range(self.n):
                if d[i][k] + d[k][j] < d[i][j]:
                    d[i][j] = d[i][k] + d[k][j]
    return d
```

Floyd auf einen Blick

Datenstruktur	Matrix $n \times n$, drei geschachtelte Schleifen (k außen!)
Ergebnis	kürzeste Weglängen zwischen *allen* Paaren
Laufzeit	$O(n^3)$
Typischer Fehler	die Schleife über den Zwischenknoten k nach innen setzen – dann werden Umwege über mehrere Knoten nicht gefunden

Ablauf des Algorithmus von Floyd für den Graphen oben

Der Algorithmus nimmt nacheinander jeden Knoten $k = 0, 1, \ldots, 4$ als möglichen *Zwischenknoten*. Für jedes Paar (i, j) wird geprüft, ob der Umweg über k kürzer ist als der bisher bekannte Weg:

$$d[i][j] = \min\big(d[i][j],\ d[i][k] + d[k][j]\big)$$

So liest man die Matrizen: Der Umweg über k setzt sich aus einem Wert der *Zeile* k und einem Wert der *Spalte* k zusammen – beide sind grau hinterlegt. Für den Eintrag in Zeile i, Spalte j addiert man den grauen Wert in Zeile i (Spalte k) und den grauen Wert in Spalte j (Zeile k) und vergleicht die Summe mit dem alten Eintrag. **Fett** sind die Einträge, die sich in diesem Schritt verkürzt haben. Zeile und Spalte k selbst ändern sich in Schritt k nie.

Auf der Diagonalen steht zunächst ebenfalls ∞; dort entsteht am Ende die Länge des kürzesten *Rundwegs* durch den Knoten (z. B. $d[0][0] = 4$: 0 – 1 – 0). Wer die Diagonale mit 0 belegt, erhält außerhalb der Diagonalen dieselben Werte.

Start: direkte Kanten

x	0	1	2	3	4
0	∞	2	∞	4	∞
1	2	∞	5	1	∞
2	∞	5	∞	∞	2
3	4	1	∞	∞	3
4	∞	∞	2	3	∞

Fehlende Kanten (auch auf der Diagonalen) sind ∞.

Zwischenknoten $k = 0$

x	0	1	2	3	4
0	∞	2	∞	4	∞
1	2	**4**	5	1	∞
2	∞	5	∞	∞	2
3	4	1	∞	**8**	3
4	∞	∞	2	3	∞

Beispiel $d[1][1]$: bisher ∞, über 0: $d[1][0] + d[0][1] = 2 + 2 = 4 \Rightarrow$ neuer Wert. Insgesamt 2 Einträge verkürzt.

Zwischenknoten $k = 1$

x	0	1	2	3	4
0	**4**	2	**7**	**3**	∞
1	2	4	5	1	∞
2	**7**	5	**10**	**6**	2
3	**3**	1	**6**	**2**	3
4	∞	∞	2	3	∞

Beispiel $d[0][0]$: bisher ∞, über 1: $d[0][1] + d[1][0] = 2 + 2 = 4 \Rightarrow$ neuer Wert. Insgesamt 9 Einträge verkürzt.

Zwischenknoten $k = 2$

x	0	1	2	3	4
0	4	2	7	3	**9**
1	2	4	5	1	**7**
2	7	5	10	6	2
3	3	1	6	2	3
4	**9**	**7**	2	3	**4**

Beispiel $d[0][4]$: bisher ∞, über 2: $d[0][2] + d[2][4] = 7 + 2 = 9 \Rightarrow$ neuer Wert. Insgesamt 5 Einträge verkürzt.

Zwischenknoten $k = 3$

x	0	1	2	3	4
0	4	2	7	3	**6**
1	2	**2**	5	1	**4**
2	7	5	10	6	2
3	3	1	6	2	3
4	**6**	**4**	2	3	4

Beispiel $d[0][4]$: bisher 9, über 3: $d[0][3]+d[3][4] = 3 + 3 = 6 \Rightarrow$ kürzer. Insgesamt 5 Einträge verkürzt.

Zwischenknoten $k = 4$

x	0	1	2	3	4
0	4	2	7	3	6
1	2	2	5	1	4
2	7	5	**4**	**5**	2
3	3	1	**5**	2	3
4	6	4	2	3	4

Beispiel $d[2][2]$: bisher 10, über 4: $d[2][4] + d[4][2] = 2 + 2 = 4 \Rightarrow$ kürzer. Insgesamt 3 Einträge verkürzt.

Die Endmatrix enthält für jedes Paar (i, j) die Länge des kürzesten Weges, z. B. von 0 nach 4 die Länge 6 (0 – 1 – 3 – 4). Den Weg selbst liefert der Algorithmus so nicht; dafür müsste man – wie bei Dijkstra – zu jedem Eintrag zusätzlich den Zwischenknoten speichern.

1.3 Breitensuche und Tiefensuche

Ein *Spannbaum* eines zusammenhängenden Graphen ist ein Teilgraph, der alle Knoten enthält, zusammenhängend ist und keinen Zyklus hat. Ein Graph hat in der Regel viele Spannbäume. Breiten- und Tiefensuche besuchen von einem Startknoten aus alle erreichbaren Knoten; die Kanten, über die ein Knoten zum ersten Mal erreicht wird, bilden einen *Breitensuch-* bzw. *Tiefensuchbaum*.

Beide Verfahren sind gleich aufgebaut und unterscheiden sich nur in der Datenstruktur für die noch zu bearbeitenden Knoten: Die Breitensuche nutzt eine *Warteschlange* (Queue, FIFO: vorne entnehmen, hinten anfügen), die Tiefensuche einen *Stapel* (Stack, LIFO: oben ablegen und oben entnehmen).

