Algorithmen und Datenstrukturen (ESE)
Entwurf, Analyse und Umsetzung von Algorithmen (IEMS)
WS 2013 / 2014

Vorlesung 12, Donnerstag, 23. Januar 2014
(Graphen, Breiten/Tiefensuche, Zusammenhangskomponenten)

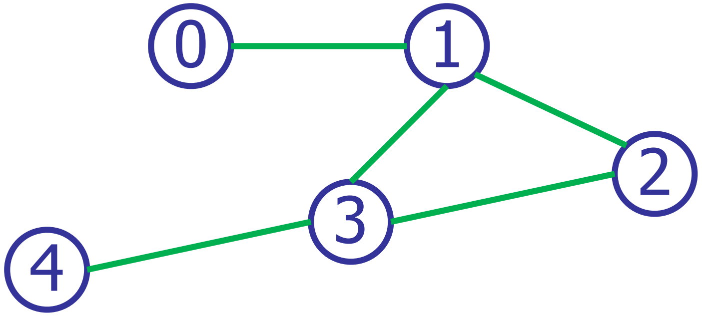
Junior-Prof. Dr. Olaf Ronneberger
Image Analysis Lab
Institut für Informatik
Universität Freiburg

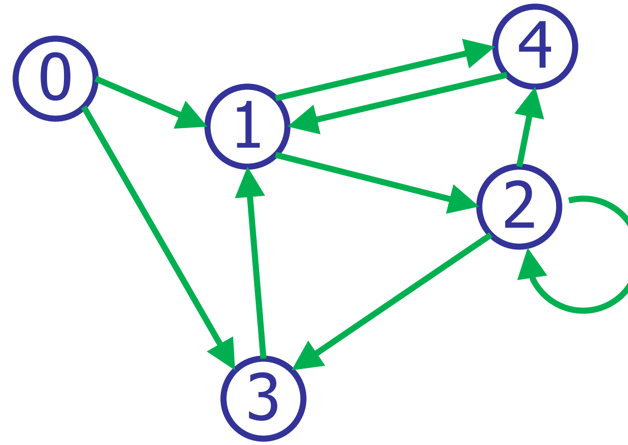
[basiert auf der letztjährigen Veranstaltung von Prof. Dr. Hannah Bast, Lehrstuhl für Algorithmen und Datenstrukturen, mit Folien aus „Informatik 2“ von Prof. Dr. Matthias Teschner]

Blick über die Vorlesung heute

- Graphen
  - Neben Feldern, Listen und Bäumen die häufigste Datenstruktur (Bäume sind eine spezielle Art von Graph)
  - Darstellung im Rechner
  - **Breitensuche** (Breadth First Search = BFS)
  - **Tiefensuche** (Depth First Search = DFS)
  - **Zusammenhangskomponenten** eines Graphen
  - **Übungsblatt 12:** Berechnung der größten Zusammenhangs-komponente in einem Straßengraphen mittels BFS oder DFS

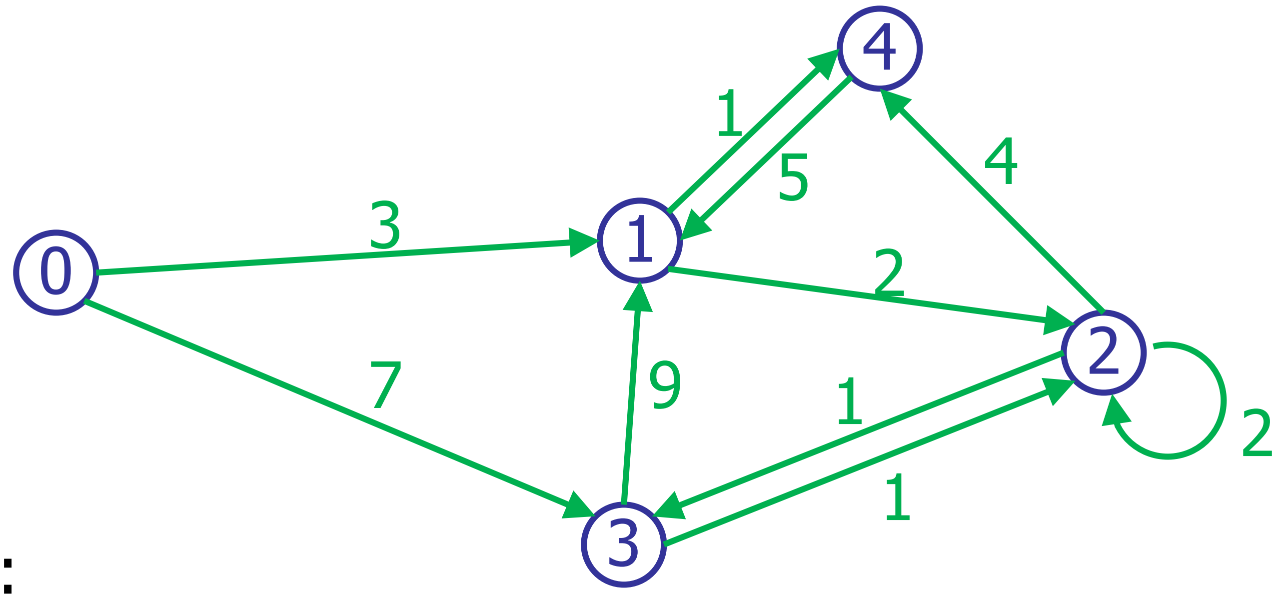
Graphen: Definition





- Definition:
  - Ein Graph $G = (V, E)$ besteht aus einer Menge V von Knoten ...
    - Englisch: vertices (daher V) oder nodes
  - ... und einer Menge E von Kanten
    - English: edges (daher E) oder arcs
  - Eine Kante e verbindet jeweils zwei Knoten u und v
    - ungerichtete Kante: $e = \{u, v\}$ (Menge)
    - gerichtete Kante: $e = (u, v)$ (Tupel)
  - Es kann auch „self-loops" geben: (u, u)

