Theoretische Informatik I

Prof. Dr. Andreas Goerdt
Professur Theoretische Informatik
Technische Universität Chemnitz

WS 2013/2014

Bitte beachten:

Beim vorliegenden Skript handelt es sich um eine vorläufige, unvollständige Version nach handschriftlicher Vorlage. Ergänzungen und nachfolgende Korrekturen sind möglich.

Version vom 9. Oktober 2013

Inhaltsverzeichnis

1 Gerichtete Graphen **5**

1.1 Datenstruktur Adjazenzmatrix 9

1.2 Datenstruktur Adjazenzliste 10

1.3 Breitensuche 12

1.4 Datenstruktur Schlange 13

1.5 Algorithmus Breitensuche (Breadth first search = BFS) 14

1.6 Topologische Sortierung 21

1.7 Algorithmus Top Sort 22

1.8 Algorithmus Verbessertes Top Sort 24

2 Ungerichtete Graphen **27**

2.1 Algorithmus Breitensuche (BFS) 31

2.2 Algorithmus (Finden von Zusammenhangskomponenten) 37

3 Zeit und Platz **38**

4 Tiefensuche in gerichteten Graphen **43**

4.1 Algorithmus Tiefensuche 44

5 Starke Zusammenhangskomponenten **55**

5.1 Algorithmus Starke Komponenten 60

6 Zweifache Zusammenhangskomponenten **66**

6.1 Berechnung von l[v] 74

6.2 Algorithmus (l-Werte) 75

6.3 Algorithmus (Zweifache Komponenten) 76

7 Minimaler Spannbaum und Datenstrukturen **78**

7.1 Algorithmus Minimaler Spannbaum 82

7.2 Datenstruktur Union-Find-Struktur 85

7.3 Algorithmus Union-by-Size 88
7.4 Algorithmus Wegkompression 91
7.5 Algorithmus Minimaler Spannbaum nach Prim 1963 94
7.6 Algorithmus (Prim mit Q in heap) 99

8 Kürzeste Wege 102

8.1 Algorithmus (Dijkstra 1959) 104
8.2 Algorithmus (Dijkstra ohne mehrfache Berechnung desselben D[w]) . 106
8.3 Algorithmus Floyd Warshall 112

9 Flüsse in Netzwerken 115

9.1 Algorithmus (Ford Fulkerson) 119

10 Kombinatorische Suche und Rekursionsgleichungen 122

10.1 Algorithmus (Erfüllbarkeitsproblem) 124
10.2 Algorithmus (Davis-Putnam) 127
10.3 Algorithmus (Pure literal rule, unit clause rule) 128
10.4 Algorithmus (Monien, Speckenmeyer) 133
10.5 Algorithmus 137
10.6 Algorithmus (Backtracking für TSP) 141
10.7 Algorithmus (Offizielles branch-and-bound) 145
10.8 Algorithmus (TSP mit dynamischem Programmieren) 149
10.9 Algorithmus (Lokale Suche bei KNF) 152

11 Divide-and-Conquer und Rekursionsgleichungen 158

11.1 Algorithmus: Quicksort 160
11.2 Multiplikation großer Zahlen 164

Vorwort Diese Vorlesung ist eine Nachfolgeveranstaltung zu Algorithmen und Programmierung im 1. Semester und zu Datenstrukturen im 2. Semester.

Theoretische Informatik I ist das Gebiet der effizienten Algorithmen, insbesondere der Graphalgorithmen und algorithmischen Techniken. Gemäß dem Titel „Theoretische Informatik“ liegt der Schwerpunkt der Vorlesung auf beweisbaren Aussagen über Algorithmen.

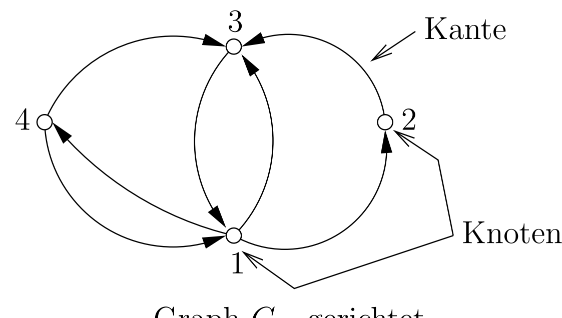
Der Vorlesung liegen die folgenden Lehrbücher zugrunde:

Schöning:	Algorithmik. Spektrum Verlag 2000
Corman, Leiserson, Rivest:	Algorithms.[1] The MIT Press 1990
Aho, Hopcroft, Ullmann:	The Design and Analysis of Computer Algorithms. Addison Wesley 1974
Aho, Hopcroft, Ullmann:	Data Structures and Algorithms. Addison Wesley 1989
Ottman, Widmayer:	Algorithmen und Datenstrukturen. BI Wissenschaftsverlag 1993
Heun:	Grundlegende Algorithmen. Vieweg 2000

Besonders die beiden erstgenannten Bücher sollte jeder Informatiker einmal gesehen (d.h. teilweise durchgearbeitet) haben.