Quelltext 4: Breitensuche mit einer Queue

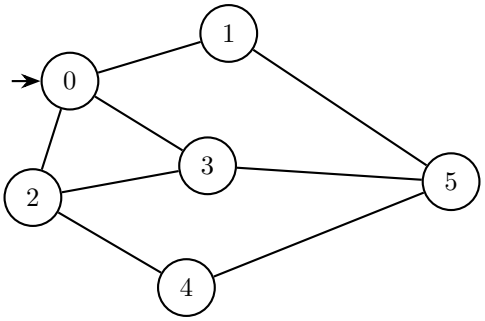
```
def breitensuche(self, start):
    besucht = [start]
    q = Queue()
    q.enqueue(start)
    while not q.empty():
        u = q.dequeue()
        for v in self.ad_liste[u]:
            if v not in besucht:                    # zum ersten Mal entdeckt
                besucht.append(v)
                q.enqueue(v)                        # Kante u-v gehört zum Baum
    return besucht
```

Quelltext 5: Tiefensuche mit einem Stack

```
def tiefensuche(self, start):
    besucht = []
    s = Stack()
    s.push(start)
    while not s.empty():
        u = s.pop()
        if u not in besucht:
            besucht.append(u)
            nb = self.ad_liste[u]
            for i in range(len(nb) - 1, -1, -1):    # rückwärts ablegen: kleinster zuerst
                if nb[i] not in besucht:
                    s.push(nb[i])
    return besucht
```

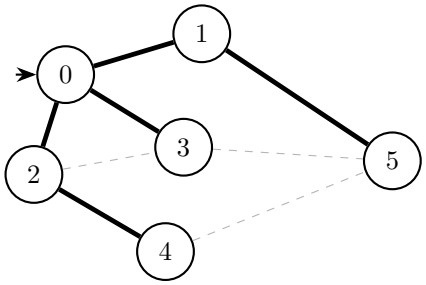
Breiten- und Tiefensuche auf einen Blick	
Datenstruktur	Queue (Breitensuche) bzw. Stack (Tiefensuche), Liste der besuchten Knoten
Ergebnis	Reihenfolge aller erreichbaren Knoten und ein Spannbaum; die Breitensuche findet Wege mit den *wenigsten Kanten*
Laufzeit	$O(n+m)$ mit Adjazenzliste (m = Anzahl der Kanten)
Typischer Fehler	Knoten erst beim Entnehmen als besucht markieren (Breitensuche) – dann landen sie mehrfach in der Queue

Beispiel



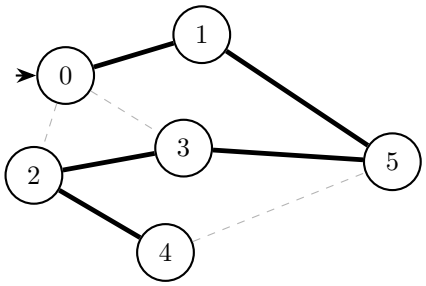
Breitensuche ab 0 – Reihenfolge 0, 1, 2, 3, 5, 4

Schritt	entnommen	Queue danach	besucht
1	0	1, 2, 3	0, 1, 2, 3
2	1	2, 3, 5	0, 1, 2, 3, 5
3	2	3, 5, 4	0, 1, 2, 3, 5, 4
4	3	5, 4	0, 1, 2, 3, 5, 4
5	5	4	0, 1, 2, 3, 5, 4
6	4	leer	0, 1, 2, 3, 5, 4



Tiefensuche ab 0 – Reihenfolge 0, 1, 5, 3, 2, 4

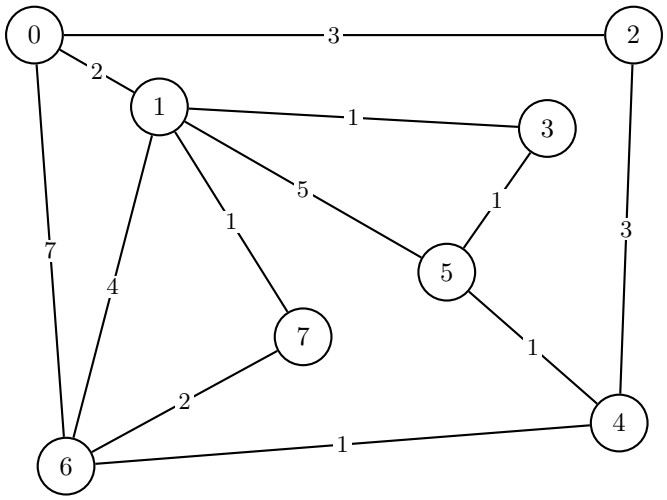
Schritt	entnommen	Stack danach	besucht
1	0	3, 2, 1	0
2	1	3, 2, 5	0, 1
3	5	3, 2, 4, 3	0, 1, 5
4	3	3, 2, 4, 2	0, 1, 5, 3
5	2	3, 2, 4, 4	0, 1, 5, 3, 2
6	4	3, 2, 4	0, 1, 5, 3, 2, 4



Bäume sind besondere Graphen. Wendet man die Tiefensuche auf einen Baum an und beginnt an der Wurzel, erhält man die *Preorder*-Reihenfolge (Wurzel, linker Teilbaum, rechter Teilbaum), z. B. für den Suchbaum mit den Schlüsseln 37, 26, 40, 25, 30, 38, 45, 27, 31: 37, 26, 25, 30, 27, 31, 40, 38, 45.

1.4 Der Algorithmus von Dijkstra

Der Algorithmus von Dijkstra bestimmt in einem Graphen mit *nichtnegativen* Kantengewichten die kürzesten Wege von einem Startknoten zu allen anderen Knoten. Er arbeitet ähnlich wie die Breitensuche, merkt sich aber zu jedem entdeckten Knoten die bisher kürzeste bekannte Entfernung und den Vorgänger auf diesem Weg. Statt einer Queue verwendet er eine *Prioritätswarteschlange*: Als Nächstes wird immer der noch nicht fertige Knoten mit der kleinsten Entfernung gewählt. Seine Entfernung steht damit endgültig fest – jeder andere Weg führt über einen Knoten mit mindestens gleich großer Entfernung und wird durch weitere Kanten nur länger. Findet man über den gewählten Knoten einen kürzeren Weg zu einem Nachbarn, werden Entfernung und Vorgänger des Nachbarn aktualisiert.



Quelltext 6: Algorithmus von Dijkstra mit Prioritätswarteschlange

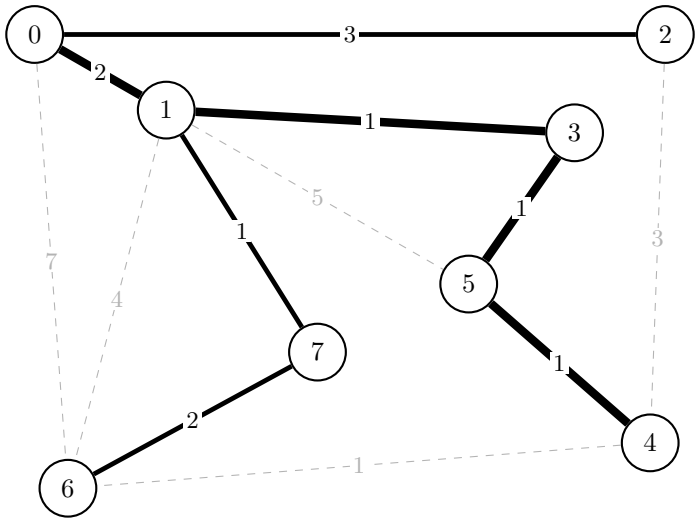
```
def dijkstra(self, start):
    dist, vorg = [], []
    for i in range(self.n):
        dist.append(float("inf"))
        vorg.append(None)
    dist[start] = 0
    fertig = []
    pq = PrioQueue()
    pq.enqueue(0, start)
    while not pq.empty():
        d, u = pq.dequeue()                          # kleinste Entfernung
        if u in fertig:
            continue                                 # veralteter Eintrag
        fertig.append(u)
        for v in self.ad_liste[u]:
            neu = dist[u] + self.matrix[u][v]
            if v not in fertig and neu < dist[v]:
                dist[v] = neu
                vorg[v] = u
                pq.enqueue(neu, v)
    return dist, vorg
```

Ablauf ab Knoten 0

Einträge: Vorgänger | Länge, ∞ = noch nicht erreicht, **fett** = endgültig, x = fertig. Jede Zeile zeigt den Stand, wenn der Knoten der ersten Spalte gewählt wird.