Gewichteter Graph



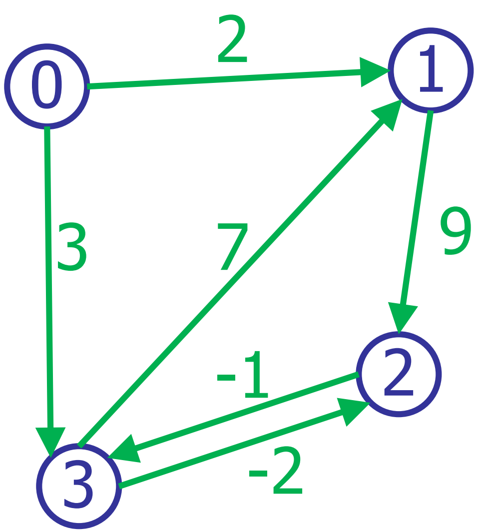
- Definition:
  - **Gewichteter** Graph: Eine reelle Zahl pro Kante, das sogenannte Gewicht der Kante, je nach Anwendung auch Länge oder Kosten der Kante genannt
  - Beispiel: Straßennetz.
    - Kreuzungen: Knoten
    - Straßen: Kanten
    - Fahrzeit: Kosten der Kante

Repräsentation im Rechner

- Wie repräsentiert man Graphen im Rechner
  - Da gibt es zwei klassischen Arten

    Adjazenzmatrix … Platzverbrauch $\Theta(|V|^2)$

    Adjazenzlisten bzw. **–felder** … Platzverbrauch $\Theta(|V| + |E|)$



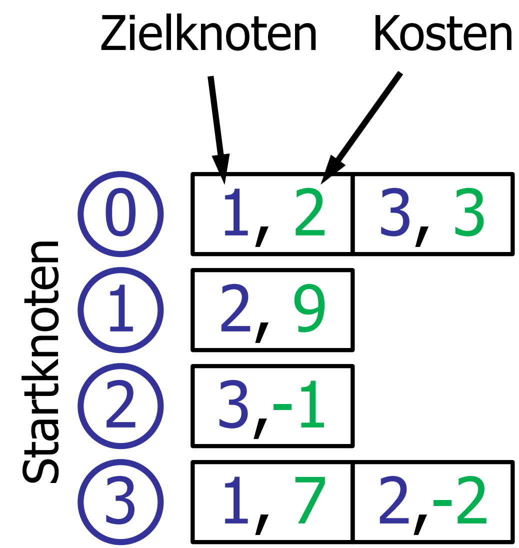
Gerichteter Graph mit Kantengewichen

$|V| = 4, \ |E| = 6$

Zielknoten

Startknoten	0	1	2	3
0		2		3
1			9	
2				-1
3		7	-2	

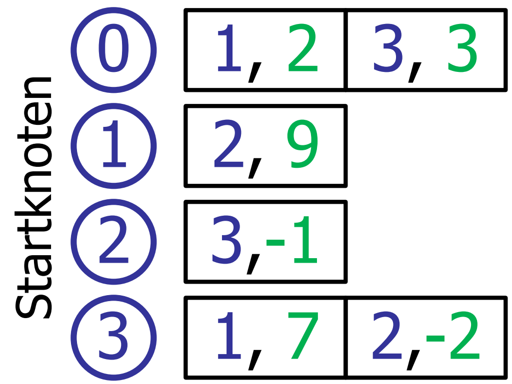
Adjazenzmatrix

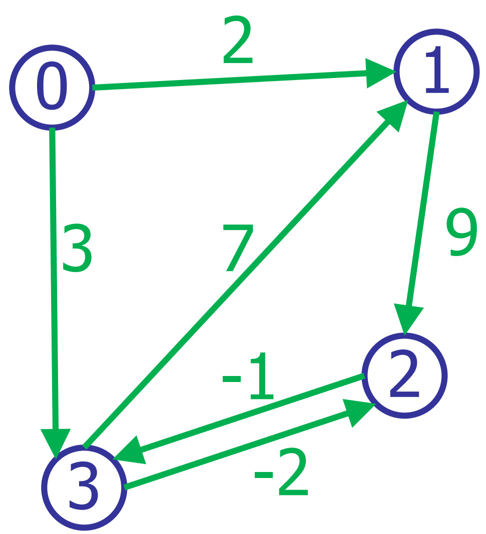


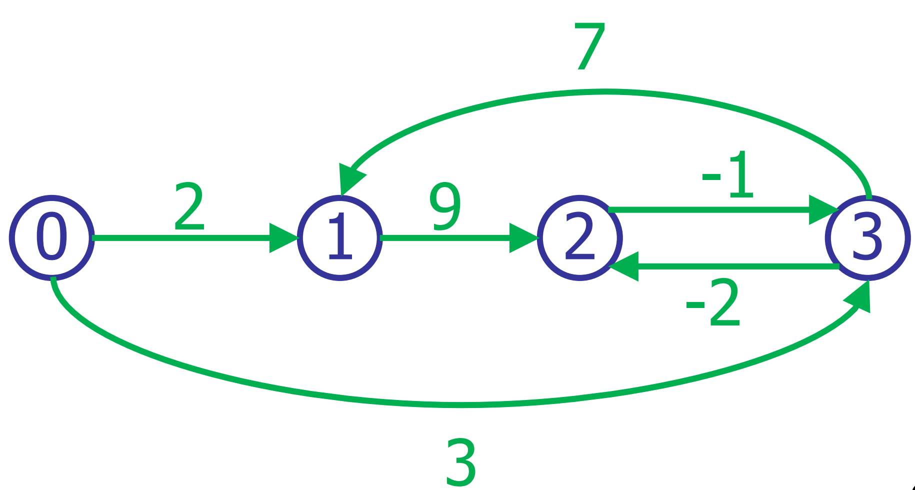
Adjazenzlisten

Graphen: Anordnung

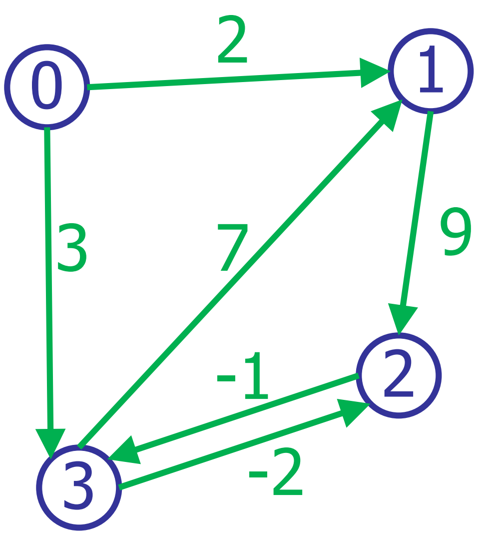
- Der Graph ist durch die Adjazenzmatrix, bzw. durch die Adjazentlisten eindeutig definiert.
- Die Anordnung bei der Zeichnung des Graphen spielt keine Rolle:

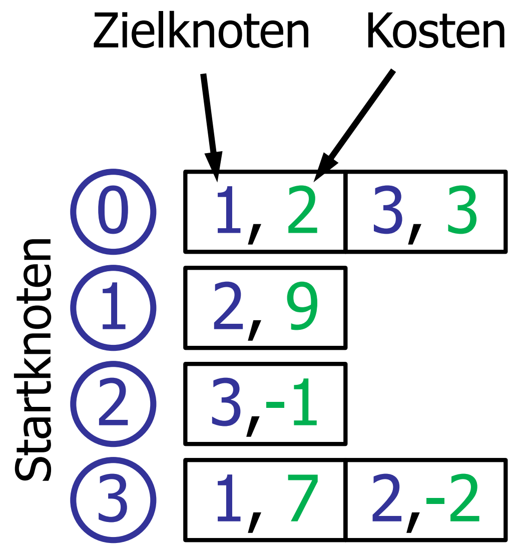






Graphen Implementierung





Adjazenzlisten

```
class Arc {
 public:
  int headNodeId;
  int cost;
 };
```

