Algorithmen und Datenstrukturen

Graphen

Prof. Justus Piater, Ph.D.

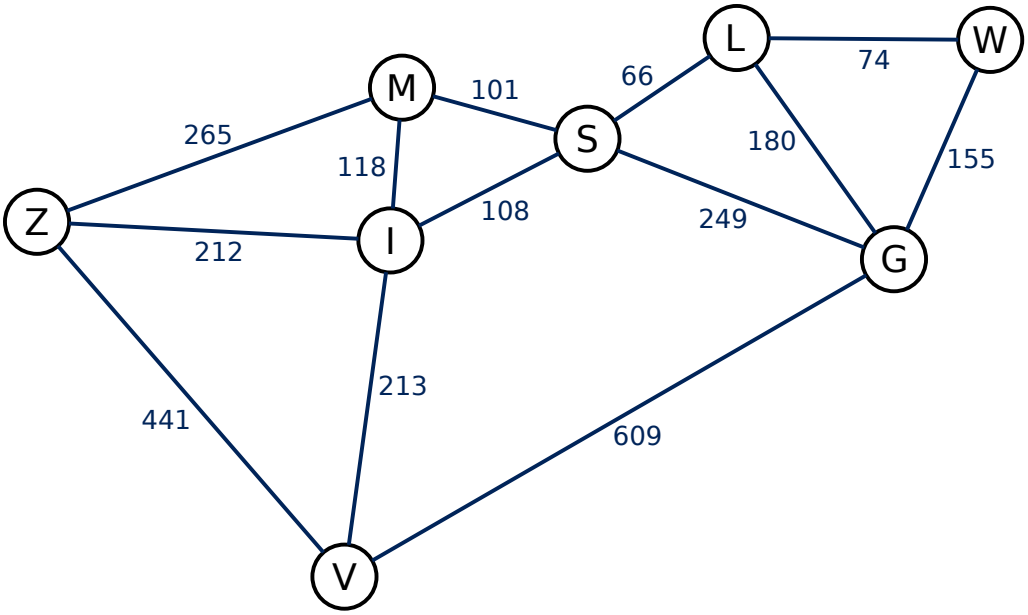
20. Juni 2021

Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch *Data Structures and Algorithms in Java* [Goodrich u. a. 2014].

Inhaltsverzeichnis

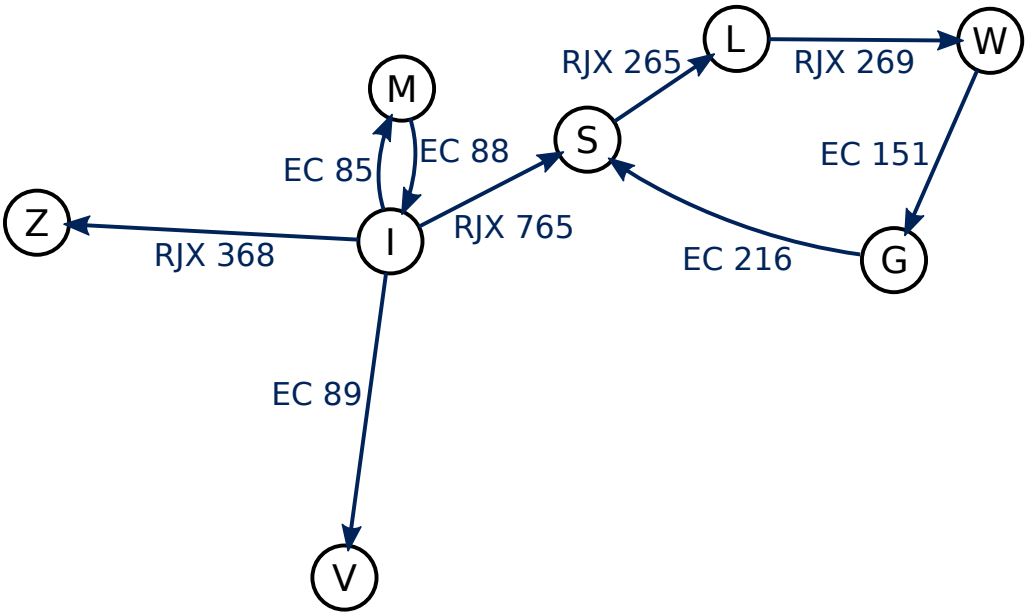
1 Konzepte und ADT

Beispiel: Fahrtzeiten in Minuten [Slide 1]

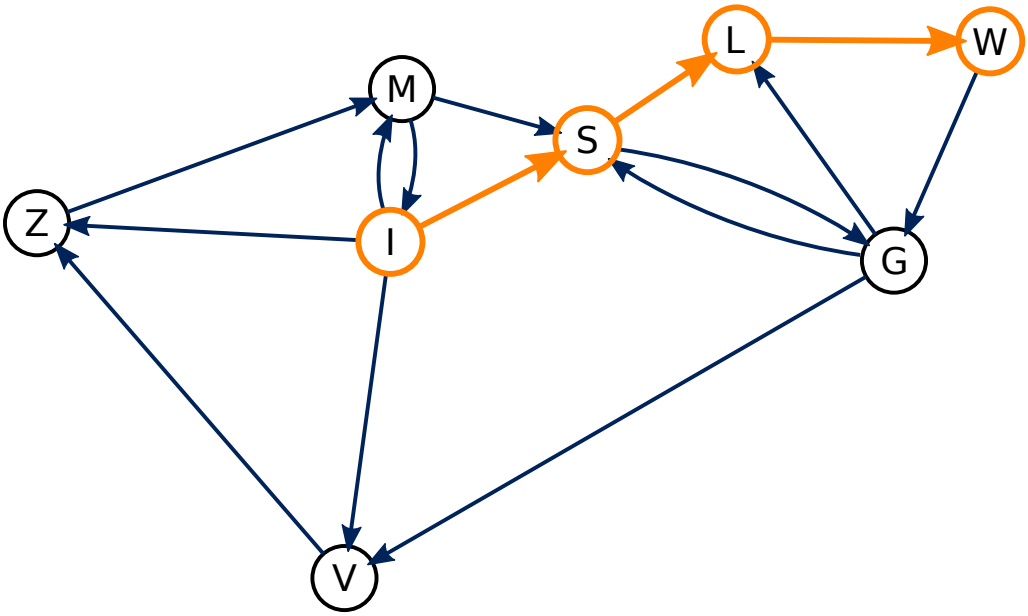


Die Knoten dieses Graphen repräsentieren Innsbruck, Salzburg, Linz, Wien, Graz, Verona, Zürich und München. (Die Anordnung ist maßstabsgetreu.)
Die Kanten repräsentieren Bahnverbindungen. (Die Fahrtzeiten sind überwiegend typisch (2021), aber es gibt deutlich schnellere Verbindungen zwischen Zürich und Verona, und zwischen Salzburg und Graz verzögert eine Baustelle die Fahrt um 10 Minuten.)

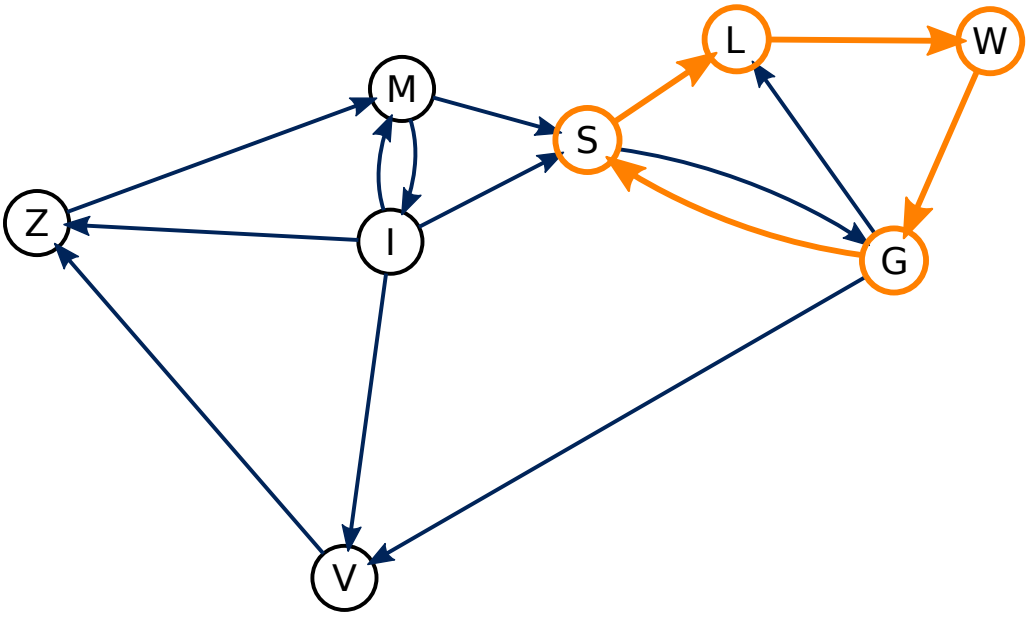
Beispiel: Bahnverbindungen [Slide 2]



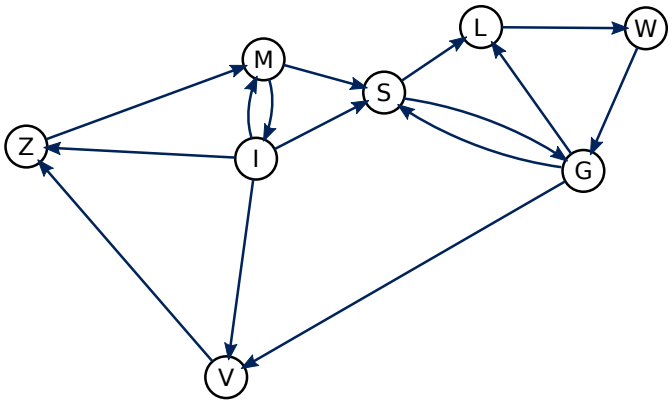
Gerichteter Pfad [Slide 3]



Gerichteter Zykel [Slide 4]



Quiz [Slide 5]



Wie viele (gerichtete) Zykel enthält dieser Graph?

