Graphen

Graphen

(a) Ein **ungerichteter Graph** $G = (V, E)$ besteht aus einer endlichen Knotenmenge V und einer Kantenmenge

$$E \subseteq \{ \{i, j\} \mid i, j \in V, i \neq j\}$$

- Die **Endpunkte** u, v einer ungerichteten Kante $\{u, v\}$ sind gleichberechtigt.
- u und v heißen **Nachbarn**.

(b) Die Kantenmenge E eines **gerichteten Graphen** $G = (V, E)$ ist

$$E \subseteq \{(i, j) \mid i, j \in V, i \neq j\}$$

- Der Knoten u ist **Anfangspunkt** und der Knoten v **Endpunkt** der Kante (u, v).
- v heißt auch ein **direkter Nachfolger** von u und u ein **direkter Vorgänger** von v.

Warum Graphen?

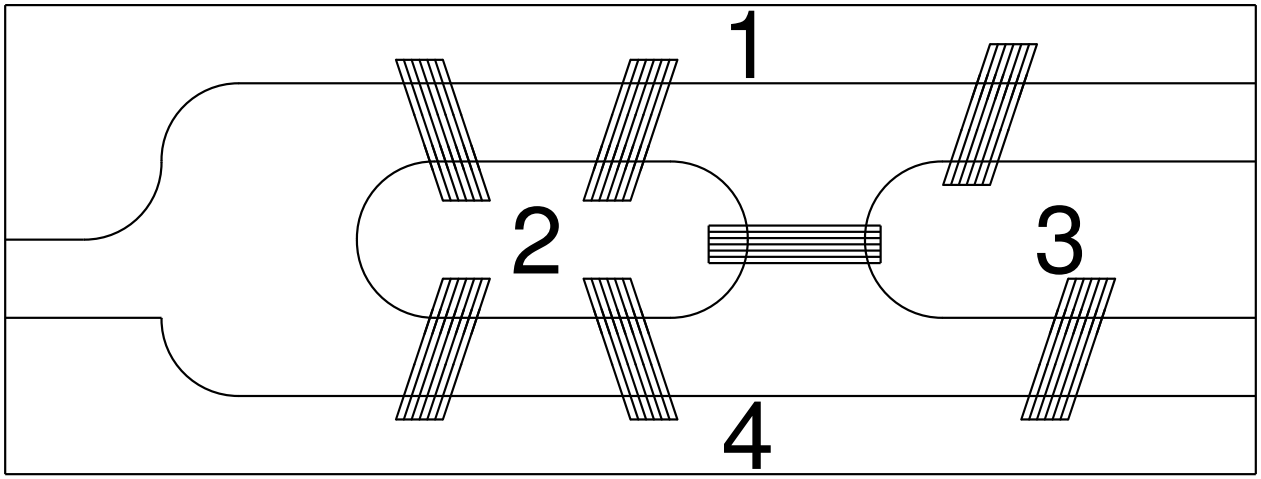
Graphen modellieren

- das **World Wide Web**: Die Knoten entsprechen Webseiten, die (gerichteten) Kanten entsprechen Hyperlinks.
- **Rechnernetzwerke**: Die Knoten entsprechen Rechnern, die (gerichteten und/oder ungerichteten) Kanten entsprechen Direktverbindungen zwischen Rechnern.
- Das **Schienennetz der Deutschen Bahn**: Die Knoten entsprechen Bahnhöfen, die (ungerichteten) Kanten entsprechen Direktverbindungen zwischen Bahnhöfen.

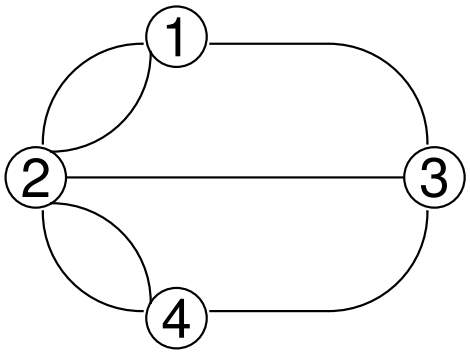
  Bei der Erstellung von Reiseplänen bestimme kürzeste (gewichtete) Wege zwischen einem Start- und einem Zielbahnhof.
- **Schaltungen**: die Knoten entsprechen Gattern, die (gerichteten) Kanten entsprechen Leiterbahnen zwischen Gattern.

Das Königsberger Brückenproblem

Gibt es einen Rundweg durch Königsberg, der alle Brücken über die Pregel genau einmal überquert?



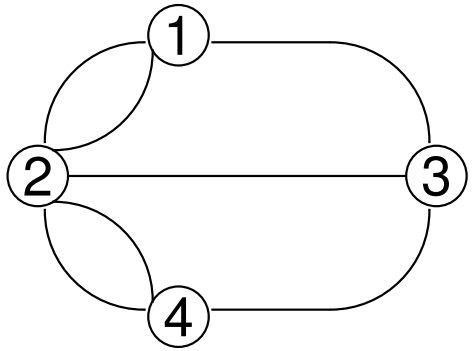
Der zugehörige ungerichtete Graph:



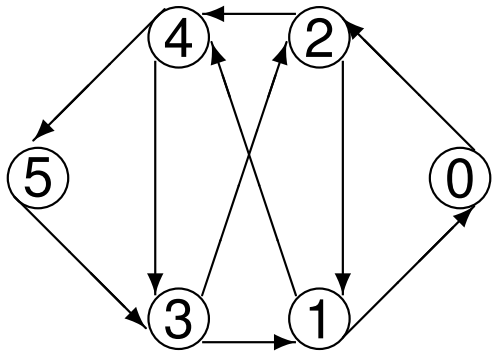
Euler-Kreise

Ein **Euler-Kreis** beginnt in einem Knoten v, durchläuft alle Kanten genau einmal und kehrt dann zu v zurück.

Das Königsberger Brückenproblem besitzt keine Lösung. Der Graph



hat keinen Euler-Kreis: Ansonsten hätte jeder Knoten eine gerade Anzahl von Nachbarn! Und der folgende Graph?



Wichtige Begriffe

Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph.

- Eine Folge $(v_0, v_1, ..., v_m)$ heißt ein **Weg** in G, falls für jedes i $(0 \leq i < m)$
  - $(v_i, v_{i+1}) \in E$ (für gerichtete Graphen) oder
  - $\{v_i, v_{i+1}\} \in E$ (für ungerichtete Graphen).

  Die **Weglänge** ist m, die Anzahl der Kanten. Ein Weg heißt **einfach**, wenn kein Knoten zweimal auftritt.
- Ein Weg heißt ein **Kreis**, wenn $v_0 = v_m$ und $(v_0, ...v_{m-1})$ ein einfacher Weg ist.
  G heißt **azyklisch**, wenn G keine Kreise hat.
- Ein ungerichteter Graph heißt **zusammenhängend**, wenn je zwei Knoten durch einen Weg miteinander verbunden sind.

Topologisches Sortieren

n Aufgaben $a_0, \ldots, a_{n-1}$ sind auszuführen. Allerdings gibt es eine Menge P von p Prioritäten zwischen den einzelnen Aufgaben.

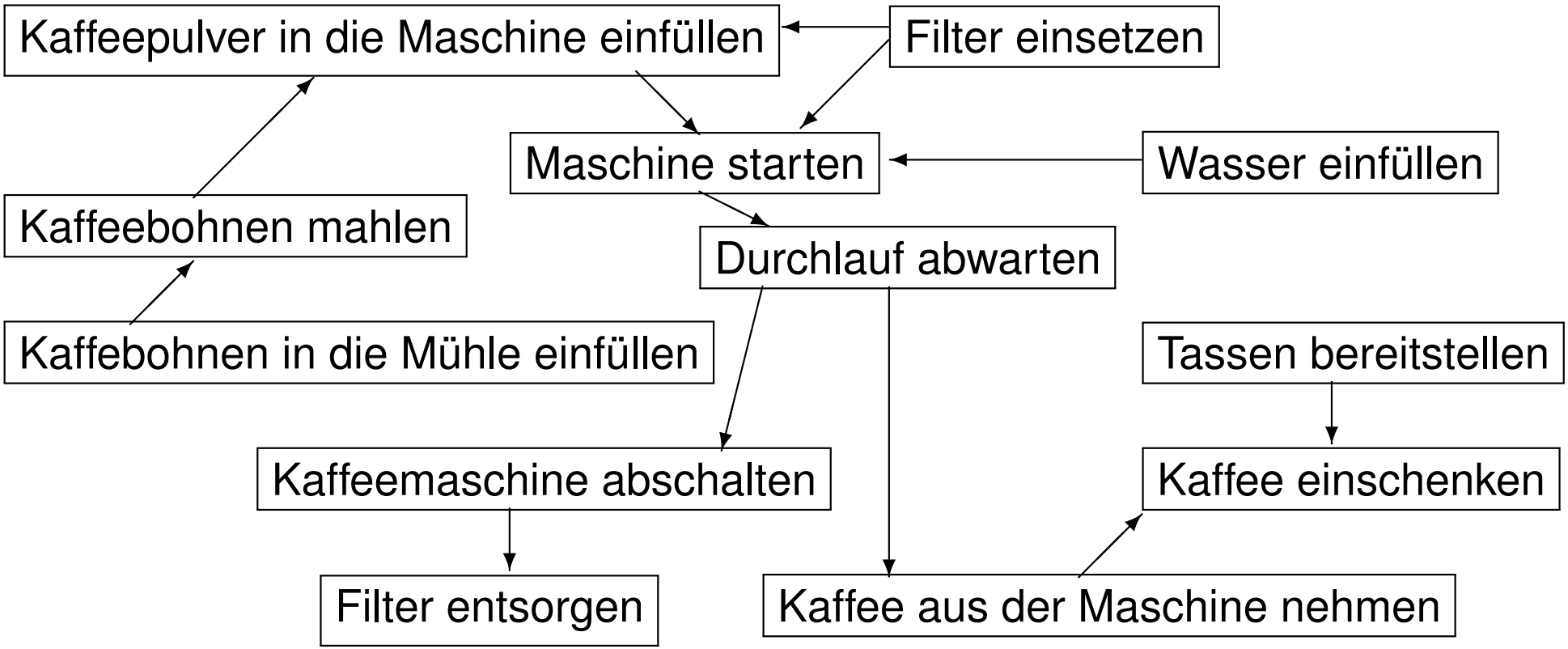
Die Priorität (i, j) impliziert, dass Aufgabe a_i vor Ausführung der Aufgabe a_j ausgeführt werden muss.

Bestimme eine Reihenfolge, in der alle Aufgaben ausgeführt werden können, bzw. stelle fest, dass eine solche Reihenfolge nicht existiert.

- Eine **graph-theoretische Formulierung** mit Knotenmenge $V = \{0, \ldots, n-1\}$:
  - Knoten i entspricht der Aufgabe a_i.
  - Wir setzen für jede Priorität (i, j) die Kante (i, j) ein.
- Wie ist das **Ziel** zu formulieren?

  Bestimme eine **Reihenfolge $v_1, \ldots, v_i, \ldots, v_n$ der Knoten**, so dass es keine Kante (v_i, v_j) mit $j < i$ gibt.

Kaffeekochen



Die Idee

Eine Aufgabe a_j kann als **erste** Aufgabe ausgeführt werden, wenn es keine Priorität der Form (i, j) in P gibt.

