Algorithmen und Datenstrukturen

Graphen

Prof. Justus Piater, Ph.D.

8. Juni 2022

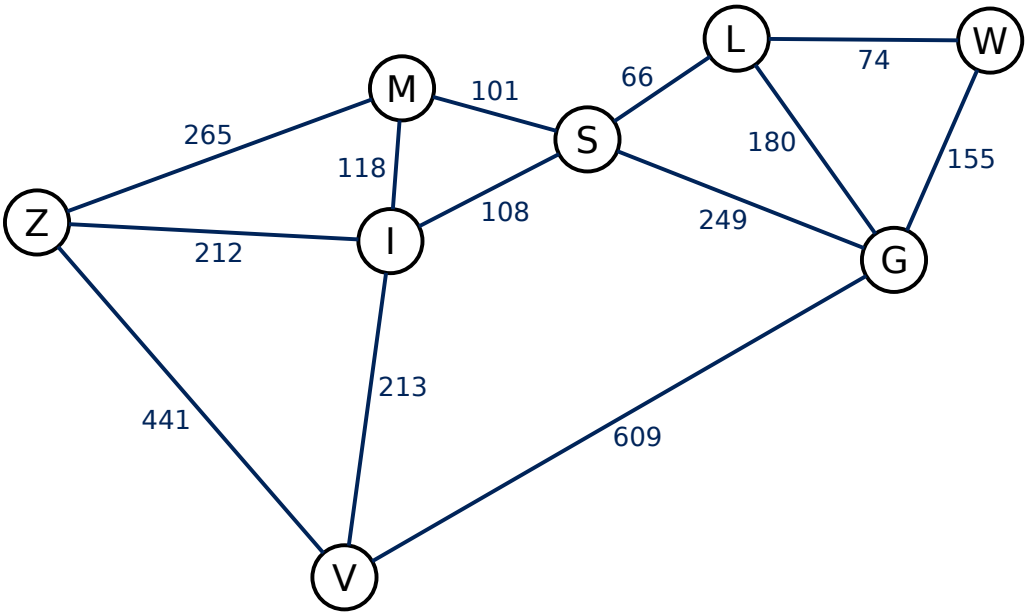
Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch *Data Structures and Algorithms in Java* [Goodrich u. a. 2014].

Inhaltsverzeichnis

1 Konzepte und ADT

Ein ***Graph*** beschreibt paarweise Relationen zwischen Elementen. Die Bezeichnung *Graph* hat ihren Ursprung in seiner graphischen, also bildlichen Darstellung. Bevor wir Graphen als mathematisches Objekt und als abstrakten Datentyp einführen, sehen wir uns ein paar Beispiele an.

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.)

Dieser Graph beschreibt Fahrtzeiten zwischen zwischen Städten. Die mit den Buchstaben markierten ***Knoten*** stehen für Innsbruck, Salzburg, Linz, Wien, Graz, Verona, Zürich und München, in maßstabsgetreuer Anordnung. Die ***Kanten*** zwischen den Städten sind mit aktuellen Fahrtzeiten typischer Bahnverbindungen in Minuten bezeichnet. Nur zwischen Zürich und Verona gibt es in Wirklichkeit deutlich schnellere Verbindungen, und zwischen Salzburg und Graz gehen wir von einer Baustelle aus, die die Fahrt um 10 Minuten verzögert.

Mit Hilfe eines solchen Graphen kann z.B. ein Paketzustelldienst (bzw. eine automatische Fahrplanauskunft) ausrechnen, über welche Bahnverbindungen ein Paket am effizientesten von einer Stadt in die andere geschickt werden kann. Darüber werden wir später noch mehr erfahren.

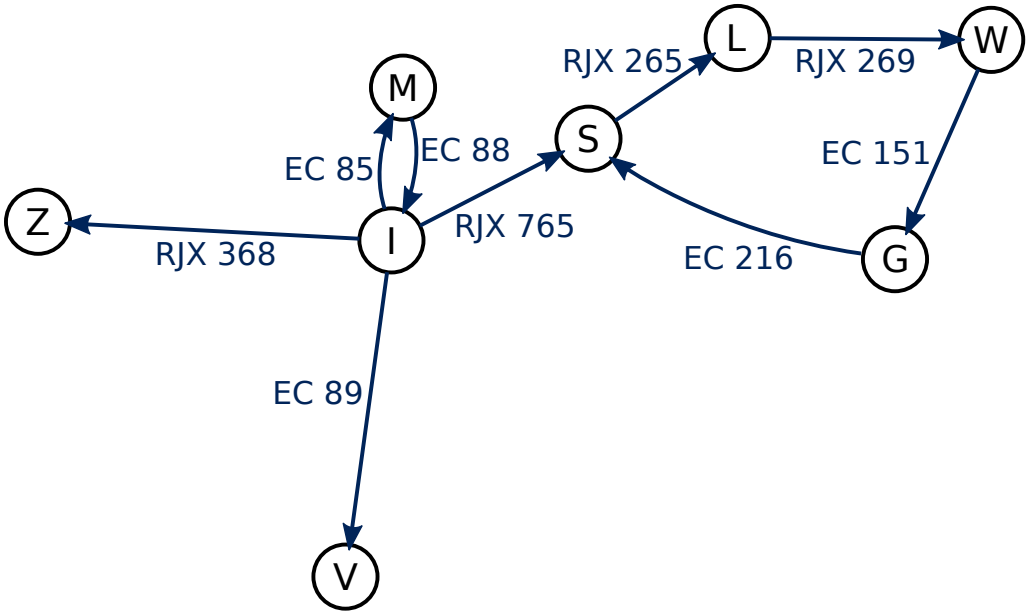
Zwei Knoten, die durch eine Kante verbunden sind, sind zueinander ***adjazent***. Beispielsweise sind Innsbruck und Salzburg adjazent, aber Innsbruck und Graz sind nicht adjazent.

Ein Knoten und eine Kante, die miteinander verbunden sind, sind zueinander ***inzident***. Hier ist die mit 108 bezeichnete Kante zu Innsbruck und zu Salzburg inzident, und umgekehrt sind Innsbruck und Salzburg jeweils zu dieser Kante inzident.

In diesem Graphen repräsentieren die Kanten symmetrische Fahrtzeiten; sie sind also ***ungerichtet***. Da alle Kanten dieses Graphen ungerichtet sind, handelt es sich um einen

ungerichteten Graphen.

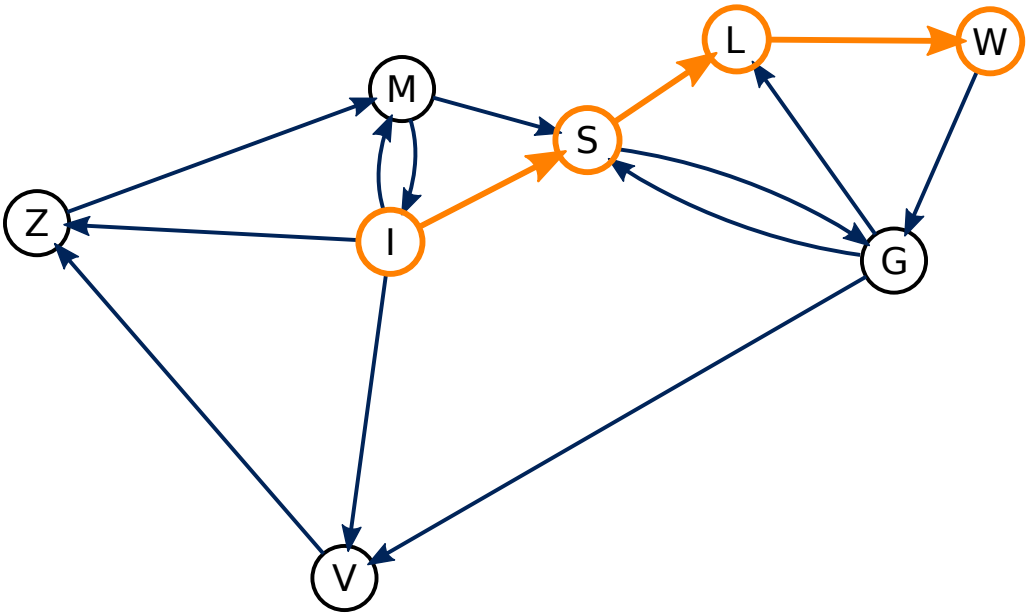
Beispiel: Bahnverbindungen [Slide 2]



Hier sehen wir einen *gerichteten* Graphen. Alle seine Kanten sind *gerichtet*, wie durch den Pfeil angegeben, und sind jeweils mit der Zugnummer einer typischen Verbindung markiert. Diese Züge verkehren bekanntlich in einer Richtung. Züge in der umgekehrten Richtung tragen andere Nummern. Möchten wir beide Richtungen in unserem Graphen repräsentieren, dann müssen wir gerichtete Kanten in beiden Richtungen verwenden, wie die Verbindungen zwischen Innsbruck und München.

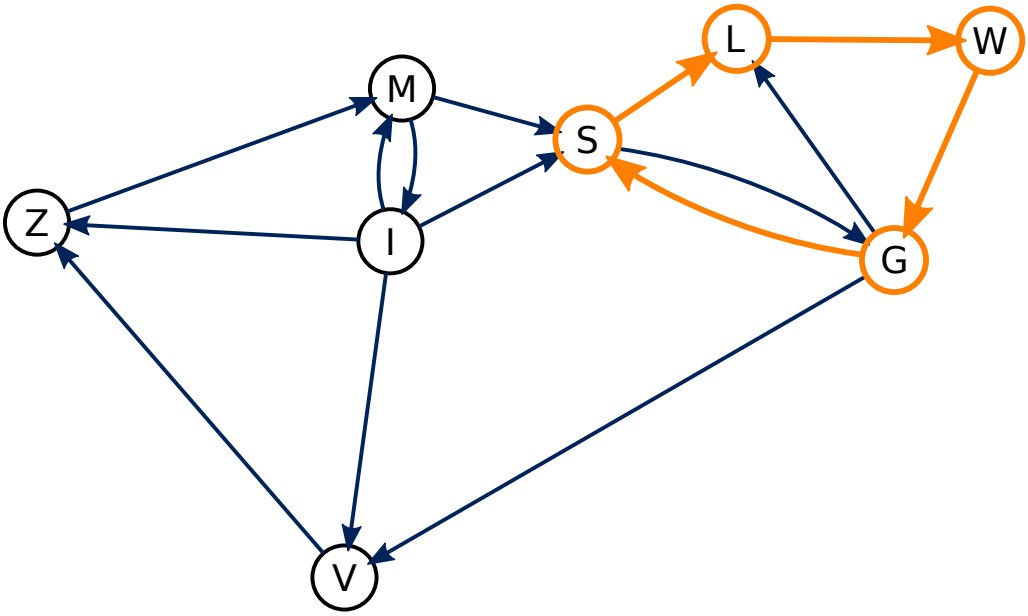
Somit sind zum Knoten Innsbruck vier *ausgehende* Kanten und eine *eingehende* Kante inzident. Anders gesagt, der Knoten Innsbruck hat einen *Ausgangsgrad* von 4 und einen *Eingangsgrad* von 1.

Gerichteter Pfad [Slide 3]



Eine Sequenz inzidenter Knoten und Kanten, die mit jeweils einem Knoten beginnt und endet, bildet einen *Pfad*. Hier sehen wir einen gerichteten Pfad von Innsbruck nach Wien. Ein Pfad kann gerichtet oder ungerichtet sein, aber er kann nur gerichtete Kanten enthalten, die der Richtung des Pfades entsprechen.

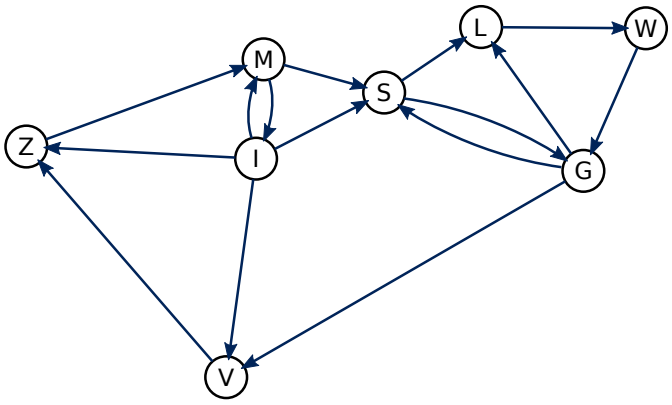
Gerichteter Zykel [Slide 4]



Übung: Finden Sie alle gerichteten Zykel in diesem Graphen! Wie viele Kanten müssen Sie entfernen, um alle Zykel zu eliminieren? Sind diese Kanten eindeutig?

Ein Pfad, dessen erster und letzter Knoten identisch sind, heißt *Zykel*. Hier ist einer von vielen Zykeln hervorgehoben, die dieser Graph besitzt.

Quiz [Slide 5]



Wie viele (gerichtete) Zykel enthält dieser Graph?

