

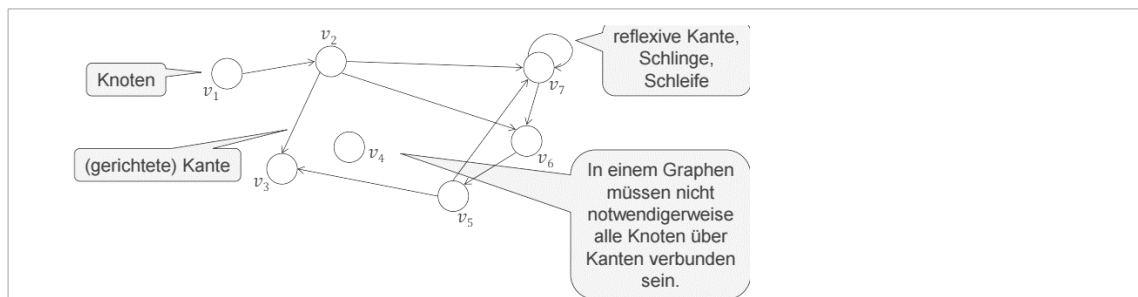
Graphen und Graphalgorithmen

Objekte und Beziehungen

- Gegeben: Menge von Objekten zusammen mit einer Beziehung (Relation) auf diesen Objekten
- Beispiel: Objekt (Personen), Beziehung (Person A kennt Person B)

Gerichtete Graphen

- Ein (gerichteter) Graph (auch: Digraph, engl. directed graph) ist ein Paar $G = (V, E)$, wobei gilt:
 - V ist eine endliche, nichtleere Menge (die Elemente werden Knoten genannt),
 - E ist eine zweistellige Relation auf V : $E \subseteq V \times V$ (die Elemente heißen (gerichtete) Kanten (engl. edge)).
- Eine (gerichtete) Kante ist also ein Paar von Knoten v, w ; (gerichtete) Kante (v, w) wird graphisch wie folgt dargestellt:
- Sprechweisen:
 - Kante (v, w) ist inzident mit v und w .
 - □ Knoten v und w sind benachbart, adjazent oder Nachbarn
- Grafische Darstellung eines gerichteten Graphen G_1 :



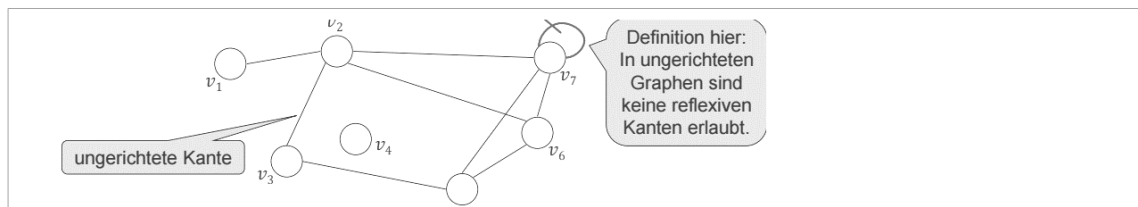
- Mengendarstellung von G_1 :

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$$

$$E = \{(v_1, v_2), (v_2, v_3), (v_2, v_6), (v_2, v_7), (v_5, v_3), (v_5, v_7), (v_6, v_5), (v_7, v_6), (v_7, v_7)\}$$

Ungerichtete Graphen

- Ein ungerichteter Graph ist ein Graph $G = (V, E)$, für den gilt: $\forall v_i, v_j \in V: (v_i, v_j) \in E \Rightarrow (v_j, v_i) \in E$.
- Bei ungerichteten Graphen ist E also symmetrisch.
- Eine ungerichtete Kante ist ein Paar von Knoten v, w .
- In der grafischen Darstellung gehört zu jedem Pfeil ein Pfeil in die Gegenrichtung. Statt Pfeilpaar zeichnet man eine spitzenlose Linie.
- Kante (v, w) wird graphisch wie folgt dargestellt:
- In der Mengenschreibweise schreibt man $[v_i, v_j]$ für beide Paare
- Grafische Darstellung eines ungerichteten Graphen G_2



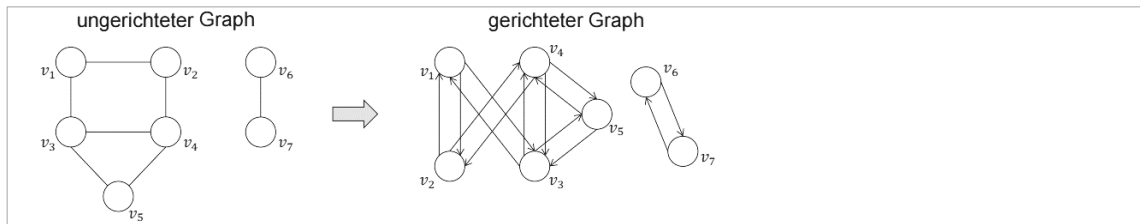
- Mengendarstellung von G_2 :

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$$

$$E = \{[v_1, v_2], [v_2, v_3], [v_2, v_6], [v_2, v_7], [v_5, v_3], [v_5, v_7], [v_6, v_5], [v_7, v_6]\}$$

Ungerichtete Graphen

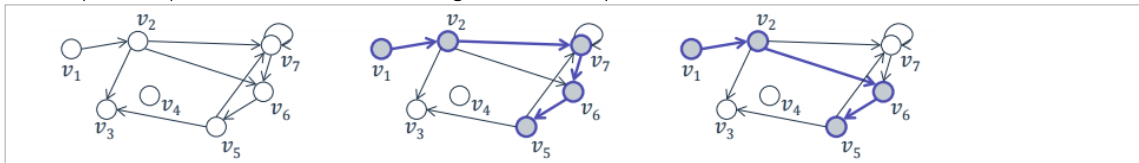
Da eine symmetrische Beziehung ein Spezialfall einer Beziehung ist, lässt sich ein ungerichteter Graph immer durch einen gerichteten Graphen darstellen:



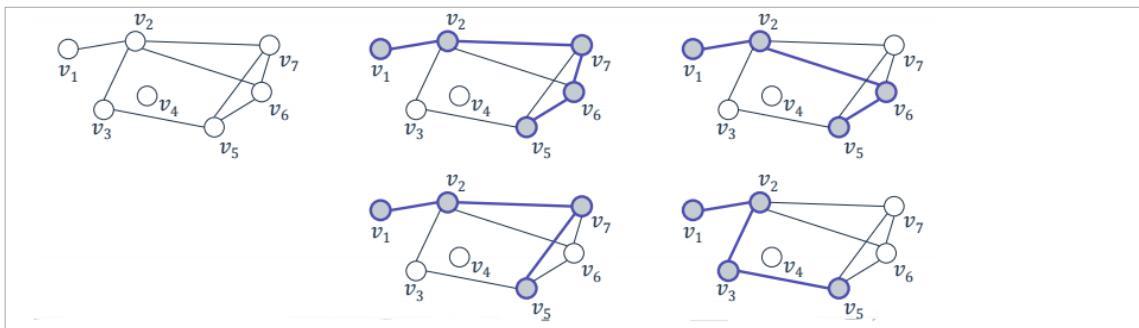
- Beispiel zeigt auch, dass Platzierung von Knoten und Kanten in der graphischen Darstellung keine Rolle spielt.
- Es gibt also verschiedene Darstellungen desselben Graphen.

Pfade in Graphen

- Ein Pfad (Weg, Kantenzug) von v nach w ist eine endliche Folge von Knoten $v = v_1, v_2, \dots, v_n = w$, so dass für $1 \leq i \leq n-1$ gilt $(v_i, v_{i+1}) \in E$.
- Sprechweisen:
 - Der Pfad verbindet v und w .
 - w ist von v aus erreichbar.
- Hinweis: In unserer Graph-Definition kann es keine „parallelen Kanten (mit gleicher Richtung)“ zwischen zwei Knoten geben (sonst sog. Pseudograph).
- Die Länge l eines Pfades ist die Anzahl der Kanten im Pfad, also $l = n - 1$. Ein einzelner Knoten ist demnach ein Pfad der Länge 0.
- Ein Pfad heißt einfach, wenn alle Knoten im Pfad paarweise verschieden sind.
- Zwei (einfache) Pfade von v_1 nach v_5 im gerichteten Graphen:

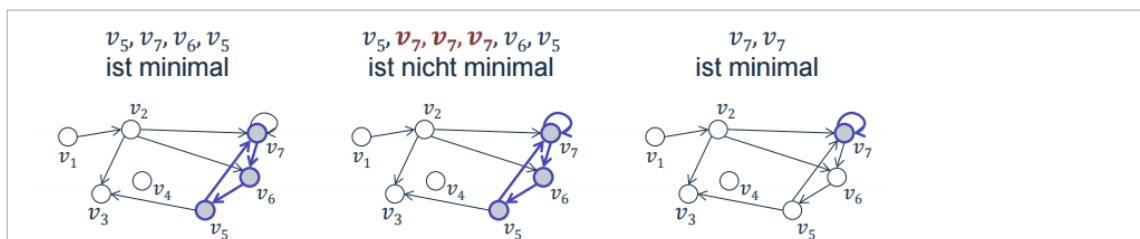


- Vier (einfache) Pfade von v_1 nach v_5 im ungerichteten Graphen:



Zyklen in Graphen

- In einem gerichteten Graphen heißt ein Pfad der Länge $l \geq 1$ von vv nach vv ein Zyklus (von v nach v).
- Ein Zyklus von v nach v heißt minimaler Zyklus (von v nach v), wenn außer v kein anderer Knoten mehr als einmal vorkommt.



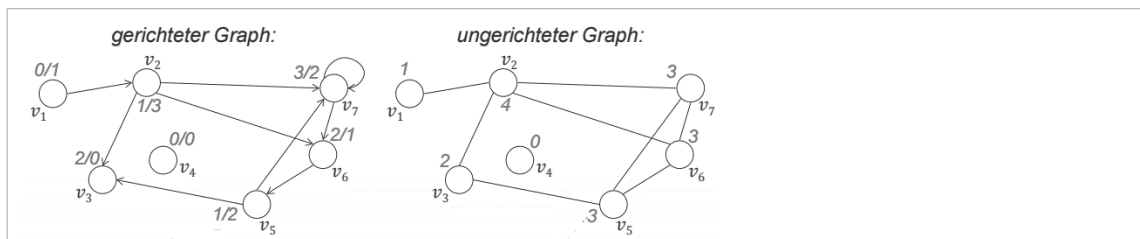
- Bei ungerichteten Graphen muss ein Zyklus mindestens drei Knoten enthalten.

