

Graphen

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S

Letzte Änderung: 9. März 2023

Vorlesungsfolien

ac  ALGORITHMS AND
COMPLEXITY GROUP



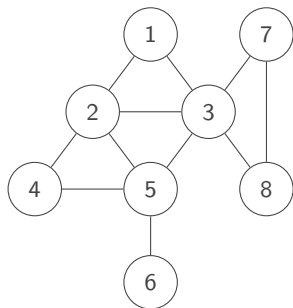
Informatics

Grundlegende Definitionen und Anwendungen

Ungerichtete Graphen

Ungerichteter Graph: $G = (V, E)$

- V = Menge der Knoten (*vertices, nodes*).
- E = Menge der Kanten zwischen Paaren von Knoten (*edges*).
- Notation für Kante zwischen Knoten a und b : (a, b) bzw. (b, a) .
- Alternativ wird auch $a - b$ bzw. $b - a$ verwendet.
- Parameter für Größen: $n = |V|$, $m = |E|$



$$V = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

$$E = \{1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6, 7-8\}$$

$$n = 8$$

$$m = 11$$

Ungerichtete Graphen: Weitere Definitionen

Adjazent, inzident, Nachbarschaft: Sei $e = (u, v)$ eine Kante in E .

- u und v sind adjazent, d.h. u ist Nachbar von v und v ist Nachbar von u .
- v (bzw. u) und e sind inzident.
- $(u, v) = (v, u)$.

Knotengrad (*degree*): $deg(v)$ bezeichnet den Knotengrad des Knotens v .

- $deg(v)$ entspricht der Anzahl der zu v inzidenten Kanten.
- Es gilt: $\sum_{v \in V} deg(v) = 2 \cdot |E|$ (Handshaking-Lemma).

Ungerichtete Graphen: Weitere Definitionen

Grundlegende Definitionen:

- Mehrfachkante: Mehrere Kanten zwischen zwei Knoten.
- Schleife: Eine Kante, die einen Knoten mit sich selbst verbindet.

Schlichter Graph: Ein ungerichteter Graph ohne Mehrfachkanten und ohne Schleifen.

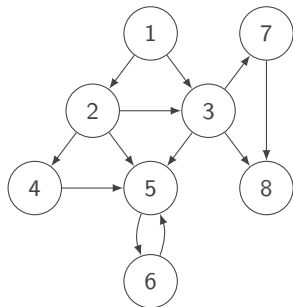
Hinweise:

- In dieser Vorlesung werden, wenn nicht anders verlautbart, schlichte Graphen betrachtet.
- Bei bestimmten Problemstellungen werden gewichtete Graphen verwendet, bei denen Knoten und/oder Kanten eine reelle Zahl zugeordnet bekommen.

Gerichtete Graphen

Gerichteter Graph (Digraph): $G = (V, E)$

- V = Menge der Knoten (*vertices, nodes*).
- E = Menge der gerichtete Kanten (*arcs*) zwischen Paaren von Knoten.
- Notation für Kante von a zu b : (a, b) bzw. $a \rightarrow b$
- $(a, b) \neq (b, a)$



$$V = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

$$E = \{1 \rightarrow 2, 1 \rightarrow 3, 2 \rightarrow 3, 2 \rightarrow 4, 2 \rightarrow 5, 3 \rightarrow 5, 3 \rightarrow 7, 3 \rightarrow 8, 4 \rightarrow 5, 5 \rightarrow 6, 6 \rightarrow 5, 7 \rightarrow 8\}$$

$$n = 8$$

$$m = 12$$

Hinweis: Kanten in entgegengesetzter Richtung sind auch in schlichten Digraphen erlaubt.

Gerichtete Graphen: Weitere Definitionen

Eingangsknotengrad: $deg^-(v)$ ist die Anzahl der eingehenden inzidenten Kanten.

Ausgangsknotengrad: $deg^+(v)$ ist die Anzahl der ausgehenden inzidenten Kanten.

Es gilt: $deg(v) = deg^+(v) + deg^-(v)$.

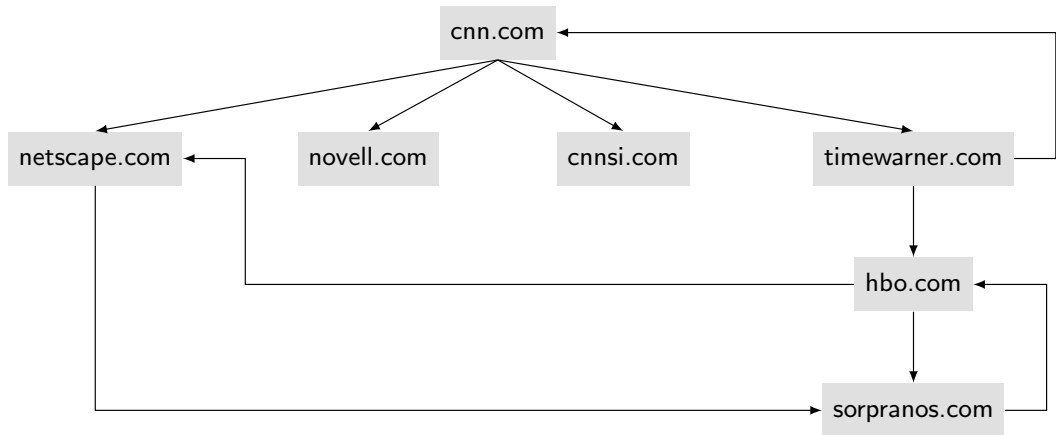
Einige Anwendungen von Graphen

<i>Graph</i>	<i>Knoten</i>	<i>Kanten</i>
Verkehr	Kreuzungen	Straßen
Netzwerke	Computer	Glasfaserkabel
World Wide Web	Webseiten	Hyperlinks
Sozialer Bereich	Personen	Beziehungen
Nahrungsnetz	Spezies	Räuber-Beute-Beziehung
Software	Funktionen	Funktionsaufrufe
Scheduling	Aufgaben	Ablaufeinschränkungen
elektronische Schaltungen	Gatter	Leitungen

World Wide Web

Web Graph:

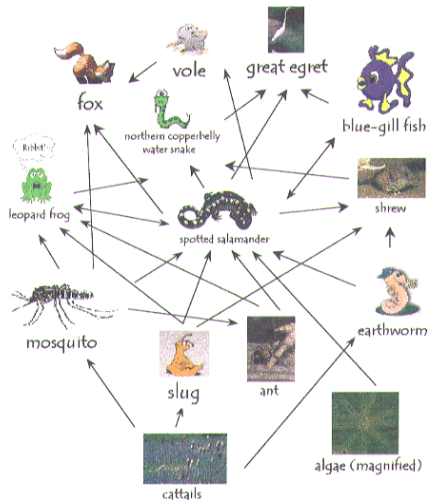
- Knoten: Webseiten.
- Kante: Hyperlink von einer Seite zur anderen.



Ökologisches Nahrungsnetz

Nahrungsnetz als Graph: Knoten = Spezies, Kante = von der Beute zum Raubtier.

Example:
This  →  means that the salamander eats the earthworm.



Königsberger Brückenproblem [Euler 1736]

128

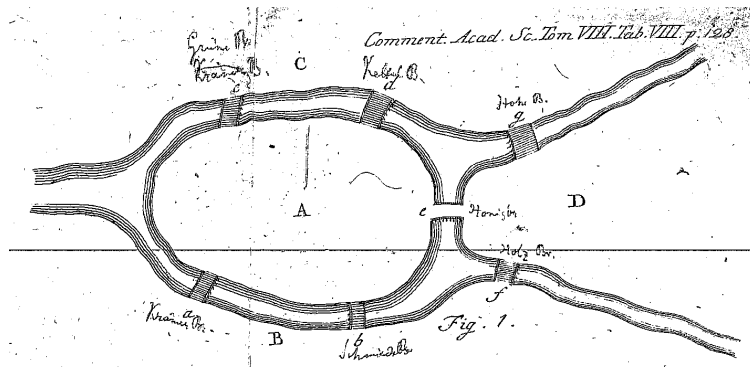
SOLVTIO PROBLEMATIS

SOLVTIO PROBLEMATIS AD GEOMETRIAM SITVS

PERTINENTIS.

AVCTORE

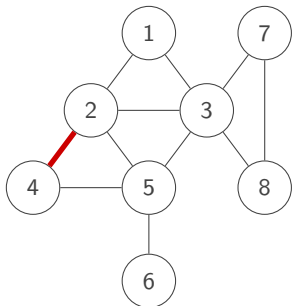
Leonb. Eulero.



Repräsentation von Graphen: Adjazenzmatrix

Adjazenzmatrix: n -mal- n Matrix mit $A_{uv} = 1$ wenn (u, v) eine Kante ist.

- Knoten: $1, 2, \dots, n$.
- Zwei Einträge für jede ungerichtete Kante.
- Für gewichtete Graphen: Reelle Matrix statt Boolesche Matrix.
- Platzbedarf in $\Theta(n^2)$.
- Überprüfen, ob (u, v) eine Kante ist, hat Laufzeit $\Theta(1)$.
- Aufzählen aller Kanten hat eine Laufzeit von $\Theta(n^2)$.

