TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Graphen

Literaturhinweis: Kapitel 4.5 aus R. Sedgewick, K. Wayne: Einführung in die Programmierung mit Java. 2011, Pearson Studium.

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Motivation

- *Graphen* sind Datentypen zur Modellierung von (paarweisen) Verbindungen von Dingen.
- Sie eignen sich daher zur Darstellung von Netzwerken (z.B. Netzwerk von Freundschaften und Verwandschaft, von Straßen, von Nervenzellen)
- $\neq$ Funktionsgraphen in der Mathematik
- Verallgemeinerung von Bäumen und Listen
  - Listenknoten haben (max.) einen Nachfolger und keine Zyklen
  - Knoten in Binärbäumen haben (max.) zwei Nachfolger und keine Zyklen

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Definitionen: Graph

Ein **gerichteter Graph** (engl. **digraph**) $G = (V, E)$ besteht aus

- einer endlichen Menge V von **Knoten** (engl. **vertices**)
- einer Menge $E \subseteq V \times V$ von **Kanten** (engl. **edges**)

Ist (va, ve) eine Kante, dann nennt man

- va den *Anfangs-* oder *Startknoten* oder *Quelle*
- ve den *Endknoten* oder das *Ziel*

der Kante.

ve heißt von va *direkt erreichbar* und *Nachfolger* von va;
va ist der *Vorgänger* von ve.

Anwendungsbeispiele: Graphenmodelle

System	Knoten	Kanten
Internet	Webseite	Link
Soziales Netzwerk	Person	Freundschaft
Software	Klasse	Verwendung von Objekten
Transport	Kreuzung	Straße
Biochemie	Protein	Interaktion

Abstrakter Datentyp: Graph

```
interface Graph<T> {
  // Fuegt Knoten mit Markierung v hinzu
  void addVertex (T v);
  // Fuegt Kanten hinzu
  void addEdge (T start, T end);
  // Anzahl der Kanten
  int numEdges();
  // Anzahl der Knoten
  int numVertices();
  // Liefert alle Knoten
  Collection<T> vertices();
  // Liefert die direkten Nachbarn eines Knotens
  Collection<T> neighbors(T v);
}
```


Repräsentierungen von Graphen

Sei $G = (V, E)$ ein gerichteter Graph, $V = \{1, \ldots, n\}$. G lässt sich z.B. speichern als:

- **Adjazenzmatrix**: boolesche $n \times n$-Matrix, wobei der Eintrag an Position (x, y) `true` ist genau dann, wenn es in G eine Kante von x nach y gibt.
- **Adjazenzlisten**: Speichere für jeden Knoten die Liste der Knoten, die mittels einer Kante verbunden sind

Implementierung mit Adjazenzlisten

Siehe Implementierung in `Graph.zip`!

Kürzeste Pfade

- Ein **Pfad** von Knoten A zu Knoten B ist die Folge von Kanten, die A und B im Graph miteinander verbindet.
- Ein **kürzester** Pfad ist ein Pfad mit einer minimalen Anzahl an Kanten in der Verbindung.
- Kürzeste Pfade müssen nicht eindeutig sein.
- Typische Fragestellungen:
  - Existiert ein Pfad zwischen zwei gegebenen Knoten?
  - Was ist ein kürzester Pfad zwischen zwei gegebenen Knoten?
  - Wie groß ist die **Distanz**, d.h. wie lang ist ein kürzester Pfad?

Berechnung der Distanz I

Fragestellung: Wie groß ist die Distanz von Knoten S zu allen anderen (erreichbaren) Knoten des Graphs?

Beobachtung:

- Die Distanz des Startknotens S zu sich selbst ist 0.
- Die Distanz der direkten Nachbarn von S ist 1.
- Achtung: Die Distanz der direkten Nachbarn der direkten Nachbarn ist nicht immer 2! (Gegenbeispiel: Startknoten S!)

$\Rightarrow$ Betrachte die Knoten in der Reihenfolge ihrer Distanz zu S!