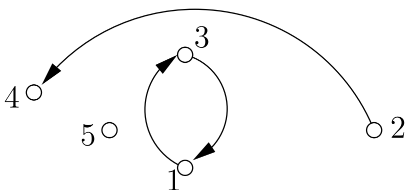
[1] Auf Deutsch im Oldenburg Verlag

1 Gerichtete Graphen

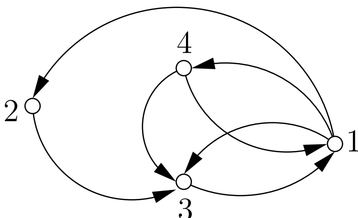


Graph G_1, gerichtet

Ein gerichteter Graph besteht aus einer Menge von Knoten (vertex, node) und einer Menge von Kanten (directed edge, directed arc).



Hier ist die Menge der Knoten $V = \{1, 2, 3, 4, 5\}$ und die Menge der Kanten $E = \{(3, 1), (1, 3), (2, 4)\}$.



Hierbei handelt es sich um den gleichen Graphen wie im ersten Bild. Es ist $V = \{1, 2, 3, 4\}$ und $E = \{(4, 1), (1, 4), (3, 1), (1, 3), (4, 3), (1, 2), (2, 3)\}$.

Definition 1.1(gerichteter Graph): *Ein gerichteter Graph besteht aus zwei Mengen:*

- V*, eine beliebige, endliche Menge von Knoten*
- E*, eine Menge von Kanten wobei* $E \subseteq \{(u, v) \mid u, v \in V, u \neq v\}$

Schreibweise: $G = (V, E)$*.*

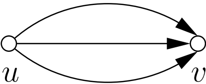
Beim gerichteten Graphen sind die Kanten geordnete Paare von Knoten. Die Kante (u, v) ist somit von der Kante (v, u) verschieden.

(u, v) u v (v, u) v u

In der Vorlesung *nicht* betrachtet werden Schleifen (u, u):

(u, u) u

und Mehrfachkanten $(u, v)(u, v)(u, v)$:



Folgerung 1.1: *Ist $|V| = n$, so gilt: Jeder gerichtete Graph mit Knotenmenge V hat $\leq n \cdot (n-1)$ Kanten. Es ist $n \cdot (n-1) = n^2 - n$ und damit $O(n^2)$.*

Ein Beweis der Folgerung folgt auf Seite 8.

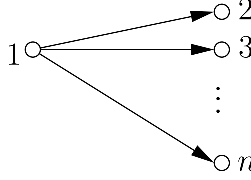
Erinnerung: *$f(n)$ ist $O(n^2)$ bedeutet: Es gibt eine Konstante $C > 0$, so dass $f(n) \leq C \cdot n^2$ für* alle *hinreichend großen n gilt.*

- Ist eine Kante (u, v) in G vorhanden, so sagt man: v ist adjazent zu u.
- Ausgangsgrad$(v) = \big|\{(v, u) \mid (v, u) \in E\}\big|$.
- Eingangsgrad$(u) = \big|\{(v, u) \mid (v, u) \in E\}\big|$.

Für eine Menge M ist $\#M = |M| =$ die Anzahl der Elemente von M.

Es gilt immer:

$$\begin{array}{lll} 0 \leq & \text{Agrad}(v) & \leq n-1 \quad \text{und} \\ 0 \leq & \text{Egrad}(u) & \leq n-1 \end{array}$$

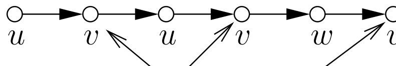


Egrad$(1) = 0$, Agrad$(1) = n - 1$

Definition 1.2(Weg): *Eine Folge von Knoten* $(v_0, v_1, ..., v_k)$ *ist ein Weg in* $G = (V, E)$ *gdw. (genau dann, wenn)* $v_0 \to v_1 \to v_2 \to \cdots \; v_{k-1} \to v_k$ *Kanten in* E *sind.*

Die Länge von $v_0 \to v_1 \to v_2 \to \cdots \; v_{k-1} \to v_k$ ist k. Also die Länge ist die Anzahl der Schritte. Die Länge von $v_0 = 0$.

Ist $E = \{(u, v), (v, w), (w, v), (v, u)\}$, so ist auch



ein Weg. Seine Länge ist 5.

- Ein Weg $(v_0, v_1, ..., v_k)$ ist *einfach* genau dann, wenn $|\{v_0, v_1, ..., v_k\}| = k + 1$ (d.h., alle v_i sind verschieden).

- Ein Weg $(v_0, v_1, ..., v_k)$ ist *geschlossen* genau dann, wenn $v_0 = v_k$.