- Ein Knoten v von G heißt eine **Quelle**, wenn In-Grad$(v) = 0$, wenn v also kein Endpunkt einer Kante ist.
- Also bestimme eine Quelle v, führe v aus und entferne v.
- Wiederhole dieses Verfahren, solange G noch Knoten besitzt:

  bestimme eine Quelle v, führe v aus und entferne v.

Welche Datenstrukturen sollten wir verwenden?

Ein erster Versuch

Wir verketten alle p Kanten in einer Liste „Priorität“ und benutzen ein integer Array „Reihenfolge“ sowie zwei boolesche Arrays „Erster“ und „Fertig“ mit jeweils n Zellen.

Zaehler$=0$. Für alle i setze Fertig$[i] =$ falsch.

Wiederhole n-mal:

(0) Setze Erster$[i] =$ wahr genau dann, wenn Fertig$[i] =$ falsch.

(1) Durchlaufe die Liste Priorität. Wenn Kante (i, j) angetroffen wird, setze Erster$[j]$ = falsch.

(2) Bestimme das kleinste j mit Erster$[j]$ = wahr. Danach setze

- (a) Fertig$[j] =$ wahr,
- (b) Reihenfolge[Zaehler++] $= j$ (Aufgabe j wird ausgeführt)
- (c) und durchlaufe die Prioritätsliste: entferne jede Kante (j, k), da a_j eine Ausführung von Aufgabe a_k nicht mehr behindert.

Laufzeit Topologisches Sortieren (1) demogr.

Worst-Case Laufzeit für den ersten Versuch?

- (1) $\Theta(n+p)$
- (2) $\Theta(n \cdot p)$
- (3) $\Theta(n \cdot \log n)$
- (4) $\Theta(n \cdot (n+p))$

Auflösung: (4) $\Theta(n \cdot (n+p))$

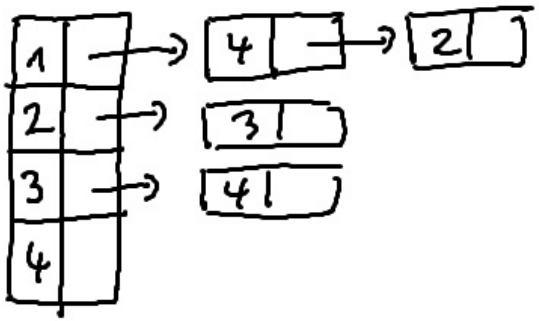
Eine Laufzeitanalyse

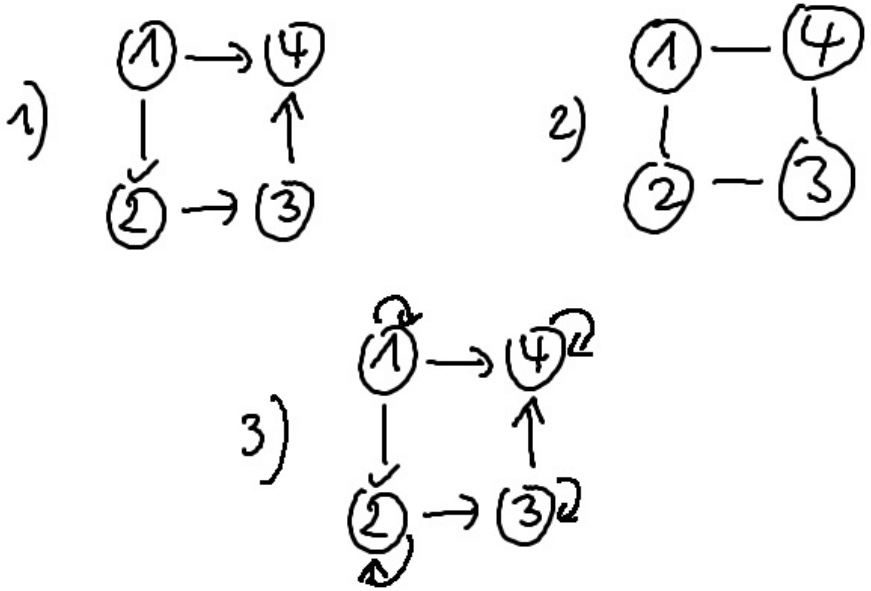
- Was ist besonders teuer?
  - In jeder Iteration wird die Liste Priorität vollständig durchlaufen:
    Zeit $= O(p)$.
  - Weiterhin wird das Array Erster jeweils initialisiert:
    Zeit $= O(n)$.
  - Die Laufzeit pro Iteration ist dann durch $O(n + p)$ beschränkt.
    Die Gesamtlaufzeit ist $O(n \cdot (n + p))$, da wir n Iterationen haben.
- Was können wir verbessern?
  - Wir müssen nur die Kanten entfernen, die im gerade ausgeführten Knoten j beginnen.
  - Warum kompliziert nach der ersten ausführbaren Aufgabe suchen?

    Eine vorher nicht in Betracht kommende Aufgabe k wird nur interessant, wenn (j, k) eine Priorität ist.

Adjazenzlistendarstellung demogr.

Welcher Graph wird repräsentiert?





Auflösung: (1)

Der zweite Versuch

Stelle die Prioritäten durch eine Adjazenzliste mit dem Kopf-Array **Priorität** dar. Benutze ein Array **In-Grad** mit In-Grad$[v] = k$, falls v Endpunkt von k Kanten ist.

(1) Initialisiere Adjazenzliste Priorität durch Einlesen aller Prioritäten. (Zeit = $O(n + p)$).

(2) Initialisiere das Array In-Grad. (Zeit = $O(n + p)$).

(3) Alle Knoten v mit In-Grad$[v] = 0$ werden in eine **Schlange** eingefügt. (Zeit = $O(n)$).

(4) Setze Zähler $= 0$; Wiederhole solange, bis *Schlange* leer ist:

  (a) Entferne einen Knoten i aus *Schlange*.

  (b) Setze **Reihenfolge** [Zähler++] $= i$.

  (c) Durchlaufe die Liste Priorität [i] und reduziere In-Grad für jeden Nachfolger j von i um 1. Wenn jetzt In-Grad[j] $= 0$, dann füge j in *Schlange*: Aufgabe a_j ist jetzt ausführbar.

Laufzeit Topologisches Sortieren (2) demogr.

Worst-Case Laufzeit für den zweiten Versuch?

- (1) $\Theta(n+p)$
- (2) $\Theta(n \cdot p)$
- (3) $\Theta(n \cdot \log n)$
- (4) $\Theta(n \cdot (n+p))$

Auflösung: (1) $\Theta(n+p)$

Die Analyse

- Die Vorbereitungsschritte (1), (2) und (3) laufen in $O(n+p)$ Schritten ab.
- Ein Knoten wird nur einmal in die Schlange eingefügt. Also beschäftigen sich höchstens $O(n)$ Schritte mit der Schlange.
- Eine Kante (i,k) wird, mit Ausnahme der Vorbereitungsschritte, nur dann inspiziert, wenn i aus der Schlange entfernt wird.
  - Jede Kante wird nur einmal „angefasst“
  - und höchstens $O(p)$ Schritte behandeln Kanten.

Das Problem des topologischen Sortierens wird für einen Graphen mit n Knoten und p Kanten in Zeit $O(n+p)$ gelöst.

Schneller geht’s nimmer.

Adjazenzlisten und die Adjazenzmatrix

Graph-Implementierungen

- Welche Datenstruktur sollten wir für die Darstellung eines Graphen G wählen?
- Welche Operationen sollen schnell ausführbar sein?
  - Ist e eine Kante von G?
    Die **Adjazenzmatrix** wird sich als eine gute Wahl herausstellen.
  - Bestimme Nachbarn (bzw. Vorgänger/Nachfolger) eines Knotens.
    Die **Adjazenzlistendarstellung** ist unschlagbar.

Besonders die Nachbar- und Nachfolgerbestimmung ist wichtig, um Graphen zu durchsuchen.

Die Adjazenzmatrix

Für einen Graphen $G = (V, E)$ (mit $V = \{0, ..., n-1\}$) ist

$$A_G[u, v] = \begin{cases} 1 & \text{wenn } \{u, v\} \in E \text{ (bzw. wenn } (u, v) \in E), \\ 0 & \text{sonst} \end{cases}$$

die Adjazenzmatrix A_G von G.

+ Eine Kantenfrage „**ist (u, v) eine Kante?**“ wird sehr schnell beantwortet, nämlich in Zeit $O(1)$.
- Die Bestimmung aller Nachbarn oder Nachfolger eines Knotens v ist hingegen langwierig:
  - Die Zeile von v muss durchlaufen werden.
  - Zeit $\Theta(n)$ ist sogar notwendig, wenn v nur wenige Nachbarn hat.
- Speicherplatzbedarf $\Theta(n^2)$ auch für Graphen mit wenigen Kanten:

  Die Datenstruktur passt sich nicht der Größe des Graphen an!

Die Adjazenzliste

G wird durch ein Array A von Listen dargestellt. Die Liste $A[v]$ führt alle Nachbarn von v auf, bzw. alle Nachfolger von v für gerichtete Graphen.

+ Die **Nachbar- bzw. Nachfolgerbestimmung** für Knoten v gelingt in Zeit proportional zur Anzahl der Nachbarn oder Nachfolger.
+ Der benötigte Speicherplatz ist $O(n + |E|)$: Die Datenstruktur passt sich der Größe des Graphen an.
- Für die Beantwortung der Kantenfrage „ist (u, v) eine Kante?“ muss die Liste $A[v]$ durchlaufen werden: Die benötigte Zeit ist also proportional zur Anzahl der Nachbarn oder Nachfolger.

Da die Nachbar- bzw. Nachfolgerbestimmung für das Durchlaufen von Wegen benötigt wird, ist die sich der Größe des Graphen anpassende Adjazenzliste die Datenstruktur der Wahl.

Suche in Graphen: Tiefensuche

Suche in Graphen

Wie durchsucht man ein Labyrinth? Können wir Präorder benutzen?

- Präorder terminiert nur für Wälder.
  - Präorder wird von Kreisen in eine Endlosschleife gezwungen,
  - es erkennt nicht, dass Knoten bereits besucht wurden!
- Können wir Präorder reparieren?

Wie findet man Wege aus einem Labyrinth?

Ein Auszug aus *Umbert Eco's „Der Name der Rose"*.

William von Baskerville und sein Schüler Adson van Melk sind heimlich in die als Labyrinth gebaute Bibliothek eines hochmittelalterlichen Klosters irgendwo im heutigen Norditalien eingedrungen.

Fasziniert von den geistigen Schätzen, die sie beherbergt, haben sie sich nicht die Mühe gemacht, sich ihren Weg zu merken.

Erst zu spät erkennen sie, dass die Räume unregelmäßig und scheinbar wirr miteinander verbunden sind.

Man sitzt fest.