Vorgänger und Nachfolger

- Ist $(v, w) \in E$, so heißt
 - v direkter Vorgänger (bei gerichteten Graphen auch: Elternknoten) von w ,
 - w direkter Nachfolger (bei gerichteten Graphen auch: Kind) von v .
- Falls ein Pfad von v_i nach v_j führt, so heißt
 - v_i Vorgänger (bei gerichteten Graphen auch: Vorfahr) von v_j ,
 - v_j Nachfolger (bei gerichteten Graphen auch: Nachkomme) von v_i .
- v_2 und v_7 sind direkte Vorgänger (Elternknoten) von v_6
- v_3, v_6 und v_7 sind direkte Nachfolger (Kinder) von v_2
- v_1 ist Vorgänger (Vorfahr) aller anderer Knoten (bis auf v_4)
- v_6 ist Nachfolger (Nachkomme) von v_1

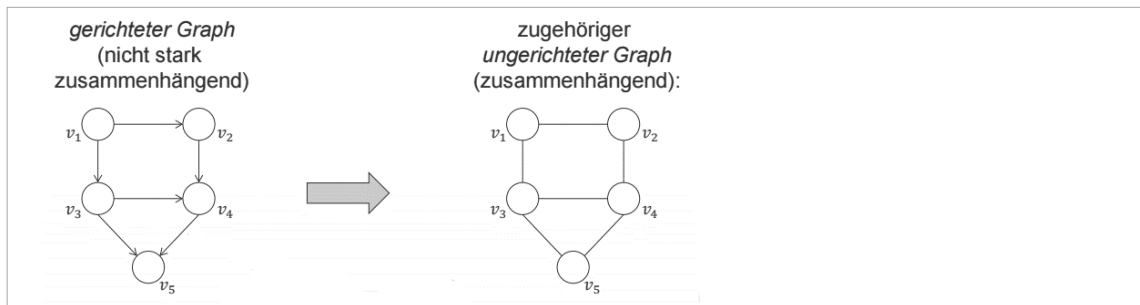
Grad von Knoten und Graph

- Die Anzahl der direkten Vorgänger eines Knotens heißt Eingangsgrad des Knotens. Die Anzahl der direkten Nachfolger eines Knotens heißt Ausgangsgrad des Knotens.
- Bei ungerichteten Graphen spricht man vom Grad des Knotens.
- Ein Knoten mit Grad 0/* heißt Quelle
- Ein Knoten mit Grad */0 heißt Senke.



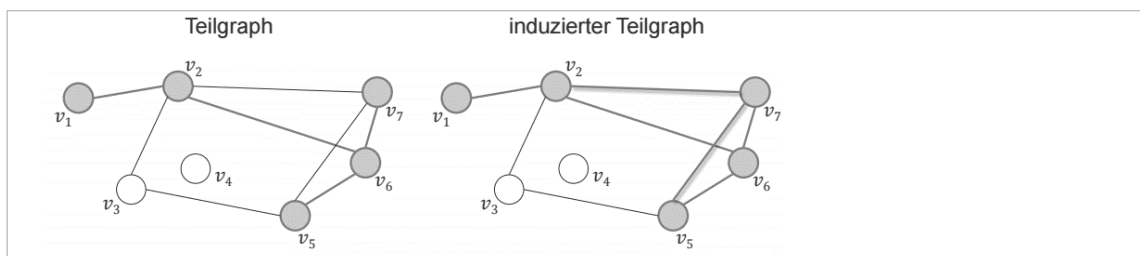
Zusammenhang in Graphen

- Ein gerichteter Graph $G = (V, E)$ heißt stark zusammenhängend (oder: stark verbunden), wenn es für alle $v, w \in V$ einen Pfad von v nach w gibt. Bei ungerichteten Graphen lässt man „stark“ weg.
- anschaulich: Jeder Knoten ist von jedem anderen Knoten aus erreichbar.
- Ein gerichteter Graph heißt schwach zusammenhängend, wenn der zugehörige ungerichtete Graph (der durch Hinzunahme aller Rückwärtskanten entsteht) zusammenhängend ist.



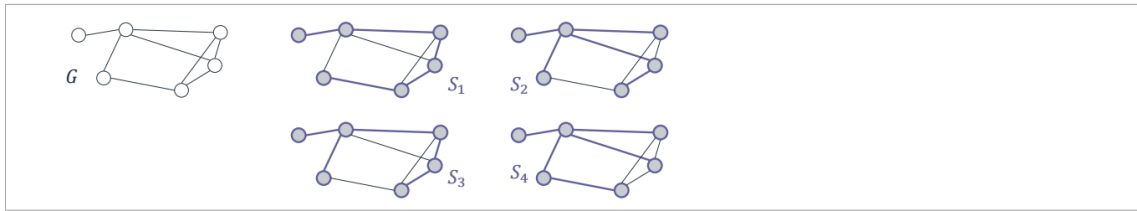
Teilgraphen

- Ein Graph $G' = (V', E')$ ist ein Teilgraph eines Graphen $G = (V, E)$ genau dann, wenn $V' \subseteq V$ und $E' \subseteq E$.
- Ein Graph $G' = (V', E')$ ist ein induzierter Teilgraph eines Graphen $G = (V, E)$ genau dann, wenn $V' \subseteq V$ und $E' = \{(v, w) \in E \mid v, w \in V'\}$ anschaulich: Wähle einige Knoten aus G aus; in E' werden alle Kanten übernommen, die diese ausgewählten Knoten verbinden.



Teilgraphen

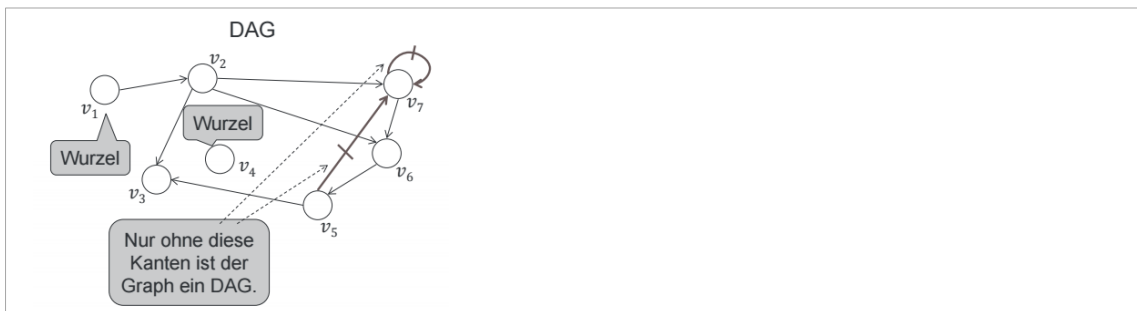
- Sind G ein Graph und S ein zyklenfreier Teilgraph von G , der alle Knoten von G enthält, und sind G und S beide zusammenhängend, dann heißt S Spannbaum (auch: aufspannender Baum) von G .



- Ist G ein Baum (siehe frühere LE, Baum ist spezieller ungerichteter Graph), dann gibt es zwischen je zwei verschiedenen Knoten genau einen einfachen Pfad.

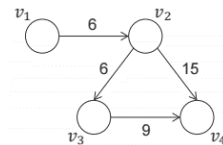
Gerichteter azyklischer Graph, Wurzel

Ein Knoten v eines gerichteten azyklischen Graphen (DAG) heißt Wurzel, falls es keine auf ihn gerichteten Kanten gibt.



Bewertete/gewichtete Graphen

- Ein Graph $G = (V, E)$ wird zu einem bewerteten/gewichteten Graphen, indem man eine Gewichtsfunktion $c: E \rightarrow \mathbb{N}$ bzw. $c: E \rightarrow \mathbb{R}^+$ ergänzt, die jeder Kante $e \in E$ ein positives Gewicht $c(e)$ zuordnet.
- Die Kosten (auch: bewertete Länge) $c(p)$ eines Pfades p definiert man in gewichteten Graphen als die Summe der Gewichte seiner Kanten: für $p = (v = v_1, v_2, \dots, v_n = w)$ ist $c(p) = \sum_{i=1}^{n-1} c((v_i, v_{i+1}))$
- Beispiel: Städte mit Straßennetz und Entfernungen, Reisezeiten oder Transportkosten.
- Beispielgraph:
 - Kosten auf jedem Pfad von Knoten v_1 zu Knoten v_4 sind immer 21,
 - auch über den „Umweg“ mit Knoten v_3 .

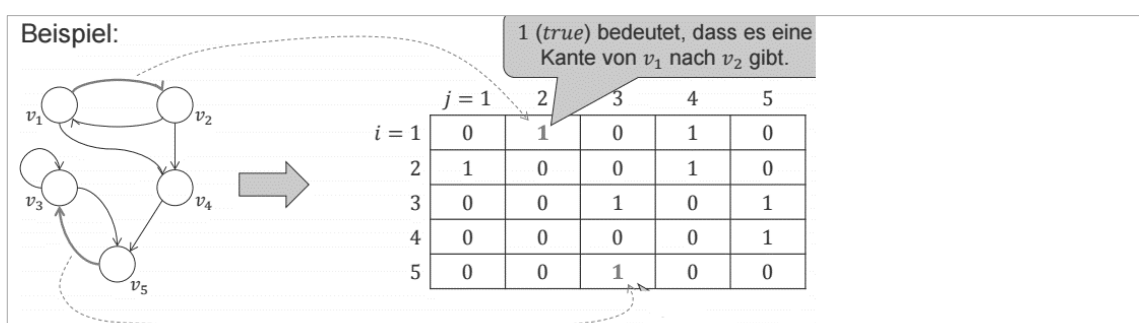


Darstellungen von Graphen

- Bisher betrachtete Darstellungsformen für Graphen:
 - graphische Darstellung: gute Visualisierung für Menschen
 - Mengendarstellung: gut für mathematische Operationen
- Nun gesucht: geeignete Datenstruktur, mit der ein Graph im Rechner repräsentiert werden kann.
- Anforderungen: Datenstruktur soll ...
 - ... möglichst effizient unterstützen:
 - Feststellung, ob Kante zwischen zwei Knoten existiert
 - Bestimmung aller Nachfolger eines Knoten
 - Graphen-Durchlauf
 - Änderungen am Graphen
 - ... möglich speichereffizient sein.

Adjazenzmatrix

- Sei $V = \{v_1, \dots, v_n\}$.
- Die Adjazenzmatrix für $G = (V, E)$ ist eine boolesche $n \times n$ -Matrix A mit $A_{ij} = \begin{cases} \text{true, falls } (v_i, v_j) \in E \\ \text{false, sonst} \end{cases}$
- Meist werden **true** und **false** durch 1 und 0 dargestellt.



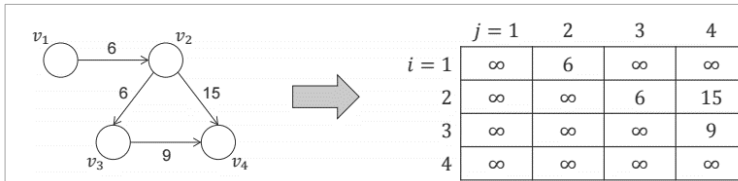
Adjazenzmatrix

- Bei ungerichteten Graphen sind Adjazenzmatrizen symmetrisch, es reicht die Speicherung einer Dreiecksmatrix.
- Graphen ohne Schlingen haben in der Diagonalen stets false.

