7. Graphen

7.1. Motivation

Graphen sind ein mathematisches Modell, mit dem sich Algorithmen zur Lösung verschiedener Aufgaben formulieren lassen. Gelingt es, eine konkrete Aufgabe, wie das Heraussuchen optimaler Zugverbindungen aus einem Fahrplan, untereinander abhängige Arbeiten in einer sinnvollen Reihenfolge anzuordnen oder einen Garbage Collector für eine Software-Laufzeitumgebung zu planen, mit einem Graphen-Modell zu formulieren, kann man die dafür bekannten Algorithmen verwenden. Da Graphen eine einfache und gleichzeitig mächtige Struktur sind, lassen sich viele Aufgaben damit modellieren. Die wichtigsten davon sind Thema dieses Kapitels.

7.2. Lernziele

- Sie können Datenstrukturen zur Speicherung von Graphen konzeptuell beschreiben, Stärken und Schwächen angeben und in Bezug auf eine bestimmte Aufgabe auswählen.
- Sie können mindestens eine Datenstruktur zur Speicherung von Graphen in Java implementieren.
- Sie können Algorithmen auf Graphen zur *Topologischen Sortierung*, zum systematischen *Absuchen*¸ und zur *Bestimmung eines kürzesten Weges* konzeptuell beschreiben und in Java implementieren.

7.3. Graphen – Definitionen

Auf eine grundsätzliche Einführung von *Graphen* soll an dieser Stelle verzichtet werden. Zur Auffrischung bzw. zum Nachlesen grundlegender Definitionen und Eigenschaften, die aus der Mathematik bekannt sein sollten, steht ein separates Blatt zur Verfügung.

7.4. Datenstrukturen zum Speichern von Graphen

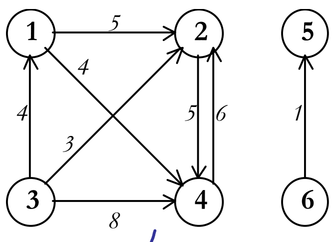
Es gibt verschiedene Arten, Graphen im Speicher eines Computers in Datenstrukturen zu speichern. Die Auswahl hängt von den Eigenschaften der konkreten Daten ab, wie der absoluten Anzahl von Knoten und Kanten sowie dem Verhältnis dieser Zahlen; weiter davon, ob oder wie häufig Knoten oder Kanten hinzugefügt oder entfernt werden sollen. Schliesslich spielt es auch noch eine wesentliche Rolle, welche Algorithmen auf der Datenstruktur ausgeführt werden sollen.

7.4.1. Adjazenzmatrix

Adjazenzmatrizen stellen Kanten zwischen Knoten als Wert in einer quadratischen Matrix dar. Deren Dimension wird durch die Anzahl der Knoten vorgegeben: Bei *n* Knoten ergibt sich eine *n*×*n*-Matrix.

Bei gewichteten Graphen werden als Werte die entsprechenden Kantengewichte eingesetzt, bei ungewichteten Graphen genügen Wahrheitswerte. Typische Deklarationen in Java wären also:

```
int[][] g = new int[N][N]; // gewichteter Graph
```
bzw.
```
boolean[][] g = new boolean[N][N]; // ungewichteter Graph
```



Der rechts gezeichnete Graph ist darunter in einer (zu ergänzenden) Adjazenzmatrix dargestellt. Die Matrix zeigt durch den Wert 5 in Zeile 1 und Spalte 2 an, dass eine Kante mit Gewicht 5 vom Knoten 1 zum Knoten 2 führt.

In der Gegenrichtung (Zeile 2, Spalte 1) gibt es keine Kante, was durch einen Strich angezeigt wird.

nach

von	1	2	3	4	5	6
1	-	5	-	4	-	-
2	-	-	–	5	–	–
3	4	3	–	8	–	–
4	–	6	–	–	–	–
5	–	–	–	–	–	–
6	–	–	–	–	1	–

Bei der Implementierung in Software stellt sich die Frage, wie die Tatsache, dass keine Kante existiert, repräsentiert werden soll. Da Matrizen mit Element-Typ *int* benutzt werden sollen, ist die von uns hier verwendete Darstellung "-" nicht möglich.