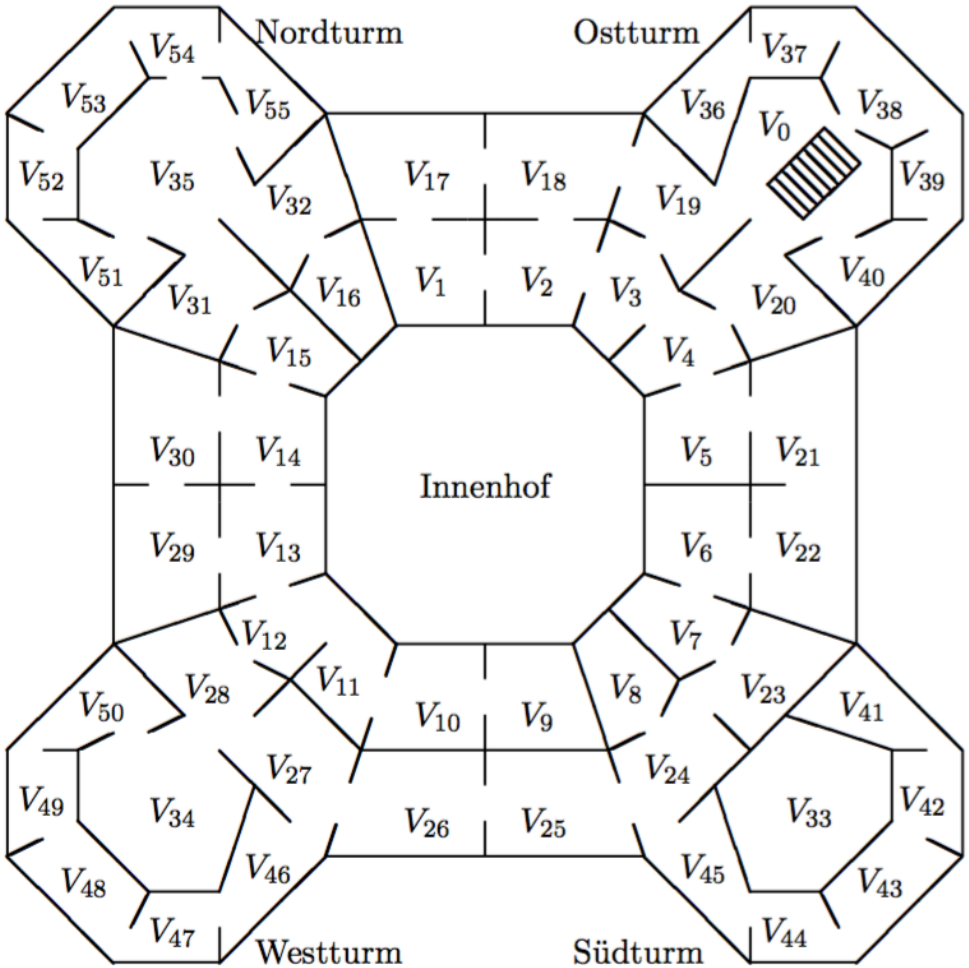
William erinnert sich

„Um den Ausgang aus einem Labyrinth zu finden,“, dozierte William, „gibt es nur ein Mittel. An jedem Kreuzungspunkt wird der Durchgang, durch den man gekommen ist, mit drei Zeichen markiert. Erkennt man an den bereits vorhandenen Zeichen auf einem der Durchgänge, dass man an der betreffenden Kreuzung schon einmal gewesen ist, bringt man an dem Durchgang, durch den man gekommen ist, nur ein Zeichen an. Sind alle Durchgänge schon mit Zeichen versehen, so muss man umkehren und zurückgehen. Sind aber einer oder zwei Durchgänge der Kreuzung noch nicht mit Zeichen versehen, so wählt man einen davon und bringt zwei Zeichen an. Durchschreitet man einen Durchgang, der nur ein Zeichen trägt, so markiert man ihn mit zwei weiteren, so dass er nun drei Zeichen trägt. Alle Teile des Labyrinthes müßten durchlaufen worden sein, wenn man, sobald man an eine Kreuzung gelangt, niemals den Durchgang mit drei Zeichen nimmt, sofern noch einer der anderen Durchgänge frei von Zeichen ist.“

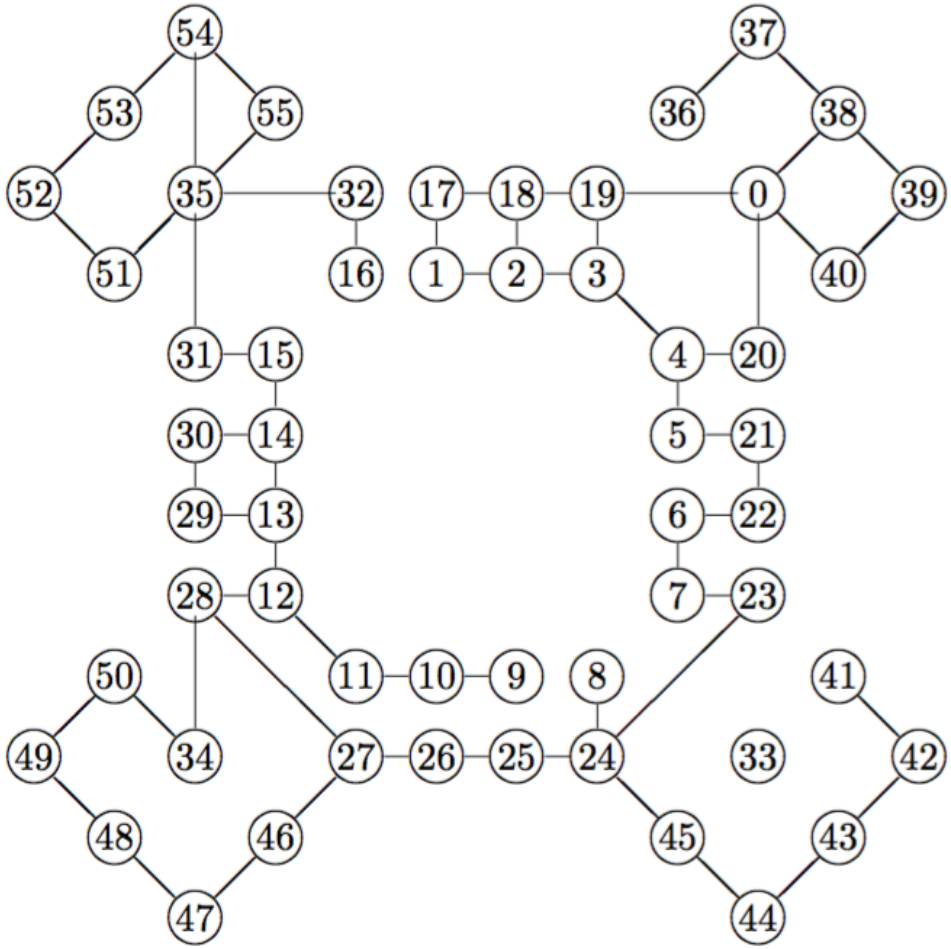
„Woher wißt Ihr das? Seid Ihr ein Experte in Labyrinthen?“
„Nein, ich rezitiere nur einen alten Text, den ich einmal gelesen habe.“
„Und nach dieser Regel gelangt man hinaus?“
„Nicht dass ich wüßte. Aber wir probieren es trotzdem.[...]“

„Der Name der Rose“: Das Labyrinth

Kann man vom Treppenaufgang V_0 aus alle Räume V_i besuchen?



Das Labyrinth als ungerichteter Graph



Der Ariadne-Faden

Geht das denn nicht viel einfacher?

1. Prinzessin Ariadne, Tochter des Königs Minos, hat Theseus den „Ariadne-Faden“ geschenkt, um den Minotauros in einem Labyrinth aufzuspüren und danach wieder aus dem Labyrinth herauszufinden.
2. Theseus hat den Ariadne-Faden während der Suche im Labyrinth abgerollt.
   - Nachdem er den Minotauros getötet hat, braucht er nur den Faden zurückverfolgen, um das Labyrinth wieder verlassen zu können.
3. Aber auch Präorder benutzt den Ariadne-Faden. Und wie, bitte schön, **durchsucht** man das Labyrinth systematisch mit Hilfe eines Fadens?

Tiefensuche: Farbeimer + Ariadne-Faden

Der Algorithmus „**Tiefensuche**“ implementiert und erweitert die Methode des Ariadne-Fadens.

1. Der ungerichtete Graph $G = (V, E)$ und ein Startknoten $s \in V$ ist gegeben.
2. Ganz zu Anfang sind alle Knoten „unmarkiert“. Wir besuchen und **markieren** s.
   // Wir besuchen stets nur **unmarkierte** Knoten.
3. Wenn wir den Knoten u besuchen, betrachten wir nacheinander alle Nachbarn v von u in irgendeiner Reihenfolge.
   - Wenn v markiert ist, tun wir nichts.
   - Wenn v unmarkiert ist, besuchen und **markieren** wir v. Dann wiederholen wir unser Vorgehen rekursiv (für alle mit v benachbarten Knoten).

   Wenn schließlich alle Nachbarn von v markiert sind, dann kehren wir zu u zurück.

(Wir benutzen den Ariadne-Faden und einen Farbeimer.)

Tiefensuche(): Die globale Struktur

Im Array besucht wird vermerkt, welche Knoten bereits besucht wurden.

```
void Tiefensuche()
  {for (int k = 0; k < n; k++) besucht[k] = 0;
   for (k = 0; k < n; k++)
     if (! besucht[k]) tsuche(k); }
```

- Jeder Knoten wird besucht, aber tsuche(v) wird nur dann aufgerufen, wenn v nicht als „besucht“ markiert ist.
- Wie funktioniert tsuche(v)?

tsuche()

- Der gerichtete oder ungerichtete Graph G werde durch seine Adjazenzliste A repräsentiert.
- Die Adjazenzlisten werden definiert durch Listeneinträge der Form

```
struct Knoten {
  int name;
  Knoten * next; }
```

tsuche(v):

1. Zuerst wird v markiert.
2. Dann rufe tsuche rekursiv für alle unmarkierten Nachbarn/Nachfolger von v auf.

```
void tsuche(int v)
  { Knoten *p ; besucht[v] = 1;
    for (p = A[v]; p !=0; p = p->next)
      if (!besucht [p->name]) tsuche(p->name);
        // Die Kante {v, p->name} heißt eine Baumkante }
```

Ungerichtete Graphen: Baum- und Rückwärtskanten

Der Wald der Tiefensuche: ungerichtete Graphen

Wir veranschaulichen das Vorgehen von Tiefensuche, indem wir die Kanten des Graphen $G = (V, E)$ in zwei Klassen aufteilen, nämlich

- **Baumkanten** und
- **Rückwärtskanten**.