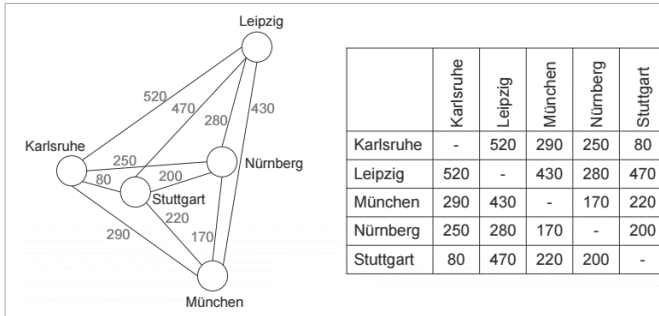
```
boolean[][] edges = {
    {false, true, false, true, false},
    {true, false, false, true, false},
    {false, false, true, false, true},
    {false, false, false, false, true},
    {false, false, true, false, false}
}
```

Adjazenzmatrix bei bewerteten/gewichteten Graphen

- Bei bewerteten/gewichteten Graphen speichert man statt true/false die Kantengewichte in der Matrix.
- Fehlende Kanten stellt man mit einem Spezialwert für Unendlich dar: ∞ oder im Programm-Code z.B. Integer.MAX_VALUE/3 (damit man sicher ggf. noch zwei solche Werte addieren kann).



- Entfernungstabelle ist die bekannteste Form der Adjazenzmatrix

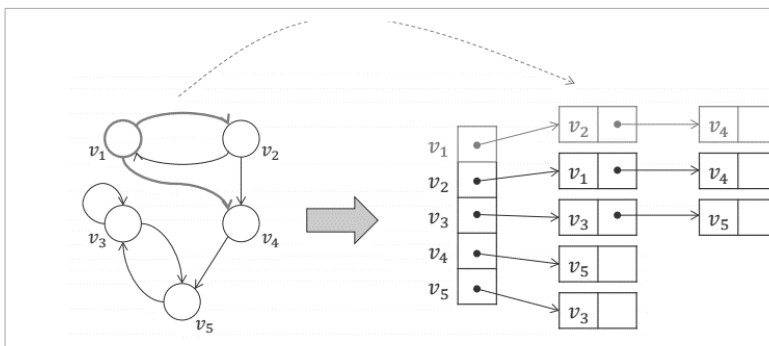


Eigenschaften von Adjazenzmatrizen

- Vorteile:
 - Es ist mit Laufzeit $O(1)$ feststellbar, ob Kante von v nach w existiert.
 - Manche Graphenoperationen lassen sich als Matrizenoperationen darstellen.
- Nachteile (fallen vor allem bei spärlich besetzten Graphen, also bei Graphen mit wenigen Kanten in's Gewicht):
 - Hoher Platzbedarf von $O(n^2)$: Speicherverschwendung, wenn Anzahl der Kanten relativ klein gegenüber dem Quadrat der Knotenanzahl.
 - Auffinden aller k Nachfolger eines Knotens benötigt $O(n)$ Zeit (ebenfalls relativ ungünstig).
 - Initialisierung der Matrix benötigt Laufzeit von $\Theta(n^2)$.
- Also: diese Repräsentation wählen, wenn Tests auf das Vorhandensein von Kanten häufig vorkommen.

Adjazenzlisten

- Man verwaltet für jeden der n Knoten eine Liste seiner (Nachfolger-) Nachbarknoten.
- Über ein Array der Länge $n = |V|$ ist jede Liste direkt zugänglich.



Eigenschaften von Adjazenzlisten

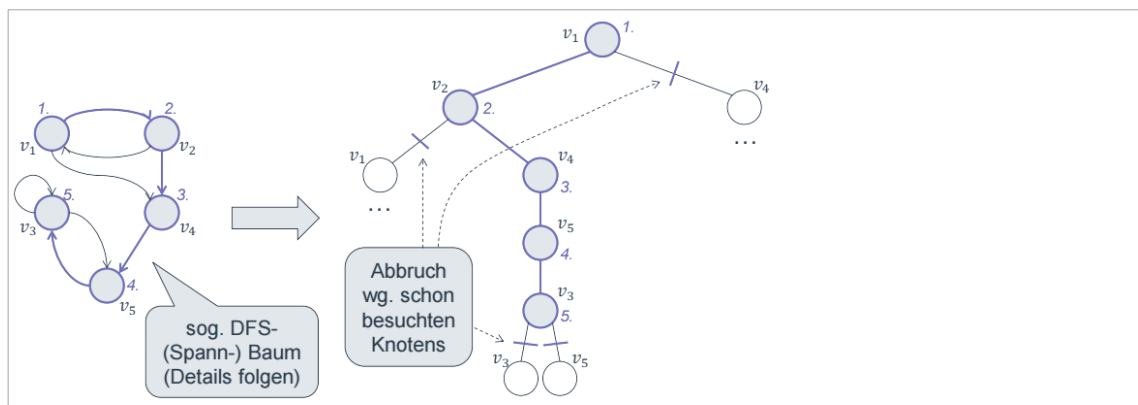
- Vorteile:
 - Geringerer Platzbedarf von $O(|V| + |E|)$.
 - Alle k Nachfolger eines Knotens sind in Zeit $O(k)$ erreichbar.
- Nachteile:
 - Test, ob v und w benachbart sind, kann nicht mehr in konstanter Zeit durchgeführt werden, da Adjazenzliste von v durchlaufen werden muss, um auf das Vorhandensein von w zu prüfen (und ggf. auch umgekehrt).
- Erweiterung: inverse Adjazenzlisten
 - Falls zu Knoten Vorgängerknoten aufgesucht werden müssen, kann man noch inverse Adjazenzlisten verwalten, die zu jedem Knoten eine Liste seiner Vorgänger enthalten.

Motivation Graphdurchlauf

- Bisher war Durchlaufen von Eingabedaten einfach:
 - Sequentielle Vorgehensweise
 - Betrachtung eines Elements der Eingabedaten nach dem anderen
- Jetzt: Durchlauf von Graphen
 - Gegeben: Graph G
 - Gesucht: Durchlauf/Traversierung des Graphen, der/die jeden Knoten einmal besucht.
 - Aufgabe, die in vielen Graph-Algorithmen bewältigt werden muss.
- Strategien:
 - Tiefensuche (auch: Tiefendurchlauf) (engl. depth first search (DFS) / depth first traversal)
 - Breitensuche (auch: Breitendurchlauf) (engl. breadth first search (BFS) / breath first traversal)

Tiefensuche (engl. depth-first-search – DFS)

Ein Durchlauf eines Graphen G per Tiefensuche (engl. depth-first-search – DFS) entspricht einem Preorder-Durchlauf von G , der jeweils in einem bereits besuchten Knoten von G abgebrochen wird.



- Analogie: Besuch eines Museums mit vielen Gängen, wobei man jeden Gang ablaufen möchte.
- Verfahren:
 - Man läuft in das Museum hinein.
 - Man betritt einen neuen Gang, sobald er sich öffnet. Wenn sich mehrere Gänge gleichzeitig öffnen, wählt man einen (meist den linken)
 - und hinterlässt an der Kreuzung einen Kieselstein.
 - Wenn man an eine Kreuzung stößt, in der bereits ein Kiesel liegt, kehrt man um und geht d. gleichen Weg zurück bis zur vorhergehenden Kreuzung (Gänge u. Kreuzungen können mehrfach betreten werden).
 - Wenn von dieser noch ein unausprobierter Gang abgeht, wird dieser erforscht.
 - Gibt es keinen unausprobieren Gang mehr, muss weiter zurück gegangen werden.
- In der griechischen Mythologie benutzte Ariadne ein Garnknäuel, um Theseus die Rückkehr aus dem Labyrinth zu ermöglichen, in dem er den Minotaurus tötete (sog. Ariadne-Faden).
- Siehe Rücksetzverfahren („backtracking“).
- Algorithmus (rekursiv):

```
Algorithmus dfs(v):  
markiere v als besucht;  
für jeden Nachfolger vi von v führe aus: {  
    Wenn vi noch nicht besucht wurde,  
    dann dfs(vi);  
}
```

Tiefensuche

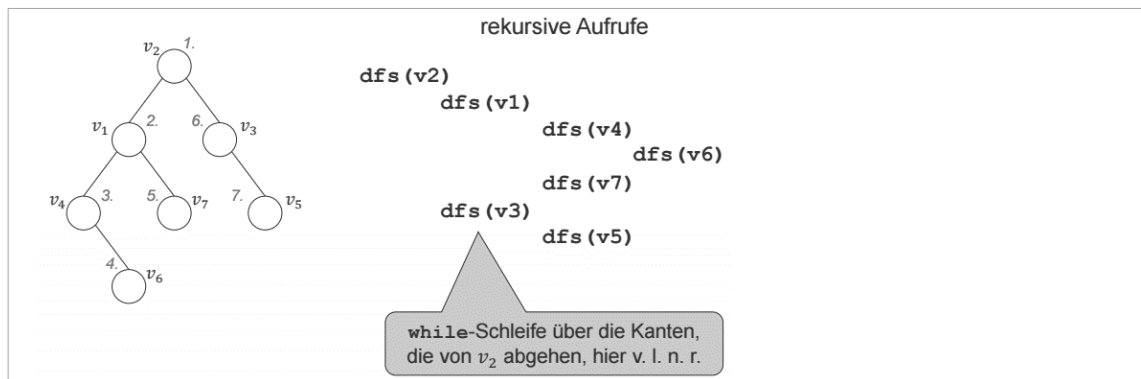
- Z. B. in Array merken, ob *v* besucht wurde: `boolean[] visited`
- Erinnerung: Der Graph muss nicht zusammenhängend sein. Daher wie folgt um alle Knoten zu besuchen:

```
markiere alle Knoten als unbesucht;  
für jeden Knoten v des Graphen führe aus: {  
  wenn v noch nicht besucht wurde,  
  dann dfs(v);  
}
```

Tiefensuche: rekursive Implementierung

```
void dfs(Node v) {  
    // markiere v als besucht  
    v.mark();  
    // hole alle Kanten zu Nachfolgern;  
    // verwaltet als Adjazenzliste  
    Iterator<Edge> iter = v.getEdges();  
    while (iter.hasNext()) {  
        Edge e = (Edge) iter.next();  
        if (e.target.isUnmarked()) {  
            dfs(e.target);  
        }  
    }  
}
```

- Jeder Knoten des zusammenhängenden Teilgraphen wird erreicht: sonst gäbe es einen Knoten ohne Markierung, der über eine Kante mit einem markierten Knoten verbunden ist.
- Das kann nicht vorkommen.
- Aufwand $O(|V| + |E|)$:
 - Jede der $|E|$ Kanten wird (höchstens) von beiden Seiten betrachtet.
 - Zusätzlich gibt es ggf. isolierte Knoten in V .
- Rekursion am Beispiel: Besuch der Nachfolger von links nach rechts (v. l. n. r.)



Anwendungsbeispiele für Tiefensuche

- Nachfolgend: Anwendungsbeispiele für eine Tiefensuche
- Dazu erforderlich: Aktionen auf besuchten Knoten.
- Betrachtete Anwendungsbeispiele:
 - Knoten eines Unterbaums eines Knotens zählen.
 - Inorder-/Preorder-/Postorder-Durchlauf eines Ausdruckbaums und dessen Ausgabe in Infix-/Präfix-/Postfix-Form.