gewählt	0	1	2	3	4	5	6	7
0	**0**	∞	∞	∞	∞	∞	∞	∞
1	x	**0 | 2**	0 | 3	∞	∞	∞	0 | 7	∞
2	x	x	**0 | 3**	1 | 3	∞	1 | 7	1 | 6	1 | 3
3	x	x	x	**1 | 3**	2 | 6	1 | 7	1 | 6	1 | 3
7	x	x	x	x	2 | 6	3 | 4	1 | 6	**1 | 3**
5	x	x	x	x	2 | 6	**3 | 4**	7 | 5	x
4	x	x	x	x	**5 | 5**	x	7 | 5	x
6	x	x	x	x	x	x	**7 | 5**	x

Den kürzesten Weg liest man rückwärts über die Vorgänger ab. Zum Knoten 4 etwa: 4 hat den Vorgänger 5, 5 den Vorgänger 3, 3 den Vorgänger 1, 1 den Vorgänger 0 – also 0 – 1 – 3 – 5 – 4 mit der Länge 5. Die fetten Kanten im Bild unten bilden den *Kürzeste-Wege-Baum*, der Weg zu Knoten 4 ist besonders hervorgehoben.



Dijkstra auf einen Blick	
Datenstruktur	Prioritätswarteschlange (Entfernung, Knoten), Listen `dist` und `vorg`
Ergebnis	kürzeste Wege von *einem* Startknoten zu allen Knoten (Kürzeste-Wege-Baum)
Laufzeit	$O(n^2)$ mit einfacher Liste, $O((n+m)\log n)$ mit Heap
Typischer Fehler	den Weg vorwärts ablesen wollen – er wird *rückwärts* über die Vorgänger rekonstruiert; negative Kantengewichte (dann liefert Dijkstra falsche Ergebnisse, siehe Abschnitt 4)

1.5 Der Algorithmus von Kruskal

Gesucht ist ein *minimaler Spannbaum*: ein Spannbaum, dessen Kantengewichte zusammen möglichst klein sind (z. B. das günstigste Leitungsnetz, das alle Orte verbindet). Joseph B. Kruskal (* 1928 in New York City) veröffentlichte 1956 ein einfaches Verfahren dafür. Ein anderes Verfahren für dasselbe Problem geht auf Vojtěch Jarník (1930) zurück und wurde später von Robert C. Prim neu entdeckt (Abschnitt 1.6).

Beide Verfahren arbeiten nach dem *Greedy-Prinzip* („gierig“): In jedem Schritt wird die lokal beste Wahl getroffen – hier die leichteste noch passende Kante –, ohne frühere Entscheidungen zu revidieren. Für minimale Spannbäume führt das nachweislich zum globalen Optimum (das gilt nicht für jedes Problem).

Vorgehen:

1. Alle Kanten werden aufsteigend nach ihrem Gewicht sortiert.
2. Jeder Knoten bildet zunächst einen eigenen Teilbaum (er ist sein eigener „Vater“, also Wurzel).
3. Die leichteste noch nicht geprüfte Kante wird betrachtet: Liegen ihre Endknoten in *verschiedenen* Teilbäumen, wird sie aufgenommen und die beiden Teilbäume werden vereinigt (*union*). Liegen sie im selben Teilbaum, würde sie einen Zyklus schließen und wird verworfen.
4. Das wird wiederholt, bis der Baum $n - 1$ Kanten hat (oder die Kantenliste leer ist).

Ob zwei Knoten im selben Teilbaum liegen, entscheidet die *Union-Find*-Struktur: Jeder Knoten speichert seinen Vater; `finde` läuft die Väter bis zur Wurzel hoch, `vereinige` hängt eine Wurzel unter die andere.

Quelltext 7: Algorithmus von Kruskal mit Union-Find

```
def finde(self, x):                                  # Wurzel des Teilbaums von x
    while self.vater[x] != x:
        x = self.vater[x]
    return x

def kruskal(self):
    kanten = []
    for u in range(self.n):
        for v in range(u + 1, self.n):               # jede Kante nur einmal
            if self.matrix[u][v] != 0:
                kanten.append((self.matrix[u][v], u, v))
    kanten.sort()                                    # nach Gewicht (erstes Tupelelement)
    self.vater = []
    for i in range(self.n):
        self.vater.append(i)                         # jeder Knoten ist eigene Wurzel
    baum = []
    for w, u, v in kanten:
        if len(baum) == self.n - 1:
            break
        wu = self.finde(u)
        wv = self.finde(v)
        if wu != wv:                                 # verschiedene Teilbäume
            self.vater[wv] = wu                      # vereinigen: wu wird Vater von wv
            baum.append((u, v, w))
    return baum
```

Trockenlauf für den Graphen aus Abschnitt 1.1

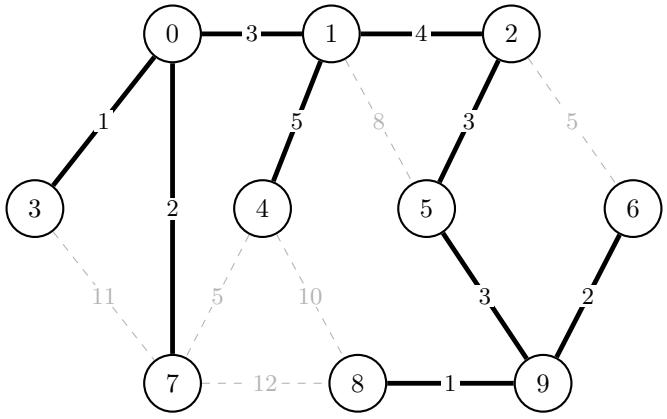
Sortierte Kantenliste (Gewicht, Knoten, Knoten), jede Kante einmal:
(0,3,1), (8,9,1), (0,7,2), (6,9,2), (0,1,3), (2,5,3), (5,9,3), (1,2,4), (1,4,5), (2,6,5), (4,7,5), (1,5,8), (4,8,10), (3,7,11), (7,8,12)

In den Spalten steht nach jedem Schritt der Vater jedes Knotens (**fett**: in diesem Schritt geändert). Die Wurzel des ersten Endknotens wird Vater der Wurzel des zweiten.

Kante	0	1	2	3	4	5	6	7	8	9
0–3 (1)	0	1	2	**0**	4	5	6	7	8	9
8–9 (1)	0	1	2	0	4	5	6	7	8	**8**
0–7 (2)	0	1	2	0	4	5	6	**0**	8	8
6–9 (2)	0	1	2	0	4	5	6	0	**6**	8
0–1 (3)	0	**0**	2	0	4	5	6	0	6	8
2–5 (3)	0	0	2	0	4	**2**	6	0	6	8
5–9 (3)	0	0	2	0	4	2	**2**	0	6	8
1–2 (4)	0	0	**0**	0	4	2	2	0	6	8
1–4 (5)	0	0	0	0	**0**	2	2	0	6	8

So liest man die Tabelle: Bei der Kante 5–9 (3) ist die Wurzel von 5 der Knoten 2 ($5 \to 2$) und die Wurzel von 9 der Knoten 6 ($9 \to 8 \to 6$). Die Wurzeln sind verschieden, also wird vereinigt: 2 wird Vater von 6. Bei 1–2 sind die Wurzeln 0 und 2 – auch das passt. Eine Kante, deren beide Endknoten dieselbe Wurzel haben, liegt innerhalb eines Teilbaums und würde einen Zyklus schließen.

Nach der neunten aufgenommenen Kante hat der Baum $n-1=9$ Kanten; die übrigen Kanten (2–6, 4–7, 1–5, 4–8, 3–7, 7–8) müssen nicht mehr geprüft werden – jede von ihnen würde einen Zyklus schließen.