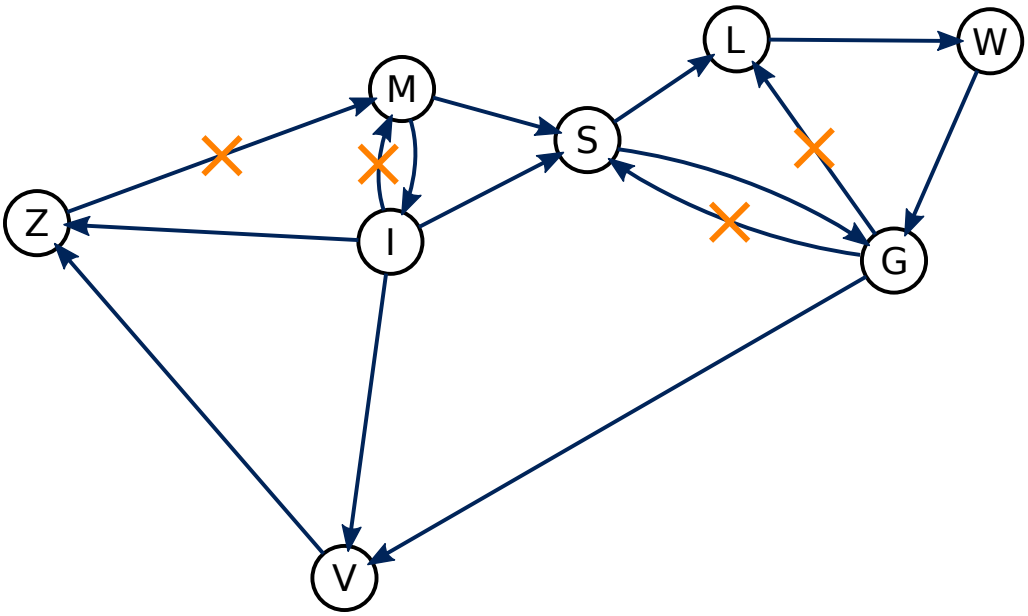
A: 4

B: 6

C: mehr als 8

D: weiß nicht

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



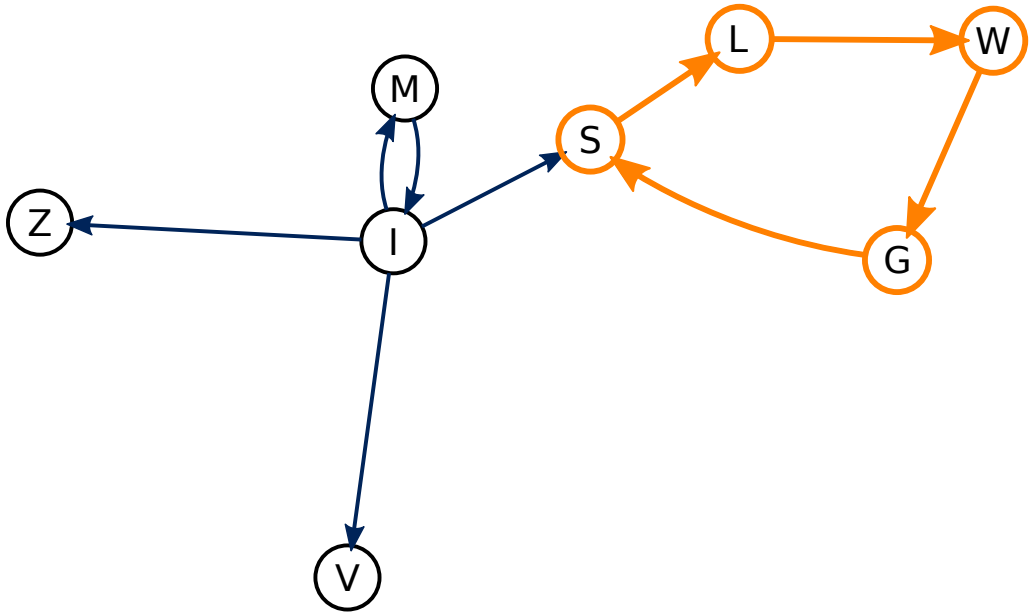
Wir können Zykel eliminieren, indem wir einzelne Kanten entfernen. Nach dem Entfernen dieser vier Kanten ist der Graph frei von Zykeln, und ist damit ein *Directed Acyclic Graph*.

Gerichtete Graphen ohne Zykel sind in vielen Anwendungen von großer Bedeutung. Beispielsweise können sie Abhängigkeiten in Berechnungs- oder Produktionsverfahren beschreiben. Eine Kante von a nach b drückt dann aus, dass Aufgabe a erledigt sein muss,

bevor Aufgabe b beginnen kann. Aufgaben, die nicht durch einen gerichteten Pfad miteinander verbunden sind, können parallel bearbeitet werden.

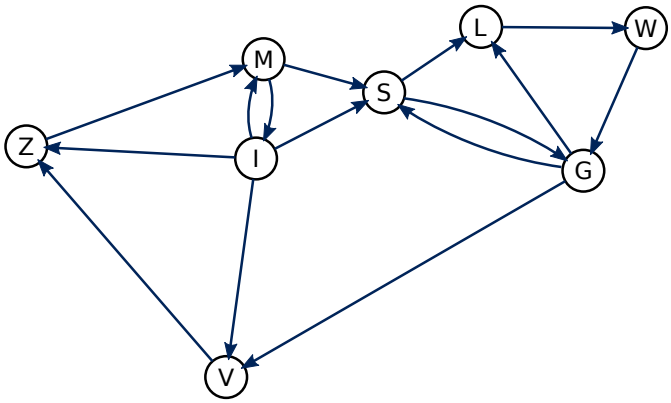
Lassen wir in diesem Graphen die markierten Kanten intakt, dann sind alle seine Knoten von jedem anderen Knoten über gerichtete Pfade erreichbar.

Erreichbarkeit ausgehend von Salzburg [Slide 7]



In diesem reduzierten Graphen sind jedoch von Salzburg aus nicht alle Knoten erreichbar, denn es existiert kein gerichteter Pfad von Salzburg nach Innsbruck, Verona, Zürich oder München.

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

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*.

> Fassen wir hier die wichtigsten Definitionen und Terminologie zu Graphen zusammen.
>
> Ein Graph ist ein geordnetes Paar zweier Mengen, nämlich der Menge V seiner Knoten und der Menge E seiner Kanten. Knoten werden auch als Ecken bezeichnet, englisch *vertex*, plural *vertices*.
>
> Eine Kante wird für unsere Zwecke durch das Paar ihrer beiden inzidenten Knoten beschrieben. Ist die Kante gerichtet, dann ist dieses Paar geordnet. Zwei Knoten sind adjazent genau dann, wenn beide zu derselben Kante inzident sind.
>
> Der ***Grad*** eines Knotens ist die Anzahl seiner inzidenten Kanten.
>
> Ein Pfad ist eine Sequenz von Knotenpaaren, die jeweils eine Kante repräsentieren, so dass der erste Knoten jedes Paares mit dem zweiten Knoten des vorhergehenden Paares identisch ist.

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 such already exists
removeVertex(v)  // removes vertex v and all its incident edges
removeEdge(e)    // remove edge e
```

[Goodrich u. a. 2014]

Definieren wir nun einen abstrakten Datentyp für Graphen.

Die Methoden `vertices()` und `edges()` liefern iterierbare Container der Knoten und Kanten, und `numVertices()` und `numEdges()` liefern deren jeweilige Anzahl. `getEdge(u,v)` liefert die Kante von Knoten `u` nach Knoten `v`, sofern sie existiert, `endVertices(e)` liefert umgekehrt die beiden Endknoten der Kante `e`, und `opposite(v,e)` liefert den anderen Knoten, der zur zu Knoten `v` inzidenten Kante `e` inzident ist. (Schöner Satz, nicht wahr?)

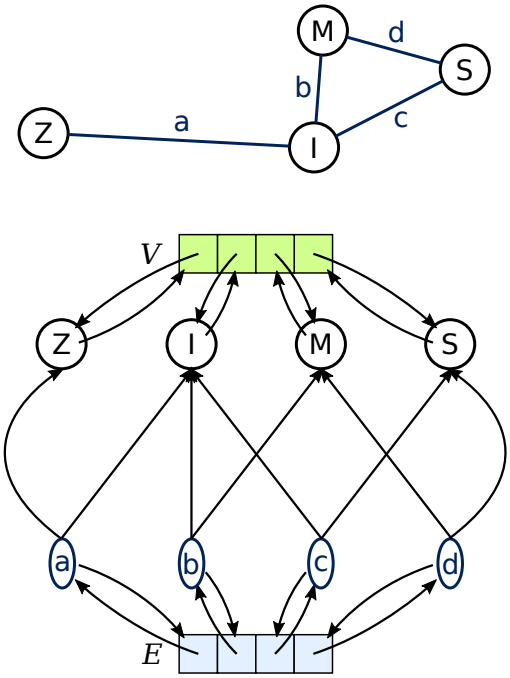
`outgoingEdges(v)` und `incomingEdges(v)` liefern iterierbare Container der zu Knoten `v` inzidenten ausgehenden bzw. eingehenden Kanten, und `outDegree(v)` und `inDegree(v)` liefern deren jeweilige Anzahl.

Dies waren die Abfragemethoden. Es folgen noch die Änderungsmethoden `insertVertex(x)` und `insertEdge(u,v,x)`, die einen neuen Knoten bzw. eine neue Kante einfügen, zusammen mit ihrem Datenelement `x`, sowie die Methoden `removeVertex()` und `removeEdge()`.

2 Datenstrukturen

Graphen sind der komplexeste abstrakte Datentyp, den wir in dieser Lehrveranstaltung behandeln, und entsprechend vielfältig und komplex sind die verwendeten Datenstrukturen. Wie immer hängt die beste Datenstruktur davon ab, welche Methoden des abstrakten Datentyps man effizient unterstützen möchte.

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.

Eine einfache Datenstruktur für einen Graphen ist die sogenannte Kantenliste. Wir sehen sie hier an einem Beispielgraphen illustriert.

Die Knoten des Graphen liegen in einer positionsbasierten Liste V vor, hier grün illustriert. Jede Position enthält einen Zeiger auf ihr Knotenobjekt, hier `Z`, `I`, `M` oder `S`. Jedes Knotenobjekt enhält seinerseits einen Zeiger zurück auf seine Position, sowie einen Zeiger auf sein Datenelement, hier symbolisiert durch die Markierungen `Z`, `I`, `M` und `S`.

Genauso ist die positionsbasierte Liste E für die Kanten organisiert, die hier blau gezeichnet ist. Zusätzlich enthält jedes Kantenobjekt zwei Zeiger auf ihre inzidenten Knotenobjekte.

Welche Methoden des abstrakten Datentyps werden durch diese Datenstruktur effizient unterstützt, und welche nicht?

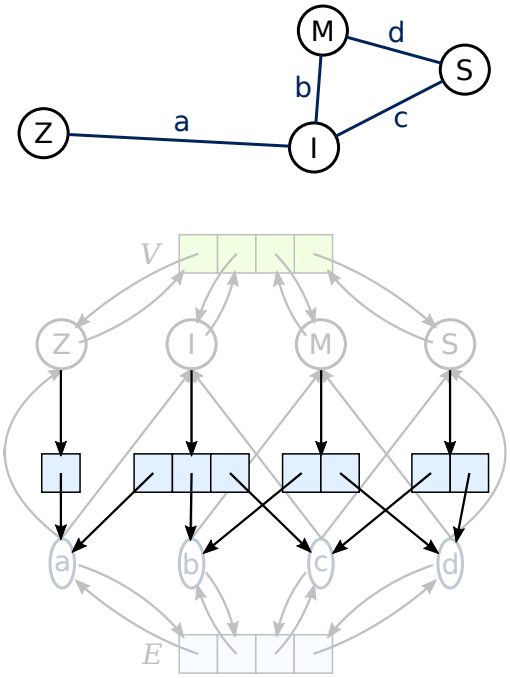
In der Kantenliste verfügen die Knoten nicht über Referenzen zu ihren inzidenten Kanten. Daher muss `getEdge(u,v)` die Liste der Knoten absuchen, um eine zu den Knoten `u` und `v` inzidente Kante zu finden. Ihre Laufzeit ist also $O(m)$ für einen Graphen mit insgesamt m Kanten. Das ist ineffizient, denn diese Laufzeit sollte allenfalls vom Grad der Knoten `u` und `v` abhängen.

Das Gleiche gilt für `outDegree()`, `inDegree()`, `outgoingEdges()` und `incomingEdges()`.

`endVertices()` und `opposite()` laufen dagegen in kostanter Zeit, da jede Kante Verweise auf ihre inzidenten Knoten besitzt.

Ebenso lassen sich `insertEdge()` und `removeEdge()` mit konstanter Laufzeit implementieren. `removeVertex(v)` dagegen hat wiederum eine Laufzeit von $O(m)$, da die zu Knoten `v` inzidenten Kanten in der Liste E gesucht werden müssen.

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.

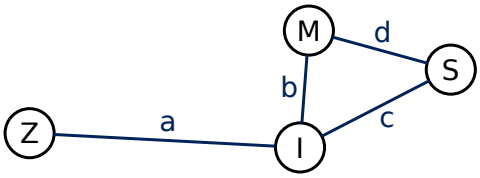
Die Datenstruktur Adjazenzliste erweitert die Kantenliste um Verweise von den Knoten auf ihre inzidenten Kanten. Hierzu erhält jedes Knotenobjekt v einen Zeiger auf eine sogenannte ***Inzidenzliste***, hier hellblau gezeichnet, die ihrerseits Zeiger auf die zu v inzidenten Kantenobjekte enthält, wie hier gezeigt. Handelt es sich um einen gerichteten Graphen, dann erhält jedes Knotenobjekt zwei Inzidenzlisten, je eine für ihre eingehenden und ausgehenden Kanten.

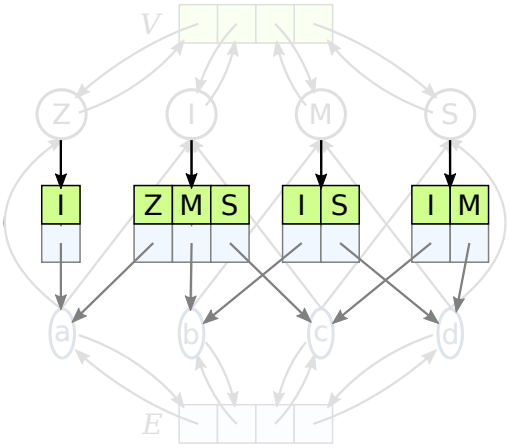
Dank dieser zusätzlichen Verweise beschleunigen sich einige Methoden. `getEdge(u,v)` muss nun nicht mehr die Liste E durchlaufen, sondern lediglich die kürzere der beiden mit `u` und `v` assoziierten Inzidenzlisten. Damit ist ihre Laufzeit Groß-O des Minimums der Grade dieser beiden Knoten. Ebenso lassen sich `outDegree()` und `inDegree()` leicht mit konstanter Laufzeit implementieren, und `outgoingEdges()` und `incomingEdges()` müssen lediglich den Inhalt der jeweiligen Liste verarbeiten.

Wir haben also die Laufzeiten derjenigen Methoden deutlich verbessert, die über Knoten auf ihre Kanten zugreifen müssen.

Die Methode `getEdge()` muss noch eine Inzidenzliste durchsuchen, lediglich um die gesuchte Kante zu finden, falls sie existiert. Zusätzlich muss nun auch die Methode `removeEdge()` zwei Inzidenzlisten durchsuchen, um die Referenzen auf die zu entfernende Kante zu löschen. Können wir diese lineare Suche vermeiden?

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





Wie die Adjazenzliste, außer:

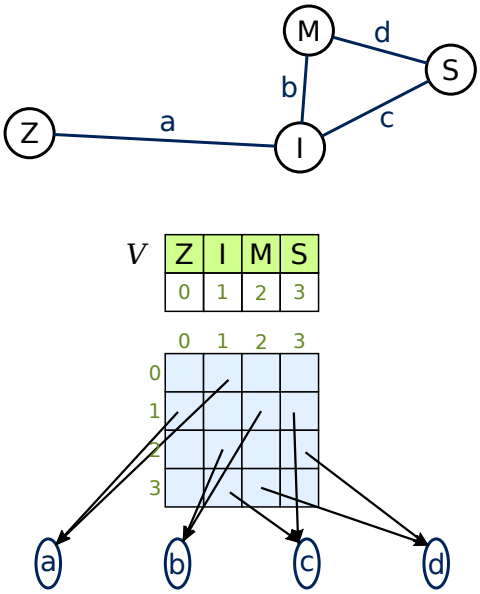
- Statt der *Liste* der inzidenten Kantenobjekte wird eine *Zuordnungstabelle* verwendet, in der für jede Kante der benachbarte Knoten als Schlüssel dient.

Anmerkung

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

> Wie finden wir denn effizient ein gesuchtes Objekt bzw. stellen seine Nicht-Existenz fest? Natürlich – mit einer Zuordnungstabelle!
>
> Um von einem Knoten aus die Kante zu einem anderen Knoten zu finden, nutzen wir also eine Zuordnungstabelle, mit dem anderen Knoten als Schlüssel. Wir indizieren also gewissermaßen die Liste der inzidenten Kanten mittels des über diese Kante adjazenten Knotens, wie hier illustriert.
>
> Damit verkürzt sich die erwartete Laufzeit sowohl von `getEdge()` als auch von `removeEdge()` auf $O(1)$.
>
> Damit ermöglicht uns die *Adjacency Map* optimale asymptotische Laufzeiten für alle Methoden des abstrakten Datentyps Graph. Die Laufzeiten sind entweder konstant oder proportional zur Anzahl der zu manipulierenden Elemente. Damit ist die *Adjacency Map* eine gute Allround-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*.

Zu guter Letzt führen wir noch die Adjazenzmatrix ein. Sie bietet gegenüber der *Adjacency Map* mehr Nachteile als Vorteile, aber in bestimmten Situationen überwiegen die Vorteile.

Die Adjazenzmatrix erweitert die Kantenliste um eine Matrix, die über die Knoten indiziert wird. Deren Elemente sind Zeiger auf die Kantenobjekte, die zu diesen Knoten inzident sind. Die Adjazenzmatrix eines ungerichteten Graphen ist immer symmetrisch, und die Adjazenzmatrix eines gerichteten Graphen ist allgemein unsymmetrisch.

Gegenüber der *Adjacency Map* hat die Adjazenzmatrix den Vorteil, dass sie erheblich einfacher zu implementieren ist, und die konstanten erwarteten Laufzeiten der Letzteren sind tatsächlich konstant im schlechtesten Fall.

Allerdings sind andere Laufzeiten dafür schlechter. Insbesondere sind das Einfügen und Entfernen von Knoten mit $\Theta(n^2)$ Laufzeit sehr kostspielig, da die Matrix um eine Zeile und eine Spalte vergrößert bzw. verkleinert werden muss.

Außerdem ist der Platzbedarf im Gegensatz zu den drei vorhergehenden Datenstrukturen $\Theta(n^2)$, unabhängig von der Anzahl der Kanten. Damit ist die Adjazenzmatrix vor allem für ***dichte*** Graphen interessant, also für Graphen, die, gemessen an der Anzahl ihrer Knoten, viele Kanten besitzen.

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.

> Viele Anwendungen erfordern das Besuchen aller Knoten eines Graphen über seine Kanten. Eine systematische Prozedur, die dies bewerkstelligt, heißt ***Traversierung***. Eine Traversierung gilt als effizient, falls sie alle Knoten und Kanten des Graphen in einer Zeit proportional zu ihrer Anzahl besucht.
>
> Traversierungen eignen sich insbesondere zur Lösung graphentheoretischer Probleme, die mit Pfaden und Erreichbarkeit zu tun haben. Solche Probleme treten in vielen praktischen Anwendungen auf.
>
> Zum Beispiel lässt sich bestimmen, ob ein gegebener Knoten v von einem gegebenen anderen Knoten u aus *erreichbar* ist, und falls ja, über welchen Pfad. Verallgemeinert lassen sich die ***Zusammenhangskomponenten*** eines Graphen bestimmen. Ein Graph ist ***zusammenhängend***, falls zwischen jeden zwei beliebigen seiner Knoten ein Pfad existiert. Ein Graph ist ***stark zusammenhängend***, falls ein solcher Pfad in beiden Richtungen existiert. Dies ist insbesondere bei gerichteten Graphen relevant.
>
> Eine ***Zusammenhangskomponente*** eines Graphen ist ein zusammenhängender Subgraph, der nicht weiter vergrößert werden kann, ohne dass er seine zusammenhängende Eigenschaft verliert.
>
> Ein ***Spannbaum*** eines Graphen entsteht, wenn man aus einem zusammenhängenden Graphen so viele Kanten wie möglich entfernt, ohne dass er seine zusammenhängende Eigenschaft verliert.
>
> Darüber hinaus eignen sich Traversierungsmethoden dazu, Zykel zu finden bzw. nachzuweisen, dass ein Graph frei von Zykeln ist.

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)
```

