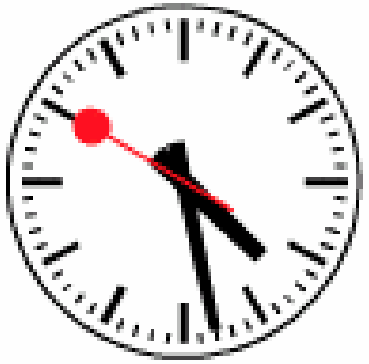
Graphen

Anwendung, Repräsentation, Tiefensuche, Breitensuche, Warshall‘s Algorithmus, kürzeste Wege.

Klausurtermine

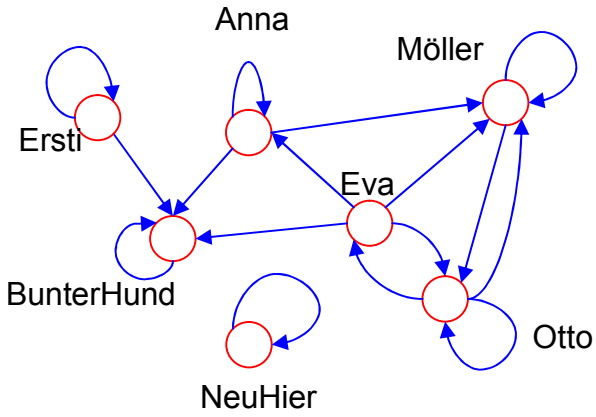


- Nachklausur
  - Do. 11.10.07, 9-12 Uhr HS V
- Abschlussklausur
  - Di. 17.7.07 im AudiMax
  - bisherige Uhrzeit: 16:00 – 19:00
  - **geplante Verschiebung:** 13:00 – 16:00

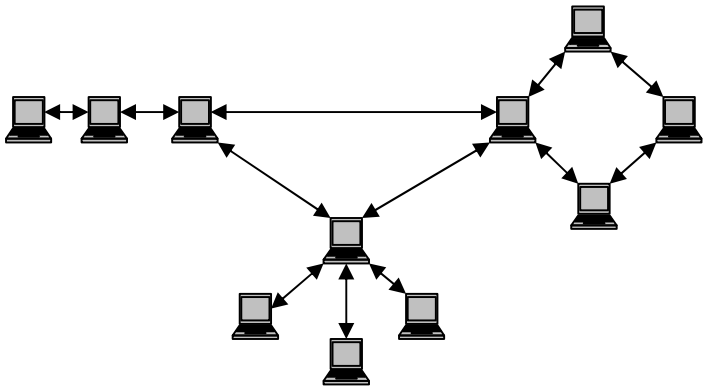
- Falls Sie Einwände gegen die Vorverlegung haben, teilen Sie uns diese bis Montag (unter Angabe von Gründen) mit.
  - gumm@mathematik.uni-marburg.de
  - fohry@mathematik.uni-marburg.de

Graphen

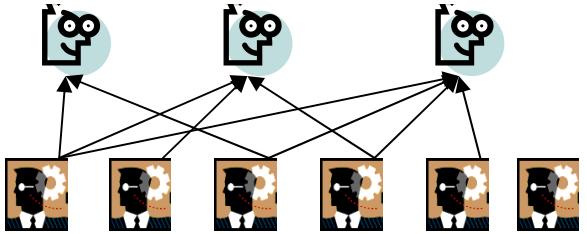
- ... im Sozialen Bereich
  - V = Personen in Marburg
  - $v_1 \rightarrow v_2 \quad :\Leftrightarrow$ v_1 kennt v_2



- Netze
  - V = Rechner im Fachbereich
  - $v_1 \rightarrow v_2 \quad :\Leftrightarrow$ v_1 ist direkt verbunden mit v_2

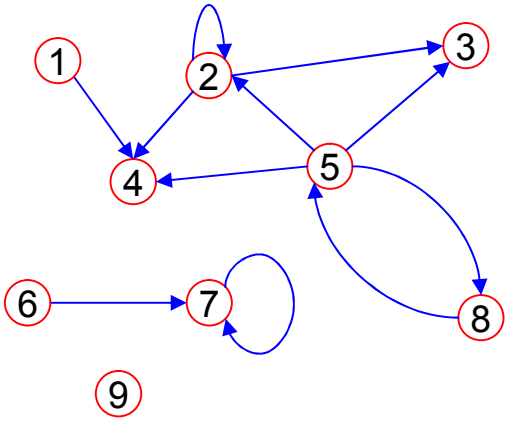


- ... mit mehreren Sorten
  - V = Professoren $\cup$ Studenten
  - $v_1 \rightarrow v_2 \quad :\Leftrightarrow$ v_1 hört bei v_2



Graph - die Definition

- Ein Graph G=(V,E) besteht aus
- einer Menge V von Knoten
  - dargestellt durch Punkte oder kleine Kreise
- einer Menge E von Kanten
  - Gerichtete Verbindungen zweier Knoten
  - dargestellt durch Pfeile



Ein Graph, bestehend aus
3 disjunkten Komponenten

Die englischen Bezeichnungen sind: *vertex* (Knoten) und *edge*(Kante)

Graph = Relation

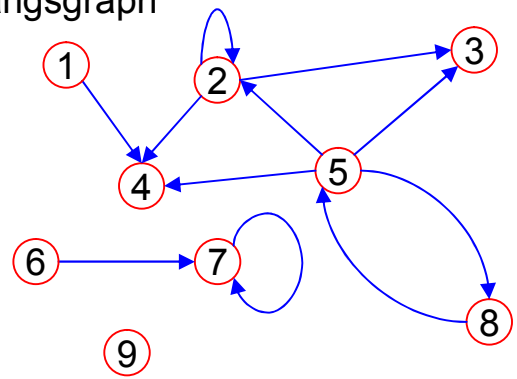
- Ein Graph definiert eine *zweistellige Relation* $R \subseteq V \times V$ auf der Knotenmenge
  - $R = \{ (v_1,v_2) \mid \text{Es ex. Kante k von } v_1 \text{ nach } v_2 \}$.

