Lernzettel

Graphalgorithmen Grundlegende Traversierung: DFS und BFS und deren Anwendungen

Universität:	Technische Universität Berlin
Kurs/Modul:	Algorithmen und Datenstrukturen
Erstellungsdatum:	September 6, 2025

Lernzettel: Graphalgorithmen Grundlegende Traversierung: DFS und BFS und deren Anwendungen

(1) Grundlegende Begriffe
G = (V,E) sei ein Graph mit V Knoten und E Kanten.
- Grapharten: ungerichtet oder gerichtet; gewichtete oder ungewichtete Kanten.
- Adjazenzdarstellung: Adjazenzliste oder Adjazenzmatrix.
- Traversierung bedeutet, alle Knoten des Graphen genau einmal zu besuchen, wobei eine Reihenfolge festgelegt wird.

(2) Tiefensuche (DFS)
Die Tiefensuche erkundet so lange wie möglich einen Pfad, bevor sie zurückkehrt.
- Ziel: Besuch aller Knoten, Erzeugung eines Suchbaums (DFS-Baum).
- BAU: Rekursive oder iterative Implementierung mit einem Stack.

Rekursive Implementierung (DFS-Visit).
DFS-Visit(v): markiere v als besucht;
für jeden Nachbarn w von v in Adj(v) tue
falls w noch nicht besucht ist, rufe DFS-Visit(w) auf.

Iterative Implementierung (DFS mit Stack).
Stack S initial mit Startknoten s;
markiere s als besucht;
while S nicht leer:
v = Top(S); pop(S);
für jeden Nachbarn w von v in Adj(v) tue
falls w noch nicht besucht ist, markiere w als besucht und push(w) auf S.

Eigenschaften von DFS.
- Zeitkomplexität: $O(|V| + |E|)$.
- Raumkomplexität: $O(|V|)$ durch den Rekursions- bzw. Stack-Speicher.
- DFS-Baum: Jede Kante wird entweder als Baumkante oder als Rückkante klassifiziert.

(3) Breitensuche (BFS)
Die Breitensuche erkundet Knoten schichtweise nach der Distanz vom Startknoten.
- Ziel: kürzeste Pfade in ungewichteten Graphen, Nachbarschaften Ebene für Ebene.
- BAU: Queue-basiert.

BFS-From(s).
Q := leere Warteschlange;
markiere s als besucht; enqueu s;
while Q nicht leer:
v = dequeue(Q);
für jeden Nachbarn w von v in Adj(v) tue
falls w noch nicht besucht ist, markiere w als besucht und en-queue w.

Eigenschaften von BFS.
- Zeitkomplexität: $O(|V| + |E|)$.
- Raumkomplexität: $O(|V|)$ für die Besuchsmarkierungen und die Queue.
- BFS-Baum: Jede entdeckte Kante wird zur Baumkante, Distanz zu Startknoten wird festgelegt.
- Kürzeste Pfade in ungewichteten Graphen: Die Distanz $d(v)$ entspricht der Anzahl der Kanten im kürzesten Pfad von Startknoten s nach v.

(4) Anwendungen der DFS- und BFS-Traversierung
- Bestimmen zusammenhängender Komponenten (in einem ungerichteten Graphen): Starte DFS/BFS von jedem unbesuchten Knoten; jede entdeckte Komponente entspricht einer zusammenhängenden Teilmenge.
- Zyklenerkennung (ungerichtete Graphen):
- DFS: Bei einer Nachbarschaft w von v, die bereits besucht ist und $w \neq$ Vorgänger von v, existiert ein Zyklus.
- BFS: Ein gefundenes bereits besuchtes Nachbarelement weist ebenfalls auf einen Zyklus hin, je nach Implementierung.
- Kürzeste Pfade in ungewichteten Graphen: BFS liefert Längen der kürzesten Pfade von einem Startknoten zu allen erreichbaren Knoten.
- Topologische Sortierung (DAGs):
- DFS-basierte Topologische Ordnung: Knoten werden in der Reihenfolge fertiggestellt (Finish-Time-Stack) gelesen, die umgekehrt ergibt die Topologie.
- Alternative: Kahn-Algorithmus (BFS-ähnlich) für DAGs mit Indegrée-Counts.
- Spanning Tree (Traversierungsbäume): DFS- bzw. BFS-Baum dient als Spanning Tree des Graphen; Baumkanten definieren den Baum.
- Praxis-Hinweise zur Pfad- und Baumerzeugung: Während DFS/BFS Parent-Array speichern, um Pfade oder Baumstrukturen rekonstruieren zu können.

(5) Komplexität und praktische Hinweise
- Zeitkomplexität beider Traversierungen: $O(|V| + |E|)$.
- Raumkomplexität: $O(|V|)$ für Besuchsmarker und Struktur (Stack/Queue).
- Graphdarstellung: Adjazenzliste bevorzugt für skalierbare Graphen (schlechtere Fallzeit bei Adjazenzmatrix für dünn besetzte Graphen).
- Unterscheidung von Anwendungen:
- DFS eignet sich gut, um Zyklenstrukturen und Pfade in mehreren Richtungen zu explorieren;
- BFS eignet sich gut, um minimale Distanz- oder Pfad-Infos in ungewichteten Graphen zu gewinnen.

(6) Beispiel-Überblick
Gegeben sei ein ungerichteter Graph G mit $V = \{a, b, c, d, e\}$ und $E = \{\{a, b\}, \{a, c\}, \{b, d\}, \{c, e\}, \{d, e\}\}$.
- Starte DFS von a: mögliche DFS-Reihenfolge könnte a, b, d, e, c sein; der DFS-Baum ergibt Baumkanten $\{(a, b), (b, d), (d, e), (a, c)\}$.
- Starte BFS von a: mögliche BFS-Reihenfolge könnte a, b, c, d, e sein; der BFS-Baum ergibt Baumkanten $\{(a, b), (a, c), (b, d), (c, e)\}$.

(7) Hinweise zur Implementierung in Java/Allgemein
- Verwende eine Adjazenzliste als Map oder Array von Listen.

- Halte ein Array $visited[v]$ für DFS/BFS.
- Für DFS: entweder Rekursion oder eigener Stack; bei großen Graphen ist der Rekursionsstapel limitierend.
- Für BFS: nutze eine Queue (z. B. LinkedList oder ArrayDeque).
- Speichere ggf. ein Parent-Array, um Pfade oder den Traversierungsbaum zu rekonstruieren.