6. Gewichtete Graphen

In diesem Kapitel behandeln wir gewichtete Graphen, also Graphen, deren Kanten positiv gewichtet sind. Im Einzelnen befassen wir uns mit der Berechnung minimaler aufspannender Bäume, dem Abstandsproblem und der Berechnung des maximalen Flusses in einem Netzwerk.

Im ersten Abschnitt studieren wir Algorithmen, die wir später anwenden. Es geht um Priority-Queues, den Union-Find-Datentyp, das LCA-Problem und effizientere Verfahren für das RMQ-Problem aus dem ersten Kapitel. Priority-Queues implementieren wir mit binären Heaps. Bei der Analyse des Union-Find-Datentyps wenden wir die Ackermann-Funktion an. Beim LCA-Problem geht es um die Bestimmung des letzten gemeinsamen Vorgängers von zwei Knoten in einem Wurzelbaum. Die Lösung dieses Problems in linearer Laufzeit ist eine Voraussetzung für einen Algorithmus zur Verifikation eines minimalen aufspannenden Baums in linearer Laufzeit.

Die Algorithmen von Borůvka, Kruskal und Prim konstruieren minimale aufspannende Bäume und der Algorithmus von Dijkstra löst das Abstandsproblem. Die Algorithmen von Prim und Dijkstra verallgemeinern die Breitensuche aus dem Kapitel 5. Bei der Implementierung ersetzt die Priority-Queue die Queue, die wir bei der Breitensuche einsetzen. Der Union-Find-Datentyp kommt in Kruskals Algorithmus zur Anwendung.

Der probabilistische Algorithmus von Karger, Klein und Tarjan berechnet in linearer Laufzeit einen minimalen aufspannenden Baum. Wesentlich dafür ist ein Algorithmus, der in linearer Laufzeit die Verifikation eines minimalen aufspannenden Baums durchführt.

Der Algorithmus von Warshall ermittelt den transitiven Abschluss eines Graphen und der Algorithmus von Floyd die Abstandsmatrix. Die Algorithmen von Ford-Fulkerson und Edmonds-Karp lösen das Flussproblem in Netzwerken. Zunächst präzisieren wir den Begriff des gewichteten Graphen.

Definition 6.1. Ein Graph $G = (V, E)$ mit einer Abbildung $g : E \longrightarrow \mathbb{R}_{>0}$ heißt *gewichteter Graph*. Die Abbildung g heißt *Gewichtsfunktion*. Für $e \in E$ heißt $g(e)$ das *Gewicht von* e. Das *Gewicht von* G ist die Summe der Gewichte aller Kanten, $g(G) = \sum_{e \in E} g(e)$.

Zur Darstellung von gewichteten Graphen verwenden wir – wie zur Darstellung von Graphen – die Datenstrukturen Adjazenzliste und Adjazenzmatrix. Wir müssen die Definitionen nur geringfügig erweitern.

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2019
H. Knebl, *Algorithmen und Datenstrukturen*, https://doi.org/10.1007/978-3-658-26512-0_6

1. Die Adjazenzmatrix adm ist eine $n \times n$–Matrix mit Koeffizienten aus $\mathbb{R}_{\geq 0}$,
$$adm[i,j] := \begin{cases} g(\{i,j\}), & \text{falls } \{i,j\} \in E, \\ 0 & \text{sonst.} \end{cases}$$
2. In den Listenelementen der Adjazenzliste speichern wir das Gewicht einer Kante. Wir erweitern deshalb das Listenelement für Graphen auf Seite 225 um die Komponente *weight*.

6.1 Grundlegende Algorithmen

Wir studieren in diesem Abschnitt grundlegende Algorithmen, die wir im restlichen Kapitel anwenden. Es handelt sich um Priority-Queues und um den Union-Find-Datentyp. Diese beiden Datentypen zählen zu den hervorgehobenen Datenstrukturen und sind in viele Situationen anwendbar.

Im Abschnitt 6.5 geht die Berechnung des letzten gemeinsamen Vorgängers (lowest common ancestor, kurz LCA) von zwei Knoten in einem Baum wesentlich ein. Wir behandeln das LCA-Problem, das linear äquivalent zum RMQ-Problem ist (Abschnitt 1.4.4), als eigenständiges Problem. Es geht um ein grundlegendes algorithmisches Problem, das intensiv studiert wurde.

6.1.1 Die Priority-Queue

Die Priority-Queue verallgemeinert den abstrakten Datentyp Queue. Bei einer Queue verlassen die Elemente in der gleichen Reihenfolge die Queue, in der sie gespeichert wurden (first in, first out – FIFO-Prinzip). Bei einer Priority-Queue weisen wir jedem Element beim Einspeichern eine Priorität zu. Unsere spätere Anwendung erfordert es, die Priorität eines gespeicherten Elementes zu erniedrigen. Das Element mit der geringsten Priorität verlässt als nächstes Element die Queue. Eine Priority-Queue ist durch folgende Funktionen definiert:

1. PQInit(int *size*) initialisiert eine Priority-Queue für *size* viele Elemente.
2. PQUpdate(element k, priority n) fügt k in die Priority-Queue mit der Priorität n ein. Befindet sich k bereits in der Priority-Queue und besitzt k höhere Priorität als n, dann erniedrigt PQUpdate die Priorität von k auf n. PQUpdate gibt true zurück, falls eine Einfügeoperation oder ein Update stattgefunden hat, ansonsten false.
3. PQRemove liefert das Element mit der geringsten Priorität und entfernt es aus der Priority-Queue.
4. PQEmpty überprüft die Priority-Queue auf Elemente.

Bei der Implementierung der Priority-Queue verwenden wir folgende Datentypen: type element = $1..n$, type index = $0..n$ und