

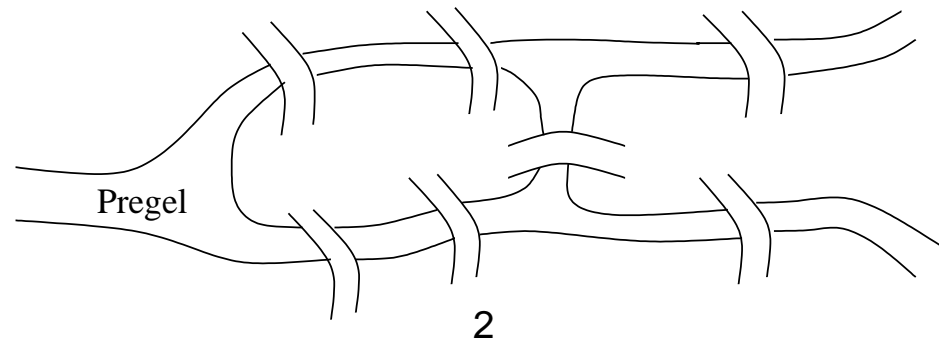
Algorithmen und Datenstrukturen

Graphen

Wolfram Burgard und Bernhard Nebel

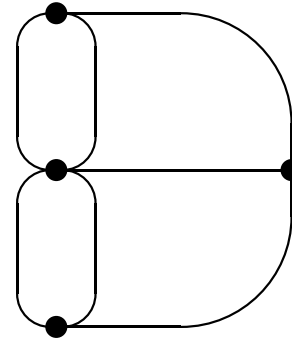
Motivation

- Wie komme ich am besten **von** Freiburg **nach** Ulm?
- Was ist die **kürzeste Rundreise** durch eine gegebene Menge von Städten?
- Welche **Menge** an Wasser kann die Kanalisation von Freiburg **maximal verkraften**?
- Gibt es einen **Rundweg** über die Brücken von Königsberg (Kaliningrad), derart dass jede Brücke nur einmal überquert wird und man zum Ausgangspunkt zurückgelangt?
- Diese und viele andere Probleme lassen sich als **Graphenprobleme** definieren.

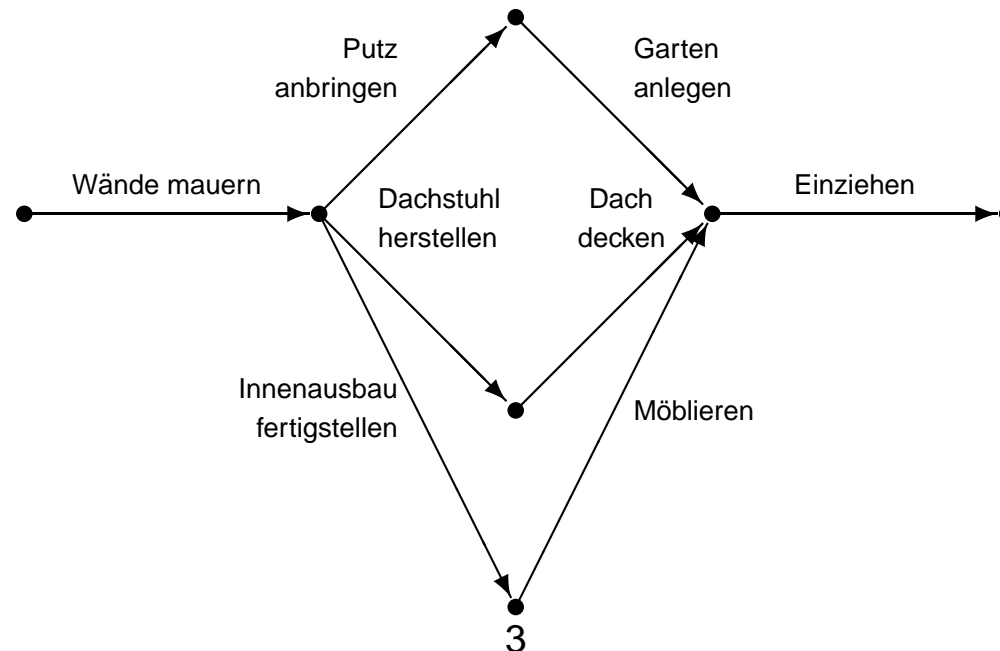


Repräsentation von Problemen durch Graphen

Das Königsberger Brückenproblem:



Ein Planungsproblem:



Definition von Graphen

Definition: Ein *gerichteter Graph* $G = (V, E)$ (englisch: *digraph*) besteht aus einer Menge $V = \{1, 2, \dots, |V|\}$ von *Knoten* (englisch: *vertices*) und einer Menge $E \subseteq V \times V$ von *Pfeilen* oder *Kanten* (englisch: *edges, arcs*). Ein Paar $(v, v') \in E$ heißt *Pfeil* oder *Kante von v nach v'*.

Darstellung:

- Knoten werden durch Punkte dargestellt und
- Kanten bzw. Pfeile werden durch Verbindungslinien mit Pfeilspitze auf den Endknoten dargestellt.

Einschränkung: Endliche Graphen, d.h. $|V| < \infty$.