Berechnung der Distanz II

Algorithmische Idee: Verwalte die Knoten in einer Queue, beginnend mit dem Startknoten S.

1. Entnehme den nächsten Knoten v aus der Queue.
2. Für alle Nachbarn, für die noch keine Distanz ermittelt wurde, weise eine Distanz zu, die um eins größer ist als die Distanz von v.
3. Füge diese Nachbarn in die Queue ein.

Implementierung

```
public static Map<String,Integer> distanceToOthers(String s,
      Graph<String> g) {
  Map<String,Integer> result = new HashMap<String,Integer>();
  Queue<String> queue = new LinkedList<String>();
  // Initialisierung mit Startknoten
  result.put(s,0);
  queue.add(s);
  // Verwalten der Queue
  while(!queue.isEmpty()) {
    String v = queue.remove();
    int dist = result.get(v);
    // Betrachte die Nachbarknoten
    for(String neighbor : g.neighbors(v)) {
      // falls noch nicht besucht
      if (!result.containsKey(neighbor)) {
        result.put(neighbor,dist+1);
        queue.add(neighbor);
      }
    }
  }
  return result;
}
```

Berechnung der kürzesten Pfade

Fragestellung: Was ist ein jeweils kürzester Pfad von Knoten S zu allen anderen (erreichbaren) Knoten des Graphs?
Idee:

- Beim Ermitteln der Distanz wird ein solcher Pfad konstruiert.
- Der Vorgänger auf dem Pfad ist der Nachbarknoten mit Distanz um 1 kleiner.

$\Rightarrow$ Verwalte diese Vorgängerknoten in eigener Map, ähnlich der Distanzinformation.

- Der Pfad kann dann anhand dieser Information rekonstruiert werden.

Implementierung

- Informationen zu Distanzen und Pfaden werden für jeden Knoten in einem Objekt der Klasse `ShortestPath` verwaltet.
- Im Konstruktor werden gleichzeitig Distanz und Vorgänger auf dem kürzesten Pfad vorberechnet.
- Implementierung: siehe `Graph.zip` auf Homepage

Vom Problem zum Programm

Phasen der Algorithmenentwicklung

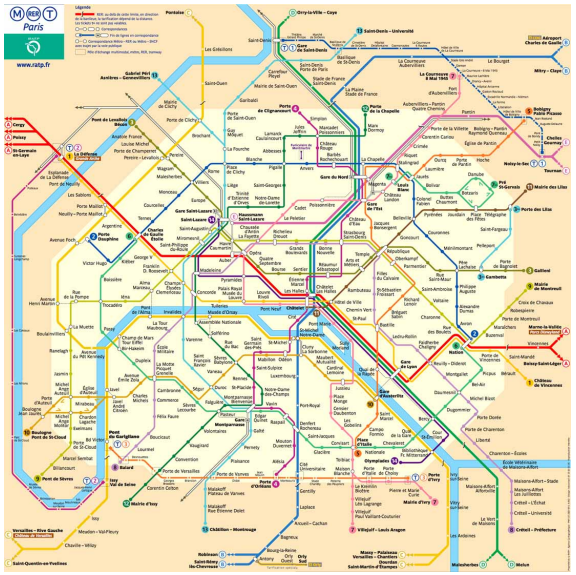
1. Konkretes Problem / Fragestellung
2. Problemabstraktion und -formulierung
3. Entwickeln einer algorithmischen Idee
4. Ermitteln wichtiger Eigenschaften des Problems
5. Grobentwurf eines Algorithmus‘ unter Zuhilfenahme bekannter Teillösungen
6. Entwickeln bzw. Festlegen der Datenstrukturen
7. Ausarbeiten des Algorithmus’

1. Problemformulierung “Pfadplanung”

Entwicklung eines Routenplaners für die Nutzung eines U-Bahn-Netzes:

Was ist die schnellste Verbindung von Station A nach Station B?

Beispiel: Paris



Beispiel: London

2. Problemabstraktion und -formulierung I

U-Bahnnetz (mit Fahrtzeiten) können als Graphen modelliert werden.