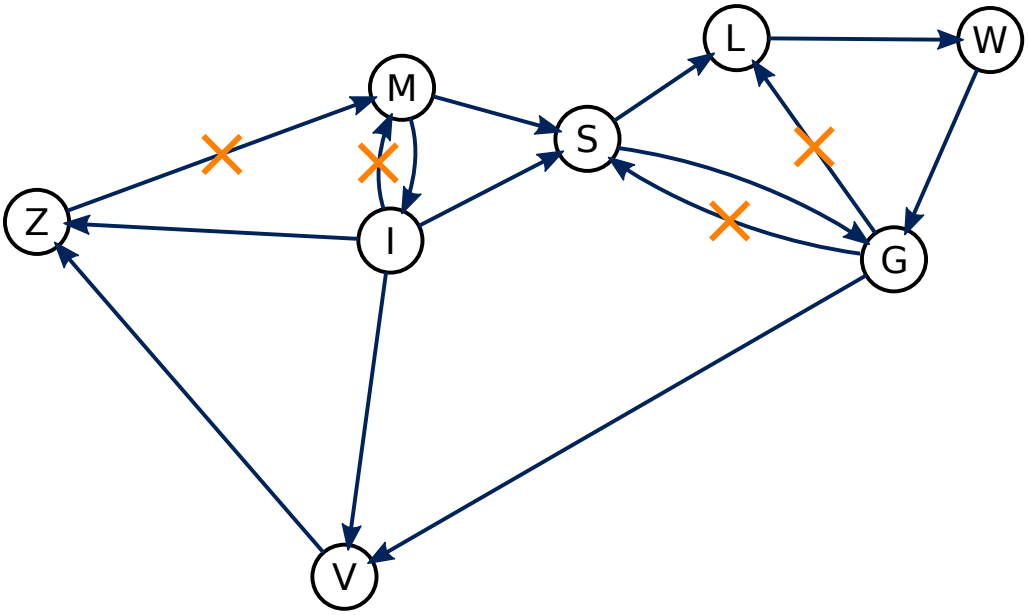
A: 4

B: 6

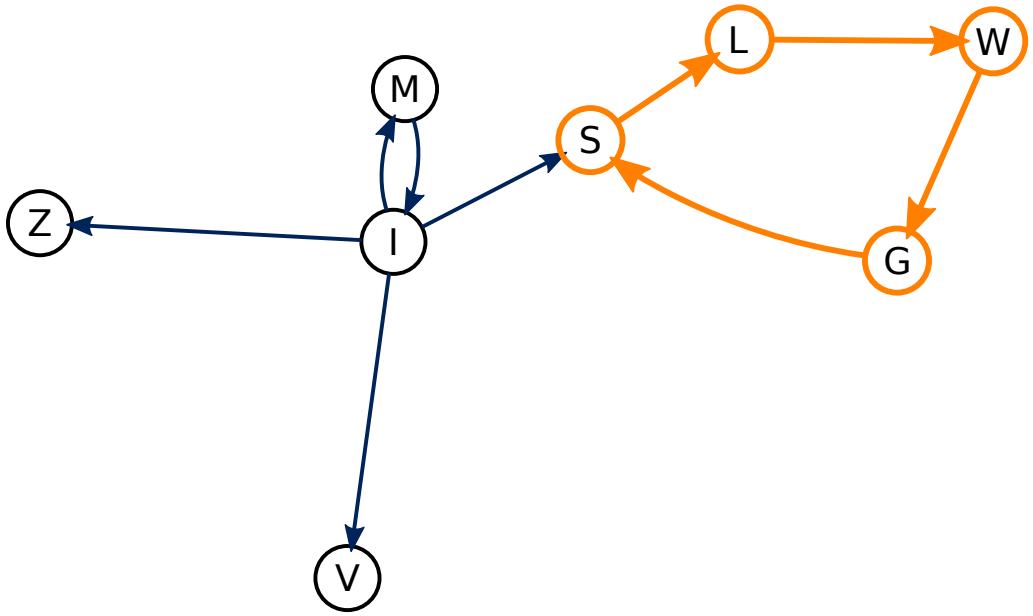
C: mehr als 8

D: weiß nicht

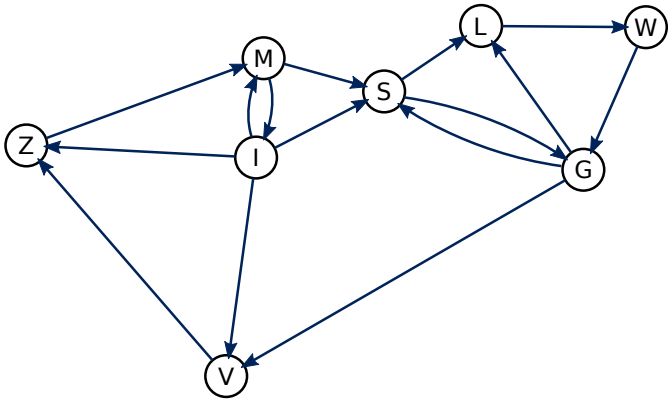
Gerichteter Graph ohne Zykel (*Directed Acyclic Graph, DAG*) [Slide 6]



Erreichbarkeit ausgehend von Salzburg [Slide 7]



Quiz [Slide 8]



Gibt es in diesem Graphen zwei Knoten A und B so dass B nicht von A aus erreichbar ist?

A: ja

B: nein

D: weiß nicht

Nein. Beweis: Es gibt einen Zykel, der alle Knoten enthält, also einen *Hamiltonkreis (Hamiltonian cycle)*.

Graphen: Definitionen und Terminologie [Slide 9]

$G = (V, E)$

- V ist die Menge der ***Knoten*** (oder ***Ecken***; engl. *nodes* bzw. *vertices*).
- E ist die Menge der (***ungerichteten*** oder ***gerichteten***) ***Kanten*** (engl. *edges*).
- Eine Kante $e = (u, v) \in E$ mit $u, v \in V$ ist entweder ***ungerichtet*** oder ***gerichtet*** (geordnetes Paar). Die Kante e ist zu den Knoten u und v ***inzident*** (und umgekehrt), und die Knoten u und v sind (dank der sie verbindenden Kante e) ***adjazent***.
- Ein Graph, der ausschließlich *(un)gerichtete* Kanten enthält, ist ein ***(un)gerichteter Graph***.
- Der ***Grad*** eines Knotens ist die Anzahl seiner inzidenten Kanten. Sind diese gerichtet, wird zwischen ***Eingangs-*** und ***Ausgangsgrad*** unterschieden.
- Ein ***Pfad*** ist eine Sequenz $((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k))$ von Knotenpaaren, die jeweils eine (ggf. entsprechend gerichtete) Kante repräsentieren, so dass der erste Knoten jedes Paares mit dem zweiten Knoten des vorhergehenden Paares identisch ist.

- In unserem ADT sind Knoten und Kanten *Positionen* und enthalten Anwendungsdaten.

Mehr zu Graphen nächstes Semester in der LV *Diskrete Strukturen*.

ADT: Graph [Slide 10]

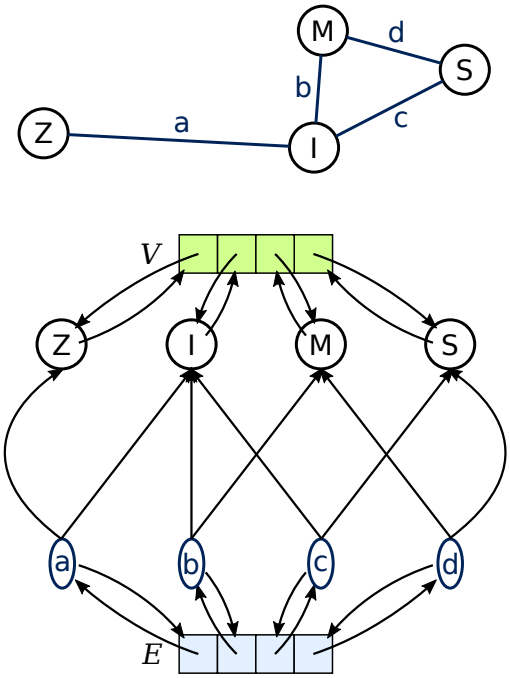
```
numVertices()      // returns the number of vertices
vertices()         // returns an iterator of the vertices
numEdges()         // returns the number of edges
edges()            // returns an iterator of the edges
getEdge(u,v)       // returns an edge from vertex u to vertex v, or null;
                   //   for an undirected graph, getEdge(u,v) = getEdge(v,u)
endVertices(e)     // returns an array of the two end vertices of edge e;
                   //   for a directed graph, the first edge is the origin
opposite(v, e)     // for edge e incident to vertex v,
                   //   returns the other end vertex of e
outDegree(v)       // returns the number of outgoing edges from vertex v
inDegree(v)        // returns the number of incoming edges to vertex v;
                   //   for an undirected graph, this equals outDegree(v)
outgoingEdges(v)   // returns an iterator of the outgoing edges from vertex v
incomingEdges(v)   // returns an iterator of the incoming edges to vertex v

insertVertex(x)    // creates and returns a new vertex storing element x
insertEdge(u, v, x) // creates and returns a new edge from vertex u to
                   //   vertex v storing element x;
                   //   error if there already exists an endge from u to v
removeVertex(v)    // removes vertex v and all its incident edges
removeEdge(e)      // remove edge e
```

[Goodrich u. a. 2014]

2 Datenstrukturen

Kantenliste [Slide 11]

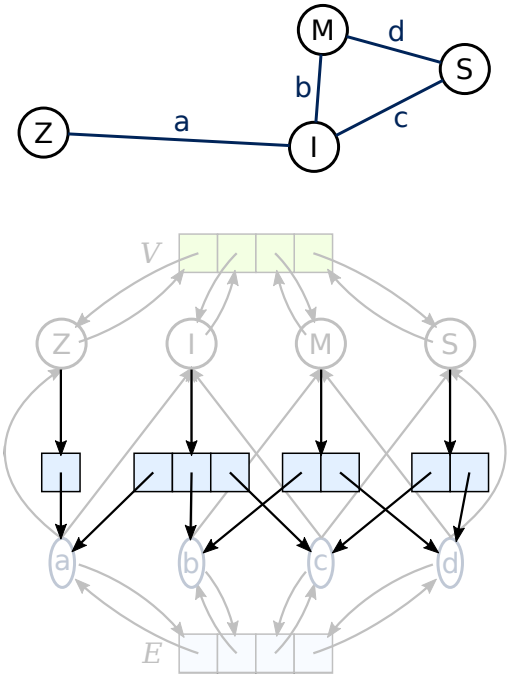


Knoten und Kanten liegen jeweils in einer positionsbasierten Liste:

- Ein Knotenobjekt besitzt Zeiger auf
  - sein Element,
  - seine Position in V.

  Knoten zeigen *nicht* auf ihre Kanten!
- Ein Kantenobjekt besitzt Zeiger auf
  - sein Element,
  - seine beiden Knotenobjekte,
  - seine Position in E.

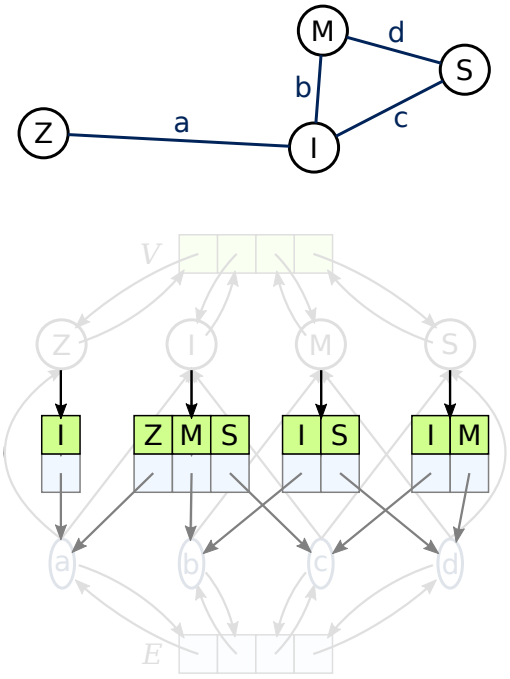
Adjazenzliste [Slide 12]



Wie die Kantenliste, plus:

- Ein Knotenobjekt besitzt einen Zeiger auf
  - eine Liste seiner inzidenten Kantenobjekte bzw. je eine Liste seiner eingehenden und ausgehenden Kantenobjekte.

Adjazenz-Zuordnungstabelle *(Adjacency Map)* [Slide 13]



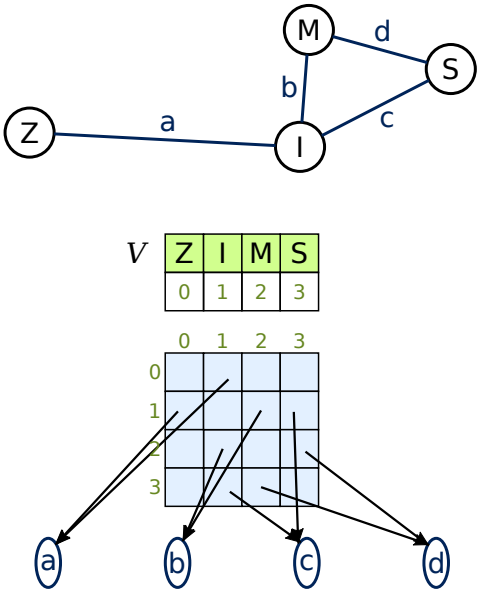
Wie die Adjazenzliste, außer:

- Statt der *Liste* der inzidenten Kantenobjekte wird eine *Zuordnungstabelle* verwendet, in der für jede Kante der benachbarte Knoten als Schlüssel dient.