- Eine Kante $\{v, w\} \in E$ des Graphen G heißt eine

  Baumkante,

  falls tsuche(w) in der for-Schleife von tsuche(v) aufgerufen wird oder umgekehrt.
  - Die Baumkanten definieren einen Wald, den wir den

    Wald der Tiefensuche

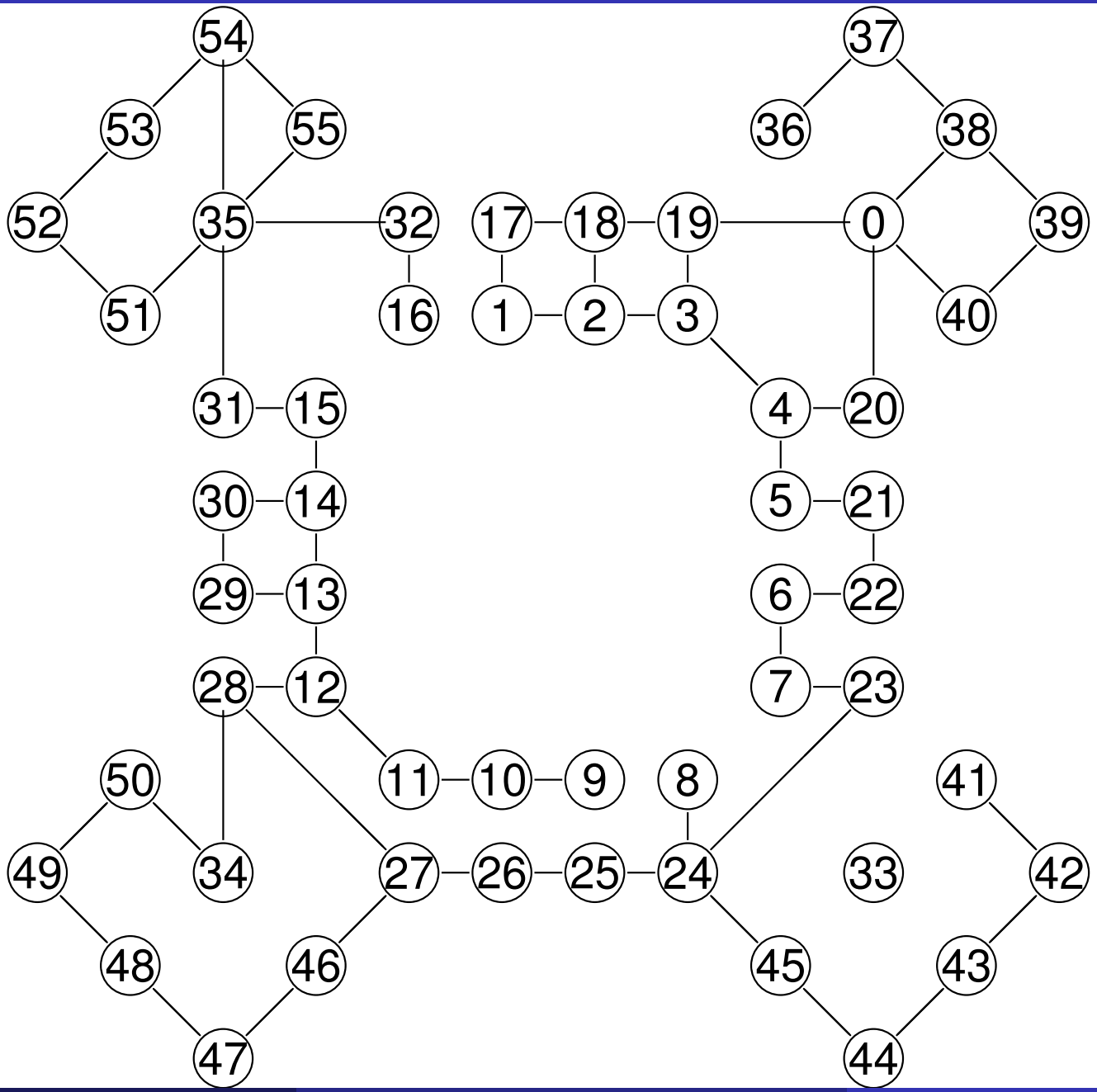
    nennen.
- Eine Kante $\{v, w\} \in E$ heißt eine

  Rückwärtskante,

  falls $\{v, w\}$ keine Baumkante ist.

Warum sprechen wir von Rückwärtskanten?

Tiefensuche: Ein Beispiel



Tiefensuche für ungerichtete Graphen (1/2)

Sei $G = (V, E)$ ein ungerichteter Graph und $\{v, w\}$ sei eine Kante von G. W_G sei der Wald der Tiefensuche für G.

(a) tsuche(v) werde vor tsuche(w) aufgerufen. Dann ist w ein Nachfahre von v in W_G. Insbesondere gehören v und w zum selben Baum von W_G.

(b) Alle Kanten des Graphen verbinden einen Vorfahren mit einem Nachfahren.

(a) Warum ist w ein Nachfahre von v in W_G?
- tsuche(v) wird vor tsuche(w) aufgerufen:
  Knoten w ist zum Zeitpunkt der Markierung von Knoten v *unmarkiert*.
- tsuche(v) kann nur dann terminieren, wenn w (zwischenzeitlich) markiert wird: w muss während der Ausführung von tsuche(v) markiert werden.

(b) O.B.d.A. werde tsuche(v) vor tsuche(w) aufgerufen. Dann ist w Nachfahre von v:
Die Graphkante $\{v, w\}$ ist eine Baumkante oder eine Rückwärtskante:
In beiden Fällen wird ein Vorfahre mit einem Nachfahren verbunden.

Tiefensuche für ungerichtete Graphen (2/2)

Tiefensuche besucht jeden Knoten genau einmal.

- Das Programm Tiefensuche wird von einer for-Schleife gesteuert, die tsuche(v) für alle noch nicht besuchten Knoten v aufruft.
- Wenn aber tsuche(v) aufgerufen wird, dann wird v sofort markiert: Nachfolgende Besuche sind ausgeschlossen.

Der Baum von v in W_G enthält genau die Knoten der **Zusammenhangskomponente** von v: Die Bäume von W_G entsprechen den Zusammenhangskomponenten von G.

- T sei ein Baum im Wald W_G und T besitze v als Knoten.
  - v erreicht jeden Knoten in T, denn der Baum T ist zusammenhängend: Die Zusammenhangskomponente von v enthält alle Knoten in T.
- Wenn $v = v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_m = u$ ein Weg in G ist, dann gehören $v_0, v_1, \ldots, v_m$ alle zum selben Baum:
  Die Knotenmenge von T enthält die Zusammenhangskomponente von v.

Labyrinthe

Tiefensuche löst jedes Labyrinth-Problem, das sich als ungerichteter Graph interpretieren läßt.

- Wenn es möglich ist, von irgendeinem Punkt p aus den Ausgang zu erreichen, dann befinden sich p und Ausgang in derselben Zusammenhangskomponente.
- Der **Tiefensuchbaum** von p wird uns stets einen Weg aus dem Labyrinth zeigen.

Wie schnell findet man aus einem Labyrinth heraus? D.h., wie schnell ist Tiefensuche?

Die Laufzeit von Tiefensuche

Tiefensuche terminiert nach höchstens $O(n + |E|)$ Schritten.

- Zuerst muss der Aufwand für die for-Schleife in Tiefensuche bestimmt werden: $O(n)$ Schritte.
- Wieviele Schritte werden direkt von tsuche(v) ausgeführt (und nicht in nachfolgenden rekursiven Aufrufen)?
  $O(\text{grad}(v))$ Operationen, wobei grad(v) die Anzahl der Nachbarn von v ist.
- Wieviele Operationen werden insgesamt ausgeführt?

$$\begin{aligned} O\left(\sum_{v \in V} (1 + \text{grad}\,(v))\right) &= O\left(\sum_{v \in V} 1 + \sum_{v \in V} \text{grad}\,(v)\right) \\ &= O\left(|V| + |E|\right). \end{aligned}$$

Tiefensuche ist sehr schnell.

Tiefensuche und Darstellung multiple choice

Wie lange dauert eine Tiefensuche in einem Graphen mit n Knoten und m Kanten, je nach Darstellungsform des Graphen?

- $m = \Theta(n)$, Adjanzenzliste: (1) $\Theta(n+m)$ (2) $\Theta(n^2)$
- $m = \Theta(n)$, Adjanzenzmatrix: (3) $\Theta(n+m)$ (4) $\Theta(n^2)$
- $m = \Theta(n^2)$, Adjanzenzliste: (5) $\Theta(n+m)$ (6) $\Theta(n^2)$
- $m = \Theta(n^2)$, Adjanzenzmatrix: (7) $\Theta(n+m)$ (8) $\Theta(n^2)$

Auflösung: (1), (4), (5), (6), (7), (8)

Anwendungen der Tiefensuche

Sei $G = (V, E)$ ein ungerichteter Graph. Dann kann in Zeit $O(|V| + |E|)$ überprüft werden, ob

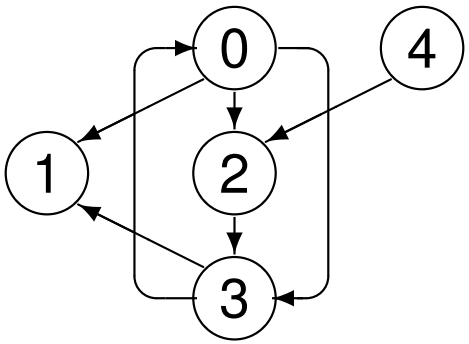
(a) G **zusammenhängend** ist:

- G ist genau dann zusammenhängend, wenn G genau eine Zusammenhangskomponente hat.
- Die Bäume von W_G entsprechen den Zusammenhangskomponenten von G.
- G ist genau dann zusammenhängend, wenn W_G aus genau einem Baum besteht.

(b) G ein **Wald** ist:

- G ist genau dann ein Wald, wenn G keine Rückwärtskanten hat.
- Überprüfe für jede Kante $\{v, w\}$, ob entweder tsuche(w) direkt in tsuche(v) aufgerufen wird oder ob tsuche(v) direkt in tsuche(w) aufgerufen wird.

Tiefensuche für gerichtete Graphen (1/2)



Wenn Tiefensuche im Knoten 0 beginnt und die Knoten aufsteigend in jeder Liste aufgeführt sind:

- Die Kanten $(0, 1), (0, 2)$ und $(2, 3)$ sind **Baumkanten**.
- Die Kante $(3, 0)$ ist eine **Rückwärtskante**.
- Die Kante $(0, 3)$ ist eine **Vorwärtskante**, sie verbindet einen Knoten mit einem Nachfahren, der kein Kind ist.
- Die Kanten $(3, 1)$ und $(4, 2)$ sind **Querkanten**, sie verbinden zwei Knoten, die nicht miteinander „verwandt“ sind.

Gerichtete Graphen:

Baum-, Rückwärts und Vorwärtskanten sowie Rechts-nach-Links Querkanten

Tiefensuche für gerichtete Graphen (2/2)

Sei $G = (V, E)$ ein gerichteter Graph, der als Adjazenzliste vorliegt.

(a) Tiefensuche besucht jeden Knoten genau einmal.

(b) Die Laufzeit von Tiefensuche() ist durch $O(|V| + |E|)$ beschränkt.

(c) Während der Ausführung von tsuche(v) wird ein Knoten w genau dann besucht, wenn w auf einem Weg liegt,

dessen Knoten vor Beginn von tsuche(v) unmarkiert sind.

- Zu Beginn von tsuche sei w von v aus durch einen „unmarkierten Weg“ erreichbar.
- Dann kann tsuche(v) nur dann terminieren, wenn w während der Ausführung von tsuche(v) markiert wird.

Die folgende Aussage ist falsch: Während der Ausführung von tsuche(v) werden genau die Knoten besucht, die auf einem Weg mit Anfangsknoten v liegen.

Kantentypen für gerichtete Graphen

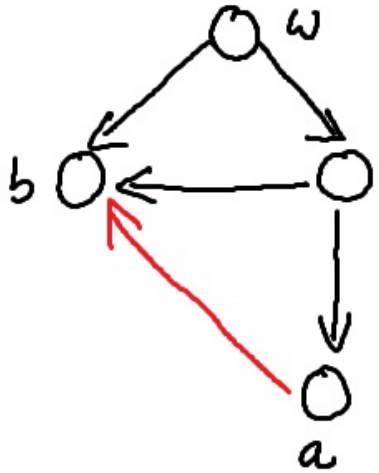
- Es ist nicht verwunderlich, dass durch die Kantenrichtungen neben **Rückwärtskanten** jetzt auch **Vorwärtskanten** vorkommen.
- **Querkanten** sind ein gänzlich neuer Kantentyp.
  - Ein Querkante heißt eine **rechts-nach-links Querkante**, wenn sie von einem **später** besuchten zu einem **früher** besuchten Knoten führt.