- Ein Weg $(v_0, v_1, ..., v_k)$ ist ein *Kreis* genau dann, wenn:
  - $k \geq 2$ und
  - $(v_0, v_1, ..., v_{k-1})$ einfach und
  - $v_0 = v_k$

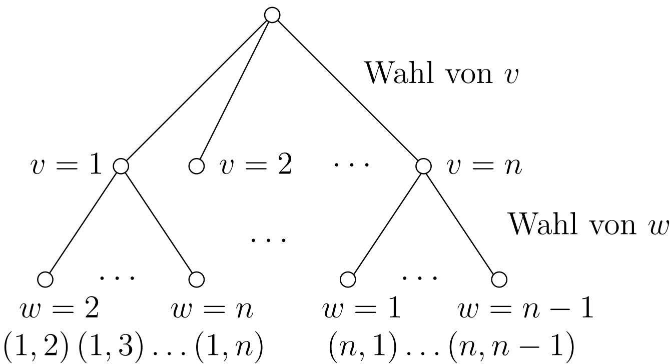
Es ist z.B. $(1, 2, 1)$ ein Kreis der Länge 2 mit $v_0 = 1, v_1 = 2$ und $v_2 = 1 = v_0$. Beachte noch einmal: Im Weg $(v_0, v_1, ..., v_k)$ haben wir $k+1$ v_i's aber nur k Schritte (v_i, v_{i+1}).

Die Kunst des Zählens ist eine unabdingbare Voraussetzung zum Verständnis von Laufzeitfragen. Wir betrachten zwei einfache Beispiele:

1. Bei $|V| = n$ haben wir insgesamt genau $n \cdot (n - 1)$ gerichtete Kanten.

   Eine Kante ist ein Paar (v, w) mit $v, w \in V, v \neq w$.

   # Möglichkeiten für v: n
   Ist v gewählt, dann # Möglichkeiten für w: $n - 1$
   Jede Wahl erzeugt genau eine Kante. Jede Kante wird genau einmal erzeugt.
   Multiplikation der Möglichkeiten ergibt: $n \cdot (n - 1)$



Veranschaulichung durch „Auswahlbaum" für Kante (v, w)

Jedes Blatt = genau eine Kante, # Blätter $= n \cdot (n - 1) = n^2 - n$.

Alternative Interpretation von $n^2 - n$:

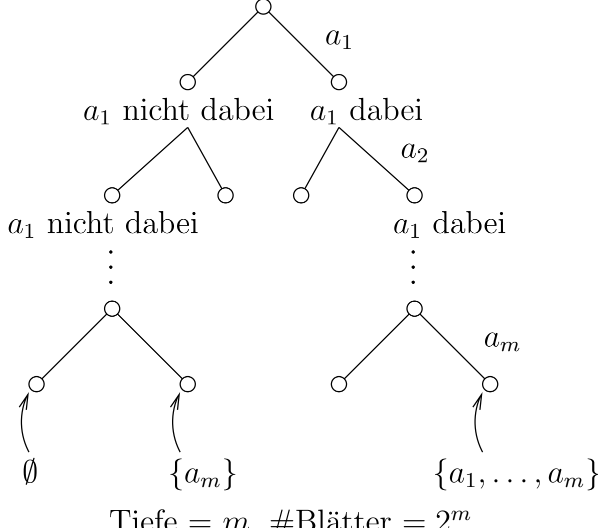
- $n^2 = \#$ aller Paare (v, w), $v, w \in V$ (auch $v = w$) (Auswahlbaum)
- $n = \#$ Paare (v, v) $v \in V$. Diese werden von n^2 abgezogen.

2. Bei einer Menge M mit $|M| = m$ haben wir genau 2^m Teilmengen von M. Teilmenge von M = Bitstring der Länge m, $M = \{a_1, a_2, ..., a_m\}$

$$\begin{array}{lcl}
000\dots00 & \to & \emptyset \\
000\dots01 & \to & \{a_m\} \\
000\dots10 & \to & \{a_{m-1}\} \\
000\dots11 & \to & \{a_{m-1}, a_m\} \\
& \vdots & \\
111\dots11 & \to & \{a_1, a_2, ..., a_m\} = M
\end{array}$$

2^m Bitstrings der Länge m. Also # gerichtete Graphen bei $|V| = n$ ist genau gleich $2^{n(n-1)}$. Ein Graph ist eine Teilmenge von Kanten.

Noch den Auswahlbaum für die Teilmengen von M:



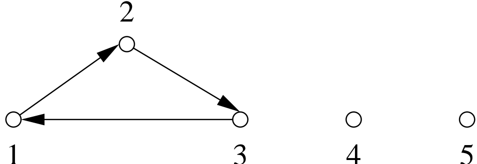
Tiefe $= m$, #Blätter $= 2^m$

1.1 Datenstruktur Adjazenzmatrix

Darstellung von $G = (V, E)$ mit $V = \{1, ..., n\}$ als Matrix A.

$$\begin{aligned} A &= (a_{u,v}),\ 1 \leq u \leq n,\ 1 \leq v \leq n,\ a_{u,v} \in \{0,1\} \\ a_{u,v} &= \begin{cases} 1, \text{ wenn } (u,v) \in E \\ 0, \text{ wenn } (u,v) \notin E. \end{cases} \end{aligned}$$

Implementierung durch 2-dimensionales Array $A[\,][\,]$.