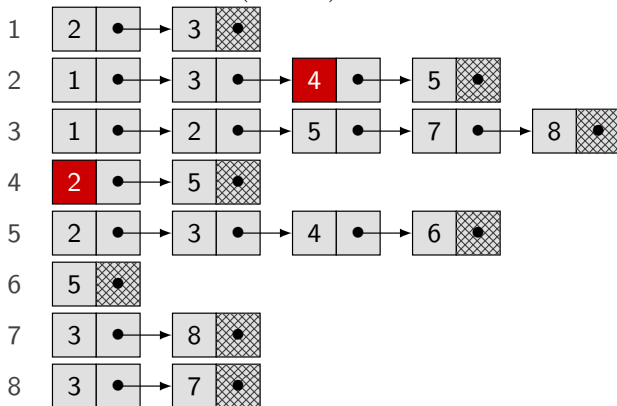
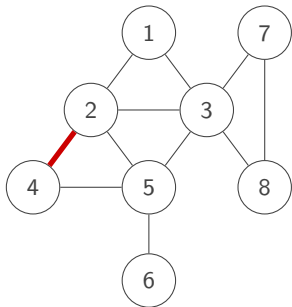


	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Repräsentation von Graphen: Adjazenzlisten

Adjazenzlisten: Array von Listen. Index ist die Knotennummer.

- Knoten: $1, 2, \dots, n$.
- Zwei Einträge für jede Kante.
- Für gewichtete Graphen: Speichere Gewicht in Liste.
- Platzbedarf in $\Theta(m + n)$.
- Überprüfen, ob (u, v) eine Kante ist, hat eine Laufzeit von $O(\deg(u))$.
- Aufzählen aller Kanten hat eine Laufzeit von $\Theta(m + n)$.



Adjazenzmatrix oder Adjazenzlisten

Kantenanzahl:

- Ein Graph kann bis zu $m = \frac{n(n-1)}{2} = \binom{n}{2} = \Theta(n^2)$ viele Kanten enthalten.
- Graphen sind **dicht (dense)** falls $m = \Theta(n^2)$.
- Graphen sind **licht (sparse)** falls $m = O(n)$.
- Für dichte Graphen sind beide Darstellungsformen (Adjazenzmatrix oder Adjazenzlisten) vergleichbar.

Praxis:

- Graphen, die sich aus Anwendungen ergeben, enthalten aber oft erheblich weniger Kanten.
- Typischerweise gilt dann $m = O(n)$.
- In diesem Fall ist die Darstellung mittels Adjazenzlisten günstiger.

Hinweis: Wenn wir sagen, dass ein Algorithmus auf Graphen in **Linearzeit** läuft, gehen wir von einer Darstellung mit Adjazenzlisten aus und betrachten eine Laufzeit von $O(n + m)$.

Ungerichtete Graphen: Kanten ausgeben

Adjazenzmatrix: Adjazenzmatrix M gegeben, n Knoten nummeriert von 0 bis $n - 1$

```
for  $u \leftarrow 0$  bis  $n - 2$ 
  for  $v \leftarrow u + 1$  bis  $n - 1$ 
    if  $M[u, v] = 1$ 
      Gib Kante  $(u, v)$  aus
```

Adjazenzliste: n Knoten nummeriert von 0 bis $n - 1$, jeder Knoten besitzt Liste der adjazenten Knoten

```
for  $u \leftarrow 0$  bis  $n - 1$ 
  foreach Kante  $(u, v)$  inzident zu  $u$ 
    if  $u < v$ 
      Gib Kante  $(u, v)$  aus
```


Gerichtete Graphen: Kanten ausgeben

Adjazenzmatrix: Adjazenzmatrix M gegeben, n Knoten nummeriert von 0 bis $n - 1$

```
for  $u \leftarrow 0$  bis  $n - 1$ 
  for  $v \leftarrow 0$  bis  $n - 1$ 
    if  $M[u, v] = 1$ 
      Gib Kante  $(u, v)$  aus
```

Adjazenzliste: n Knoten nummeriert von 0 bis $n - 1$, jeder Knoten besitzt Liste der adjazenten Knoten

```
for  $u \leftarrow 0$  bis  $n - 1$ 
  foreach Kante  $(u, v)$  inzident zu  $u$ 
    Gib Kante  $(u, v)$  aus
```

Kantenzüge und Pfade

Definition: Ein **Kantenzug** (eng: non-simple path) in einem ungerichteten Graphen $G = (V, E)$ ist eine Folge von Knoten $v_1, v_2, \dots, v_{k-1}, v_k$, $k \geq 1$, mit der Eigenschaft, dass jedes aufeinanderfolgende Paar v_i, v_{i+1} durch eine Kante in E verbunden ist. Die **Länge** des Kantenzugs ist $k - 1$.

Definition: Ein **Pfad** oder **Weg** (eng: simple path) in einem ungerichteten Graphen $G = (V, E)$ ist ein Kantenzug v_1, v_2, \dots, v_k bei dem sich kein Knoten wiederholt, also bei dem $v_i \neq v_j$ für alle $1 \leq i, j \leq k$ gilt.

Hinweis: Wir sagen auch: Der Pfad geht von v_1 nach v_k und wir bezeichnen den Pfad als v_1 - v_k -Pfad.

Achtung: Die Begriffe Pfad, Weg und Kantenzug werden in der Literatur nicht einheitlich verwendet.

Zusammenhang und Distanz

Definition: Knoten u ist von Knoten v in einem Graph G **erreichbar**, falls G einen u - v -Pfad enthält.

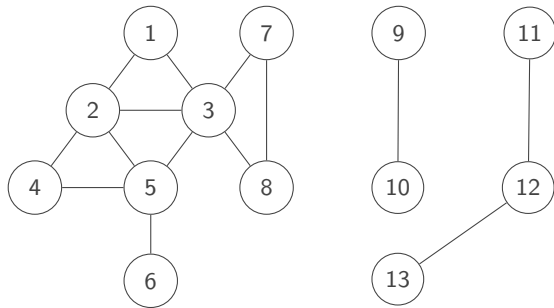
Definition: Ein ungerichteter Graph ist **zusammenhängend**, wenn jedes Paar von Knoten u und v von einander erreichbar ist.

Definition: Die **Distanz** zwischen Knoten u und v in einem ungerichteten Graphen ist die Länge eines kürzesten u - v -Pfades.

Hinweis: Falls u von v nicht erreichbar ist, nehmen wir die Distanz als ∞ an.

Zusammenhang: Beispiel

Nicht zusammenhängender Graph:

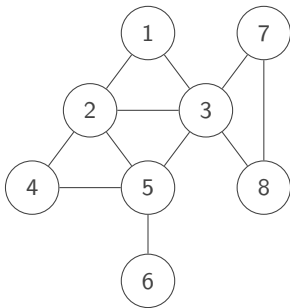


Nicht zusammenhängend: Es gibt zum Beispiel keinen Pfad vom Knoten 1 zu Knoten 10.

Beispiel für Zusammenhang: Die Knoten 1 bis 8 und ihre inzidenten Kanten bilden einen zusammenhängenden Graphen.

Kreis

Definition: Ein **Kreis** (eng: simple cycle) ist ein Kantenzug v_1, v_2, \dots, v_k in dem $v_1 = v_k$, $k \geq 4$, und die ersten $k - 1$ Knoten alle unterschiedlich sind. Die Länge des Kreises ist $k - 1$.



Beispiel für Kreis: $C = 1, 2, 4, 5, 3, 1$

Pfade und Kreise in gerichteten Graphen

Pfad: Ein **Kantenzug** in einem gerichteten Graphen $G = (V, E)$ ist eine Folge von Knoten $v_1, v_2, \dots, v_{k-1}, v_k$, $k \geq 1$, mit der Eigenschaft, dass jedes aufeinanderfolgende Paar v_i, v_{i+1} durch eine gerichtete Kante (v_i, v_{i+1}) in E verbunden ist. Ein **Pfad** ist ein Kantenzug bei dem alle Knoten unterschiedlich sind.

Hierbei gilt:

- Der Pfad geht von einem Startknoten u zu einem Endknoten v (u - v -Pfad). Die Umkehrung muss aber nicht gelten.
- v kann von u aus erreicht werden, falls ein u - v -Pfad existiert.
- Kürzeste u - v -Kantenzüge sind Pfade.

Kreis: Ein **gerichteter Kreis** ist ein Kantenzug $v_1, v_2, \dots, v_{k-1}, v_k$ in dem $v_1 = v_k$, $k \geq 3$, und die ersten $k - 1$ Knoten alle unterschiedlich sind.

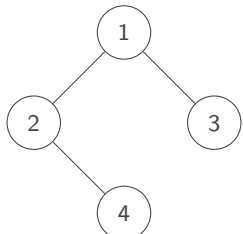
Pfade und Kreise als Graphen

Falls ein Graph G aus nur einem Pfad oder nur einem Kreis besteht, so nennen wir den ganzen Graphen einen Pfad/Kreis. Formal sagen wir:

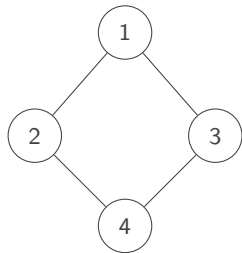
Pfad: Ein Graph G ist ein **Pfad**, falls es eine Aufzählung v_1, v_2, \dots, v_k der Knoten von G gibt, so dass es in G genau dann eine Kante zwischen zwei Knoten v_i und v_j gibt, falls $j = i + 1$.

Kreis: Ein Graph G ist ein **Kreis**, falls es eine Aufzählung v_1, v_2, \dots, v_k der Knoten von G gibt, so dass es in G genau dann eine Kante zwischen zwei Knoten v_i und v_j gibt, falls entweder $j = i + 1$ oder $i = 1$ und $j = k$ gilt.

