Kapitel 4 Graphen und Graphalgorithmen

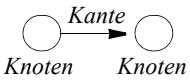
Graph ➙ Menge von Objekten (‘*Knoten*’) + Beziehungen zwischen den Objekten (‘*Kanten*’).
Ein Graph stellt diese Beziehungen ‘graphisch’ dar.

Beispiele

Objekte	Beziehungen zwischen den Objekten
Städte	es gibt eine Flugverbindung zwischen den Städten
Stellungen beim Schachspiel	es gibt einen Zug von einer Stellung zur anderen
Zustände eines Automaten	es gibt einen Übergang zwischen zwei Zuständen
WWW-Seiten	es gibt einen Link zwischen den beiden Seiten

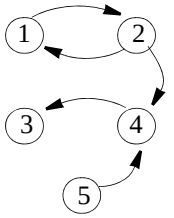
Definition: *(endlicher) Graph*

Ein **Graph** G ist ein Paar $G = (V, E)$ mit:
V: endliche, nichtleere Menge von Knoten
E: Menge von Kanten: $E \subseteq V \times V$.

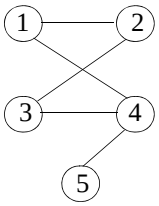


Bezeichnungen:

• Wir unterscheiden:
gerichtete Graphen ($(v_1, v_2) \neq (v_2, v_1), \{v_1, v_2\} \subseteq V$);
➙ Kanten als ‘$\rightarrow$’.
ungerichtete Graphen (Symmetrie: $(v_1, v_2) \equiv (v_2, v_1), \{v_1, v_2\} \subseteq V$)
➙ Kanten als ‘—’.



gerichteter Graph



ungerichteter Graph

Ungerichtete Graphen sind Spezialfälle gerichteter Graphen und lassen sich stets als solche darstellen.

- Als *Teilgraph* eines Graphen $G = (V, E)$ bezeichnet man einen Graphen $G' = (V', E')$ mit: $V' \subseteq V$ und $E' \subseteq E$.

- In einem ungerichteten Graphen heißen zwei Knoten v_1 und v_2 *benachbart* oder *adjazent*, falls $(v_1, v_2) \in E$. In einem gerichteten Graphen heißen zwei Knoten v_1 und v_2 *benachbart* oder *adjazent*, falls $(v_1, v_2) \in E$ oder $(v_2, v_1) \in E$.

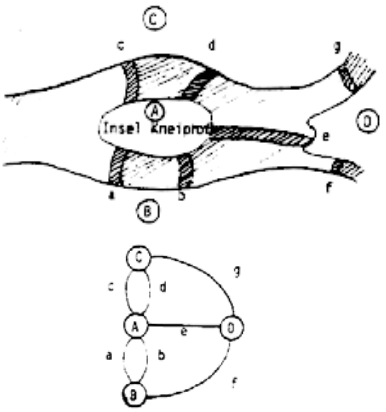
- Der *Grad* eines Knotens ist die Anzahl der von ihm ausgehenden Kanten.

Das *Königsberger Brückenproblem*:

Ist es möglich, ausgehend von einem der Punkte A, B, C oder D, alle Brücken einmal zu überqueren und wieder an den Ausgangspunkt zurückzukehren?

(Beispiel für einen ungerichteten *Multigraphen*, d.h. zwei Knoten können durch mehr als eine Kante verbunden sein)

→ genau dann lösbar, wenn der Grad jedes Knotens gerade ist (Euler 1736).



- Ein *Weg (Pfad) der Länge n* vom Knoten v_1 zum Knoten v_2 ist eine Folge von Knoten $(w_0, w_1, \ldots, w_n)$ mit $w_0 = v_1$, $w_n = v_2$, $w_i \in V, 0 \le i \le n$ und $(w_i, w_{i+1}) \in E$.

  In einem *einfachen Weg* sind alle Knoten paarweise verschieden: $w_i \neq w_j, 0 \le i, j \le n, i \neq j$.

  Ausnahme: $w_0 = w_n$ ist erlaubt.

- Ein *Zyklus* ist ein einfacher Weg mit $w_0 = w_n$, $n \ge 2$.

  Ein *Eulerscher Kreis* ist ein spezieller Zyklus, bei dem jede Kante genau einmal durchlaufen wird.

  Ein *Hamiltonscher Kreis* ist ein Zyklus, der jeden Knoten genau einmal durchläuft.

- In einem ungerichteten Graphen heißt ein Knoten v_1 mit dem Knoten v_2 *verbunden*, falls es einen Weg von v_1 nach v_2 gibt.

  In einem gerichteten Graphen heißen zwei Knoten v_1 und v_2 *stark verbunden*, falls es einen Weg von v_1 nach v_2 und von v_2 nach v_1 gibt.

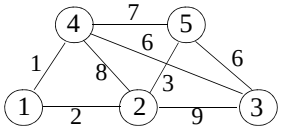
- Ein ungerichteter Graph heißt *zusammenhängend*, wenn alle Knoten paarweise miteinander verbunden sind. Eine (*Zusammenhangs-*)*Komponente* von *G* ist ein maximaler zusammenhängender Teilgraph von *G.*

- Eine *stark verbundene Komponente* eines gerichteten Graphen ist ein maximaler Teilgraph (maximale Knoten- und Kantenmenge), in dem alle Knoten paarweise stark verbunden sind.

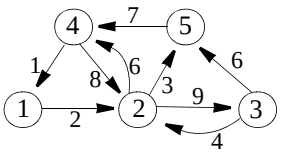
Bemerkung: Bäume sind Spezialfälle von Graphen.

Den Knoten und speziell den Kanten eines Graphen sind häufig weitere Informationen zugeordnet.
➙ *markierte Graphen*.

Beispiele für Kantenmarkierungen:
- Bedingungen für den Übergang von einem Knoten zum anderen (z.B.: endl. Automaten).
- Entfernungen oder Reisezeiten in einem Verkehrsnetz.
- allgemein: 'Kosten' für den Übergang von einem Knoten auf einen benachbarten Knoten.