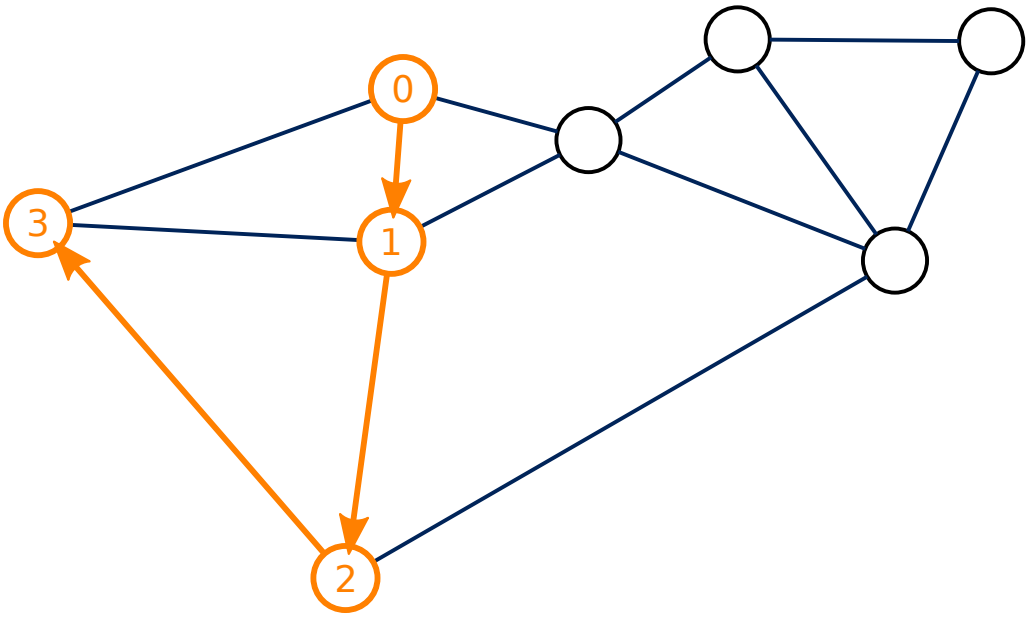
Analog zu Theseus:

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

Vgl. BFS

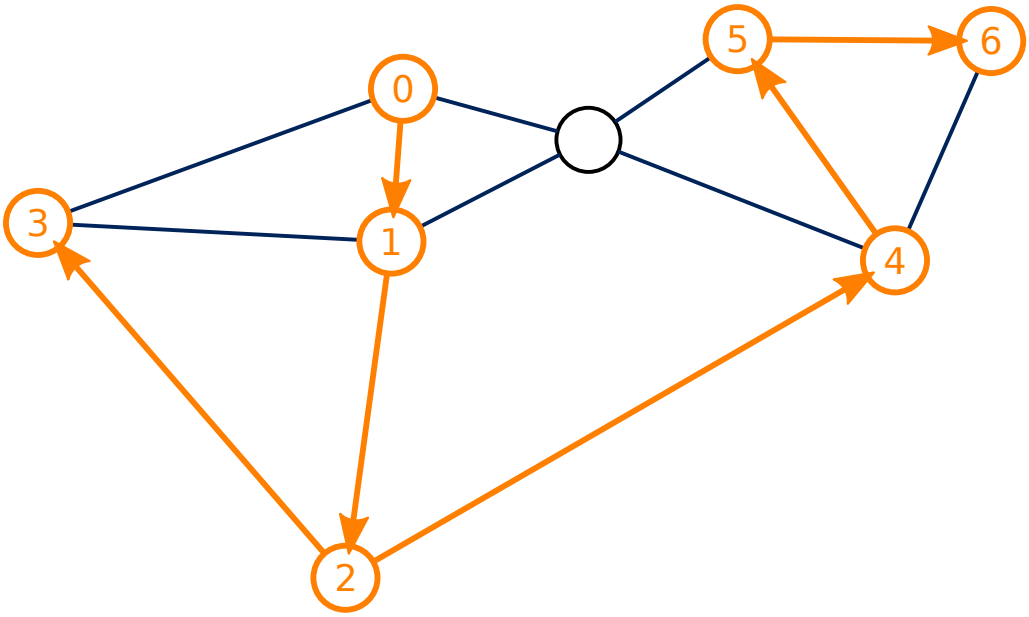
> Unser erster Traversierungsalgorithmus ist die ***Tiefentraversierung***, englisch *depth-first traversal*, auch als Tiefensuche bzw. *depth-first search* bekannt und daher oft DFS abgekürzt.
>
> Die zentrale Idee des Tiefentraversierungsalgorithmus ist, jeden besuchten Knoten zu markieren, damit wir ihn nicht noch einmal besuchen. Wir markieren also als erstes unseren Startknoten u. Anschließend iterieren wir über alle von u ausgehenden Kanten (wobei auch ungerichtete Kanten als ausgehend gelten). Falls der über diese Kante adjazente Knoten noch nicht als besucht markiert ist, rufen wir uns rekursiv auf diesem Knoten auf.
>
> Ist die `foreach`-Schleife beendet, dann kehrt der aktuelle Aufruf zurück. Damit wird die Tiefentraversierung beim Vorgängerknoten des aktuellen Knotens im Tiefensuchbaum in der nächsten Iteration *seiner* `foreach`-Schleife fortgesetzt.
>
> Dabei führen wir Buch über die Kanten, über die wir neue, unbesuchte Knoten erschließen. Jeder der besuchten Knoten, außer dem allerersten, besitzt genau eine solche sogenannte ***Baumkante***. Die bei der Traversierung besuchten Knoten bilden zusammen mit den Baumkanten den sogenannten ***Tiefensuchbaum***.

Tiefentraversierung eines ungerichteten Graphen (1) [Slide 19]



Sehen wir nun eine Tiefentraversierung unseres ungerichteten Eisenbahn-Netzwerks. Wir beginnen in München, und markieren es orange als besucht. Beim Iterieren über die zu München inzidenten Kanten erwischen wir zuerst die Kante nach Innsbruck. Innsbruck ist noch nicht als besucht markiert, also fahren wir hin und markieren es. Auf dieselbe Weise geht es weiter nach Verona und anschließend nach Zürich. Beim Iterieren über die zu Zürich inzidenten Kanten stellen wir fest, dass alle drei gegenüberliegenden Knoten bereits markiert sind, nämlich München, Innsbruck und Verona. Hier endet die `forach`-Schleife in Zürich, der Zürcher rekursive Aufruf kehrt zurück, und Verona setzt seine Iteration über seine inzidenten Kanten fort.

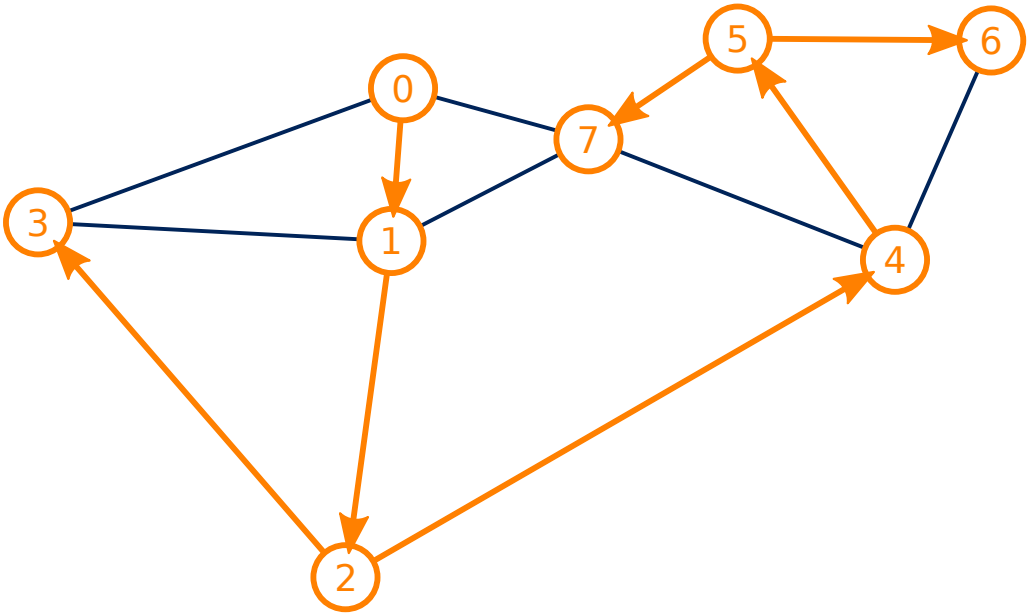
Tiefentraversierung eines ungerichteten Graphen (2) [Slide 20]



Innsbruck ist bereits als besucht markiert, aber Graz noch nicht. Also geht es von Verona nach Graz, und anschließend auf dieselbe Weise nach Linz und Wien. In Wien sind

wiederum die allen inzidenten Kanten gegenüberliegenden Knoten bereits als besucht markiert, weshalb der Wiener Aufruf zurückkehrt.

