

Lernzettel

Graphen: Repräsentationen, Traversierung und Pfadsuche

Universität: Technische Universität Berlin
Kurs/Modul: Einführung in die Informatik - Vertiefung
Erstellungsdatum: September 20, 2025



Zielorientierte Lerninhalte, kostenlos!
Entdecke zugeschnittene Materialien für deine Kurse:

<https://study.AllWeCanLearn.com>

Einführung in die Informatik - Vertiefung

Lernzettel: Graphen: Repräsentationen, Traversierung und Pfadsuche

(1) **Graph-Repräsentationen.** Klar definiert ist ein Graph als

$$G = (V, E), \quad V = \{v_1, \dots, v_n\}, \quad E \subseteq V \times V.$$

Dabei gilt:

ungerichtet : (u, v) gleichbedeutend mit (v, u) , gerichtet : (u, v) besitzt *Richtungsinformation*.

Repräsentationen

- *Adjazenzliste*: Für jedes $(u, v) \in E$ ist v in der Liste von u enthalten.

$$N(u) = \{v \mid (u, v) \in E\}.$$

Speicherbedarf: $\mathcal{O}(V + E)$. Vorteil: Iteration über Nachbarn ist schnell. Nachteil: indirekter Zugriff auf existierende Kanten langsamer.

- *Adjazenzmatrix*: Eine Matrix A mit $A_{uv} = w(u, v)$ bzw. 0/1, je nach Gewichtung. Speicherbedarf: $\mathcal{O}(V^2)$. Vorteil: Kantenzugriff in $\mathcal{O}(1)$; Nachteil: bei großen, spärlich besetzten Graphen Speicherverbrauch hoch.
- *Kantenliste* (Edge List): Liste aller Kanten (u, v) (mit ggf. Gewicht $w(u, v)$). Speicherbedarf: $\mathcal{O}(E)$. Vorteil: einfach; Nachteil: Nachbarschafts-Abfragen teuer.

Graph-Typen

- *ungerichtet* vs. *gerichtet*
- *gewichtete* Graphen vs. *ungewichtete*

Zusätzliche Größen

$$\deg^+(v) = |\{u \mid (v, u) \in E\}|, \quad \deg^-(v) = |\{u \mid (u, v) \in E\}| \quad (\text{auswärts/eingangs})$$

Beispiele Ein unveränderter Graph G mit drei Knoten und zwei Kanten könnte so aussehen:

$$V = \{a, b, c\}, \quad E = \{(a, b), (b, c)\}.$$

(2) Traversierung (Traversierung von Graphen).

Tiefensuche (DFS)

Ziel: eine Tiefen-first Erkundung der Knoten. Grundidee: Nachbarn rekursiv bzw. mit Stack besuchen, sobald ein Knoten entdeckt wird. Eigenschaften:

- erzeugt einen DFS-Baum,
- Laufzeit $\mathcal{O}(V + E)$,

- benötigt Speicher $\mathcal{O}(V)$ für Stack/Visited.

Breitensuche (BFS)

Ziel: Erkundung in Schritten der Minimaldistanz (Level-Order). Grundidee: Nutze eine Queue; Startknoten s , Distanz zu s ist 0. Eigenschaften:

- liefert Abstände in ungewichteten Graphen,
- Laufzeit $\mathcal{O}(V + E)$,
- Speicherbedarf $\mathcal{O}(V)$ für die Queue.

Pfadrekonstruktion Durch Parent-Pointer während DFS/BFS speichern, dann Pfad von Zielknoten t nach Startknoten s rekonstruieren.

(3) Pfadsuche (kürzeste Wege).

Kürzeste Pfade in ungewichteten Graphen (BFS-Variante)

- Startknoten s setzen; Distanz $\text{dist}(s) = 0$; alle anderen $\text{dist} = \infty$.
- Queue initialisieren mit s .
- Während Queue nicht leer: entferne u ; für alle Nachbarn v von u :

falls $\text{dist}(v) = \infty$ dann $\text{dist}(v) = \text{dist}(u) + 1$, $\text{parent}(v) = u$, enqueue v .

Pfad zu einem Zielknoten t rekonstruiert man über die Parent-Pointer. Laufzeit: $\mathcal{O}(V + E)$.

Kürzeste Pfade in gewichteten Graphen (Dijkstra-Algorithmus)

- Initialisierung: $\text{dist}(s) = 0$, $\text{dist}(v) = \infty$ für alle $v \neq s$; Parent-Vektor leer.
- Verwende eine Prioritätswarteschlange (PQ) nach $\text{dist}(\cdot)$.
- Solange PQ nicht leer: wähle u mit kleinstem $\text{dist}(u)$; für jeden Nachbarn v mit Gewicht $w(u, v)$:

falls $\text{dist}(u) + w(u, v) < \text{dist}(v)$ dann $\text{dist}(v) = \text{dist}(u) + w(u, v)$, $\text{parent}(v) = u$, PQaktualisieren.

Laufzeit typischer Implementierung: $\mathcal{O}((V + E) \log V)$. Hinweis: Diese Methode setzt nichtnegative Gewichte voraus; negative Gewichte erfordern andere Verfahren.

Negative Gewichte

- *Bellman-Ford* kann Graphen mit negativen Gewichten verarbeiten.
- Laufzeit: $\mathcal{O}(V \cdot E)$.
- Geprüft wird auf negative Zyklen.

Alternative/Erweiterungen

- *A*-Algorithmus* nutzt eine Heuristik $h(n)$, um Suchraum zu fokussieren.
- Anwendung: Pfade in Karten, Spiele, Netzwerke.

Pfadrekonstruktion Wie bei BFS/Dijkstra lässt sich der Pfad von s nach t über die Parent-Pointer rekonstruieren.

(4) Zusammenfassung und Ausblick.

- Graph-Repräsentationen wählen je nach Anwendung: speicherbewusst (Liste) vs. schneller Kantenzugriff (Matrix).
- Traversierung liefert grundlegende Strukturen (Suchbaum vs. Level-Order).
- Pfadsuche unterscheidet ungewichtete vs. gewichtete Graphen; passende Algorithmen verwenden.
- Negative Gewichte erfordern spezielle Algorithmen (Bellman-Ford) statt Dijkstra.

Hinweis zur Umsetzung in Java:

- Adjazenzliste: `Map<V, List<Edge<V>>>` oder `List<List<Edge>>`.
- Adjazenzmatrix: zweidimensionale Arrays `int[][]` oder `double[][]`.
- Edge-Liste: Liste von `Edge`-Objekten (u, v, w) .