Tiefensuche mit Nutzarbeit: rekursive Implementierung

- Graph wird besucht, um eine bestimmte Aufgabe zu erledigen.
- Abhängig von der Aufgabe sind zu konkretisieren:
 - (1) Aktion(en) beim ersten Betreten des Knotens v
 - (2) Aktion(en) zwischen den Besuchen der Nachfolger des Knotens v
 - (3) Aktion(en) beim Abstieg: im Falle erfüllter Bedingung und immer
 - (4) Aktion(en) nach Besuch aller Nachfolger des Knotens v

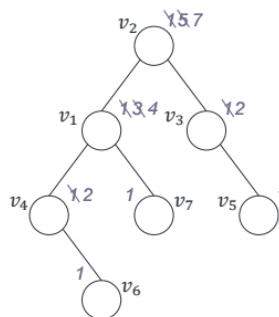
Tiefensuche mit Nutzarbeit: rekursive Implementierung

```
void dfs(Node v) {
    v.mark();
    // preNodeWork(v); (1)
    Iterator iter = v.getEdges();
    while (iter.hasNext()) {
        Edge e = (Edge) iter.next();
        if (e.target.isUnmarked()) {
            // inNodeWork(v); (2)
            dfs(e.target);
            // edgeWorkIf(e); (3)
        }
        // edgeWorkAlways(e); (3)
    }
    // afterNodeWork(v); (4)
}
```

Knotenanzahl des Unterbaums per Tiefensuche bestimmen

- Gegeben: Baum G
- Gesucht: Methode, die für jeden Knoten v die Anzahl der Knoten des Unterbaums bestimmt, der v als Wurzel hat.
- G ist ein Baum: dort gibt es keine bereits markierten Nachfolger.

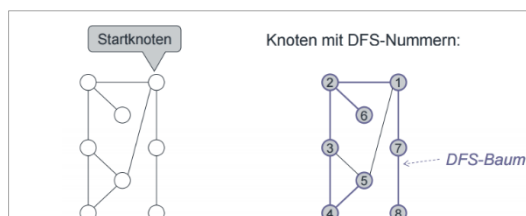
```
void dfsCountSubtreeNodes(Node v) {
    v.mark();
    // preNodeWork(v):
    v.counter = 1;
    Iterator iter = v.getEdges();
    while (iter.hasNext()) {
        Edge e = (Edge) iter.next();
        if (e.target.isUnmarked()) {
            dfsCountSubtreeNodes(e.target);
        }
        // edgeWorkAlways(e):
        v.counter += e.target.counter;
    }
}
```



```
v2.counter = 1;
v1.counter = 1;
v4.counter = 1;
v6.counter = 1;
v4.counter += v6.counter; // = 2
v1.counter += v4.counter; // = 3
v7.counter = 1;
v1.counter += v7.counter; // = 4
v2.counter += v1.counter; // = 5
v3.counter = 1;
v5.counter = 1;
v3.counter += v5.counter; // = 2
v2.counter += v3.counter; // = 7
```

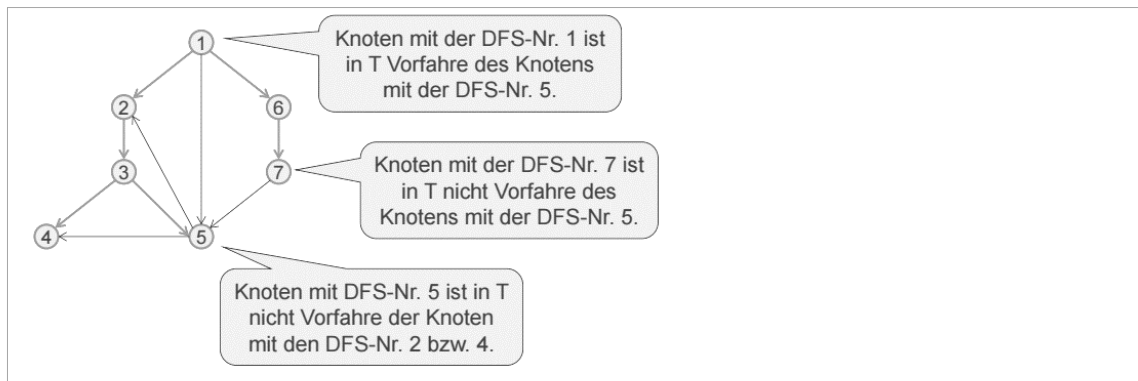
DFS-Nummerierung

- Die Tiefensuche durchläuft die Knoten eines Graphen in einer bestimmten Reihenfolge.
- Wenn jedem Knoten die Position in dieser Reihenfolge zugeordnet wird, ist das eine DFS-Nummerierung.
- DFS-Nummern sind für viele Graph-Algorithmen nützlich.
- Umsetzung mit DFS-Algorithmus:
 - statische Variable dfsPos = 1 vor dem Start initialisieren
 - preNodeWork(v): v.dfsNr = dfsPos++ DFS-Nummer des Knotens wird auf den Wert der statischen Variablen gesetzt; diese wird inkrementiert.
- Prinzip:
 - dicke Kanten verbinden Knoten mit denjenigen Nachfolgern, die die DFS-Nummerierung als noch unmarkiert vorgefunden hat.
 - edgeWorkIf baut aus diesen Kanten den sog. DFS-Baum (oder Tiefensuchbaum, engl.: DFS-tree) auf, einen Spannbaum.



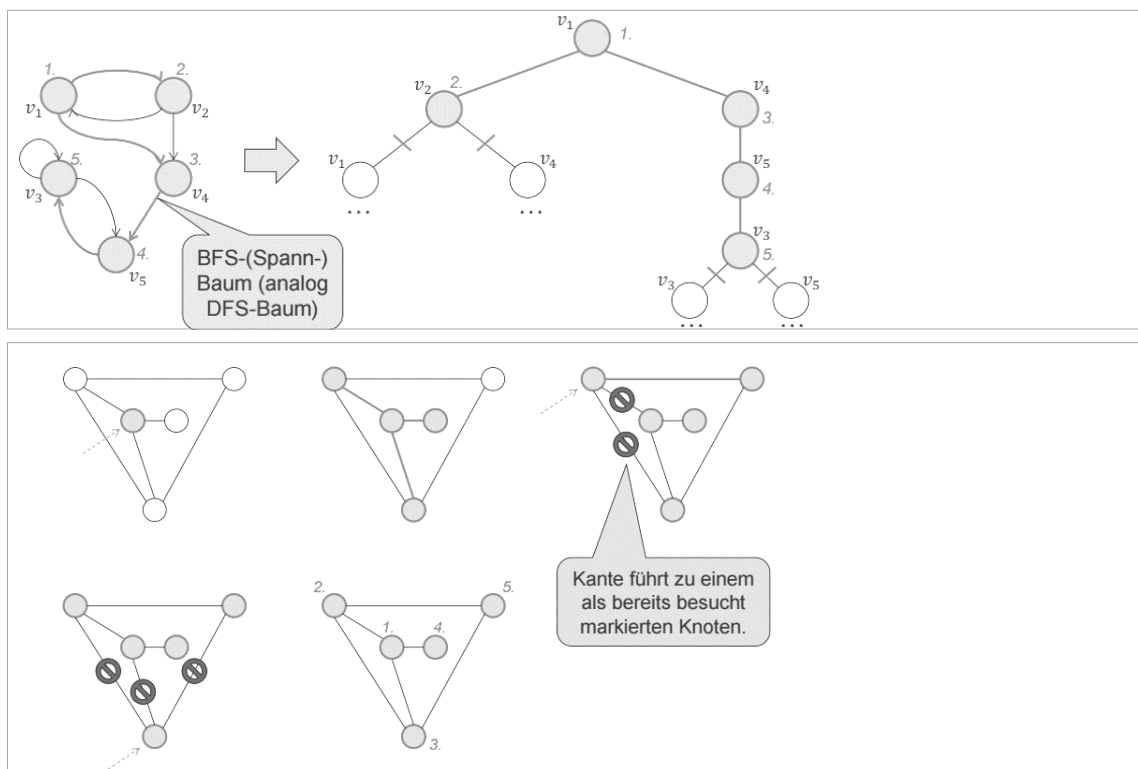
DFS-Baum gerichteter Graphen: Vorfahre

Seien $G = (V, E)$ ein gerichteter Graph und $T = (V, E')$ ein DFS-Baum von G , dann gilt für alle Kanten $e = (v, w) \in E$: wenn $v.\text{dfsNr} < w.\text{dfsNr}$, dann ist v Vorfahre von w in T .



Breitensuche (engl. breadth-first-search – BFS)

Ein Durchlauf eines Graphen G per Breitensuche (engl. breadthfirst-search – BFS) besucht die Knoten von G „ebenenweise“, also zuerst die Knoten, die über einen Pfad der Länge 1 von der Wurzel aus erreichbar sind, dann über einen Pfad der Länge 2 usw. In einem bereits besuchten Knoten von G wird abgebrochen.



Breitensuche

Idee: benutze Schlange, in der jeweils Nachfolger eines Knotens gemerkt werden, während dessen Geschwister verarbeitet werden. Algorithmus (iterativ):

```
Algorithmus bfs(v):  
  füge in die leere Schlange q den Knoten v ein;  
  solange q enthält Elemente führe aus: {  
    entnimm q das erste Element =: w;  
    markiere w als besucht;  
    verarbeite w;  
    für jeden Nachfolger wi von w führe aus: {  
      wenn wi noch nicht besucht,  
      dann hänge wi an q an;  
    }  
  }
```

Breitensuche



- bfs muss für jeden unbesuchten Knoten aufgerufen werden, um das Erreichen jedes Knotens sicherzustellen:

```
markiere alle Knoten als unbesucht;
für jeden Knoten v des Graphen führe aus: {
    wenn v noch nicht besucht wurde,
    dann bfs(v);
}
```

- Aufwand (bei Adjazenzlistendarstellung): $O(|V| + |E|)$

Breitensuche: iterative Implementierung mit Schlange

```
void bfs(Node v) {
    Queue<Node> q = new LinkedList<>();
    q.add(v); // merke Wurzel fuer Besuch vor
    while (!q.isEmpty()) {
        // besuche vorderstes Schlängenelement
        v = q.poll();
        v.mark(); // markiere v als besucht
        // hier: ggfs. Aktion auf v;
        Iterator<Edge> iter = v.getEdges();
        while (iter.hasNext()) {
            // lege alle Nachfolger in Schlange
            Edge e = iter.next();
            if (e.target.isUnmarked()) {
                q.add(e.target);
            }
        }
    }
}
```