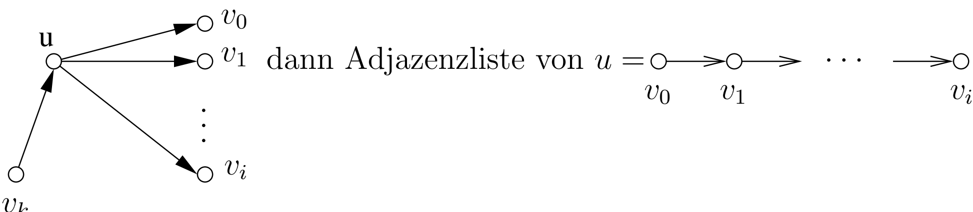


$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

A hat immer n^2 Speicherplätze, egal wie groß $|E|$ ist. Jedes $|E| \geq \frac{n}{2}$ ist sinnvoll möglich, denn dann können n Knoten berührt werden.

1.2 Datenstruktur Adjazenzliste

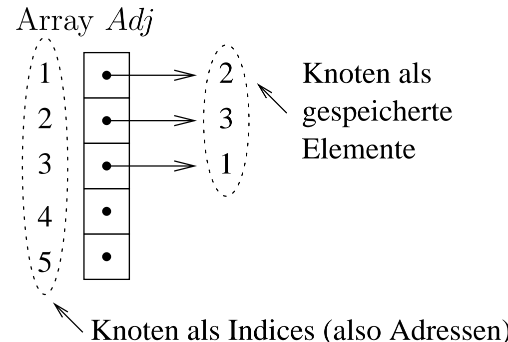
Sei $G = (V, E)$ ein gerichteter Graph. Adjazenzliste von u = Liste der direkten Nachbarn von u



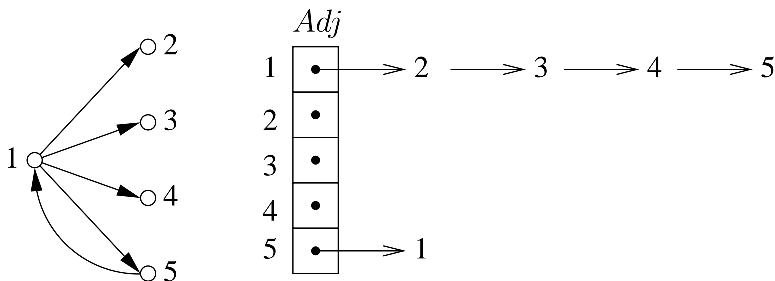
Nicht in der Adjazenzliste ist v_k, jede Reihenfolge von $v_1, ..., v_i$ erlaubt.

Adjazenzlistendarstellung von G = Array aus $Adj[\,]$, dessen Indices für $v \in V$ stehen, $Adj[v]$ zeigt auf Adjazenzliste von v.

Beispiel wie oben:



ein anderes Beispiel:

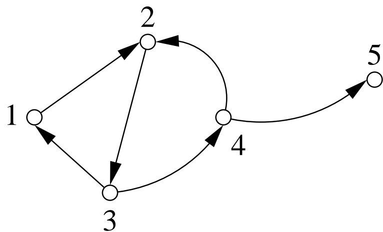


Platz zur Darstellung von $G = (V, E)$: $c + n + d \cdot |E|$ Speicherplätze mit

c:	Initialisierung von A (konstant)
n:	Größe des Arrays A
$d \cdot \lvert E \rvert$:	jede Kante einmal, d etwa 2, pro Kante 2 Plätze

Also $O(n+|E|)$ Speicherplatz, $O(|E|)$ wenn $|E| \geq \frac{n}{2}$. Bei $|E| < \frac{n}{2}$ haben wir isolierte Knoten, was nicht sinnvoll ist.

Was, wenn keine Pointer? Adjazenzlistendarstellung in Arrays. An einem Beispiel:

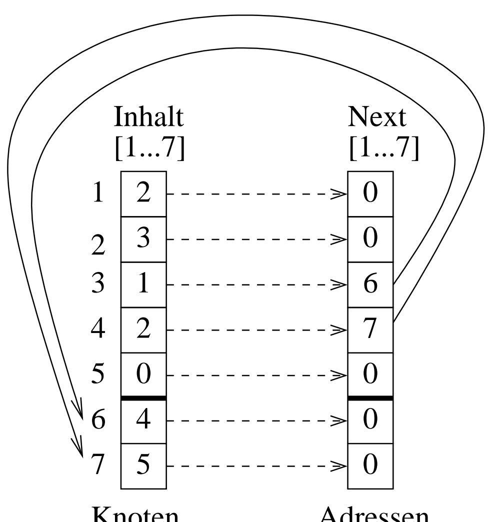


Die Größe des Arrays ist nicht unbedingt gleich der Kantenanzahl, aber sicherlich $\leq |E| + |V|$. Hier ist $A[1 \ldots 6]$ falsch. Wir haben im Beispiel $A[1 \ldots 7]$. Die Größe des Arrays ist damit $|E|$ + #Knoten vom Agrad = 0.

