Algorithmen für Routenplanung

Übung 1

Thomas Pajor

Lehrstuhl für Algorithmik I
Institut für theoretische Informatik
Karlsruher Institut für Technologie
Universität Karlsruhe (TH)

8. Mai 2009

Organisatorisches

Willkommen zur Übung zur Vorlesung
Algorithmen für Routenplanung

Euer Übungsleiter

- Name: Thomas Pajor
- E-Mail: `pajor@ira.uka.de`
- Raum: 322 im Informatik-Hauptgebäude
- Sprechzeiten: Kommt einfach vorbei

Outline

Organisatorisches

Geplant sind...

- Ungefähr 6 Übungsblätter
- zur Vertiefung des Theoriestoffs
- Abgabe jeweils in der Vorlesung der darauffolgenden Woche

Routenplanung ist zu großen Teilen *Algorithm Engineering*!

- $\Rightarrow$ Kleine Implementierungsaufgaben
- Beschleunigungstechniken ausprobieren
- Sehr einfach gehalten. Wir geben das meiste vor!

Übungsablauf

Normalerweise...

- Besprechung der Übungsaufgaben vom vorherigen Mal
- Vorführen von tollen Implementierungen und Ergebnissen

Heute allerdings...

- Kurze Einführung in unser Implementierungs-„Framework“
- Damit ihr zu Hause leicht loslegen könnt
- Dijkstra's Algorithmus auf kleinen Graphen!

Schnell, schneller, am schnellsten...

Maxime

Bei der Routenplanung müssen

- große Datenmengen (riesige Graphen)
- schnell und ressourcenschonend verarbeitet werden.

$\Rightarrow$ Wir benutzen C++ mit einem (sehr) schlanken Framework!

Slim-Framework

Das Framework...

- ist selbst geschrieben (und *sehr* klein)
- benutzt lediglich ein paar Komponenten der STL

Vorteile:

- sehr schnell
- kein (kaum) Overhead
- für euch überschaubar :-)

Nachteile:

- Unflexibel (aber für die Übung ausreichend)

Überblick

Das Übungs-Framework besteht aus folgenden Bestandteilen:

- **Graph-Klasse**
  Verwaltet den Graphen
- **Query-Algorithmen**
  Zum Beispiel DIJKSTRA's Algorithmus
- **Statistik-Klasse**
  Zur Auswertung der Laufzeitmessungen

... und noch ein bisschen IO.

Euer Fokus: Ausbau der Query-Algorithmen.

Graphen

Umgehen mit Graphen...

Grundlegende Datentypen

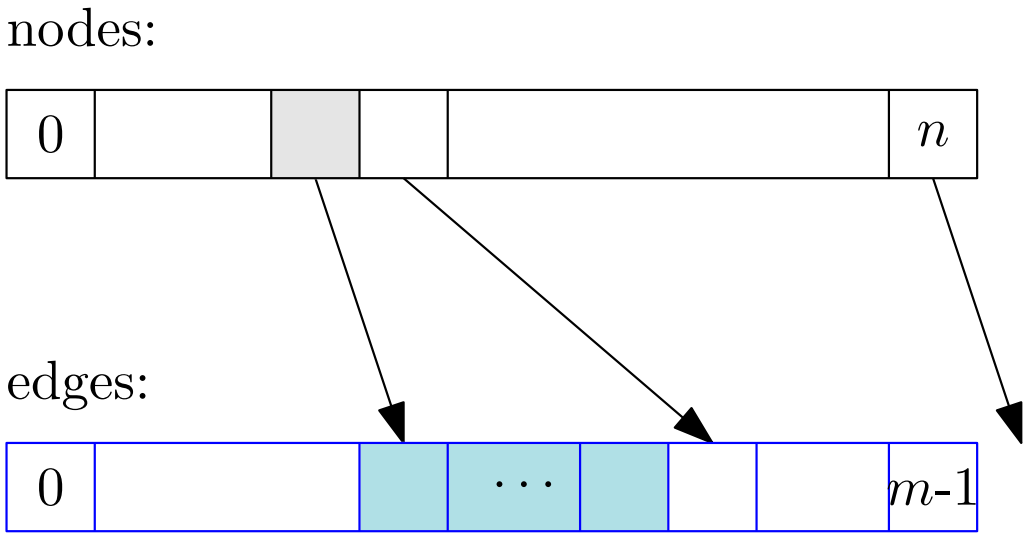
Grundlegende Datentypen aus unserem Framework sind

- `typedef unsigned int NodeID;`
  Index in dem Knotenvektor `nodes`
- `typedef unsigned int EdgeID;`
  Index in dem Kantenvektor `edges`
- `typedef int EdgeWeight;`
  Kantengewichte

- `INFTY`
  Spezieller `EdgeWeight`-Wert für ∞.