- Umgekehrt definiert jede zweistellige Relation $R \subseteq V \times V$ einen Graph G=(V,E) mit
  - $E = \{ (v_1,v_2) \mid (v_1,v_2) \in R \}$

- .. und aus der Relation $R \subseteq V \times V$ erhält man wieder den Graph G=(V,E) zurück mit

$$v_1 \xrightarrow{k} v_2 \quad :\Leftrightarrow \quad (v_1,v_2) \in R$$

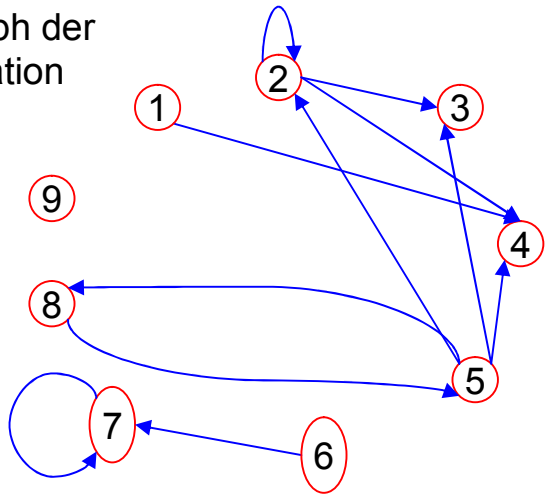
Ausgangsgraph



als Relation

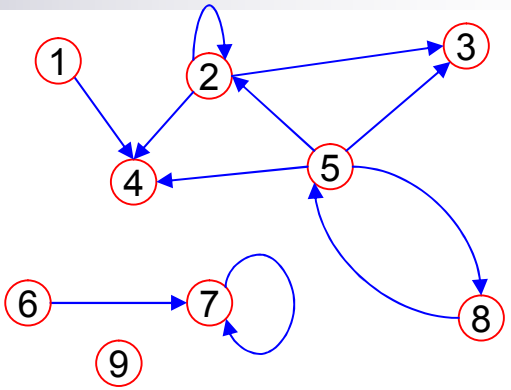
R = { (1,4), (2,2), (2,3), (2,4), (5,2),
(5,3), (5,4), (5,8), (6,7), (7,7), (8,5) }

Graph der Relation



... der gleiche Graph wie oben – nur anders gezeichnet

Graph = Tabelle

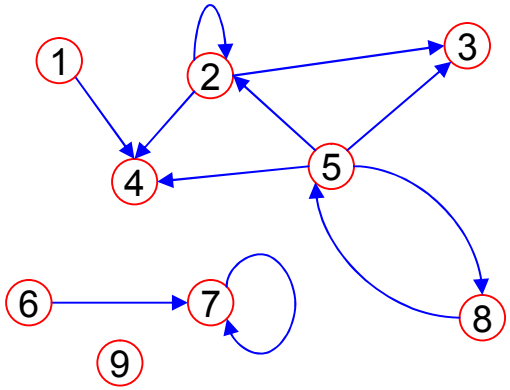


- Ein Graph kann als 0-1-Tabelle dargestellt werden
  - Spalten := Knoten
  - Zeilen := Knoten
  - i-te Zeile mit j-ter Spalte ist 1 $:\Leftrightarrow$ i $\xrightarrow{k}$ j
- Aus der Tabelle kann man den Graphen eindeutig rekonstruieren:
  - V := Spalten = Zeilen
  - $v_1 \xrightarrow{k} v_2 :\Leftrightarrow$ v_1-te Zeile mit v_2-ter Spalte ist 1
- Statt 0-1-verwendet man häufig auch true-false und erhält eine Boolesche Tabelle. Diese heißt: *Adjazenzmatrix*.
- Oft sind solche Matritzen „*dünn besetzt*“

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

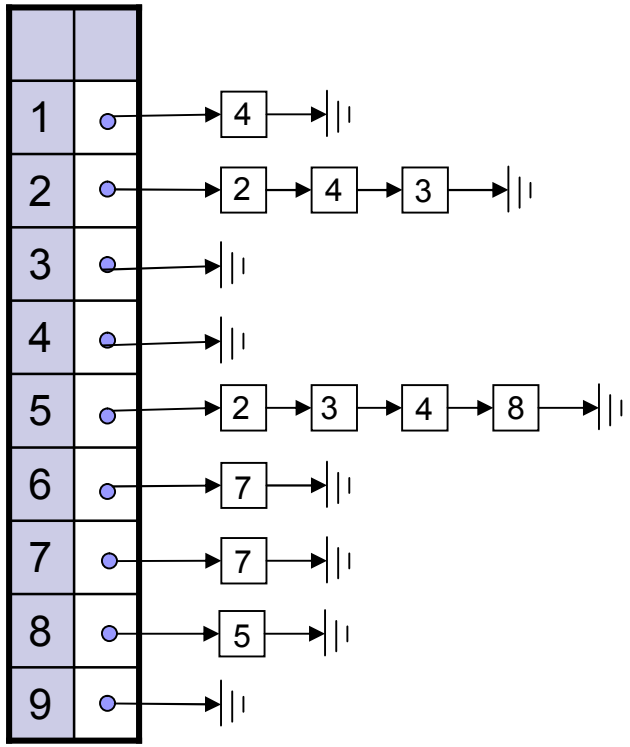
Der obige Graph als *Adjazenz*matrix.
Leere Felder sind 0.

Adjazenzlisten



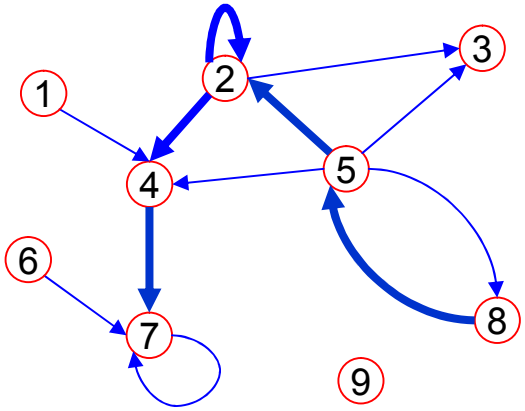
- Adjazenzmatrizen sind oft dünn besetzt
  - Speicherplatz wird verschwendet
  - Zuviele Tests if(verbunden[x][y]) sind erfolglos

- Adjazenzlisten:
  - Speichere für jeden Knoten die Menge seiner Nachbarn
  - ... z.B. als Listen
  - nachbarn(x) liefert Liste der Nachbarn