- Implementierung z.B. als Feld von Feldern
  - **C++:** std::vector< std::vector<Arc> > adjacency_lists;
  - **Java:** ArrayList<ArrayList<Arc>> adjacencyLists;
- Oder als Feld von verketteten Listen
  - **C++:** std::vector< std::list<Arc> > adjacency_lists;
  - **Java:** ArrayList<LinkedList<Arc>> adjacencyLists;

Implementierung: GraphTest.cpp

```
#include <gtest/gtest.h>
#include "./Graph.h"

TEST(GraphTest, connector) {
  Graph graph;
  ASSERT_EQ("{0, 0}", graph.toString());
}

TEST(GraphTest, addNodesAndArcs) {
  Graph graph;
  // Add three nodes: 0, 1, 2.
  graph.addNode();
  graph.addNode();
  graph.addNode();
  // Add four arcs.
  graph.addArc(0, 1);
  graph.addArc(1, 2);
  graph.addArc(0, 2);
  graph.addArc(2, 0);
  // Check that the correct graph was constructed.
  ASSERT_EQ("{3, 4, (0,1), (0,2), (1,2), (2,0)}", graph.toString());
}
```

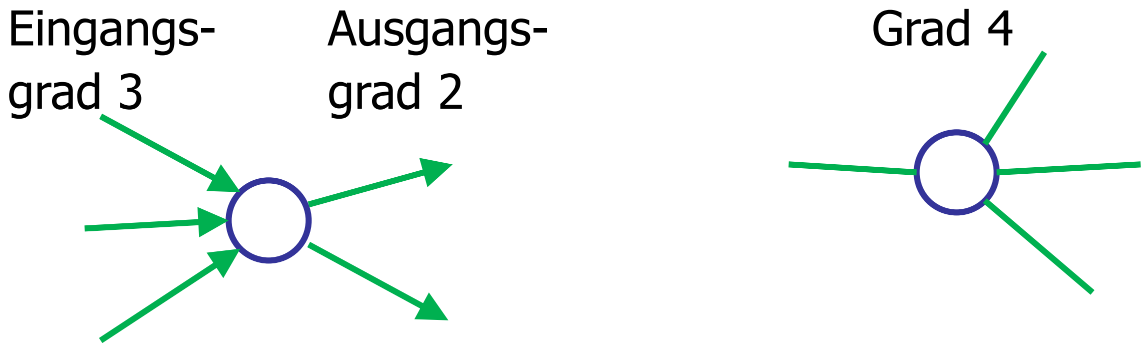
Implementierung: Graph.h

```
#include <string>

class Graph {
 public:
  // Construct an empty graph.
  Graph();

  // Add a node to this graph.
  void addNode();

  // Add an arc = a directed edge to this graph.
  void addArc(int u, int v);

  // Show the graph in human-readable form
  // (useful for debugging and testing).
  std::string toString();
};
```