Graph G_1



Graph G_2

Hierbei gilt: Die *Kosten eines Weges* entsprechen der Summe der Kosten aller beteiligten Kanten.

4.1 Speicherdarstellung von Graphen

```
public abstract class NodeSet {
    public NodeSet() {}
    public abstract void insert ( int );// Knoten sind durch ihre int-Schlüssel repräsentiert
    public abstract void delete( int );
    public abstract boolean isEmpty();
    public abstract boolean contains ( int );
    public abstract void nextElement();
    public abstract void union ( NodeSet );
    }

public abstract class Graph {
    public int n;   // Anzahl der Knoten
    public abstract NodeSet neighbors( int );
    public abstract float cost ( int, int );

    public void breadthFirstTraversal()
    ... }
```

Es gibt verschiedene Datenstrukturen zur Implementierung von Graphen. Die geeignete Speicherdarstellung ist in der Regel vom Anwendungsfall abhängig. Es stehen sich dabei gegenüber:

- der benötigte Speicheraufwand
- die Effizienz, mit der Zugriffe auf Knoten/Kanten durchgeführt werden können.

Wir stellen im folgenden die beiden gebräuchlichsten Datenstrukturen zur Implementierung der obigen abstrakten Klasse *Graph* vor. Diese sind als statische Strukturen zu verstehen, d.h. die Unterstützung dynamischer Einfügungen und Entfernungen während der Laufzeit eines darauf basierenden Algorithmus ist kein vorrangiges Ziel.

4.1.1 Adjazenzmatrix

Gegeben sei ein Graph $G = (V, E)$ mit $|V| = n$.
Seien im folgenden die Knoten mit 1 bis n durchnumeriert.
Eine **Adjazenzmatrix** ist eine boolsche $n \times n$-Matrix A mit:

$$A[i,j] = \begin{cases} 1 & \text{falls } (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$$

Adjazenzmatrix zu Graph G_1:

	1	*2*	*3*	*4*	*5*
1	0	1	0	1	0
2	1	0	1	1	1
3	0	1	0	1	1
4	1	1	1	0	1
5	0	1	1	1	0

Die Adjazenzmatrix eines ungerichteten Graphen ist symmetrisch. Es genügt daher die Speicherung der oberen Dreiecksmatrix.

Werden in A die Kosten der Kanten eines markierten Graphen eingetragen, so nennt man A eine **markierte Adjazenzmatrix** oder **Kostenmatrix**. Nichtexistierende Kanten werden dabei durch ein ausgewähltes Symbol (hier: '∞') bezeichnet. Für alle i, $1 \le i \le n$, gilt $A[i,i] = 0$.

Markierte Adjazenzmatrix zu Graph G_2:

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

Für die Graphdarstellung mit Adjazenzmatrizen gilt:

Vorteil: • die Existenz einer bestimmten Kante kann in $O(1)$ Zeit ermittelt werden.

Nachteile: • hoher Platzbedarf : $O(n^2)$ (bei kleiner Kantenanzahl unverhältnismäßig).

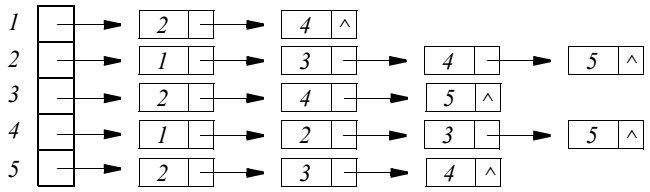
• hohe Initialisierungszeit von $O(n^2)$.

• Algorithmen, die den gesamten Graphen betrachten: $\Theta(n^2)$ Aufwand.

4.1.2 Adjazenzliste

Jedem der n Knoten wird eine Liste aller seiner adjazenten Knoten (Nachfolger) zugeordnet. Für einen effizienten Zugriff werden die Listenköpfe in einem Array vom Typ '*Adjazenzliste*' organisiert:

Adjazenzliste zu Graph G_1:



Im Falle von markierten Graphen können die Listenelemente weitere Informationen, wie etwa die Kosten der jeweiligen Kante, enthalten.

Für die Darstellung mit Adjazenzlisten gilt:

Vorteile: • der Platzbedarf beträgt nur $O(n + |E|)$.

• alle Nachbarn eines Knotens werden in linearer Zeit (in der Anzahl der Kanten, die von diesem Knoten ausgehen) gefunden.

Nachteil: • der Zugriff auf eine bestimmte Kante (v, w) ist nicht immer in konstanter Zeit möglich, da die Adjazenzliste von v zu durchlaufen ist.

4.2 Graphdurchläufe

Problem: Man besuche systematisch alle Knoten eines Graphen $G = (V, E)$, die von einem Knoten v aus erreichbar sind.
Auf der Lösung dieses Grundproblems bauen die meisten Graphalgorithmen auf.

Definition:

Ein Graphdurchlauf beginnt mit einem Knoten $v \in V$, der sogenannten **Wurzel**, und sucht jeden von v aus erreichbaren Knoten in G auf. Die Menge dieser Knoten bildet zu v einen **Wurzelgraphen**.

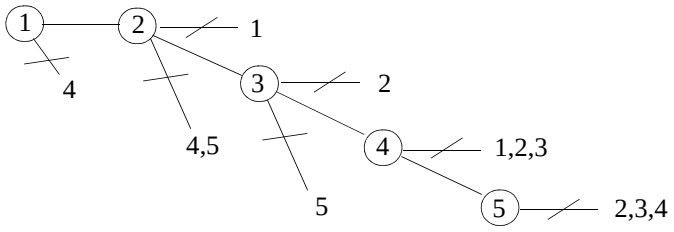
➙ Der Graphdurchlauf erzeugt einen Baum der besuchten Knoten; dieser wird unendlich, falls der Wurzelgraph mindestens einen Zyklus enthält. Der Durchlauf wird daher dort abgebrochen, wo er auf einen bereits zuvor besuchten Knoten trifft.