Pfade - Erreichbarkeit

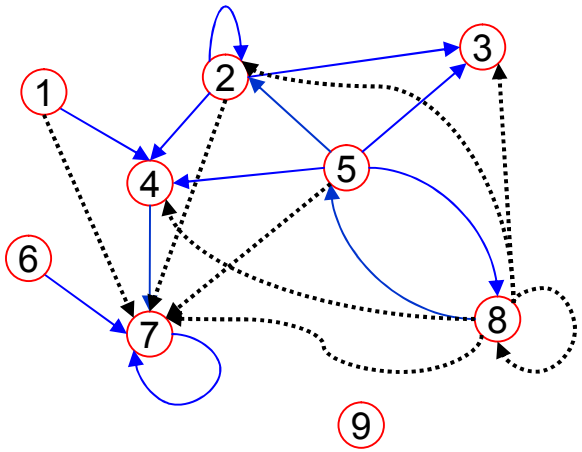
- Ein *Pfad* von a nach b ist eine Folge $v_0, v_1, \ldots, v_k$ von Knoten mit
  - $a=v_0$, $v_k=b$,
  - $\forall i<k \,.\, (v_i,v_{i+1}) \in R$.
- k heißt *Länge* des Pfades
- Falls es einen Pfad von a nach b gibt, so heißt b von a aus *erreichbar*.
- Der *Abstand* von a zu b ist die Länge des kürzesten Pfades von a zu b – falls ein solcher existiert.
- Die *transitive Hülle* eines Graphen hat zusätzlich jeweils eine Kante von a nach b, falls b von a erreichbar ist.



Der eingezeichnete Pfad von Knoten 8 nach Knoten 7 hat Länge 5.

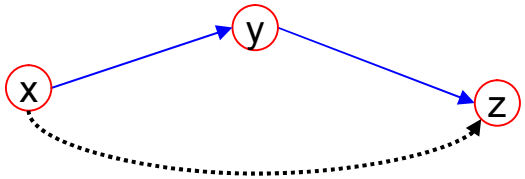
Knoten 8 ist von Knoten 7 aus nicht erreichbar.

Der Abstand von Knoten 8 nach Knoten 7 ist 3.



Berechnung der transitiven Hülle

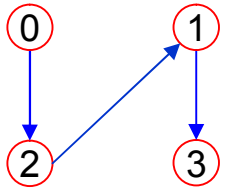
```
void warshall(){
    for(int y=0; y < table.length; y++)
      for(int x=0; x < table.length; x++)
        for(int z=0; z < table.length; z++)
          table[x][z] |= table[x][y] && table[y][z];
}
```

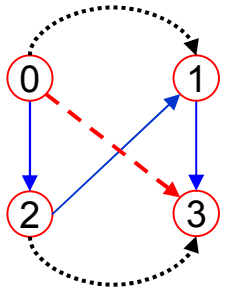


- Beachte die Reihenfolge der äußeren Schleifen !!
  - Wieso findet der Algorithmus in einem Durchlauf schon die komplette transitive Hülle ?
- Wenn Sie glauben dass das offensichtlich sei:
  - Warum funktioniert der folgende Algorithmus nicht ?
  - Es wurden nur die Schleifenvariablen (x und y) vertauscht ?

```
void fehlerhaft(){
    for(int x=0; x < table.length; x++)
      for(int y=0; y < table.length; y++)
        for(int z=0; z < table.length; z++)
          table[x][z] |= table[x][y] && table[y][z];
}
```

Ausgangsgraph





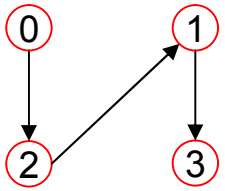
Kante 0-3 nicht gefunden

Lauf des korrekten Algorithmus

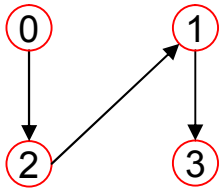
```
void warshall(){
    for(int y=0; y < table.length; y++)
      for(int x=0; x < table.length; x++)
        for(int z=0; z < table.length; z++)
          table[x][z] |= table[x][y] && table[y][z];
}
```

y = 0,1,2,3

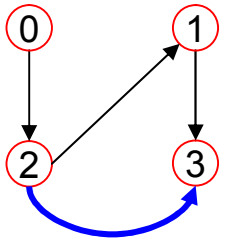
Ausgangsgraph



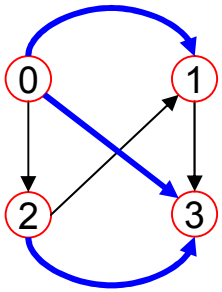
y=0



y=1



y=2



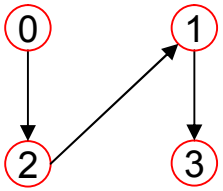
Alle Kanten gefunden

Lauf des fehlerhaften Algorithmus

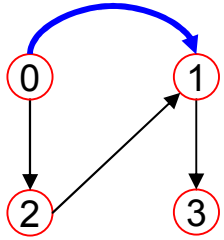
```
void fehlerhaft(){
    for(int x=0; x < table.length; x++)
      for(int y=0; y < table.length; y++)
        for(int z=0; z < table.length; z++)
          table[x][z] |= table[x][y] && table[y][z];
}
```

x = 0,1,2,3

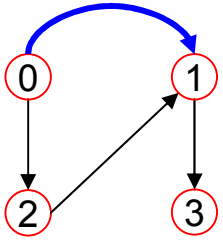
Ausgangsgraph



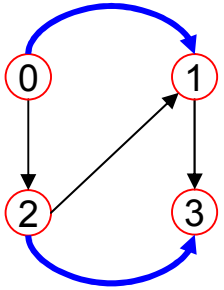
x=0



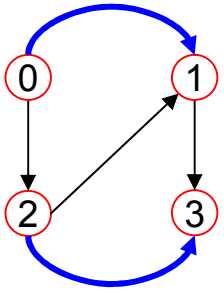
x=1



x=2



x=3

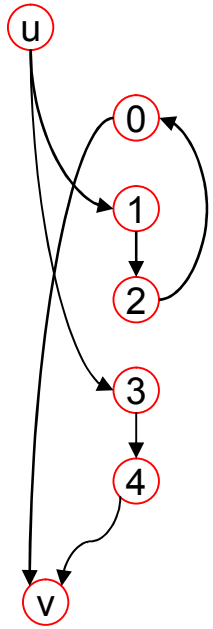


Kante 0-3 nicht gefunden

Analyse von Warshalls Algorithmus

```
void warshall(){
    for(int y=0; y < table.length; y++)
      for(int x=0; x < table.length; x++)
        for(int z=0; z < table.length; z++)
          table[x][z] |= table[x][y] && table[y][z];
}
```

- Korrektheitsbeweis per Induktion über y:
  - Ind.Hyp. P(y):
    - Für alle Knoten u,v gilt:
      - **Gibt es einen Pfad von u nach v, so dass alle Zwischenknoten in { 0, 1, …, y } liegen, so gilt nach dem (y+1)-ten Durchlauf der äußeren Schleife: table[u][v] == true;**