Die Lösung des Problems hängt davon ab, wie die Kantengewichte interpretiert werden sollen.

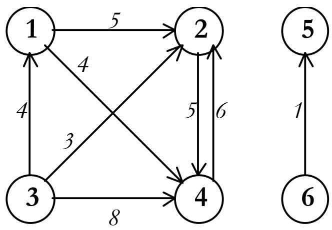
a) Handelt es sich um Distanz- oder Kostenangaben, so bedeutet eine nicht vorhandene Kante eine unendlich weite bzw. unendlich teure Verbindung. Daher sollte ein möglichst grosser Wert gewählt werden. In der Praxis: *Integer.MAX_VALUE*.

b) Handelt es sich um Kapazitätsangaben (z.B. wie viel Energie eine Stromleitung pro Zeiteinheit transportieren kann, wie viele Autos pro Stunde den Gotthardtunnel passieren können, oder wie viele Bit pro Sekunde eine Datennetzwerkverbindung übermitteln kann), so bedeutet eine nicht vorhandene Kante eine Verbindung mit der Kapazität 0. Daher wird dann der Wert *0* für die Darstellung einer nicht-bestehenden Kante verwendet.

Mit den beschriebenen Werten wird erreicht, dass Algorithmen keine Fallunterscheidungen machen müssen, sondern sich auf Grund des Wertes für sehr hohe Kosten bzw. sehr niedrige Kapazität „automatisch" so verhalten, als gäbe es die entsprechende Kante gar nicht.

Im Fall von ungewichteten Graphen stellt sich das oben diskutierte Problem natürlich gar nicht, denn es wird mit den Werten *true* und *false* angezeigt, ob eine Kante existiert.

Für ungerichtete Graphen ist die Adjazenzmatrix symmetrisch. Das heisst, es gilt $A = A^T$.[1]



7.4.2. Kantentabelle

Eine Kantentabelle ist eine Liste, mit je einem Eintrag pro Kante im Graphen. Jeder Eintrag enthält Start- und Zielknoten der Kante sowie bei gewichteten Graphen das Gewicht. Eine solche Liste kann man als Tabelle oder als verlinkte Liste speichern. Für letzteren Fall bietet sich als Typdeklaration eine *Linked List* von Objekten des Typs *Edge* an:

```
class Edge {
  int from, to, weight;
}
```

Nebenstehende (zu ergänzende) Kantentabelle zeigt denselben Graphen wie im Abschnitt 7.4.1. Die Kante von 1 nach 2 mit Gewicht 5 ist als Beispiel bereits eingetragen.

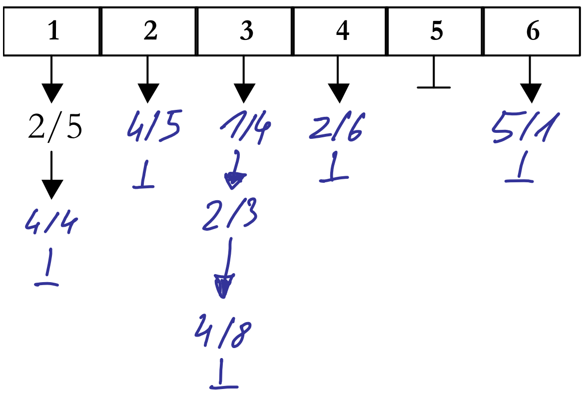
from	1	1	2	3	3	3	4	6
to	2	4	4	1	2	4	2	5
weight	5	4	5	4	3	8	6	1

Im Falle von ungerichteten Graphen kann man entweder jede Kante doppelt in die Tabelle eintragen (einmal für jede Richtung) oder die Werte von *from* und *to* jeweils gleich, das heisst gleichzeitig als Start- und Zielknoten behandeln.