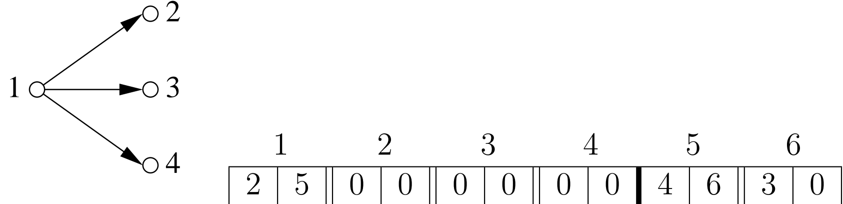
1		2		3		4		5		6		7	
2	0	3	0	1	6	2	7	0	0	4	0	5	0

Erklärung:

> Position 1-5 repräsentieren die Knoten, 6 und 7 den Überlauf. Knoten 1 besitzt eine ausgehende Kante nach Knoten 2 und keine weitere, Knoten 2 besitzt eine ausgehende Kante zu Knoten 3 und keine weitere. Knoten 3 besitzt eine ausgehende Kante zu Knoten 1 und eine weitere, deren Eintrag an Position 6 zu finden ist. Dieser enthält Knoten 4 sowie die 0 für „keine weitere Kante vorhanden“. Alle weiteren Einträge erfolgen nach diesem Prinzip. Das Ganze in zwei Arrays:



Ein weiteres Beispiel:

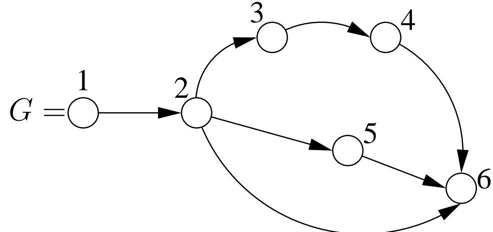


Anzahl Speicherplatz sicherlich immer $\leq 2 \cdot |V| + 2 \cdot |E|$, also $O(|V| + |E|)$ reicht aus.

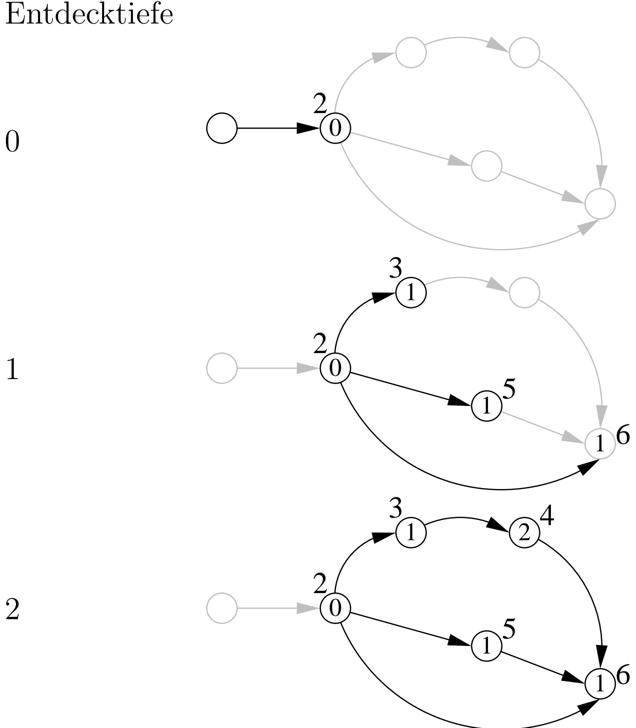
1.3 Breitensuche

Gibt es einen Weg von u nach v im gerichteten Graphen $G = (V, E)$? Algorithmische Vorgehensweise: Schrittweises Entdecken der Struktur.

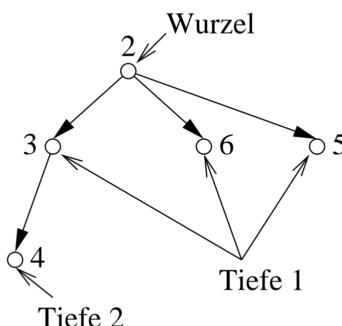
Dazu ein Beispiel:



Weg von 2 nach 4 (4 von 2 erreichbar)?



Definition 1.3(Breitensuchbaum): *Der Breitensuchbaum ist ein Teilgraph von G und enthält alle Kanten, über die entdeckt wurde:*



Die Breitensuche implementiert man zweckmäßigerweise mit einer Schlange.

1.4 Datenstruktur Schlange

Array $Q[1..n]$ mit Zeigern in Q hinein, *head*, *tail*. Einfügen am Ende (*tail*), Löschen vorne (*head*).

	1	2	3	4	
5 einfügen:	5				$head = 1$, $tail = 2$
7 und 5 einfügen:	5	7	5		$head = 1$, $tail = 4$
Löschen und 8 einfügen:		7	5	8	$head = 2$, $tail = 1$
Löschen:			5	8	$head = 3$, $tail = 1$
2 mal löschen:					$head = 1$, $tail = 1$

Schlange leer!

Beachte: Beim Einfügen immer $tail + 1 \neq head$, sonst geht es nicht mehr, *tail* ist immer der nächste freie Platz, $tail+1$ wird mit „Rumgehen“um das Ende des Arrays berechnet.