Es gibt zwei prinzipielle Ansätze, einen Graphen systematisch zu durchlaufen: den *Tiefendurchlauf* und den *Breitendurchlauf*.

Tiefendurchlauf (*depth-first-search*)

Prinzip: Ausgehend von v wird ein zu v adjazenter Knoten $v' \in V$ besucht.
Bevor nun weitere zu v adjazente Knoten betrachtet werden, wird das Verfahren rekursiv auf *v'* angewandt.
Der auf diese Weise erzeugte Baum heißt **depth-first Spannbaum** (*depth-first spanning tree*).

Depth-first Spannbaum zu Graph G_1 (Startknoten v = '1'):



Die folgende Methode *depthFirstTraversal* erzeugt bzw. durchläuft für jede Komponente des Graphen einen depth-first Spannbaum. Die Knoten seien dabei mit 0,..., *n -1* bezeichnet. Es wird eine rekursive Hilfsmethode '*depthSearch*' verwendet:

```
public void  depthSearch ( int v, boolean visited[] ) {
    int w;
    visited[v] = true;
    Verarbeitung des Knotens v ...;
    Für alle w  ∈ neighbors(v)  (* aus Adjazenzliste / Adjazenzmatrix *)
        if (!visited[w]) depthSearch(w, visited);
    }
```

```
void depthFirstTraversal() {
    boolean visited[] = new boolean [n];
    int v ;
    for (v = 0; v < n; v++) visited[v] = false;
    for (v = 0; v < n; v++)
        if (!visited[v]) depthSearch(v, visited);
    }
```

Laufzeitverhalten des Tiefendurchlaufs im schlechtesten Fall:

mit Adjazenzliste: $O(n + |E|) = O(max(n, |E|))$.

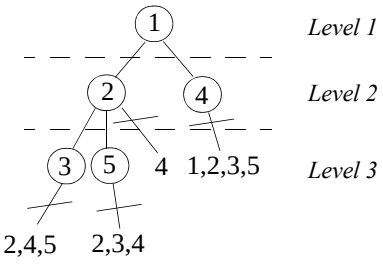
mit Adjazenzmatrix: $O(n^2)$.

Breitendurchlauf (*breadth-first-search*)

Prinzip: Ausgehend von *v* werden zunächst *alle* zu *v* adjazenten Knoten besucht und in einer Schlange notiert.
Anschließend werden nacheinander die Knoten in der Schlange auf dieselbe Weise behandelt, wobei bereits besuchte Knoten übergangen werden.
➙ 'ebenenweiser' Durchlauf

Der auf diese Weise erzeugte Baum heißt **breadth-first Spannbaum** (*breadth-first spanning tree*).

Breadth-first Spannbaum zu Graph G_1 (Startknoten *v* = '1'):



Die folgende Methode *breadthFirstTraversal* erzeugt bzw. durchläuft für jede Komponente des Graphen einen breadth-first Spannbaum.

Als Datenstruktur zur Verwaltung der noch zu besuchenden Knoten wird eine Schlange (*Queue*) verwendet. Die Knoten seien mit 0,..., *n -1* bezeichnet.

```
public void breadthFirstTraversal() {
    boolean visited[] =  new boolean [n];
    int v, w, w';
    Queue q;     // Schlange von Knoten mit Queue-Methoden
    for (v = 0; v < n; v++) visited[v] = false;
    q = new Queue();
    for (v = 0; v < n; v++) {
        if (!visited[v]) {
            q.enqueue(v);
            visited[v] = true;
            while (!q.isEmpty()) {
                q.dequeue(w);
                Verarbeitung des Knotens w ...;
                Für alle w' ∈ neighbors(w)  // aus Adjazenzliste / Adjazenzmatrix
                    if (!visited[w']) {
                        q.enqueue(w');
                        visited[w'] = true;}
                } // end while
            } // end if
        } // end for
    } // end breadthFirstTraversal
```

Das *Laufzeitverhalten* des Breitendurchlaufs ist dasselbe wie beim Tiefendurchlauf.

4.3 Bestimmung kürzester Wege in Graphen

4.3.1 Single Source Shortest Path-Problem

Problem: Gegeben sei ein (gerichteter) Graph $G = (V, E)$, dessen Kanten mit nicht-negativen reellen Zahlen (*Kosten*) beschriftet sind.
Aufgabe ist es, für einen beliebigen Knoten $v \in V$ die kürzesten Wege zu allen anderen Knoten in G zu berechnen, wobei die Länge eines Weges durch die Summe der Kosten der Kanten beschrieben ist (*single source shortest path*-Problem).

Der Algorithmus von *Dijkstra* (1959)

Prinzip: Ausgehend von *v* wird *G* sukzessive bearbeitet, wobei die Knoten und Kanten des bereits bearbeiteten Teilgraphen von *G* jeweils in zwei Klassen eingeteilt werden: *grüne* und *gelbe Knoten* bzw. *gelbe* und *rote Kanten*. Dabei gilt die folgende Klassifizierung:

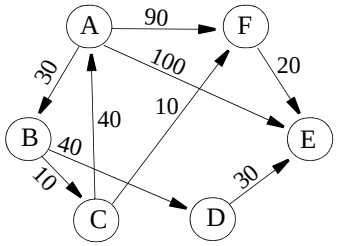
Knoten	*Grün*	Knoten, bei denen bereits alle Kanten zu Nachfolgern betrachtet wurden (d.h. diese liegen ebenfalls im Teilgraphen).
	Gelb	Knoten, bei denen die ausgehenden Kanten noch nicht betrachtet wurden ('Peripherie' des Teilgraphen).
Kanten	*Gelb*	'Normale' Kanten des Teilgraphen.
	Rot	Bilden einen Baum der kürzesten Wege innerhalb des Teilgraphen (Lösung eines Teilproblems).

Jedem Knoten des Teilgraphen wird weiterhin der kürzeste Abstand zu *v* über Kanten des Teilgraphen (rote Kanten) zugeordnet.