Problemstellung „Kürzeste Wege 1:n“

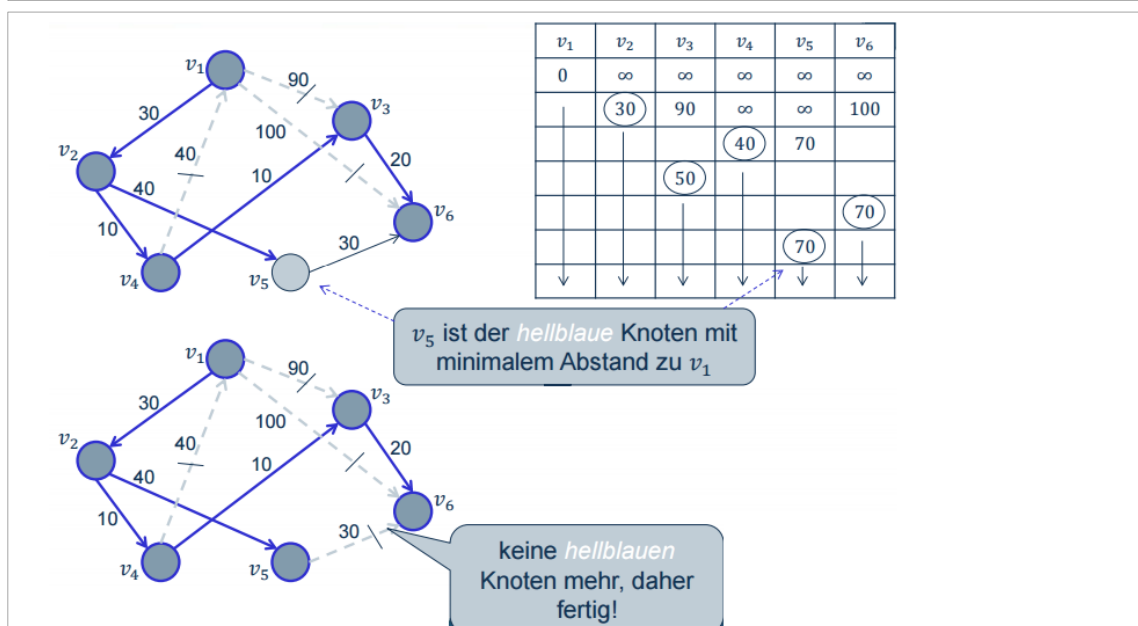
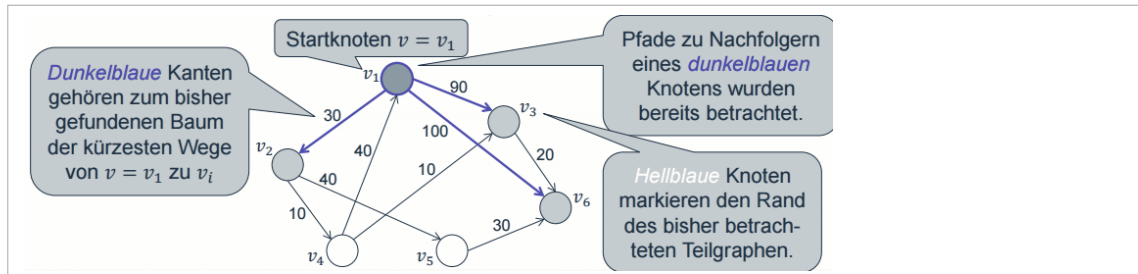
- Gegeben: gerichteter Graph G , dessen Kanten mit positiven reellen Zahlen (Kosten) beschriftet sind.
- Aufgabe:
 - Berechne für einen beliebigen Startknoten vv die kürzesten Wege zu allen anderen Knoten des Graphen.
 - Länge eines Pfades = Summe der Kantenkosten.
 - sog. single source shortest path-Problem

Grundidee des Algorithmus von Dijkstra

- Vom Startknoten vv aus wächst „erkundeter“ Teilgraph von G .
- Knoten
 - ungefärbt: noch nicht vom Algorithmus erkundet.
 - dunkelblau: alle Kanten zu Nachfolgern wurden betrachtet; Nachfolger liegen bereits im erkundeten Teilgraphen; dunkelblaue Knoten im Inneren des Teilgraphen; kürzeste Wege zu ihnen sind gefunden.
 - hellblau: ausgehende Kanten noch nicht betrachtet; hellblaue Knoten bilden Rand des Teilgraphen.
- Kanten
 - ungefärbt: noch nicht vom Algorithmus erkundet.
 - dunkelblau: dunkelblaue Kanten bilden innerhalb des Teilgraphen einen Baum der (bisher gefundenen) kürzesten Wege; in jedem Knoten wird der Abstand von vv verwaltet (über den bisher bekannten kürzesten Weg).
 - hellblau: Kante, die sicher nicht zum Baum der kürzesten Wege gehört (z. B. wegen eines kürzeren Weges daraus entfernt wurde)

Grundidee des Algorithmus von Dijkstra

- Teilgraph wächst schrittweise, indem der hellblaue Knoten w mit minimalem Abstand von v dunkelblau markiert wird.
- Drei Fälle für die Farbe der Nachfolger von w :
 - Ungefärbt (nicht schon im Teilgraphen) → Knoten hellblau markieren, die Kanten dorthin dunkelblau.
 - Hellblau → ggf. die für sie bisher bekannten kürzesten Pfade (dunkelblaue Kanten) korrigieren (und hellblau markieren).
 - Dunkelblau → alter Pfad dorthin kürzer, neue Kante dorthin hellblau



Algorithmus von Dijkstra

```

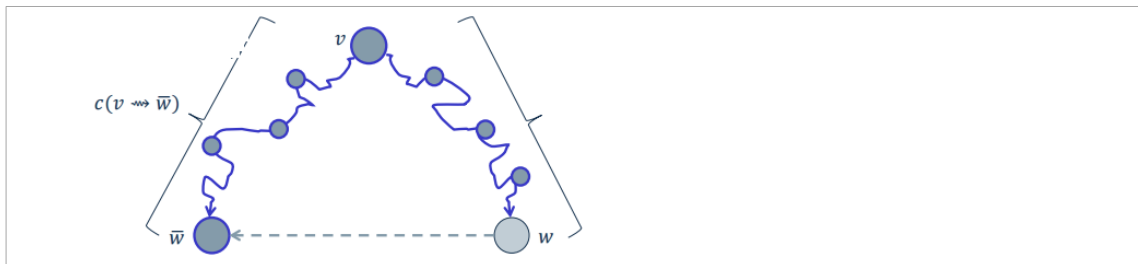
Algorithmus dijkstra(v)
dunkelblaueKnoten := ∅; hellblaueKnoten := {v}; dist(v) := 0;
solange hellblaueKnoten ≠ ∅ führe aus: {
  wähle w ∈ hellblaueKnoten, so dass  $\forall w' \in \text{hellblaueKnoten}: \text{dist}(w) \leq \text{dist}(w')$ ;
  färbe w dunkelblau; //zum Teilgraph hinzunehmen
  für alle Nachfolger wi von w führe aus: {
    falls wi ∉ (dunkelblaueKnoten ∪ hellblaueKnoten) // noch nicht besucht
    dann führe aus: { // Knoten, ungefaerbt
      färbe wi hellblau;
      färbe die Kante (w, wi) dunkelblau;
      dist(wi) := dist(w) + cost(w,wi);
    }
    sonst falls wi ∈ hellblaueKnoten // wi erneut besucht, hellblau
    dann falls dist(wi) > dist(w) + cost(w,wi)
    dann führe aus: { // es gibt kuerzere als bisher
      färbe die Kante (w, wi) dunkelblau;
      färbe die bisherige Kante zu wi hellblau;
      dist(wi) := dist(w) + cost(w,wi);
    }
    sonst färbe (w, wi) hellblau;
  }
  // sonst keine Aktion da wi dunkelblau
}
  
```

Korrektheit des Algorithmus von Dijkstra

- Lemma:
 - Zu jeder Zeit ist für jeden hellblauen Knoten w der dunkelblaue Pfad (vom Startknoten) zu w minimal unter allen blauen Pfaden zu w , das heißt solchen, die nur hellblaue oder dunkelblaue Kanten haben.
- Beweis (per Induktion über die Folge dunkelblau gefärbter Knoten):
 - Ind.-Anfang: Behauptung gilt für v , da alle Kanten ungefärbt sind.
 - Ind.-Annahme: Aussage gilt für alle bisher hellblau markierten Knoten.
 - Ind.-Schluss:
 - Es wird w dunkelblau gefärbt. Seien w_1, \dots, w_n die Nachfolger von w . Für die w_i sind folgende Fälle zu unterscheiden:
 - w_i wurde zum ersten Mal erreicht, war also bisher ungefärbt und wird jetzt hellblau. Der einzige blaue Pfad zu w_i hat die Form $v \rightsquigarrow w \rightarrow w_i$ (\rightsquigarrow : Pfad; \rightarrow : einzelne Kante). Der Pfad $v \rightsquigarrow w$ ist nach Induktionsannahme minimal. Also ist $v \rightsquigarrow w_i$ minimal.
 - w_i ist hellblau und wurde erneut erreicht. Der bislang dunkelblaue Pfad zu w_i war $v \rightsquigarrow x \rightarrow w_i$; der neue dunkelblaue Pfad zu w_i ist der kürzere der Pfade
 - $v \rightsquigarrow x \rightarrow w_i$
 - $v \rightsquigarrow w \rightarrow w_i$
 - Pfad 1 ist minimal unter allen blauen Pfaden, die nicht über w führen, Pfad 2 ist minimal unter allen, die über w führen, also ist der neue Pfad minimal unter allen blauen Pfaden.
 - w_i ist dunkelblau: keine Auswirkungen für irgendeinen hellblauen Knoten und Pfade dort hin. (Die ihn erreichende Kante wird hellblau).

Informelle Begründung für Fall c

- $c(v \rightsquigarrow \underline{w})$ ist gem. Lemma (bislang) minimal für alle bekannten Wege von v nach \underline{w} .
- $c(v \rightsquigarrow w)$ ist gem. Lemma (bislang) minimal für alle bekannten Wege von v nach w . Da \underline{w} vor w dunkelblau gefärbt wurde gilt: $c(v \rightsquigarrow \underline{w}) \leq c(v \rightsquigarrow w)$
- Daher muss diese Kante hellblau sein. $v \rightsquigarrow w \rightarrow \underline{w}$ kann nicht kürzer als $v \rightsquigarrow \underline{w}$ sein. Denn sonst hätte man zuerst w dunkelblau gefärbt.



Hinweise zum Algorithmus von Dijkstra

- In der Literatur finden sich:
 - verschiedene Färbungsvarianten von Knoten und Kanten mit tlw. auch leicht unterschiedlicher Semantik,
 - die vollständigen Beweise der Korrektheit.
- Dijkstras Algorithmus ist ein weiteres Beispiel eines gierigen Algorithmus.
- Wenn alle Kanten dieselben Kosten haben, dann löst „normale“ Breitensuche das Problem der kürzesten Wege auch (und sogar effizienter).