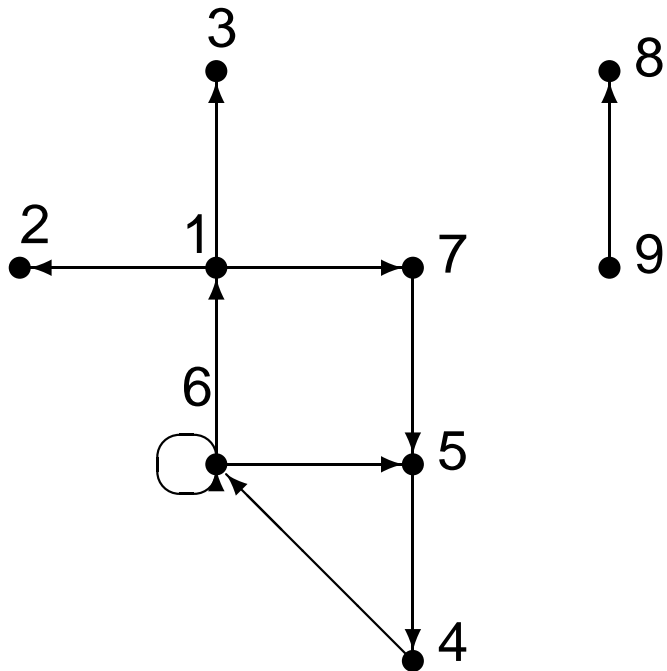
Adjazenzmatrizen

- Adjazenzmatrizen dienen der Speicherung von Graphen.
- Ein Graph $G = (V, E)$ wird in einer Boole'schen $|V| \times |V|$ -Matrix $A_G = (a_{ij})$, mit $1 \leq i \leq |V|$, $1 \leq j \leq |V|$ gespeichert, wobei

$$a_{ij} = \begin{cases} 0 & \text{falls } (i, j) \notin E; \\ 1 & \text{falls } (i, j) \in E. \end{cases}$$

```
class graph{
    graph(int n){
        this.numberOfNodes = n;
        this.a = new boolean[n][n];
    }
    private int numberOfNodes;
    private boolean[][] a;
}
```

Beispiel einer Adjazenzmatrix



	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	1	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0
5	0	0	0	1	0	0	0	0	0
6	1	0	0	0	1	1	0	0	0
7	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	1	0

Eigenschaften von Adjazenzmatrizen

- Bei der Speicherung eines Graphen mit Knotenmenge V in einer **Adjazenzmatrix** ergibt sich ein **Speicherbedarf von $\Theta(|V|^2)$** .
- Dieser **Speicherbedarf ist nicht abhängig von der Anzahl der Kanten** im Graphen.
- Demnach sind **Adjazenzmatrizen ungünstig, wenn der Graph vergleichsweise wenige Kanten enthält**.
- Wegen der erforderlichen **Initialisierung** der Matrix oder der Berücksichtigung aller Einträge der Matrix benötigen **die meisten Algorithmen $\Omega(|V|^2)$ Rechenschritte**.
- Dies kann man jedoch teilweise dadurch umgehen, dass man in jedem Feld Zusatzinformationen speichert.

Adjazenzlisten

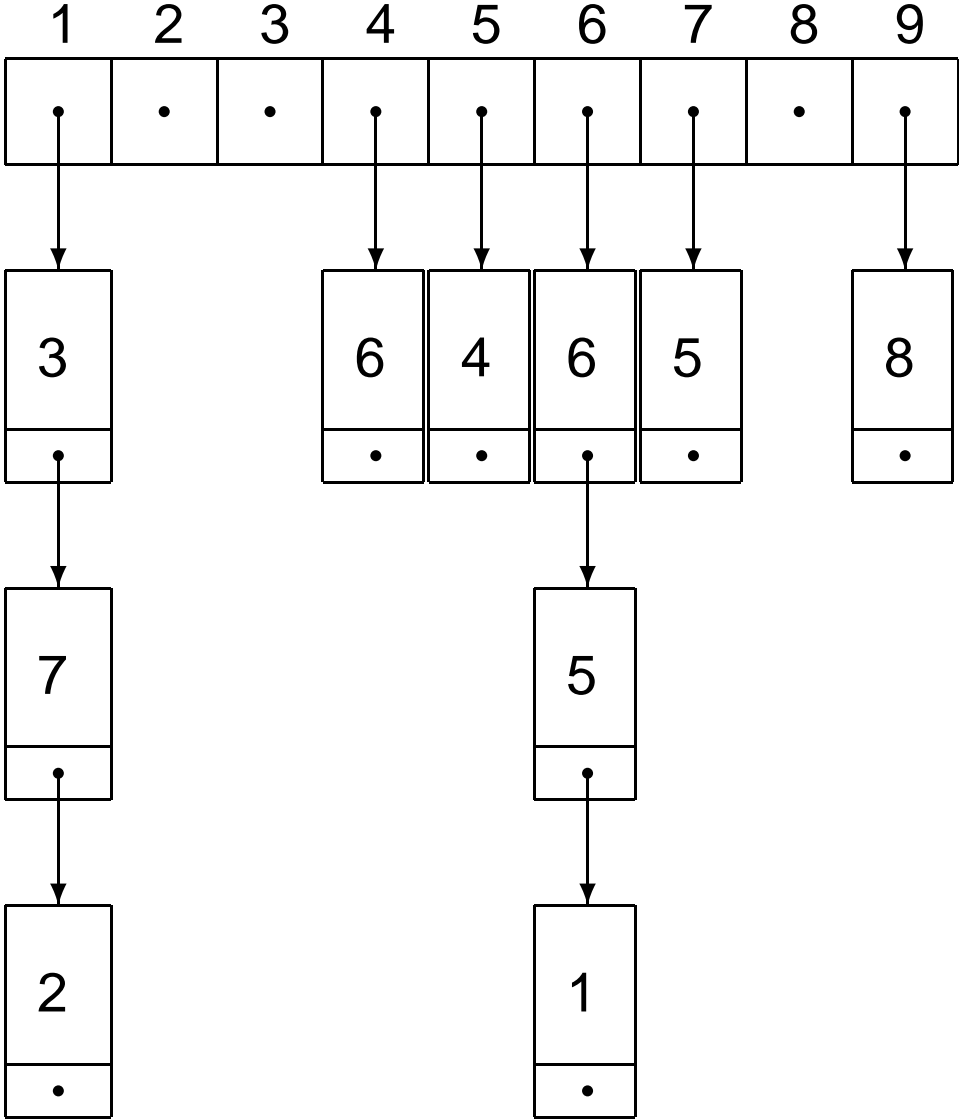
- Bei **Adjazenzlisten** wird für jeden Knoten eine **lineare, verkettete Liste** der von diesem Knoten ausgehenden **Kanten** gespeichert.
- Die **Knoten** werden als **lineares Feld** von $|V|$ **Anfangszeigern** auf je **eine solche Liste** verwaltet.
- Die i -te **Liste** enthält ein **Listenelement** mit **Eintrag** j für jeden **Endknoten eines Pfeils** $(i, j) \in E$.
- **Adjazenzlisten unterstützen** viele Operationen, z. B. das **Verfolgen von Pfeilen in Graphen**, sehr gut.
- **Andere Operationen** dagegen werden **nur schlecht unterstützt**, insbesondere das **Hinzufügen und Entfernen von Knoten**.