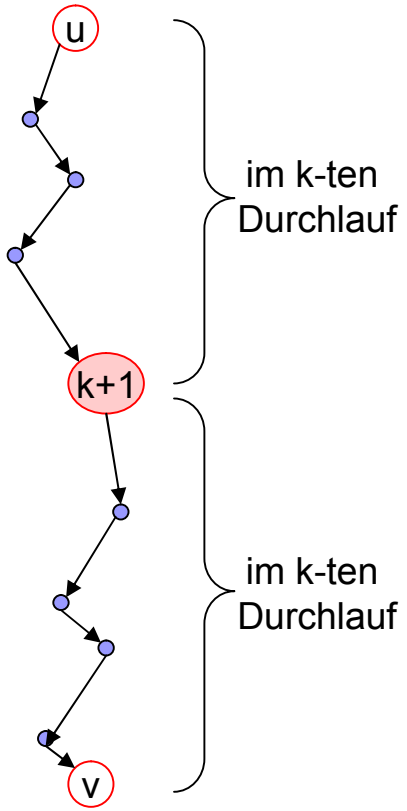


nicht der kürzeste,
wird aber im 3. Durchlauf
gefunden

Analyse von Warshalls Algorithmus

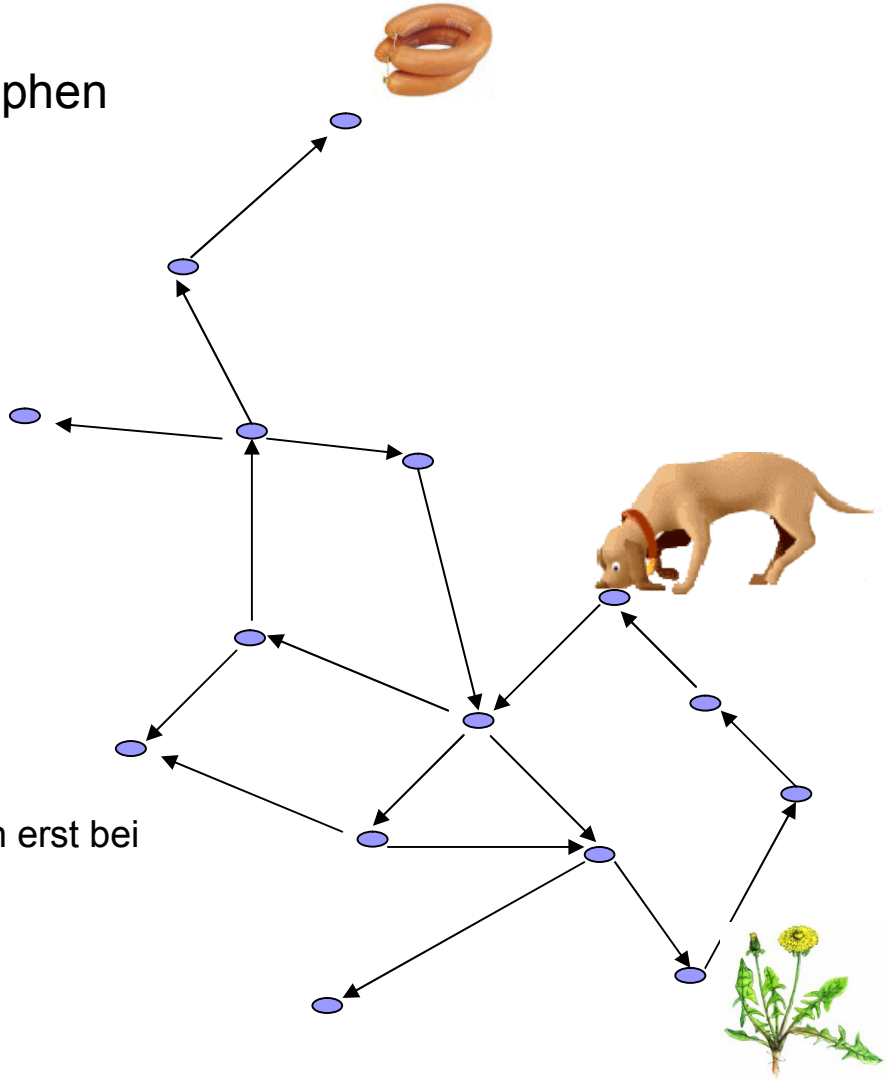
```
void warshall(){
    for(int y=0; y < table.length; y++)
      for(int x=0; x < table.length; x++)
        for(int z=0; z < table.length; z++)
          table[x][z] |= table[x][y] && table[y][z];
}
```

- Korrektheitsbeweis per Induktion über y:
  - Ind.Hyp. P(y):
    - Für alle Knoten u,v gilt:
      - **Gibt es einen Pfad von u nach v, so dass alle Zwischenknoten in { 0, 1, …, y } liegen, so gilt nach dem (y+1)-ten Durchlauf der äußeren Schleife: table[u][v] == true;**
  - y=0:
    - Aus table[u][0] & table[0][v] wird nach dem 0-ten Durchlauf table[u][v].
  - y=k+1:
    - Gibt es einen Weg von u nach v, der {0,1,…,k,k+1} benutzt, so gibt es auch einen, auf dem k+1 nur einmal vorkommt.
    - Er setzt sich zusammen aus
      - einem Weg von u nach k+1, der nur Zwischenknoten aus {0 .. k} benutzt
      - einem Weg von k+1 nach v, der nur Zwischenknoten aus {0 .. k} benutzt
    - Nach Ind.Hypothese gelten nach dem k-ten Durchlauf:
      - table[u][k+1] und table[k+1][v]
    - Im k+1-ten Durchlauf (y=k+1) erhalten wir dann: table[u][v] = true;



Suche im Graphen

- Systematisches Durchsuchen eines Graphen
  - Ausgehend von einem Startelement
  - Folge immer Kanten
  - bis Element gefunden
- Analog: Traveriserung
  - Besuche alle Knoten
- Sinnvoll in Graphen, die
  - als Adjazenzlisten gespeichert sind
  - dynamisch generiert werden
    - d.h. die Nachfolger eines Knotens werden erst bei Bedarf berechnet
- Wir nehmen an, dass die Nachbarn in irgendeiner Reihenfolge vorliegen
  - Nachbarn(x) = { x_1, ..., x_k }



Zyklen

- Ein Zyklus ist ein Pfad, der wieder zu seinem Ausgangspunkt zurückkehrt
  - Formal: Ein Pfad $s_0,...,s_n$ mit $s_0=s_n$.
- Zyklen sind bei der Suche gefährlich
  - Suche kann sich im Kreis drehen

- Lösung:
  - markiere besuchte Knoten
  - Wenn V={0,…,k-1} die Knotenmenge repräsentiert, z.B. durch
  - `boolean [] markiert = new boolean[k];`

wenn x besucht wird:

```
markiert[x] = true;
```

bzw.