7.4.3. Verbindungslisten (Adjazenzlisten)

Verbindungslisten beschreiben für jeden Knoten die von ihm wegführenden Kanten, bzw. die direkt erreichbaren Nachbarknoten in einer Liste. Es brauchen in diesem Fall nur noch die jeweiligen Zielknoten und ein allfälliges Gewicht gespeichert zu werden.

Nebenstehende Datenstruktur soll die für die 6 Knoten des Graphen aus Abschnitt 7.4.1 die Listen der jeweils wegführenden Kanten enthalten. Die Kante von 1 nach 2 mit Gewicht 5 ist als Beispiel eingetragen. Vom Knoten 5 führen keine Kanten weg.



Verbindungslisten für ungerichtete Graphen brauchen doppelte Einträge für jede Kante, denn diese müssen für jeden der beiden Endknoten registriert werden.

7.5. Topologisches Sortieren

Allgemein für Graphen formuliert lautet die Aufgabenstellung für den Algorithmus *Topologisches Sortieren*:

Gegeben ist ein *gerichteter Graph*.

Definition: Eine topologische Ordnung ist eine Reihenfolge von Knoten, so dass ein Knoten v_i vor dem Knoten v_j erscheint, falls es einen Pfad von v_i nach v_j gibt.

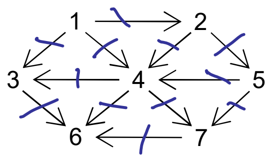
Bemerkung: Damit eine topologische Ordnung existiert, muss der Graph zyklenfrei sein.[2]

---

[1] A^T bezeichnet die *transponierte* Matrix von A, d.h. die Matrix, die sich durch Spiegelung von A an der Hauptdiagonalen ergibt.

[2] *Gerichtete azyklische Graphen*, engl. *Directed Acyclic Graph* (DAG) sind eine oft vorkommende Art von Graphen.

Topologische Ordnungen sind i.a. nicht eindeutig. Für nebenstehenden Graphen gibt es verschiedene Ordnungen:



1 2 5 4 3 7 6

und

1 2 5 4 7 3 6

Algorithmen zum Ermitteln einer solchen Ordnung, also zum topologischen Sortieren, verwenden den *Eingangsgrad* (*in degree*) der Knoten:

$$indeg(x) = |\{ y \mid \langle y, x\rangle \in E \}|$$

Adj.-List

die Anzahl Elemente der Menge aller Knoten *y*, von denen eine Kante zum Knoten *x* führt

Eine Folge topologisch sortierter Knoten kann schrittweise aufgebaut werden, indem immer ein (beliebiger) noch nicht darin eingeordneter Knoten mit Eingangsgrad 0 daran angehängt wird.

Es ergibt sich also folgender Algorithmus:

1. $O(n+m)$ Für alle Knoten den Eingangsgrad bestimmen
2. $O(n)$ Alle Knoten mit Eingangsgrad 0 in einem *Set* sammeln
3. $O(n+m)$ Solange das *Set* der Knoten mit Eingangsgrad 0 nicht leer ist,
    1. Einen Knoten *x* dem *Set* der Knoten mit Eingangsgrad 0 entnehmen und an die Folge anhängen
    2. Bei allen Knoten *y*, zu denen von *x* eine Kante führt, den Eingangsgrad um 1 reduzieren. Wird dabei der Eingangsgrad von *y* zu 0, *y* dem *Set* aller Knoten mit Eingangsgrad 0 hinzufügen.
4. $O(1)$ Sind alle Knoten des Graphen verarbeitet, ist dabei eine topologische Ordnung entstanden. Sind noch unverarbeitete Knoten übrig, hat aber keiner davon Eingangsgrad 0, ist der Graph zyklisch und es gibt keine topologische Ordnung dafür.

Aufwand (bei *n* Knoten und *m* Kanten):

- Der beschriebene Algorithmus muss zunächst alle Knoten besuchen, um alle mit Eingangsgrad 0 zu finden: $O(n)$
- Anschliessend wird jede Kante einmal verfolgt (um den Eingangsgrad ihres Zielknotens zu reduzieren): $O(m)$

Insgesamt sind für das topologische Sortieren also $O(n + m)$ Schritte notwendig.