Gesamtgewicht: $1+1+2+2+3+3+3+4+5=24$

Kruskal auf einen Blick	
Datenstruktur	sortierte Kantenliste, Union-Find (Vaterliste)
Ergebnis	minimaler Spannbaum (bei nicht zusammenhängenden Graphen ein minimaler Spannwald)
Laufzeit	$O(m \log m)$ für das Sortieren
Typischer Fehler	nur prüfen, ob die beiden Endknoten schon „im Baum“ sind – entscheidend ist, ob sie im *selben Teilbaum* liegen (sonst werden nötige Verbindungskanten verworfen)

1.6 Der Algorithmus von Prim

Auch der Algorithmus von Prim liefert einen minimalen Spannbaum eines zusammenhängenden, gewichteten, ungerichteten Graphen. Anders als Kruskal lässt er *einen* Baum von einem Startknoten aus wachsen:

1. Der Baum besteht anfangs nur aus einem beliebigen Startknoten.
2. Alle Kanten vom Startknoten zu seinen Nachbarn kommen in eine Prioritätswarteschlange.
3. Die leichteste Kante wird entnommen. Führt sie zu einem Knoten, der noch nicht im Baum ist, wird sie aufgenommen, und die Kanten vom neuen Knoten zu Knoten außerhalb des Baums werden eingereiht. Führt sie zu einem Knoten, der schon im Baum ist, wird sie verworfen (sie würde einen Zyklus schließen).
4. Das wird wiederholt, bis alle Knoten im Baum sind.

Die Effizienz hängt davon ab, wie schnell die leichteste Kante gefunden wird. Deshalb verwendet man eine Prioritätswarteschlange, die in der Regel als *Heap* organisiert ist (in Python: Modul `heapq`).

Quelltext 8: Algorithmus von Prim mit heapq

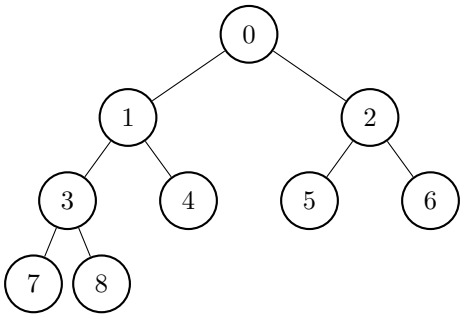
```
import heapq

def prim(self, start):
    im_baum = [start]
    baum = []
    heap = []
    for v in self.ad_liste[start]:
        heapq.heappush(heap, (self.matrix[start][v], start, v))
    while heap and len(im_baum) < self.n:
        w, u, v = heapq.heappop(heap)                  # leichteste Kante
        if v in im_baum:
            continue                                   # würde Zyklus schließen
        im_baum.append(v)
        baum.append((u, v, w))
        for x in self.ad_liste[v]:
            if x not in im_baum:
                heapq.heappush(heap, (self.matrix[v][x], v, x))
    return baum
```

Heap als Array: Die Baumstruktur eines Heaps existiert nur logisch. Jede Position i im Array ist ein Knoten; Eltern und Kinder ergeben sich aus dem Index:

$$\text{eltern}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor, \qquad \text{links}(i) = 2i+1, \qquad \text{rechts}(i) = 2i+2$$

Beim Einfügen steigt ein Element auf (`sift-up`), beim Entnehmen des Minimums rückt das letzte Element an die Wurzel und sinkt ab (`sift-down`). Beispiel: $\text{parent}(3) = \lfloor 2/2 \rfloor = 1$; $\text{left}(1) = 3$, $\text{right}(1) = 4$.



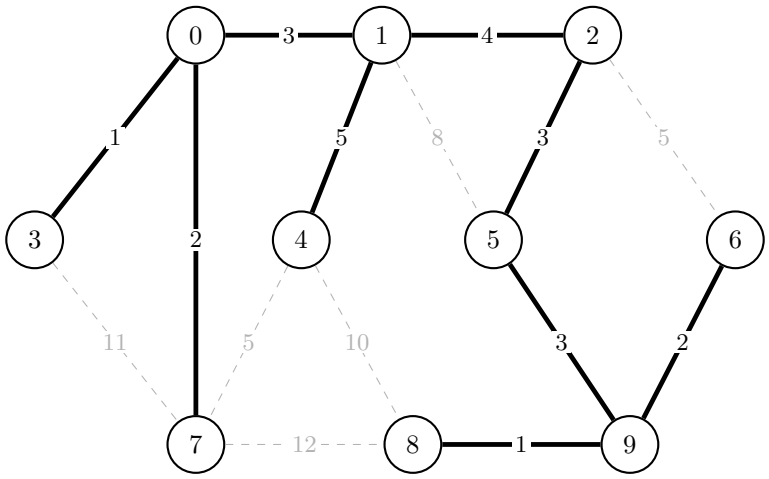
Indizes eines Heaps mit neun Elementen

Trockenlauf ab Knoten 0 (Graph aus Abschnitt 1.1)

Die Prioritätswarteschlange ist zur besseren Lesbarkeit aufsteigend sortiert dargestellt; intern liegt sie als Heap vor, entnommen wird immer das erste Element.

Schritt	Kante	Prioritätswarteschlange danach (Gewicht: Kante)	neu im Baum
0	Start	1: 0–3, 2: 0–7, 3: 0–1	0
1	**0–3**	2: 0–7, 3: 0–1, 11: 3–7	3
2	**0–7**	3: 0–1, 5: 7–4, 11: 3–7, 12: 7–8	7
3	**0–1**	4: 1–2, 5: 1–4, 5: 7–4, 8: 1–5, 11: 3–7, 12: 7–8	1
4	**1–2**	3: 2–5, 5: 1–4, 5: 2–6, 5: 7–4, 8: 1–5, 11: 3–7, 12: 7–8	2
5	**2–5**	3: 5–9, 5: 1–4, 5: 2–6, 5: 7–4, 8: 1–5, 11: 3–7, 12: 7–8	5
6	**5–9**	1: 9–8, 2: 9–6, 5: 1–4, 5: 2–6, 5: 7–4, 8: 1–5, 11: 3–7, 12: 7–8	9
7	**9–8**	2: 9–6, 5: 1–4, 5: 2–6, 5: 7–4, 8: 1–5, 10: 8–4, 11: 3–7, 12: 7–8	8
8	**9–6**	5: 1–4, 5: 2–6, 5: 7–4, 8: 1–5, 10: 8–4, 11: 3–7, 12: 7–8	6
9	**1–4**	5: 2–6, 5: 7–4, 8: 1–5, 10: 8–4, 11: 3–7, 12: 7–8	4

Aufgenommene Kanten: 0–3, 0–7, 0–1, 1–2, 2–5, 5–9, 9–8, 9–6, 1–4. Bei der Kante mit Gewicht 5 gibt es einen Gleichstand (1–4 und 4–7); beide führen zu einem minimalen Spannbaum mit demselben Gesamtgewicht.



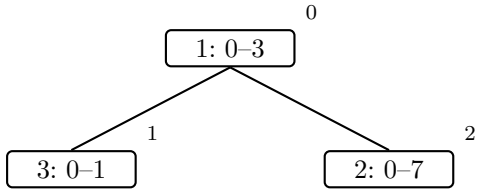
Gesamtgewicht: $1+2+3+4+3+3+1+2+5 = 24$ – wie bei Kruskal; ein minimaler Spannbaum ist nicht immer eindeutig, sein Gewicht schon.