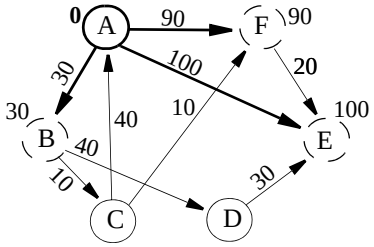
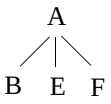
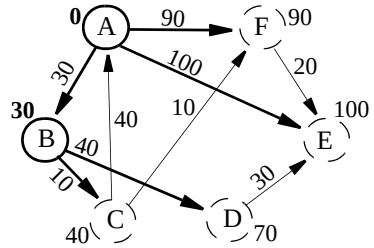
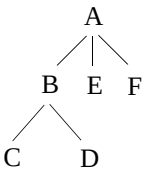
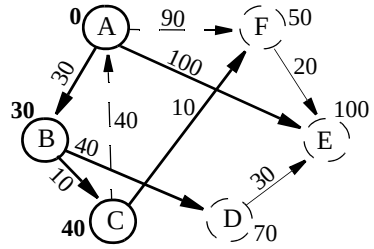
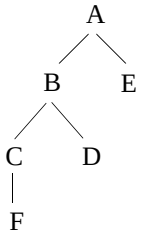
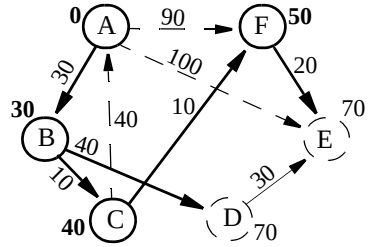
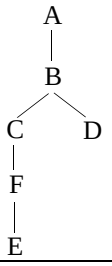
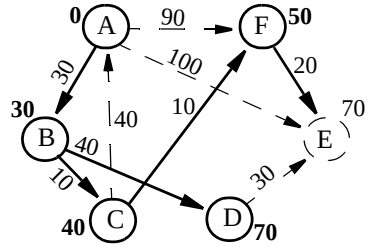
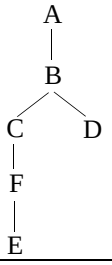
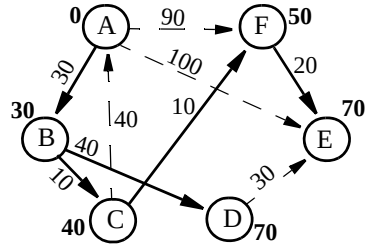
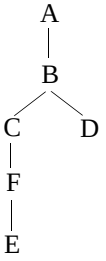
Der Algorithmus beginnt mit *v* als einzelnem gelben Knoten.
Schrittweise wächst der Teilgraph dadurch, dass derjenige gelbe Knoten grün gefärbt wird, der den minimalen Abstand zu *v* besitzt. Alle noch nicht betrachteten Nachfolger dieses Knotens werden als gelbe Knoten und die Verbindungskanten als rote Kanten in den Teilgraphen aufgenommen. Für bereits gelb markierte Nachfolgerknoten ist der bisher bekannte kürzeste Pfad (rote Kanten) gegebenenfalls zu korrigieren.
Der Algorithmus terminiert, wenn alle von *v* aus erreichbaren Knoten in *G* bearbeitet (grün gefärbt) sind.

Beispiel:

Gegeben sei der folgende Graph:



Sei A der Knoten, für den die kürzesten Wege zu bestimmen sind.
Im folgenden zeigen wir die Situationen, die sich jeweils nach einer Markierung des nächsten gelben Knotens als 'grün' ergeben:
Die gestrichelten Kanten stellen dabei die gelben Kanten des Teilgraphen dar, rote Kanten sind als fett und die restlichen Kanten durch normale Striche gekennzeichnet. Grüne Knoten sind fett, gelbe Knoten gestrichelt, die restlichen Knoten normal dargestellt.

Schritt	grün (gelb) markiert:	Ausgangsgraph mit markierten Knoten und Kanten	Baum der kürzesten Wege (➙ rote Kanten)
1	A (B,E,F)		
2	B (C,D)		
3	C (F korrig.)		
4	F (E korrig.)		
5	D		
6	E		

Betrachten wir nun den Algorithmus von Dijkstra in Java-Notation. Dabei gelten die folgenden Vereinbarungen:

- der Startknoten $v \in V$ wird beim Aufruf des Algorithmus spezifiziert.
- dist[w] ordnet jedem Knoten *w* seinen (bisherigen) minimalen Abstand zu *v* zu.
- neighbors(*w*) liefert die Menge aller Nachfolger von *w* (z.B. über Adjazenzliste).
- cost(*w, w'*) sei das Kostenmaß der Kante von *w* nach *w'*.

Weiterhin seien für den Algorithmus die Variablen '*green'* und '*yellow*' vom abstrakten Typ *NodeSet* vereinbart. Insbesondere die geeignete Organisation der Knotenmenge '*yellow*' ist für die Laufzeit des Algorithmus von Bedeutung.

```
public void dijkstra ( int v ) {
    int  dist[] = new int [n];
    int w, w';
    NodeSet green, yellow, allNodes;
    green = new NodeSet(); yellow = new Nodeset();
    yellow.insert(v);
    dist[v] = 0;
    while (!yellow.isEmpty()) {
(1)     wähle den Knoten w aus yellow mit minimalem Wert dist[w];
        green.insert(w); yellow.delete(w);      // w grün färben
(2)     for (each  w' aus neighbors(w)) { // über Adjazenzliste / Adjazenzmatrix
            allNodes = new NodeSet();
            allNodes.union(yellow); allNodes.union(green);
            if (! allNodes.contains(w') { // Knoten w' noch nicht besucht
                    färbe die Kante (w, w') rot;   (* bisher kürzester Weg *)
                    yellow.insert (w');
                    dist[w'] = dist[w] + cost(w, w');
                    }
            else if (yellow.contains(w')) { // w' über einen neuen Weg erreicht
                    if (dist[w'] > dist[w] + cost(w, w')) {
                                färbe die Kante (w, w') rot;
                                färbe die bisher rote Kante zu w' gelb;
                                dist[w'] = dist[w] + cost(w, w');
                                }
                    else  färbe (w, w') gelb
            else // w' bereits endgültig bearbeitet (w' in green)
                    färbe (w, w') gelb
            } // end if
        } // end for
    } // end while
} // end dijkstra
```

Bemerkung: Bei Terminierung des Algorithmus sind alle von *v* aus erreichbaren Knoten grün. Es können jedoch nicht erreichbare Knoten auftreten; diese bleiben ungefärbt.