Anmerkung

Aufgrund der Laufzeitverhalten ihrer Methoden ist die *Adjacency Map* eine ausgezeichnete Allzweck-Datenstruktur für Graphen.

Adjazenzmatrix [Slide 14]



Wie die Kantenliste, jedoch erweitert um eine Adjazenzmatrix.

Adjazenzmatrix hier im mathematischen Sinne, während sich dieser Begriff im Titel auf die Datenstruktur bezieht.

Anmerkung

Kann für *dichte* Graphen effizienter sein als die *Adjacency Map*.

Datenstrukturen: Laufzeitübersicht [Slide 15]

Methode	Kantenliste	Adj.liste	Adj.Zutab.	Adj.matrix
`numVertices()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`numEdges()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`vertices()`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`edges()`	$O(m)$	$O(m)$	$O(m)$	$O(m)$
`getEdge`(u, v)	$O(m)$	$O(\min\{d_u, d_v\})$	$O(1)$ erw.	$O(1)$
`out/inDegree`(v)	$\Theta(m)$	$O(1)$	$O(1)$	$\Theta(n)$
`outgoing/incomingEdges`(v)	$\Theta(m)$	$O(d_v)$	$O(d_v)$	$\Theta(n)$
`insertVertex`(x)	$O(1)$	$O(1)$	$O(1)$	$\Theta(n^2)$
`removeVertex`(v)	$\Theta(m)$	$\Theta(d_v)$	$\Theta(d_v)$	$\Theta(n^2)$
`insertEdge`(u, v, x)	$O(1)$	$O(1)$	$O(1)$ erw.	$O(1)$
`removeEdge`(e)	$O(1)$	$O(d_u + d_v)$	$O(1)$ erw.	$O(1)$
Platzbedarf	$\Theta(n + m)$	$\Theta(n + m)$	$\Theta(n + m)$	$\Theta(n^2)$

n Knoten, m Kanten, Knoten v mit Grad d_v

3 Tiefentraversierung

Ariadne, Theseus und der Minotaurus [Slide 16]

[Pixabay]

Die Überlieferung ist offenbar unvollständig : Theseus muss neben seiner Fadenrolle auch ein Stück Kreide dabei gehabt haben.

Traversierung [Slide 17]

Traversierung:

- Systematische Prozedur, alle Knoten und Kanten eines Graphen zu besuchen.
- Gilt als effizient, falls alle Knoten und Kanten insgesamt in einer Zeit proportional zu ihrer Anzahl besucht werden (d.h. in linearer Zeit).

Beispielanwendungen:

- Finden eines Pfades von Knoten u zu Knoten v, bzw. Beweis seiner Nicht-Existenz.
- Finden von Pfaden von einem Knoten u zu allen (erreichbaren) Knoten v, bzw. Nachweis der Nicht-Erreichbarkeit eines Knotens v von Knoten u.
- Nachweis, ob ein Graph (*stark*) ***zusammenhängend*** ist oder nicht; Berechnung seiner ***Zusammenhangskomponenten*** (*connected components*).

  Ein Graph $G = (V, E)$ ist ***zusammenhängend*** bzw. Subgraph $G = (U \subseteq V, D \subseteq E)$ bildet eine ***Zusammenhangskomponente*** eines Graphen (V, E), falls zwischen jeden zwei beliebigen Knoten in G ein Pfad existiert. G ist ***stark zusammenhängend***, falls jeder Knoten von jedem beliebigen anderen Knoten aus erreichbar ist (nicht Inhalt des Kurses; der Unterschied ist bei *gerichteten* Graphen relevant).

- Berechnung eines ***Spannbaums***.

  Ein ***Spannbaum*** von $G = (V, E)$ ist ein (zusammenhängender) Baum $(V, D \subseteq E)$.

- Finden von Zyklen bzw. Nachweis ihrer Nicht-Existenz.

Tiefentraversierung *(Depth-First Traversal/Search, DFS)* [Slide 18]

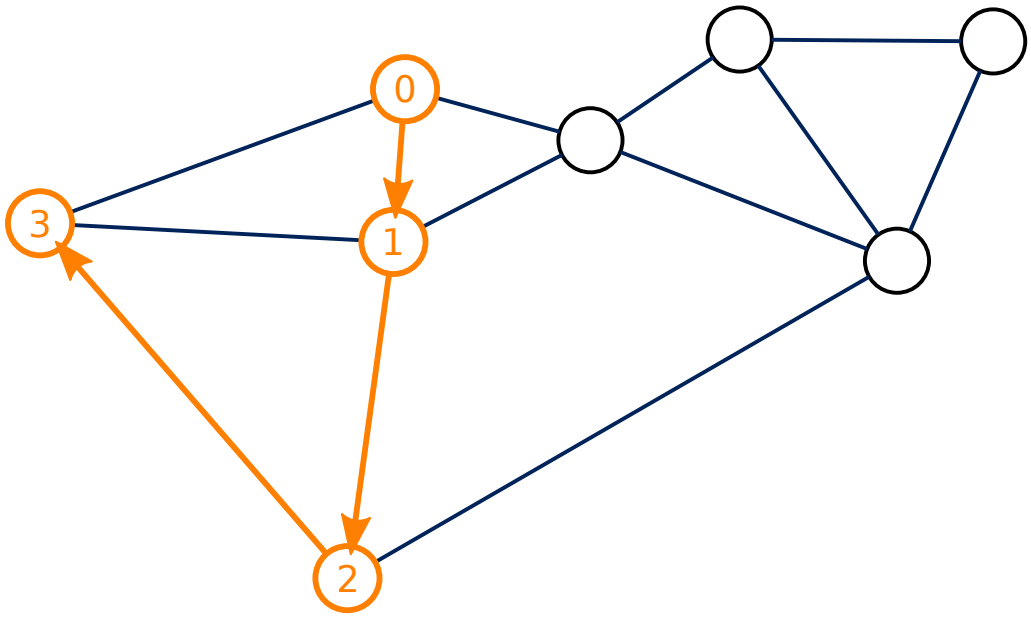
```
Algorithm DFS(G, u):
  Require: A graph G and a vertex u of G.
  Ensure: All vertices reachable from u and their tree edges have been marked.
  mark u as visited
  foreach of u's outgoing edges e = (u, v) do
    if vertex v has not been visited then
      mark e as the tree edge of vertex v
      DFS(G, v)
```

Analog zu Theseus:

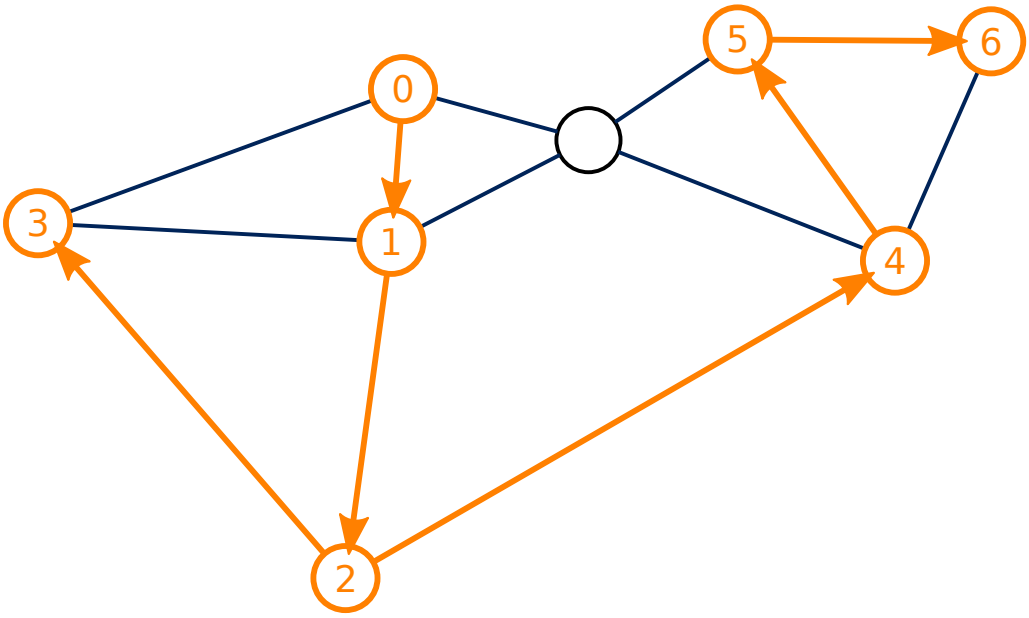
- Markierung wie mit Kreide
- Rückkehr von einem rekursiven Aufruf entspricht dem Zurückverfolgen (und Aufrollen) des Fadens bis zur vorhergehenden Einmündung.

Vgl. BFS

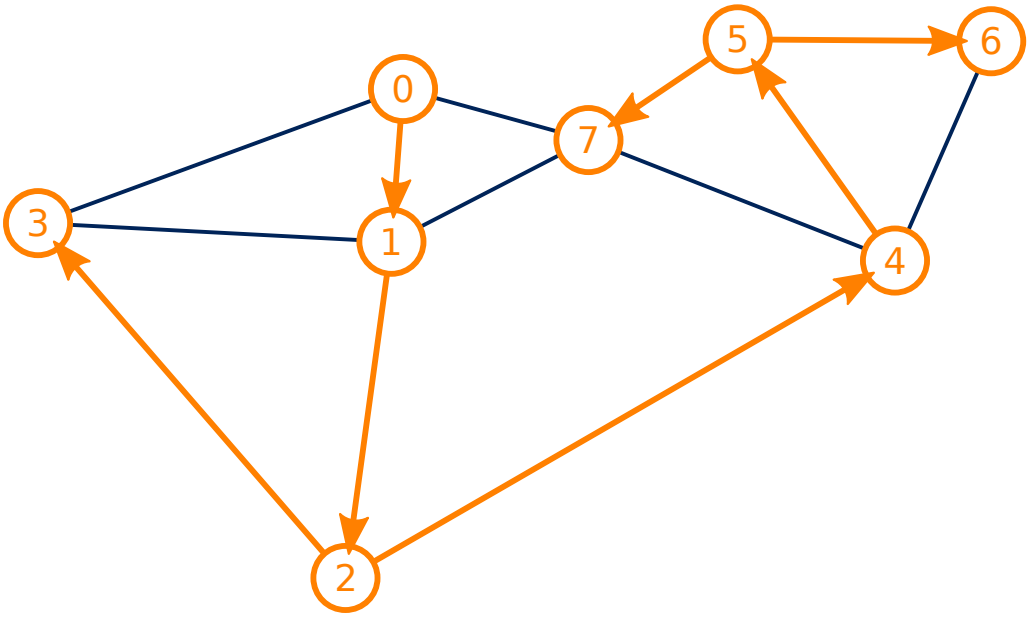
Tiefentraversierung eines ungerichteten Graphen (1) [Slide 19]