Wie der Heap im Trockenlauf aussieht

`heapq` speichert die Prioritätswarteschlange als gewöhnliche Python-Liste. Jedes Element ist ein Tupel (Gewicht, von, nach); Tupel werden zuerst nach dem Gewicht verglichen. Die *Heap-Bedingung* lautet: Jedes Element ist höchstens so groß wie seine Kinder – das kleinste steht daher immer bei Index 0.

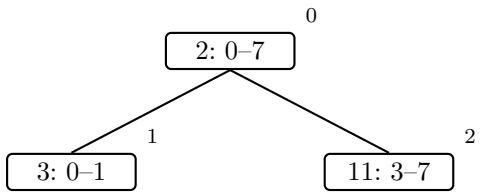
Zeile 0 (Start bei 0). Die Kanten zu den Nachbarn 1, 3, 7 werden nacheinander eingefügt:

1. `(3, 0, 1)` kommt in die leere Liste an Index 0.
2. `(1, 0, 3)` wird hinten angehängt (Index 1). Sein Elternteil ist Index $\lfloor (1-1)/2 \rfloor = 0$ mit Gewicht 3. Da $1 < 3$, werden beide getauscht (`sift-up`).
3. `(2, 0, 7)` kommt an Index 2; Elternteil Index 0 mit Gewicht 1. Da $2 > 1$, bleibt es stehen.



Liste: `[(1, 0, 3), (3, 0, 1), (2, 0, 7)]`
Die Liste ist *nicht* sortiert – (3, 0, 1) steht vor (2, 0, 7). Für die Heap-Bedingung genügt, dass jedes Element höchstens so groß ist wie seine Kinder.

Zeile 1 (Kante 0–3 wird entnommen). `heappop` gibt das Element bei Index 0 zurück, also `(1, 0, 3)`. Das letzte Element `(2, 0, 7)` rückt an die Wurzel und wird mit seinen Kindern verglichen (`sift-down`): Das einzige Kind ist (3, 0, 1); da $2 < 3$, bleibt es oben. Zwischenstand: `[(2, 0, 7), (3, 0, 1)]`. Danach werden die Kanten vom neuen Baumknoten 3 zu Knoten außerhalb des Baums eingefügt – nur `(11, 3, 7)`; sie kommt an Index 2, ihr Elternteil (Gewicht 2) ist kleiner, also kein Tausch.



Liste: `[(2, 0, 7), (3, 0, 1), (11, 3, 7)]`
Als Nächstes wird (2, 0, 7) entnommen – genau die Kante 0–7 aus Zeile 2 der Tabelle.

Prim auf einen Blick	
Datenstruktur	Prioritätswarteschlange (Heap) mit Kanten, Liste der Baumknoten
Ergebnis	minimaler Spannbaum, wächst von einem Startknoten aus
Laufzeit	$O(m \log m)$ mit Heap
Typischer Fehler	beim Entnehmen nicht prüfen, ob der Zielknoten schon im Baum ist – dann entstehen Zyklen und zu viele Kanten

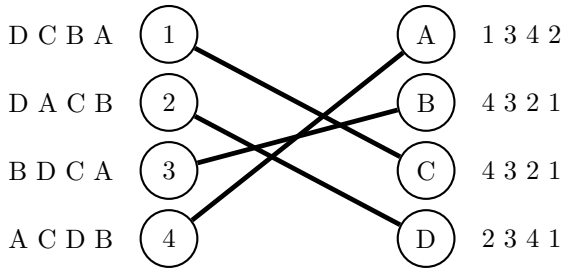
2 Ausgewählte Probleme

2.1 Das Heiratsproblem

Das Heiratsproblem ist der Ausgangspunkt für viele Zuordnungsprobleme: Bewerberinnen und Bewerber sollen Stellen, Studierende Studienplätze erhalten. Anschaulich geht es um zwei gleich große Gruppen – im klassischen Beispiel Männer und Frauen –, in denen jede Person eine eigene Rangliste (*Präferenzliste*) der anderen Gruppe hat.

Graphentheoretisch beginnt man mit zwei disjunkten Knotenmengen; gesucht ist eine *Paarung* (Matching): eine Teilmenge der Kanten, in der keine zwei Kanten einen gemeinsamen Knoten haben – jede Person hat höchstens einen Partner. Eine Paarung, in der jede Person einen Partner hat, heißt *perfekt*.

Eine Paarung heißt **stabil**, wenn es kein Paar (m, f) gibt, das nicht miteinander verbunden ist, sich aber gegenseitig ihren jeweiligen Partnern vorziehen würde. Ein solches Paar würde die Zuordnung „sprengen“.



Links die Präferenzlisten der Männer, rechts die der Frauen (vorne = am liebsten); fett die stabile Paarung

2.2 Der Algorithmus von Gale und Shapley



David Gale und Lloyd Shapley

David Gale und Lloyd Shapley zeigten 1962, dass es immer eine stabile Paarung gibt, und gaben ein Verfahren an, das sie findet: Solange es einen freien Mann gibt, macht er der Frau, die auf seiner Liste am weitesten vorne steht und der er noch keinen Antrag gemacht hat, einen Antrag. Ist sie frei, verloben sich die beiden. Ist sie verlobt, entscheidet sie nach ihrer Liste: Zieht sie den Antragsteller vor, löst sie die Verlobung, und der bisherige Partner ist wieder frei; andernfalls bleibt sie bei ihrem Partner, und der Antragsteller versucht es bei der Nächsten.

Das Ergebnis ist immer stabil und für die werbende Seite optimal: Jeder Mann erhält die beste Partnerin, die er in irgendeiner stabilen Paarung bekommen könnte. Werben die Frauen, ist es umgekehrt.

Ablauf für das Beispiel:

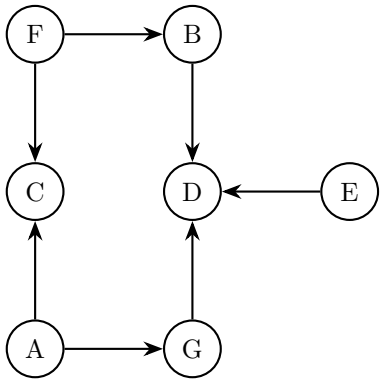
Antrag von	an	Ergebnis
1	D	D ist frei: Verlobung 1–D
2	D	D zieht 2 dem bisherigen Partner 1 vor: 1 wird frei
3	B	B ist frei: Verlobung 3–B
4	A	A ist frei: Verlobung 4–A
1	C	C ist frei: Verlobung 1–C

Ergebnis: 1–C, 2–D, 3–B, 4–A – eine perfekte und stabile Paarung. Nur Mann 1 muss einen zweiten Antrag machen, weil D den Mann 2 vorzieht.

