Kapitel 6

Suchen und Sortieren in Graphen

6.1 Breitensuche und Tiefensuche

Wenn wir einen Baum vollständig durchlaufen wollen, dann kann das unter anderem auf die folgenden beiden Arten bewerkstelligt werden: 1. Schicht für Schicht, indem wir also möglichst spät weiter absteigen und 2. indem wir möglichst früh weiter absteigen. Die erste Vorgehensweise wird auch als Breitensuche (engl. *breadth-first search (BFS)*) bezeichnet, die zweite als Tiefensuche (engl. *depth-first search (DFS)*), siehe Abbildung 6.1. Analoge Vorgehensweisen sind auch in beliebigen Graphen möglich.

Der Ansatz zur Breitensuche in einem beliebigen Graphen besteht darin, von einem Startknoten u ausgehend den Graphen in alle Richtungen zu durchlaufen wobei die Knoten Ebene für Ebene abgearbeitet werden. Alle neu entdeckten Knoten werden in einer Warteschlange zur späteren Bearbeitung eingereiht. Dabei führen wir eine Prozedur P auf allen gefundenen Knoten aus, siehe Algorithmus 24.

Für die folgende Laufzeitanalyse wollen wir davon ausgehen, dass der Eingabegraph duch eine Adjazenzliste gegeben ist und dass die Prozedur P nur konstante Laufzeit benötigt. Wie bereits in Abschnitt 5.4 festgestellt, benötigen HINZUFÜGEN und ENTFERNEN jeweils $\Theta(1)$ Zeit. Jeder Knoten der von u aus erreichbar ist wird genau ein Mal in die Warteschlange eingereiht und genau ein Mal aus ihr entnommen, also finden insgesamt $O(|V|)$ viele Warteschlangen-Operationen statt. Für jeden erreichten Knoten wird genau ein Mal die Liste seiner ausgehenden

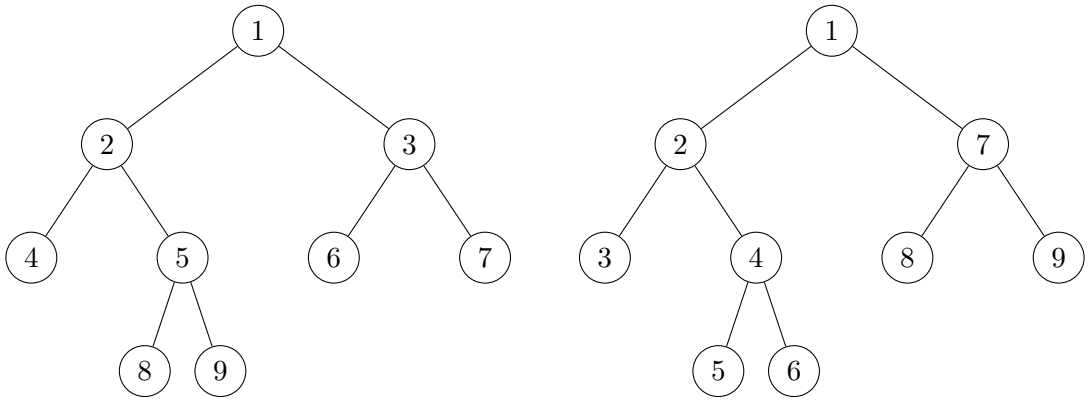

Abbildung 6.1: Breitensuche (links) und Tiefensuche (rechts) in einem Baum

Algorithmus 24 Breitensuche (engl. *breadth-first search*)

```
Prozedur BREITENSUCHE(V, E, u)
    Sei bekannt neues Datenfeld der Länge |V|, überall mit falsch initialisiert
    Sei Q eine neue Warteschlange, leer initialisiert
    bekannt[u] := wahr
    HINZUFÜGEN(Q, u)
    Solange Q nicht leer
        v := ENTFERNEN(Q)
        P(v)
        Für jede Kante (v, w) ∈ E
            Falls bekannt[w] = falsch dann
                bekannt[w] = wahr
                HINZUFÜGEN(Q, w)
            Ende Falls
        Ende Für
    Ende Solange
Ende Prozedur
```

Kanten durchlaufen, somit wird also die innerste Schleife $O(|E|)$ mal durchlaufen. Insgesamt erhalten wir also die Laufzeit $O(|V| + |E|)$. Dieser Algorithmus ruft die Prozedur P auf allen von u aus erreichbaren Knoten auf. Knoten die von u aus nicht erreicht werden können werden nicht besucht.