Implementierung: Graph.cpp

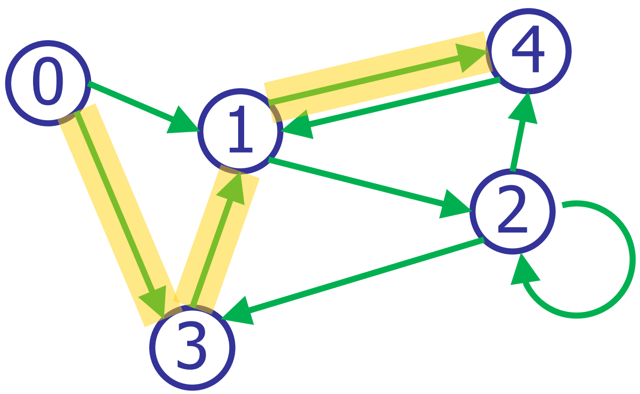
```
#include "./Graph.h"

Graph::Graph() {
}

void Graph::addNode() {
}

void Graph::addArc(int u, int v) {
}

std::string Graph::toString() {
  return "";
}
```

Graphen: Eingangs- / Ausgangsgrad

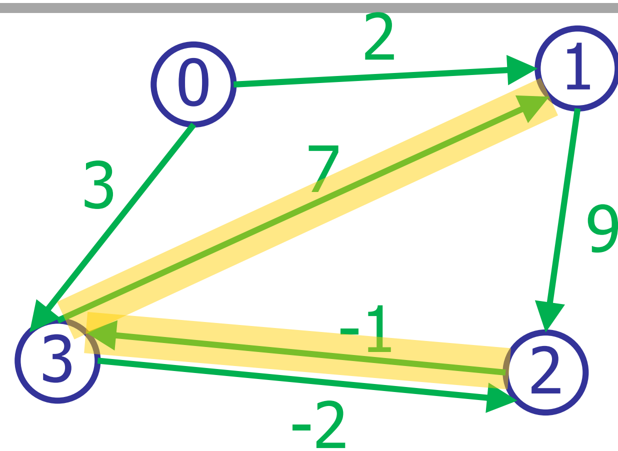


- Grade in einem Graphen $G = (V, E)$
  - Falls gerichtet
    - **Eingangsgrad** von einem Knoten u
      $= $ Anzahl eingehender Kanten $= |\{(v,u) : (v,u) \in E\}|$
    - **Ausgangsgrad** von einem Knoten u
      $= $ Anzahl ausgehender Kanten $= |\{(u,v) : (u,v) \in E\}|$
  - Falls ungerichtet
    - **Grad** von einem Knoten u
      $= $ Anzahl adjazenter Kanten $= |\{\{u,v\} : \{u,v\} \in E\}|$