`markiert[x]` ⇔ **x** schon besucht

Tiefensuche – depth first

Rekursiv

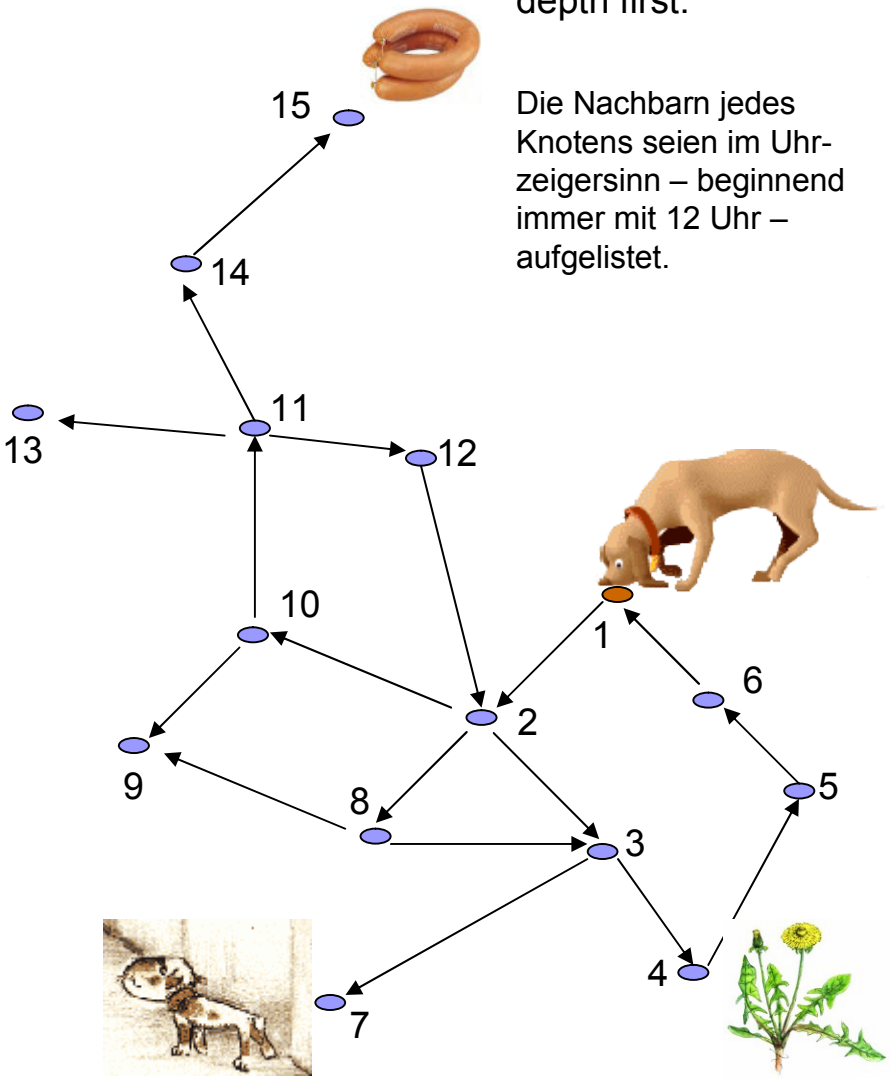
```
void tiefensuche(Knoten x):
  markiert[x] = true;
 for(Knoten k : nachbarn(x))
    if (! markiert[k]) tiefensuche(k);
```

Nicht-rekursiv, mit Stack s

```
void tiefensuche(Knoten x){
     Stack<Knoten> s = new Stack();
      markiert[x]=true;
      s.push(x);
      while (! s.isEmpty()){
        tuWasSinnvollesMit(y);
        y=s.getNext();
        for(Knoten k: nachbarn(y))
          if (! markiert[k]) {
              markiert[x]=true;
              s.push(k);
       } }
```

Besuchsreihenfolge : depth first.

Die Nachbarn jedes Knotens seien im Uhrzeigersinn – beginnend immer mit 12 Uhr – aufgelistet.