Laufzeitanalyse

Die Laufzeitanalyse hängt ab von den zugrundeliegenden Datenstrukturen. Wir betrachten im folgenden zwei Varianten:

(I) Der Graph *G* sei über eine Kostenmatrix *cost[i, j]* (mit Einträgen '∞' für nicht-existierende Kanten) repräsentiert; die Kostenwerte *dist[i]* seien über ein Array realisiert (initialisiert als '∞').

Weiterhin werden zwei Arrays '*father*' und '*green*' verwendet:

- '*father[i]*':
  repräsentiert den Vorgängerknoten von Knoten *i* im Baum der roten Kanten und damit die roten Kanten selbst.
- gelbe Kanten sind für kürzeste Wege irrelevant
  ➞ nicht explizit gespeichert.
- '*green[i]*':
  boolescher Wert, der angibt, ob Knoten *i* bereits grün gefärbt wurde.
  ➞ Implementierung von *'green.contains(w) ⇔ green[w] = true*'
- die Menge der gelben Knoten ist implizit (*dist[i]* < '∞') gegeben.

Der Aufwand des Algorithmus setzt sich dann zusammen aus:

(1) Durchlauf des Arrays *dist*, um den gelben Knoten mit minimalem Abstand zum Ausgangsknoten zu finden
  ➞ Aufwand $O(n)$.

(2) Durchlauf einer Zeile der Matrix *cost[i,·]*, um alle Nachfolger des neuen grünen Knotens zu finden; *dist*-Wert sowie *father*-Eintrag werden gegebenenfalls korrigiert
  ➞ Aufwand $O(n)$.

➞ Da (1) und (2) *n*-mal ausgeführt werden:
  Gesamtaufwand des Algorithmus: $O(n^2)$.

(II) G ist durch Adjazenzlisten mit Kosteneinträgen dargestellt.

Weiterhin gebe es Arrays '*dist*', '*father*' und '*green*' wie oben.
Der NodeSet '*yellow*' der gelben Knoten sei nun jedoch über einen partiell geordneten Heap mit dem Abstand als Ordnungskriterium verwaltet. Die Heap-Einträge seien dabei doppelt verkettet und ein weiteres Array '*heapaddress*' enthalte für jeden gelben Knoten dessen Position im Heap (für effizientes update der Kosten).

Jeder Einzelschritt des Algorithmus setzt sich dann zusammen aus:

(1) Entfernen des gelben Knotens mit minimalem Abstand aus dem Heap
→ Aufwand $O(\log n)$.

n-mal ausgeführt → Gesamtaufwand für (1): $O(n \cdot \log n)$.

(2) Finden der m' Nachfolger über die Adjazenzliste: Aufwand $O(m')$.

- 'neue' Nachfolger: Einfügen als gelbe Knoten in den Heap ($O(\log n)$).
- bereits gelbe Nachfolger: gegebenenfalls Korrektur der Einträge im Heap (über '*heapaddress*' und doppelte Verkettung: $O(\log n)$).
- bereits grün gefärbte Nachfolger: werden übergangen.
→ Aufwand insgesamt: $O(m' \cdot \log n)$.

Da $\sum m' = |E|$ ergibt sich ein Gesamtaufwand über alle Schritte des Algorithmus für (2) von: $O(|E| \cdot \log n)$.

→ Gesamtaufwand des Algorithmus: $O(|E| \cdot \log n)$.
im Allgemeinen: $|E| > n$

Vergleich der beiden Implementierungen

- Die Implementierung über Adjazenzlisten ist effizienter, falls $|E| \ll n^2$.
- Die Implementierung über Adjazenzlisten hat einen Speicherplatzbedarf von nur $O(n + |E|)$ im Vergleich zu $O(n^2)$ für die Implementierung mit einer Kostenmatrix.

4.3.2 All Pairs Shortest Path-Problem

Problem: Vorgaben wie beim *single source shortest path*-Problem.
Bestimmung der kürzesten Wege zwischen *allen* Paaren von Knoten des Graphen *G*, zwischen denen eine Verbindung besteht.

Das Problem kann trivialerweise durch iterative Anwendung des Algorithmus von Dijkstra auf alle Knoten des Graphen gelöst werden. Wir stellen hier jedoch einen wesentlich einfacheren Algorithmus vor, der das Problem direkt löst:

Der Algorithmus von *Floyd*

Prinzip: Die Knoten des Graphen *G* seien mit 1 bis *n* durchnumeriert.

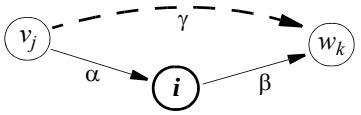
Der Algorithmus berechnet eine Folge $G_0 = G, G_1, \ldots, G_n$ von Graphen, wobei Graph G_i, $1 \le i \le n$, wie folgt definiert ist:

- G_i besitzt die gleiche Knotenmenge wie G
- in G_i existiert eine Kante von Knoten *v* nach Knoten *w* mit Kosten α

  $\Leftrightarrow$ es gibt einen Pfad von *v* nach *w*, der nur Knoten aus $\{1, \ldots, i\}$ verwendet und der kürzeste dieser Pfade hat die Länge α.

G_i entsteht durch folgende Modifikation von G_{i-1} im *i*-ten Schritt des Algorithmus:

Seien $v_1, \ldots, v_r$ die Vorgänger und $w_1, \ldots, w_s$ die Nachfolger von Knoten *i* im Graphen G_{i-1}, so betrachtet man alle Paare (v_j, w_k), $1 \le j \le r$ und $1 \le k \le s$.