7.6. Depth-First-Search (DFS)[3]

7.6.1. Graphen traversieren

Programme können nicht einfach Graphen „anschauen". Sie müssen sich über Knoten und Kanten „entlangtasten", vergleichbar mit der Suche in einem Labyrinth. Dieses „Entlangtasten" ist für spezielle Graphen – nämlich Bäume – bereits bekannt. Für allgemeine Graphen lässt sich die Idee der *pre-order Traversierung* geeignet anpassen.

Im Gegensatz zu Bäumen muss man bei allgemeinen Graphen mit folgender Situation umgehen:

- Man kann auf mehreren Wegen zu denselben Knoten gelangen
- Bei Zyklen will man nicht ewig im Kreis laufen

Dazu drängt sich folgende Lösung auf:

- Besuchte Knoten markieren visited = true;
- Markierte Knoten nicht nochmals besuchen if(!visited){...}

[3] auch als *Tiefensuche* bezeichnet

Diese Art der Suche strebt danach möglichst rasch die weit entfernten Knoten zu erreichen und wird deswegen als *Depth-First-Search* (DFS) oder *Tiefensuche* bezeichnet. Gegensatz dazu ist *Breadth-First-Search* oder *Breitensuche*, die danach strebt, parallel verschiedene Wege auszuprobieren (s. 7.7.2).

Wie die Traversierung von Bäumen lässt sich die Tiefensuche am besten als rekursives Verfahren formulieren (die unterstrichenen Zeilen sind hierbei neu gegenüber der Baum-Traversierung):

O(n)

1. Alle Knoten als bisher *nicht besucht* markieren
2. *dfs(startKnoten)* aufrufen

for (Vertex<K> v : vertices.values())
v.visited = false;

dfs(Vertex v):

falls *v* nicht als *besucht* markiert ist:

1. *v* „besuchen“ (verarbeiten, ausgeben, etc.)
2. *v* als *besucht* markieren
3. Für alle *Vertex w* zu denen eine Kante von *v* führt:
   dfs(w) aufrufen

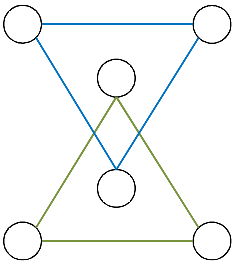
if (!v.visited) {
print(v);
v.visited = true;
for (Vertex w : v.adjList) dfs(w);
}

O(m) Aufrufe

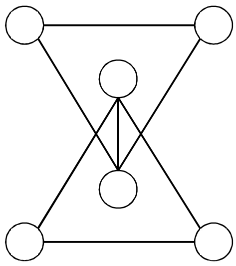
7.6.2. Anwendung: Zusammenhangskomponenten finden

Graphen-Traversierung eignet sich beispielsweise, um zu prüfen ob ein Graph zusammenhängend ist.

Definition: Ein ungerichteter Graph ist zusammenhängend, falls es für jedes Paar von verschiedenen Knoten einen verbindenden Pfad gibt.



2 Zusammenhangs-komponenten



zusammenhängend

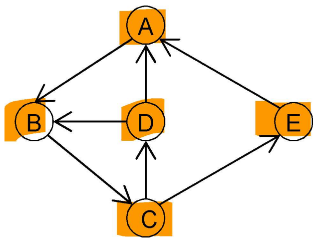
Um zu prüfen, ob ein Graph zusammenhängend ist, kann man eine Traversierung mit *DFS* ausgehend von einem beliebigen Knoten starten. Alle von diesem Knoten erreichbaren Knoten sind anschliessend als *besucht* markiert:

1. Alle Knoten als bisher *nicht besucht* markieren
2. *dfs(v)* für irgendeinen Knoten *v* aufrufen
3. Sequenzielle Suche in der Menge aller Knoten nach dem ersten, der *nicht besucht* wurde
   Wird *kein* solcher gefunden, ist der Graph zusammenhängend

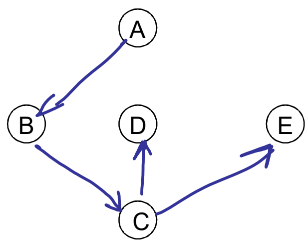
Findet die abschliessende sequenzielle Suche einen noch *nicht besuchten* Knoten, kann von diesem aus erneut ein *dfs* gestartet werden. Iteriert man diesen Prozess, ergibt die Anzahl der *dfs*-Aufrufe die nötig sind, bis alle Knoten besucht sind, die Anzahl der *Zusammenhangskomponenten* des Graphen.