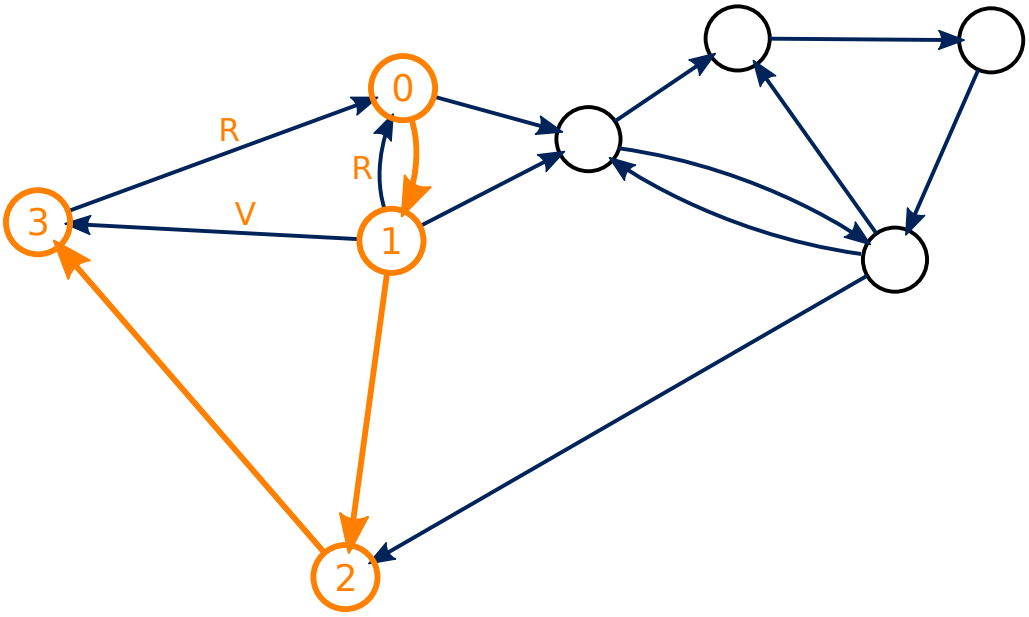
Tiefentraversierung eines ungerichteten Graphen (2) [Slide 20]



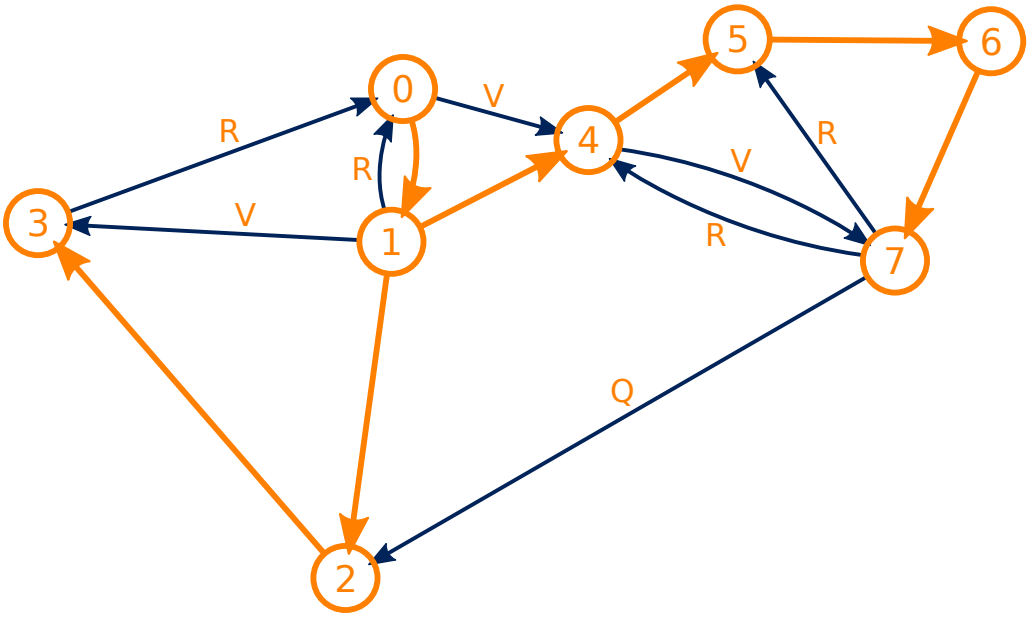
Tiefentraversierung eines ungerichteten Graphen (3) [Slide 21]



Tiefentraversierung eines gerichteten Graphen (1) [Slide 22]



Tiefentraversierung eines gerichteten Graphen (2) [Slide 23]



Tiefensuchbaum und Kantenkategorien [Slide 24]

Tiefentraversierung erzeugt einen ***Tiefensuchbaum*** bestehend aus ***Baumkanten***, die einen neuen Knoten erschließen; alle anderen Kanten sind ***Nichtbaumkanten***.

Nichtbaumkanten in *ungerichteten* Graphen:

- ***Rückwärtskanten***, die zu einem beteits besuchten Knoten zurück führen

Nichtbaumkanten in *gerichteten* Graphen:

- ***Rückwärtskanten***, die zu einem Vorfahren im Tiefensuchbaum führen
- ***Vorwärtskanten***, die zu einem Nachkommen im Tiefensuchbaum führen
- ***Querkanten***, die zu einem Knoten führen, der weder Vorfahr noch Nachkomme des Ausgangsknotens im Tiefensuchbaum ist

Wichtig
Ein Tiefensuchbaum ist kein Suchbaum!

Quiz [Slide 25]

Ein gegebener Tiefensuchbaum determiniert die Kategorisierung aller Nichtbaumkanten.

A: Richtig.

B: Falsch; die Kategorisierung der Nichtbaumkanten hängt von der Reihenfolge ab, in der diese Kanten besucht werden.

D: weiß nicht

Laufzeit der Tiefentraversierung [Slide 26]

Da `DFS()` dank der Markierung für jeden Knoten nur einmal aufgerufen wird und folglich jede Kante höchstens zweimal besucht wird (einmal an jedem Ende), ist die Gesamtlaufzeit $O(n_s + m_s) = O(m_s)$, falls

- `outgoingEdges(`v`)` $O(\text{Grad}(v))$ und `opposite(`v`,`e`)` $O(1)$ Laufzeit beanspruchen, und

  Die Adjazenzliste erlaubt dies, nicht jedoch die Adjazenzmatrix.

- Markierungen eines gegebenen Knotens in konstanter Zeit erstellt und abgefragt werden können,

wobei n_s und m_s jeweils für die Anzahl der vom Startknoten s erreichbaren Knoten und Kanten stehen.

In einer praktischen Implementierung müssen jedoch die Markierungen sämtlicher Knoten und Kanten initialisiert werden. Damit ist die Gesamtlaufzeit $O(n + m)$.

Eigenschaften der Tiefentraversierung [Slide 27]

- Ein Tiefensuchbaum eines *ungerichteten* Graphen ist ein Spannbaum der Zusammenhangskomponente seiner Wurzel.

  Beweis: jeweils recht einfach per Widerspruch (a) für die vollständige Zusammenhangskomponente und (b) für die Baumstruktur

- Ein Tiefensuchbaum eines *gerichteten* Graphen besteht aus gerichteten Pfaden von seinem Wurzelknoten zu jedem erreichbaren Knoten des Ausgangsgraphen.

  Beweis: analog zu (a)

- Jede Rückwärtskante ist Teil eines Zykels (und beweist damit seine Existenz); der Rest dieses Zykels besteht aus Baumkanten.

Welche der Beispielprobleme lassen sich mittels einer Tiefentraversierung effizient lösen?

Alle.

4 Breitentraversierung

Breitentraversierung *(Breadth-First Traversal/Search, BFS)* [Slide 28]

Während die Tiefentraversierung von einem einzelnen Helden durchführbar ist, entspricht die Breitentraversierung dem koordinierten Vorgehen eines Teams, das sich an jedem neu entdeckten Knoten auf alle ausgehenden Kanten aufteilt.

Breitentraversierung erzeugt einen ***Breitensuchbaum*** bestehend aus ***Baumkanten***, die jeweils einen neuen Knoten erschließen; alle anderen Kanten sind ***Nichtbaumkanten***.

Nichtbaumkanten in *ungerichteten* Graphen:

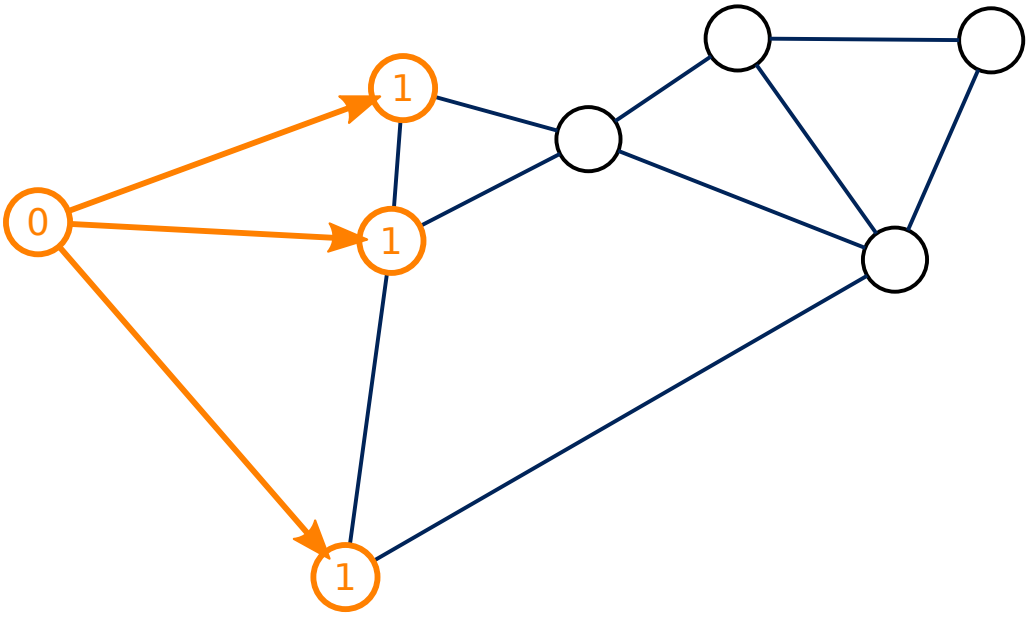
- ***Querkanten***, die zu einem simultan besuchten Knoten führen (der also weder ein Vorfahr noch ein Nachkomme im Breitensuchbaum ist)

Nichtbaumkanten in *gerichteten* Graphen:

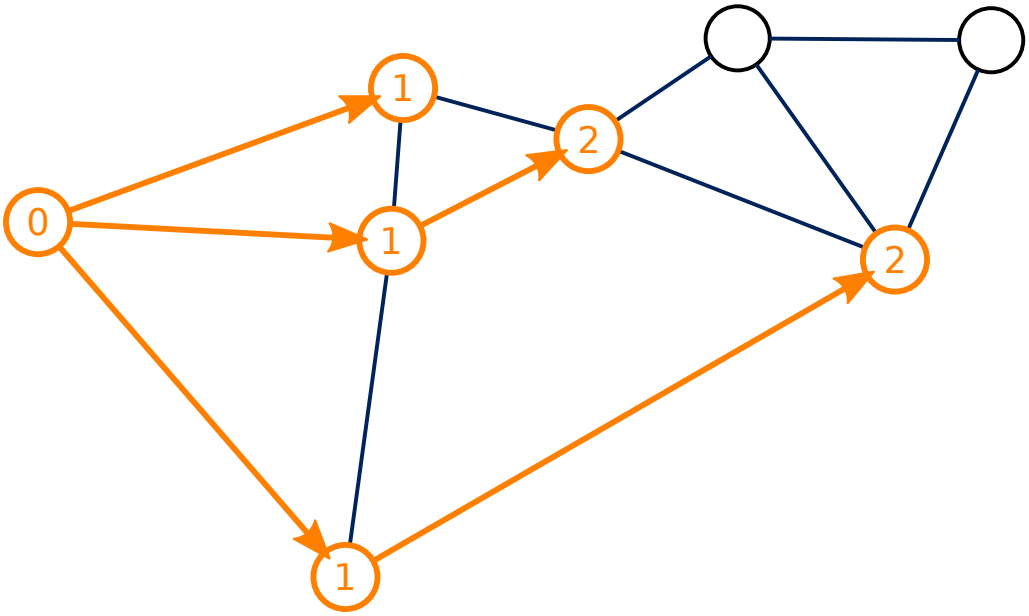
- ***Rückwärtskanten***, die zu einem Vorfahren im Breitensuchbaum führen
- ***Querkanten***

Wichtig
Ein Breitensuchbaum ist kein Suchbaum!