Pfade im Graphen



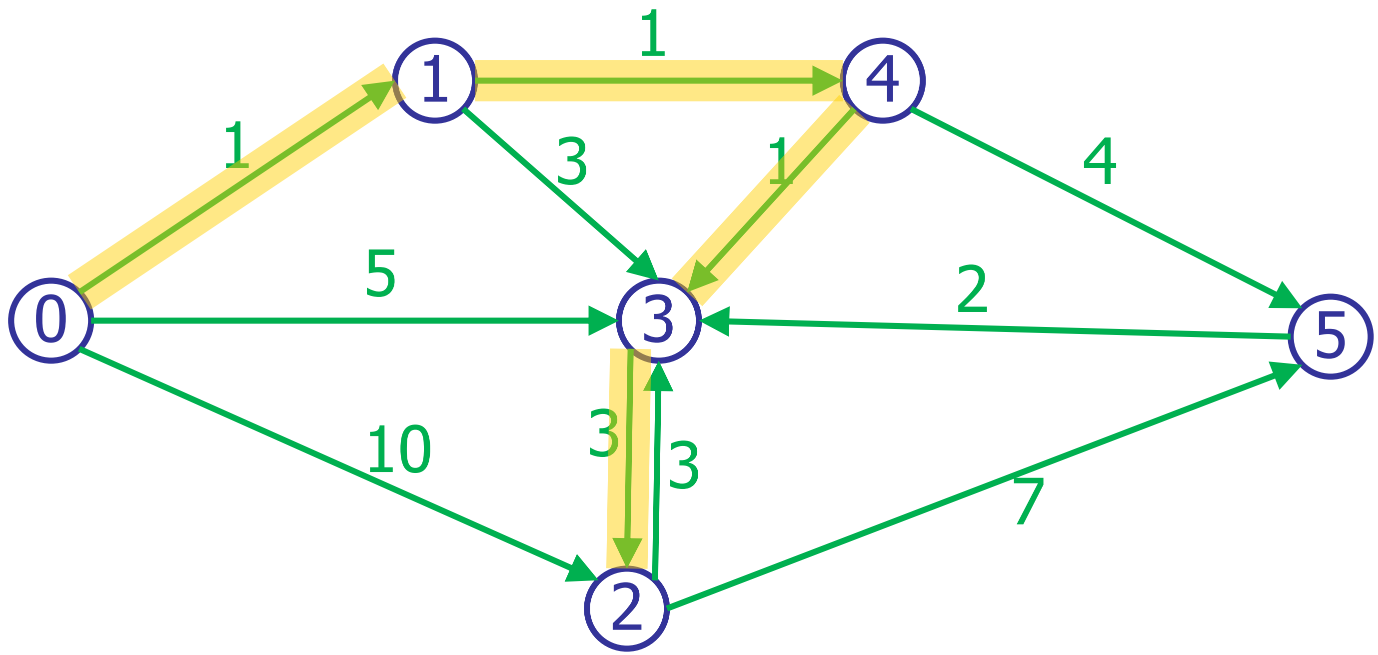
Pfad 0,3,1,4 mit Länge 3



Pfad 2,3,1 mit Kosten 6

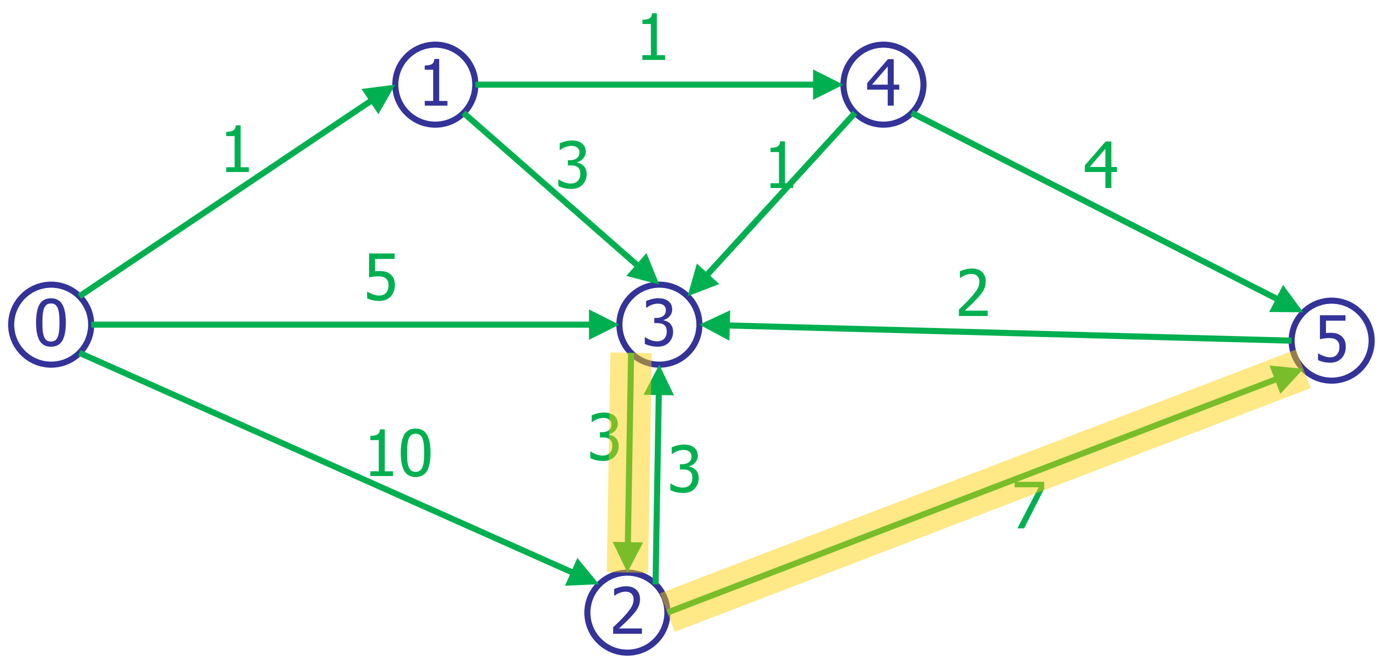
- Pfade in einem Graphen $G = (V, E)$
  - Ein Pfad in G ist eine Folge $u_1, u_2, u_3, \ldots, u_l \in V$ mit
    - $(u_1,u_2), (u_2,u_3), \ldots, (u_{l-1},u_l) \in E$ [gerichteter Graph]
    - $\{u_1,u_2\}, \{u_2,u_3\}, \ldots, \{u_{l-1},u_l\} \in E$ [ungerichteter Graph]
  - Die **Länge des Pfades** (auch: Kosten des Pfades)
    - ohne Kantengewichte: Anzahl der Kanten
    - mit Kantengewichten: Summe der Gewichte auf dem Pfad

Kürzester Pfad



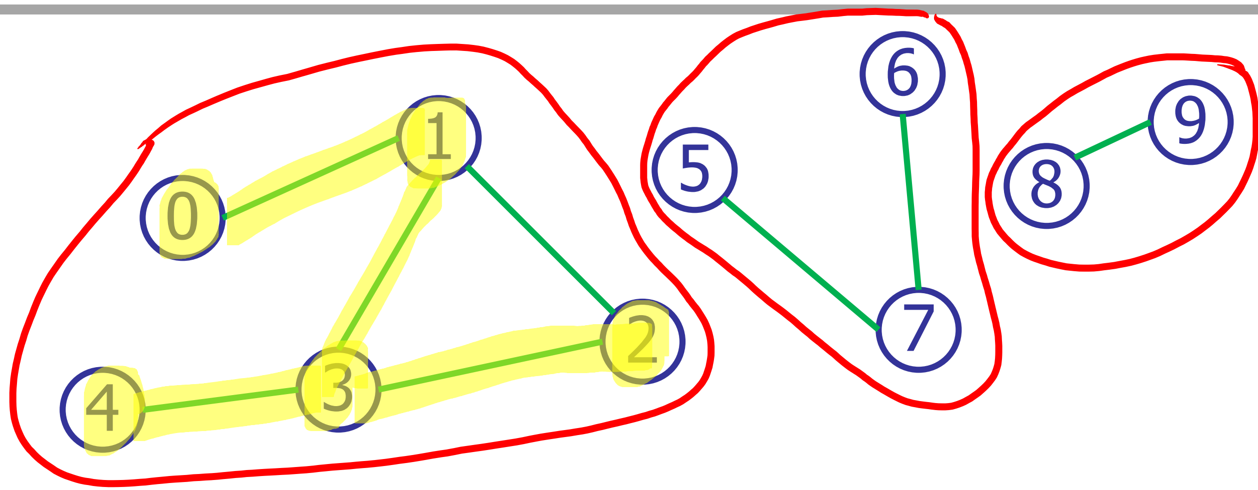
- Pfade in einem Graphen G = (V, E)
  - Der **kürzeste Pfad** (engl. shortest path) zwischen zwei Knoten u und v ist der Pfad u, ..., v mit der kürzesten Länge (bzw. geringsten Kosten)
  - Beispiel: Der kürzeste Pfad von 0 nach 2?
    - Pfad 0,1,4,3,2 mit Kosten 6