$tail < head$	$\iff$	Schlange geht ums Ende
$tail = head$	$\iff$	Schlange leer

First-in, first-out (FIFO) = Schlange.

Am Beispiel von G oben, die Schlange im Verlauf der Breitensuche:

Bei $|V| = n \geq 2$ reichen n Plätze. Einer bleibt immer frei.

Startknoten rein: $head = 1$, $tail = 2$

1	2	3	4	5	6
2					

3, 6, 5 rein, 2 raus: $head = 2$, $tail = 5$

1	2	3	4	5	6
	3	6	5		

4 rein, 3 raus: $head = 3$, $tail = 6$

1	2	3	4	5	6
		6	5	4	

5, 6 hat Entdecktiefe 1, 4 Entdecktiefe 2.

6 raus nichts rein: $head = 4$, $tail = 2$

1	2	3	4	5	6
			5	4	

5, 4 raus, Schlange leer. $head = tail = 6$

1.5 Algorithmus Breitensuche (Breadth first search = BFS)

Wir schreiben BFS(G, s). Eingabe: $G = (V, E)$ in Adjazenzlistendarstellung, $|V| = n$ und $s \in V$ der Startknoten.

Datenstrukturen:

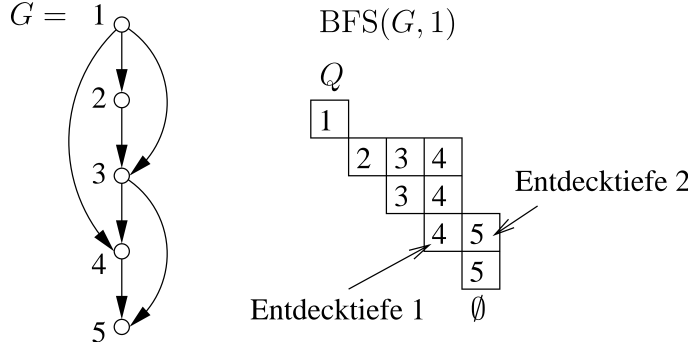
- Schlange $Q = Q[1..n]$ mit *head* und *tail* für entdeckte, aber noch nicht untersuchte Knoten.
- Array $col[1..n]$, um zu merken, ob Knoten bereits entdeckt wurde.

$$col[u] = \begin{cases} \text{weiß} \Leftrightarrow u \,\text{noch nicht entdeckt,} \\ \text{grau} \Leftrightarrow u \,\text{entdeckt, aber noch nicht abgeschlossen, d.h.}\ u \,\text{in Schlange,} \\ \text{schwarz} \Leftrightarrow u \,\text{entdeckt und abgearbeitet.} \end{cases}$$

Algorithmus 1: BFS(G, s)

```
   /* Initialisierung */
 1 foreach u ∈ V do
 2 |  col[u] = weiß;
 3 end
 4 col[s] = grau;
 5 Q = (s) ;                                  /* s ist Startknoten */
   /* Abarbeitung der Schlange */
 6 while Q ≠ () do                        /* Testbar mit tail ≠ head */
 7 |  u = Q[head] ;                 /* u wird bearbeitet (expandiert) */
 8 |  foreach v ∈ Adj[u] do
 9 |  |  if col[v] == weiß then                    /* v wird entdeckt */
10 |  |  |  col[v] = grau;
11 |  |  |  v in Schlange einfügen              /* Schlange immer grau */
12 |  |  end
13 |  end
14 |  u aus Q entfernen;
15 |  col[u] = schwarz ;          /* Knoten u ist fertig abgearbeitet */
16 end
```

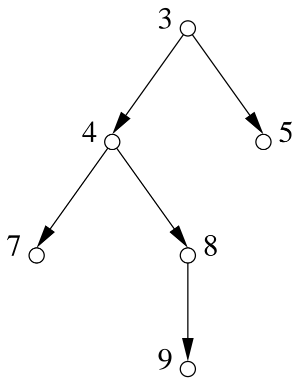
Noch ein Beispiel:



Zusätzlich: Array $d[1..n]$ mit $d[u]$ = Entdecktiefe von u. Anfangs $d[s] = 0$, $d[u] = \infty$ für $u \neq s$. Wird v über u entdeckt, so setzen wir $d[v] := d[u] + 1$ (bei Abarbeitung von u).

Zusätzlich: Breitensuchbaum durch Array $\pi[1..n]$. $\pi[s] = nil$, da s Wurzel. Wird v über u entdeckt, dann $\pi[v] := u$.

Interessante Darstellung des Baumes durch π: Vaterarray $\pi[u]$ = Vater von u. Wieder haben wir Knoten als Indices und Inhalte von π.



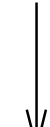
Das Array π hat die Einträge $\pi[9] = 8$, $\pi[8] = 4$, $\pi[4] = 3$, $\pi[5] = 3$, $\pi[7] = 4$. $\pi[u] = v \Rightarrow$ Kante (v, u) im Graph. Die Umkehrung gilt nicht.