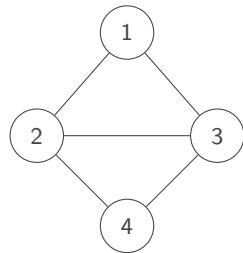
Pfade und Kreise als Graphen: Beispiele



Pfad



Kreis



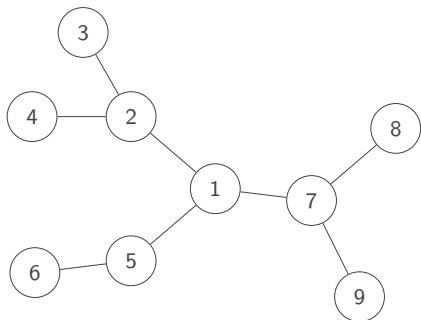
Weder Kreis noch Pfad

Bäume

Definition: Ein ungerichteter Graph ist ein **Baum**, wenn er zusammenhängend ist und keinen Kreis enthält.

Theorem: Sei G ein ungerichteter Graph mit n Knoten. Jeweils zwei der nachfolgenden Aussagen implizieren die dritte Aussage:

- G ist zusammenhängend.
- G enthält keinen Kreis.
- G hat $n - 1$ Kanten.



Bäume

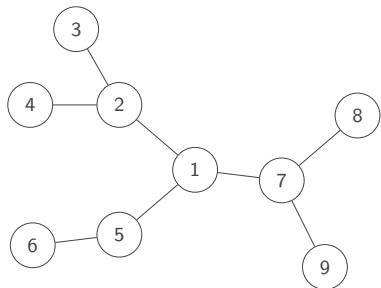
Theorem: Sei G ein ungerichteter Graph. G ist ein Baum genau dann wenn es für jedes Paar von Knoten u und v genau eine Pfad von u nach v gibt.

Beweis: G ist zusammenhängenden genau dann wenn es für jedes Paar von Knoten u und v mindestens einen Pfad von u nach v gibt. G enthält keinen Kreis, genau dann wenn es für jedes Paar von Knoten u und v maximal einen Pfad von u nach v gibt.

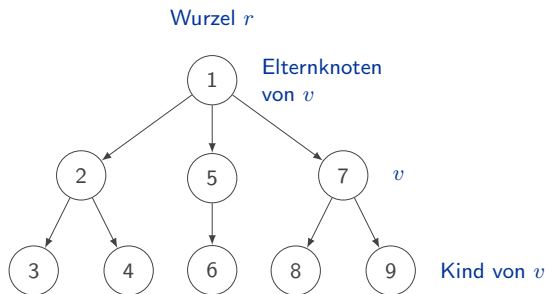
Wurzelbaum (*rooted tree, arborescence*)

Wurzelbaum: Gegeben sei ein Baum T . Wähle einen Wurzelknoten r und gib jeder Kante eine Richtung von r weg.

Bedeutung: Modelliert hierarchische Strukturen.



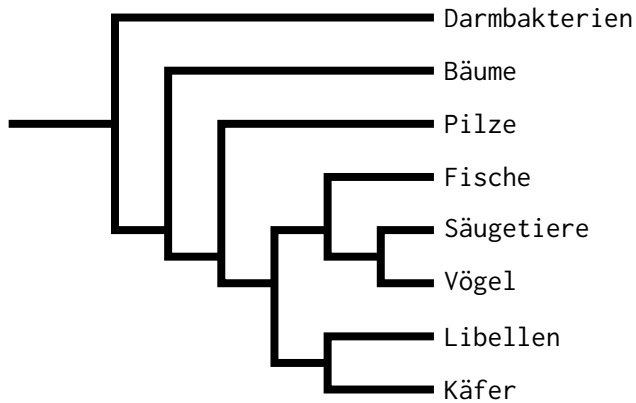
ein Baum



Ein entsprechender Wurzelbaum mit Wurzelknoten 1

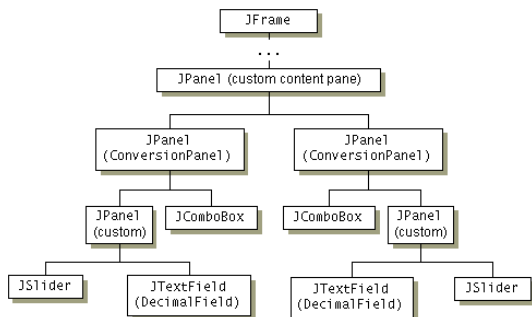
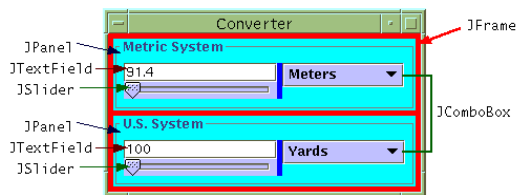
Phylogenetischer Baum

Phylogenetischer Baum: Beschreibt die evolutionären Beziehungen zwischen verschiedenen Arten.



GUI-Hierarchien

GUI-Hierarchien: Beschreiben die Organisation von GUI-Komponenten.



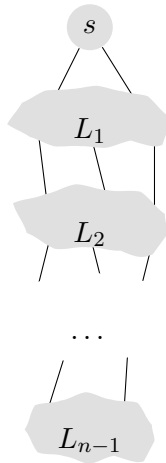
Durchmusterung von Graphen (*Graph Traversal*)

Breitensuche (*Breadth First Search, BFS*)

BFS Ansatz: Untersuche alle Knoten von einem Startknoten s ausgehend in alle möglichen Richtungen, wobei die Knoten Ebene für Ebene abgearbeitet werden.

BFS Algorithmus:

- $L_0 = \{s\}$.
- $L_1 =$ alle Nachbarn von L_0 .
- $L_2 =$ alle Knoten, die nicht zu L_0 oder zu L_1 gehören und die über eine Kante mit einem Knoten in L_1 verbunden sind.
- $L_{i+1} =$ alle Knoten, die nicht zu einer vorherigen Ebene gehören und die über eine Kante mit einem Knoten in L_i verbunden sind.



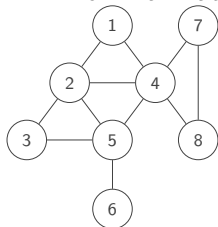
Anwendung der Breitensuche

s-t Zusammenhangsproblem: Existiert zwischen zwei gegebenen Knoten s und t ein Pfad?

s-t kürzester Pfad: Wie viele Kanten hat ein kürzester Pfad zwischen s und t (= Distanz zwischen s und t)?

Anwendungen:

- Facebook.
- Labyrinth durchschreiten.
- Kevin-Bacon-Zahl.
- Die kleinste Anzahl an Hops (kürzester Pfad) zwischen zwei Knoten in einem Kommunikationsnetzwerk.



Breitensuche: Theorem

Theorem: Für jede Ebene $i = 0, 1, \dots$ gilt, dass L_i alle Knoten mit Distanz i von s beinhaltet.

Beweis: Angenommen, sei $v_0, v_1, v_2, \dots, v_n$ ein kürzester Pfad zwischen v_0 und v_n .

- v_0 liegt in L_0 .
- v_1 liegt in L_1 , da v_1 ein Nachbar von v_0 ist.
- v_2 liegt in L_2 , da v_2 ein Nachbar von v_1 ist und kein Nachbar von v_0 sein kann, da es ansonsten einen kürzeren Pfad zwischen v_0 und v_n geben würde.
- Für alle weiteren Knoten gilt die gleiche Argumentation, d.h. v_n liegt schließlich in L_n . \square

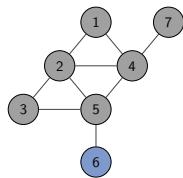
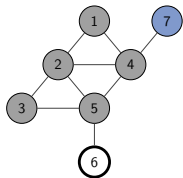
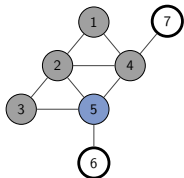
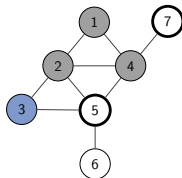
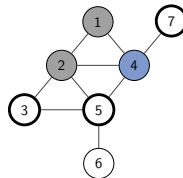
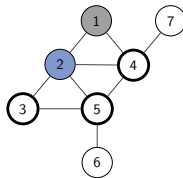
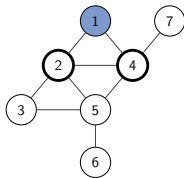
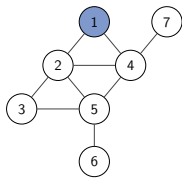
Breitensuche: Implementierung mit einer Queue

Implementierung: Array Discovered, Queue Q , Graph $G = (V, E)$, Startknoten s .

```
BFS( $G, s$ ):  
  Discovered[ $s$ ]  $\leftarrow$  true  
  Discovered[ $v$ ]  $\leftarrow$  false für alle anderen Knoten  $v \in V$   
   $Q \leftarrow \{s\}$   
  while  $Q$  ist nicht leer  
    Entferne ersten Knoten  $u$  aus  $Q$   
    Führe Operation auf  $u$  aus (z.B. Ausgabe)  
    foreach Kante  $(u, v)$  inzident zu  $u$   
      if !Discovered[ $v$ ]  
        Discovered[ $v$ ]  $\leftarrow$  true  
        Füge  $v$  zu  $Q$  hinzu
```

Breitensuche: Beispiel

Möglicher Ablauf: Startknoten = 1, bearbeitete Knoten sind grau, aktiver Knoten ist blau, alle anderen Knoten sind weiß, Knoten in Queue sind mit dicken Rahmen gekennzeichnet.