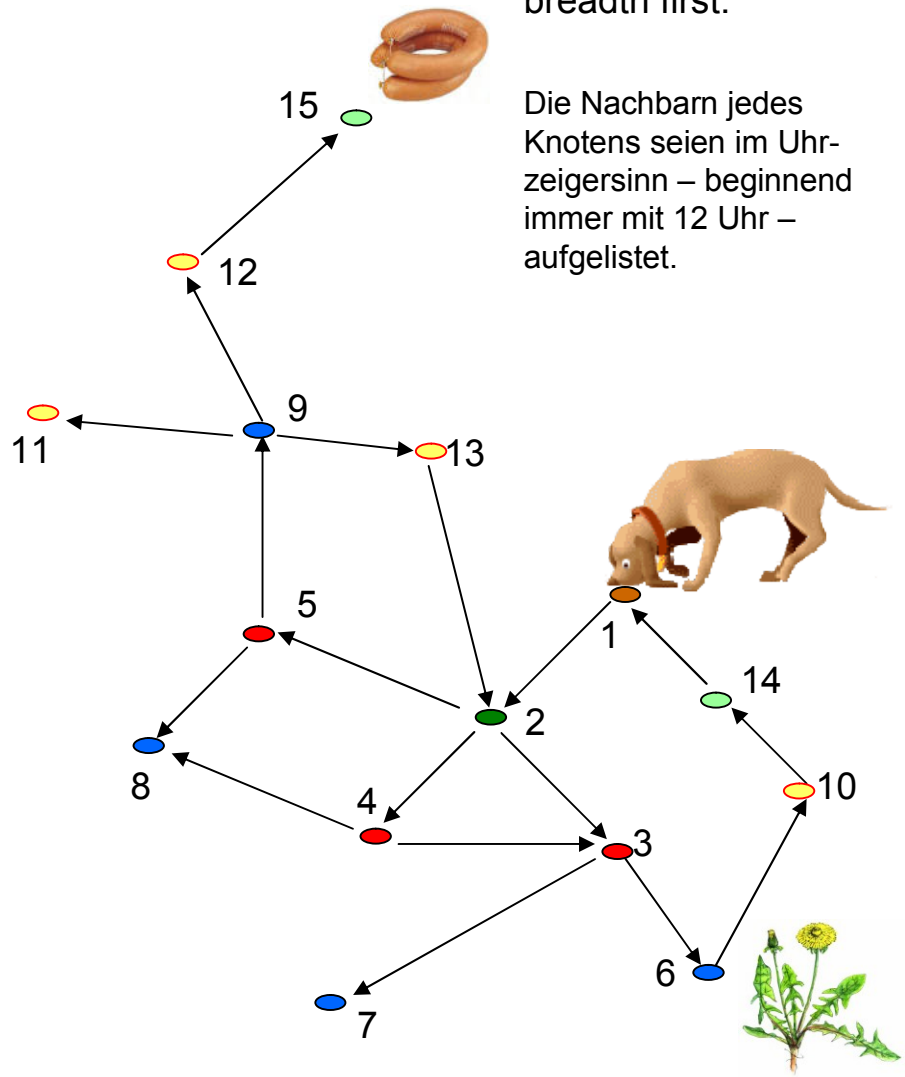


Breitensuche – breadth first

- Besuche alle Nachbarn
- dann alle Knoten mit Abstand 2
- ...
- Iterative Implementierung wie Tiefensuche, nur ..
- ersetze Stack durch Queue

Besuchsreihenfolge : breadth first.

Die Nachbarn jedes Knotens seien im Uhrzeigersinn – beginnend immer mit 12 Uhr – aufgelistet.

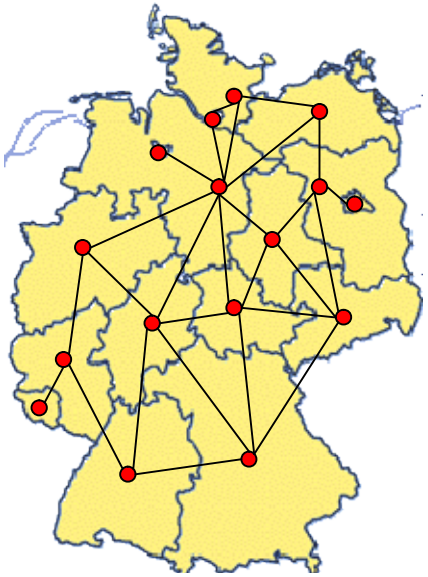


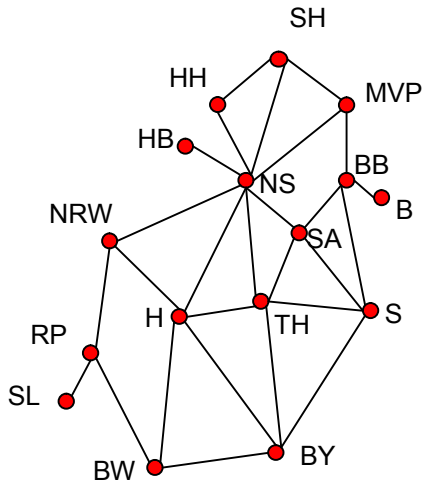
Nicht-rekursiv, mit Queue q

```
void breitensuche(Knoten x){
    Queue<Knoten> q = new Queue();
    markiert[x]=true;
    q.enQ(x);
    while (! q.isEmpty()){
      y=q.getNext();
      tuWasSinnvollesMit(y);
      for(Knoten k:nachbarn(y))
        if (! markiert[k])
            markiert[y]=true;
            s.enQ(k);
     }}
```

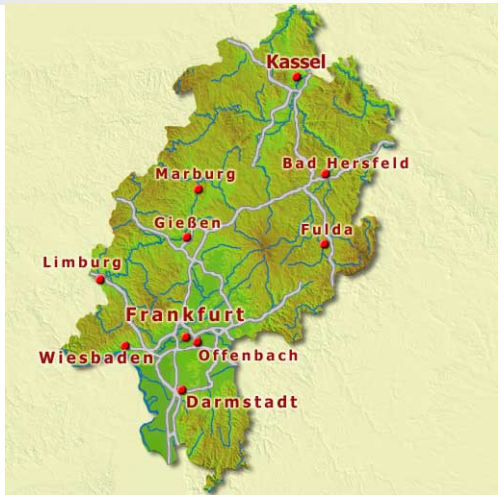
Ungerichtete Graphen

- repräsentieren symmetrische Relationen
  - $(x,y) \in R \Rightarrow (y,x) \in R$
  - $k_1 \rightarrow k_2 \quad \Rightarrow \quad k_1 \leftarrow k_2$.
- Statt zwei Kanten $k_1 \rightarrow k_2$ und $k_1 \leftarrow k_2$:
  - eine *ungerichtete Kante* $k_1 — k_2$.
- Adjazenzmatrix:
  - Symmetrisch
- Beispiel: Land k_1 *grenzt an* Land k_2.
- Graphen bieten für viele Fragen sinnvolle Abstraktionen

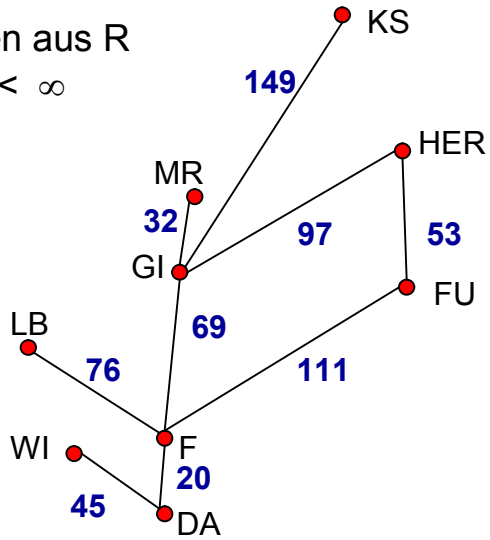




Bewertete Graphen



- Graphen G=(V,E) mit *Kantenbewertung*
  - $d : E \times E \rightarrow R$
- d kann z.B. stehen für
  - Distanz
  - Kapazität
- Graph kann gerichtet sein oder ungerichtet
- Darstellung:
  - Tabelle `distanz` mit Werten aus R
  - Kante existiert gdw. $d(k_1,k_2) < \infty$