Die Tiefensuche kann nun, bis auf die Reihenfolge der Knoten, auf die gleiche Weise organisiert werden. Die für die Tiefensuche gewünschte Reihenfolge erhalten wir, indem wir die Warteschlange (FIFO) durch einen Stapel (LIFO) ersetzen. Wie vorhin führen wir auch jetzt eine Prozedur P auf genau den von u aus erreichbaren Knoten aus, siehe Algorithmus 25. Alles was

Algorithmus 25 Tiefensuche (engl. *depth-first search*)

```
Prozedur TIEFENSUCHE(V, E, u)
    Sei bekannt neues Datenfeld der Länge |V|, überall mit falsch initialisiert
    Sei S ein neuer Stapel, leer initialisiert
    bekannt[u] := wahr
    HINZUFÜGEN(S, u)
    Solange S nicht leer
        v := ENTFERNEN(S)
        P(v)
        Für jede Kante (v, w) ∈ E
            Falls bekannt[w] = falsch dann
                bekannt[w] = wahr
                HINZUFÜGEN(S, w)
            Ende Falls
        Ende Für
    Ende Solange
Ende Prozedur
```

wir vorhin über die Laufzeit gesagt haben gilt auch hier. Die Tiefensuche hat also ebenfalls Laufzeit $O(|V| + |E|)$.

Beispiel 6.1. siehe `bsp.breitensuche.tiefensuche.pdf`.

Breitensuche und Tiefensuche kann nun für eine bestimmte Prozedur P, wie in Algorithmen 24 und 25 angegeben, direkt verwendet werden. Größere Bedeutung als diese Prozeduren im Wortlaut hat aber das Designprinzip, einen Graphen entlang seiner Kanten zu durchlaufen wobei problemspezifische zusätzliche Information (wie z.B. das *bekannt*-Datenfeld) mitgeführt wird. In diesem Sinn bilden die Breitensuche und Tiefensuche eine wichtige Grundlage für viele ähnliche Algorithmen.

So kann zum Beispiel durch eine ganz einfache Modifikation etwa der Tiefensuche ein Algorithmus zur Detektion von Zyklen angegeben werden: Man kann sich leicht überlegen dass ein zusammenhängender (ungerichteter) Graph G genau dann einen Zyklus enthält wenn die Breitensuche oder Tiefensuche einen Knoten erreicht, der bereits bekannt ist. Einen Algorithmus zur Detektion eines Zyklus in einem Graphen erhält man also durch Ersetzung des Körpers der innersten Schleife durch:

Falls *bekannt*(w) = **wahr dann**
    Antworte "Graph enthält Zyklus!"
sonst
    bekannt(v) = **wahr**
    HINZUFÜGEN(S, w)
Ende Falls

6.2 Topologisches Sortieren

Ein weiteres Beispiel für ein Problem das durch geeignetes Durchlaufen eines Graphen gelöst werden kann ist die Erstellung einer Linearisierung eines gerichteten Graphen (in der Literatur oft auch als topologisches Sortieren bezeichnet).

Definition 6.1. Sei $G = (V, E)$ ein gerichteter endlicher Graph und $n = |V|$. Eine *Linearisierung* von G ist eine endliche Folge $v_1, \ldots, v_n$ von Knoten so dass $i \neq j \Rightarrow v_i \neq v_j$ und $(v_i, v_j) \in E \Rightarrow i < j$. Ein *gerichteter Zyklus* in G ist ein gerichteter Pfad $v_1, \ldots, v_k$ mit $k \geq 2$ und $(v_k, v_1) \in E$.

Beispiel 6.2. Wenn wir in einem Binärbaum als gerichteten Graphen auffassen, indem die Kanten von oben nach unten orientiert werden, dann ist die Datenfeld-Darstellung einer Halde eine topologische Sortierung der Baumdarstellung der Halde.

Satz 6.1. *Sei $G = (V, E)$ ein endlicher gerichteter Graph. Dann hat G eine Linearisierung genau dann wenn G zyklenfrei ist.*

Beweis. Angenommen G hat eine Linearisierung $v_1, \ldots, v_n$ und einen Zyklus. Dann muss der Zyklus die Form $v_{i_1}, \ldots, v_{i_k}$ mit $(v_{i_k}, v_{i_1}) \in E$ haben. Damit gilt $i_1 < i_2 < \cdots < i_k < i_1$, Widerspruch. Jeder Graph der eine Linearisierung hat muss also zyklenfrei sein.