Tiefentraversierung eines ungerichteten Graphen (3) [Slide 21]



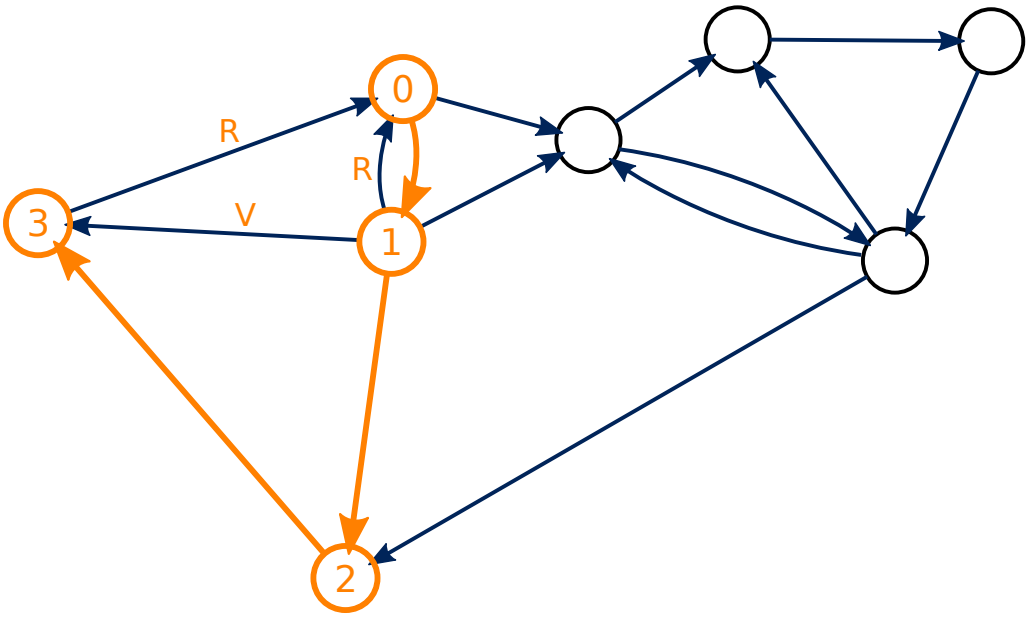
Wiens Vorgänger, Linz, hat jedoch noch eine Kante mit einem nicht markierten, gegenüber liegenden Knoten, nämlich Salzburg. Also geht es weiter nach Salzburg. In Salzburg sind wiederum alle vier gegenüber liegenden Knoten bereits als besucht markiert, und sein rekursiver Aufruf kehrt zurück.

Ebenso kehren alle verbleibenden rekursiven Aufrufe zurück, denn alle Knoten des Graphen sind bereits markiert. Damit ist die Tiefentraversierung beendet.

Wir sehen, dass die orange markierten Baumkanten tatsächlich einen Baum bilden, da jeder besuchte Knoten genau eine eingehende Baumkante besitzt, bis auf den Startknoten, den wir als Wurzel des Tiefensuchbaums betrachten.

Alle nicht markierten Kanten sind hier sogenannte ***Rückwärtskanten***, denn jede dieser Kanten führt, wenn sie das erste Mal betrachtet wird, zurück zu einem Vorfahren im Tiefensuchbaum.

Tiefentraversierung eines gerichteten Graphen (1) [Slide 22]

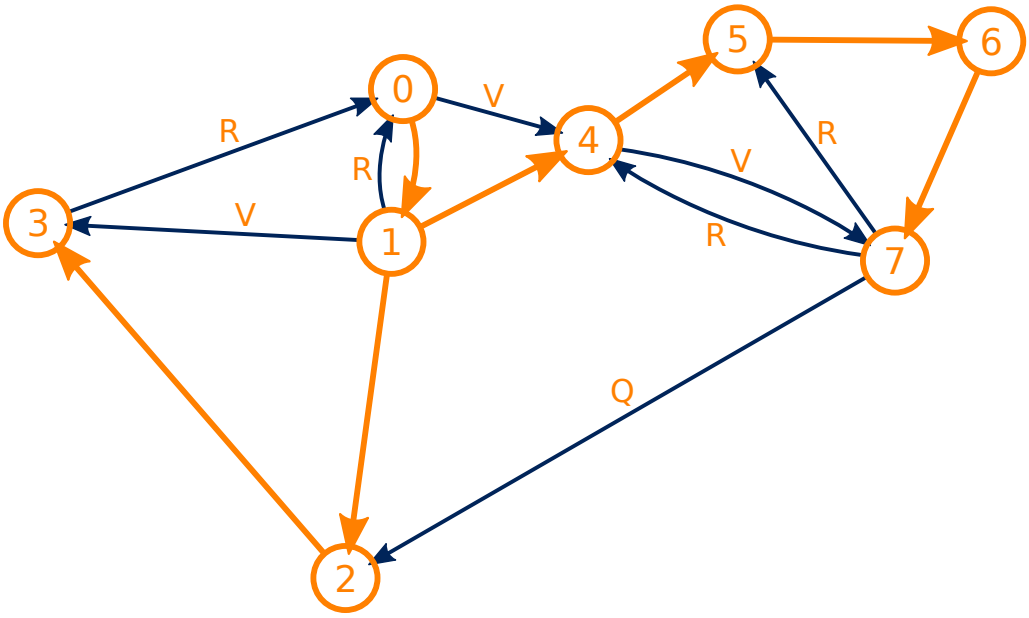


Sehen wir nun eine Tiefentraversierung eines gerichteten Graphen. Wir beginnen wieder in München und fahren entlang der gerichteten Kanten über Innsbruck und Verona nach Zürich. Dort finden wir nur eine einzige ausgehende Kante, nämlich die nach München. München ist ein Vorfahr Zürichs im Tiefensuchbaum; daher handelt es sich wieder um eine Rückwärtskante und wird hier mit `R` markiert.

Vom rekursiven Aufruf zurückgekehrt nach Verona, sind wir dort bereits am Ende der Iteration über die ausgehenden Kanten, und kehren ebenfalls zurück.

In Innsbruck hat unser Iterator über die ausgehenden Kanten noch drei weitere Kanten parat. Die erste, nach Zürich, führt zu einem bereits besuchten Knoten. Dieser ist ein Nachfolger Innsbrucks im Tiefensuchbaum; daher markieren wir diese sogenannte ***Vorwärtskante*** mit `V`. Die nächste Kante führt zurück nach München und ist somit eine Rückwärtskante.

Tiefentraversierung eines gerichteten Graphen (2) [Slide 23]



Die dritte und letzte von Innsbruck ausgehende Kante führt zu einem noch nicht besuchten Knoten, und ist also wieder eine Baumkante. Nun geht es weiter nach Salzburg, Linz, Wien und Graz.

In Graz stoßen wir auf drei ausgehende Kanten, aber alle führen zu bereits besuchten Knoten. Zwei davon, Salzburg und Linz, sind Vorfahren im Tiefensuchbaum; daher sind diese Kanten Rückwärtskanten.

Die dritte von Graz ausgehende Kante führt nach Verona, und Verona ist weder ein Vorfahr noch ein Nachkomme von Graz im Tiefensuchbaum. Daher handelt es sich bei dieser Kante um eine *Querkante*, denn sie verbindet zwei verschiedene Äste unseres Tiefensuchbaums.

Bei der Rückkehr von der Rekursion treffen wir nur noch in Salzburg und in München auf weitere ausgehende Kanten. In beiden Fällen handelt es sich um Vorwärtskanten.

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!

> Fassen wir hier noch einmal die verschiedenen Kantenkategorien zusammen, in die die Kanten eines Graphen bei seiner Tiefentraversierung eingeteilt werden.
>
> Alle Kanten, die einen neuen Knoten besuchen, sind Baumkanten.
>
> In einem ungerichteten Graphen sind alle anderen Kanten Rückwärtskanten.
>
> In einem gerichteten Graphen treten neben Rückwärtskanten auch Vorwärts- und Querkanten auf.

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 $O(n + m)$.

> Die Laufzeit der Tiefentraversierung ist einfach zu bestimmen. Der Algorithmus stellt sicher, dass jeder Knoten lediglich einmal besucht wird. Damit wird auch jede Kante höchstens zweimal betrachtet.
>
> Die beiden entscheidenden Operationen sind die Iteration über die ausgehenden Kanten mittels `outgoingEdges()`, und die Bestimmung des über eine gegebene Kante adjazenten Knotens. Letztere Methode lässt sich mit allen Datenstrukturen, die wir besprochen haben, in konstanter Zeit implementieren, da jedes Kantenobjekt auf seine beiden inzidenten Knoten verweist. Die Methode `outgoingEdges()` lässt sich mittels der Adjazenzliste und demnach auch der *Adjacency Map* in einer Zeit proportional zur Anzahl der zurückgelieferten Kanten implementieren. Die Kantenliste und die Adjazenzmatrix lassen dies jedoch nicht zu, da die von einem Knoten ausgehenden Kanten aus der gesamten Liste der Kanten zusammengesucht werden müssen.
>
> Folglich lässt sich die Tiefentraversierung mittels der Adjazenzliste und der *Adjacency Map* in $O(n + m)$ implementieren, nicht jedoch mittels der Kantenliste oder der Adjazenzmatrix.
>
> Wir setzen natürlich auch voraus, dass wir Markierungen in konstanter Zeit anbringen und abfragen können. Dies ist z.B. mit einer Hash-Tabelle einfach realisierbar.

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:

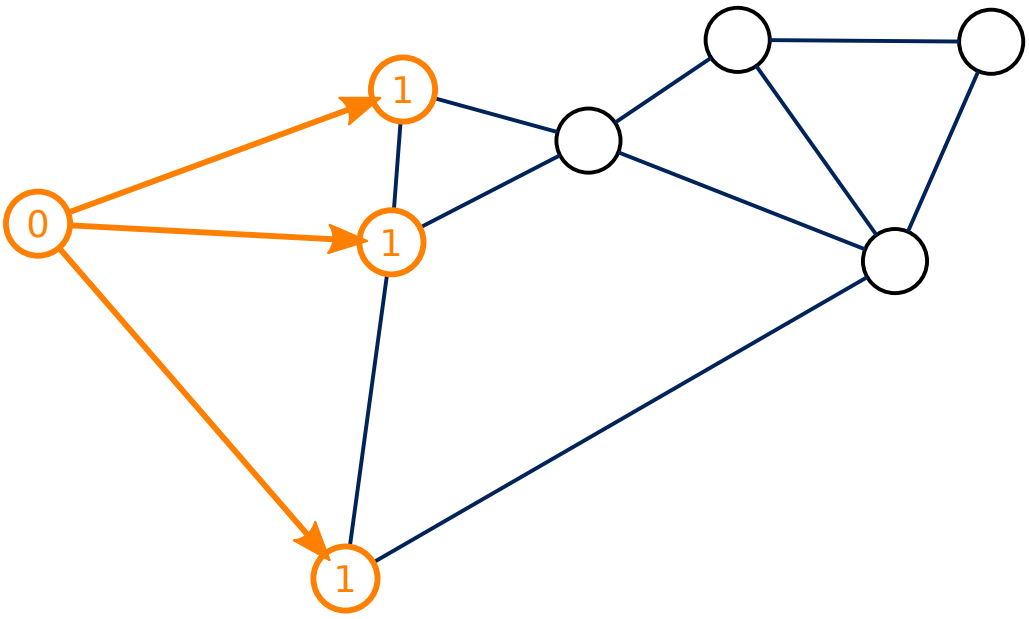
- ***Rückwärtskanten***, die zu einem Vorfahren im Breitensuchbaum führen
- ***Querkanten***

Wichtig
Ein Breitensuchbaum ist kein Suchbaum!

> Die Tiefentraversierung entspricht einem einsamen Helden, der ein Labyrinth systematisch absucht, indem er den jeweils aktuellen Pfad bis zum Ende verfolgt.
>
> Die Breitentraversierung entspricht dagegen einem koordinierten Suchtrupp, der sich an jeder Verzweigung aufteilt und in geschlossener Front vorrückt.
>
> Ebenso wie die Tiefentraversierung teilt die Breitentraversierung die Kanten in Baumkanten und Nichtbaumkanten ein, wobei die Baumkanten den sogenannten Breitensuchbaum bilden.

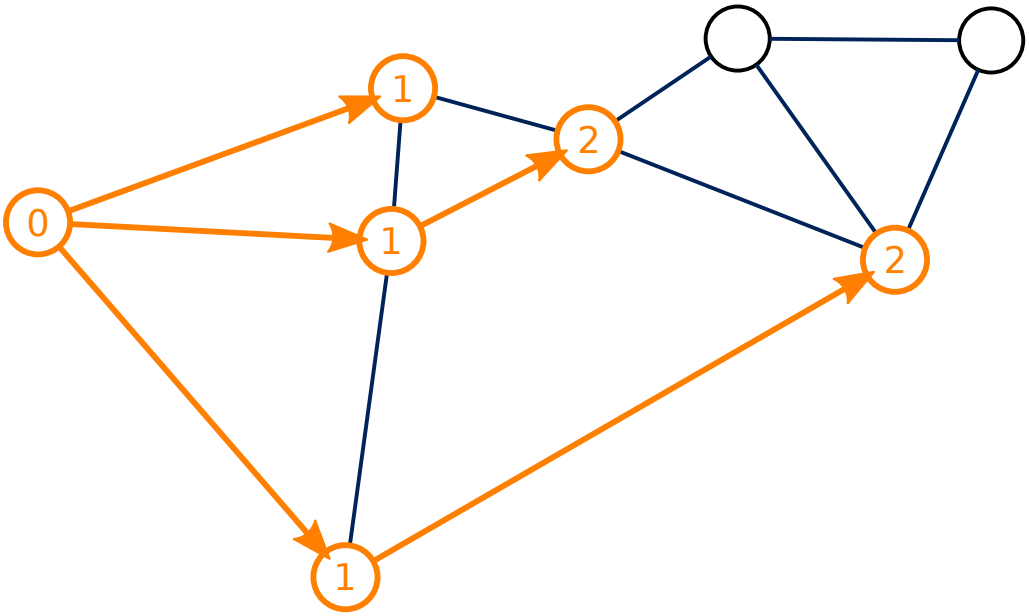
In einem ungerichteten Graphen sind alle Nichtbaumkanten Querkanten, da der Suchtrupp synchron vorrückt. Aus demselben Grund können bei gerichteten Graphen keine Vorwärtskanten auftreten, sondern lediglich Rückwärts- und Querkanten.