Breitensuche: Analyse

Theorem: BFS hat eine Laufzeit von $O(m + n)$.

Laufzeit: Für die Laufzeitabschätzung müssen wir drei Teile betrachten:

- Initialisierung vor der while-Schleife
- while-Schleife
- foreach-Schleife

Breitensuche: Analyse

Initialisierung vor der while-Schleife:

- Jeder Knoten wird genau einmal betrachtet
- Pro Knoten können die Anweisungen in konstanter Zeit ausgeführt werden.
- Daher benötigt die Initialisierung $O(n)$ Zeit.

while-Schleife:

- Jeder Knoten u wird höchstens einmal in Q gegeben, denn nachdem er das erste mal in Q gegeben wird, wird ja $\text{Discovered}[u]=\text{true}$ gesetzt.
- Daher wird die while-Schleife für jeden Knoten höchstens einmal durchlaufen.

Breitensuche: Analyse

foreach-Schleife:

- Sei u der gerade aktuelle Knoten bevor die foreach-Schleife ausgeführt wird.
- Dann werden in der foreach-Schleife alle Knoten v in der Adjazenzliste von u betrachtet.
- Das sind genau $\text{deg}(u)$ viele. Daher wird die Schleife $\text{deg}(u)$ mal durchlaufen. Die einzelnen Anweisungen in der Schleife benötigen konstante Zeit.

Gesamt:

- Insgesamt beträgt die Laufzeit also $O(n + \sum_{u \in V} \text{deg}(u))$.
- Da $\sum_{u \in V} \text{deg}(u) = 2m$, liegt die Laufzeit in $O(n + m)$.

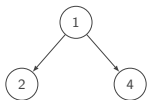
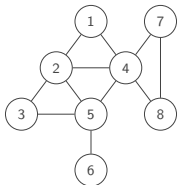
BFS-Baum

BFS-Baum: Breitensuche erzeugt einen Baum (BFS-Baum), dessen Wurzel ein Startknoten s ist und der alle von s erreichbaren Knoten beinhaltet.

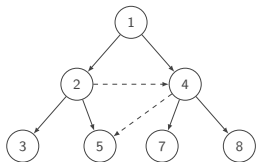
Aufbau: Man startet bei s . Wird nun ein Knoten v in der Ebene L_j gefunden, ist er zu mindestens einem Knoten u der Ebene L_{j-1} benachbart. Der Knoten u von dem aus v gefunden wurde wird ausgewählt und zum Elternknoten von v im BFS-Baum gemacht.

BFS-Baum: Eigenschaft

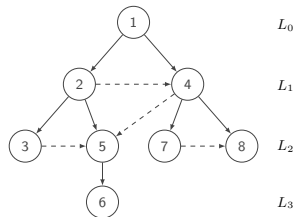
Eigenschaft: Sei T ein BFS-Baum von $G = (V, E)$ und sei (x, y) eine Kante von G . Dann können sich die Ebenen von x und y höchstens um 1 unterscheiden.



(a)



(b)



(c)

Breitensuche: Ermitteln der Ebenen

Anwendung von BFS: Ermitteln der Ebene jedes einzelnen Knotens.

Implementierung: Array Level, Queue Q , Graph $G = (V, E)$, Startknoten s .

```
BFS( $G, s$ ):  
Level[ $s$ ]  $\leftarrow 0$   
Level[ $v$ ]  $\leftarrow -1$  für alle anderen Knoten  $v \in V$   
 $Q \leftarrow s$   
while  $Q$  ist nicht leer  
    Entferne ersten Knoten  $u$  aus  $Q$   
    foreach Kante  $(u, v)$  inzident zu  $u$   
        if Level[ $v$ ] == -1  
            Level[ $v$ ]  $\leftarrow$  Level[ $u$ ] + 1  
            Füge  $v$  zu  $Q$  hinzu
```

Tiefensuche (*Depth First Search, DFS*)

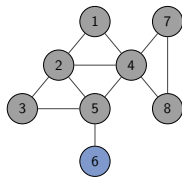
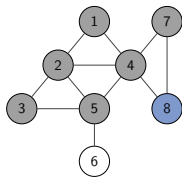
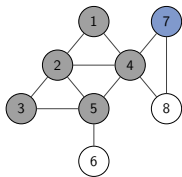
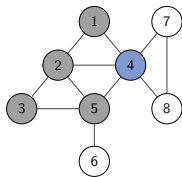
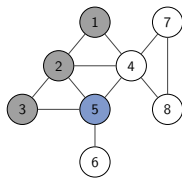
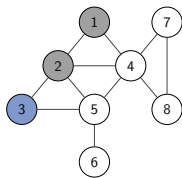
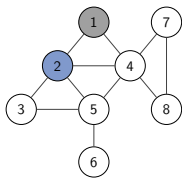
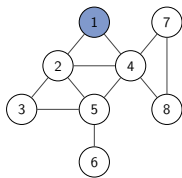
DFS Ansatz: Von einem besuchten Knoten u wird zuerst immer zu einem weiteren noch nicht besuchten Nachbarknoten gegangen (DFS-Aufruf), bevor die weiteren Nachbarknoten von u besucht werden.

DFS Algorithmus: Startknoten s , globales Array Discovered, Graph $G = (V, E)$.

```
DFS( $G, s$ ):  
  Discovered[ $v$ ]  $\leftarrow$  false für alle Knoten  $v \in V$   
  DFS1( $G, s$ )  
  
DFS1( $G, u$ ):  
  Discovered[ $u$ ]  $\leftarrow$  true  
  Führe Operation auf  $u$  aus (z.B. Ausgabe)  
  foreach Kante  $(u, v)$  inzident zu  $u$   
    if !Discovered[ $v$ ]  
      DFS1( $G, v$ )
```

Tiefensuche: Beispiel

Möglicher Ablauf: Startknoten = 1, bearbeitete Knoten sind grau unterlegt, aktiver Knoten ist blau, alle anderen Knoten sind weiß.



Tiefensuche: Analyse

Theorem: DFS hat eine Laufzeit von $O(m + n)$.

Laufzeit: Für Laufzeitabschätzung betrachten wir:

- Initialisierung
- foreach-Schleife

Initialisierung:

- Initialisierung vor dem Aufruf von DFS1 in $O(n)$ Zeit.
- DFS1(G, u) wird für jeden Knoten u höchstens einmal aufgerufen.

Tiefensuche: Analyse

foreach-Schleife in $\text{DFS1}(G, u)$:

- Es werden alle Knoten v in der Adjazenzliste von u betrachtet. Das sind genau $\text{deg}(u)$ viele.
- Daher wird die Schleife $\text{deg}(u)$ mal durchlaufen.
- Die einzelnen Anweisungen in der Schleife benötigen konstante Zeit (außer dem rekursiven Aufruf $\text{DFS1}(G, v)$, aber dessen Laufzeit wird ja in der Analyse für den Knoten v berücksichtigt).

Gesamt:

- Insgesamt beträgt die Laufzeit also $O(n + \sum_{u \in V} \text{deg}(u))$.
- Da $\sum_{u \in V} \text{deg}(u) = 2m$, erhalten wir eine Laufzeit von $O(n + m)$.

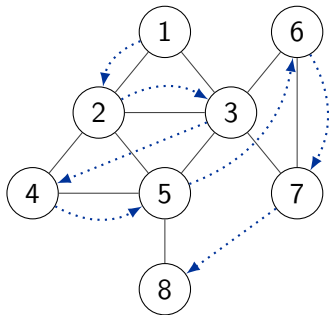
Tiefensuche: Durchmusterung

Durchmusterung: Durchmusterung bei DFS unterscheidet sich von der bei BFS.

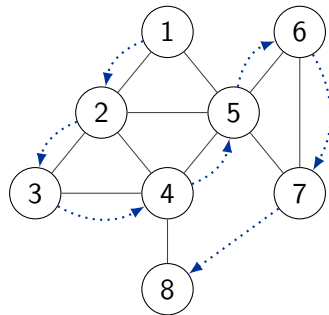
- Es wird zunächst versucht, möglichst weit vom Startknoten weg zu kommen.
- Gibt es in der Nachbarschaft keine möglichen Knoten, dann wird durch den rekursiven Aufstieg bis zu einer möglichen Verzweigung zurückgegangen (Backtracking).

Beispiel

Vergleich: Tiefensuche und Breitensuche im Vergleich.



Breitensuche:



Tiefensuche:

Zusammenhangskomponente

Zusammenhang (Wiederholung): Ein ungerichteter Graph ist **zusammenhängend**, wenn für jedes Paar von Knoten u und v ein Pfad zwischen u und v existiert.

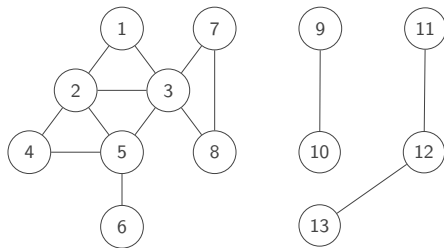
Nicht zusammenhängend: Gibt es zwischen einem Paar von Knoten keinen Pfad, dann ist der Graph nicht zusammenhängend.