Implementierung von Adjazenzlisten

```
class graphAL{
    graphAL(int n){
        this.numberOfNodes = n;
        this.edgeTo = new edge[n];
    }
    private int numberOfNodes;
    private edge[] edgeTo;
}
```

```
class edge {
    edge(int node, edge next){
        this.node = node;
        this.next = next;
    }
    int node;
    edge next;
}
```

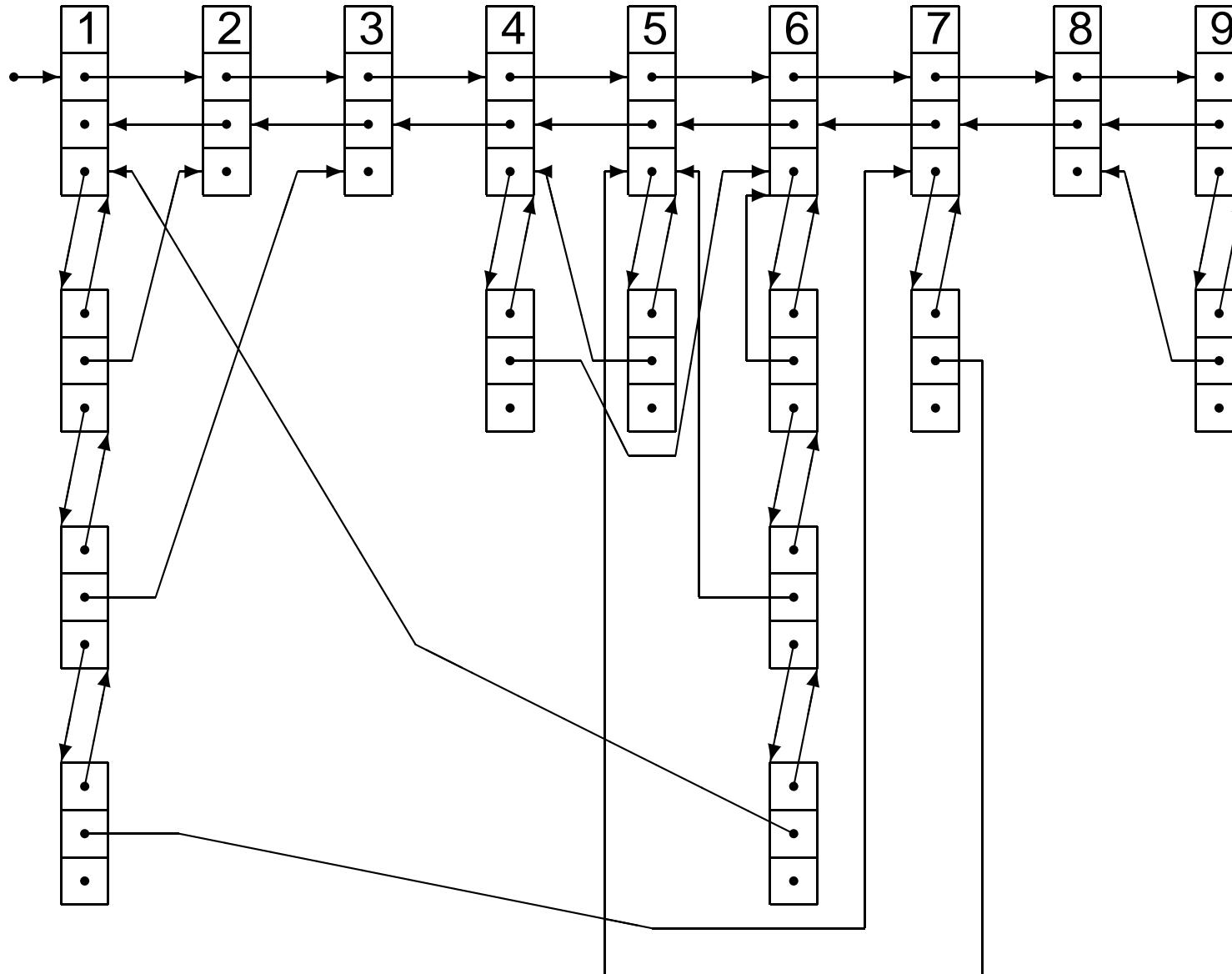
Ein Beispiel



Doppelt verkettete Kantenliste

- Die bei Adjazenzlisten fehlende **Dynamik kann erreicht werden**, indem man die **Knoten in einer doppelt verketteten Liste speichert**, anstatt sie in einem Feld fester Größe zu verwalten.
- **Jedes Listenelement** dieser Liste **enthält drei Verweise**, zwei davon **auf benachbarte Listenelemente** und **einen auf eine Kantenliste**, wie **bei Adjazenzlisten**.
- Jede **Kantenliste ist doppelt verkettet**; statt einer Knotennummer besitzt jedes **Kantenlistenelement einen Verweis auf ein Element der Knotenliste**.