Bevor wir die Gegenrichtung beweisen, beobachten wir dass ein endlicher gerichteter zyklenfreier Graph einen Knoten v mit $\mathrm{d}^-(v) = 0$ enthält: dieser kann gefunden werden, indem wir mit einem beliebigen Knoten v_0 starten und eine beliebige eingehende Kante rückwärts gehen. Dieser Schritt wird solange wiederholt bis wir an einem Knoten ohne eingehende Kanten angelangt sind. Dieser Prozess terminiert, da der Graph endlich und zyklenfrei ist.

Nun zeigen wir mit Induktion nach $|V|$ dass ein zyklenfreier Graph $G = (V, E)$ eine Linearisierung hat. Die Induktionsbasis $|V| = 0$ ist trivial. Für den Induktionsschritt sei $v \in V$ mit $\mathrm{d}^-(v) = 0$. Sei $G' = G - v$, also der Graph der aus G entsteht wenn v und alle Kanten die v enthalten entfernt worden sind. Dann hat G' nach Induktionshypothese eine Linearisierung

$v_2, \ldots, v_n$. Nachdem es kein $u \in V \setminus \{v\}$ gibt mit $(u, v) \in E$, ist $v, v_2, \ldots, v_n$ eine Linearisierung von G. □

Wir wollen nun das folgende Berechnungsproblem betrachten:

Linearisierung (Topologisches Sortieren)
Eingabe: Ein endlicher zyklenfreier Graph $G = (V, E)$
Ausgabe: Eine Linearisierung von G

Als Anwendung kann man sich zum Beispiel Folgendes vorstellen: die Knoten repräsentieren die Teilaufgaben eines Projekts, von Aufgabe A nach Aufgabe B wird eine Kante gesetzt falls A vor B erledigt werden muss. Gesucht ist dann eine Reihenfolge in der diese Aufgaben unter Beachtung ihrer Abhängigkeiten abgearbeitet werden können.

Der Beweis von Satz 6.1 suggeriert ja bereits die folgende algorithmische Vorgehensweise:

Wiederhole $|V|$ mal:

bestimme einen Knoten v mit $\mathrm{d}^-(v) = 0$, gib v aus und entferne v aus dem Graphen.

Die Bestimmung eines Knoten v mit $\mathrm{d}^-(v) = 0$ kann, wie im Beweis von Satz 6.1, so gemacht werden, dass von einem beliebigen Knoten u ausgehend die Kanten zurück verfolgt werden bis ein solcher Knoten v gefunden werden wird. Wie im Beweis von Satz 6.1 garantiert die Zyklenfreiheit des Graphen dass dieser Algorithmus terminiert. Diese Methode zur Bestimmung eines $v \in V$ mit $\mathrm{d}^-(v) = 0$ benötigt Zeit $O(|V|)$. Da sie $|V|$ mal wiederholt wird ist der Zeitbedarf dieses Algorithmus nur mit $O(|V|^2)$ abschätzbar.

Für dünn besetzte Graphen ist das nicht besonders gut. Eine bessere Vorgehensweise besteht darin, inspiriert von Breitensuche und Tiefensuche, den Graphen entlang seiner Kanten zu durchlaufen, wobei als nächste Knoten nur solche mit Eingangsgrad 0 (bezüglich des noch nicht durchlaufenen Teils) in Frage kommen. Um diese Knoten zu kennen wird statt dem *bekannt*-Datenfeld ein Datenfeld *Grad* mitgeführt in dem der Eingangsgrad aller Knoten bezüglich des noch nicht durchlaufenen Teils abgelegt (und auf dem aktuellen Stand gehalten) wird, siehe Algorithmus 26. Analog zur Laufzeitabschätzung der Breitensuche und Tiefensuche kann man sich auch hier leicht überlegen, dass die Laufzeit dieses Algorithmus $\Theta(|V| + |E|)$ ist.

Algorithmus 26 Linearisieren (Topologisches Sortieren)

```
Prozedur LINEARISIEREN(V, E)
    Sei Grad ein neues Datenfeld der Länge |V|, überall mit 0 initialisiert
    Für alle (v, w) ∈ E
        Grad[w] := Grad[w] + 1
    Ende Für
    Sei M neue Warteschlange oder Stapel, leer initialisiert
    Für alle v ∈ V
        Falls Grad[v] = 0 dann
            HINZUFÜGEN(M, v)
        Ende Falls
    Ende Für
    Solange M nicht leer
        v := ENTFERNEN(M)
        AUSGABE(v)
        Für alle (v, w) ∈ E
            Grad[w] := Grad[w] − 1
            Falls Grad[w] = 0 dann
                HINZUFÜGEN(M, w)
            Ende Falls
        Ende Für
    Ende Solange
Ende Prozedur
```