	K	M	H	G	U	L	W	F	D
K	∞	∞	∞	149	∞	∞	∞	∞	∞
M	∞	∞	∞	32	∞	∞	∞	∞	∞
H	∞	∞	∞	97	53	∞	∞	∞	∞
G	149	32	97	∞	∞	∞	∞	∞	∞
U	∞	∞	53	∞	∞	∞	∞	111	∞
L	∞	∞	∞	∞	∞	∞	∞	76	∞
W	∞	∞	∞	∞	∞	∞	∞	∞	45
F	∞	∞	∞	∞	111	76	∞	∞	20
D	∞	∞	∞	∞	∞	∞	45	20	∞

∞ können wir in Java z.B. durch
`Integer.MAX_VALUE`
repräsentieren

Kürzeste Abstände(Floyd‘s Algorithmus)

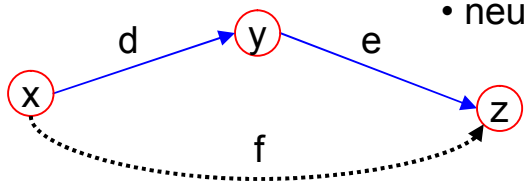
- Den kürzesten Abstand zwischen je zwei Punkten :
  - Leichte Modifikation von Warshall‘s Algorithmus
  - Noch kleine Effizienzsteigerung:
    Abfrage, die nicht von z abhängig ist,
    aus der inneren Schleife herausnehmen
  - Unsere Implementierung:
    Keine Verbindung: distanz[x][y] == Integer.MAX_VALUE

```
void floyd(){
    for(int y=0; y < distanz.length; y++)
      for(int x=0; x < table.length; x++)
        if(distanz[x][y] < Integer.MAX_VALUE)
          for(int z=0; z < table.length; z++)
            if(distanz[y][z] < Integer.MAX_VALUE)
              distanz[x][z] = Math.min(distanz[x][z],distanz[x][y]+distanz[y][z]);
}
```

Verbindung vorhanden

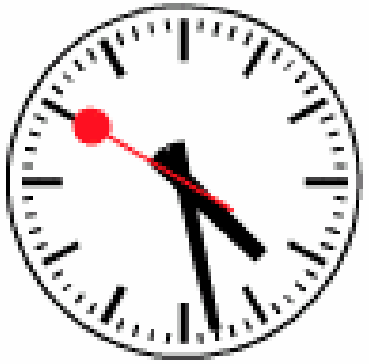
Minimum von
- bisheriger kürzeste Verbindung
- neu gefundener Verbindung

Abfrage
(distanz[u][v] < Integer.MAX_VALUE)
aus der for-Schleife gezogen
- wieso ??



f := min (f, d+e)

Klausurtermine



- Abschlussklausur
  - Di. 17.7.07 im AudiMax
  - **neue Uhrzeit :** 13:00 – 16:00
- Nachklausur
  - Do. 11.10.07, 9-12 Uhr HS V

Keine Hilfsmittel

- Ausweis mit Photo mitbringen
- Studentenausweis

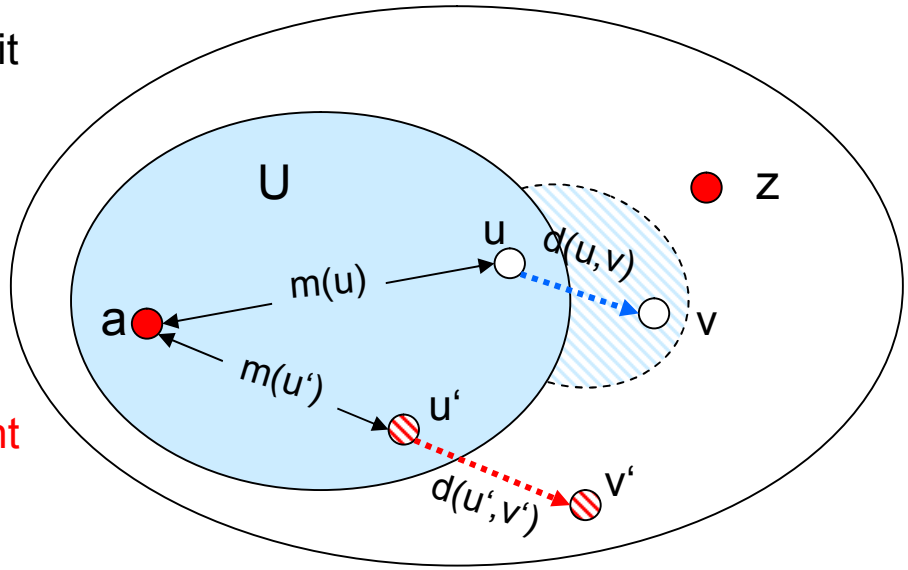
Dijkstra‘s Algorithmus

- Gegeben ein bewerteter Graph G=(V,E) mit Bewertung $d: V \times V \rightarrow R^+$.
- Von einem Knoten a berechne kürzesten Abstand $m_a(z)$ zu beliebigem Knoten z.

Betrachte eine Menge U mit Invariante:

- Inv: Für alle Knoten $u \in U$ ist $m_a(u)$ bekannt
  - Anfangs: $U = \{ a \}$ und $m_a(a)=0$.
  - … vergrößere U unter Beibehaltung von Inv.
  - … bis $z \in U$.
- Wie wird U vergrößert ?
  - Wähle $u \in U$ und $v \in U - V$ so dass

    $m_a(u) + d(u,v)$ minimal
  - $U := U \cup \{v\}$;
  - Speichere $m_a(v)$



Falls

$m_a(u)+d(u,v) \leq m_a(u‘)+d(u‘,v‘)$

für jedes $u‘ \in U$, $v‘ \in V-U$,

nehme v in U auf.