Verkettete Kantenliste am Beispiel



Durchlaufen von Graphen

- Für manche Probleme ist es wichtig **Graphen vollständig zu traversieren**, d.h. **alle Knoten eines Graphen zu betrachten**.
- Fasst man die **Web-Seiten im Internet als Knoten** und die **Links auf diesen Seiten als Kanten** auf, so muss man beim **Suchen nach einem bestimmten Schlüsselwort alle Knoten dieses Web-Seiten-Graphen inspizieren**.
- Das Betrachten oder Inspizieren eines Knotens in einem Graphen nennt man auch oft **Besuchen** des Knotens.
- Manchmal ist es wichtig die **Knoten nach einer gewissen Systematik** zu besuchen.
- Wir werden im Folgenden die **Tiefensuche** und die **Breitensuche** als zwei **Spezialfälle eines allgemeinen Knotenbesuchsalgorithmus** kennen lernen.

Ein allgemeines Schema für das Traversieren

- **Im Gegensatz zu Bäumen** kann es **bei Graphen Zyklen** geben.
- Deswegen kann es **beim Traversieren** passieren, dass wir **bei einem schon einmal besuchten Knoten ankommen**.
- Aus diesem Grund müssen wir uns **die bereits besuchten Knoten in einer Tabelle** merken, um **Endlosschleifen zu vermeiden**.
- Da jeder Knoten mehrere Nachfolger haben kann, müssen wir darüber hinaus eine **Datenstruktur** verwenden, in der wir **die noch zu besuchenden Knoten ablegen**.
- Für die noch zu besuchenden Knoten verwenden wir eine **Kollektionsklasse**, die uns die folgenden Methoden bietet:
 - isEmpty
 - addElement
 - firstElement
 - removeFirstElement

Eine Abstrakte Java-Methode für das Traversieren

```
public void traverse(int node, NodeList l){
    boolean[] visited = new boolean[this.numberOfNodes];
    l.addElement(new Integer(node));
    visited[node] = true;

    while (!l.isEmpty()){
        Integer first = (Integer) l.firstElement();
        int i = first.intValue();
        System.out.println(i+": "+ l);
        l.removeFirstElement();
        for (int j = 0; j < this.numberOfNodes; j++){
            if (i != j && this.connected(i,j) && !visited[j]){
                l.addElement(new Integer(j)); // (*)
                visited[j] = true;
            }
        }
    }
}
```

Nachteile dieser Implementierung

- Bei dieser Version werden in der `for`-Schleife stets alle möglichen Knoten aufgezählt.
- Dies ist beispielsweise bei einer Implementierung mit **Adjazenzmatrizen** der Fall.
- Für diese Implementierung von Graphen, benötigt **ein Durchlauf daher stets $O(|V|^2)$** .
- Verwendet man hingegen **Adjazenzlisten**, so wird **jeder Knoten genau ein mal besucht**.
- Gleichzeitig wird auch **jede Kante höchstens einmal besucht**.
- Die **Laufzeit bei Adjazenzlistenimplementierung ist somit $O(|V| + |E|)$** .

Abstrakte Implementierung der Traversierung

Von der **konkreten Implementierung** kann man sehr gut durch **Verwendung einer Enumeration abstrahieren**:

```
public void traverse(int node, NodeList l){
    ...
    while (!l.isEmpty()){
        Integer first = (Integer) l.firstElement();
        ...
        l.removeFirstElement();

        Enumeration enum = this.successors(first.intValue());
        while (enum.hasMoreElements()) {
            Integer j = enum.nextElement();
            int jValue = j.intValue();
            if (i != jValue && !visited[jValue]){
                l.addElement(j);
                visited[jValue] = true;
            }
        }
        ...
    }
}
```

Konkrete Traversierungsverfahren durch geeignete Realisierung von `nodeList`

- Die Reihenfolge, in der die Knoten ausgegeben werden, hängt offensichtlich von der Datenstruktur `nodeList` ab, d. h. der Art, wie die Knoten darin abgelegt werden.
- Die Klasse `nodeList` ist daher ein Interface, das die entsprechenden Methoden definiert:

```
interface NodeList {  
    public Object firstElement();  
    public void removeFirstElement();  
    public void addElement(Object o);  
    public boolean isEmpty();  
    public String toString();  
}
```

Implementierung von NodeList durch eine Queue

```
class NodeListQueued extends Queue implements NodeList{
    NodeListQueued(){
        super();
    }

    public Object firstElement(){
        return super.top();
    }

    public void removeFirstElement(){
        super.dequeue();
    }

    public void addElement(Object o){
        super.enqueue(o);
    }
}
```

Implementierung von NodeList durch einen Stack

```
class NodeListStacked extends Stack implements NodeList{
    NodeListStacked(){
        super();
    }

    public Object firstElement(){
        return super.top();
    }

    public void removeFirstElement(){
        super.pop();
    }

    public void addElement(Object o){
        super.push(o);
    }
}
```