Quelltext 9: Algorithmus von Gale und Shapley (Männer werben)

```
def gale_shapley(maenner, frauen):
    # maenner[m] und frauen[f]: Präferenzlisten, vorne = am liebsten
    naechste = {}                                   # Position des nächsten Antrags
    for m in maenner:
        naechste[m] = 0
    verlobt = {}                                    # Frau -> Mann
    frei = list(maenner)
    while frei != []:
        m = frei.pop(0)
        f = maenner[m][naechste[m]]
        naechste[m] += 1
        if f not in verlobt:
            verlobt[f] = m
        elif frauen[f].index(m) < frauen[f].index(verlobt[f]):
            frei.append(verlobt[f])                 # bisheriger Partner wird frei
            verlobt[f] = m
        else:
            frei.append(m)                          # Antrag abgelehnt
    return verlobt
```

3 Topologische Sortierung



Die topologische Sortierung bringt die Knoten eines gerichteten, azyklischen Graphen in eine Reihenfolge, in der jede Kante von einem früheren zu einem späteren Knoten zeigt – etwa eine Reihenfolge von Arbeitsschritten, die ihre Abhängigkeiten einhält.
Zuerst wird für jeden Knoten der *Eingangsgrad* (EG) bestimmt, die Anzahl der Kanten, die zu ihm führen:

Knoten	A	B	C	D	E	F	G
EG	0	1	2	3	0	0	1

Vorgehen (Algorithmus von Kahn): Alle Knoten mit EG 0 kommen in die Liste. Die Liste wird von vorne abgearbeitet: Für jeden Nachfolger des aktuellen Knotens sinkt der Eingangsgrad um 1; erreicht er 0, wird der Nachfolger hinten an die Liste angehängt. Bleiben am Ende Knoten übrig, enthält der Graph einen Zyklus.

bearbeitet	A	B	C	D	E	F	G	neu mit EG 0	Liste
Start	0	1	2	3	0	0	1		A – E – F
A	0	1	**1**	3	0	0	**0**	G	A – E – F – G
E	0	1	1	**2**	0	0	0	–	A – E – F – G
F	0	**0**	**0**	2	0	0	0	B, C	A – E – F – G – B – C
G	0	0	0	**1**	0	0	0	–	A – E – F – G – B – C
B	0	0	0	**0**	0	0	0	D	A – E – F – G – B – C – D

Ergebnis: A – E – F – G – B – C – D (fett: in diesem Schritt verringerter Eingangsgrad). Eine topologische Sortierung ist nicht eindeutig; jede Reihenfolge, die alle Kanten respektiert, ist richtig.

Quelltext 10: Topologische Sortierung

```
def top_sort(self):
    e_grad = []
    t_sort = []
    for i in range(self.n):
        e_grad.append(self.eingangsgrad(i))
        if e_grad[i] == 0:
            t_sort.append(i)
    c = 0
    while len(t_sort) < self.n:
        if c >= len(t_sort):
            return "Zyklus"                          # kein Knoten mehr mit EG 0
        for v in self.ad_liste[t_sort[c]]:           # Nachfolger
            e_grad[v] -= 1
            if e_grad[v] == 0:
                t_sort.append(v)
        c += 1
    return t_sort
```

4 Überblick: Welches Verfahren wofür?

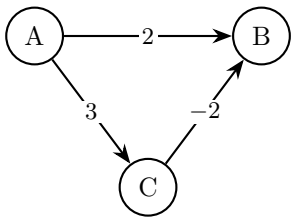
Verfahren	Frage	Idee	Datenstruktur	Laufzeit
Breitensuche	Welche Knoten sind erreichbar? Weg mit den wenigsten Kanten?	schichtweise vom Start aus	Queue	$O(n+m)$
Tiefensuche	Welche Knoten sind erreichbar? Gibt es Zyklen?	so tief wie möglich, dann zurück	Stack (oder Rekursion)	$O(n+m)$
Dijkstra	kürzeste Wege von *einem* Start (Gewichte ≥ 0)	immer den nächstgelegenen offenen Knoten festlegen	Prioritätswarte-schlange	$O((n+m)\log n)$
Floyd	kürzeste Wege zwischen *allen* Paaren	Umweg über jeden Zwischenknoten k prüfen	Matrix	$O(n^3)$
Warshall	Wer erreicht wen?	wie Floyd, nur mit 0/1	Matrix	$O(n^3)$
Kruskal	minimaler Spannbaum	leichteste Kante ohne Zyklus, Teilbäume vereinigen	sortierte Kanten, Union-Find	$O(m \log m)$
Prim	minimaler Spannbaum	einen Baum über die leichteste Randkante wachsen lassen	Heap	$O(m \log m)$
Kahn	Reihenfolge mit Abhängigkeiten (gerichtet, azyklisch)	Knoten mit Eingangsgrad 0 zuerst	Liste/Queue, Eingangsgrade	$O(n+m)$
Gale-Shapley	stabile Paarung zweier Gruppen	Anträge nach Präferenz, Verlobung lösbar	Präferenzlisten	$O(n^2)$

n = Anzahl der Knoten, m = Anzahl der Kanten.

Verwechslungsgefahr: kürzester Weg ≠ minimaler Spannbaum

Dijkstra und Prim sehen sich im Code sehr ähnlich (Prioritätswarteschlange, Knoten werden nacheinander „fertig“). Der Unterschied liegt in der Priorität: Dijkstra ordnet nach der *Entfernung vom Start* ($d[u] + w$), Prim nach dem *Gewicht der einzelnen Kante* (w). Im Graphen aus 1.4 enthält der Kürzeste-Wege-Baum von 0 aus die Kante 0–1 mit Gewicht 2; ein minimaler Spannbaum muss dagegen nicht jeden Knoten auf dem kürzesten Weg mit dem Start verbinden – er minimiert die *Summe aller* Kanten.

Warum Dijkstra keine negativen Gewichte verträgt



Ab A wählt Dijkstra zuerst B mit der Entfernung 2 und erklärt sie für endgültig. Danach wird C (3) gewählt; der Weg A – C – B hat aber nur die Länge $3 + (-2) = 1$. Die Begründung „jeder andere Weg wird durch weitere Kanten nur länger“ gilt nur für nichtnegative Gewichte. Für negative Gewichte gibt es andere Verfahren (z. B. Bellman-Ford).

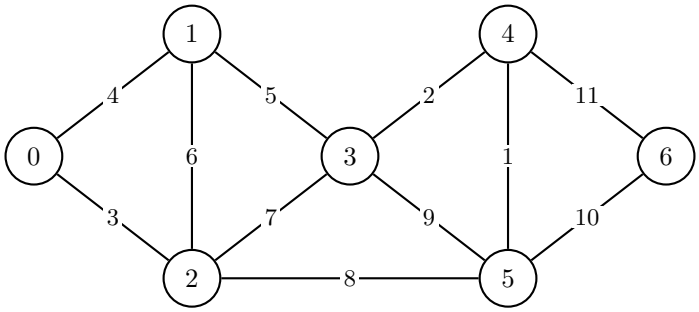
Selbstkontrolle

1. Warum markiert die Breitensuche einen Knoten schon beim Einreihen als besucht und nicht erst beim Entnehmen?
2. Wie viele Kanten hat ein Spannbaum eines zusammenhängenden Graphen mit n Knoten – und warum?
3. Bei Kruskal liegen die Endknoten einer Kante beide schon in Teilbäumen. Wann wird die Kante trotzdem aufgenommen?
4. Was steht nach dem Algorithmus von Floyd auf der Diagonalen, wenn man mit ∞ startet?
5. Wann gibt die topologische Sortierung „Zyklus“ zurück?

Lösungshinweise: 1. Sonst kann ein Knoten mehrfach in der Queue landen. 2. $n-1$: jede Kante verbindet einen weiteren Knoten, eine Kante mehr würde einen Zyklus schließen. 3. Wenn die Teilbäume verschieden sind (verschiedene Wurzeln). 4. Die Länge des kürzesten Rundwegs durch den Knoten. 5. Wenn die Liste abgearbeitet ist, aber noch Knoten fehlen – deren Eingangsgrad wird nie 0.