- falls noch keine Kante von v_j nach w_k existiert, so erzeuge eine entsprechende Kante.
- existiert bereits eine solche Kante mit Kosten γ, so ersetze γ durch $\alpha + \beta$, falls $\alpha + \beta < \gamma$.

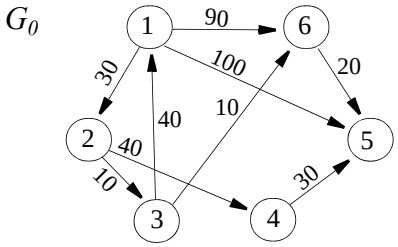
G_i erfüllt dabei wiederum obige Eigenschaften, denn:

- in G_{i-1} waren alle kürzesten Pfade durch Kanten repräsentiert, die nur auf Basis der Zwischenknoten $\{1, \ldots, i-1\}$ entstanden sind.
- jetzt sind alle Pfade bekannt, die Knoten aus $\{1, \ldots, i\}$ benutzen.

Nach *n* Schritten sind in Graph G_n die Kosten aller kürzesten Wege von Graph *G* repräsentiert.

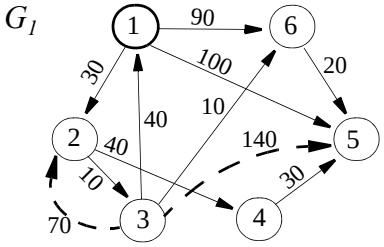
Beispiel

Gegeben sei der Beispielgraph von oben, wobei die Knoten numeriert sind:



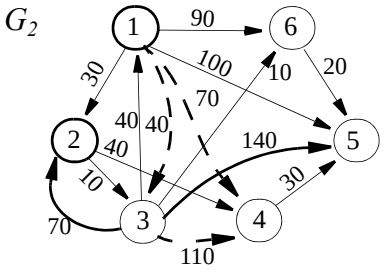
Im ersten Schritt werden alle Paare von Vorgängern (3) und Nachfolgern (2,5,6) von Knoten ‘1’ betrachtet. In den Graphen G_0 werden dabei die folgenden Kanten eingefügt, resultierend im Graphen G_1:

Kante (3,2) mit Kosten 70
und Kante (3,5) mit Kosten 140.



Im nächsten Schritt betrachten wir innerhalb von G_1 die Vorgänger (1,3) und Nachfolger (3,4) von Knoten ‘2’. Hinzugefügte Kanten (resultierend in Graph G_2) sind hierbei:

Kante (1,3) (Kosten 40),
Kante (1,4) (Kosten 70),
Kante (3,4) (Kosten 110).



Denselben Vorgang wiederholen wir nun für die Knoten ‘3’, ‘4’, ‘5’ und ‘6’:

Knoten	Vorgänger	Nachfolger	Neue Kanten	Veränderte Kanten
'3'	1, 2	1, 2, 4, 5, 6	(2,1): Kosten 50 (2,5): Kosten 150 (2,6): Kosten 20	(1,6): Kosten 50
'4'	1, 2, 3	5		(2,5): Kosten 70
'5'	1, 2, 3, 4, 6	--		
'6'	1, 2, 3	5		(1,5): Kosten 70 (2,5): Kosten 40 (3,5): Kosten 30

Als Resultat ergibt sich ein gerichteter Graph G_6, der durch die folgende Kostenmatrix beschrieben wird:

G_6	1	2	3	4	5	6
1	--	30	40	70	70	50
2	50	--	10	40	40	20
3	40	70	--	110	30	10
4	--	--	--	--	30	--
5	--	--	--	--	--	--
6	--	--	--	--	20	--

Anmerkung: Die Knoten '1', '2', '3' und '6' sind von Knoten '4' aus und die Knoten '1' bis '4' sind von Knoten '6' aus nicht erreichbar; von Knoten '5' aus erreicht man keinen anderen Knoten. Die entsprechenden Kanten existieren daher im Ergebnisgraphen G_6 nicht; es gibt keine (und damit keine kürzeste) zugehörige Verbindung.

Im folgenden betrachten wir eine konkrete, vereinfachte **Implementierung des Algorithmus von Floyd** auf Basis der Implementierung des Graphen mit Hilfe einer Kostenmatrix.

Sei C die Kostenmatrix für den Ausgangsgraphen $G = (V, E)$. Diese sei als globale Variable vorgegeben. Dabei sei vereinbart, dass $C[i, i] = 0$, $1 \le i \le n$, und $C[i, j] = \infty$, falls es keine Kante von Knoten i nach Knoten j in G gibt.

```
double [][] floyd () {
    double [][]a= new double [n][n];
    Zuweisung der Kostenmatrix an Matrix a (by value !)
        for (int i = 1; i < n; i++) {          // aktueller Knoten i
            for (int j = 1; j < n; j++) {      // Vorgänger j betrachten
                for (int k = 1; k < n; k++) // Nachfolger k betrachten
                    if (a[j][i]+ a[i][k]< a[j][ k]) // kürzerer Weg über i gefunden
                        a[j] [k]= a[j][i] + a[i][k];
            }
        }
    return a;
    }
```

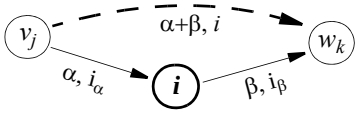
Die Laufzeit für diese Implementierung ist offensichtlich $O(n^3)$.

Im Laufe des Algorithmus erhöht sich die Kantenzahl zum Teil deutlich und kommt in der Regel der Maximalzahl von n^2 recht nahe. Daher kann auch für eine Darstellung über Adjazenzlisten kein entscheidend verbessertes Laufzeitverhalten erwartet werden. Aufgrund ihrer Einfachheit ist daher die obige Implementierung im Allgemeinen zu bevorzugen.

Bisher: Berechnung ausschließlich der Kosten der kürzesten Pfade.

Jetzt: Repräsentation der Pfade selbst mittels geeigneter zusätzlicher Kantenmarkierungen.