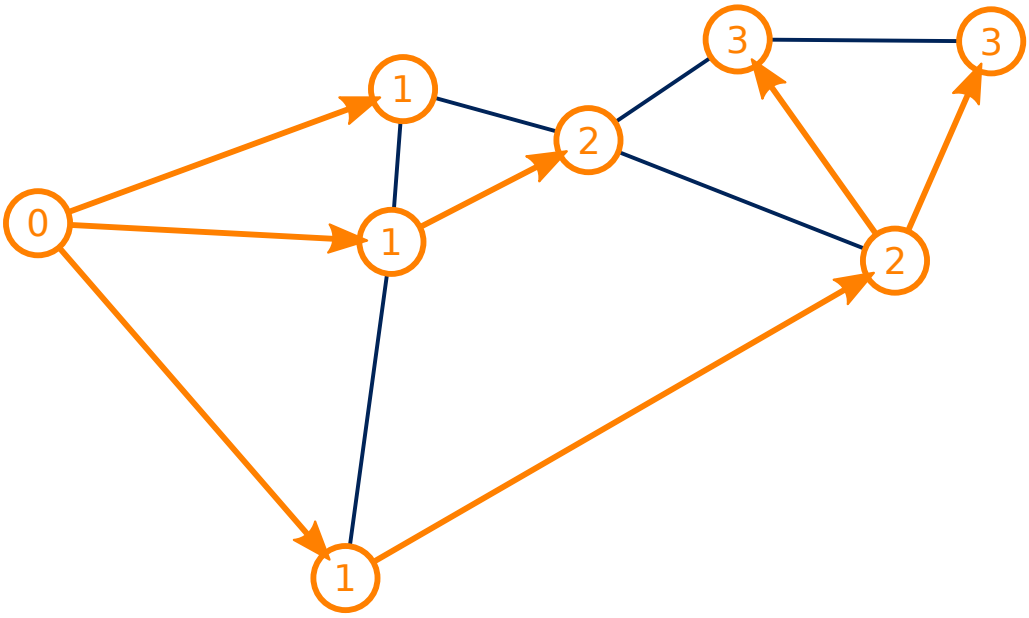
Breitentraversierung eines ungerichteten Graphen (1) [Slide 29]



Breitentraversierung eines ungerichteten Graphen (2) [Slide 30]

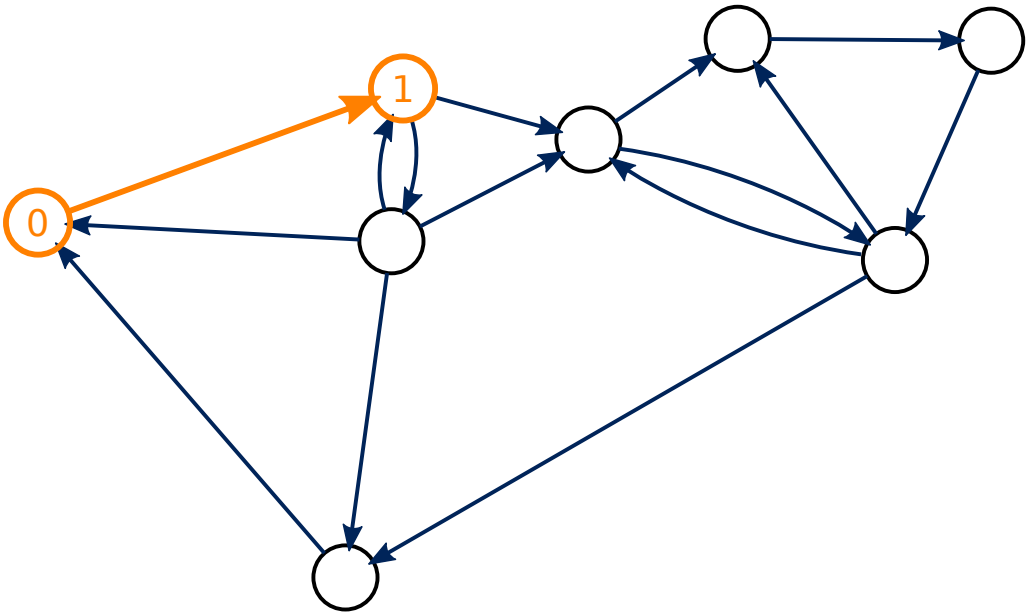


Breitentraversierung eines ungerichteten Graphen (3) [Slide 31]

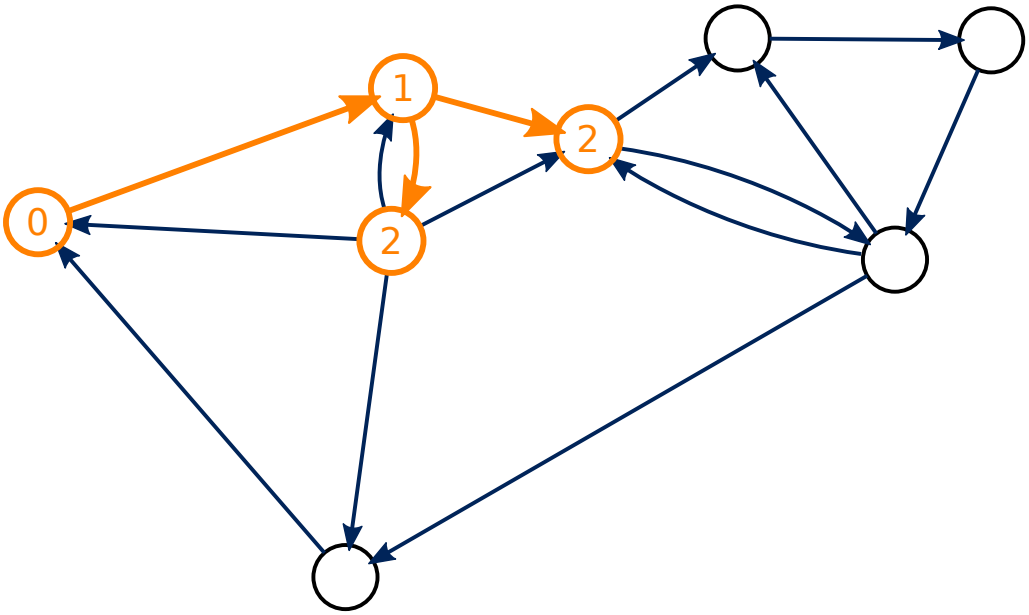


Eine Breitentraversierung eines Graphen teilt seine Knoten in *Niveaus* ein, die jeweils diejenigen Knoten enthalten, die von den koordinierten metaphorischen Suchtrupps quasi gleichzeitig erreicht werden. Somit ist das Niveau eines Knotens seine *Tiefe im Breitensuchbaum*, hier als orange Zahl dargestellt.

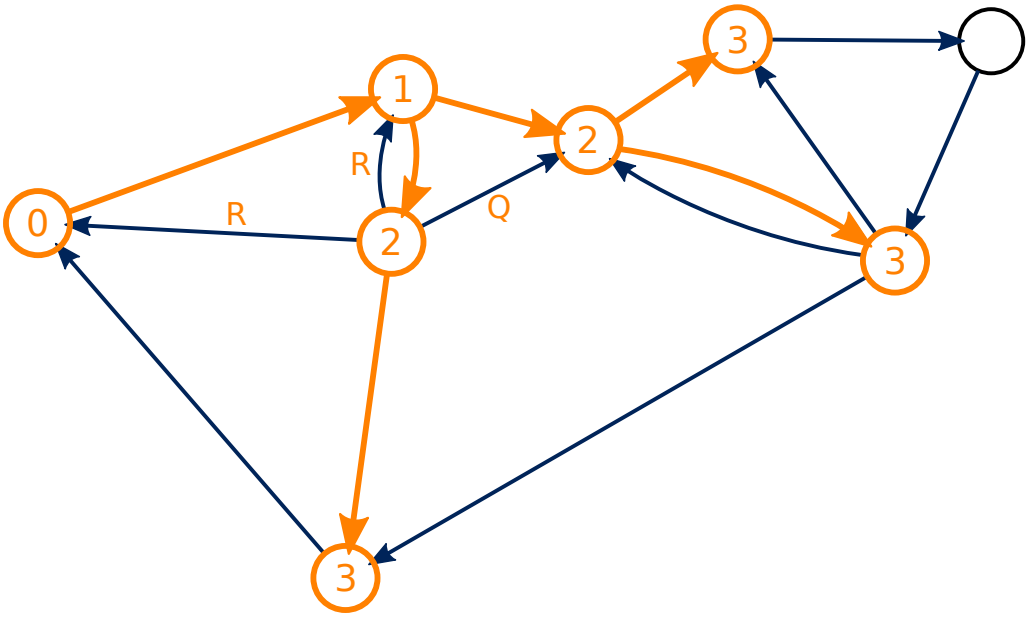
Breitentraversierung eines gerichteten Graphen (1) [Slide 32]



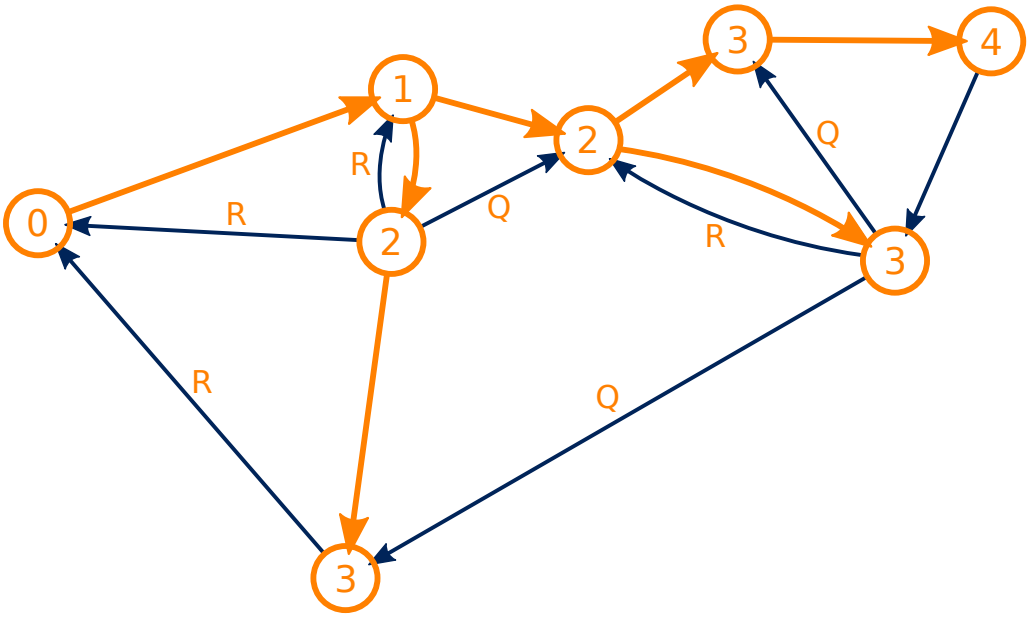
Breitentraversierung eines gerichteten Graphen (2) [Slide 33]