Eine schöne Animation findet sich bei

http://students.ceid.upatras.gr/~papagel/minDijk.htm

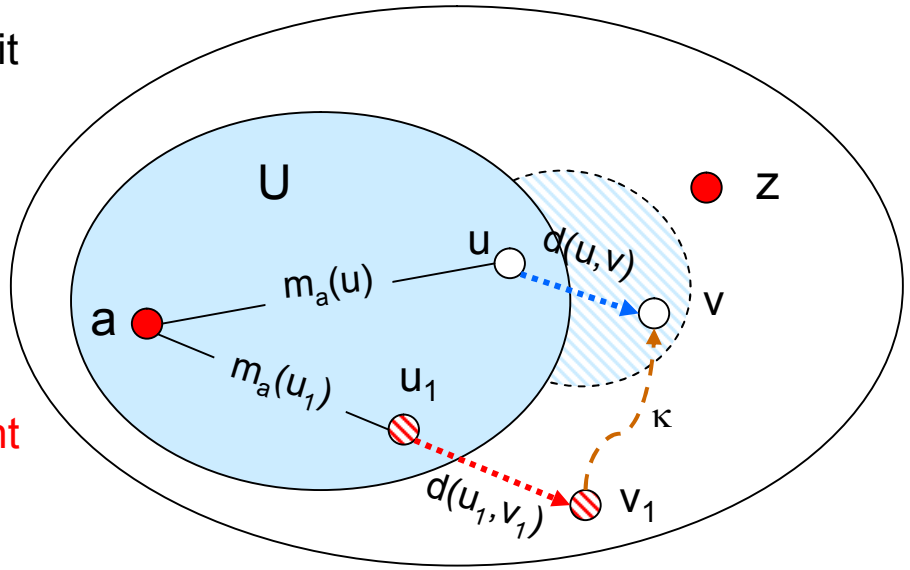
Dijkstra‘s Algorithmus

- Gegeben ein bewerteter Graph G=(V,E) mit Bewertung $d: V \times V \rightarrow R^+$.
- Von einem Knoten a berechne kürzesten Abstand $m_a(z)$ zu beliebigem Knoten z.

Betrachte eine Menge U mit Invariante:

- Inv: Für alle Knoten $u \in U$ ist $m_a(u)$ bekannt
  - Anfangs: $U = \{ a \}$ und $m_a(a)=0$.
  - … vergrößere U unter Beibehaltung von Inv.
  - … bis $z \in U$.
- Wie wird U vergrößert ?
  - Wähle $u \in U$ und $v \in V - U$ so dass

    $m_a(u) + d(u,v)$ minimal
  - $U := U \cup \{v\}$;
  - Speichere $m_a(v)$



Warum ist **a**, u, **v** kürzester Weg ?

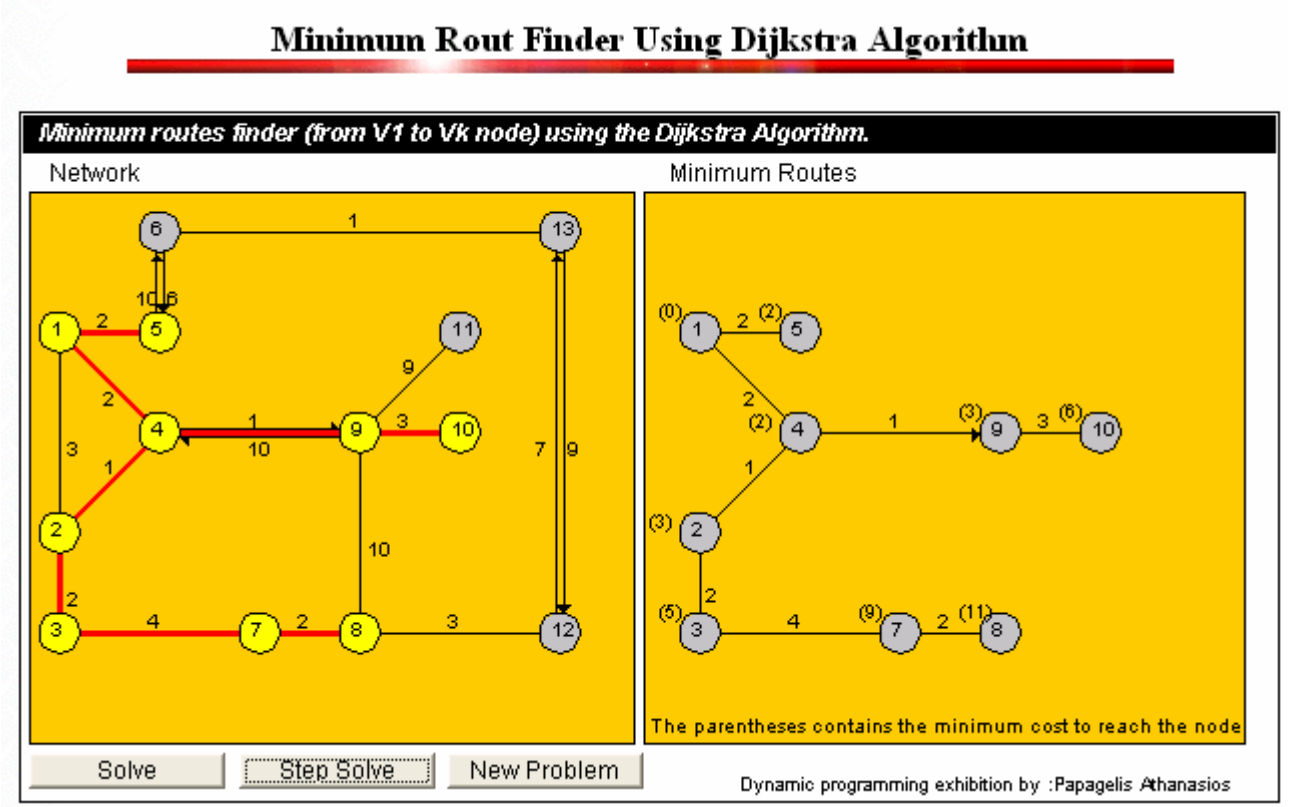
Jeder andere Weg von a nach v müsste irgendwo die Grenze zwischen U und V-U überschreiten, z.B. bei u_1,v_1. Dann gilt aber

$$\begin{aligned} & m_a(u) + d(u,v) \\ \leq\ & m_a(u_1) + d(u_1,v_1) \\ \leq\ & m_a(u_1) + d(u_1,v_1) + \kappa \end{aligned}$$

Eine schöne Animation findet sich bei

http://students.ceid.upatras.gr/~papagel/minDijk.htm

Animations-Applet



http://students.ceid.upatras.gr/~papagel/minDijk.htm

- Links der Original-Graph.
- Anfangsknoten: 1
- Das linke Bild zeigt den Graphen mit den bisher expandierten Wegen
- In Frage kommen als nächstes die Kanten:
  - **5 → 6**
  - **9 → 11**
  - **8 → 12**

- Rechts die Menge U mit der Angabe des kürzesten Abstands m(k) zu 1. Es gilt
  - $m_1(5) + d(5,6) = 2 + 10 = 12$
  - $m_1(9) + d(9,11) = 3 + 9 = 12$
  - $m_1(8) + d(8,12) = 11 + 3 = 14$
- Als nächstes kann (5,6) oder (9,11) expandiert werden