

Graphen – Übersicht, Breitensuche (BFS) und Tiefensuche (DFS)

1 Grundidee: Was ist ein Graph?

Ein **Graph** ist eine Datenstruktur zur Darstellung von **Objekten (Knoten / Vertices)** und deren **Beziehungen (Kanten / Edges)**.

Formal:

- $G = (V, E)$
- V = Menge der Knoten
- E = Menge der Kanten

Graphen sind sehr allgemein und werden z. B. verwendet für:

- Netzwerke (Computer, Soziale Netzwerke)
 - Routenplanung
 - Abhängigkeiten (Build-Systeme, Module)
 - Zustandsräume
-

2 Arten von Graphen (Übersicht)

Ungerichteter Graph

- Kanten haben **keine Richtung**
- $(u, v) = (v, u)$

Gerichteter Graph (Digraph)

- Kanten haben eine **Richtung**
- $(u \rightarrow v) \neq (v \rightarrow u)$

Gewichteter Graph

- Kanten besitzen ein **Gewicht** (Kosten, Distanz)

Ungewichteter Graph

- Alle Kanten gleichwertig

Zyklischer Graph

- Enthält **Zyklen**

Azyklischer Graph

- Enthält **keine Zyklen**
 - Spezialfall: **DAG** (Directed Acyclic Graph)
-

3 Graph-Darstellungen

Adjazenzliste

- Für jeden Knoten: Liste der Nachbarn
- Speicher: $O(V + E)$
- Standarddarstellung in der Praxis

Adjazenzmatrix

- $V \times V$ Matrix
 - Speicher: $O(V^2)$
 - Vorteilhaft bei dichten Graphen
-

4 Laufzeiten-Grundlagen

Für Traversierungen gilt:

- V = Anzahl Knoten
- E = Anzahl Kanten

→ **BFS und DFS:** $O(V + E)$

5 Breitensuche (BFS) im Graphen

Grundidee

Die **Breitensuche (Breadth-First Search)** besucht Knoten **schichtweise**:

- zuerst alle Nachbarn
- dann deren Nachbarn

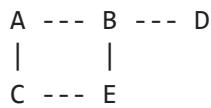
→ BFS nutzt eine **Queue (FIFO)**.

Eigenschaften von BFS

Eigenschaft	Wert
Laufzeit	$O(V + E)$
Speicher	$O(V)$
Kürzester Weg	ja (ungewichtet)
Rekursiv	nein

Schritt-für-Schritt-Beispiel

Graph (ungerichtet):



Start bei A:

1. A besuchen → Queue: [A]
2. Nachbarn B, C → Queue: [B, C]
3. Nachbarn von B → D, E → Queue: [C, D, E]
4. C, D, E besuchen

Besuchsreihenfolge:

A, B, C, D, E

6 Tiefensuche (DFS) im Graphen

Grundidee

Die **Tiefensuche (Depth-First Search)** geht **so tief wie möglich** entlang eines Pfades, bevor sie zurückgeht.

→ DFS nutzt:

- **Rekursion** oder
- einen **Stack (LIFO)**

Eigenschaften von DFS

Eigenschaft	Wert
Laufzeit	$O(V + E)$
Speicher	$O(V)$
Kürzester Weg	nein
Rekursiv	ja (typisch)

Schritt-für-Schritt-Beispiel

Gleicher Graph, Start bei A:

1. A → B → D
2. Backtracking → E
3. Backtracking → C

Eine mögliche Reihenfolge:

A, B, D, E, C

(DFS ist **nicht eindeutig** – Reihenfolge hängt von Nachbarn ab.)

7 Besonderheiten / Prüfungsrelevante Hinweise

BFS

- findet kürzeste Wege in **ungewichteten Graphen**
- verwendet Queue
- Levelweise Traversierung

DFS

- geeignet für:
 - Zyklus-Erkennung
 - Zusammenhangskomponenten
 - Topologische Sortierung
 - verwendet Rekursion / Stack
-

8 Vergleich BFS vs. DFS

Merkmal	BFS	DFS
Datenstruktur	Queue	Stack / Rekursion
Strategie	breit	tief
Kürzester Weg	ja	nein
Typische Anwendung	Distanzen	Struktur, Zyklen

9 Mögliche Algorithmen und deren Ziele

- Kürzeste Wege:
 - BFS (ungewichtete Graphen)
 - Dijkstra (gewichtete Graphen ohne negative Kanten)
 - Warshall (alle Paare)
 - Ziel: kürzeste Distanz
- Spannbäume:
 - Prim
 - Kruskal
 - Ziel: minimale Gesamtkosten
- Fluss- und Netzwerkalgorithmen:
 - Ford-Fulkerson
 - Edmonds-Karp
 - Ziel: maximaler Fluss (Durchsatz von Quelle zu Senke)



Merksätze für die Prüfung

- *BFS besucht Graphen schichtweise und findet kürzeste Wege in ungewichteten Graphen.*
- *DFS geht tief in den Graphen und eignet sich für Analyse von Strukturen und Zyklen.*



9 Python-Implementierung

```
In [1]: from collections import deque

# Graph als Adjazenzliste
graph = {
    "A": ["B", "C"],
    "B": ["A", "D", "E"],
    "C": ["A", "E"],
    "D": ["B"],
    "E": ["B", "C"]
}

def bfs(graph, start):
```

```

visited = set()
queue = deque([start])
order = []

while queue:
    node = queue.popleft()
    if node not in visited:
        visited.add(node)
        order.append(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                queue.append(neighbor)

return order

def dfs(graph, start, visited=None, order=None):
    if visited is None:
        visited = set()
        order = []

    visited.add(start)
    order.append(start)

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited, order)

    return order

# Beispiel
print("BFS:", bfs(graph, "A"))
print("DFS:", dfs(graph, "A"))

```

BFS: ['A', 'B', 'C', 'D', 'E']
DFS: ['A', 'B', 'D', 'E', 'C']