Es gibt nur rechts-nach-links Querkanten.

Warum? Sei $e = (v, w)$ eine beliebige Kante.

- Wenn tsuche(v) terminiert, dann ist w markiert.
- Wenn w vor dem Aufruf von tsuche(v) markiert wurde, dann ist e entweder eine Rückwärtskante oder eine rechts-nach-links Querkante.
- Wenn w während des Aufrufs markiert wird, dann ist e entweder eine Vorwärtskante oder eine Baumkante.

Kantenklassifizierung multiple choice



Betrachte die möglichen Abläufe der Tiefensuche mit Startknoten w.

Je nach Sortierung der Nachbarknoten wird Kante (a, b) zu einer

- (1) Baumkante
- (2) Rückwärtskante
- (3) Vorwärtskante
- (4) Querkante

Auflösung: (1), (4)

Eine automatische Erkennung der Kantentypen

Wir benutzen zwei integer-Arrays „**Anfang**“ und „**Ende**“ als Uhren, um den Zeitpunkt des Beginns und des Endes des Besuchs festzuhalten.

```
Anfangnr=Endenr=0;
void tsuche(int v)
  {Knoten *p; Anfang[v] = ++Anfangnr;
    for (p = A[v]; p != 0; p = p->next)
      if (!Anfang[p->name]) tsuche(p->name);
    Ende[v] = ++Endenr; }
```

- $e = (v, w)$ ist eine **Vorwärtskante** $\Leftrightarrow$
  Anfang$[v] <$ Anfang$[w]$ und $e = (v, w)$ ist keine Baumkante.
- $e = (v, w)$ ist eine **Rückwärtskante** $\Leftrightarrow$
  Anfang$[v] >$ Anfang$[w]$ und Ende$[v] <$ Ende$[w]$.
- $e = (v, w)$ ist eine **Querkante** $\Leftrightarrow$
  Anfang$[v] >$ Anfang$[w]$ und Ende$[v] >$ Ende$[w]$.

Kantenklassifizierung multiple choice

Betrachte die möglichen Abläufe der Tiefensuche mit Startknoten w.

Es gilt also immer Anfang[w] = 1.

Welche Werte können bei Anfang[v] auftreten, je nach Sortierung der Nachbarknoten?

Auflösung: Anfang[v] $\in \{3, 4\}$

Kantenklassifizierung demogr.

Für die Kante (v, w) gilt

- Anfang[v] = 3, Ende[v] = 1
- Anfang[w] = 1, Ende[w] = 3

Diese Kante ist eine

- (1) Baumkante
- (2) Rückwärtskante
- (3) Vorwärtskante
- (4) Querkante

Auflösung: (2) Rückwärtskante

Anwendungen der Tiefensuche (1/2)

Sei $G = (V, E)$ ein gerichteter Graph, der als Adjazenzliste repräsentiert ist. Dann lassen sich die folgenden Probleme in Zeit $O(|V| + |E|)$ lösen:

(a) Ist G azyklisch?

- Jede Rückwärtskante schließt einen Kreis.
- Baum-, Vorwärts- und Querkanten allein können keinen Kreis schließen.
- G ist azyklisch genau dann, wenn G keine Rückwärtskanten hat.

(b) Führe eine topologische Sortierung durch.

- Führe eine Tiefensuche durch.
- G muß azyklisch sein, hat also nur Baum-, Vorwärts- und rechts-nach-links Querkanten.
- „Sortiere“ die Knoten absteigend nach ihrem Endewert:
  - ⋆ keine Kante führt von einem Knoten mit kleinem Endewert zu einem Knoten mit großem Endewert.

Anwendungen der Tiefensuche (2/2)

G ist **stark zusammenhängend**, wenn es für jedes Knotenpaar (u, v) einen Weg von u nach v gibt.

(c) Ist G stark zusammenhängend?

Es genügt zu zeigen, dass alle Knoten von Knoten 1 aus erreichbar sind und dass jeder Knoten auch Knoten 1 erreicht.

- Alle Knoten sind genau dann von Knoten 1 aus erreichbar, wenn während der Ausführung von tsuche(1) alle Knoten besucht werden.
- Kann jeder Knoten den Knoten 1 erreichen?
  - ⋆ Kehre die Richtung aller Kanten um,
  - ⋆ führe tsuche(1) auf dem neuen Graphen aus
  - ⋆ und überprüfe, ob alle Knoten besucht wurden.

Breitensuche
und die Bestimmung kürzester Wege

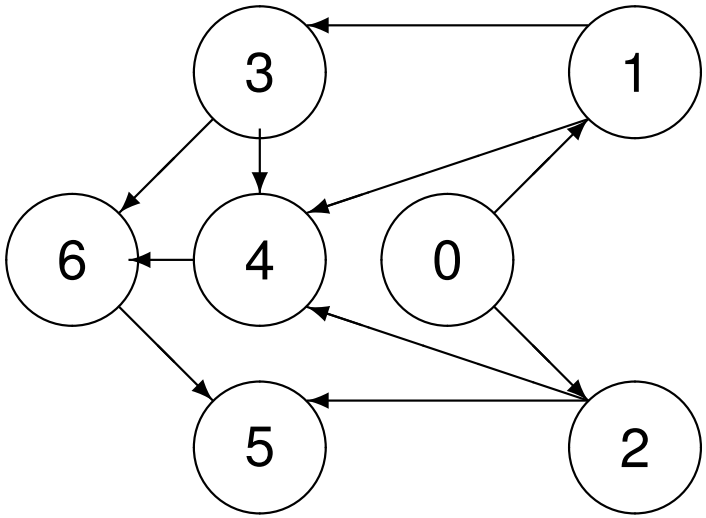
Breitensuche

Breitensuche für einen Knoten v soll zuerst v, dann die „Kindergeneration von v“, gefolgt von den „Enkelkindern“ und den „Urenkeln“ von v .... besuchen.

```
void Breitensuche(int v)

{Knoten *p; int w; queue q;

for (int k =0; k < n ; k++) besucht[k] = 0;
q.enqueue(v); besucht[v] = 1;

while (!q.empty ( ))
  {w = q. dequeue ( );
  for (p = A[w]; p != 0; p = p->next)
    if (!besucht[p->name])
      {q.enqueue(p->name); besucht[p->name] = 1;
      // (w,p->name) ist eine Baumkante. }}}
```

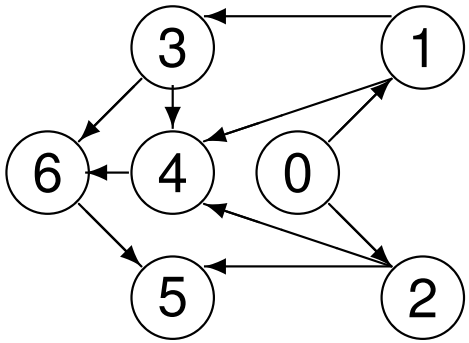
Ein Beispiel

Breitensuche(v) berechnet einen Baum mit Wurzel v, wenn wir alle Baumkanten einsetzen.



Wir beginnen sowohl Tiefensuche wie auch Breitensuche im Knoten 0. Wie sehen der Baum der Tiefensuche und der Baum der Breitensuche aus?

Baumkanten in der Breitensuche multiple choice



Betrachte die möglichen Abläufe der Breitensuche mit Startknoten 0.

Welche der folgenden Kanten ist dann niemals eine Baumkante – also für keine einzige Sortierung der Nachbarn?

- (1) Kante (1,4)
- (2) Kante (2,4)
- (3) Kante (3,4) $\surd$
- (4) Kante (3,6)
- (5) Kante (4,6)
- (6) Kante (6,5) $\surd$

Eigenschaften der Breitensuche (1/2)

Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph. Für Knoten w setze

$$V_w = \{u \in V \mid \text{Es gibt einen Weg von } w \text{ nach } u\}$$

und

$$E_w = \{e \in E \mid \text{beide Endpunkte von } e \text{ gehören zu } V_w\}.$$

(a) Breitensuche(w) besucht jeden Knoten in V_w genau einmal und sonst keinen anderen Knoten.

- Nur bisher nicht besuchte Knoten werden in die Schlange eingefügt, dann aber **sofort** als besucht markiert:
  - ⋆ Jeder Knoten wird höchstens einmal besucht.
- Bevor Breitensuche(w) terminiert, müssen alle von w aus erreichbaren Knoten besucht werden.

(b) Breitensuche(w) benötigt Zeit höchstens $O(|V_w| + |E_w|)$.

- Die Schlange benötigt Zeit höchstens $O(|V_w|)$, da genau die Knoten aus V_w eingefügt werden und zwar genau einmal.
- Jede Kante wird für jeden Endpunkt genau einmal in seiner Adjazenzliste „angefasst“. Insgesamt Zeit $O(|E_w|)$.

Eigenschaften der Breitensuche (2/2)

Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph.

Ein Baum $T(w)$ mit Wurzel w heißt ein **Baum kürzester Wege** für G, falls

(a) $T(w)$ die Knotenmenge V_w hat und

(b) falls für jeden Knoten $u \in V_w$,
der Weg in $T(w)$ von der Wurzel w nach u ein **kürzester Weg** ist.

- Zuerst werden Knoten u im Abstand 1 von w in die Schlange eingefügt:
  Die Kinder von w in $T(w)$ stimmen mit den Nachbarn von w in G überein.
- Zu jedem Zeitpunkt:
  - Wenn ein Knoten u im Abstand d von der Wurzel aus der Schlange entfernt wird, dann werden noch nicht besuchte Nachbarn von u (im Abstand $d + 1$ von w) eingefügt.
  - Breitensuche baut seinen Baum „Generation für Generation“ auf.

Der Baum der Breitensuche ist ein Baum kürzester Wege.

Breitensuche und kürzeste Wege

Der gerichtete oder ungerichtete Graph $G = (V, E)$ liege als Adjazenzliste vor. Dann können wir in Zeit $O(|V| + |E|)$ kürzeste Wege von einem Knoten w zu allen anderen Knoten bestimmen.

- Breitensuche(w) terminiert in Zeit $O(|V| + |E|)$.
- Der Baum der Breitensuche ist ein Baum kürzester Wege!
- Wir können somit sämtliche kürzesten Wege kompakt als einen Baum darstellen:
  - Implementiere den Baum als Eltern-Array.
  - Wir erhalten für jeden Knoten u einen kürzesten Weg von w nach u durch Hochklettern im Eltern-Array.

Dijkstra's Algorithmus

Kürzeste gewichtete Wege

Für einen gerichteten Graphen $G = (V, E)$, einen Knoten $s \in V$ und Kantenlängen

$$\text{länge} : E \to \mathbb{R}_{\geq 0},$$

bestimme kürzeste Wege von s zu allen anderen Knoten in V.

- Die Länge des Weges $p = (v_0, v_1, \ldots, v_m)$ ist
$$\text{länge}(p) = \sum_{i=1}^{m} \text{länge}\,(v_{i-1}, v_i).$$
- Breitensuche funktioniert nur, falls $\text{länge}(e) = 1$ für alle Kanten.