5 Übungen

Die Aufgaben 1 bis 5 beziehen sich auf den folgenden Graphen. Alle Kantengewichte sind verschieden, es gibt also keine Gleichstände. Werden mehrere Nachbarn gleichzeitig eingereiht oder abgelegt, gilt die Reihenfolge der Knotennummern (bei der Tiefensuche: der kleinste Nachbar wird zuerst besucht, siehe Quelltext 5).



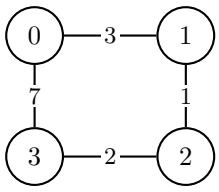
Aufgabe 1: **Darstellung.** Stellen Sie die Adjazenzmatrix und die Adjazenzliste auf. Bestimmen Sie den Grad jedes Knotens und begründen Sie, warum die Summe aller Grade immer doppelt so groß ist wie die Anzahl der Kanten.

Aufgabe 2: **Suchen.** Führen Sie ab Knoten 0 eine Breitensuche und eine Tiefensuche durch. Notieren Sie nach jedem Schritt den Inhalt der Queue bzw. des Stacks und die Liste der besuchten Knoten.

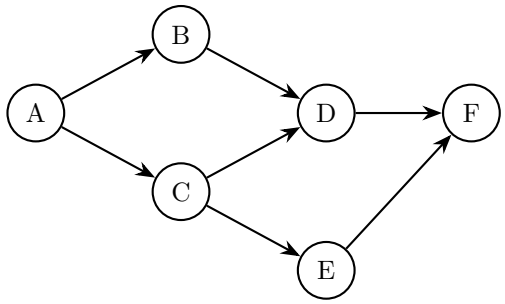
Aufgabe 3: **Dijkstra.** Bestimmen Sie mit dem Algorithmus von Dijkstra die kürzesten Wege ab Knoten 0 (Tableau mit Vorgänger | Länge). Geben Sie den kürzesten Weg von 0 nach 6 und seine Länge an.

Aufgabe 4: **Kruskal.** Bestimmen Sie einen minimalen Spannbaum mit dem Algorithmus von Kruskal. Führen Sie die Vaterliste der Union-Find-Struktur mit und kennzeichnen Sie verworfene Kanten.

Aufgabe 5: **Prim.** Bestimmen Sie den minimalen Spannbaum mit dem Algorithmus von Prim ab Knoten 0 und vergleichen Sie mit Aufgabe 4. In welcher Reihenfolge werden die Kanten aufgenommen?



Aufgabe 6: **Floyd.** Bestimmen Sie für den nebenstehenden Graphen die Entfernungsmatrix mit dem Algorithmus von Floyd. Belegen Sie die Diagonale mit 0 und fehlende Kanten mit ∞. Welche Einträge ändern sich bei welchem Zwischenknoten k?



Aufgabe 7: Topologische Sortierung. Ein Programmierprojekt besteht aus den Schritten A (Aufgabe analysieren), B (Klassen entwerfen), C (Testfälle festlegen), D (programmieren), E (Testdaten erzeugen) und F (testen und abgeben). Ein Pfeil X $\rightarrow$ Y bedeutet: X muss vor Y erledigt sein. Bestimmen Sie mit dem Algorithmus von Kahn eine zulässige Reihenfolge.

Aufgabe 8: Gale-Shapley. Drei Männer und drei Frauen haben folgende Präferenzlisten (vorne = am liebsten):

Mann 1: A, B, C	Frau A: 2, 1, 3
Mann 2: A, C, B	Frau B: 1, 3, 2
Mann 3: B, A, C	Frau C: 1, 2, 3

Bestimmen Sie mit dem Algorithmus von Gale und Shapley (Männer werben, freie Männer stellen sich hinten an) die Paarung und prüfen Sie, ob sie stabil ist.

Aufgabe 9: Nachdenken. (a) Begründen Sie, warum der Algorithmus von Prim bei einem nicht zusammenhängenden Graphen keinen Spannbaum liefert. Was liefert Kruskal? (b) Im Graphen von Aufgabe 3 wird das Gewicht der Kante 2–5 von 8 auf 13 erhöht. Wie lautet nun der kürzeste Weg von 0 nach 6? Begründen Sie ohne neue Rechnung, dass sich der minimale Spannbaum nicht ändert.

6 Lösungen

Aufgabe 1

x	0	1	2	3	4	5	6
0	0	4	3	0	0	0	0
1	4	0	6	5	0	0	0
2	3	6	0	7	0	8	0
3	0	5	7	0	2	9	0
4	0	0	0	2	0	1	11
5	0	0	8	9	1	0	10
6	0	0	0	0	11	10	0

Adjazenzliste:
$0 \mapsto 1, 2$
$1 \mapsto 0, 2, 3$
$2 \mapsto 0, 1, 3, 5$
$3 \mapsto 1, 2, 4, 5$
$4 \mapsto 3, 5, 6$
$5 \mapsto 2, 3, 4, 6$
$6 \mapsto 4, 5$

Grade: $d(0) = 2$, $d(1) = 3$, $d(2) = 4$, $d(3) = 4$, $d(4) = 3$, $d(5) = 4$, $d(6) = 2$. Summe: $22 = 2 \cdot 11$. Jede Kante hat zwei Enden und wird deshalb bei genau zwei Knoten gezählt (*Handschlaglemma*).

Aufgabe 2

Breitensuche – Reihenfolge 0, 1, 2, 3, 5, 4, 6

Schritt	entnommen	Queue danach	besucht
1	0	1, 2	0, 1, 2
2	1	2, 3	0, 1, 2, 3
3	2	3, 5	0, 1, 2, 3, 5
4	3	5, 4	0, 1, 2, 3, 5, 4
5	5	4, 6	0, 1, 2, 3, 5, 4, 6
6	4	6	0, 1, 2, 3, 5, 4, 6
7	6	leer	0, 1, 2, 3, 5, 4, 6

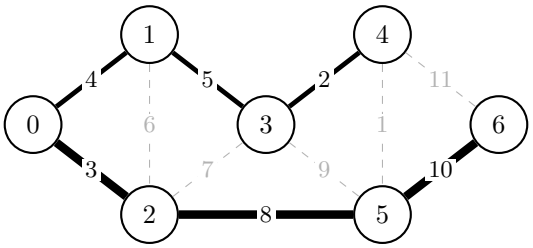
Tiefensuche – Reihenfolge 0, 1, 2, 3, 4, 5, 6

Schritt	entnommen	Stack danach	besucht
1	0	2, 1	0
2	1	2, 3, 2	0, 1
3	2	2, 3, 5, 3	0, 1, 2
4	3	2, 3, 5, 5, 4	0, 1, 2, 3
5	4	2, 3, 5, 5, 6, 5	0, 1, 2, 3, 4
6	5	2, 3, 5, 5, 6, 6	0, 1, 2, 3, 4, 5
7	6	2, 3, 5, 5, 6	0, 1, 2, 3, 4, 5, 6

Aufgabe 3

gewählt	0	1	2	3	4	5	6
0	**0**	∞	∞	∞	∞	∞	∞
2	x	$0 \mid 4$	$\mathbf{0 \mid 3}$	∞	∞	∞	∞
1	x	$\mathbf{0 \mid 4}$	x	$2 \mid 10$	∞	$2 \mid 11$	∞
3	x	x	x	$\mathbf{1 \mid 9}$	∞	$2 \mid 11$	∞
4	x	x	x	x	$\mathbf{3 \mid 11}$	$2 \mid 11$	∞
5	x	x	x	x	x	$\mathbf{2 \mid 11}$	$4 \mid 22$
6	x	x	x	x	x	x	$\mathbf{5 \mid 21}$

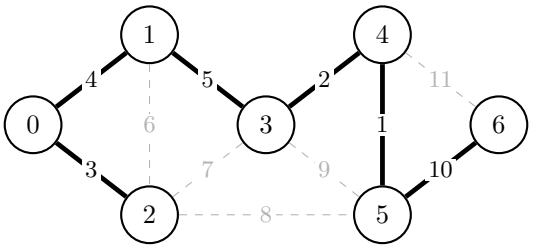
Kürzester Weg von 0 nach 6: 0 – 2 – 5 – 6 mit der Länge 21 (rückwärts über die Vorgänger gelesen).