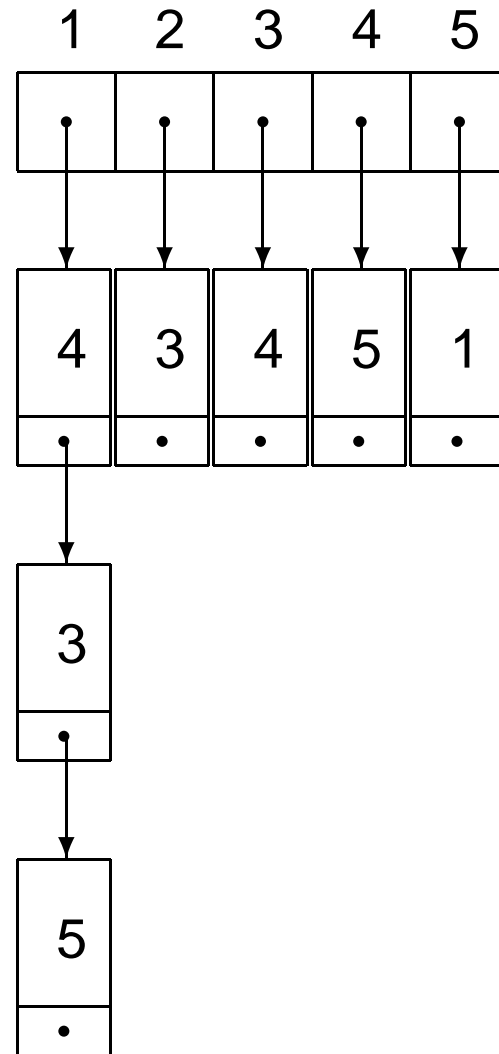
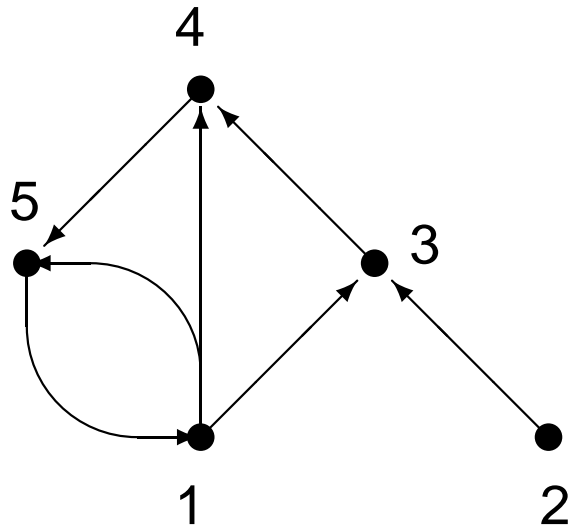
Tiefendurchlauf und Breitendurchlauf

- Verwendet man für die Knotenliste einen **Stack**, so ergibt sich ein **Tiefendurchlauf** durch den Graphen.
- Verwendet man hingegen eine **Queue**, so entspricht das Verhalten einem **Breitendurchlauf**

```
public void depthFirst(int node){
    NodeListStacked l = new NodeListStacked();
    this.traverse(node, l);
}
```

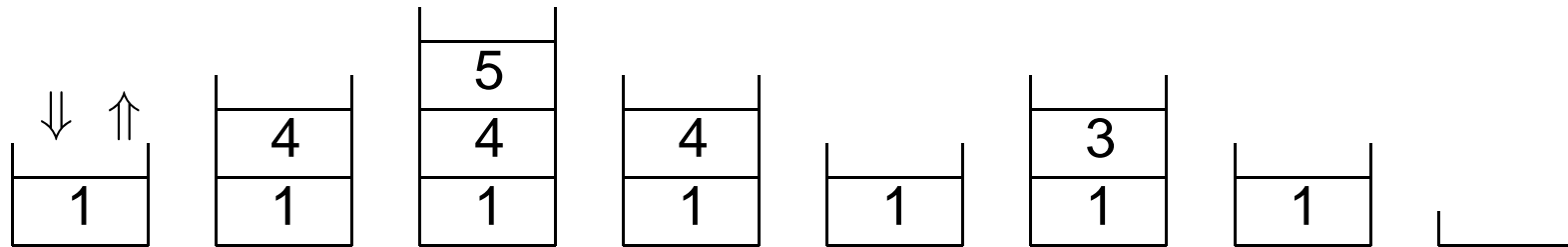
```
public void breadthFirst(int node){
    NodeListQueued l = new NodeListQueued();
    this.traverse(node, l);
}
```

Beispiel

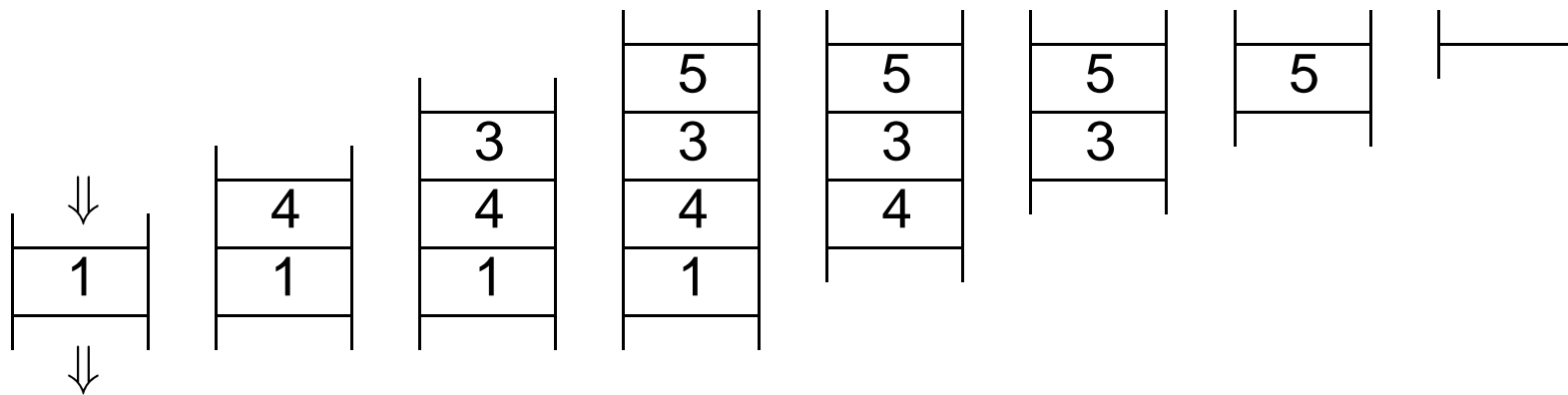


Breiten und Tiefendurchlauf

Durchlauf ab Knoten 1 mit **Stack** (Ausgabe: 1 → 4 → 5 → 3):