Breitentraversierung eines ungerichteten Graphen (1) [Slide 29]



Führen wir nun eine Breitentraversierung unseres ungerichteten Graphen durch. Wir beginnen in Zürich und erreichen von dort München, Innsbruck und Verona.

Breitentraversierung eines ungerichteten Graphen (2) [Slide 30]

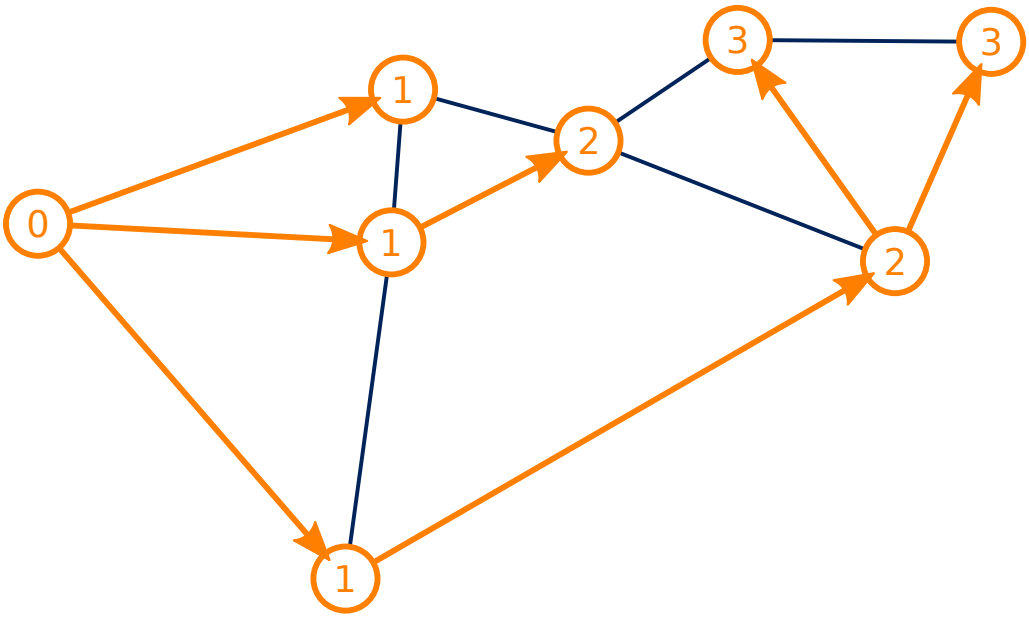


Im nächsten Schritt werden die Verbindungen zwischen Innsbruck und München und zwischen Innsbruck und Verona nicht beschritten, da der jeweils gegenüberliegende Knoten bereits als besucht markiert ist. Diese Verbindungen sind also Querkanten.

München, Innsbruck und Verona besitzen jeweils eine weitere ausgehende Kante, die nun nacheinander traversiert werden. Hier geht es z.B. zuerst von Verona nach Graz und dann von Innsbruck nach Salzburg.

Wenn München an der Reihe ist, ist Salzburg bereits als besucht markiert. Diese Verbindung ist also ebenfalls eine Querkante, da sie zwei Knoten verbindet, die im Breitensuchbaum nicht Vorfahr und Nachkomme voneinander sind. München verfügt über keine weitere ausgehende Kante, und damit ist München im Breitensuchbaum ein Blatt.

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.

Weiter geht es nun mit Graz. Graz hat vier ausgehende Kanten, von denen zwei zu bereits besuchten Knoten führen, nämlich Verona und Salzburg. Die beiden anderen, Linz und Wien, werden nun samt ihrer Baumkanten markiert.

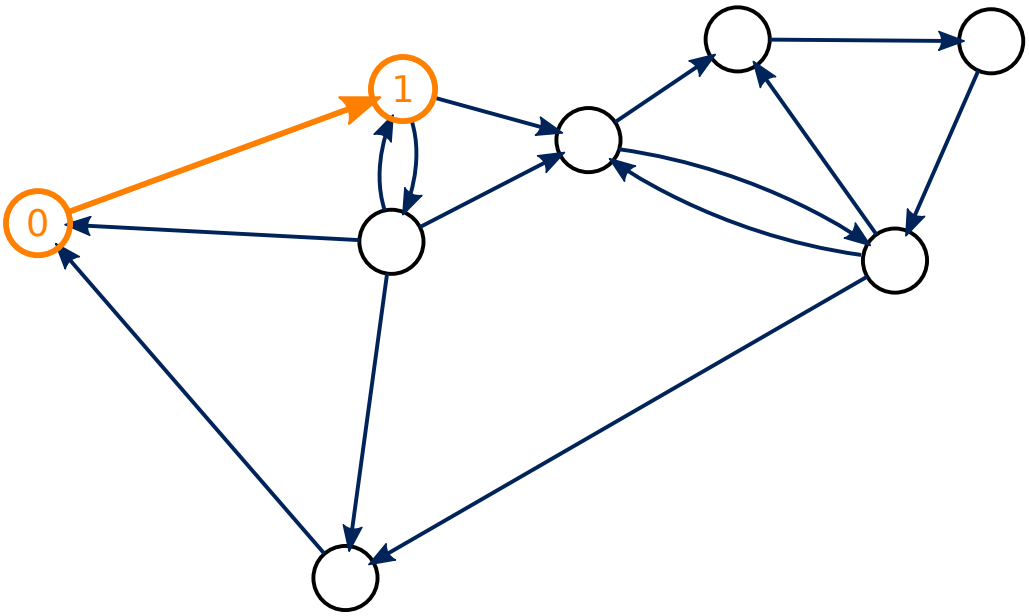
Als nächstes ist Salzburg an der Reihe, aber sämtliche zu Salzburg inzidenten Kanten führen zu bereits besuchten Knoten.

Im nächsten Schritt geht es bei Linz und Wien weiter, aber von ihnen führen ebenfalls keine Kanten zu noch nicht besuchten Knoten.

Die Zahlen in den Knoten markieren hier die Zeitschritte, in denen unser sich aufteilender Suchtrupp vorrückt. Im ersten Zeitschritt werden alle Nachbarn des Ausgangsknotens erreicht und mit 1 markiert, im zweiten Zeitschritt werden alle noch nicht besuchten Nachbarn der mit 1 markierten Knoten erreicht und mit 2 markiert, und so weiter.

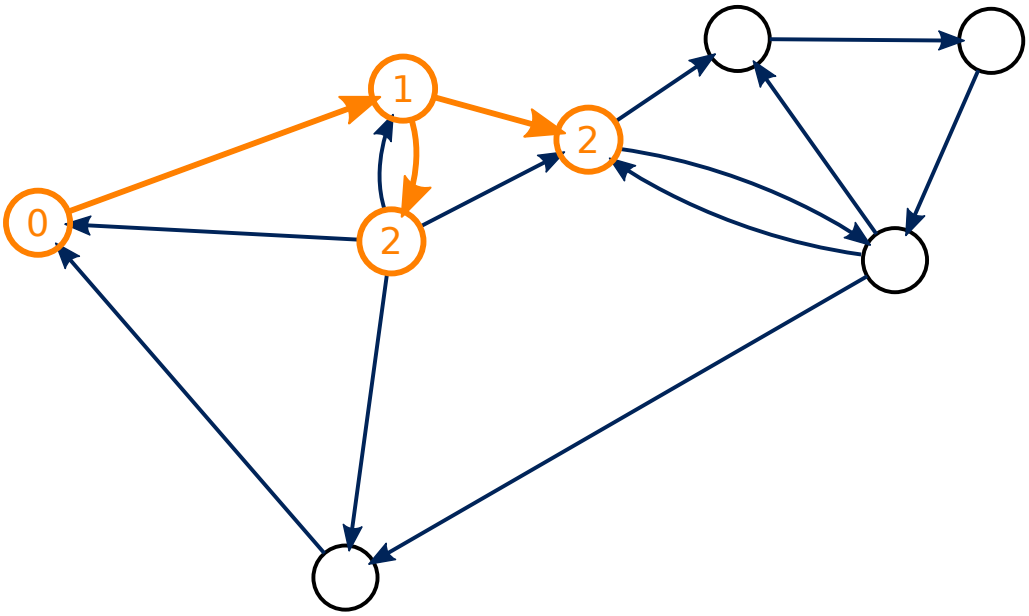
Hierdurch wird der Graph in sogenannte *Niveaus* eingeteilt. Alle mit der Zahl k markierten Knoten bilden das Niveau k des Graphen bei einer Breitentraversierung ausgehend von Zürich.

Breitentraversierung eines gerichteten Graphen (1) [Slide 32]



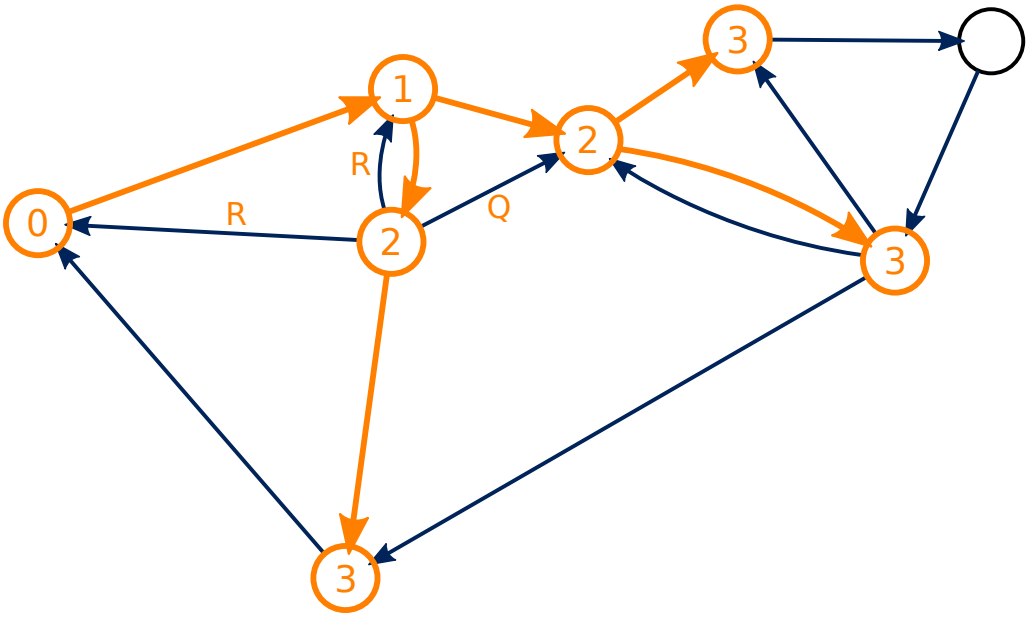
Sehen wir nun eine Breitentraversierung unseres *gerichteten* Graphen, wiederum ausgehend von Zürich. Zu Beginn folgen wir der einzigen ausgehenden Kante nach München. Damit umfasst das Niveau 1 lediglich München als einzigen Knoten.

Breitentraversierung eines gerichteten Graphen (2) [Slide 33]



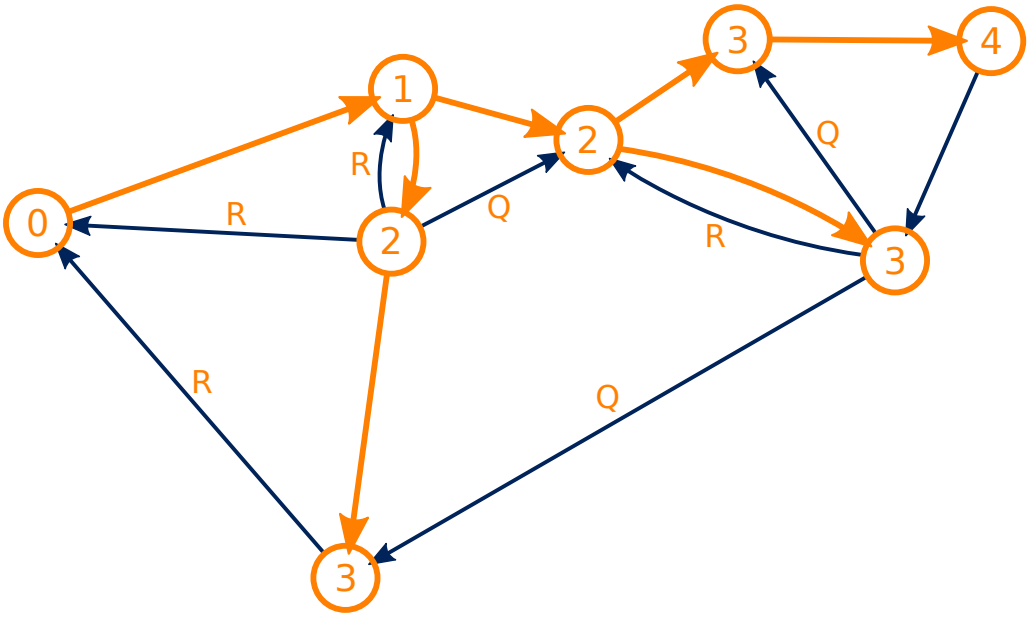
Im nächsten Schritt folgen wir den beiden von München ausgehenden Kanten nach Innsbruck und Salzburg.