Teilgraph: Ein Graph $G_1 = (V_1, E_1)$ heißt Teilgraph von $G_2 = (V_2, E_2)$, wenn seine Knotenmenge V_1 Teilmenge von V_2 und seine Kantenmenge E_1 Teilmenge von E_2 ist, also $V_1 \subseteq V_2$ und $E_1 \subseteq E_2$ gilt.

Zusammenhangskomponente: Einen maximalen zusammenhängenden Teilgraphen eines beliebigen Graphen nennt man Zusammenhangskomponente. Ein nicht zusammenhängender Graph zerfällt in seine Zusammenhangskomponenten.

Zusammenhangskomponente

Beispiel: Ein nicht zusammenhängender Graph mit 3 Zusammenhangskomponenten.



Zusammenhangskomponente

Zusammenhangskomponente: Finde alle Knoten, die von s aus erreicht werden können.

Lösung:

- Rufe $\text{DFS}(G,s)$ oder $\text{BFS}(G,s)$ auf.
- Ein Knoten u ist von s genau dann erreichbar, wenn $\text{Discovered}[u]=\text{true}$ ist.

Zusammenhangskomponenten zählen

DFSNUM Algorithmus: Startknoten s , globales Array Discovered, Graph $G = (V, E)$.

```
DFSNUM( $G$ ):  
  Discovered[ $v$ ]  $\leftarrow$  false für alle Knoten  $v \in V$   
   $i \leftarrow 0$   
  foreach Knoten  $v \in V$   
    if Discovered[ $v$ ] = false  
       $i \leftarrow i + 1$   
      DFS1( $G, v$ )  
  return  $i$ 
```

Zusammenhangskomponenten zählen

Laufzeit: Die Laufzeit liegt in $O(n + m)$.

Analyse:

- Sei $G = (V, E)$ der gegebene Graph und $G_1 = (V_1, E_1), \dots, G_r = (V_r, E_r)$ seine Zusammenhangskomponenten. Sei $|V| = n$ und $|E| = m$, sowie $|V_i| = n_i$ und $|E_i| = m_i$, für $1 \leq i \leq r$.
- Klarerweise gilt $n = n_1 + \dots + n_r$ und $m = m_1 + \dots + m_r$.
- Für jede einzelne Zusammenhangskomponente G_i ($1 \leq i \leq r$) führt der Algorithmus eine Tiefensuche aus. Dies hat eine Laufzeit von $O(n_i + m_i)$.
- Die Initialisierung benötigt $O(n)$ Zeit.
- Insgesamt erhalten wir eine Laufzeit von $O(n + \sum_{i=1}^r (n_i + m_i)) = O(2n + m) = O(n + m)$.

Zusammenhang in gerichteten Graphen

Suche in gerichteten Graphen

Gerichtete Erreichbarkeit: Gegeben sei ein Knoten s , finde alle Knoten, die von s aus erreicht werden können.

Gerichteter kürzester s - t Pfad: Gegeben seien zwei Knoten s und t , ermittle einen kürzesten Pfad von s nach t .

Suche in gerichteten Graphen: BFS und DFS können auch auf gerichtete Graphen angewendet werden.

Beispiel Webcrawler: Starte von einer Webseite s . Finde alle Webseiten, die von s aus direkt oder indirekt verlinkt sind.

Starker Zusammenhang

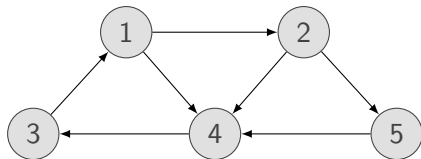
Definition: Knoten u und v in einem gerichteten Graphen sind **gegenseitig erreichbar**, wenn es einen Pfad von u zu v und einen Pfad von v zu u gibt.

Definition: Ein gerichteter Graph ist **stark zusammenhängend**, wenn jedes Paar von Knoten gegenseitig erreichbar ist.

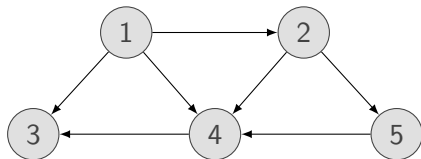
Hinweis: Ein gerichteter Graph heißt **schwach zusammenhängend**, falls der zugehörige ungerichtete Graph (also der Graph, der entsteht, wenn man jede gerichtete Kante durch eine ungerichtete Kante ersetzt) zusammenhängend ist.

Starker Zusammenhang: Beispiel

Stark zusammenhängend:



Nicht stark zusammenhängend (aber schwach zusammenhängend): Knoten 1 kann von keinem anderen Knoten erreicht werden, vom Knoten 3 führt kein Pfad weg.



Starker Zusammenhang

Lemma: Sei s ein beliebiger Knoten in einem gerichteten Graphen G . G ist stark zusammenhängend dann und nur dann, wenn jeder Knoten von s aus und s von jedem Knoten aus erreicht werden kann.

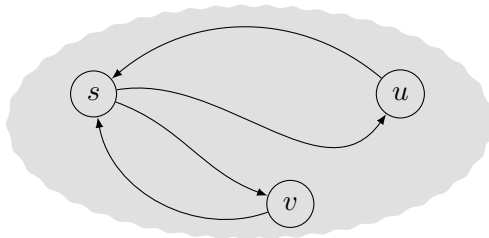
Beweis: \Rightarrow Folgt aus der Definition.

Beweis: \Leftarrow Pfad von u zu v : verbinde u - s Pfad mit s - v Pfad.

u : verbinde v - s Pfad mit s - u Pfad. \square

Pfad von v zu

\square auch ok, wenn Pfade überlappen



Starker Zusammenhang: Algorithmus

Theorem: Laufzeit für die Überprüfung, ob G stark zusammenhängend ist, liegt in $O(m + n)$.

Beweis:

- Wähle einen beliebigen Knoten s .
- Führe BFS mit Startknoten s in G aus.
- Führe BFS mit Startknoten s in G^{rev} aus.
- Gib true zurück dann und nur dann, wenn alle Knoten in beiden BFS-Ausführungen erreicht werden können.
- Korrektheit folgt unmittelbar aus dem vorherigen Lemma. \square
 - *umgekehrte Orientierung von jeder Kante in G*

DAGs und Topologische Sortierung

Gerichteter azyklischer Graph (*Directed Acyclic Graph, DAG*)

Definition: Ein **DAG** ist ein gerichteter Graph, der keine gerichteten Kreise enthält.

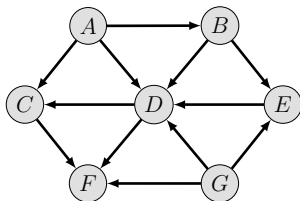
Beispiel: Knoten: Aufgaben, Kanten: Reihenfolgebeschränkungen
Kante (u, v) bedeutet, Aufgabe u muss vor Aufgabe v erledigt werden.

Definition: Wir nennen einen Knoten v ohne eingehende Kanten in einem gerichteten Graphen (i.e., $\text{deg}^-(v) = 0$) **Quelle**.

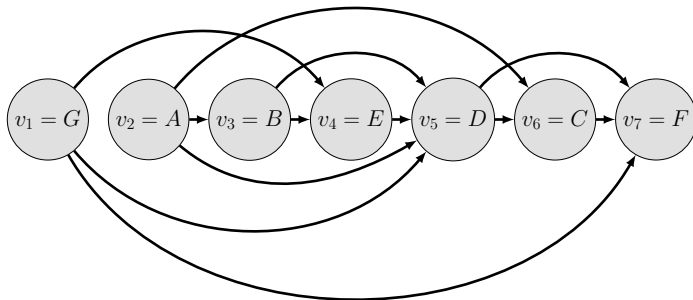
Definition: Eine **topologische Sortierung** eines gerichteten Graphen $G = (V, E)$ ist eine lineare Ordnung seiner Knoten, bezeichnet mit v_1, v_2, \dots, v_n , sodass für jede Kante (v_i, v_j) gilt, dass $i < j$.

Topologische Sortierung: Beispiel

Ein DAG:



Eine topologische Sortierung:



Reihenfolgebeschränkung

Reihenfolgebeschränkung: Kante (u, v) bedeutet, dass Aufgabe u vor v bearbeitet werden muss.

Anwendungen:

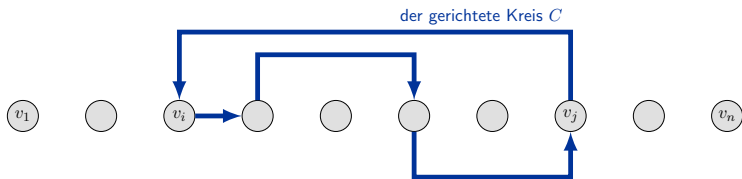
- Voraussetzungen bei Kursen: Kurs u muss vor Kurs v absolviert werden.
- Übersetzung: Modul u muss vor Modul v übersetzt werden.
- Pipeline von Prozessen: Ausgabe von Prozess u wird benötigt, um die Eingabe von v zu bestimmen.

Gerichteter azyklischer Graph

Lemma: Wenn G eine topologische Sortierung hat, dann ist G ein DAG.