Konkret markieren wir beim Übergang von Graph G_{i-1} auf G_i (s.o.) die neu hinzukommenden bzw. geänderten Kanten mit Knoten *i*, d.h. mit dem Knoten, über den der (neue) minimale Weg verläuft.



Hierzu benötigen wir eine zusätzliche (zu Beginn mit 0-Feldern initialisierte) Matrix:

```
int [][] pathCost = new int [n][n];
```

In der **if**-Anweisung des obigen Algorithmus von Floyd wird der folgende zusätzliche Befehl eingefügt:

```
pathCost [j][k] = i;
```

Nach Terminierung des Algorithmus enthält die Matrix '*pathCost*' die folgende Information:

- pathCost [i][j] = 0
  ➞ die Kante zwischen Knoten i und j ist kostenminimal, falls sie existiert; ansonsten: kein Weg von Knoten i nach Knoten j vorhanden.
- pathCost [i][j] = $k > 0$
  ➞ gibt an, über welchen Knoten k die kürzeste Verbindung verläuft (rekursiv!).

Aus dieser Matrix lässt sich der kürzeste Weg zwischen zwei beliebigen Knoten i und j in $O(r)$ Zeit konstruieren, wobei r die Anzahl der Knoten auf dem kürzesten Weg bezeichnet.

4.4 Transitive Hülle

Häufig ist (unabhängig von den Kosten) nur von Interesse:
'existiert ein Weg in Graph G von Knoten i nach Knoten j ?', oder allgemeiner:

Problem: Welche Knoten eines Graphen sind von welchen anderen Knoten aus durch einen Weg erreichbar? (*Zusammenhangsproblem* oder *Erreichbarkeitsproblem*)

Definition: *Transitive Hülle*

Gegeben sei ein Graph $G = (V, E)$. Die *transitive Hülle* von G ist der Graph $\overline{G} = (V, \overline{E})$, wobei $\overline{E}$ definiert ist durch: $(v, w) \in \overline{E} \Leftrightarrow$ es gibt einen Weg in G von v nach w .

Die Lösung stellt im wesentlichen eine Vereinfachung des Algorithmus von Floyd dar:

Der Algorithmus von *Warshall*

Nehmen wir an, G sei durch eine (boolesche) Adjazenzmatrix $A[i, j]$ gegeben. Hieraus wird eine Adjazenzmatrix $\overline{A}[i, j]$ für $\overline{G}$ generiert:

Prinzip: Ähnlich zum Algorithmus von Floyd wird beginnend mit G eine Folge von Graphen $G, G_1, \ldots, G_n$ mit einer zunehmenden Menge von Kanten generiert.

Die Einträge $A_i[j, k]$ der Adjazenzmatrix A_i von Graph G_i, $1 \le i \le n$, entstehen dabei aus der Adjazenzmatrix A_{i-1} von Graph G_{i-1} wie folgt:

$$A_i[j, k] := A_{i-1}[j, k] \text{ OR } (A_{i-1}[j, i] \text{ AND } A_{i-1}[i, k]) \quad .$$

In der resultierenden Adjazenzmatrix A_n schließlich sind genau diejenigen Einträge $A_n[j, k]$ mit **true** belegt, für die gilt: ‘es gibt einen Weg in *G* von Knoten *j* nach Knoten *k*’. G_n repräsentiert somit die transitive Hülle von *G*.

Die folgende Implementierung entspricht der des Algorithmus von Floyd mit Ausnahme der **if**-Anweisung im Inneren der geschachtelten Schleifen:

```
booelan [][] warshall () {
    boolean a[][] = new boolean [n][n];
    // Zuweisung der Adjazenzmatrix an Matrix a (by value !)
      for (int i = 1; i < n; i++) {          //aktueller Knoten i
          for (int j = 1; j < n; j++) {      // Vorgänger j betrachten
              for (int k = 1; k < n; k++) // Nachfolger k betrachten
                  if (!a[j][k])// Kante (j, k) existiert noch nicht
                      a[j][k] = (a[j][i] && a[i][k]);
          }
      }
    return a;
    }
```

4.5 Minimaler Spannbaum

Definition: *Spannbaum eines Graphen*

Gegeben sei ein zusammenhängender, ungerichteter Graph $G = (V, E)$.

Ein *Spannbaum* von *G* ist ein Teilgraph $\overline{G} = (V, \overline{E})$, bei dem zwei (und damit alle drei) der folgenden Bedingungen erfüllt sind:

(1) $\overline{G}$ ist zusammenhängend.

(2) $\overline{G}$ besitzt $n - 1$ Kanten ($n = |V|$).

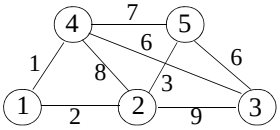
(3) $\overline{G}$ ist zyklenfrei.

Bemerkungen:

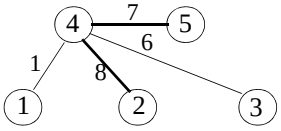
- Jeder zusammenhängende Graph besitzt mindestens einen Spannbaum.
- Die Anzahl verschiedener Spannbäume von *n* Knoten beträgt n^{n-2}.
  (ohne Beweis)

Definition: *Minimaler Spannbaum*

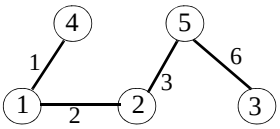
Sind den Kanten des Graphen *G* Kosten zugeordnet, so besitzt jeder Spannbaum Kosten, die sich aus der Summe seiner Kantenkosten ergeben.
Man spricht dann von einem *minimalen Spannbaum (kostenminimaler, zyklenfreier, zusammenhängender Teilgraph),* falls dessen Kosten minimal unter allen möglichen Spannbäumen von G sind.



Graph G_1



ein Spannbaum
Gesamtkosten = 22



ein *minimaler* Spannbaum
Gesamtkosten = 12

Es gibt zwei prinzipielle Ansätze, einen minimalen Spannbaum zu ermitteln:

Grow-Algorithmus:

Beginne mit einer leeren Kantenmenge;
solange noch kein Spannbaum gefunden ist:
➙ füge der Kantenmenge die Kante mit minimalen Kosten aus *G* hinzu, die keinen Zyklus erzeugt.