Breitentraversierung eines gerichteten Graphen (3) [Slide 34]



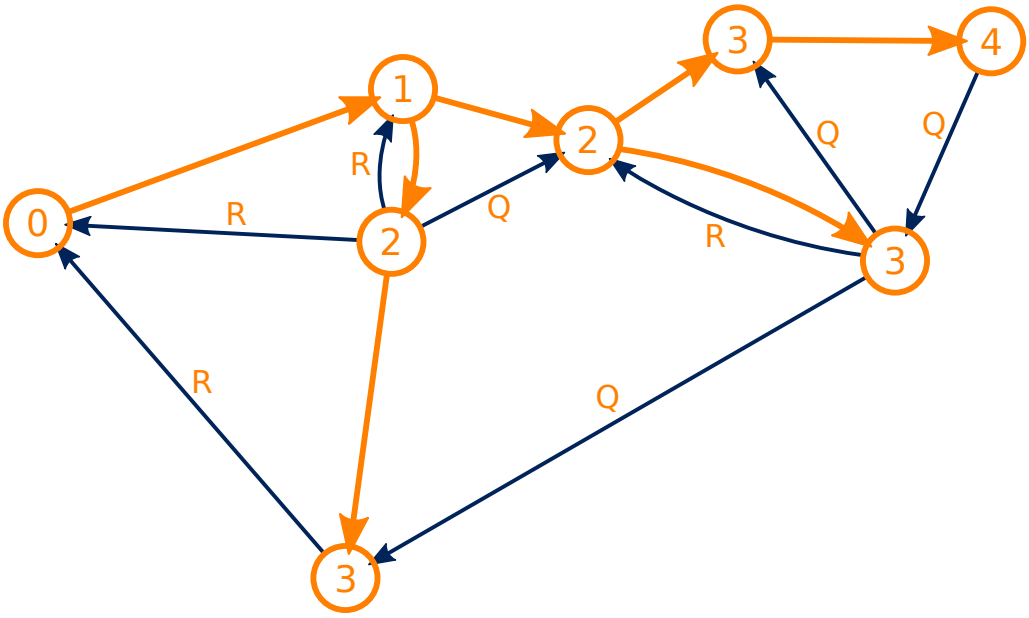
Von Innsbruck gehen 4 Kanten aus, eine Baumkante nach Verona, je eine Rückwärtskante nach Zürich und nach München, und eine Querkante nach Salzburg. Von Salzburg gehen zwei Baumkanten aus, nämlich nach Linz und nach Graz. Niveau 3 unseres Breitensuchbaums umfasst also Linz, Graz und Verona.

Breitentraversierung eines gerichteten Graphen (4) [Slide 35]



Im 4. Zeitschritt explorieren wir alle Kanten, die von Knoten des 3. Niveaus ausgehen. Die einzige von Verona ausgehende Kante ist eine Rückwärtskante nach Zürich. Von Graz geht eine Rückwärtskante nach Salzburg aus, sowie je eine Querkante nach Verona und nach Linz. Von Linz geht eine einzige Kante aus, nämlich eine Baumkante nach Wien.

Breitentraversierung eines gerichteten Graphen (5) [Slide 36]



> Das Niveau 4 unseres Breitensuchbaums umfasst lediglich Wien, und von Wien geht nur eine einzige Kante aus, eine Querkante hinüber nach Graz. Damit ist unsere Breitentraversierung beendet.

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

> Dies ist der Breitentraversierungsalgorithmus. Er funktioniert analog zur Breitentraversierung eines Baums nicht rekursiv, sondern nutzt eine Warteschlange. Da die neu entdeckten Knoten immer hinten in die Warteschlange eingereiht werden, werden zuerst die Knoten des aktuellen Niveaus vorn aus der Warteschlange herausgeholt und abgearbeitet, bevor deren neu entdeckte Nachbarn an die Reihe kommen. Daher werden die Knoten Niveau für Niveau besucht, auch wenn diese Niveaus im Algorithmus nicht explizit auftauchen.

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?

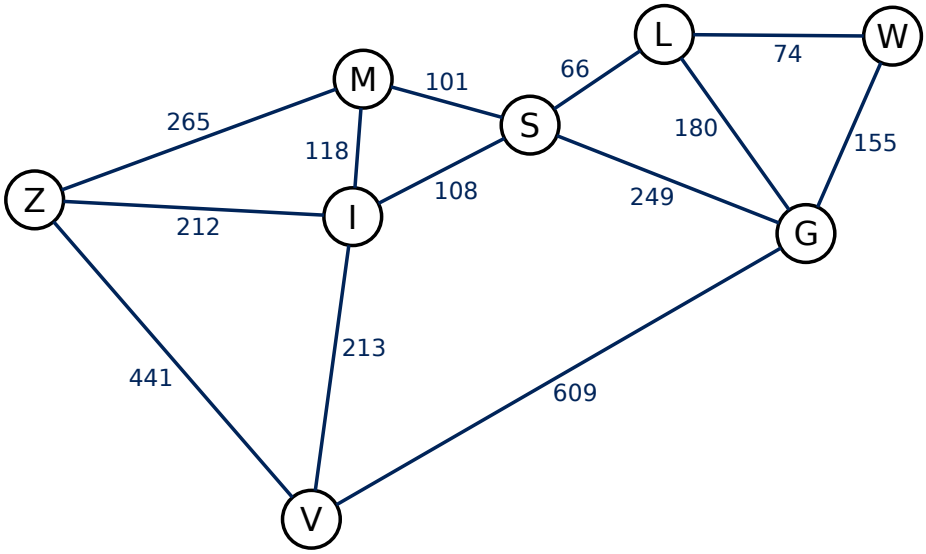
  identisch zur Tiefentraversierung

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

> Bei einem ***kantenbewerteten*** Graphen, englisch *weighted graph*, sind die Kanten mit skalaren Werten belegt. In diesem Beispiel sind diese Werte die Fahrtzeiten in Minuten zwischen den jeweiligen Städten.
>
> Damit können wir nach ***kürzesten Pfaden*** fragen, die die Summe der Kantenwerte eines Pfades minimieren. In diesem (gegenüber der Realität leicht verzerrten) Beispiel ist der kürzeste Pfad von Salzburg nach Graz nicht der direkte Weg mit 249 Minuten, sondern der Umweg über Linz mit 66+180=246 Minuten.
>
> Sind alle Kanten gleich oder gar nicht bewertet, dann bemisst sich die Pfadlänge nach der Anzahl der traversierten Kanten. Diese entsprechen genau den Niveaus einer Breitentraversierung. Um kürzeste Pfade in kantenbewerteten Graphen zu finden, müssen wir uns jedoch etwas mehr ins Zeug legen.

Kürzeste Pfade in kantenbewerteten Graphen [Slide 41]

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

Führen wir zunächst etwas Notation ein. Eine gerichtete Kante e ist ein geordnetes Paar seiner beiden inzidenten Knoten (u, v). Eine ungerichtete Kante ist ein ungeordnetes Paar.

Den Wert einer Kante $e = (u, v)$ bezeichnen wir mit $w(e)$, oder, leicht verkürzt, als $w(u, v)$.

Ein Pfad ist eine Sequenz $((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k))$ von Knotenpaaren, die jeweils eine Kante repräsentieren, so dass der erste Knoten jedes Paares mit dem zweiten Knoten des vorhergehenden Paares identisch ist. Der Wert eines Pfades ist die Summe der Werte seiner Kanten.

Den Wert eines kürzesten Pfades zwischen zwei Knoten u und v bezeichnen wir als $d(u, v)$. Falls kein solcher Pfad existiert, ist dieser Wert unendlich.

Wenn wir negative Kantenwerte zulassen, dann kann ein Graph Zykel negativen Wertes enthalten. Ist ein solcher Zykel von zwei Knoten u und v erreichbar, dann gibt es zwischen u und v keinen kürzesten Pfad, denn man kann den Wert eines solchen Pfades durch wiederholtes Verfolgen dieses Zykels beliebig verkleinern.

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 $\underset{u \notin C}{\operatorname{argmin}}\{D[u]\}$ und füge ihn 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**
    $D[v] \leftarrow D[u] + w(u, v)$

> Der klassische Algorithmus zum Finden kürzester Pfade von einem designierten Knoten zu allen anderen Knoten eines Graphen wurde bereits 1956 vom niederländischen Informatik-Pionier Edsger Dijkstra erdacht.
>
> Dijkstras Algorithmus teilt die Knoten des Graphen in zwei Mengen auf. Diejenigen Knoten, zu denen die Länge des kürzesten Pfades vom Startknoten s definitiv bekannt ist, holt er sich in die Menge C, oft als Wolke bezeichnet, und alle anderen Knoten befinden sich außerhalb von C.
>
> Für jeden Knoten v bezeichnet $D[v]$ die Länge des aktuell bekannten kürzesten Pfades von s nach v. Diese oberen Schranken der tatsächlich kürzesten Pfadlängen werden iterativ verkleinert.
>
> Die zentrale Idee von Dijkstras Algorithmus besteht darin, dass der *minimale* Wert $D[u]$ über alle Knoten u außerhalb von C seine Pfadlänge von s nicht überschätzt. Damit können wir diesen Knoten u als nächstes in die Wolke ziehen. Danach müssen wir dann sicherstellen, dass der minimale Wert $D[u]$ über alle Knoten u *nun* außerhalb der Wolke seine Pfadlänge nicht überschätzt.
>
> Dies wird folgendermaßen erreicht:
>
> Zu Beginn ist $D[s] = 0$, für alle anderen Knoten v ist $D[v] =$ unendlich, und C ist leer.
>
> Nun wird der Knoten u außerhalb von C mit minimalem $D[u]$ in die Wolke gezogen. Damit kennen wir eine obere Schranke der Pfadlänge für die Nachbarn v von u außerhalb der Wolke: Ein solcher Pfad kann nicht länger sein, als die Pfadlänge von s nach u plus der Wert der Kante von u nach v, also $D[u] + w(u, v)$. Andererseits ist nicht garantiert, dass dieser Pfad über u tatsächlich der kürzeste von s nach v ist; es könnte sein, dass der kürzeste Pfad nicht über u, sondern über einen anderen letzten Knoten in C führt. Daher setzen wir $D[v]$ auf das Minimum von $D[u] + w(u, v)$ und der aktuell besten oberen Schranke $D[v]$.
>
> Diese schrittweise Reduktion von $D[v]$ wird als ***Edge Relaxation*** bezeichnet.

Dijkstras Algorithmus [Slide 43]

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

> Nun trennen uns nur noch einige kleine Details vom vollständigen Algorithmus.
>
> Wie besprochen, setzen wir $D[s] = 0$ und $D[v] = \infty$ für alle anderen Knoten.
>
> Um jederzeit effizient den niedrigsten Wert $D[u]$ über alle Knoten u außerhalb von C zu finden, verwenden wir eine Vorrangwarteschlange Q mit den $D[v]$ als Schlüssel und v als Wert. Diese Vorrangwarteschlange ist die Komplementärmenge von C. Jeder Knoten, den wir aus der Vorrangwarteschlange holen, landet damit in der Wolke C.
>
> Nun iterieren wir, wie vorhin beschrieben: Wir holen uns den Knoten u mit minimalem Wert $D[u]$ aus der Vorrangwarteschlange, und verringern ggf. die Werte $D[v]$ aller seiner Nachbarn v außerhalb von C.
>
> Hier müssen wir noch eine Feinheit beachten: Wir ändern hier den Wert des Schlüssels eines Elements, während dieses sich in der Vorrangwarteschlange befindet. Damit ändert sich möglicherweise seine Priorität, und der Zustand der Vorrangwarteschlange muss entsprechend angepasst werden. Bei einem Heap bedeutet dies, dass die Heap-Ordnung wieder hergestellt werden muss, was sich mit einer entsprechenden Datenstruktur leicht in logarithmischer Zeit bewerkstelligen lässt. Diese Möglichkeit der Anpassung formalisiert der abstrakte Datentyp ***Anpassbare Vorrangwarteschlange***, auf englisch *Adaptable Priority Queue*.