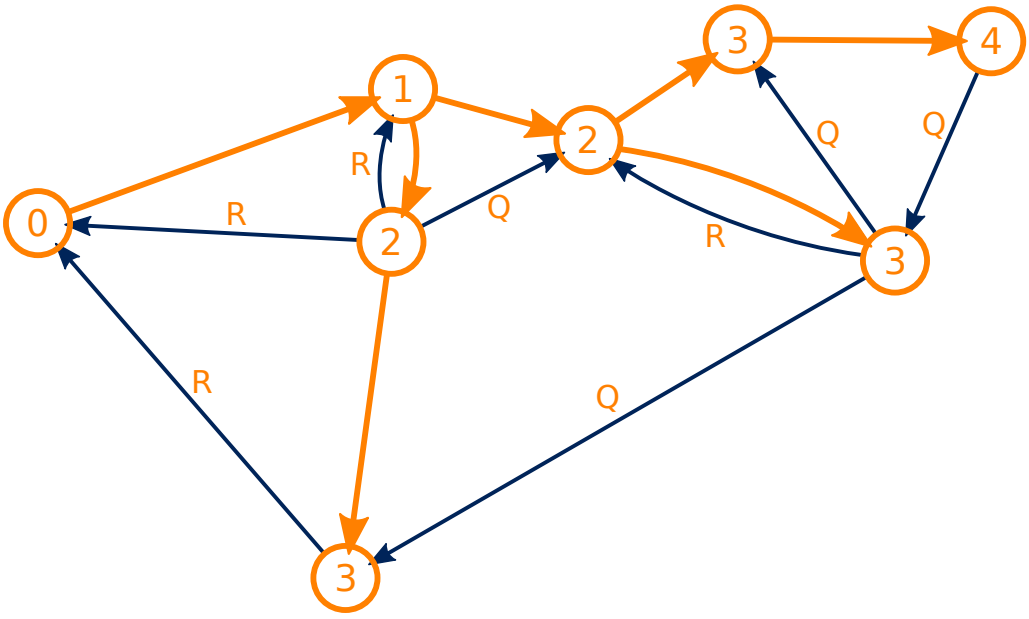
Breitentraversierung eines gerichteten Graphen (3) [Slide 34]



Breitentraversierung eines gerichteten Graphen (4) [Slide 35]



Breitentraversierung eines gerichteten Graphen (5) [Slide 36]



Quiz [Slide 37]

Die Niveaus eines Graphen bei einer Breitentraversierung ausgehend von einem bestimmten Knoten sind …

A: eindeutig.

B: nicht eindeutig; sie hängen davon ab, in welcher Reihenfolge die ausgehenden Kanten eines Knotens besucht werden.

D: weiß nicht

Breitentraversierung: Algorithmus [Slide 38]

```
Algorithm BFS(G, u):
  Require: A graph G and a vertex u of G.
  Ensure: All vertices reachable from u and their tree edges have been marked.
  mark u as visited
  initialize queue Q to contain u
  while not Q.isEmpty() do
    v = Q.dequeue()
    foreach of v's outgoing edges e = (v, w) do
      if vertex w has not been visited then
        mark w as visited
        mark e as the tree edge of vertex w
        Q.enqueue(w)
```

Ähnlich wie die Breitentraversierung eines Baums dessen Knoten Zeile für Zeile besucht, besucht die Breitentraversierung eines Graphen seine Knoten Niveau für Niveau. Diese Einteilung in Niveaus ist in diesem Algorithmus nicht explizit sichtbar.
Vgl. DFS

Eigenschaften der Breitentraversierung [Slide 39]

- Ein Breitensuchbaum eines ungerichteten Graphen ist ein Spannbaum der Zusammenhangskomponente seiner Wurzel.
- Ein Breitensuchbaum besteht aus ***kürzesten*** Pfaden von seinem Wurzelknoten zu jedem erreichbaren Knoten des Ausgangsgraphen, wobei die Pfadlänge durch das Niveau des Zielknotens gegeben ist.
- Laufzeit?

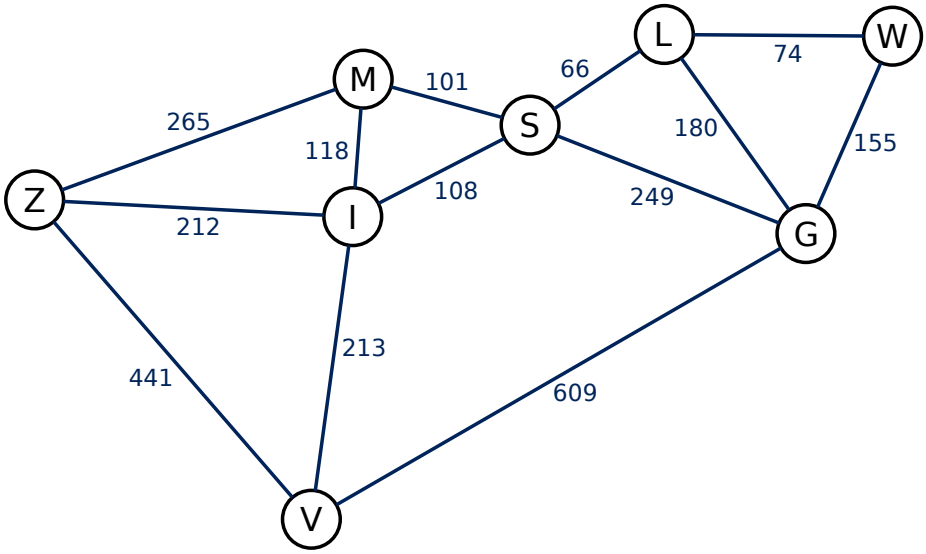
  identisch zur Tiefentraversierung

Welche der Beispielprobleme lassen sich mittels einer Breitentraversierung effizient lösen?

Für *ungerichtete* Graphen alle; für manche Probleme auf *gerichteten* Graphen ist die Breitentraversierung nicht unmittelbar geeignet, wie z.B. die Suche nach gerichteten Zyklen oder die Identifizierung der starken Zusammenhangskomponenten.

5 Kürzeste Pfade und Dijkstras Algorithmus

Kantenbewerteter *(Weighted)* Graph [Slide 40]



Der ***kürzeste Pfad*** von Salzburg nach Graz geht durch Linz und hat einen Gesamtwert von 246.

Vgl. Breitentraversierung

Kürzeste Pfade in kantenbewerteten Graphen [Slide 41]

e	$=$	(u, v)	Kante
$w(e)$	$=$	$w(u, v)$	Wert einer Kante
P	$=$	$((v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k))$	Pfad
$w(P)$	$=$	$\sum_{l=0}^{k-1} w(v_l, v_{l+1})$	Wert eines Pfades
$d(u, v)$	$=$	$\min_P\{w(((u, .), \dots, (., v)))\}$	Wert eines kürzesten Pfades von u nach v

Anmerkung

- $d(u, v) = \infty$ falls kein Pfad von u nach v existiert.
- Enthält die Zusammenhangskomponente von u und v einen Zykel negativen Gesamtwerts, dann gibt es keinen kürzesten Pfad von u nach v, und $d(u, v)$ ist nicht definiert.

Dijkstras Algorithmus: Idee [Slide 42]

Findet kürzeste Pfade von einem designierten Knoten s zu allen anderen Knoten eines Graphen *(single-source shortest paths)*.

- Ähnlich einer gewichteten Breitentraversierung
- Erweitert iterativ eine *Wolke* C von Knoten jeweils um den *am nächsten liegenden* Knoten außerhalb der Wolke (*gierig*)
- Sei $D[v]$ zu jeder Zeit die Länge des kürzesten, bekannten Pfades von s nach v.
- Zu Beginn: $D[s] = 0$, $D[v] = \infty \; \forall v \neq s$, $C = \emptyset$.
- Iteriere:
  - Wähle $\operatorname{argmin}_{u \notin C}\{D[u]\}$ und füge sie zu C hinzu.
  - Bringe die $D[v]$ aller Nachbarn $v \notin C$ von u auf den neuesten Stand (***edge relaxation***):

    if $D[u] + w(u,v) < D[v]$ **then**
    $D[v] \leftarrow D[u] + w(u,v)$

Dijkstras Algorithmus [Slide 43]

Algorithm DijkstraShortestPaths(G, s):
Require: *A graph G with nonnegative edge weights, and a distinguished vertex s of G.*
Ensure: *Return the length of a shortest path from s to v for each vertex v of G.*
$D[s] \leftarrow 0$
foreach vertex $v \neq s$ **of** G **do**
  $D[v] \leftarrow +\infty$
$Q \leftarrow$ **an adaptable PQ containing all** $v \in V$ **using the** $D[v]$ **as keys**
while not Q.isEmpty() **do**
  $u \leftarrow Q$.removeMin()
  foreach edge (u,v) **with** $v \in Q$ **do**
    if $D[u] + w(u,v) < D[v]$ **then**
      $D[v] \leftarrow D[u] + w(u,v)$
      update Q **accordingly**
return D

Quiz [Slide 44]