Shrink-Algorithmus:

Beginne mit allen Kanten des Graphen als Kantenmenge;
solange noch kein Spannbaum gefunden ist:
➙ entferne aus der Kantenmenge die Kante mit maximalen Kosten, die nicht die Zusammenhangseigenschaft verletzt.

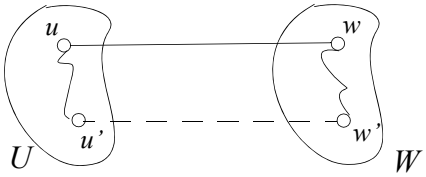
Der Algorithmus von *Kruskal*

Grundlage: das '*grow*-Prinzip'.

Vorbemerkung: (*ohne formalen Beweis*)

Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph und $\{U, W\}$ eine Zerlegung der Knotenmenge *V*. Sei (u, w), $u \in U$ und $w \in W$, eine Kante in *G* mit minimalen Kosten unter allen Kanten $\{(u', w') | u' \in U, w' \in W\}$. Dann gibt es einen minimalen Spannbaum für *G*, der (u, w) enthält.

Veranschaulichung:
(u',w') beliebig!

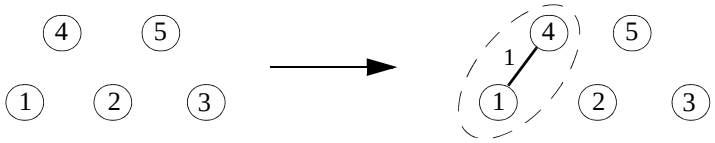


Kosten (u,w) ≤ Kosten (u',w')

Hieraus ergibt sich der Ansatz des Kruskal'schen Algorithmus, schrittweise anwachsende Teilmengen der Knotenmenge V zu betrachten und deren minimale Verbindungskante zu ermitteln:

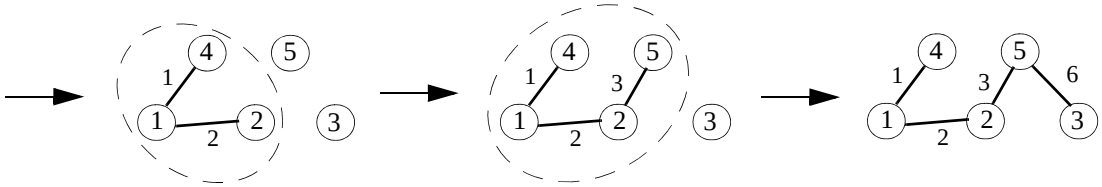
Prinzip: Zu Beginn wird jeder der Knoten als einelementige Teilmenge von V betrachtet.

Der Algorithmus betrachtet die Kanten in E nach aufsteigenden Kosten. Im ersten Schritt werden die beiden durch die kostenminimale Kante verbundenen Knoten zu einer zweielementigen Teilmenge verschmolzen.



Dieser Schritt wird $(n-1)$-mal iteriert, wobei jeweils diejenige Kante mit minimalen Kosten gewählt wird, die zwei bisher getrennte Knotenmengen verbindet; diese beiden Mengen werden verschmolzen.

Schließlich sind alle Knoten zu einer einzigen Knotenmenge vereinigt. Die schrittweise eingefügten Kanten bilden einen minimalen Spannbaum von *G*.



Die folgende informelle Beschreibung des Algorithmus basiert auf der Tatsache, dass kostenminimale Kanten schnell gefunden werden können. Dazu werden die Kanten nach aufsteigenden Kosten in einem dynamischen Heap '*K_Heap*' organisiert. Der Zugriff erfolgt mit Hilfe der Methode '*get_minimum*', wobei die kostenminimale Kante zurückgeliefert und gleichzeitig aus der Struktur gelöscht wird.

Weitere benötigte Datenstrukturen sind:

- '*Knoten_Sets*': organisiert die aktuellen Teilmengen der Knotenmenge.
  Methoden: '*find_component*' und '*merge_components*'.
- '*Spann_Graph*': zweite Graphstruktur mit gleicher Knotenmenge wie *G*.
  Aufbau des minimalen Spannbaumes.
  Methode: '*insert_edge*'.

```
ALGORITHMUS Kruskal;
    VAR   Schrittzahl : CARDINAL;
          v, w : Knoten;
          a, b : Component;     (* abstrakter Datentyp für 'Knoten_Sets' *)
    BEGIN
      initialisiere Knoten_Sets so, dass jeder Knoten eine eigene Komponente bildet;
      füge alle Kanten aus E gemäß ihrer Kosten in K_Heap ein;
      initialisiere die Kantenmenge von Spann_Graph als leer;
      Schrittzahl := 1;
      WHILE Schrittzahl < n DO
        get_minimum ( K_Heap, v, w );
        a := find_component ( Knoten_Sets, v );
        b := find_component ( Knoten_Sets, w );
        IF a ≠ b THEN    (* andernfalls wird die Kante einfach übergangen *)
            insert_edge ( Spann_Graph, v, w );  (* minimalen Spannbaum aufbauen *)
            merge_components ( Knoten_Sets, a, b );
            INC ( Schrittzahl );
        END
      END
    END Kruskal;
```

Laufzeitanalyse:

Sei $n = |V|$ und $e = |E|$, dann besitzt obiger Algorithmus das folgende Laufzeitverhalten:

- Initialisierung von '*Spann_Graph*' und '*Knoten_Sets*': $O(n)$
- Initialisierung von '*K_Heap*': $O(e)$
- maximal *e* Operationen 'get_minimum': $O(e \cdot \log e)$
- maximal *2e* Operationen 'find_component': $O(e \cdot \log n)$
- *n-1* Operationen 'merge_components': $O(n \cdot \log n)$

➞ Gesamtlaufzeit: $O(e \cdot \log e)$,

da $e \geq n-1$ (*G* ist zusammenhängend).