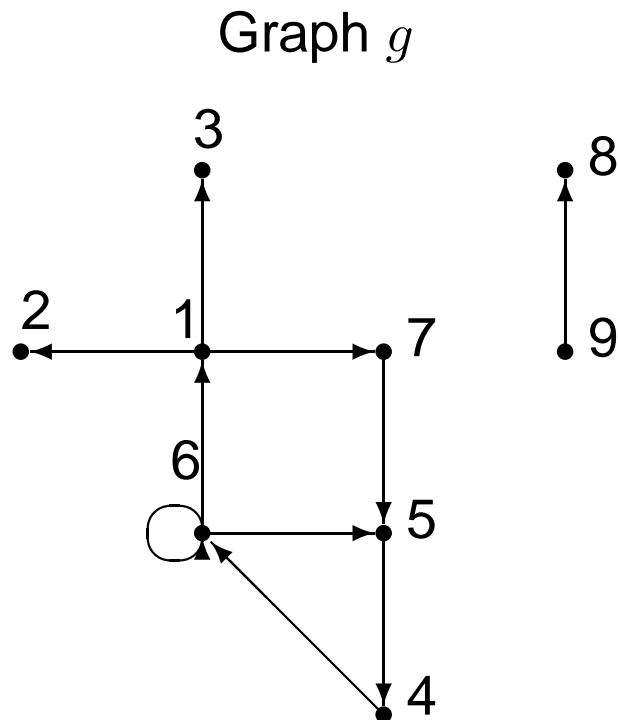
Durchlauf ab Knoten 1 mit **Queue** (Ausgabe: 1 → 4 → 3 → 5):



Kürzeste Wege in ungewichteten Graphen

Definition: Das **Single-Source-Shortest-Path-Problem** besteht darin, für einen Graph $G = (V, E)$ und einen Knoten $v \in V$ die kürzesten Pfade von v zu allen anderen Knoten in G zu bestimmen.

Beispiel:



Kürzeste Pfade ausgehend von Knoten 1

$1 \rightarrow 2$

$1 \rightarrow 3$

$1 \rightarrow 7$

$1 \rightarrow 7 \rightarrow 5$

$1 \rightarrow 7 \rightarrow 5 \rightarrow 4$

$1 \rightarrow 7 \rightarrow 5 \rightarrow 4 \rightarrow 6$

Lösung des Single-Source-Shortest-Path-Problems

- Der Knoten v ist **von sich selbst genau 0 Schritte** weit **entfernt**.
- Die **Nachbarn von v sind genau 1 Schritt** entfernt.
- Die **Knoten der Entfernung j sind alle Knoten, die von den $j - 1$ Schritt entfernten in genau einem Schritt erreicht werden** können.
- Also können wir dieses Problem durch einen **Breitendurchlauf** lösen.
- Anstelle der Besucht-Markierungen verwenden wir jedoch ein **Feld `distance`, um den Abstand der einzelnen Knoten abzulegen**.
- Dabei ist $|V| - 1$ **die Maximaldistanz eines Knoten von v** .
- Damit ist die **Komplexität bei Verwendung von Adjazenzlisten $O(|V| + |E|)$** .

Lösung des Single-Source-Shortest-Path-Problems

```
public void sssp(int node){
    NodeListQueued l = new NodeListQueued();
    int[] distance = new int[this.numberOfNodes];
    for (int i = 0; i < this.numberOfNodes; i++)
        distance[i] = this.numberOfNodes;

    l.addElement(new Integer(node));
    distance[node] = 0;
    while (!l.isEmpty()){
        int i = ((Integer) l.firstElement()).intValue();
        l.removeFirstElement();
        Enumeration enum = this.successors(i);
        while (enum.hasMoreElements()) {
            int j = (Integer enum.nextElement()).intValue();
            if (i != j && distance[j]==this.numberOfNodes){
                l.addElement(new Integer(j));
                distance[j] = distance[i]+1;
            }
        }
    }
    // Hier noch Ausgabe einfügen
}
```

