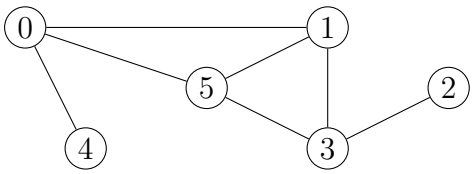
Ungerichtete Graphen

Ein *ungerichteter Graph* G ist ein Paar $G = (V, E)$, bestehend aus einer Knotenmenge V (engl. *vertices*) und einer Kantenmenge E (engl. *edges*). Graphen lassen sich bildlich darstellen, indem man die Knoten als kleine Kreise zeichnet und für jede Kante die entsprechenden Knoten verbindet.



Anwendungen

In der Informatik werden Graphen oft dazu verwendet, um reale Objekte und ihre gegenseitigen Beziehungen zu modellieren.

- Internet mit Webseiten (Knoten) und Links (Kanten)
- Verkehrsnetze mit Kreuzungen/Bahnhöfen (Knoten) und Strassen/Gleisen (Kanten)
- Produktionsabläufe bei denen eine Aufgabe (Knoten) erst dann erledigt werden kann, nachdem eine oder mehrere andere Aufgaben erfolgreich beendet wurden (Kanten)
- usw.

Datenstrukturen für Graphen

Wir werden in diesem Dokument zwei Datenstrukturen für Graphen vorstellen, wobei wir den Schwerpunkt auf die Adjazenzlisten legen.

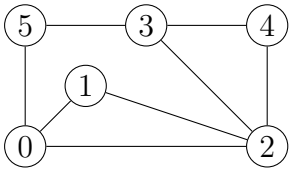
- Adjazenzmatrix
- Adjazenzlisten

Adjazent bedeutet *benachbart* und bezieht sich auf die Eigenschaft zweier Knoten u und v, dass sie durch eine Kante $\{u, v\}$ verbunden sind.

Während bei gerichteten Graphen eine Kante (u, v) nur von u nach v existiert, bedeutet in ungerichteten Graphen die Mengenschreibweise $\{u, v\}$, dass sowohl (u, v) als auch (v, u) Kanten des Graphen sind.

Adjazenzmatrix

Gibt es zwischen den Knoten u und v eine Kante, so wird in einer rechteckigen Tabelle (Matrix) im Schnittpunkt aus der u-ten Zeile und der v-ten Kolonnen eine Eins geschrieben. Gibt es keine Kante von u nach v, tragen wir stattdessen eine Null ein.

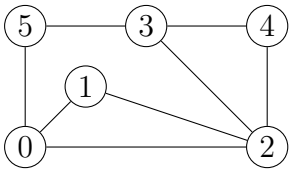


	0	1	2	3	4	5
0	0	1	1	0	0	1
1	1	0	1	0	0	0
2	1	1	0	1	1	0
3	0	0	1	1	1	0
4	0	0	1	1	1	0
5	1	0	1	0	0	0

Da in ungerichteten Graphen die Richtung einer Kante keine Rolle spielt, befindet sich mit der Kante von u nach v auch automatisch die Kante von v nach u in der Adjazenzmatrix. Die Matrix ist somit *symmetrisch*.

Adjazenzlisten

Diese Datenstruktur besteht aus einer assoziativen Liste, die jedem Knoten (*key*) eine Liste seiner Nachbarknoten (*value*) zuordnet. Sind die Knoten u und v adjazent, so wird u in die Adjazenzliste von v und v in die Adjazenzliste von u eingetragen.



von (*key*)	nach (*value*)
0	1, 2, 5
1	0, 2
2	0, 1, 3, 4
3	2, 4, 5
4	2, 3
5	0, 3

Aufgabe 1

Vergleiche den Aufwand der Darstellungen für einen Graphen $G = (V, E)$ in Bezug auf

(a) die Speicherung von G,

(b) das Hinzufügen einer Kante $\{v, w\}$,

(c) das Testen, ob Knoten w Nachbar von Knoten v ist,

(d) die Iteration über alle Nachbarknoten von v.

	(a)	(b)	(c)	(d)
Adjazenzmatrix	$\lvert V\rvert^2$	1	$\lvert V\rvert$	$\lvert V\rvert$
Adjazenzlisten	$\lvert E\rvert + \lvert V\rvert$	1	$\deg(v)$	$\deg(v)$

Designentscheid

Zur Darstellung von Graphen wählen wir Adjazenzlisten. Dieser Entscheid erfolgt unter der Voraussetzung, dass die von uns verwendeten Graphen *dünn besetzt* (engl. *sparse*) sind; das heisst, dass die Anzahl ihrer Kanten viel kleiner als die in einfachen Graphen maximal mögliche Kantenzahl $|V|(|V|-1)/2$ ist. Bei dichten Graphen kann es jedoch sinnvoll sein, eine Matrixdarstellung des Graphen in Betracht zu ziehen.

Python-Klasse für ungerichtete Graphen

```python
class Graph:

    def __init__(self):
        self.adj = dict() # leeres Dictionary

    def add_node(self, u):
        if u not in self.adj:
            self.adj[u] = []

    def add_edge(self, u, v):
        # falls nötig, füge Knoten hinzu
        if u not in self.adj:
            self.add_node(u)
        if v not in self.adj:
            self.add_node(v)
        # füge (symmetrisch) Kanten ein
        self.adj[u].append(v)
        self.adj[v].append(u)

    def __str__(self):
        '''Textdarstellung des Graphen'''
        txt = ''
        for u in sorted(self.adj):
            txt += '{0} -> '.format(u)
            txt += ' '.join(str(v) for v in self.adj[u])
            txt += '\n'
        return txt
```

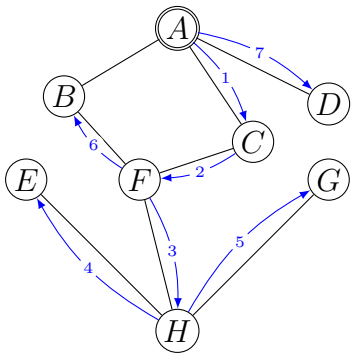
Tiefensuche

Die Tiefensuche *Depth-First-Search* ist eine Methode, um einen Graphen zu *traversieren*, d. h. von einem Startknoten aus jeden erreichbaren Knoten zu besuchen.

```
Hilfsfunktion traverse(graph, v):
    markiere v als besucht
    verarbeite Knoten v /* z.B. print(v) */
    für jeden Nachbarn w von v:
        wenn w noch nicht besucht wurde:
            traverse(w)

Funktion dfs(graph, v_start):
    Markiere alle Knoten von graph als unbesucht
    traverse(graph, v_start)
```

Beispiel



	von	nach
1	A	$\cancel{C}$ $\cancel{D}$ $\cancel{B}$
7	B	$\cancel{F}$ $\cancel{A}$
2	C	$\cancel{A}$ $\cancel{F}$
8	D	$\cancel{A}$
5	E	$\cancel{H}$
3	F	$\cancel{C}$ $\cancel{H}$ $\cancel{B}$
6	G	$\cancel{H}$
4	H	$\cancel{F}$ $\cancel{E}$ $\cancel{G}$

A, C, F, H, E, G, B, D

Python-Code

```python
def dfs(graph, start):
    '''Tiefensuche in graph, beginend in start'''

    visited = {v: False for v in graph.adj}

    def traverse(graph, u):
        '''Hilfsfunktion für die Rekursion'''
        visited[u] = True
        print(u)
        for v in graph.adj[u]:
            if visited[v] == False:
                traverse(graph, v)

    traverse(graph, start)
```