Implementierung des Algorithmus von Dijkstra: 1.) mit Adjazenzmatrix

- Benötigte Datenstrukturen:
 - Knotenklasse Node, Knotennummerierung von 1 ... n
 - Graph G dargestellt durch Kostenadjazenzmatrix $\text{cost}[i][j]$ mit Einträgen ∞ an den Matrixelementen, für die keine Kante existiert.
 - Arrays der Größe n für n Knoten:
 - $\text{double}[] \text{dist}$; Abstand des jeweiligen Knotens vom Startknoten
 - $\text{Node}[] \text{parent}$; Baum der dunkelblauen Kanten, indem zu jedem Knoten sein Vorgängerknoten festgehalten wird.
 - $\text{boolean}[] \text{darkblueNodes}$; mit true markiert Elemente der Menge der dunkelblauen Knoten.
- Ablauf:
 - Das gesamte Array dist wird durchlaufen, um den hellblauen Knoten w mit minimalem Abstand zu finden (Aufwand: $O(n)$).
 - Die Zeile $\text{cost}[w][*]$ der Matrix wird durchlaufen, um für alle Nachfolger von w ggf. den Abstand und den Vorgängerknoten zu korrigieren (Aufwand: $O(n)$).
 - Zeitaufwand insgesamt $O(n^2)$, da obige Teilschritte n -mal ausgeführt werden.
 - Implementierung ineffizient, wenn nicht n sehr klein oder $|E| \approx n^2$

Implementierung des Algorithmus von Dijkstra: 2.) mit Adjazenzlisten und Halde

- Benötigte Datenstrukturen:
 - Knotenklasse Node, Knotennummerierung von $1 \dots n$
 - Graph G dargestellt durch Adjazenzlisten mit Kosteneinträgen
 - Arrays dist und parent wie bei 1.)
 - min-Halde der Abstände hellblauer Knoten vom Ausgangsknoten
 - Haldeneinträge und Knoten sind doppelt verkettet, d. h. Haldeneintrag enthält Verweis auf Knoten und Array int[] heapaddress enthält zu jedem Knoten dessen Position in der Halde.
- Ablauf:
 - Entnimm den hellblauen Knoten w mit minimalem Abstand zu v aus der Halde (Aufwand: $O(\log |V|)$); für V Knoten: Gesamtaufwand $O(|V| \cdot \log |V|)$ (1)
 - Finde in der entsprechenden Adjazenzliste die m Nachfolger von w (Aufwand: $O(m)$)
 - Für jeden neuen hellblauen Nachfolger erzeuge Eintrag in der Halde (Aufwand: $O(\log |V|)$).
 - Für jeden alten hellblauen Nachfolger korrigiere ggfs. seinen Eintrag in der Halde (via heapaddress).
 - Da sein Abstandswert bei der Korrektur sinkt, kann der Eintrag in der Halde weiter nach oben wandern (Aufwand: $O(\log |V|)$).
 - Die Halden-Verweise der vertauschten Einträge im Array heapaddress können in $O(1)$ Zeit geändert werden.
 - Gesamtaufwand: $O(m \log |V|)$; mit $\sum m = |E|$ deshalb $O(|E| \cdot \log |V|)$ (2)
 - Gesamtaufwand (1)+(2) mit $n = |V|$: $O((|V| + |E|) \cdot \log |V|)$
 - Es gibt Optimierungen, die sogar $O(|E| + |V| \cdot \log |V|)$ erreichen

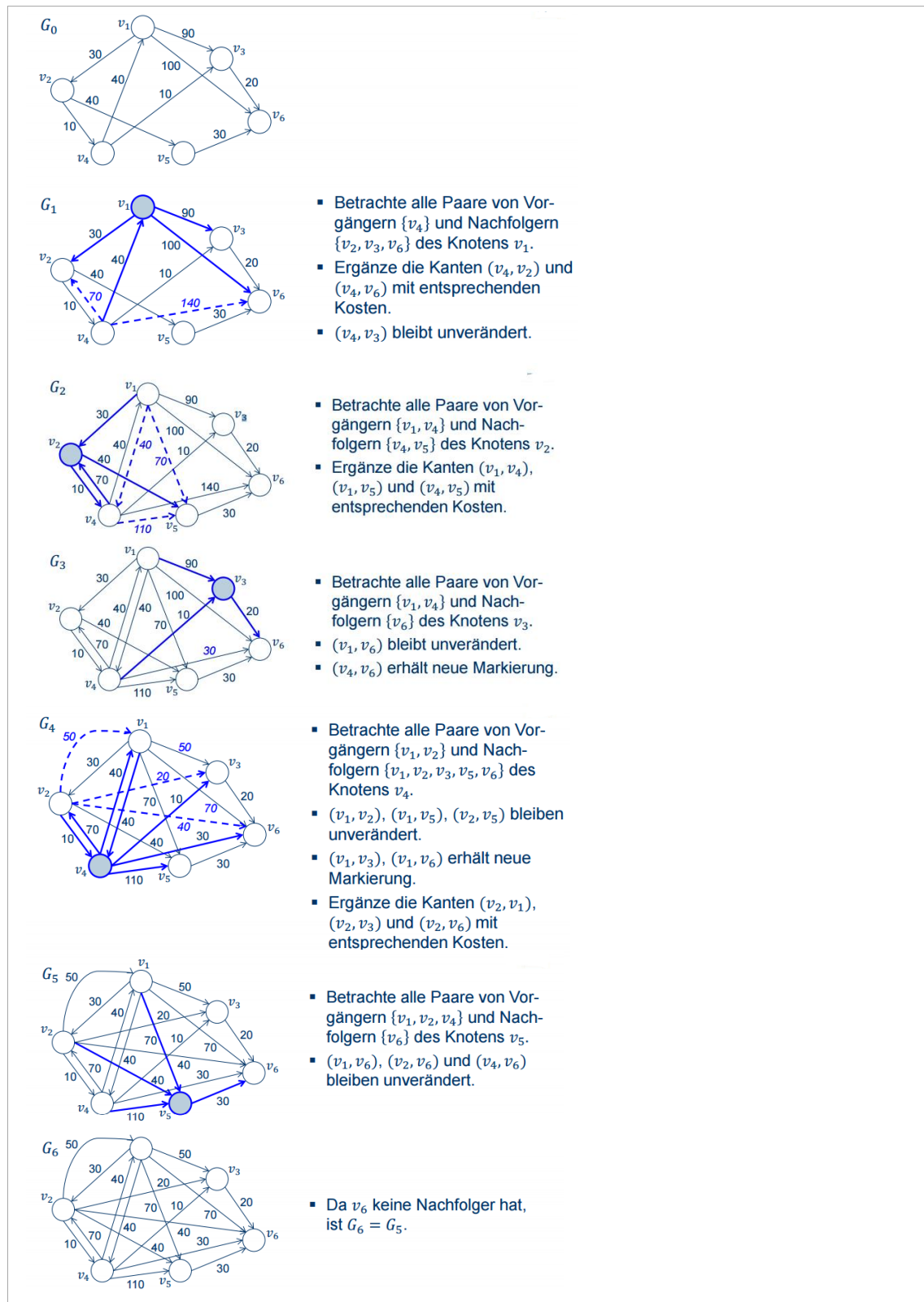
Problemstellung „Kürzeste Wege n:m“

- Gegeben: gerichteter Graph G , dessen Kanten mit positiven reellen Zahlen (Kosten) beschriftet sind
- Aufgabe:
 - Berechne die kürzesten Wege zwischen allen Paaren von Knoten des Graphen.
 - Länge eines Pfades = Summe der Kantenkosten.
 - sog. all pairs shortest path-Problem

Algorithmus von Floyd

- Gegeben: gerichteter Graph G mit n Knoten, die von 1 bis n durchnummeriert sind.
 - Algorithmus berechnet Folge von Graphen G_0, \dots, G_n , wobei G_i durch Modifikation von G_{i-1} entsteht.
 - Definition von G_i :
 - Jeder Graph G_i hat die gleiche Knotenmenge $\{v_1, \dots, v_n\}$ wie G .
 - Es existiert in G_i eine (direkte) Kante $v_j \xrightarrow{\alpha} v_k$ genau dann, wenn ein Pfad von v_j nach v_k in G existiert, der nur Knoten aus der Menge $\{v_1, \dots, v_i\}$ als Zwischenknoten verwendet. Der kürzeste derartige Pfad hat Kosten α .
 - Der Algorithmus startet mit $G_0 = G$.
 - Berechnung von G_i aus G_{i-1} :
 - Sei der Knoten v_j ein Vorgänger des Knotens v_i im Graphen G_{i-1} und sei v_k einer seiner Nachfolger.
 - Betrachte alle solche Paare (v_j, v_k) . 1. Falls noch keine (direkte) Kante von v_j nach v_k existiert, so erzeuge eine Kante $v_j \xrightarrow{\alpha+\beta} v_k$.
-
- Falls schon eine (direkte) Kante $v_j \xrightarrow{\gamma} v_k$ existiert, so ersetze die Markierung γ dieser Kante durch $\alpha + \beta$, falls $\alpha + \beta < \gamma$ (also falls man günstiger über v_i von v_j nach v_k gelangt).
- G_i erfüllt wiederum die oben geforderten Eigenschaften:
 - In G_{i-1} waren alle kürzesten Pfade bekannt und durch Kanten repräsentiert, die als Zwischenknoten nur Knoten aus $\{v_1, \dots, v_{i-1}\}$ benutzen.
 - Jetzt sind in G_i alle Pfade bekannt, die Knoten $\{v_1, \dots, v_i\}$ benutzen.
 - Deshalb sind nach n Schritten die Kosten aller kürzesten Wege von jedem Knoten zu jedem anderen Knoten aus G im letzten Graphen G_n repräsentiert.

Algorithmus von Floyd am Beispiel



Implementierung des Algorithmus von Floyd mit Adjazenzmatrix

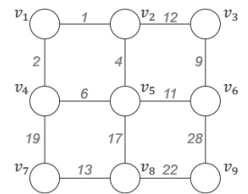
- gegeben: Graph GG mit Adjazenzmatrix $\text{cost}[j][k]$
 - $\text{cost}[i][i] = 0$ für alle v_i
 - $\text{cost}[j][k] = \infty$, falls in GG keine (direkte) Kante von v_j nach v_k

Implementierung des Algorithmus von Floyd mit Adjazenzmatrix

```
public double[][] floyd() {
    int n = cost.length;
    double[][] g = new double[n][n];
    for (int j = 0; j < n; j++) { // G0 = G initialisieren
        for (int k = 0; k < n; k++) {
            g[j][k] = cost[j][k];
        }
    }
    for (int i = 0; i < n; i++) { //vi
        for (int j = 0; j < n; j++) { //vj
            for (int k = 0; k < n; k++) { ((vk Laufzeit  $O(|V|^3)$ )
                if (g[j][i] + g[i][k] < g[j][k]) { //  $\alpha + \beta < \gamma$ 
                    g[j][k] = g[j][i] + g[i][k];
                }
            }
        }
    }
    return g;
}
```

Minimaler Spannbaum Problemstellung

- Ziel: finde den jeweils bezüglich eines Kostenmaßes „billigsten“ Spannbaum eines Graphen.
- Anwendungsbeispiel:
 - Verlegung neuer Kabel für Kabel-TV in einem Wohngebiet.
 - Kabel dürfen nur an bestimmten Pfaden (z. B. parallel/quer zu Straßen) entlang laufen.
 - Pfade haben unterschiedliche Kosten.
 - Wie sind alle Haushalte des Wohngebiets möglichst günstig zu versorgen?
- Gegeben: ungerichteter Graph G , dessen Kanten mit positiven reellen Zahlen (Kosten) beschriftet sind
- Aufgabe:
 - Berechne minimalen Spannbaum dieses Graphen.
 - Dabei ist der Spannbaum minimal, wenn die Summe seiner Kantengewichte minimal ist.