Dijkstras Algorithmus, wie beschrieben, eignet sich …

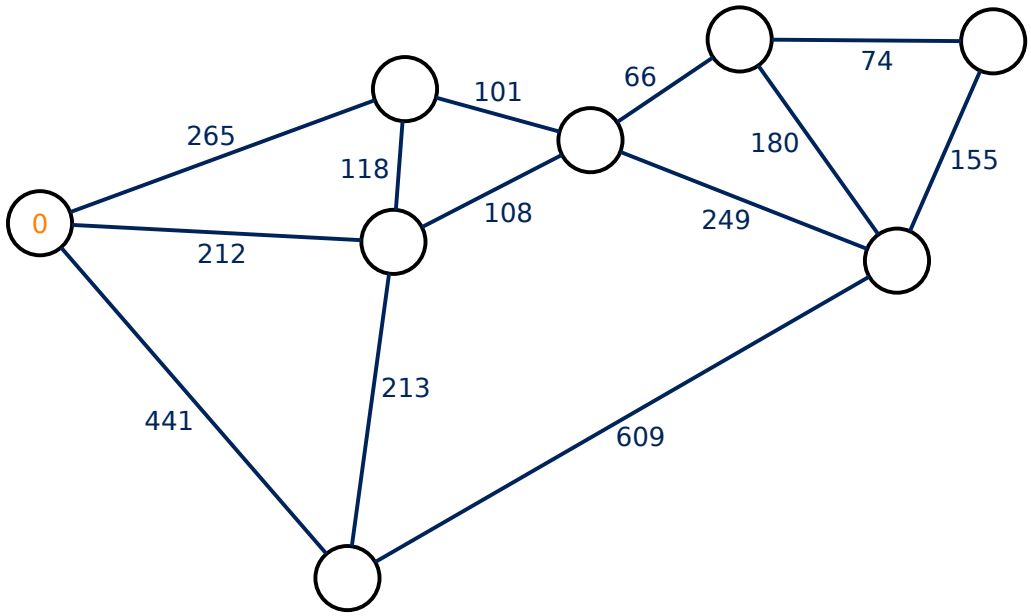
A: nur für gerichtete Graphen

B: nur für ungerichtete Graphen

C: für gerichtete und ungerichtete Graphen gleichermaßen

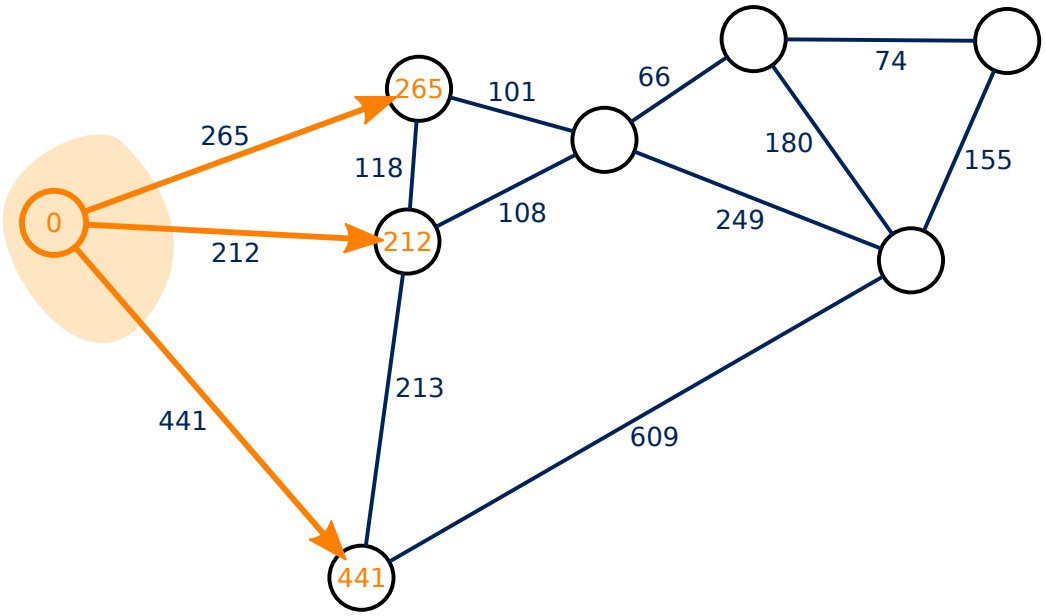
D: weiß nicht

Beispielablauf von Dijkstras Algorithmus (0) [Slide 45]



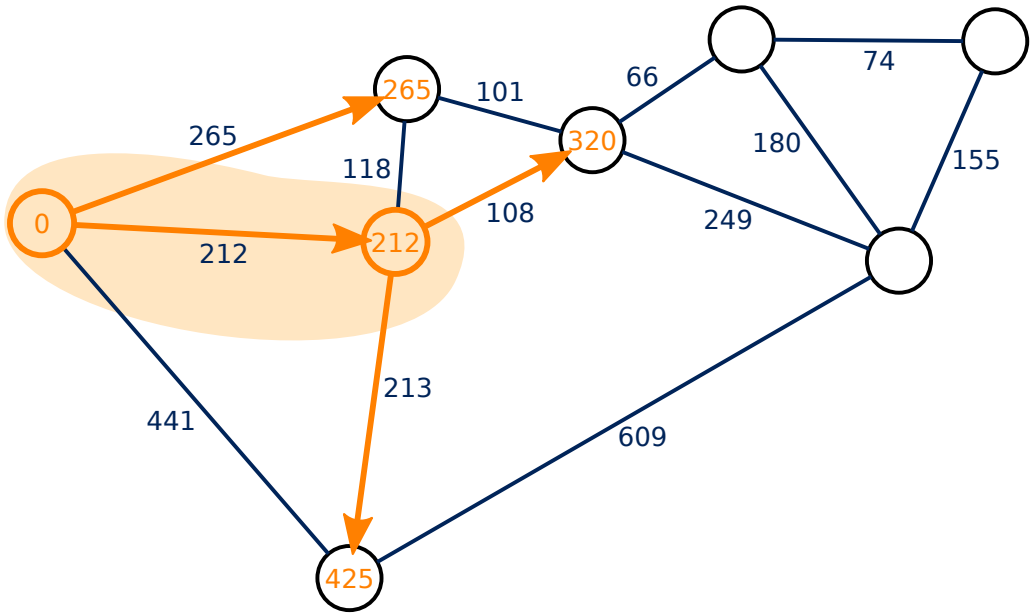
Der Startknoten s ist Zürich. Jeder Knoten v enthält den Wert $D[v]$; ein leerer Knoten v hat den Wert $D[v] = \infty$.

Beispielablauf von Dijkstras Algorithmus (1) [Slide 46]



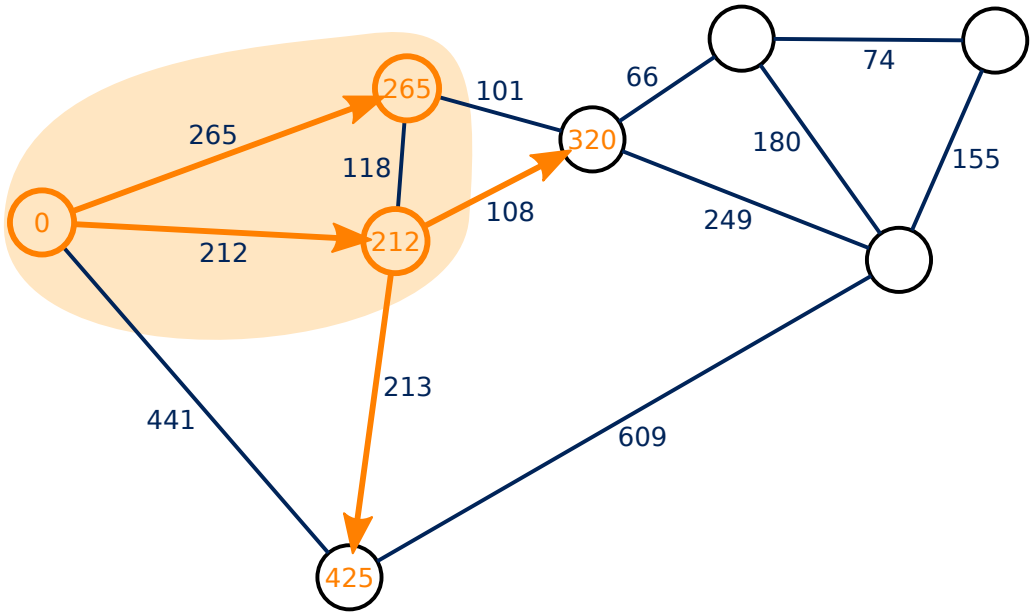
C ist hell orange hinterlegt. Für jeden Knoten $v \notin C$ ist die Verbindung vom jeweils nächsten Knoten $u \in C$ als fetter, oranger Pfeil dargestellt.

Beispielablauf von Dijkstras Algorithmus (2) [Slide 47]

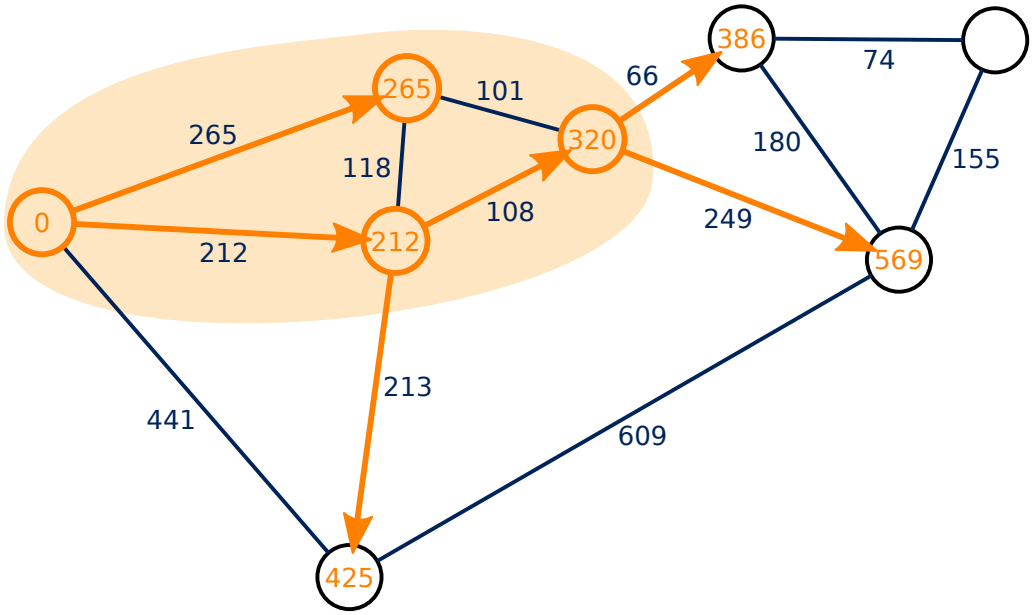


Die Kanten des Baums der kürzesten Pfade sind ebenfalls als fette, orange Pfeile dargestellt, und befinden sich komplett innerhalb von C.
Edge relaxation findet einen kürzeren Pfad nach Verona.

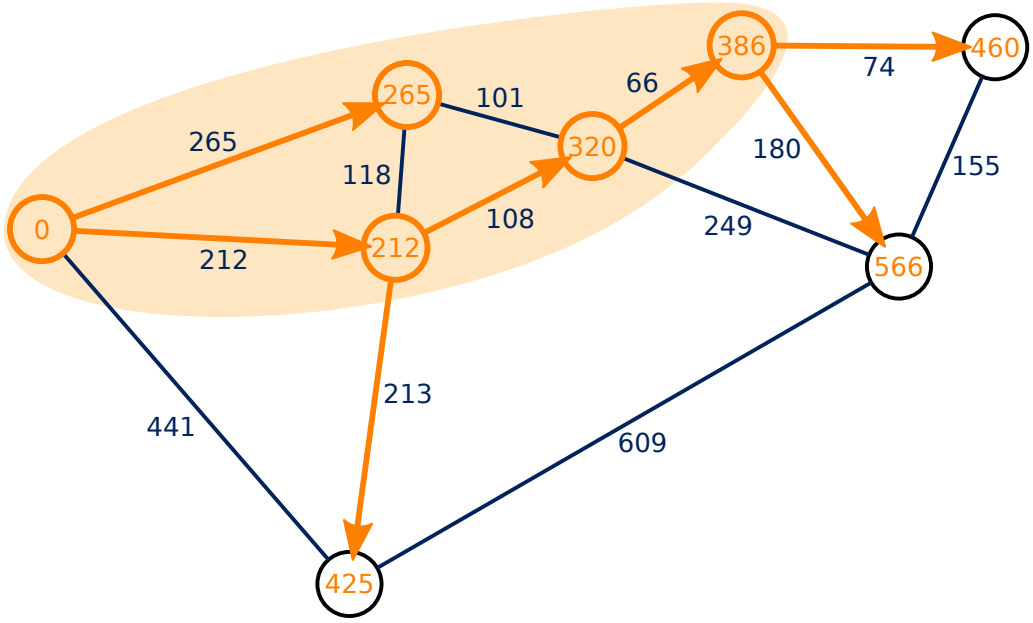
Beispielablauf von Dijkstras Algorithmus (3) [Slide 48]



Beispielablauf von Dijkstras Algorithmus (4) [Slide 49]