Warum nicht einen kürzesten Weg von s zu **einem** Zielknoten t suchen? Alle bekannten Algorithmen finden gleichzeitig kürzeste Wege zu allen Knoten.

Dijkstra's Algorithmus: Die Idee

Sei $S \subseteq V$ eine Teilmenge von V mit $s \in S$. Ein S-Weg startet in s und durchläuft mit Ausnahme des letzten Knotens nur Knoten in S.

Angenommen,

für jeden Knoten $w \in V \setminus S$ gilt die Invariante

distanz(w) = Länge eines kürzesten S-Weges von s nach w.

Intuition: Wenn der Knoten $w \in V \setminus S$ einen minimalen Distanzwert unter allen Knoten in $V \setminus S$ besitzt, dann ist

ein kürzester S-Weg nach w auch ein kürzester Weg nach w.

Warum ist die Idee richtig?

Die Invariante gelte. Wenn $\text{distanz}[w] = \min_{u \in V \setminus S} \text{distanz}[u]$ für einen Knoten $w \in V \setminus S$ ist, dann folgt

$$\text{distanz}[w] = \text{Länge eines kürzesten Weges von } s \text{ nach } w.$$

- Angenommen, der Weg $p = (s, v_1, \ldots, v_i, v_{i+1}, \ldots, w)$ ist kürzer als distanz$[w]$.
  - Sei v_{i+1} der erste Knoten in p, der nicht zu S gehört.
  - $(s, v_1, \ldots, v_i, v_{i+1})$ ist ein S-Weg nach v_{i+1} und

    $$\text{distanz}[v_{i+1}] \leq \text{länge}(s, v_1, \ldots, v_i, v_{i+1}) \leq \text{länge}(p) < \text{distanz}[w]$$

    folgt aus der Invariante, da Kantenlängen nicht-negativ sind.
- Aber

  $$\text{distanz}[w] \leq \text{distanz}[v_{i+1}],$$

  denn w hatte ja den kleinsten Distanz-Wert. Widerspruch.

Wie kann die Invariante aufrecht erhalten werden?

- Der Knoten w habe einen kleinsten distanz-Wert.
- Angenommen w wird zur Menge S hinzugefügt.

Wie sieht danach ein kürzester S-Weg $\mathcal{P}$ für $u \in V \setminus S$ aus?

- Wenn Knoten w nicht von $\mathcal{P}$ durchlaufen wird, dann ist die Länge eines kürzesten S-Wegs unverändert.
- Sonst ist $\mathcal{P} = (s, .., w, v, .., u)$ ein kürzester S-Weg. Wenn $v \neq u$:
  - v wurde vor w in S eingefügt.
  - Ein kürzester S-Weg $p(v)$ von s nach v durchläuft w deshalb nicht.
  - Ersetze das Anfangsstück $(s, \ldots, w, v)$ von $\mathcal{P}$ durch $p(v)$:

    Der neue S-Weg nach u ist mindestens so kurz wie $\mathcal{P}$, enthält aber w nicht.

Die neuen kürzesten S-Wege sind von der Form $(s, \ldots, w, u)$.

Dijkstra's Algorithmus

(1) Setze $S = \{s\}$ und

$$\text{distanz}[v] = \begin{cases} \text{länge}(s, v) & \text{wenn } (s, v) \in E, \\ \infty & \text{sonst.} \end{cases}$$

(2) Solange $S \neq V$ wiederhole

(2a) wähle einen Knoten $w \in V \setminus S$ mit kleinstem Distanz-Wert.

(2b) Füge w in S ein.

(2c) Berechne den neuen Distanz-Wert für jeden Nachfolger $u \in V \setminus S$ von w:

$$\begin{aligned} c &= \text{distanz}[w] + \text{länge}(w, u); \\ \text{distanz}[u] &= (\text{distanz}[u] > c) \; ? \; c : \text{distanz}[u]; \end{aligned}$$

// Aktualisiere distanz[u], wenn ein S-Weg über w nach u
// kürzer als der bisher kürzeste S-Weg nach u ist.

Und wo sind unsere kürzesten Wege?

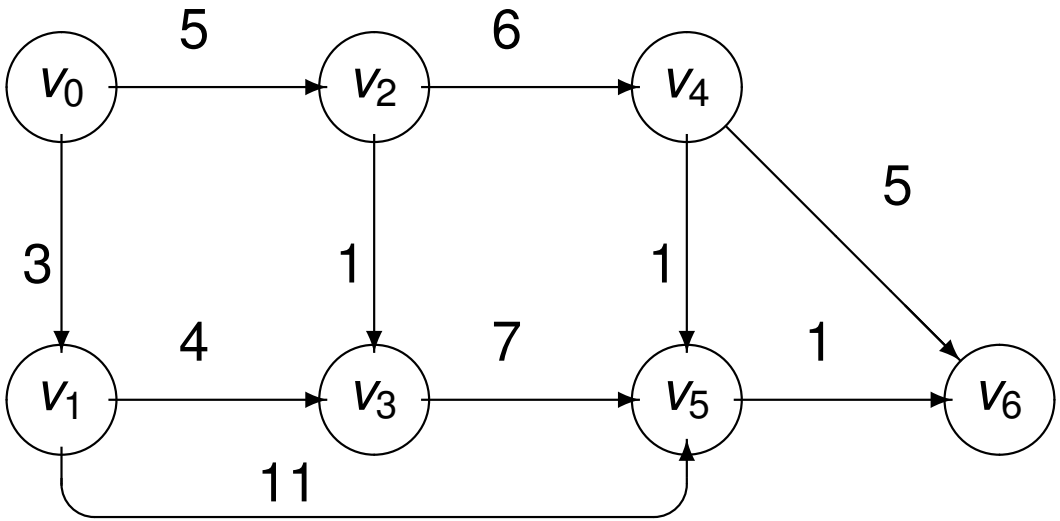
Wenn wir gerade den Knoten w in die Menge S aufgenommen haben und distanz[u] verringert sich, dann ...

- Vermerke, dass gegenwärtig ein kürzester S-Weg nach u den Knoten w besucht und dann zu u springt.
- Und wie sollen wir das vermerken?
  - Benutze einen Baum B, um kürzeste S-Wege zu speichern.
  - Implementiere B als Eltern-Array:

    Setze Vater[u] $= w$ genau dann, wenn sich der distanz-Wert von u verringert, wenn w zu S hinzugefügt wird.

Die Laufzeit steigt nur unmerklich an!

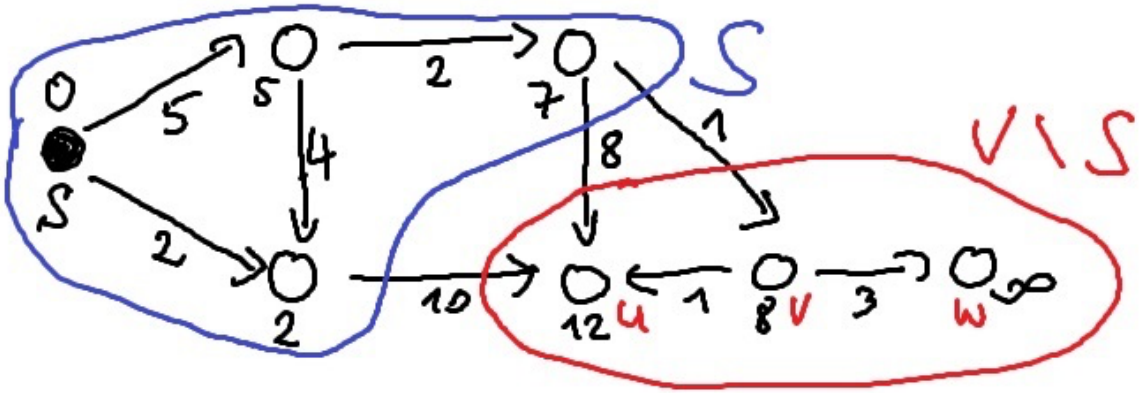
Ein Beispiel



Das Distanzarray:

	v_1	v_2	v_3	v_4	v_5	v_6
$S = \{v_0\}$	3	5	∞	∞	∞	∞
$S = \{v_0, v_1\}$	–	5	7	∞	14	∞
$S = \{v_0, v_1, v_2\}$	–	–	6	11	14	∞
$S = \{v_0, v_1, v_2, v_3\}$	–	–	–	11	13	∞
$S = \{v_0, v_1, v_2, v_3, v_4\}$	–	–	–	–	12	16
$S = \{v_0, v_1, v_2, v_3, v_4, v_5\}$	–	–	–	–	–	13

Dijkstra-Schritt demogr.



In welcher Reihenfolge werden die restlichen Knoten in S eingefügt?

- (1) u, v, w
- (2) v, u, w
- (3) w, v, u
- (4) u, w, v
- (5) w, u, v
- (6) v, w, u

Auflösung: (2) v, u, w

Dijkstra's Algorithmus: Datenstrukturen

Welche Datenstrukturen sollten wir einsetzen?

- Darstellung des Graphen G:
  - Wir implementieren G als **Adjazenzliste**, da wir dann sofortigen Zugriff auf die Nachfolger u von w im Aktualisierungschritt (2c) haben.
- Implementierung der Menge $V \setminus S$:
  - Wir benutzen einen Min-Heap, der die Knoten in $V \setminus S$ nach ihrem distanz-Wert verwaltet.
  - Ersetze die Funktion delete_max() durch die Funktion delete_min().
  - Implementiere den Aktualisierungschritt (2c) durch change_priority(wo, c). Woher kennen wir die Position wo?

Dijkstra's Algorithmus: Laufzeit

Wieviele Operationen werden benötigt?

- $|V| - 1$ mal wird nach einem Knoten in $V \setminus S$ mit kleinstem distanz-Wert gesucht.

  Es gibt höchstens $|V| - 1$ delete_min Operationen.

- Eine change_priority Operation kann nur durch eine Kante (w, u) hervorgerufen werden.

  Also gibt es höchstens $|E|$ change_priority Operationen.

Die Laufzeit ist durch $O((|V| + |E|) \cdot \log_2 |V|)$ beschränkt, denn jede Heap-Operation läuft in Zeit $O(\log_2 |V|)$.

Dijkstra's Algorithmus: Zusammenfassung

Die Laufzeit von Dijkstra's Algorithmus

Dijkstra's Algorithmus löst das „Single-Source-Shortest-Path" Problem für gerichtete Graphen mit n Knoten, m Kanten und nicht-negativen Gewichten in Zeit

$$O((n+m) \cdot \log_2 n).$$

- Was passiert, wenn Kantengewichte negativ sind?

  Wenn alle Kantengewichte mit -1 übereinstimmen, dann sind kürzeste, mit -1 gewichtete Wege längste (ungewichtete) Wege.

- Kürzeste (ungewichtete) Wege können wir blitzschnell mit Breitensuche bestimmen, längste (ungewichtete) Wege lassen sich hingegen nicht effizient bestimmen!

  Warum? Längste Wege zu bestimmen führt auf ein sog. NP-vollständiges Problem. (Mehr dazu in ALGO2)

Stabilität kürzester Wege demogr.

Was passiert mit einem Kürzeste-Wege Baum T, wenn bei jeder Kante des Ursprungsgraphen das Kantengewicht verdoppelt wird?

- (1) T kann sich ändern.
- (2) T bleibt immer gleich.

Auflösung: (2)

Und was passiert, wenn man zu jedem Kantengewicht eine Konstante $c > 0$ dazu addiert?