Verifikation? Hauptschleife ist Zeile 6-15. Wie läuft diese?

$Q_0 = (s), col_0[s] =$ grau, „$col_0 =$ weiß“ sonst.

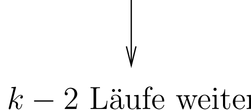
1. Lauf

$Q_1 = (\underbrace{u_1, \ldots, u_k}_{Dist=1}), col_1[s] =$ schwarz, $col_1[u_i] =$ grau, weiß sonst. $Dist(s, u_j) = 1$, $(u_1, \ldots, u_k)$ sind *alle* mit $Dist = 1$.



$Q_2 = (\underbrace{u_2, \ldots, u_k}_{Dist1}, \underbrace{u_{k+1}, \ldots, u_t}_{Dist2})$ $col[u_1] =$ schwarz

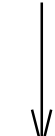
Alle mit $Dist = 1$ entdeckt.



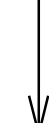
$k - 2$ Läufe weiter

$Q_k = (\underbrace{u_k}_{Dist1}, \underbrace{u_{k+1} \ldots}_{Dist2})$

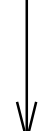
$Q_{k+1} = (\underbrace{u_{k+1}, \dots, u_t}_{Dist2}), u_{k+1} \dots u_t$ sind *alle* mit $Dist2$ (da alle mit $Dist = 1$ bearbeitet).



$Q_{k+2}(\underbrace{\dots}_{Dist=2}, \underbrace{\dots}_{Dist=3})$



$(\underbrace{\dots}_{Dist=3})$ und *alle* mit $Dist = 3$ erfasst (grau oder schwarz).



Allgemein: Nach l-tem Lauf gilt:

Schleifeninvariante: Falls $Q_l \neq ()$, so:

- $Q_l = (\overbrace{\underbrace{u_1, \dots,}_{DistD}}^{U} \overbrace{\underbrace{\dots}_{DistD+1}}^{V})$
  $D_l = Dist(s, u_1), u_1$ vorhanden, da $Q_l \neq ()$.
- *Alle* mit $Dist \leq D_l$ erfasst (*)
- *Alle* mit $Dist = D_l + 1$ (= weiße Nachbarn von U) $\cup$ V. (**)

Beweis. Induktion über l.

$l = 0$ (vor erstem Lauf): Gilt mit $D_0 = 0, U = (s), V = \emptyset$.

$l = 1$ (nach erstem Lauf): Mit $D_1 = 1, U =$ Nachbarn von $s, V = \emptyset$.

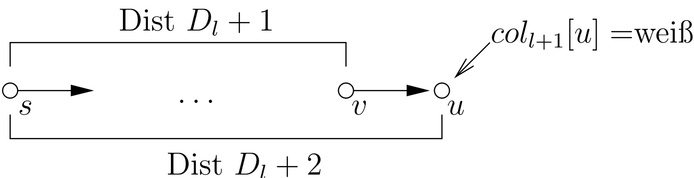
Gilt Invariante nach l-tem Lauf, also

$$Q_l = (\overbrace{u_1, \dots, u_k}^{DistD_l}, \overbrace{v_1, \dots, v_r}^{DistD_l+1}) \text{ mit (*), (**)}, \quad D_l = Dist(s, u_1).$$

Findet $l + 1$-ter Lauf statt (also v_1 expandieren).

1. Fall: $u_1 = u_k$

- $Q_{l+1} = (v_1, \dots, v_r \overbrace{\dots}^{\text{Nachbarn von } u_1}), D_l + 1 = Dist(s, v_1)$
- Wegen (**) nach l gilt jetzt Q_{l+1} = alle die Knoten mit $Dist = D_l + 1$
- Also gilt jetzt (*), wegen (*) vorher
- Also gilt auch (**), denn:

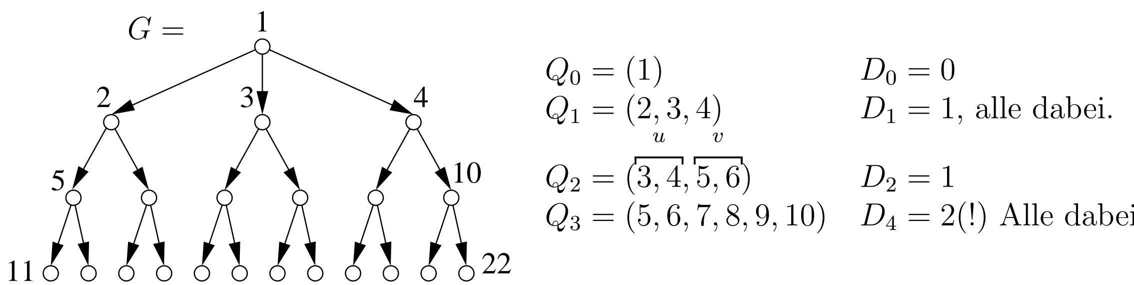


2. Fall: $u_1 \neq u_k$

- $Q_{l+1} = (u_2 \dots u_k, v_1 \dots v_r \overbrace{\dots}^{\text{weiße Nachbarn von } u_1}),\ D_{l+1} = Dist(s, u_2) = D_l$.
- (*) gilt, da (*) vorher
- Es gilt auch (**), da weiße Nachbarn von u_1 jetzt in Q_{l+1}.