Quiz [Slide 44]

Dijkstras Algorithmus, wie beschrieben, eignet sich …

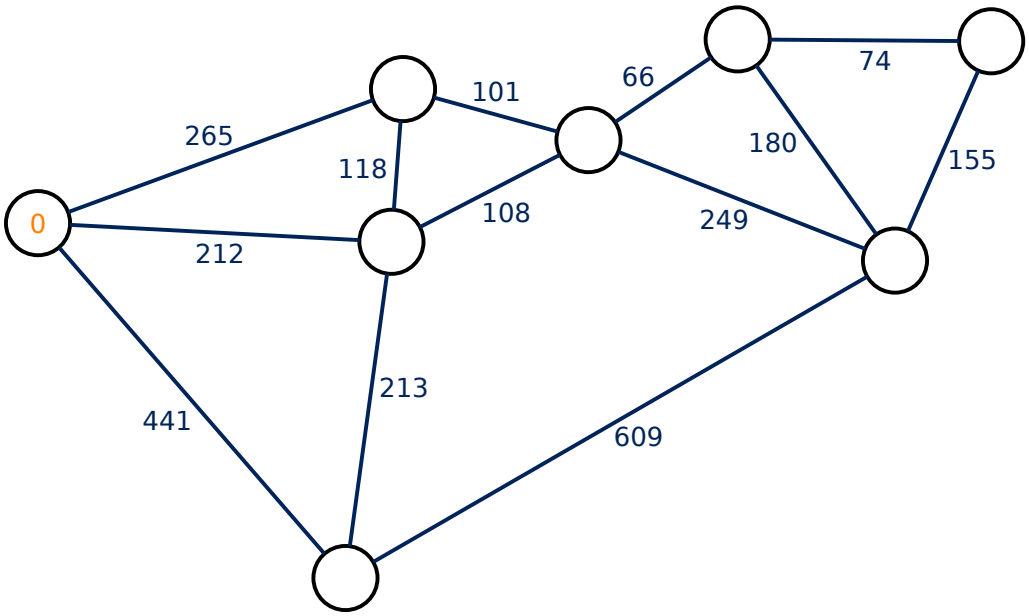
A: nur für gerichtete Graphen

B: nur für ungerichtete Graphen

C: für gerichtete und ungerichtete Graphen gleichermaßen

D: weiß nicht

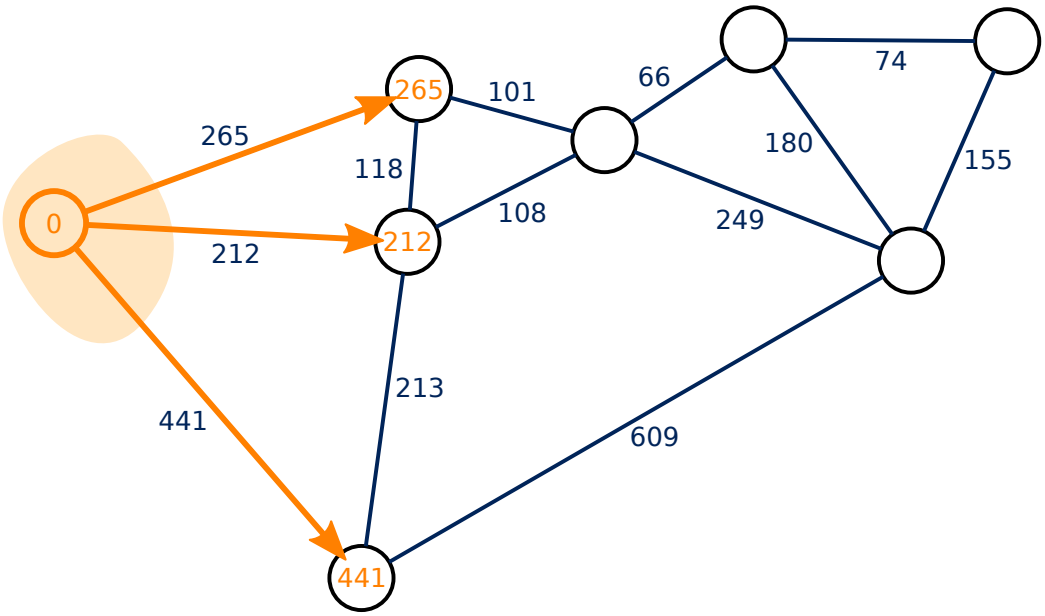
Beispielablauf von Dijkstras Algorithmus (0) [Slide 45]



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

> Beobachten wir nun Dijkstras Algorithmus in Aktion. Der Startknoten s ist Zürich. Jeder Knoten v enthält in oranger Schrift seinen Wert $D[v]$; bei leeren Knoten ist dieser unendlich.
>
> Die Vorrangwarteschlange enthält zu diesem Zeitpunkt sämtliche Knoten des Graphen.

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.

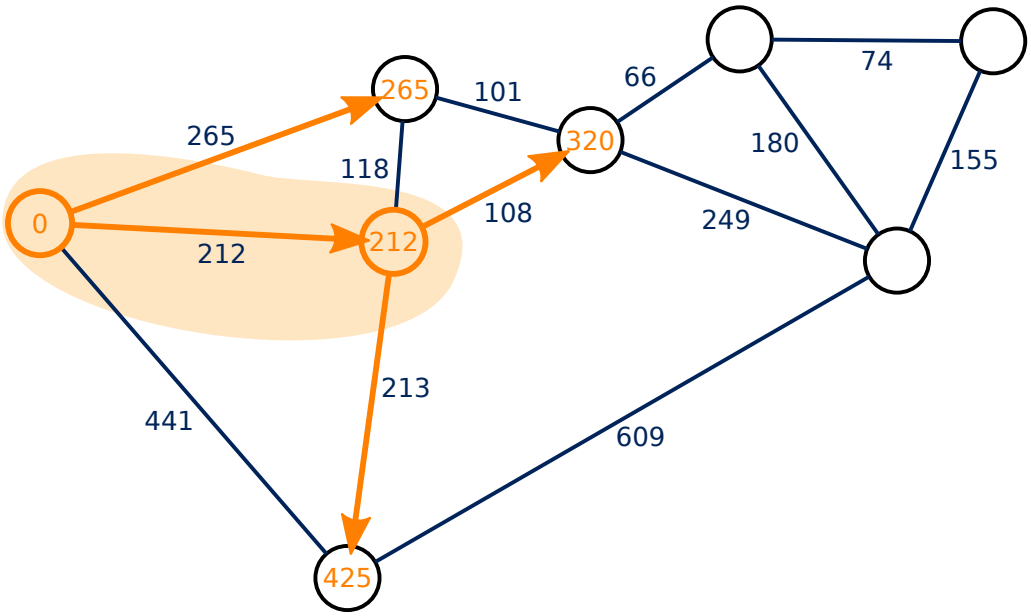
In der ersten Iteration ziehen wir den Startknoten mit $D[s] = 0$ aus der Vorrangwarteschlange. Damit ist er der bisher einzige Knoten in der Wolke C, die hier orange hinterlegt ist.

Nun betrachten wir alle Nachbarn v außerhalb der Wolke dieses soeben in die Wolke gezogenenen Knotens u, und verkleinern ihren Wert $D[v]$ auf $D[u] + w(u, v)$. Da $u = s$ und $D[s] = 0$ ist, ist dieser Wert hier jeweils gleich dem Kantenwert $w(u, v)$.

Damit ist die erste Iteration beendet.

Die kürzeste Verbindung aus der Wolke C zu jedem Nachbarknoten außerhalb der Wolke ist als fetter, oranger Pfeil dargestellt.

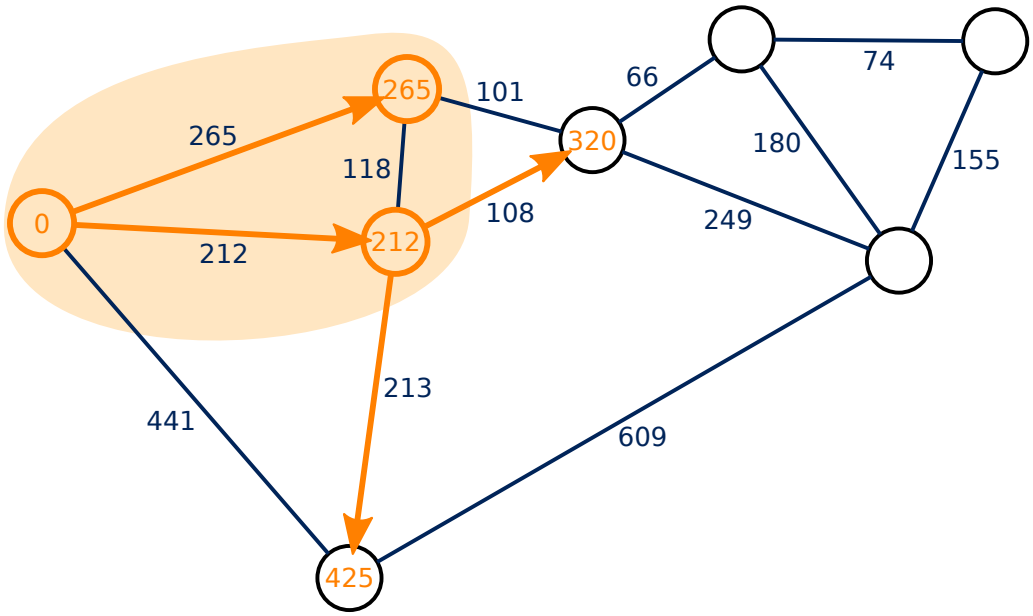
Beispielablauf von Dijkstras Algorithmus (2) [Slide 47]



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.

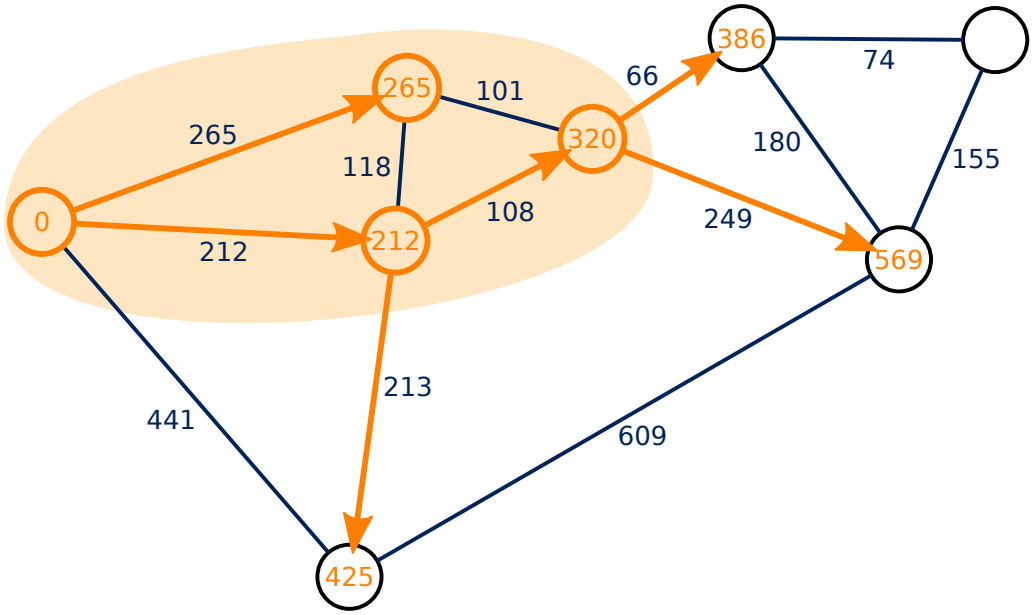
> Die zweite Iteration beginnt wieder mit dem Hineinziehen des Knotens v minimalen Werts $D[v]$ in die Wolke. Dies ist Innsbruck mit dem Wert 212.
>
> Anschließend werden die Nachbarn von Innsbruck betrachtet, um ggf. ihren Wert $D[v]$ zu reduzieren. Der Pfad nach München über Innbruck hat den Wert D[Innsbruck] + w(Innsbruck, München) = 212+118 = 330, was mehr ist als der aktuelle Wert D[München]. Daher bleibt dieser unverändert.
>
> Der folgende Nachbar von Innsbruck ist Salzburg. Sein Wert reduziert sich von unendlich auf 212+108=320, da nun ein Pfad über Innsbruck bekannt ist.
>
> Der Pfad nach Verona über Innsbruck hat hier den Wert D[Innsbruck] + w(Innsbruck, Verona) = 212+213 = 425; dies ist *weniger* als der bisherige Wert 441. Daher wird der Wert D[Verona] von 441 auf 425 reduziert, und der kürzeste bekannte Pfad von Zürich nach Verona führt nun über Innsbruck.
>
> Der orange Pfeil von Zürich nach Innsbruck liegt nun komplett innerhalb der Wolke. Damit repräsentiert er den garantiert kürzesten Pfad von Zürich nach Innsbruck, und ist Teil des Baums der kürzesten Pfade, dessen Konstruktion sich hier vor unseren Augen entfaltet.

Beispielablauf von Dijkstras Algorithmus (3) [Slide 48]



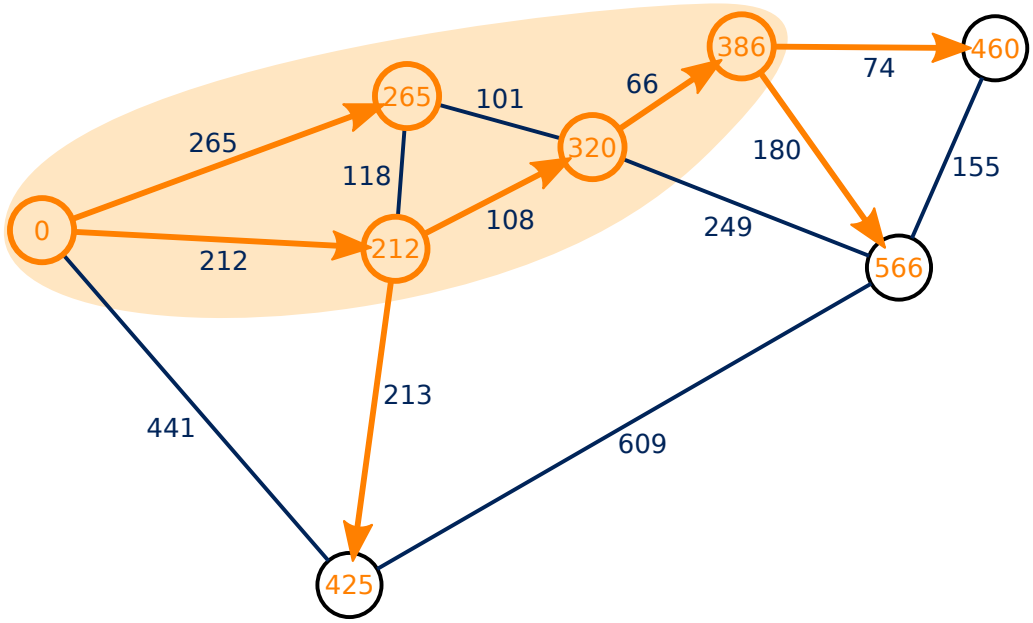
In der dritten Iteration hat München den kleinsten Wert $D[v]$ unter allen Knoten v außerhalb der Wolke. Über München führt jedoch kein Weg zu anderen Knoten außerhalb der Wolke, der kürzer wäre als der kürzeste bekannte Weg über Innsbruck.

Beispielablauf von Dijkstras Algorithmus (4) [Slide 49]



Als Nächstes ist Salzburg an der Reihe, und erschließt der Wolke die neuen Nachbarn Linz und Graz.