Beweis: (durch Widerspruch)

- Wir nehmen an, dass G eine topologische Sortierung v_1, \dots, v_n und auch einen gerichteten Kreis C besitzt.
- Sei v_i der Knoten mit dem kleinsten Index in C und sei v_j der Knoten direkt vor v_i in C ; daher gibt es die Kante (v_j, v_i) .
- Durch die Wahl von i gilt, dass $i < j$.
- Andererseits, da (v_j, v_i) eine Kante ist und v_1, \dots, v_n eine topologische Sortierung ist, müsste eigentlich $j < i$ sein. Widerspruch. \square



die angenommene topologische Sortierung: v_1, \dots, v_n

Gerichteter azyklischer Graph

Lemma: Wenn G eine topologische Sortierung hat, dann ist G ein DAG.

Frage: Hat jeder DAG eine topologische Sortierung?

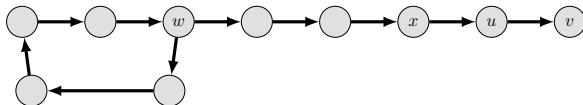
Frage: Wenn ja, wie berechnen wir diese?

Gerichteter azyklischer Graph

Lemma: Wenn G ein DAG ist, dann hat G eine Quelle.

Beweis: (durch Widerspruch)

- Wir nehmen an, G ist ein DAG ohne Quelle.
- Wähle einen beliebigen Knoten v und folge den Kanten von v aus rückwärts. Da v zumindest eine eingehende Kante (u, v) besitzt, können wir rückwärts zu u gelangen.
- Da u zumindest eine eingehende Kante (x, u) hat, können wir rückwärts zu x gelangen.
- Das wird so oft wiederholt, bis man einen Knoten w zweimal besucht.
- Sei C die Sequenz von Knoten die zwischen zwei Besuchen von w durchlaufen wurde. C ist ein Kreis. \square



Gerichteter azyklischer Graph

Lemma: G ist ein DAG genau dann wenn jeder Teilgraph von G eine Quelle hat.

Beweis:

- Angenommen G ist ein DAG, dann ist offensichtlich auch jeder Teilgraph von G ein DAG (Das Entfernen von Knoten kann keine Kreise produzieren). Deswegen hat jeder Teilgraph von G eine Quelle.
- Angenommen G ist kein DAG. Dann enthält G einen Kreis als Teilgraph. Ein Kreis hat keine Quelle.

Gerichteter azyklischer Graph - Erkennen eines DAG mittels wiederholtem Löschen von Kanten

```
while  $G$  hat mindestens einen Knoten
  if  $G$  hat eine Quelle
    Wähle eine Quelle  $v$  aus
    Gib  $v$  aus
    Lösche  $v$  und alle inzidenten Kanten aus  $G$ 
  else return  $G$  ist kein DAG
return  $G$  ist ein DAG
```

Hinweis:

- Ein Knoten kann im Lauf des Algorithmus zur Quelle werden.
- Falls G ein DAG ist, gibt dieser Algorithmus eine topologische Sortierung aus.

Gerichteter azyklischer Graph

Lemma: Wenn G ein DAG ist, dann hat G eine topologische Sortierung.

Beweis:

- Falls G ein DAG ist, können wir eine topologische Sortierung berechnen.
- Falls G kein DAG ist, enthält G eine Kreis v_1, \dots, v_n . In der Ordnung einer topologischen Sortierung müsste dann $v_1 < \dots < v_n < v_1$ gelten. Dann ist die Ordnung allerdings keine lineare Ordnung.

Topologische Sortierung

Algorithmus: Effiziente Implementierung des Löschalgorithmus: Löschen von Knoten wird mittels Hilfsarray count simuliert. Es wird zusätzlich eine anfangs leere Liste L verwendet.

```
foreach  $v \in V$ 
    count[v]  $\leftarrow$  0
foreach  $v \in V$ 
    foreach Kante  $(v, w) \in E$ 
        count[w]  $\leftarrow$  count[w]+1
foreach  $v \in V$ 
    if count[v] = 0
        Gib  $v$  zur Liste  $L$  am Anfang hinzu
while  $L$  ist nicht leer
    Sei  $v$  erstes Element in  $L$ , lösche  $v$  aus  $L$ 
    Gib  $v$  aus
    foreach Kante  $(v, w) \in E$ 
        count[w]  $\leftarrow$  count[w]-1
        if count[w] = 0
            Gib  $w$  zur Liste  $L$  am Anfang hinzu
```

Topologische Sortierung: Laufzeit

Theorem: Algorithmus findet eine topologische Sortierung in $O(n + m)$ Zeit.

Laufzeit: Dazu betrachten wir die folgenden Teile:

- Initialisierung
 - Erste foreach-Schleife für count.
 - Zwei verschachtelte foreach-Schleifen.
 - Dritte foreach-Schleife für Generierung der Liste.
- while-Schleife (mit foreach-Schleife).

Initialisierung:

- Die erste foreach-Schleife für die Initialisierung von count benötigt $O(n)$ Zeit.
- Bei den verschachtelten foreach-Schleifen wird die innere foreach-Schleife für jeden Knoten v genau $deg^+(v)$ mal ausgeführt. Daher benötigt man dafür $O(n + m)$ Zeit.
- Die Generierung der Liste L durch die dritte foreach-Schleife benötigt $O(n)$ Zeit.
- Daher benötigt die Initialisierung $O(n + m)$ Zeit.

Topologische Sortierung: Analyse

while-Schleife:

- Jeder Knoten v wird höchstens einmal aus L entnommen.
- Daher wird die while-Schleife für jeden Knoten höchstens einmal durchlaufen.

foreach-Schleife:

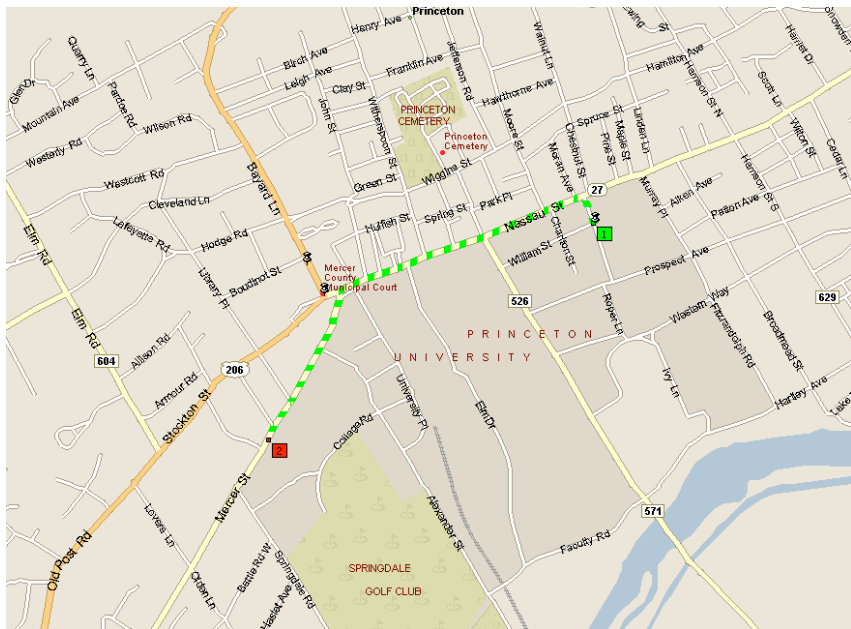
- Sei v der gerade aktuelle Knoten bevor die foreach-Schleife ausgeführt wird.
- Dann werden in der foreach-Schleife alle Knoten w in der Adjazenzliste von v betrachtet.
- Das sind genau $\text{deg}^+(v)$ viele. Daher wird die Schleife $\text{deg}^+(v)$ mal durchlaufen. Die einzelnen Anweisungen in der Schleife benötigen konstante Zeit.
- Jeder Knoten w wird höchstens einmal in L eingefügt.

Topologische Sortierung: Analyse

Gesamt:

- Initialisierung liegt in $O(n + m)$
- while-Schleife liegt in $O(n + m)$
- Daher liegt auch die gesamte Laufzeit in $O(n + m)$

Kürzeste Pfade in einem gewichteten Graphen



Kürzester Pfad vom Informatikinstitut in Princeton zu Einsteins Haus.

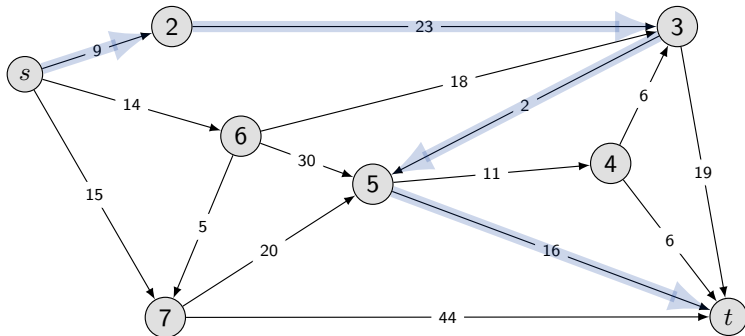
Kürzester Pfad (*Shortest Path Problem*)

Netzwerk für kürzesten Pfad:

- Gerichteter Graph $G = (V, E)$.
- Start s , Ziel t .
- Länge $\ell_e \geq 0$ ist die Länge der Kante e (Gewicht).

Kürzester Pfad: Finde **kürzesten** gerichteten Pfad von s nach t .

■ *Kürzester Pfad = Pfad mit den geringsten Kosten, wobei die Kosten eines Pfades die Summe der Gewichte seiner Kanten sind.*



Kosten des Pfades
 $s-2-3-5-t$
 $= 9 + 23 + 2 + 16$
 $= 50.$

Numerische Mathematik 1, 269—271 (1959)

A Note on Two Problems in Connexion with Graphs

By

E. W. DIJKSTRA

We consider n points (nodes), some or all pairs of which are connected by a branch; the length of each branch is given. We restrict ourselves to the case where at least one path exists between any two nodes. We now consider two problems.