Berechnung der Kürzesten Wege

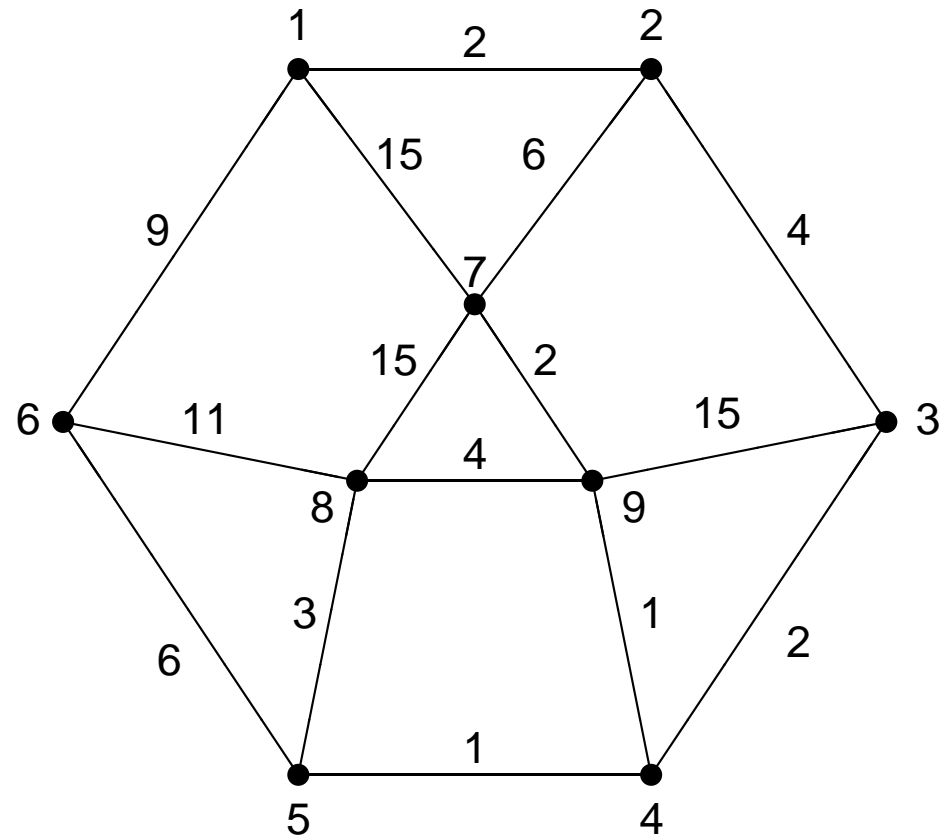
- Sind die **Distanzen** gegeben, kann man für einen **beliebigen v' Knoten** sehr **einfach den kürzesten Weg zu dem Ausgangsknoten v berechnen**.
- Hierzu gehen wir **von v' einfach zu dem Knoten v'' mit $(v'', v') \in E$, der den geringsten Abstand zu v hat**.
- Dann bestimmen wir den **kürzesten Weg von v'' zu v** .
- Sind wir **bei v angekommen**, so **stoppen wir**.
- So erhalten wir **rückwärts den kürzesten Pfad von v nach v'** .
- Allerdings kann man sich auch in **jeder Runde an der Stelle ($*$) den jeweiligen Vorgänger i merken** anhand von:

$$\text{predecessor}[j] = i;$$

Gewichtete Graphen

- **Gewichtete Graphen** unterscheiden sich von ungewichteten dadurch, dass **jede Kante mit einer reellen Zahl bewertet** ist.
- Diese Gewichte werden als **Distanzen** oder **Kosten für das Traversieren** interpretiert.
- Wir setzen im Folgenden voraus, dass diese Gewichte nicht negativ sind, d. h., dass es eine Abbildung $c : E \rightarrow \mathbb{R}_+^0$ gibt, die jeder Kante ein nicht-negatives Gewicht zuordnet.
- Das Problem, für einen Knoten die **kürzesten Wege zu allen anderen Knoten zu berechnen wird dadurch schwieriger**.
- Allerdings lassen sich die **Grundideen aus dem ungewichteten Fall übernehmen**.

Beispiel für einen gewichteten Graphen



Dijkstra's Algorithmus

Optimalitätsprinzip:

Für jeden kürzesten Weg $p = (v_0, v_1, \dots, v_k)$ von v_0 nach v_k ist jeder Teilweg $p' = (v_i, \dots, v_j)$, $0 \leq i < j \leq k$ ein kürzester Weg von v_i nach v_j .

Begründung:

1. Wäre dies nicht so, gäbe es also einen kürzeren Weg p'' von v_i nach v_j , so könnte auch in p der Teilweg p' durch p'' ersetzt werden und der entstehende Weg von v_0 nach v_k wäre kürzer als p .
2. Dies ist aber ein Widerspruch zu der Annahme, dass p ein kürzester Weg von v_0 nach v_k ist.

Folgerung (1)

Damit können wir länger werdende kürzeste Wege durch Hinzunahme einzelner Kanten zu bereits bekannten kürzesten Wegen mit folgender **Invariante** berechnen:

1. Für alle kürzesten Wege $sp(s, v)$ und Kanten (v, v') gilt:

$$c(sp(s, v)) + c((v, v')) \geq c(sp(s, v')).$$

2. Für wenigstens einen kürzesten Weg $sp(s, v)$ und eine Kante (v, v') gilt:

$$c(sp(s, v)) + c((v, v')) = c(sp(s, v')).$$

Folgerung (2)

- Sei $p = (v_0, v_1, \dots, v_k)$ ein Weg von v_0 nach v_k ist.
- Sei p'' ein kürzerer Weg von v_i nach v_j als der entsprechende Teilweg in p .
- Dann können wir in p den Teilweg von v_i nach v_j durch p'' ersetzen, um einen kürzeren Weg p' von v_0 nach v_k zu erhalten.

Idee des Verfahrens von Dijkstra

- **Anfangs** ist die **Entfernung** $d(v)$ **aller von s verschiedener Knoten** ∞ .
- Die **Entfernung von s von sich selbst** ist natürlich 0.
- Wir betrachten eine **Menge PQ von Knoten-Entfernungs-Paaren** $(v, d(v))$, die wir **mit $\{(s, 0)\}$ initialisieren**.
- Dann wird PQ nach dem Prinzip **„Knoten mit kürzester Distanz von s zuerst“** schrittweise bearbeitet, bis PQ leer ist:
 1. Entferne Knoten v aus PQ mit minimaler Distanz $d(v)$ von s ; $d(v)$ ist der kürzeste Distanz von s nach v .
 2. Für jeden Knoten $w \in V$ mit $(v, w) \in E$ verfare wie folgt:
 - (a) Falls $(w, d(w)) \in PQ$, ersetze $(w, d(w))$ durch $(w, \min\{d(w), d(v) + c(v, w)\})$.
 - (b) Falls w nicht in PQ enthalten ist, füge $(w, (d(v) + c(v, w)))$ in PQ ein.