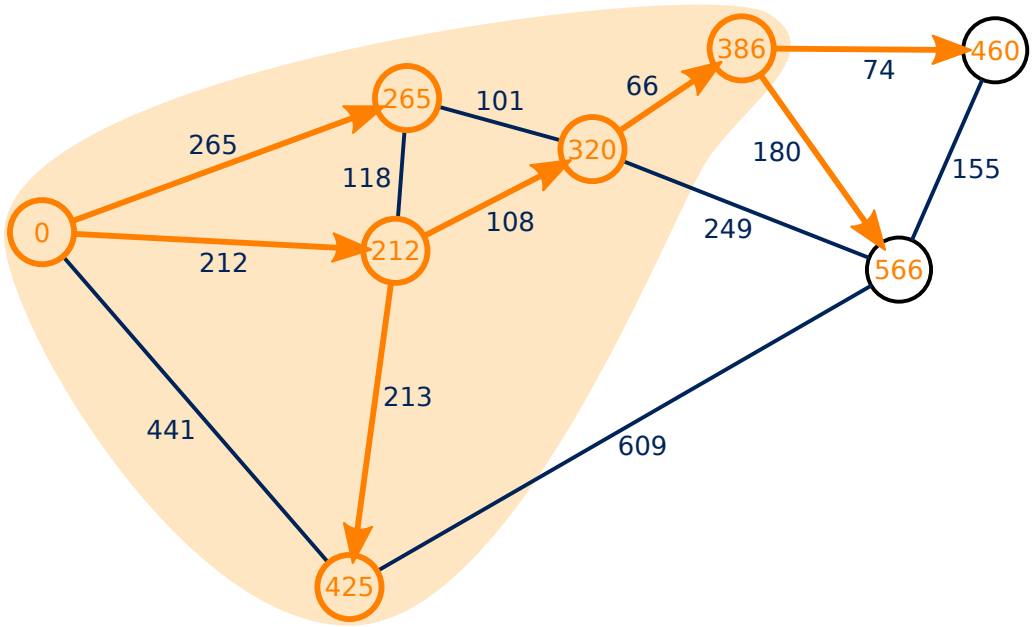
Beispielablauf von Dijkstras Algorithmus (5) [Slide 50]



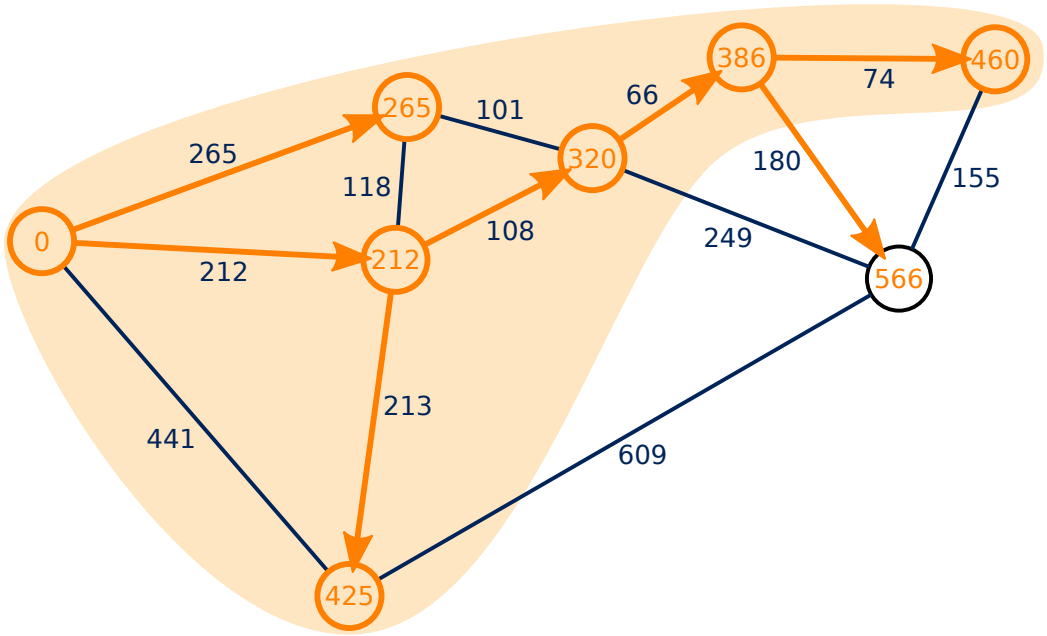
Edge relaxation findet einen kürzeren Pfad nach Graz.

> In der fünften Iteration ist Linz der Knoten v außerhalb der Wolke mit minimalem $D[v]$. Über Linz führt der erste bekannte Weg von Zürich nach Wien der Länge 386+74=460. Hiermit reduziert sich die Länge des kürzesten bekannten Pfades von Zürich nach Graz von 569 auf 386+180=566. Der kürzeste bekannte Pfad führt ab sofort über Linz.
>
> Anschließend werden noch die Knoten Verona, Wien und Graz in die Wolke gezogen, was ansonsten keine weiteren Veränderungen nach sich zieht.

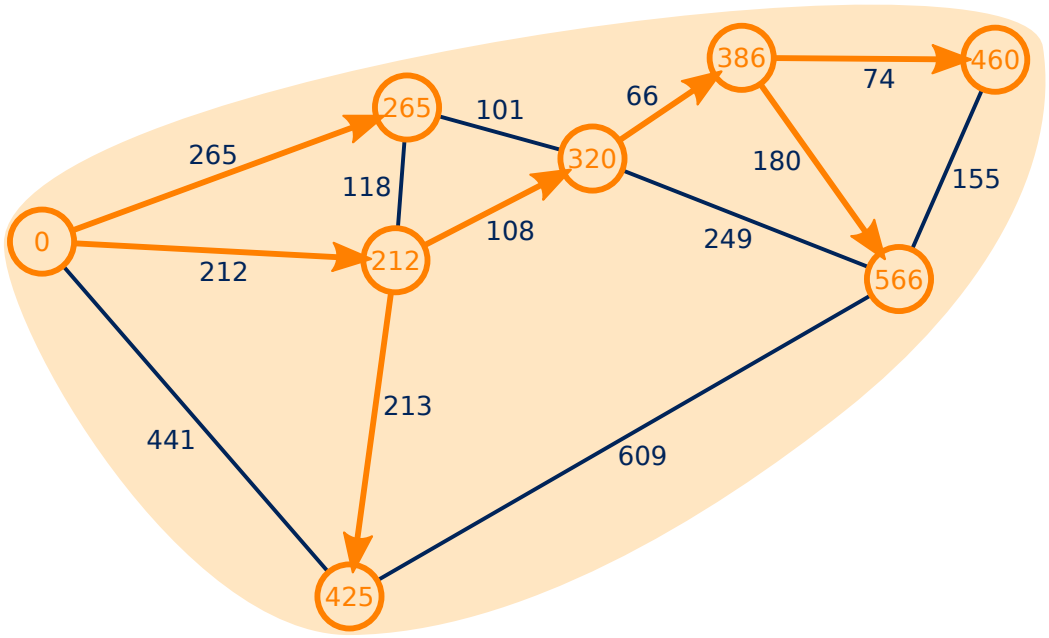
Beispielablauf von Dijkstras Algorithmus (6) [Slide 51]



Beispielablauf von Dijkstras Algorithmus (7) [Slide 52]



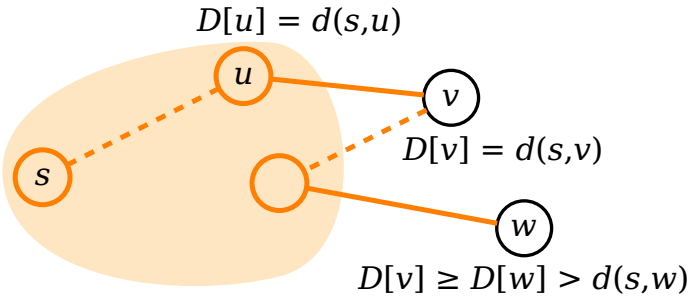
Beispielablauf von Dijkstras Algorithmus (8) [Slide 53]



Damit ist die Vorrangwarteschlange Q leer, die Wolke C umfasst den gesamten Graphen, und die orangen Pfeile markieren den Baum der kürzesten Pfade von Zürich zu allen anderen Knoten.

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.

Was garantiert, dass die von Dijkstras Algorithmus gefundenen Pfade tatsächlich die kürzesten sind? Der entscheidende Punkt ist, dass, wenn ein Knoten w in die Wolke gezogen wird, Groß-$D[w]$ tatsächlich die Länge Klein-$d(s, w)$ des kürzesten Pfades von s nach w ist. Mit anderen Worten, diese Länge kann zu diesem Zeitpunkt nicht mehr überschätzt sein. Unterschätzt werden kann sie generell nicht, da $D[w]$ ausschließlich auf die Länge real entdeckter Pfade gesetzt wird.

Beweisen wir nun, dass $D[w] = d(s, w)$ sein muss, wenn w in die Wolke gezogen wird. Hierzu nehmen wir an, es gebe einen Knoten, auf den dies *nicht* zutrifft, und führen diese Annahme zu einem Widerspruch.

Sei w der erste solche Knoten mit $D[w] > d(s, w)$. Ferner sei v der erste Knoten außerhalb von C auf dem kürzesten Pfad P von s nach w; v mag mit w identisch sein, oder auch nicht.

Sei u der Vorgänger von v in P. Damit liegt u in der Wolke C, und, gemäß unserer Annahme, dass w der *erste* Knoten mit überschätzer kürzester Pfadlänge ist, ist $D[u] = d(s, u)$.

> Zu dem Zeitpunkt, als u in die Wolke C gezogen wurde, wurde sichergestellt, dass $D[v]$ maximal $D[u] + w(u, v)$ ist, also $d(s, u) + w(u, v)$. Da u und v jedoch auf dem kürzesten Pfad von s nach w liegen, muss auch die Kante (u, v) Teil eines kürzesten Pfades von s nach v sein. Daraus folgt, dass $D[v]$ in der Tat *gleich* $d(s, v)$ ist.
>
> Ist $v = w$, dann haben wir an dieser Stelle bereits einen Widerspruch und damit bewiesen, dass ein solcher Knoten w nicht existieren kann. Nehmen wir also nun an, dass es sich bei v und w um verschiedene Knoten handelt.
>
> Da nun w in die Wolke C gezogen wird und nicht v, muss $D[w] \leq D[v]$ sein. Und da v auf dem kürzesten Pfad von s nach w liegt, muss $d(s, v) + d(v, w) = d(s, w)$ sein.
>
> Wir sehen, dass hier etwas nicht stimmen kann: Wie kann w nun in die Wolke gezogen werden, obwohl v zwischen s und w und ebenfalls außerhalb der Wolke liegt?
>
> Tatsächlich folgt aus den Relationen zwischen den verschiedenen Groß-D und Klein-d, die wir nun zusammengetragen haben, dass $D[w] \leq d(s, w)$ ist. Dies widerspricht jedoch unserer Grundannahme, dass $D[w] > d(s, w)$ sei. Daraus folgt, dass unsere Grundannahme niemals zutreffen kann. Mit anderen Worten, wenn ein Knoten w in die Wolke gezogen wird, ist sein Wert $D[w]$ garantiert gleich der Länge eines kürzesten Pfades von s nach w.

Laufzeit von Dijkstras Algorithmus [Slide 55]

- Der Graph G habe n Knoten und m Kanten.
- Annahmen:
  - Auf Kantenbewertungen kann in konstanter Zeit zugegriffen 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()`
  - $O(m)$ 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.

Was ist die Laufzeit von Dijkstras Algorithmus?

Nehmen wir an, der Graph habe n Knoten und m Kanten, Kantenbewertungen können in konstanter Zeit hinzugefügt, abgefragt und verglichen werden, und der Graph ist mittels einer Adjazenzliste oder Adjazenz-Zuordnungstabelle implementiert, damit wir die eingehenden Kanten eines Knotens effizient bestimmen können.

Die `while`-Schleife iteriert $O(n)$ Mal, da sie in jeder Iteration einen Knoten aus der Vorrangwarteschlange entfernt. Die Gesamtzahl der Iterationenn der inneren `foreach`-Schleife ist $O(m)$, da jede Iteration eine andere eingehende Kante eines Knotens behandelt.

Die Laufzeit-relevanten Operationen sind also n Einfügungen in die Vorrangwarteschlange Q, n Aufrufe von `removeMin()` und m Anpassungen der adaptiven Vorrangwarteschlange mittels seiner Methode `replaceKey()`.

Basiert die Vorrangwarteschlange auf einem Heap, laufen alle diese Aufrufe jeweils in einer Zeit von $O(\log n)$, und damit ist die Gesamtlaufzeit $O((n+m)\log n)$. Bei einem Graphen ohne Mehrfachkanten ist $m \in O(n^2)$, und bei zusammenhängenden Graphen ist $n \in O(m)$; damit lässt sich dieser Ausdruck entsprechend vereinfachen.

Basiert die Vorrangwarteschlange auf einer unsortierten Liste, dann verschlechtert sich die Laufzeit von `removeMin()` auf $O(n)$. Aber dafür verbessert sich die Laufzeit von `replaceKey()` auf $O(1)$, da es für diese Methode nichts zu tun gibt. In diesem Fall ist die Gesamtlaufzeit daher $O(n^2+m)$, also (n^2) für Graphen ohne Mehrfachkanten.

Wie sollten wir nun die Vorrangwarteschlange für Dijkstras Algorithmus implementieren, mit einem Heap oder mit einer unsortierten Liste? Die Antwort hängt davon ab, wie *dicht* der Graph ist, d.h., wie viele Kanten er im Verhältnis zu seinen Knoten besitzt. Bei $m = n^2/\log n$ haben wir asymptotischen Gleichstand. Besitzt der Graph weniger als $n^2/\log n$ Kanten, dann gewinnt der Heap; besitzt er mehr, dann gewinnt die unsortierte Liste.

In der Praxis wird Dijkstras Algorithmus übrigens mit einem sogenannten Fibonacci-Heap implementiert. Dieser garantiert eine Laufzeit von $O(m + n\log n)$, und ist damit asymptotisch immer mindestens so schnell wie sowohl der gewöhnliche binäre Heap als auch die unsortierte Liste.

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:  T is 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 Zusammenfassung

Zusammenfassung [Slide 57]

- Graph: Definition, Terminologie, ADT
- Datenstrukturen:
  - Kantenliste
  - Adjazenzliste
  - Adjazenz-Zuordnungstabelle
  - Adjazenzmatrix
- Traversierungen: Tiefen-, Breiten-
- Kürzeste Pfade: Dijkstras Algorithmus

Literatur [Slide 58]

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