Algorithmus von Dijkstra

Algorithmus von Dijkstra:

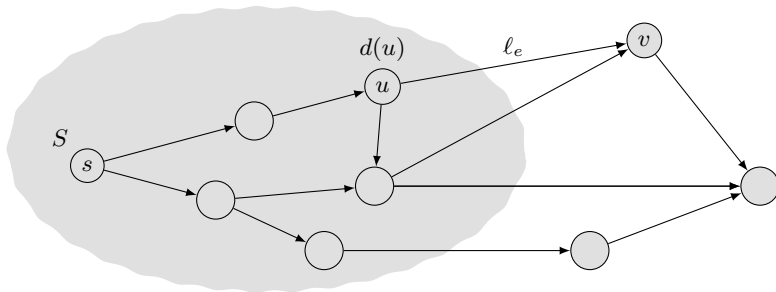
- Verwalte eine Menge S von **untersuchten Knoten**, für die wir die Kosten $d(u)$ eines kürzeste s - u -Pfades ermittelt haben.
- Initialisiere $S = \{s\}$, $d(s) = 0$.
- Wähle wiederholt einen nicht untersuchten Knoten v , für den der folgende Wert am kleinsten ist:

$$\min_{e=(u,v):u \in S} d(u) + \ell_e,$$

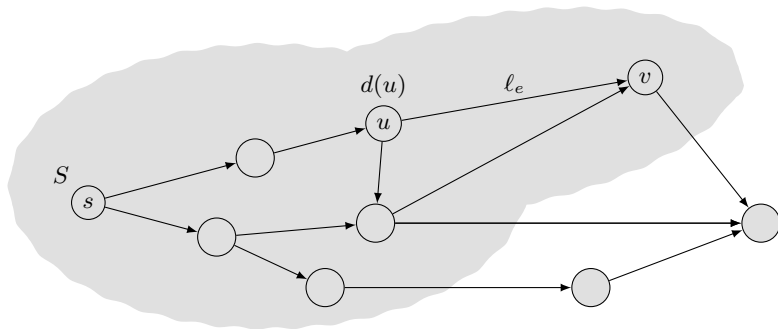
d.h. die Länge eines kürzesten Pfades zu einem u im untersuchten Teil des Graphen, gefolgt von einer einzigen Kante (u, v) .

- Füge v zu S hinzu und setze $d(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$.
- Extrahieren des Pfades entweder durch Merken des Vorgängerknotens oder mittels eines eigenen Algorithmus, der nach Dijkstra ausgeführt wird.

Algorithmus von Dijkstra: Menge S



Algorithmus von Dijkstra: Menge S



Dijkstra-Algorithmus: Implementierung

Implementierung: Wir werden zwei Implementierungen für S betrachten:

- Eine einfach verkettete Liste.
- Eine Vorrangwarteschlange (*priority queue*) von nicht untersuchten Knoten, geordnet nach den Kosten d .
Ein Eintrag in der Queue besteht aus dem Knotenindex und den dazugehörigen Kosten.

Dijkstra-Algorithmus

Algorithmus: Arrays Discovered und d , Graph $G = (V, E)$, Liste L , Startknoten s .

```
Dijkstra( $G, s$ ):  
Discovered[ $v$ ]  $\leftarrow$  false für alle Knoten  $v \in V$   
 $d[s] \leftarrow 0$   
 $d[v] \leftarrow \infty$  für alle anderen Knoten  $v \in V \setminus \{s\}$   
 $L \leftarrow V$   
while  $L$  ist nicht leer  
    wähle  $u \in L$  mit kleinstem Wert  $d[u]$   
    lösche  $u$  aus  $L$   
    Discovered[ $u$ ]  $\leftarrow$  true  
    foreach Kante  $e = (u, v) \in E$   
        if !Discovered[ $v$ ]  
             $d[v] \leftarrow \min(d[v], d[u] + \ell_e)$ 
```


Algorithmus von Dijkstra: Korrektheitsbeweis

Invariante: Für jeden Knoten $u \in S$, ist $d(u)$ die Länge eines kürzesten s - u Pfades.

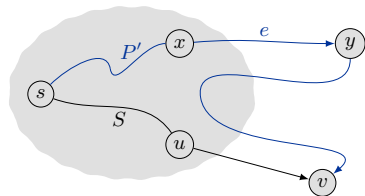
Beweis: (durch Induktion nach $|S|$)

Induktionsanfang: $|S| = 1$ ist trivial.

Induktionsbehauptung: Angenommen, wahr für $|S| = k \geq 1$.

- Sei v der nächste zu S hinzugefügte Knoten und sei (u, v) die gewählte Kante.
- Ein kürzester s - u Pfad plus (u, v) ist ein s - v Pfad der Länge $d(v)$.

- Wir betrachten einen beliebigen s - v Pfad P . Wir werden zeigen, dass er nicht kürzer als $d(v)$ ist.
- Sei $e = (x, y)$ die erste Kante in P die S verlässt und sei P' der Teilpfad zu x .
- P ist schon zu lange, wenn er S verlässt.



$$\text{Länge}(P) \geq \text{Länge}(P') + \ell_e \geq d(x) + \ell_e \geq d(y) \geq d(v)$$

- Nicht-negative Gewichte
- Induktionsbehauptung
- Definition von $d(y)$
- Dijkstra-Algorithmus wählt v anstatt y

Analyse: Dijkstra-Algorithmus mit Liste

Theorem: Der Dijkstra-Algorithmus, implementiert mit einer Liste, hat eine Worst-Case-Laufzeit von $O(n^2)$.

Laufzeiten:

- Initialisierung der Arrays benötigt $O(n)$ Zeit.
- Die while-Schleife wird n -mal ausgeführt und darin muss in jeder Iteration der Knoten u mit dem kleinsten Wert für $d[u]$ gefunden werden. Das liegt in $O(n^2)$ Zeit.
- Die foreach-Schleife wird insgesamt (über alle Iterationen der while-Schleife) höchstens m -mal ausgeführt. Für jeden Knoten werden seine ausgehenden Kanten nur einmal betrachtet und insgesamt gibt es nur m Kanten.
- Daher beträgt die Laufzeit $O(n + n^2 + m)$ und somit $O(n^2)$. \square

Wir werden sehen, dass der Dijkstra-Algorithmus mit einer Worst-Case-Laufzeit von $O((n + m) \log n)$ implementiert werden kann. Für lichte Graphen ist das effizienter als $O(n^2)$.

Priority Queue (Vorrangwarteschlange)

Priority Queue:

- Eine Priority Queue ist eine Datenstruktur, die eine Menge S von Elementen verwaltet.
- Jedes Element $v \in S$ hat einen dazugehörigen Wert i , der die Priorität von v beschreibt.
- Kleinere Werte repräsentieren höhere Prioritäten.

Operationen: Alle mit Laufzeit in $O(\log n)$.

- Einfügen eines Elements in die Menge S .
- Löschen eines Elements aus der Menge S .
- Finden eines Elements mit dem kleinsten Wert (höchster Priorität).

Frage: Wie erreicht man eine Laufzeit in $O(\log n)$?

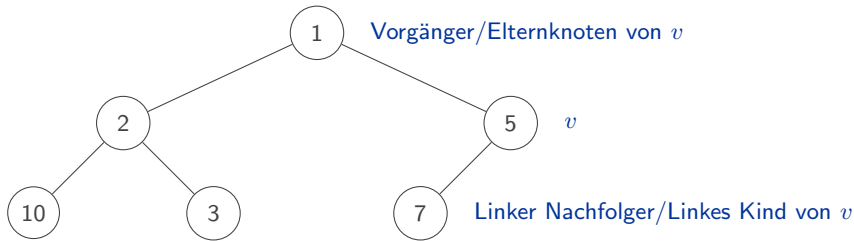
Antwort: Mit einer bestimmten Datenstruktur, dem Heap.

Heap

Heap: Ein Heap (Min-Heap) ist ein binärer Wurzelbaum, dessen Knoten mit \leq total geordnet sind, sodass gilt:

- Ist u ein linkes oder rechtes Kind von v , dann gilt $v \leq u$ (**Heap-Eigenschaft für Min-Heap**).
- Alle Ebenen von Knoten bis auf die letzte sind vollständig aufgefüllt.
- Die letzte Ebene des Baumes muss linksbündig aufgefüllt werden.

Beispiel:



Repräsentation eines Heaps

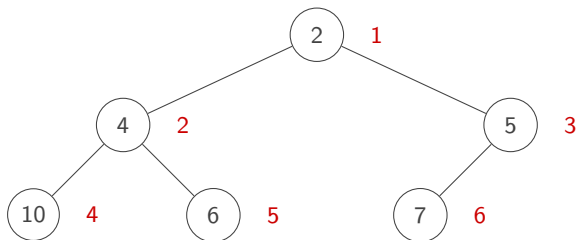
Effiziente Repräsentation: Knoten des Baums ebenenweise in einem Array speichern.