Graphen

Die Graphen sind als Adjazenzarray implementiert.

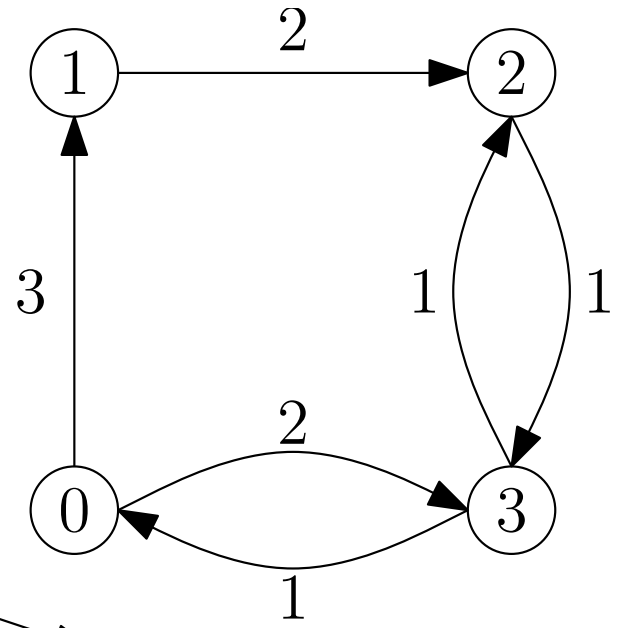



```
struct basicNode {
  EdgeID firstEdge;
  int xCoord, yCoord;
}

struct basicEdge {
  NodeID targetNode;
  bool forwardFlag;
  bool backwardFlag;
  EdgeWeight weight;
}
```

Graphen – Beispiel

Adjazenzarray an einem kleinen Beispiel



firstEdge	0	3	5	7	10				
targetNode	1	3	3	0	2	1	3	0	0	2
weight	3	2	1	3	2	2	1	1	2	1
backwardFlag	–	–	√	√	–	√	√	–	√	√
forwardFlag	√	√	–	–	√	–	√	√	–	√

Graph-Struktur – Implementierung

Wichtige Bestandteile der Graph-Klasse:

```
template<typename NodeSlot, typename EdgeSlot>
class staticGraph {
    // ...
    vector<NodeSlot> nodes;
    vector<EdgeSlot> edges;
};
```

Der einfache Graph ergibt sich damit durch

```
typedef staticGraph<basicNode, basicEdge> basicGraph;
```

Der Rest ist für uns nicht wichtig!

Größe des Graphen

Folgende Funktionen können benutzt werden Größen zu ermitteln

- `number_of_nodes()`
  Größe des `nodes`-vectors -1.
- `number_of_edges()`
  Größe des `edges`-vectors

Achtung: `number_of_edges()` entspricht in der Regel nicht der tatsächlichen Anzahl Kanten im Graph!

Iterieren auf Graphen

Folgende Konstrukte existieren zur Iteration:

- `forall_nodes(G, n)`
  Iteriert über alle Knoten in `G`
- `forall_edges(G, e)`
  Iteriert über *alle* Kanten in `G`
- `NodeID s;  forall_source_edges(G, e, s)`
  Iteriert über *alle* Kanten, `s` ist dabei der Quellknoten der Kante
- `forall_in_out_edges(G, e, n)`
  Iteriert über *alle* zu `n` inzidenten Kanten in `G`

Benutzung wie normale Schleifenkonstrukte in C++ (z. B. for)

Vorwärts und Rückwärtskanten

Problem: Nur über *ausgehende* Kanten iterieren.

```
basicGraph G;     NodeID n = 42;

// Iteriere über ausgehende Kanten von n...
forall_in_out_edges(G, e, n) {

  if (!G.edges[e].forwardFlag)
    continue;

  std::cout << "Kante " << e << " mit Gewicht "
            << G.edges[e].weight << std::endl;
}
```

Nur Rückwärtskanten: analog

Graphen

Ein erstes Beispiel...

Ein paar kleine Statistiken

Ein kleines Kennenlernprogramm zum Ausgeben:

- Anzahl Knoten,
- ...mit Grad $0, 1, 2$,
- Anzahl Vorwärts-Kanten,
- Maximal- und Minmalgewicht

...eines Graphen über die Kommandozeile.

Live

Graphen

Implementierung von Routingalgorithmen...

Vorgaben

Wir geben euch als Input:

- **Graphen**
  San Francisco Bay Area, Florida, ...
- **Demand-Files**
  zu jedem Graphen.

Demand-Files

Idee: Testen der Korrektheit der Beschleunigungstechniken.