Aufgabe 4

Kante	0	1	2	3	4	5	6	
4–5 (1)	0	1	2	3	4	**4**	6	
3–4 (2)	0	1	2	3	**3**	4	6	
0–2 (3)	0	1	**0**	3	3	4	6	
0–1 (4)	0	**0**	0	3	3	4	6	
1–3 (5)	0	0	0	**0**	3	4	6	
1–2 (6)	0	0	0	0	3	4	6	Zyklus
2–3 (7)	0	0	0	0	3	4	6	Zyklus
2–5 (8)	0	0	0	0	3	4	6	Zyklus
3–5 (9)	0	0	0	0	3	4	6	Zyklus
5–6 (10)	0	0	0	0	3	4	**0**	

Gesamtgewicht: $1 + 2 + 3 + 4 + 5 + 10 = 25$.



Aufgabe 5

Schritt	Kante	Prioritätswarteschlange danach (Gewicht: Kante)	neu im Baum
0	Start	3: 0–2, 4: 0–1	0
1	**0–2**	4: 0–1, 6: 2–1, 7: 2–3, 8: 2–5	2
2	**0–1**	5: 1–3, 6: 2–1, 7: 2–3, 8: 2–5	1
3	**1–3**	2: 3–4, 6: 2–1, 7: 2–3, 8: 2–5, 9: 3–5	3
4	**3–4**	1: 4–5, 6: 2–1, 7: 2–3, 8: 2–5, 9: 3–5, 11: 4–6	4
5	**4–5**	6: 2–1, 7: 2–3, 8: 2–5, 9: 3–5, 10: 5–6, 11: 4–6	5
6	2–1 verworfen	7: 2–3, 8: 2–5, 9: 3–5, 10: 5–6, 11: 4–6	–
7	2–3 verworfen	8: 2–5, 9: 3–5, 10: 5–6, 11: 4–6	–
8	2–5 verworfen	9: 3–5, 10: 5–6, 11: 4–6	–
9	3–5 verworfen	10: 5–6, 11: 4–6	–
10	**5–6**	11: 4–6	6

Aufgenommen werden der Reihe nach 0–2, 0–1, 1–3, 3–4, 4–5, 5–6 – dieselben Kanten wie bei Kruskal (weil alle Gewichte verschieden sind, ist der minimale Spannbaum eindeutig), aber in anderer Reihenfolge: Prim wächst von 0 aus, Kruskal nimmt die Kanten global nach Gewicht.

Aufgabe 6

Start

x	0	1	2	3
0	0	3	∞	7
1	3	0	1	∞
2	∞	1	0	2
3	7	∞	2	0

Ende

x	0	1	2	3
0	0	3	4	6
1	3	0	1	3
2	4	1	0	2
3	6	3	2	0

$k = 0$: $d[1][3] = 10$, $d[3][1] = 10$
$k = 1$: $d[0][2] = 4$, $d[2][0] = 4$
$k = 2$: $d[0][3] = 6$, $d[1][3] = 3$, $d[3][0] = 6$, $d[3][1] = 3$
$k = 3$: keine Änderung

Aufgabe 7

Eingangsgrade:

Knoten	A	B	C	D	E	F
EG	0	1	1	2	1	2

(nur A hat Eingangsgrad 0)

bearbeitet	A	B	C	D	E	F	neu mit EG 0	Liste
Start	0	1	1	2	1	2		A
A	0	**0**	**0**	2	1	2	B, C	A – B – C
B	0	0	0	**1**	1	2	–	A – B – C
C	0	0	0	**0**	**0**	2	D, E	A – B – C – D – E
D	0	0	0	0	0	**1**	–	A – B – C – D – E
E	0	0	0	0	0	**0**	F	A – B – C – D – E – F

Eine zulässige Reihenfolge: A – B – C – D – E – F. Auch A – C – E – B – D – F ist zulässig; topologische Sortierungen sind im Allgemeinen nicht eindeutig.

Aufgabe 8

Antrag von	an	Ergebnis
1	A	A ist frei: Verlobung 1–A
2	A	A zieht 2 dem bisherigen Partner 1 vor: 1 wird frei
3	B	B ist frei: Verlobung 3–B
1	B	B zieht 1 dem bisherigen Partner 3 vor: 3 wird frei
3	A	A bleibt bei 2, 3 versucht es bei der nächsten Frau
3	C	C ist frei: Verlobung 3–C

Ergebnis: 1–B, 2–A, 3–C. Stabil: Mann 1 hätte lieber A, doch A zieht ihren Partner 2 vor; Mann 3 hätte lieber B oder A, doch beide ziehen ihre Partner (1 bzw. 2) vor. Mann 2 hat seine Erstwahl. Es gibt also kein Paar, das sich gegenseitig den Partnern vorzieht.

Aufgabe 9

(a) Prim wächst nur über Kanten, die vom bisherigen Baum ausgehen; Knoten einer anderen Zusammenhangskomponente werden nie erreicht – es entsteht nur ein Spannbaum der Komponente des Startknotens. Kruskal prüft alle Kanten und liefert einen minimalen *Spannwald* (einen minimalen Spannbaum je Komponente).
(b) Bisher war 0 – 2 – 5 – 6 mit der Länge 21 am kürzesten; mit dem neuen Gewicht hat dieser Weg die Länge 26. Nun sind 0 – 1 – 3 – 4 – 6 und 0 – 1 – 3 – 4 – 5 – 6 mit jeweils der Länge 22 die kürzesten Wege – es gibt also zwei gleich lange (Dijkstra liefert je nach Reihenfolge einen davon). Der minimale Spannbaum bleibt gleich, weil die Kante 2–5 nicht zu ihm gehört: Wird eine Kante *außerhalb* des Spannbaums schwerer, bleibt jeder andere Spannbaum mindestens so schwer wie vorher, der bisherige behält sein Gewicht.