7.6.3. Anwendung: Spannbaum

Speichert man alle Kanten, die DFS benutzt, um zu bisher *nicht besuchten* Knoten zu kommen, so erhält man einen Baum, der alle vom Startknoten aus erreichbaren Knoten des Graphen enthält, sowie die kleinstmögliche Menge an Kanten um diese zu verbinden.[4] Ein solcher Baum, heisst *Spanning Tree* oder *Spannbaum*.



dfs('A') führt zu folgendem Spannbaum:



Um während einer Tiefensuche den Spannbaum zu speichern, erweitert man am besten die *dfs*-Operation aus 7.6.1 folgendermassen:

dfs(Vertex v):

falls *v* nicht als *besucht* markiert ist:

1. *v* als *besucht* markieren
2. Für alle *Vertex w* zu denen eine Kante von *v* führt:
   Falls *w* nicht als *besucht* markiert ist:
   a. Kante <*v*, *w*> dem Graphen *tree* hinzufügen
   b. *dfs(w)* aufrufen

[4] Jeder Baum mit *n* Knoten enthält genau *n-1* Kanten – eine „hin-führende“ für jeden Knoten ausser der Wurzel

7.7. Kürzester Weg

7.7.1. Definitionen

Definition: Distanz zweier Knoten:

D(x, y) = Länge eines kürzesten Weges von *x* nach *y*, falls einer existiert; ∞ sonst

Definition: Länge eines Weges:

a) In ungewichteten Graphen: Anzahl Kanten

b) In gewichteten Graphen:

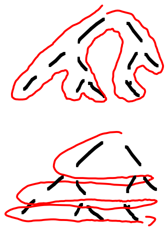
$$\sum w_{x,y} \quad \text{wobei} \quad w_{x,y} = \begin{cases} 0 & falls\ x = y \\ Gewicht\ von\ \langle x, y \rangle & falls\ \langle x, y \rangle\ \in E \\ \infty & sonst \end{cases}$$

7.7.2. Ungewichtete Graphen: Breadth-First-Search (BFS)[5]

Während *Tiefensuche* (DFS – s. 7.6) möglichst schnell die Tiefen eines Graphen zu ergründen sucht, geht *Breitensuche* oder *Breadth First Search* (BFS) schichtweise vor. Ausgehend von einem Startpunkt werden zuerst alle Knoten markiert, deren Distanz 1 ist, dann jene mit Distanz 2 usw.

Um dies Laufzeit-effizient tun zu können, verwendet man am besten eine Warteschlange (Queue), in der alle bereits *entdeckten* aber noch nicht *besuchten* Knoten gespeichert werden. Bearbeitet wird dann jeweils der erste Knoten der Liste.

Algorithmus im Pseudo-Code:



vs.

O(n) 1. Bei allen Knoten die Distanzangabe auf ∞ setzen

O(1) 2. Beim Startknoten die Distanzangabe auf 0 setzen

O(1) 3. Startknoten hinten an die Warteschlange *erreichbar* anfügen

O(n+m) 4. Solange Warteschlange *erreichbar* nicht leer ist:

  a. Vordersten Knoten *v* aus der Warteschlange entnehmen

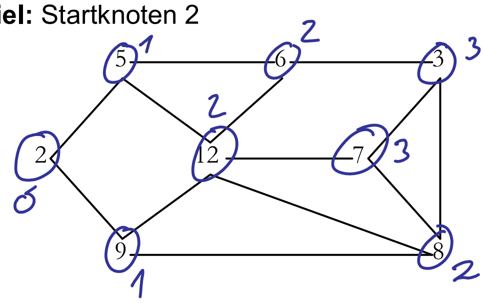
  b. Für alle Knoten *w* zu denen eine Kante von *v* führt:

   Falls die Distanzangabe dieses Knotens ∞ ist (war er bisher nicht erreichbar):

    Distanzangabe von *w* = 1 + Distanzangabe von *v*

    Knoten *w* hinten an Warteschlange *erreichbar* anfügen

Beispiel: Startknoten 2



"erreichbar": ~~2 5 9 6 12 8 3 7~~
Warteschlange

---

[5] auch als *Breitensuche* bezeichnet

7.7.3. Gewichtete Graphen: Algorithmus nach Dijkstra (Edsger W. Dijkstra, 1959)

Sind die Kanten gewichtet, muss man diese Kantengewichte als Distanzen bei der Suche berücksichtigen. Damit wird die allgemeine Breitensuche aus 7.7.2 etwas komplizierter.

1. Es wird eine Tabelle mit allen Knoten geführt. Pro Knoten werden darin 3 Dinge eingetragen:
   bekannt: (*false*) Ist die minimale Distanz zum Ausgangsknoten schon bekannt?
   dist: (*0* für Startknoten, ∞ sonst) Die bisher bekannte minimale Distanz zum Ausgangsknoten
   via: (? oder *null*) Der vorhergehende Knoten auf dem kürzesten Pfad vom Ausgangsknoten
   Die Werte in Klammern geben die Initialisierung der Tabelle an.
2. Solange sich in der Tabelle Knoten befinden mit *bekannt = false* und *dist* < ∞
   a. Den Knoten *v* aus der Tabelle suchen der unter allen mit *bekannt* = *false* den kleinsten Wert für *dist* hat.
   b. *v.bekannt* auf *true* setzen, denn es kann keinen noch kürzeren Weg zu *v* geben.
   c. Für jeden *Knoten w*, der von *v* aus direkt erreichbar ist (d.h. es gibt Kante <*v*, *w*>):

neu gegenüber BFS

      i. Distanz berechnen, die *w* vom Startknoten entfernt ist:
         int d = *v.dist* + Kantengewicht von <*v*, *w*>
      ii. Ist diese Distanz *d* kleiner als der Wert, der für *w.dist* bisher bekannt ist, so wurde ein kürzerer Weg zum Knoten *w* gefunden. Dieser führt über *v* nach *w*. Das wird in der Tabelle eingetragen.
         if (*d* < *w.dist*) { *w.dist* = *d*; *w.via* = *v*; }

Bemerkung:

Dieser Algorithmus findet die kürzesten Wege vom Startknoten zu allen erreichbaren Knoten gleichzeitig. Es ist kein Algorithmus bekannt, der mit weniger Aufwand nur den kürzesten Weg zwischen 2 Knoten findet.

Aufwandsanalyse:

Es sei *n* die Anzahl der Knoten und *m* die Anzahl der Kanten.

Der Aufwand für die einzelnen Schritte des Algorithmus ist:

Schritt 1: Tabelle für alle Knoten initialisieren → $O(n)$

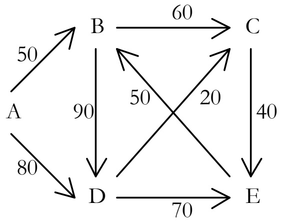
Schritt 2 a: n Knoten verarbeiten, jeweils n durchsuchen → $O(n^2)$

Schritt 2 c: Allen Kanten genau einmal folgen → $O(m)$

Insgesamt also $O(n^2 + m)$

Man kann Schritt 2a durch geschickte Wahl einer Hilfsdatenstruktur beschleunigen (s. Arbeitsblatt Implementierung des Dijkstra-Algorithmus[6]). $O((n+m)\log n)$

Beispiel: Kürzester Weg von A nach E?



Vereinfachende Annahmen:
- keine doppelten Kanten
- keine Schlingen
- nur positive Kantengewichte

[6] „07-7 Graphen Implementierung DFS und Dijkstra-Algorithmus Arbeitsblatt"