Effiziente Berechnung:

- Die beiden Nachfolgerknoten eines Knotens an der Position k befinden sich an den Positionen $2k$ und $2k + 1$. Sein Elternknoten befindet sich an der Position $\lfloor \frac{k}{2} \rfloor$.
- Damit obige Rechnung immer funktioniert, wird das Array ab Index 1 belegt.
- Würde man bei Index 0 anfangen, dann würden sich die Berechnungen folgendermaßen ändern: Nachfolger links auf $2k + 1$, Nachfolger rechts auf $2k + 2$, Elternknoten auf $\lfloor \frac{k-1}{2} \rfloor$.

Beispiel für Heap-Repräsentation

Heap:



Array: 6 Einträge, erster Platz unbelegt (mit 0 initialisiert).

Index	0	1	2	3	4	5	6
Wert	0	2	4	5	10	6	7

Heapify-up

Einfügen eines neuen Elements: Bei einem Heap mit n Elementen wird das neue Element an Position $n + 1$ eingefügt. Wir gehen dabei davon aus, dass noch genügend Plätze im Array frei sind.

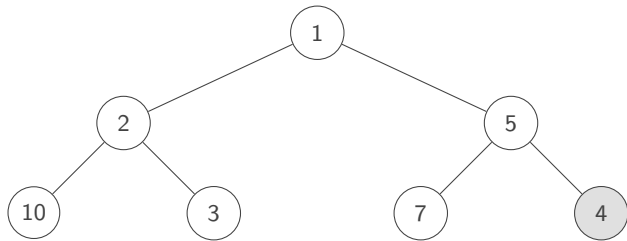
Heap-Bedingung: Die Heap-Bedingung kann durch das neue Element verletzt werden.

Reparieren: Durch Operation Heapify-up (für Heap-Array H an Position i) in $O(\log n)$ Zeit. Aufruf nach dem Einfügen des neuen Elements: $\text{Heapify-up}(H, n + 1)$.

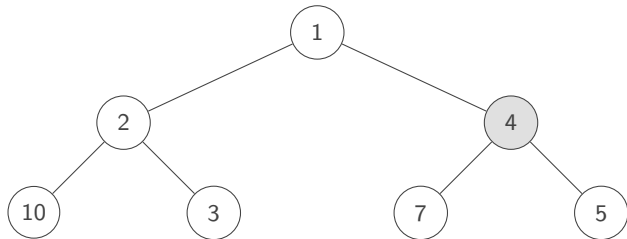
```
Heapify-up( $H, i$ ):  
if  $i > 1$   
     $j \leftarrow \lfloor i/2 \rfloor$   
    if  $H[i] < H[j]$   
        Vertausche die Array-Einträge  $H[i]$  und  $H[j]$   
        Heapify-up( $H, j$ )
```

Beispiel für Heapify-up

Einfügen von 4:



Verschieben von 4:



Heapify-down

Löschen eines Elements: Element wird an Stelle i gelöscht. Das Element an Stelle n (bei n Elementen) wird an die freie Stelle verschoben.

Heap-Bedingung: Die Heap-Bedingung kann durch das neue Element an der Stelle i verletzt werden.

Reparieren:

- Eingefügtes Element ist zu groß: Benutze Heapify-down, um das Element auf eine untere Ebene zu bringen.
- Eingefügtes Element ist zu klein: Benutze Heapify-up (wie beim Einfügen) von der Stelle i aus.

Hinweis: Beim Heap wird typischerweise die Wurzel entfernt und daher wird dann nur Heapify-down benutzt.

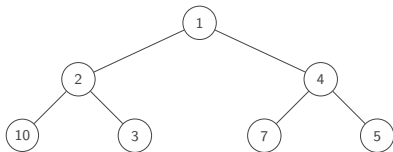
Laufzeit für Löschen: Für Heap-Array H an Position i in $O(\log n)$ Zeit.

Heapify-down

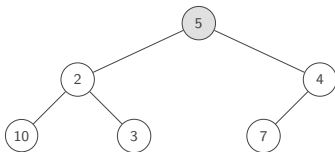
```
Heapify-down(H, i):  
  n ← length(H)-1  
  if 2 · i > n  
    return  
  elseif 2 · i < n  
    left ← 2 · i, right ← 2 · i + 1  
    j ← Index des kleineren Wertes von H[left] und H[right]  
  else  
    j ← 2 · i  
  if H[j] < H[i]  
    Vertausche die Arrayeinträge H[i] und H[j]  
    Heapify-down(H, j)
```

Beispiel für Heapify-down

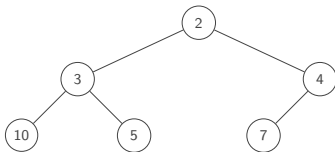
Ursprünglicher Heap:



Löschen von 1, verschieben von 5:



Heapify-down (zwei Mal)



Operationen auf Heap

Operationen auf Heap:

- $\text{Insert}(H, v)$: Element v in den Heap H einfügen. Hat der Heap n Elemente, dann liegt die Laufzeit in $O(\log n)$.
- $\text{FindMin}(H)$: Findet das Minimum im Heap H . Laufzeit ist konstant (da Wurzel).
- $\text{Delete}(H, i)$: Löscht das Element im Heap H an der Stelle i . Für einen Heap mit n Elementen liegt die Laufzeit in $O(\log n)$.
- $\text{ExtractMin}(H)$: Kombination von FindMin und Delete und daher in $O(\log n)$.

Erstellen eines Heaps

Erstellen: Das Erstellen eines Heaps aus einem Array A mit Größe n , das noch nicht die Heapeigenschaft erfüllt:

```
Init( $A, n$ ):  
for  $i = \lfloor n/2 \rfloor$  bis 1  
    Heapify-down( $A, i$ )
```

Erstellen eines Heaps: Analyse

Laufzeit: $O(n)$ ergibt sich aus folgender Berechnung:

- Einfachheitshalber nehmen wir an, der Binärbaum ist vollständig und hat n Knoten.
- Es folgt, dass $n = 2^{h+1} - 1$ wobei h die Höhe des Baumes ergibt.
- Wir lassen den Index j über die Ebenen E_j des Baumes laufen, wobei mit E_0 die Ebene mit den Blättern des Baumes bezeichnet und E_h die Ebene mit der Wurzel.
- Es folgt, dass Ebene E_j genau 2^{h-j} Knoten enthält und der Aufwand zum Einfügen eines Elements auf Ebene E_j proportional zu j ist.
- Insgesamt ergibt sich also ein Aufwand von $\sum_{j=0}^h j 2^{h-j}$, den wir folgendermaßen abschätzen:

$$\sum_{j=0}^h j 2^{h-j} = \sum_{j=0}^h j \frac{2^h}{2^j} = 2^h \sum_{j=0}^h \frac{j}{2^j} \leq 2^h 2 = 2^{h+1} \stackrel{\text{■}}{=} n + 1 = O(n)$$

- folgt aus $\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$. ■ da $n = 2^{h+1} - 1$.

Dijkstra-Algorithmus: Effizientere Variante

Algorithmus:

- Arrays Discovered und d , Graph $G = (V, E)$, Startknoten s .
- Verwende Vorrangwarteschlange Q , in der die Knoten v nach dem Wert $d[v]$ geordnet sind.

```
Dijkstra( $G, s$ ):  
Discovered[ $v$ ]  $\leftarrow$  false für alle Knoten  $v \in V$   
 $d[s] \leftarrow 0$   
 $d[v] \leftarrow \infty$  für alle anderen Knoten  $v \in V \setminus \{s\}$   
 $Q \leftarrow V$   
while  $Q$  ist nicht leer  
  wähle  $u \in Q$  mit kleinstem Wert  $d[u]$   
  lösche  $u$  aus  $Q$   
  Discovered[ $u$ ]  $\leftarrow$  true  
  foreach Kante  $e = (u, v) \in E$   
    if !Discovered[ $v$ ]  
      if  $d[v] > d[u] + l_e$   
        lösche  $v$  aus  $Q$   
         $d[v] \leftarrow d[u] + l_e$   
        füge  $v$  zu  $Q$  hinzu
```

Analyse: Dijkstra-Algorithmus mit Vorrangwarteschlange

Theorem: Der Dijkstra-Algorithmus, implementiert mit einer Vorrangwarteschlange, hat eine Worst-Case-Laufzeit von $O((n + m) \log n)$.

Laufzeiten:

- Initialisierung der Arrays benötigt $O(n)$ Zeit.
- Die while-Schleife wird n -mal ausgeführt und darin muss in jeder Iteration der Knoten u mit dem kleinsten Wert für $d[u]$ aus der Queue gelöscht werden ($O(\log n)$).
- Die foreach-Schleife liegt in $O(m \log n)$ Zeit. Für jeden Knoten werden seine ausgehenden Kanten nur einmal betrachtet und insgesamt gibt es nur m Kanten. Bei einer Neuberechnung muss aber die Queue reorganisiert werden (diese Operation liegt in $O(\log n)$).
- Daher beträgt die Laufzeit $O(n + n \log n + m \log n)$ und somit $O((n + m) \log n)$.
□

Dijkstra-Algorithmus: Abschließender Vergleich

Wir vergleichen die Laufzeit des Dijkstra-Algorithmus bei Verwendung von Listen, Vorrangwarteschlange als Heap und Vorrangwarteschlange als Fibonacci-Heap (diese verbesserte Datenstruktur haben wir nicht besprochen).

Tabelle Vergleich verschiedener Datenstrukturen für Dijkstra-Algorithmus.

Liste	Heap	FibHeap
$O(n^2)$	$O((n + m) \log n)$	$O(m + n \log n)$