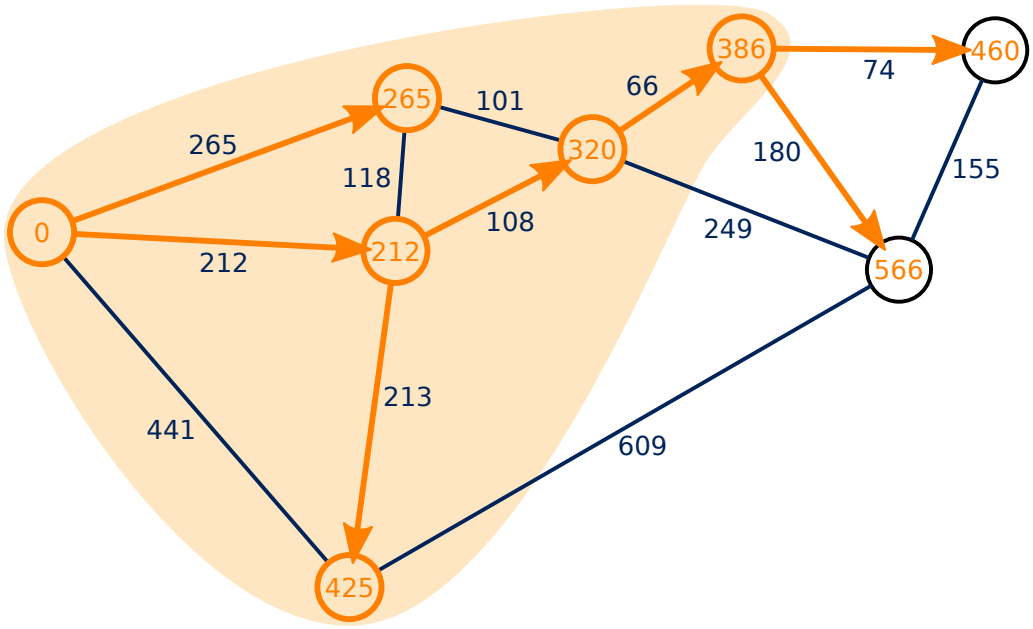


Beispielablauf von Dijkstras Algorithmus (5) [Slide 50]

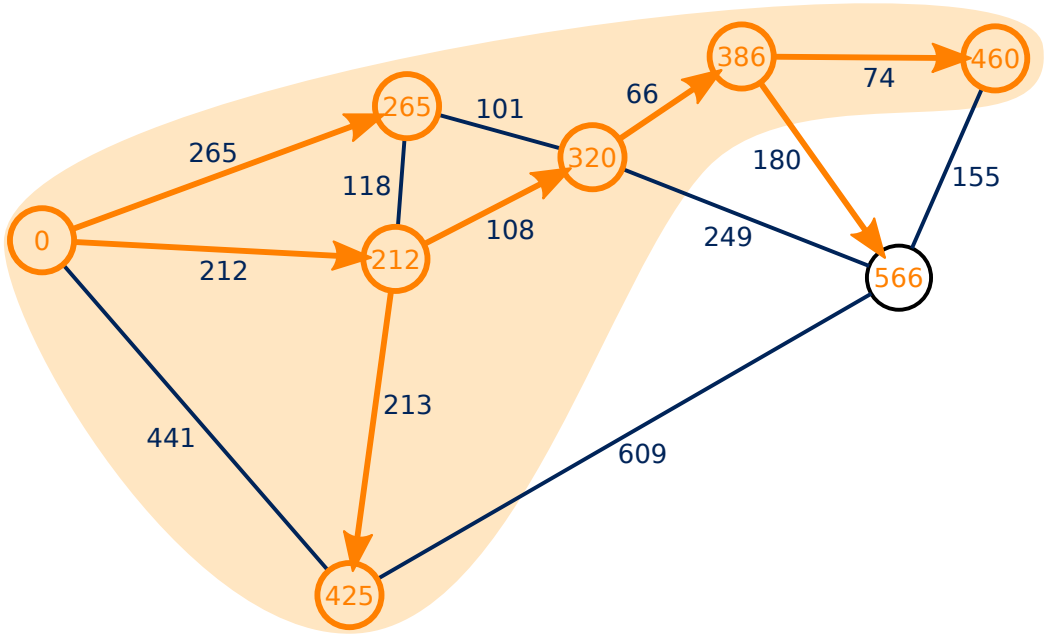


Edge relaxation findet einen kürzeren Pfad nach Graz.

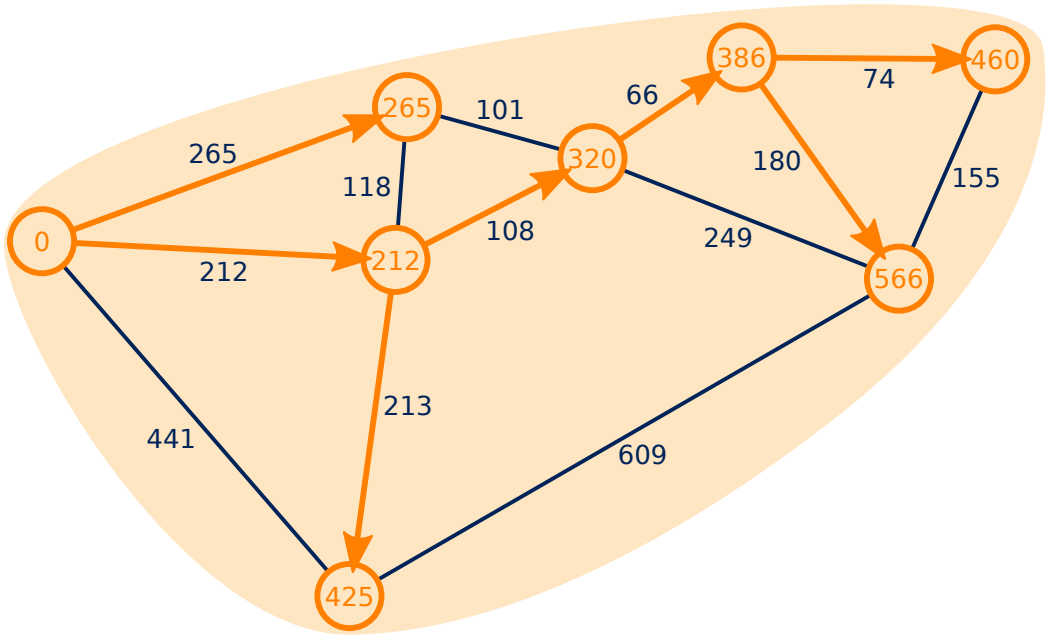
Beispielablauf von Dijkstras Algorithmus (6) [Slide 51]



Beispielablauf von Dijkstras Algorithmus (7) [Slide 52]

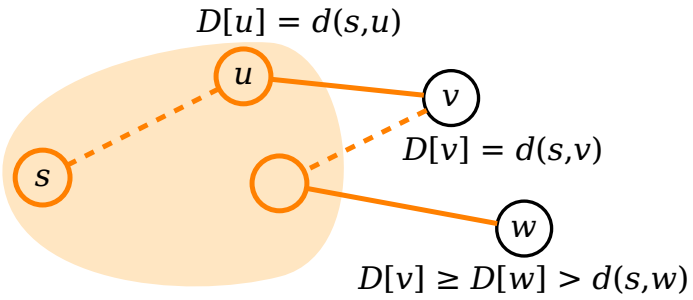


Beispielablauf von Dijkstras Algorithmus (8) [Slide 53]



Korrektheit von Dijkstras Algorithmus [Slide 54]

Proposition: Wann immer ein Knoten w in die Wolke gezogen wird, gilt $D[w] = d(s, w)$.



Beweis (durch Widerspruch):

- Sei w der erste Knoten, der mit $D[w] > d(s, w)$ in die Wolke gezogen wird. Also muss $D[w] \leq D[v]$ sein.
- Da der Graph keine negativen Kantenbewertungen besitzt, muss $D[w] \geq D[v]$ sein.
- Da $D[v] = d(s, v)$ ist, kann nicht $D[w] = D[v] > d(s, w)$ sein. Widerspruch!

- Sei v der erste Knoten $\notin C$ auf dem kürzesten Pfad P von s nach w, und sei $u \in C$ (mit $D[u] = d(s, u)$) der Vorgänger von v in P.
- Als u nach C gezogen wurde, wurde (per edge relaxation) sichergestellt, dass $D[v] \leq D[u] + w(u, v) = d(s, u) + w(u, v)$. Da v jedoch der auf u folgende Knoten in P ist, folgt daraus $D[v] = d(s, v)$.
- Da nun w nach C gezogen wird (und nicht v), muss $D[w] \leq D[v]$ sein.
- Da jedes Segment eines kürzesten Pfades seinerseits ein kürzester Pfad ist, muss $d(s, v) + d(v, w) = d(s, w)$ sein.
- Da der Graph keine negativen Kantenbewertungen besitzt, ist $d(v, w) \geq 0$, und $D[w] \leq D[v] = d(s, v) \leq d(s, v) + d(v, w) = d(s, w)$.
- Dies widerspricht jedoch der Definition von w; daher kann w nicht existieren.

Laufzeit von Dijkstras Algorithmus [Slide 55]

- Der Graph G habe n Knoten und m Kanten.
- Annahmen:
  - Kantenbewertungen können in konstanter Zeit hinzugefügt, abgefragt und verglichen werden.
  - G ist mittels einer *Adjacency List* oder *Adjacency Map* implementiert.
- Die `while`-Schleife iteriert $O(n)$ Mal.
- Die Gesamtzahl der Iterationen der inneren `foreach`-Schleife ist $\sum_{u \in V} \text{Ausgangsgrad}(u) \in O(m)$.
- Die Laufzeit-dominanten Operationen sind also:
  - n Einfügungen in Q
  - n Aufrufe von `removeMin()`
  - m Aufrufe von `replaceKey()` der adaptiven Vorrangwarteschlange

    Teil der *adaptiven Vorrangwarteschlange*; nicht besprochen, jedoch leicht ersichtlich.

  Basiert Q auf einem Heap, laufen diese in $O(\log n)$, und die Gesamtlaufzeit ist $O((n+m)\log n) = O(m \log n) \subseteq O(n^2 \log n)$.

  Basiert Q auf einer unsortierten Liste, läuft `removeMin()` in $O(n)$ Zeit, aber `replaceKey()` in $O(1)$. Daher ist die Gesamtlaufzeit $O(n^2 + m) = O(n^2)$.

Wir bevorzugen also den Heap für relativ *lichte* Graphen mit $m < \frac{n^2}{\log n}$.

Anmerkung
Mit einem Fibonacci *heap* (nicht besprochen) kann man eine Gesamtlaufzeit von Dijkstras Algorithmus in $O(m + n \log n)$ garantieren.

Rekonstruktion des Baums der kürzesten Pfade [Slide 56]

Idee: Falls für eine Kante (u, v) gilt, dass $D[u] + w(u, v) = D[v]$, dann liegt sie auf einem kürzesten Pfad von s nach v.

Der Code prüft diese Eigenschaft für jede eingehende Kante e jedes Knotens v des Graphen g (außer dem Startknoten s).

```
Algorithm spTree(G, s, D):
  Require: G, s, and D as received and returned by DijkstraShortestPaths().
  Ensure:  Return a map with keys v and values e such that
                vertex v is reached via edge e in the tree of shortest paths.
  T ← empty map
  foreach v ∈ D do
    if v ≠ s then
      foreach e ∈ incomingEdges(v) do
        u ← opposite(v, e)
        if D[u] + w(u, v) = D[v] then
          T.put(v, e)                    // edge e is used to reach v
  return T
```

Falls für einen Knoten mehrere Kanten die Baumkanteneigenschaft erfüllen, überschreibt `T.put(v,e)` den alten Wert des Schlüssels v mit dem neuen. So ist sichergestellt, dass im Baum T jeder Knoten nur eine eingehende Kante besitzt und T damit frei von Zykeln ist.

6 Literatur

Literatur [Slide 57]

Goodrich, Michael, Roberto Tamassia und Michael Goldwasser (Aug. 2014). *Data Structures and Algorithms in Java*. Wiley.