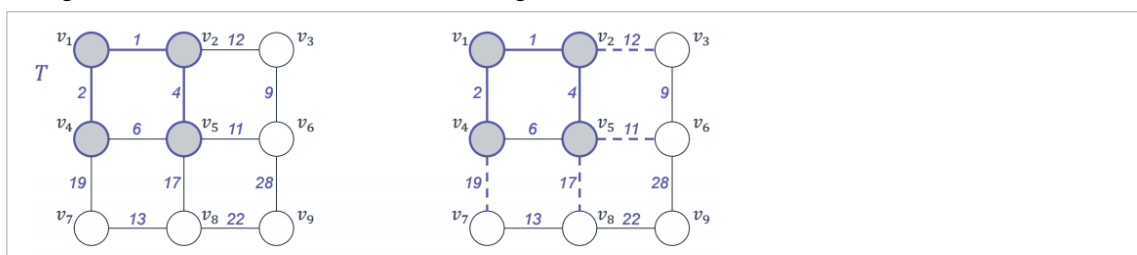


Erste Idee: Algorithmus von Dijkstra verwenden

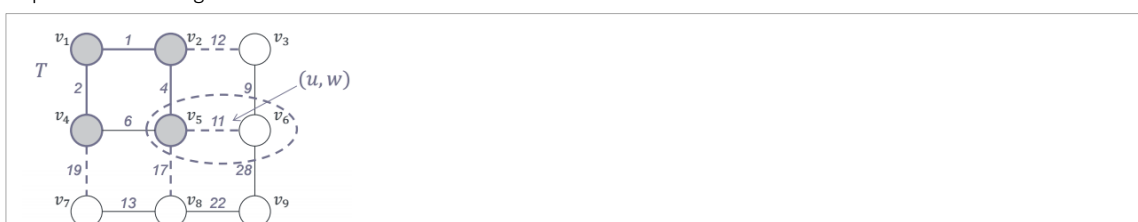
- Der Algorithmus von Dijkstra für die Suche nach dem kürzesten Pfad lieferte für einen gegebenen Knoten einen Spannbaum kürzester Pfade.
- In jedem Schritt wurde eine Kante so hinzugefügt, dass der entstehende Pfad minimale Länge hatte.
- Aufwand (mit Optimierung): $O(|E| + |V| \cdot \log |V|)$
- Strategie für minimalen Spannbaum:
 - Bestimme für jeden der $|V|$ Knoten den Spannbaum kürzester Pfade.
 - Wähle den kleinsten dieser Spannbäume aus.
- $|V|$ -fache Anwendung des Algorithmus von Dijkstra, Gesamtaufwand: $O(|V| \cdot (|E| + |V| \cdot \log |V|))$.

Grundidee des Algorithmus von Prim

- Angenommen man hat einen Teilgraph T von G gefunden, der auch Teilgraph des minimalen Spannbaums ist
- Es gibt Knoten, die über eine Kante mit T verbunden sind, die aber noch nicht zu T gehören
- Frage: Wie kann man T um eine weitere Kante vergrößern?



- Die Kante mit dem kleinsten Gewicht, die einen zu T benachbarten Knoten verbindet, wird zum minimalen Spannbaum hinzugenommen.



Grundidee des Algorithmus von Prim

- Korrektheitsüberlegungen
 - (u, w) sei die billigste Kante, die einen Knoten von T mit einem Knoten außerhalb von T verbindet.
 - (u, w) gehört zum minimalen Spannbaum.
 - Andernfalls gäbe es einen Pfad von irgendeinem Knoten aus T zu w mit niedrigeren Kosten.
 - Da jeder Pfad T verlassen muss, hat dieser Pfad mindestens die Kosten der ersten Kante, die T verlässt.
 - Diese sind aber höher als die Kosten der Kante (u, w) .
 - Einen kleineren Spannbaum bekäme man also, indem man diese erste Kante durch (u, w) ersetzt.

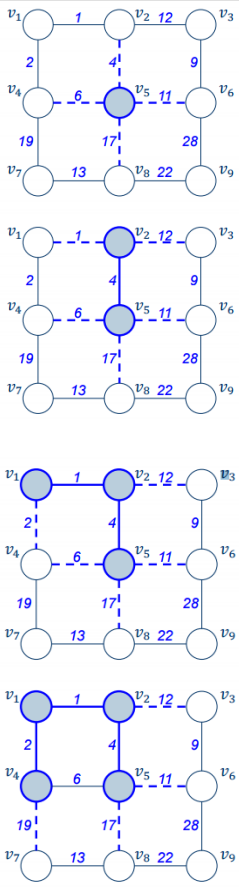
Algorithmus von Prim

```

Algorithmus prim():
  Wähle einen beliebigen Knoten als Startgraph T.
  Solange T noch nicht alle Knoten enthält, führe aus: {
    Wähle eine Kante e mit minimalen Kosten aus, die
      einen noch nicht in T enthaltenen Knoten v mit
      T verbindet.
    Füge e und v dem Graphen T hinzu.
  }
  
```

- Hinweis:
 - Der Algorithmus wählt in jedem Schritt eine Kante mit minimalen Kosten aus der Menge zu betrachtender Kanten aus.
 - Der Algorithmus von Prim ist damit ein weiteres Beispiel für einen gierigen Algorithmus.

Algorithmus von Prim am Beispiel



- Wähle als Startgraph den Knoten v_5 .
- zu untersuchende Kanten (sortiert nach Kosten):
 - (v_2, v_5) , Kosten: 4
 - (v_4, v_5) , Kosten: 6
 - (v_5, v_6) , Kosten: 11
 - (v_5, v_8) , Kosten: 17

Wähle Kante (v_2, v_5) mit minimalen Kosten für T .

- Ergänze Knoten v_2 und Kante (v_2, v_5) in T .
- zu untersuchende Kanten (sortiert nach Kosten):
 - (v_1, v_2) , Kosten: 1
 - (v_4, v_5) , Kosten: 6
 - (v_5, v_6) , Kosten: 11
 - (v_2, v_3) , Kosten: 12
 - (v_5, v_8) , Kosten: 17

Wähle Kante (v_1, v_2) mit minimalen Kosten für T .

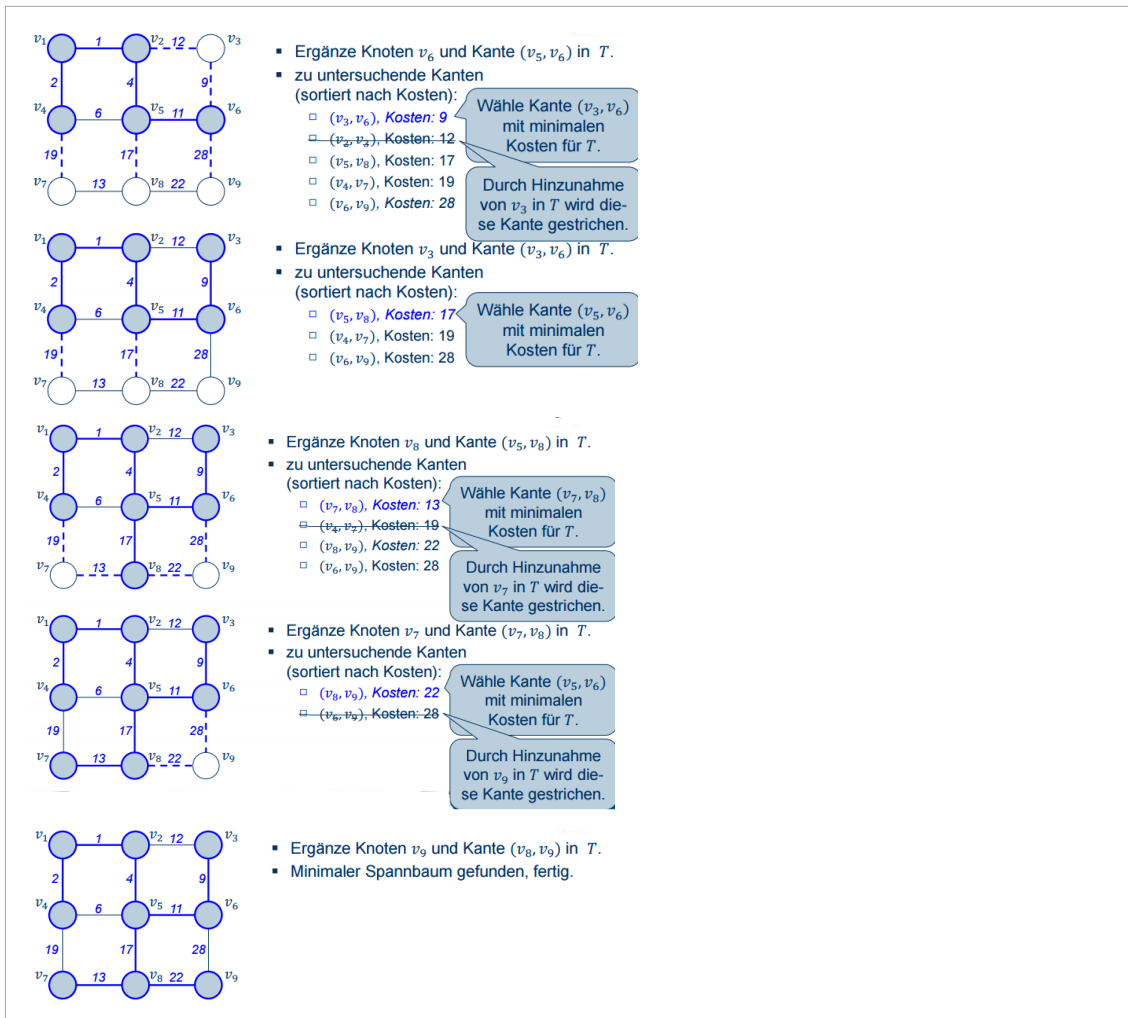
- Ergänze Knoten v_1 und Kante (v_1, v_2) in T .
- zu untersuchende Kanten (sortiert nach Kosten):
 - (v_1, v_4) , Kosten: 2
 - (v_4, v_5) , Kosten: 6
 - (v_5, v_6) , Kosten: 11
 - (v_2, v_3) , Kosten: 12
 - (v_5, v_8) , Kosten: 17

Wähle Kante (v_1, v_4) mit minimalen Kosten für T .

Durch Hinzunahme von v_4 in T wird diese Kante gestrichen.

- Ergänze Knoten v_4 und Kante (v_1, v_4) in T .
- zu untersuchende Kanten (sortiert nach Kosten):
 - (v_5, v_6) , Kosten: 11
 - (v_2, v_3) , Kosten: 12
 - (v_5, v_8) , Kosten: 17
 - (v_4, v_7) , Kosten: 19

Wähle Kante (v_5, v_6) mit minimalen Kosten für T .