Durchmesser



- Pfade in einem Graphen $G = (V, E)$
  - Der **Durchmesser** eines Graphen ist der längste kürzeste Pfad = $\max_{u,v}$\{Länge von P : P ist ein kürzester Pfad zwischen u und v\}
  - Durchmesser des Beispielgraphen?
    - Knoten 3 und 5 sind „am weitesten" entfernt: Kosten 10

Zusammenhangskomponenten



- Für einen **ungerichteten** Graphen G = (V,E)
  - Die Zusammenhangskomponenten bilden eine Partition von V, also $V = V_1 \cup V_2 \cup \cdots \cup V_k$
  - Zwei Knoten u und v sind in derselben Zusammenhangs-komponente, wenn es einen Pfad zwischen u und v gibt

(Für **gerichtete** Graphen ist die Definition komplizierter, man spricht dann von **starken** Zusammenhangskomponenten, das machen wir in dieser Vorlesung aber nicht)

Graphexploration

- Informale Definition
  - Gegeben ein Graph $G = (V, E)$ und ein Startknoten $s \in V$, besuche "systematisch" alle Knoten von V, die von s erreichbar sind
  - Breitensuche = in der Reihenfolge der "Entfernung" von s
    - Englisch: breadth first search = BFS
  - Tiefensuche = erstmal "möglichst weit weg" von s
    - Englisch: depth first search = DFS
  - Das ist kein "Problem" an sich, taucht aber oft als Teil / Subroutine von anderen Algorithmen auf

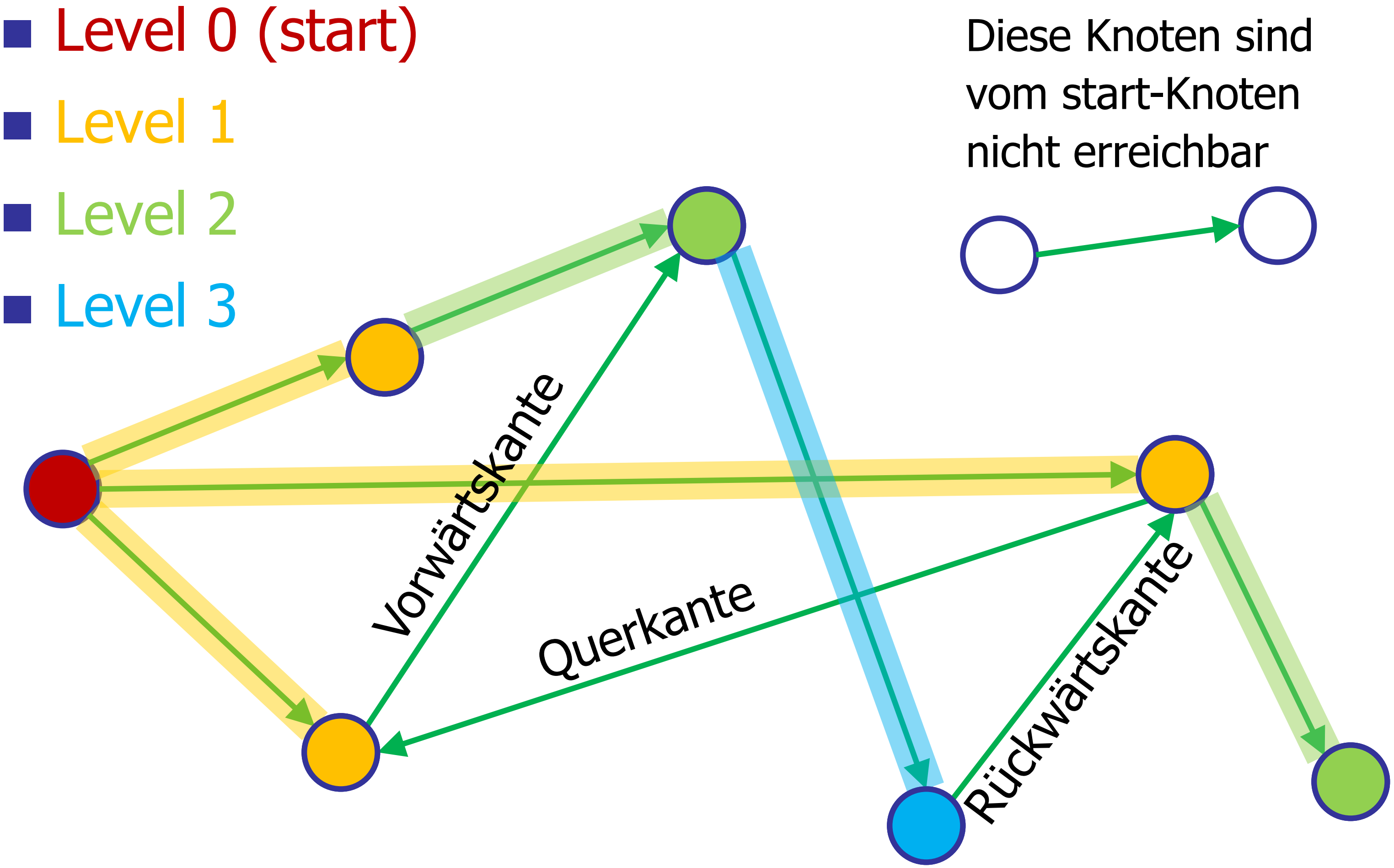
    Zum Beispiel in der Übungsaufgabe, zur Berechnung der Zusammenhangskomponenten