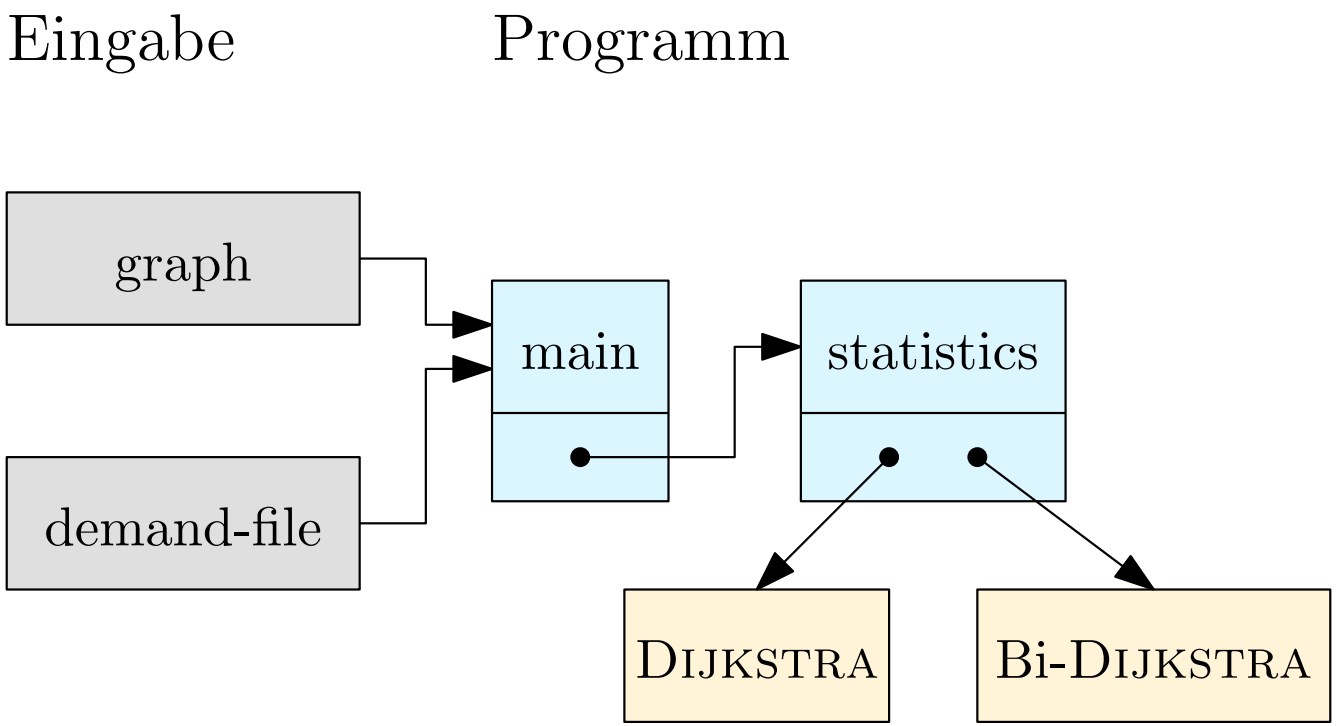
Demand-Files enthalten zu einem Graphen n Einträge mit je

- Source `NodeID` s,
- Target `NodeID` t,
- Distanz $\text{dist}(s, t)$.

$\rightsquigarrow$ Programm führt zu jedem Eintrag ein s, t-Query mit einer Beschleunigungkstechnik aus und vergleicht das Ergebnis mit $\text{dist}(s, t)$.