- (1) T kann sich ändern.
- (2) T bleibt immer gleich.

Auflösung: (1)

Minimale Spannbäume

Minimale Spannbäume

- Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph. Ein Baum $T = (V', E')$ heißt ein Spannbaum für G, falls $V' = V$ und $E' \subseteq E$.

  Spannbäume sind die kleinsten zusammenhängenden Teilgraphen, die alle Knoten von G beinhalten.

- Die Funktion

$$\text{länge} : E \to \mathbb{R}$$

  weist Kanten diesmal beliebige, auch negative Längen zu.

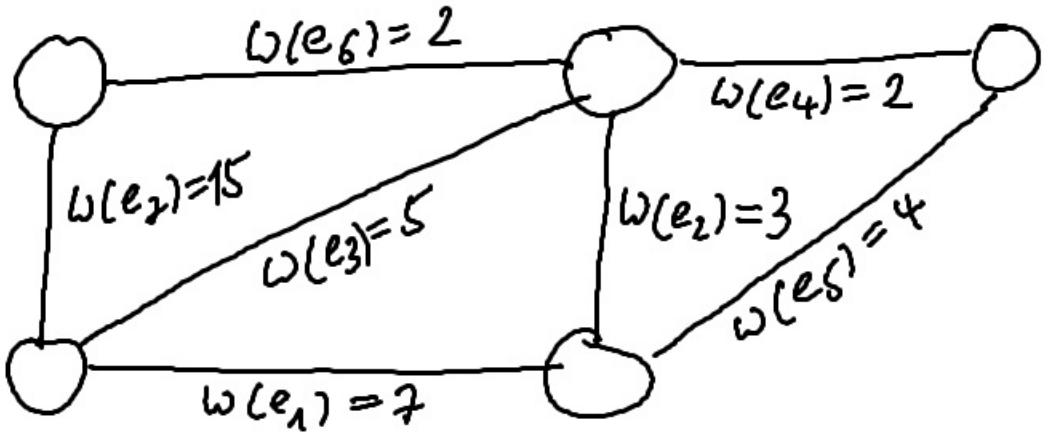
  - Für einen Spannbaum $T = (V, E')$ definieren wir

$$\text{länge}(T) = \sum_{e \in E'} \text{länge}(e)$$

    als die Länge von T.
  - Berechne einen minimalen Spannbaum T, also einen Spannbaum minimaler Länge unter allen Spannbäumen von G.

Beispiel Minimale Spannbäume multiple choice

Welche Kanten gehören nicht zum minimalen Spannbaum?



Auflösung: e_1, e_5 & e_7

Minimale Spannbäume: Anwendungen (1/2)

Ein Kommunikationsnetzwerk zwischen einer Menge von Zentralen ist zu errichten.

- Für die Verbindungen zwischen den Zentralen müssen Kommunikationsleitungen gekauft werden, deren Preis von der Distanz zwischen den zu verbindenden Zentralen abhängt.
  - Einzige Bedingung: Je zwei Zentralen müssen indirekt (über andere Zentralen) miteinander kommunizieren können.
- Wie sieht ein billigstes Kommunikationsnetzwerk aus?
  - Repräsentiere jede Zentrale durch einen eigenen Knoten.
  - Zwischen je zwei Knoten setzen wir eine Kante ein und markieren die Kante mit der Distanz der entsprechenden Zentralen.
  - Wenn G der resultierende Graph ist, dann müssen wir einen billigsten zusammenhängenden Teilgraphen, also einen minimalen Spannbaum für G suchen.

Minimale Spannbäume: Anwendungen (2/2)

Das Traveling Salesman Problem (TSP):

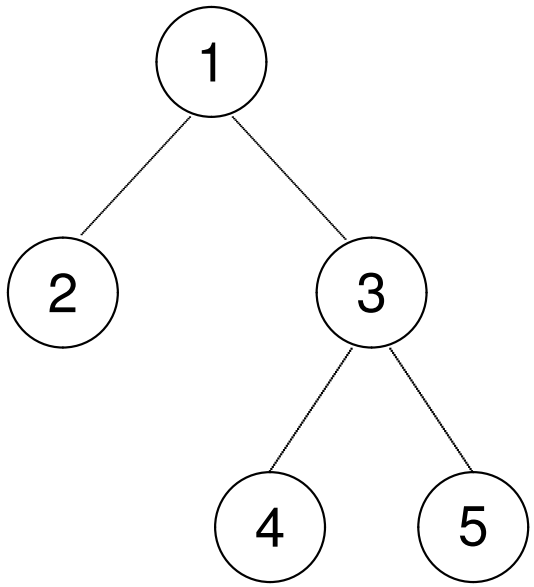
- Ein Handlungsreisender muss, von seinem Ausgangspunkt aus, eine Menge von Städten in einer Rundreise besuchen.
- **Das Ziel:** Bestimme eine Rundreise kürzester Länge. (Web)

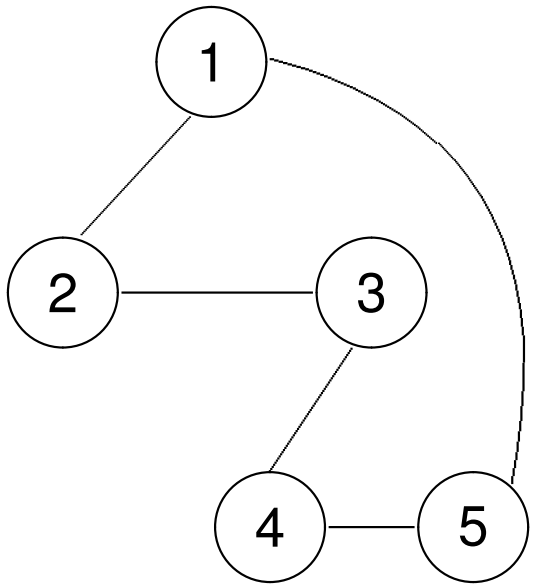
Die Heuristik des minimalen Spannbaums:

- Jeder Stadt wird ein Knoten zugeordnet.
- Für jedes Paar (s_1, s_2) von Städten wird eine Kante eingesetzt und mit der Distanz zwischen s_1 und s_2 beschriftet.

Bestimme einen minimalen Spannbaum T und durchlaufe T in Präorder-Reihenfolge. Wie lang ist die erhaltene Rundreise?

Eine Rundreise in Präorder-Reihenfolge





Wie lang ist unsere Rundreise höchstens?

$$d_{1,2} + (d_{2,1} + d_{1,3}) + d_{3,4} + (d_{4,3} + d_{3,5}) + (d_{5,3} + d_{3,1}).$$

Alle Kanten des Spannbaums treten genau zweimal auf. Es ist

$$\text{Länge}(T) \leq \text{minimale Länge einer Rundreise} \leq 2 \cdot \text{Länge}(T)$$

Unsere Rundreise ist höchstens doppelt so lang wie eine kürzeste Rundreise!

Kreuzende Kanten

Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph.
Die Kante $e = \{u, v\}$ kreuzt die Knotenmenge $S \subseteq V$, wenn genau ein Endpunkt von e in S liegt.

Kürzeste kreuzende Kanten

Der Wald $W = (V, E')$ sei in einem minimalen Spannbaum T für G enthalten.
Keine Kante von W möge die Menge S kreuzen.
Wenn e eine kürzeste S-kreuzende Kante ist, dann ist auch $W' = (V, E' \cup \{e\})$ in einem minimalen Spannbaum für G enthalten.

Warum? Wenn T die Kante e enthält, dann ist W' weiterhin in T enthalten.
Ansonsten:

- Füge e zu T hinzu. Ein Kreis wird geschlossen, der mindestens eine weitere S-kreuzende Kante e^* besitzt.
- länge$(e) \leq$ länge(e^*). Ersetze e^* in T durch e und wir erhalten einen Spannbaum, der W' enthält und nicht länger als T ist.

Minimale Spannbäume: Die Idee

Baue einen minimalen Spannbaum, indem kürzeste kreuzende Kanten für *geeignete* Knotenmengen $S \subseteq V$ eingesetzt werden.

Welche Knotenmengen S sind „geeignet“? Wir verfolgen zwei Ansätze.

- Der Algorithmus von Jarnik, Prim und Dijkstra lässt einen einzigen Baum „Kante für Kante“ wachsen.
  - Wenn der bisherige Baum die Knotenmenge S hat, dann wird eine kürzeste S-kreuzende Kante hinzugefügt.
- Der Algorithmus von Kruskal beginnt mit einem Wald aus Einzelknoten.
  - Die Kanten werden nach aufsteigender Länge sortiert und in dieser Reihenfolge abgearbeitet.
  - Füge Kante $e = \{u, v\}$ in den Wald ein, wenn kein Kreis geschlossen wird.
  - e kreuzt z.B. die Knotenmenge des Baums, der u enthält.

Der Algorithmus von Jarnik, Prim und Dijkstra

Der DJP-Algorithmus

(1) Setze $S = \{0\}$ und $E' = \emptyset$.
// Wir beginnen mit einem Baum, der nur aus dem Knoten 0
// besteht.

(2) Solange $S \neq V$, wiederhole:

(2a) Bestimme eine kürzeste S-kreuzende Kante $e = \{u, v\} \in E$ mit $u \in S$ and $v \in V \setminus S$. Es gilt also

$$\text{länge}(e) = \min\{\text{länge}(e') \mid e' \in E \text{ und } e' \text{ kreuzt } S\}.$$

(2b) Setze $S = S \cup \{v\}$ und $E' = E' \cup \{e\}$.
// Der bisherige Baum wird um die kürzeste kreuzende Kante
// erweitert. (Animation)

DJP-Algorithmus: Die Datenstruktur

Angenommen, wir kennen für jeden Knoten $v \in V \setminus S$ die Länge $e(v)$ einer kürzesten kreuzende Kante mit Endpunkt v.