Modelliere die U-Bahn-Stationen und Verbindungen durch einen gerichteten Graphen mit bewerteten Kanten:

- U-Bahn-Stationen entsprechen Knoten.
- Kante entspricht einer *direkten* Verbindung zwischen Stationen A und B, die von A nach B befahrbar ist (ggf. auch Kante für umgekehrte Richtung).
- Jede Kante bekommt als Gewichtung die Zeit in Sekunden, die man im Durchschnitt für den Weg von A nach B braucht.

2. Problemabstraktion und -formulierung II

Die Gewichtung ist eine Funktion $c : E \to \mathbb{R}^+$
Gewichtete gerichtete Graphen nennt man *Distanzgraphen*.

Damit lässt sich das Problem wie folgt formulieren:

- Gegeben ein Distanzgraph, der das U-Bahn-Netz modelliert, sowie zwei Knoten s (Start) und z (Ziel).
- Gesucht ist ein Weg $s, v_1, \ldots, v_n, z$ mit minimaler Fahrtdauer l:

$$l = c((s, v_1)) + c((v_1, v_2)) + \cdots + c((v_n, z))$$

3. Entwickeln einer algorithmischen Idee

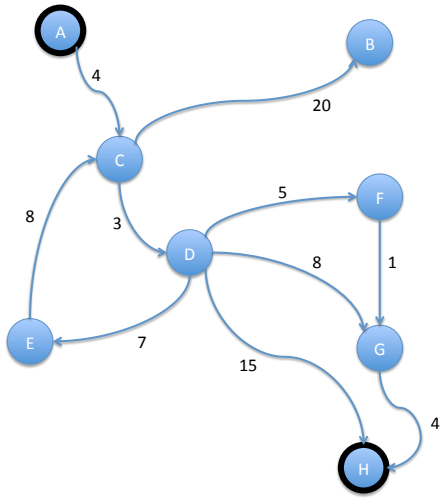
Strategie:

- Reduktion von Problemen auf ähnlich geartete einfachere (Teil-) Probleme

Reduziere die Suche des kürzesten Wegs von
s nach z auf kürzeste Wege zu den Knoten auf dem Weg

Beispiel

Aufgabe: Wie ermittelt man den kürzesten Pfad von A nach H?



Ansatz

(a) Sei die Distanz des Startknotens s zu sich selbst 0. Die Distanz aller anderen Knoten sei zunächst als unendlich angenommen.

(b) Besuche nun nach und nach alle Knoten des Graphen, um die minimale Distanz zu ermitteln.

1. Die Distanz zu den direkten Nachfolgern von s entspricht der Gewichtung der jeweiligen Kante, ihr Pfad-Vorgänger ist der Startknoten.
2. Solange der Zielknoten noch nicht besucht wurde, wähle den noch nicht besuchten Knoten mit der **kleinsten Distanz** zum Startknoten aus, ermittle seine direkten, noch unbesuchten Nachfolger und berechne die Summe der Distanz des aktuellen Knotens und des entsprechenden Kantengewichts.
3. Falls dieser Wert kleiner als die aktuell angenommene Distanz, aktualisiere sie und setze den aktuellen Knoten als Pfad-Vorgänger.

(c) Ermittle schließlich den Pfad, indem vom Zielknoten ausgehend die jeweiligen Pfad-Vorgänger gesammelt werden.

4. Ermitteln wichtiger Eigenschaften des Problems I

- Die kürzesten Teilstrecken zwischen Knoten in einem Pfad bilden zusammen die kürzeste Strecke auf diesem Pfad.
- Die Distanz bereits besuchter Knoten zum Startknoten ist minimal.
- Alle Knoten, die von bereits besuchten Knoten direkt erreichbar sind, bilden einen *Rand* um die besuchten Knoten.
- In jedem Iterationsschritt wird die Menge der besuchten Knoten um einen Knoten aus dem Rand erweitert.