Breitensuche (BFS)   1/2

- Idee
  - Markierung für jeden Knoten, zu Beginn alle unmarkiert
  - Beginne mit einem Startknoten und markiere ihn (Level 0)
  - Finde alle Knoten die zum Startknoten benachbart und noch nicht markiert sind und markiere sie (Level 1)
  - Finde alle Knoten, die zu einem Level-1 Knoten benachbart und noch nicht markiert sind und markiere sie (Level 2)
  - Usw. bis ein Level keine benachbarten Knoten mehr hat, die noch nicht markiert sind
  - Das markiert insbesondere alle Knoten, die in derselben **Zusammenhangskomponente** sind wie der Startknoten

Breitensuche (BFS) 1/2

- Level 0 (start)
- Level 1
- Level 2
- Level 3



- Die markierten Kanten bilden einen aufspannenden Baum „spanning tree" (Baum, der alle erreichbaren Knoten enthält)

Tiefensuche (DFS) 1/2

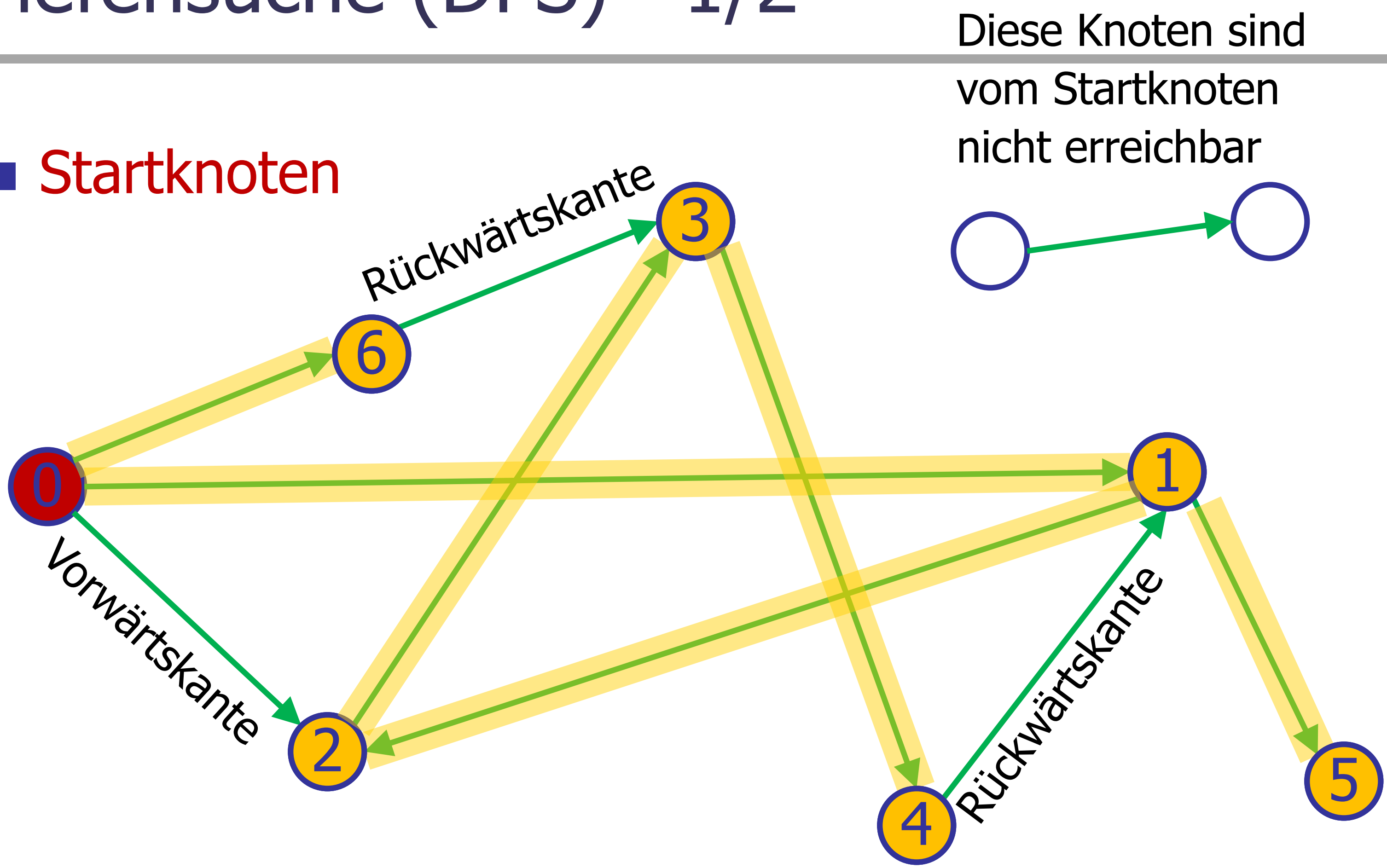
- Idee
  - Markierung für jeden Knoten, zu Beginn alle unmarkiert
  - Beginne mit einem Startknoten und markiere ihn
  - Gehe in irgendeiner Reihenfolge die zum Startknoten benachbarten Knoten durch und tue Folgendes:

    Falls der Knoten noch nicht markiert ist, markiere ihn und starte **rekursiv** eine Tiefensuche von dort aus
  - Das sucht zuerst "in die Tiefe" (vom Startknoten aus)
  - Auch DFS markiert schließlich alle Knoten, die in derselben Zusammenhangskomponenten liegen wie der Startknoten
  - Auf azyklischen Graphen liefert DFS **topologische Sortierung**

    Das ist eine Nummerierung der Knoten, so dass jede Kante von einem Knoten mit kleinerer Nummer zu einem mit größerer Nummer geht