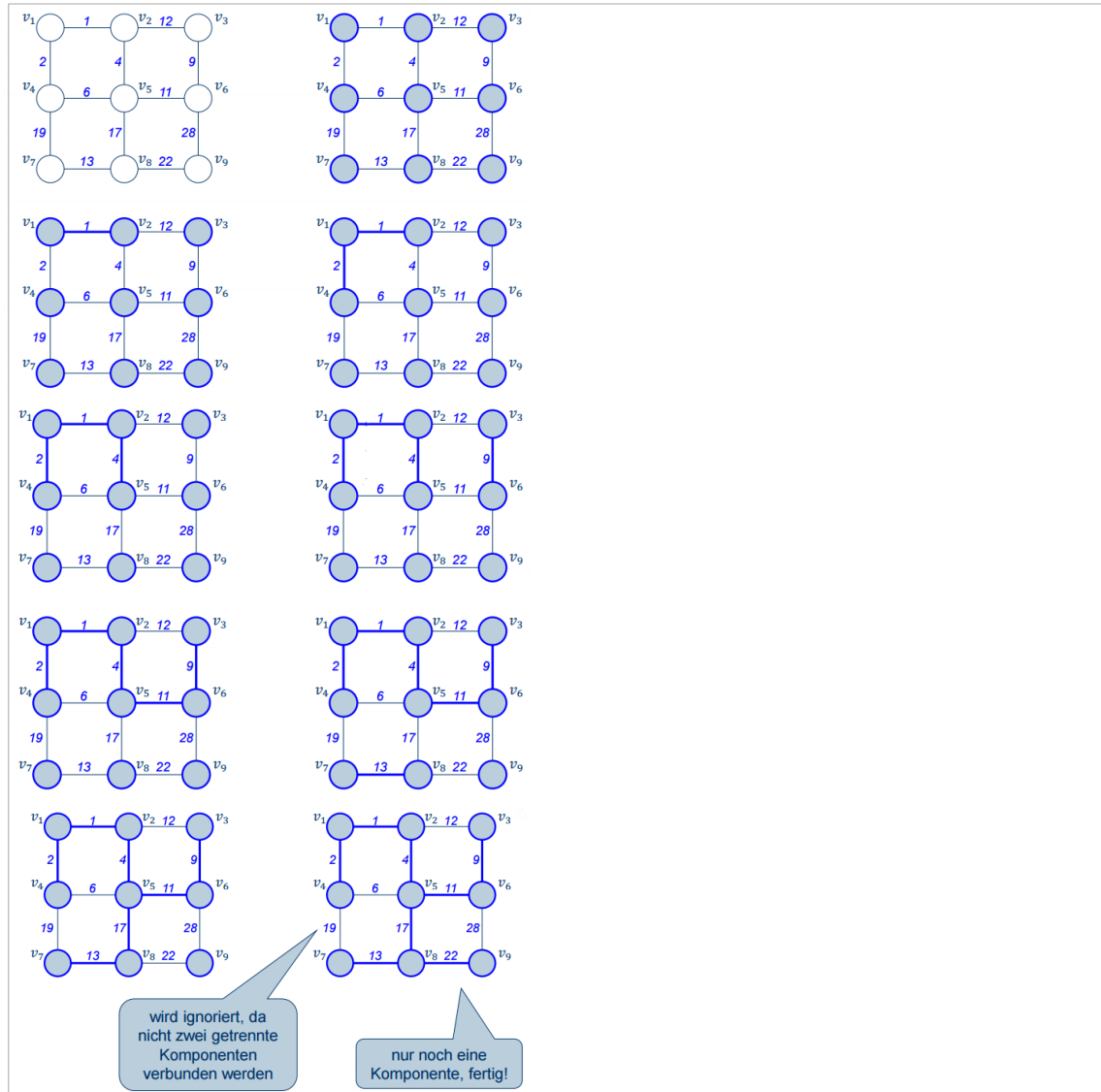
Aufwand des Algorithmus von Prim

- Verwende min-Halbe als Prioritätswarteschlange zur Verwaltung der noch zu betrachtenden Kanten.
- Damit kann (mit Optimierung) als Gesamtaufwand $O(|E| + |V| \cdot \log |V|)$ erreicht werden.
- Gut, wenn der Graph viele Kanten hat.

Grundidee des Algorithmus von Kruskal

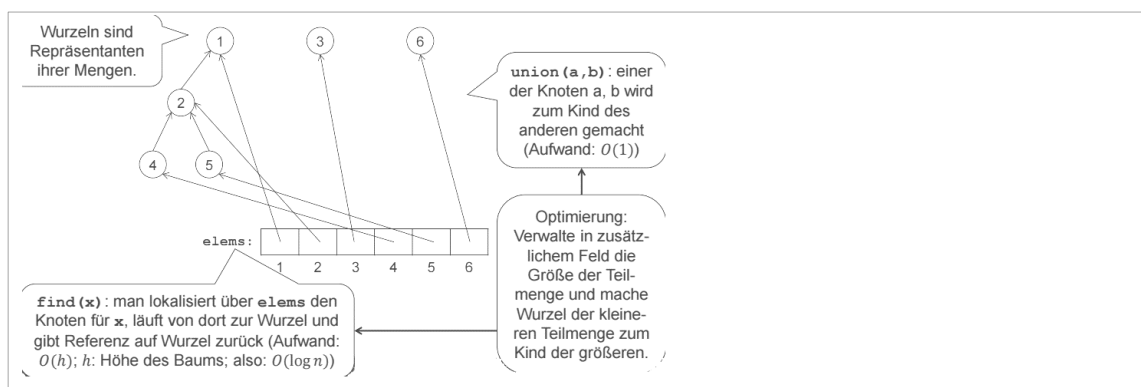
- Minimale Spannbäume haben eine interessante Eigenschaft:
- Lemma:
 - Sei für $G = (V, E)$ $\{U, W\}$ eine restlose und überschneidungsfreie Zerlegung der Knotenmenge V .
 - Sei u, w eine Kante in G mit minimalen Kosten unter allen Kanten $\{(u', w') \mid u' \in U, w' \in W\}$.
 - Dann gibt es einen minimalen Spannbaum T für G , der (u, w) enthält.
- Beweis:
 - Sei $T = (V, E')$ ein beliebiger minimaler Spannbaum für G , der (u, w) nicht enthält.
 - Dann gibt es in T mindestens eine, vielleicht aber auch mehrere Kanten, die U mit W verbinden
 - Wenn man nun T die Kante (u, w) hinzufügt, entsteht ein Graph T' mit einem Zyklus, der über (u, w) verläuft.
 - u muss also noch auf einem anderen Weg mit w verbunden sein, der mindestens eine weitere Kante (u', w') enthält, die U mit W verbindet.
 - Wenn man nun (u', w') aus T' löscht, entsteht ein Spannbaum T'' , dessen Kosten höchstens so hoch sind, wie die von T , da (u, w) minimale Kosten hat.
 - Also ist T'' ein minimaler Spannbaum, der (u, w) enthält.
- Das Lemma ist die Grundlage des Algorithmus von Kruskal zur Konstruktion minimaler Spannbäume, der wie folgt vorgeht:
 - Zu Anfang sei T ein Graph, der genau die Knoten von G enthält, aber keine Kanten.
 - Die Kanten von G werden nun in der Reihenfolge aufsteigender Kosten betrachtet.
 - Wenn eine Kante zwei getrennte Komponenten von T verbindet, so wird sie in den Graphen T eingefügt und die Komponenten werden verschmolzen.
 - Anderenfalls wird die Kante ignoriert.
 - Sobald T nur noch aus einer einzigen Komponente besteht, ist T ein minimaler Spannbaum für G .

Algorithmus von Kruskal am Beispiel



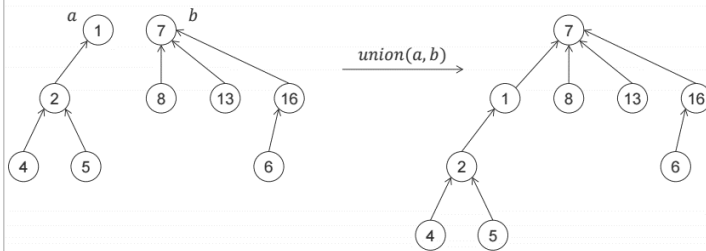
Einschub: Partitionen von Mengen mit union und find

- Aufgabe:
 - Verwaltung einer Zerlegung einer Menge in Teilmengen (sog. Partition einer Menge).
 - Benötigte Operationen:
 - union verschmilzt zwei Teilmengen, d. h. bildet die Vereinigung ihrer Elemente,
 - find stellt für ein gegebenes Element fest, zu welcher Teilmenge es gehört.
- Implementierung mit Bäumen
 - Darstellung jeder Teilmenge durch einen Baum, dessen Knoten Elemente der Teilmenge repräsentieren und in dem Verweise jeweils vom Kind zum Elternknoten führen.
 - Wurzeln sind Repräsentanten der Mengen ihrer Kinder.
- Zusätzlich: Array, um die Knoten im Baum direkt zu erreichen.
- Beispiel: Zerlegung der Menge $\{1, 2, 3, 4, 5, 6\}$ in die Teilmengen $\{1, 2, 4, 5\}$, $\{3\}$ und $\{6\}$.



Einschub: Partitionen von Mengen mit union und find

■ Beispiel: Verschmelzen zweier Teilmengen



■ Dargestellte Mengen:

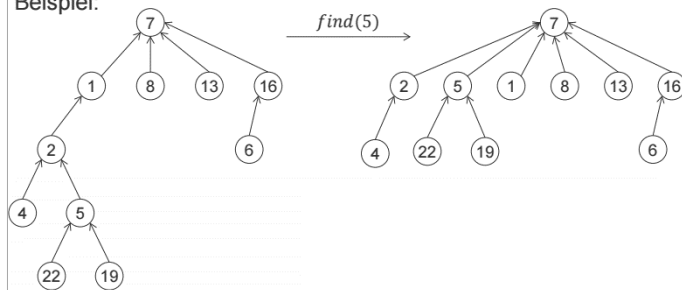
{1, 2, 4, 5}

{6, 7, 8, 13, 16}

{1, 2, 4, 5, 6, 7, 8, 13, 16}

- Letzte Verbesserung: sog. Pfadkompression
- Folge von nn find-Operationen könnte $OO(n \log n)$ Zeit brauchen.
- Idee: bei jedem find-Aufruf alle Knoten auf dem Pfad zur Wurzel direkt zu Kindern der Wurzel machen.

Beispiel:



Einschub: Partitionen von Mengen mit union und find

- Aufwand abhängig von der Höhe der Bäume
 - Höhe der Bäume im Worst-Case: $O(n)$
 - Höhe der Bäume unter Beachtung der Größe bei union-Operation: $O(\log n)$
- Aufwand für m union-/find-Operationen auf n Elementen: $O((m + n) \log n)$
- Aufwand mit Pfadkompression (ohne Beweis): $O(m + n)$ für alle praktisch relevanten n und m

Aufwand des Algorithmus von Kruskal

- Jede Kante wird aus der Halde entfernt: $O(|E| \cdot \log |E|)$.
- Aufwand für Baumerweiterung und Test auf Zyklensfreiheit?
 - Union-Find-Datenstruktur
 - Initial bildet jeder Knoten seine eigene Menge: $\{v_0\} \{v_1\} \dots \{v_{n-2}\} \{v_{n-1}\}$
 - Baumerweiterung um Kante (u, v) mittels $\text{union}(u, v)$
 - Test auf Zyklensfreiheit inkl. Kante (u, v) mittels $\text{find}(u) = ? = \text{find}(v)$
 - $|E|$ union-/find-Operationen: $O(|E| + |V|)$
- Gesamtaufwand: $O(|E| \cdot \log |E|)$
- Gut, wenn der Graph wenige Kanten (und evtl. viele Knoten hat).