4. Ermitteln wichtiger Eigenschaften des Problems II

Verfeinerung des Ansatzes:

Sei B die Menge der besuchten Knoten, und R die Menge der Randknoten.

1. Bestimme für jeden Knoten r des Randes einen Vorgänger w_r in B, so dass die Distanz zum Startknoten $d(r) = d(s, w_r) + c((w_r, r))$ minimal ist.
2. Wähle unter allen Knoten r von R den Knoten v mit minimalem $d(v)$ aus. Erweitere B um v.

5. Grobentwurf des Algorithmus' I

Wir setzen die obigen Ansätze in einen Grobentwurf um, der auf Dijkstra zurückgeht (vgl. Ottmann, Widmayer: 8.5.1):

Für jeden Knoten v verwalten wir drei zusätzliche Informationen:

`pred(v)`:	Vorgänger auf dem kürzesten Weg von s
`dist(v)`:	kürzeste bisher ermittelte Entfernung zu s
`visited`:	enthält alle Knoten, die bereits besucht wurden

Im Folgenden notieren wir die wichtigsten Teile des Algorithmus' in Form von Pseudocode.

5. Grobentwurf des Algorithmus' II

Initialisierung des Startknotens s:

```
dist.put(s,0);
visited.add(s);
```

Initialisieren des Randes R:

```
R = neuer leerer Rand;
ergaenzeRand(s,R);
```

5. Grobentwurf des Algorithmus‘ III

Schrittweise Erweiterung des Randes:

```
while R nicht leer do {
  // waehle naechstgelegenen Randknoten aus
  waehle v ∈ R mit dist.get(v) minimal ;
  ergaenzeRand(v,R);
  entferne v aus R ;
  visited.add(v);
}
```


5. Grobentwurf des Algorithmus‘ IV

```
ergaenzeRand (v, R) {
  for all w ∈ direkteNachfolger(v) do {
    if !(visited.contains(w)) and (dist.get(v) + c((v,w))
     < dist.get(w)) {
      // w ist (kuerzer) ueber v erreichbar
      pred.put(w,v);
      dist.put(w,v.dist + c((v,w))) ;
      fuege w zu R hinzu;
    }
  }
}
```


6. Entwickeln bzw. Festlegen der Datenstrukturen

In dieser Phase ist zu entscheiden, welche Datenstrukturen für die Realisierung

- des Graphen und
- des Randes

benutzt werden sollen.

Benötigte Operationen auf der Graphdatenstruktur

- Iterieren über die Knotenmenge
- Iterieren über die Kantenmenge zu einem Knoten
- Gewichtung der Kanten auslesen

⇒ Erfordert Graph-Implementierung mit gewichteten Kanten!

Benötigte Operationen auf dem Rand

- Rand als leer initialisieren
- Prüfen, ob Rand leer ist
- Wählen des Knotens mit minimaler Entfernung
- Entfernen eines Knotens aus dem Rand
- Knoten zum Rand hinzufügen bzw. Knoten im Rand modifizieren

$\Rightarrow$ z.B. Implementierung in Form eines Heaps

7. Ausarbeiten des Algorithmus'

Aus den Entscheidungen der 6. Phase entsteht ein Feinentwurf, der präzise zu formulieren und, wo möglich, zu optimieren ist.

Schließlich kann der Feinentwurf ausprogrammiert und getestet werden.

$\Rightarrow$ Siehe nächstes Übungsblatt!

Bemerkung

- Bis auf den letzten Schritt sind alle Phasen der Algorithmenentwicklung **unabhängig von Programmiersprachen**.

  Üblicherweise rechnet man die Algorithmenimplementierung auch nicht mehr zum Bereich Algorithmen und Datenstrukturen.

- **Softwareentwicklung** hat viele Parallelen zur Algorithmenentwicklung. Auch hier hat die Programmierung eine nachgeordnete Bedeutung.

  Dafür liegt der Schwerpunkt nicht so sehr auf der Lösung gut eingrenzbarer schwieriger Probleme, sondern stärker auf der Bewältigung der vielen Aspekte und des Umfangs der Aufgabenstellung.