Tiefensuche (DFS) 1/2

- Startknoten



- Die markierten Kanten bilden wieder einen aufspannenden Baum, aber einen anderen
- Wenn der Graph azyklisch ist (ohne Kanten (6,3) und (4,1)), dann liefert die Nummerierung eine topologische Sortierung

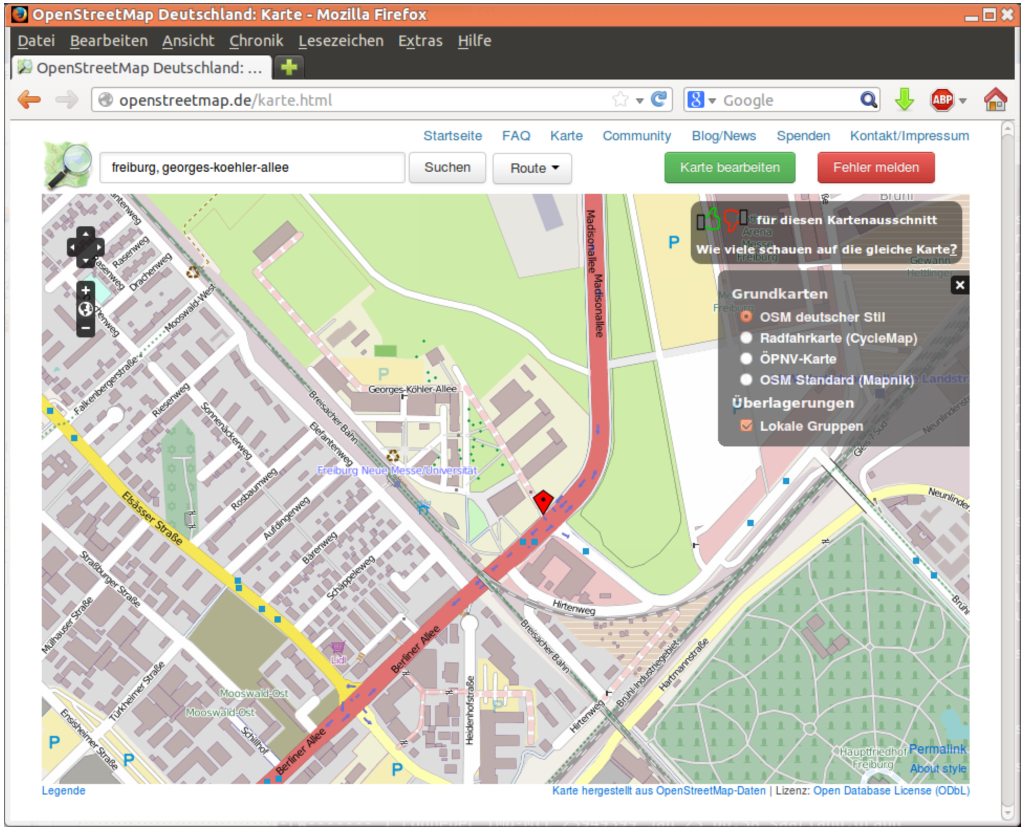
Komplexität von BFS und DFS

- Für beide Verfahren gilt:
  - Konstante Arbeit für jeden Knoten und jede Kante
  - Die Laufzeit ist also genau $\Theta(|V'| + |E'|)$

    wobei V' und E' gerade die Menge aller Knoten und Kanten in der ZK sind, in der der Startknoten liegt
  - Das kann man also (bis auf einen konstanten Faktor) nicht besser machen

Übungsblatt

- Größte Zusammenhangskomponente im Saarland aus OpenStreetMap
- http://openstreetmap.de/karte.html



Übungsblatt

- OSM-Graph aufbereitet (nur Straßen) auf Homepage als saarland.graph.zip
- Format (Details siehe „Graph.H“)

1119289			Anzahl der Knoten
497514			Anzahl der Kanten
49.3414	7.30149		Koordinaten von Knoten 0 <latitude><TAB><longitude>
49.3407	7.30063		Koordinaten von Knoten 1
49.3406	7.30042		Koordinaten von Knoten 2
…			…
0	7	1	Kante von Knoten 0 nach Knoten 7, Kosten 1 Sekunde
0	1	3.5	Kante von Knoten 0 nach Knoten 1, Kosten 3.5 Sekunden
1	0	3.5	Kante von Knoten 1 nach Knoten 0, Kosten 3.5 Sekunden
1	2	0.5	usw.
2	1	0.5	
2	174400	0.5	
3	8	0.5	
…			

Einlesen von Dateien in C++

- Speziell von zeilenbasierten Daten
  - Option 1: **FILE*** und **getline**

    Effizient und gut, wenn auch "C Style"
  - Option 2: **ifstream** und **getline**

    "C++ Style", da gab es früher Probleme mit Dateien > 2GB, aber inzwischen genauso gut wie FILE* und getline

    Felderweise << ist aber ineffizient, erst ganze Zeile lesen !
  - Option 3: **fscanf** bzw. **sscanf**

    Fehleranfällig und ineffizient, no-no bei großen Daten
  - Option 4: **FILE*** und **read**

    Alles an einem Stück in einen String einlesen ist natürlich am effizientesten, aber doof wenn nicht alles in den Speicher passt

Literatur / Links

- Graphen
  - In Mehlhorn/Sanders:

    8 Graph Representation
  - In Wikipedia

    http://en.wikipedia.org/wiki/Graph_(mathematics)
- Graphexploration und Zusammenhangskomponenten
  - In Mehlhorn/Sanders:

    9 Graph Traversal
  - In Wikipedia

    http://en.wikipedia.org/wiki/Breadth-first_search

    http://en.wikipedia.org/wiki/Depth-first_search

    http://en.wikipedia.org/wiki/Connected_component