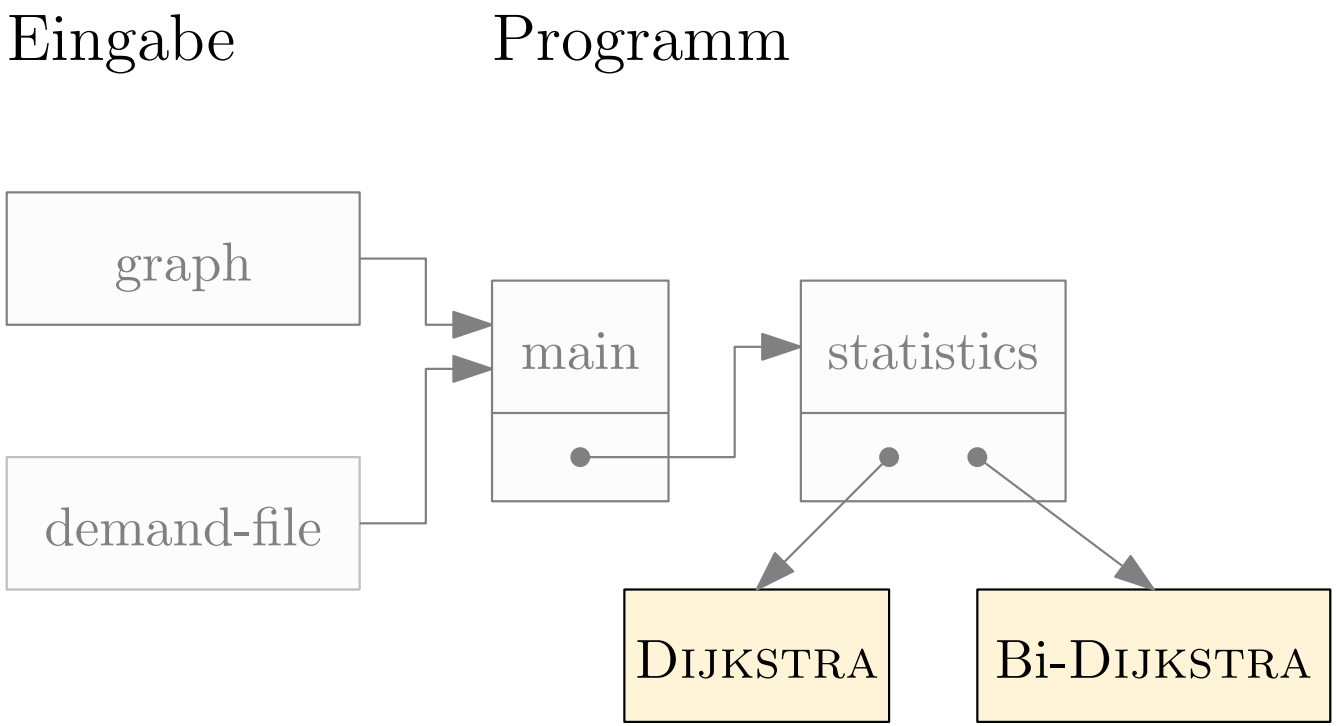
Struktur

Schematischer Ablauf in unserem Rahmenwerk:



Struktur

Schematischer Ablauf in unserem Rahmenwerk:



Verschiedene Algorithmen

Die einzelnen Algorithmen werden der Statistik-Klasse über Templates übergeben.

Ausschnitt aus `main()`:

```
basicGraph G;
[...] // Graph aus Datei laden...

Dijkstra<basicGraph> dij(G);
DijkstraStatistic<Dijkstra<basicGraph>,basicGraph>
                                    ds(G,dij,demandFilename);
ds.evaluateDijkstra();
```

Algorithmus-Interface

Die Algorithmen müssen folgende Funktionalität bereitstellen:

```
template<typename graphType>
class Dijkstra {
public:
  Dijkstra(graphType& g, bool forward=true);

  void run(NodeID s, NodeID t);

  EdgeWeight getDistance();
  int getNumberOfRelaxedEdges();
  int getNumberOfSettledNodes();

  std::string description();
};
```

Algorithmus-Interface

Die Statistik-Klasse ruft für jeden Eintrag aus dem Demand-File einmal `run(s,t)` im Algorithmus auf.

Das heißt:

- Der Algorithmus wird nicht für jede Anfrage neu erstellt.
- Manuelles Einhalten der Konsistenz, z. B. von $\text{dist}_s(\cdot)$.
  $\rightsquigarrow$ Zum Beispiel über „Counter“ (siehe Vorlesung)

Zur Verifikation holt sich die Statistik-Klasse die s-t-Distanz durch Aufruf von `getDistance()`.

Kapselung der Priority-Queue

Unsere Priority-Queue ist etwas „unhandlich“.

Kapselung wichtiger Funktionen:

- `NodeID deleteMin()`
  Gibt Knoten mit minimalem Key zurück und löscht ihn
- `void insertNode(NodeID n, EdgeWeight key)`
  Fügt Knoten `n` mit Key `key` ein
- `void decreaseNode(NodeID n, EdgeWeight key)`
  Senkt den Key von Knoten `n` auf den Wert `key`

Graphen

Dijkstra's Algorithmus...

Dijkstra's Algorithmus

++count
$d[s] = 0$
$Q.clear(),\ Q.add(s, 0)$
while $!Q.empty()$ **do**
  $u \leftarrow Q.deleteMin()$
  if $u = t$ **then RETURN**
  for *all edges* $e = (u, v) \in E$ **do**
    if $run[v] \neq$ **count then**
      $d[v] \leftarrow d[u] + \text{len}(e)$
      $Q.insert(v, d[v])$
      $run[v] \leftarrow$ **count**
    else if $d[u] + \text{len}(e) < d[v]$ **then**
      $d[v] \leftarrow d[u] + \text{len}(e)$
      $Q.decreaseKey(v, d[v])$