Benötigte Datenstrukturen

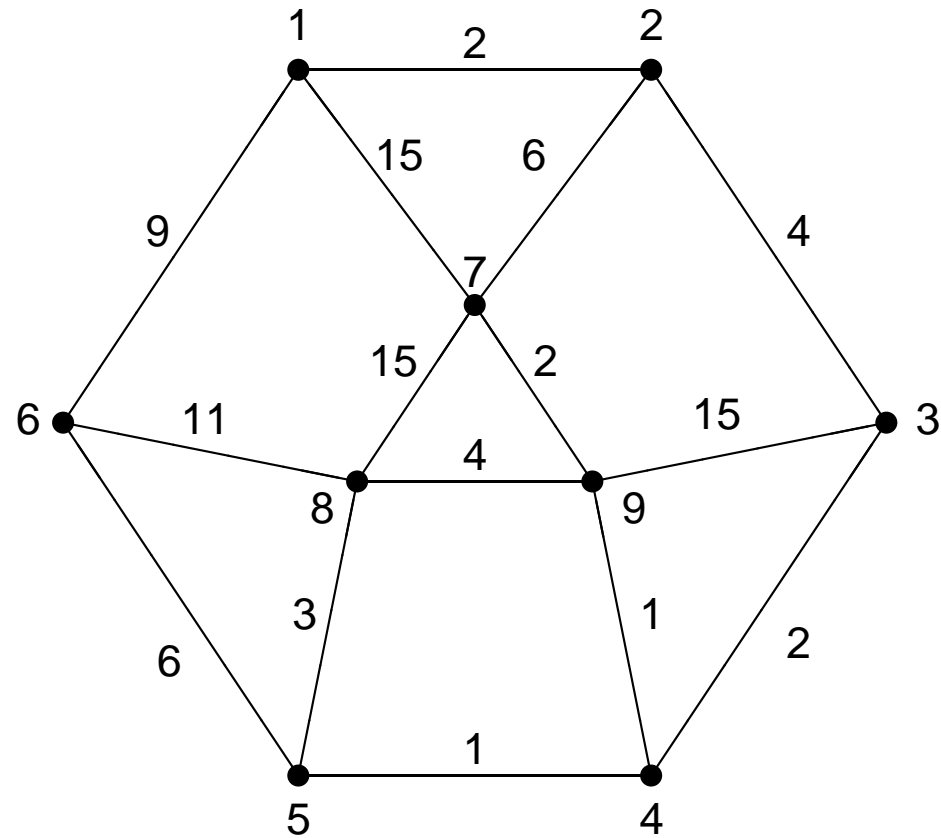
- Wir merken uns für jeden Knoten v die bisher berechnete, **vorläufige Entfernung** $d(v)$ zum Anfangsknoten s .
- Weiter speichern wir den **Vorgänger von v** auf dem bisher berechneten vorläufig kürzesten Weg.
- Weiter benötigen wir eine Datenstruktur, um die noch zu bearbeitenden Knoten zu speichern. Dazu verwenden wir eine **Priority Queue**.

Priority Queues (Vorrangwarteschlangen)

Als **Priority Queue** bezeichnet man eine Datenstruktur zur Speicherung einer Menge von Elementen, für die eine Ordnung (Prioritätsordnung) definiert ist, so dass folgende Operationen ausführbar sind:

- **Initialisieren** (der leeren Struktur),
- **Einfügen** eines Elementes,
- **Minimum suchen**,
- **Minimum entfernen**,
- **Herabsetzen der Priorität** eines Schlüssels.

Ein Beispiel



Startknoten: 1.

Ablauf des Verfahrens

Eintrag in $PQ \hat{=} (\text{Nr.}, \text{Entfernung}, \text{Vorgänger})$:

(1,0,1)

(2,2,1), (6,9,1), (7,15,1)

(6,9,1), (7,8,2), (3,6,2)

(6,9,1), (7,8,2), (4,8,3), (9,21,3)

(6,9,1), (4,8,3), (9,10,7), (8,23,7)

(6,9,1), (8,23,7), (9,9,4), (5,9,4)

(9,9,4), (5,9,4), (8,20,6)

(5,9,4), (8,13,9)

(8,12,5)

\emptyset

Implementierungen von Priority Queues

- Offensichtlich hängt die **Rechenzeit von Dijkstra's Algorithmus** von der **Implementierung der Priority Queue** ab.
- Wenn wir eine **lineare Liste zur Speicherung der Priority Queue** PQ verwenden, so benötigen einzelne Operationen, wie z.B. das **Auffinden des Minimums** das **Einfügen** oder das **Herabsetzen der Priorität** $O(|V|)$ **Schritte**.
- Auch wenn wir die Elemente in der **Liste sortieren**, benötigen wir noch **Linearzeit für das Herabsetzen der Priorität**.
- Da wir $O(|V|)$ **Schleifendurchläufe** auszuführen haben, ist der **Gesamtaufwand** $O(|V|^2)$.
- Eine bessere Datenstruktur für Dijkstra's Algorithmus ist ein so genannter **Fibonacci-Heap**.
- Damit erreicht man eine **Gesamtlaufzeit von** $O(|E| + |V| \log |V|)$.

The End

Danke für die Aufmerksamkeit!