- Benutze einen Min-Heap, um die Knoten $v \in V \setminus S$ gemäß ihren Prioritäten $e(v)$ zu verwalten: Berechne eine kürzeste kreuzende Kante mit Hilfe der delete_min-Operation.
- Wenn jetzt $w \in V \setminus S$ in die Menge S eingefügt wird,
  - können nur die Nachbarn w' von w eine neue kürzeste kreuzende Kante, nämlich die Kante $\{w, w'\}$, erhalten.
  - Für jeden Nachbarn w' von w setze deshalb

$$e(w') = (\ e(w') > \text{ länge } (\{w, w'\}))\ ?\ \text{länge}(\{w, w'\})\ :\ e(w');$$

wobei anfänglich

$$e(w) = \begin{cases} \text{länge}(\{0, w\}) & \{0, w\} \text{ ist eine Kante,} \\ \infty & \text{sonst.} \end{cases}$$

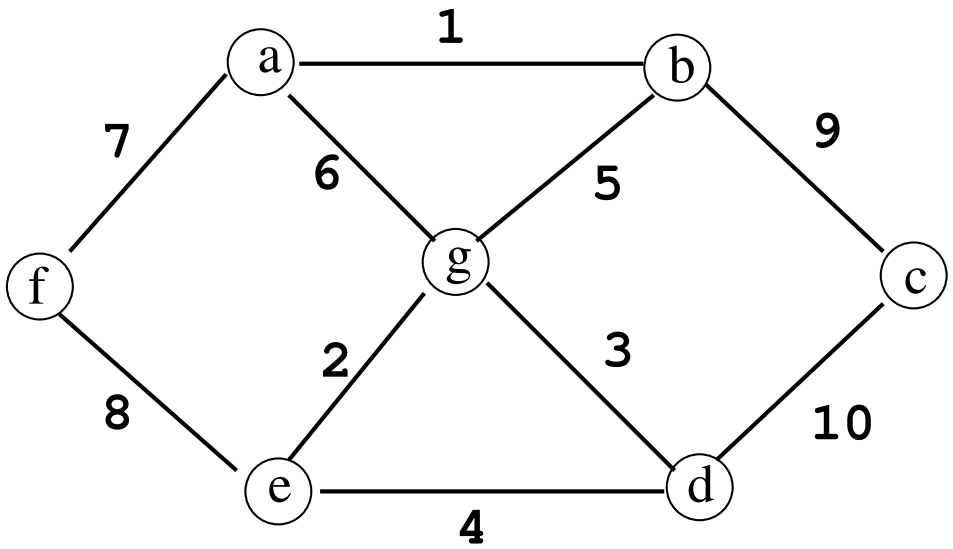
DJP-Algorithmus: Zusammenfassung

- Ein Spannbaum für einen Graphen mit n Knoten besteht aus $n-1$ Kanten.
  - Insgesamt sind somit genau $n-1$ delete_min-Operationen notwendig.
- Aktualisierungen der Kantenlängen $e(v)$ werden durch die mit v inzidenten Kanten erzwungen.
  - Also gibt es höchstens $m = |E|$ Aktualisierungen.
- Jede Heap-Operation, ob ein delete_min oder eine Aktualisierung, läuft in Zeit $O(\log_2 n)$.

DJP Algorithmus: Die Laufzeit

Der DJP-Algorithmus bestimmt einen minimalen Spannbaum für einen Graphen mit n Knoten und m Kanten in Zeit $O((n+m) \cdot \log_2 n)$.

DJP-Algorithmus: Ein Beispiel



Das Array e für den Startknoten g:

	a	b	c	d	e	f
$S = \{g\}$	6	5	∞	3	2	∞
$S = \{e, g\}$	6	5	∞	3	–	8
$S = \{d, e, g\}$	6	5	10	–	–	8
$S = \{b, d, e, g\}$	1	–	9	–	–	8
$S = \{a, b, d, e, g\}$	–	–	9	–	–	7
$S = \{a, b, d, e, f, g\}$	–	–	9	–	–	–

Der Algorithmus von Kruskal

Der Algorithmus von Kruskal

(1) Sortiere die Kanten gemäß aufsteigender Länge.
Sei $W = (V, E')$ der leere Wald, also $E' = \emptyset$.
// Kruskal beginnt mit dem leeren Wald W, der nur aus
// Einzelknoten besteht.

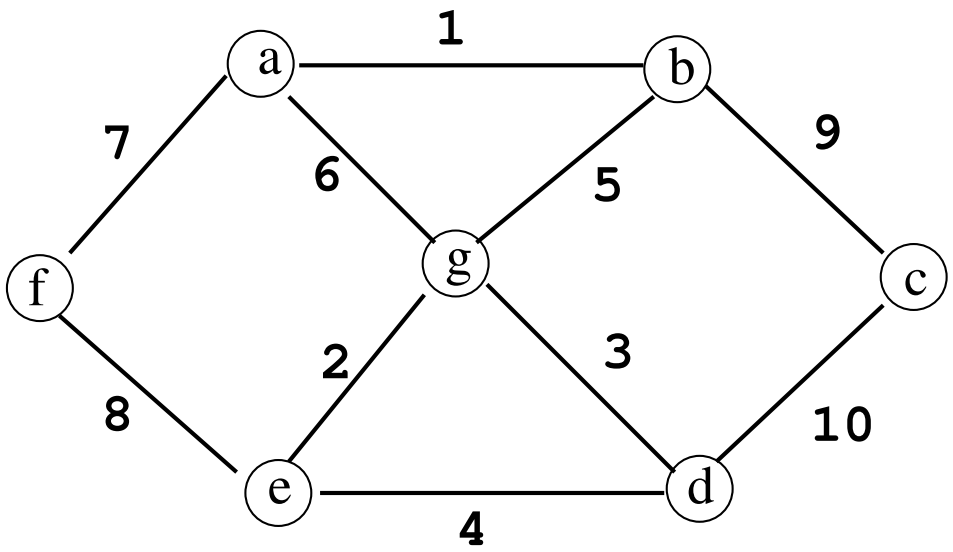
(2) Solange W kein Spannbaum ist, wiederhole

(2a) Nimm die gegenwärtige kürzeste Kante e und entferne sie aus der sortierten Folge.

(2b) Verwerfe e, wenn e einen Kreis in W schließt.
// Die Kante e verbindet zwei Knoten desselben Baums von W.

(2c) Ansonsten akzeptiere e und setze $E' = E' \cup \{e\}$.
// Die Kante e verbindet zwei Bäume T_1, T_2 von W.
// Sie wird zu W hinzugefügt. Wenn S die Knotenmenge von T_1 ist,
// dann ist e eine kürzeste S-kreuzende Kante.

Algorithmus von Kruskal: Ein Beispiel



Die Wälder in Kruskal's Algorithmus:

- Zu Anfang: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}$.
- $\{a, b\}$ mit dem neuen Wald $\{a, b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}$
- $\{e, g\}$ mit dem neuen Wald $\{a, b\}, \{c\}, \{d\}, \{e, g\}, \{f\}$
- $\{d, g\}$ mit dem neuen Wald $\{a, b\}, \{c\}, \{d, e, g\}, \{f\}$
- $\{b, g\}$ mit dem neuen Wald $\{a, b, d, e, g\}, \{c\}, \{f\}$
- $\{a, f\}$ mit dem neuen Wald $\{a, b, d, e, f, g\}, \{c\}$
- $\{b, c\}$ mit dem neuen Wald $\{a, b, c, d, e, f, g\}$

Algorithmus von Kruskal: Die Datenstruktur

- Wir überprüfen, ob die Kante $e = \{u, v\}$ einen Kreis schließt.
- Deshalb erfinden wir die union-find Datenstruktur, die die Operationen `union(i,j)` und `find(u)` unterstützt:
  - `find(u)` bestimmt die Wurzel des Baums, der u als Knoten besitzt.
  - `union(i,j)` vereinigt die Knotenmenge des Baums mit Wurzel i mit der Knotenmenge des Baums mit Wurzel j.

$e = \{u, v\}$ schließt genau dann keinen Kreis, wenn

$$\texttt{find(u)} \neq \texttt{find(v)}.$$

Ist dies der Fall, dann wende die union-Operation an:

Vereinige die Knotenmenge des Baums von u mit der Knotenmenge des Baums von v.

Die Union-Find Datenstruktur (1/2)

Die Grundidee: Implementiere

- die union-Operation durch ein Anhängen der Wurzel des einen Baums unter die Wurzel des anderen Baums und
- die find-Operation durch ein Hochklettern im Baum.

Aber wir können doch nicht einfach einen Baum an einen anderen anhängen: Der entstehende Wald entspricht nicht mehr dem von Kruskal berechneten Baum!

Wir benötigen zwei Datenstrukturen:

- Zuerst stellen wir genau den von Kruskal berechneten Baum dar.
- Die zweite Datenstruktur ist für die Unterstützung der union- und find-Operationen zuständig: Nur die Knotenmengen der einzelnen Bäume werden dargestellt.

Die Union-Find Datenstruktur (2/2)

- Wir stellen die Knotenmengen der einzelnen Bäume mit einem Eltern-Array dar: Anfänglich ist

$$\text{Vater}[u] = u \quad \text{für alle Knoten } u \in V.$$

- Generell: u ist genau dann eine Wurzel, wenn $\text{Vater}[u] = u$.

- Einen find-Schritt führen wir aus,
  - indem wir den Baum mit Hilfe des Eltern-Arrays hochklettern bis die Wurzel gefunden ist.
  - Die Zeit ist höchstens proportional zur Tiefe des Baums.
- Wie garantieren wir, dass die Bäume nicht zu tief werden?

Union-Operation demogr.

Wie garantieren wir, dass die Bäume nicht zu tief werden?

- (1) Hänge in einem Union-Schritt stets die Wurzel des kleineren Baumes unter die Wurzel des größeren Baumes.
- (2) Hänge in einem Union-Schritt stets die Wurzel des größeren Baumes unter die Wurzel des kleineren Baumes.
- (3) Die relative Ordnung ist egal.

Auflösung: (1)

Die Analyse eines Union-Schritts

Wir beobachten einen beliebigen Knoten v, während Kruskal einen minimalen Spannbaum aus einem Wald von Einzelbäumen baut.

- v beginnt mit Tiefe 0, denn v ist Wurzel seines eigenen „Einzelbäumchens“.
- Seine Tiefe vergrößert sich nur dann um 1, wenn v dem kleineren Baum in einer union-Operation angehört. Also:

  die Tiefe von v vergrößert sich nur dann um 1, wenn sich der Baum von v in der Größe mindestens verdoppelt.

Das Fazit

- Die Tiefe aller Bäume ist durch $\log_2(|V|)$ beschränkt.
- Eine union-Operation benötigt nur konstante Zeit, während ein find-Schritt höchstens logarithmische Zeit benötigt.

Algorithmus von Kruskal: Zusammenfassung

Es gibt höchstens zwei find-Operationen und eine union-Operation pro Kante, nämlich

- um herauszufinden, ob ein Kreis geschlossen wird (find),
- und um Bäume zu vereinigen (union), wenn kein Kreis geschlossen wird.

Algorithmus von Kruskal: Die Laufzeit

- Es gibt höchstens $2|E|$ viele find-Operationen und $n-1$ union-Operationen.
- Die Laufzeit des Algorithmus von Kruskal für einen Graphen mit n Knoten und m Kanten ist gegeben durch $O(m \cdot \log_2 m)$ für das Sortieren der Kanten und $O(n + m \cdot \log_2 n)$ für das Erstellen des Baumes. Insgesamt also $O(n + m \cdot \log_2 n)$.

Der “kleinere” Baum? multiple choice

Betrachte folgende Varianten der union-Operation. Dabei werden die Bäume wie folgt zusammengefügt:

Hänge den Baum

- (1) mit kleinerer Tiefe an den mit größerer Tiefe
- (2) mit weniger Knoten an den mit mehr Knoten
- (3) mit weniger Kanten an den mit mehr Kanten
- (4) mit kleinerem durchschnittlichem Knotengrad an den mit größerem durchschnittlichem Knotengrad

Welche dieser Varianten garantieren eine Laufzeit von $O(\log |V|)$ für die find-Operation?

Auflösung: (1), (2), (3), (4)

Kürzeste Wege vs. Minimale Spannbäume demogr.

Bei ganzzahligen Kantengewichten in $\{1, \ldots, n\}$ sei für einen zusammenhängenden Graph G:

$K =$ Summe der Kantengewichte in einem Kürzeste-Wege-Baum
$M =$ Summe der Kantengewichte in einem minimalen Spannbaum

Wie groß kann das Verhältnis K/M werden?

- (1) $\Theta(1)$
- (2) $\Theta(\log n)$
- (3) $\Theta(\sqrt{n})$
- (4) $\Theta(n)$
- (5) $\Theta(n^2)$

Auflösung: (4) $\Theta(n)$