Aus 1. Fall und 2. Fall für beliebiges $l \geq 1$ Invariante nach l-tem Lauf $\Longrightarrow$ Invariante nach $l+1$-tem Lauf. Also Invariante gilt nach jedem Lauf.

Etwa:



$\square$

Quintessenz (d.h. das Ende): Vor letztem Lauf gilt:

- $Q = (u), D = \Delta$
- Alle mit $Dist \leq \Delta$ erfasst.
- Alle mit $\Delta + 1$ = weiße Nachbarn von u.

(Wegen Invariante) Da letzter Lauf, danach $Q = ()$, alle $\leq \Delta$ erfasst, es gibt keine $\geq \Delta + 1$. Also alle von s Erreichbaren erfasst.

Termination: Pro Lauf ein Knoten aus Q, schwarz, kommt nie mehr in Q. Irgendwann ist Q zwangsläufig leer.

Nach BFS(G, s): Alle von s erreichbaren Knoten werden erfasst.

- $d[u] = Dist(s, u)$ für alle $u \in V$. Invariante um Aussage erweitern: Für alle erfassten Knoten ist $d[u] = Dist(s, u)$.
- Breitensuchbaum enthält kürzeste Wege. Invariante erweitern gemäß: Für alle erfassten Knoten sind kürzeste Wege im Breitensuchbaum.

Beachte: Dann ist der kürzeste Wege s $\circ\!\longrightarrow\!\circ$ u durch

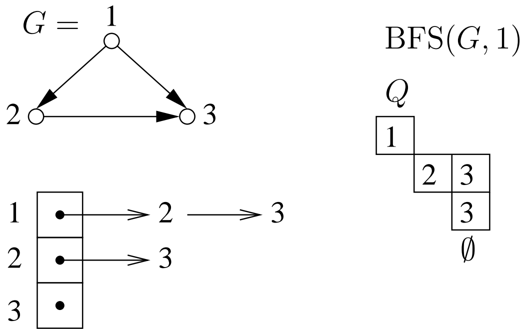
$$u \leftarrow \pi[u] \leftarrow \pi[\pi[u]] \leftarrow \circ \leftarrow \ldots \leftarrow \circ \leftarrow \pi[\pi[\ldots\pi[u]\ldots]] = s$$

bestimmt.

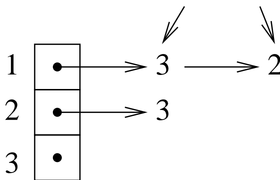
Beobachtung

In BFS(G, s) wird jede *„von s erreichbare Kante“* (Kante (u, v), wobei u von s erreichbar ist) genau einmal untersucht. Denn wenn u $\circ\!\longrightarrow\!\circ$ v, dann ist u grau, v irgendwie, danach ist u schwarz und nie mehr grau.

Wenn (u, v) gegangen wird bei BFS(G, s), d.h. wenn u expandiert wird, ist u grau ($col[u]$ = grau) und v kann schwarz, grau oder weiß sein. Die Farbe von v zu dem Zeitpunkt hängt nicht nur von G selbst, sondern auch von den Adjazenzlisten ab.

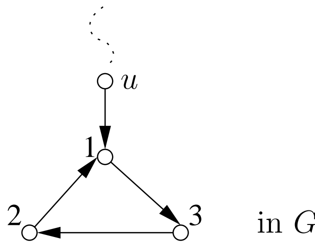


Beim Gang durch $(2, 3)$ ist 3 grau.



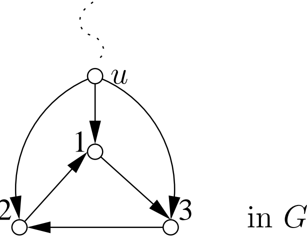
Beim Gang durch $(2, 3)$ ist 3 schwarz. Andere Farben: Übungsaufgabe.

Weiterer Nutzen der Breitensuche? – Kreise finden?



1 ist der erste Knoten, der auf dem Kreis entdeckt wird. Wenn 2 expandiert wird, ist 1 schwarz.

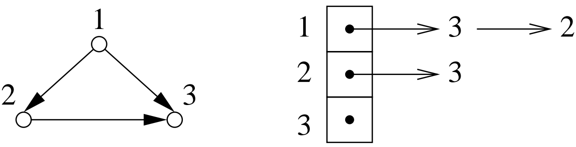
Auch möglich:



Alle drei Knoten $1, 2, 3$ werden bei der Expansion von u aus das erste Mal entdeckt. Trotzdem: Es gibt eine Kante, die bei Bearbeitung auf einen schwarzen Knoten führt, hier $(2, 1)$.

Folgerung 1.2: *Kreis* $\Longrightarrow$ *Es gibt eine Kante* $u \circ\!\!\longrightarrow\!\circ v$*, so dass v schwarz ist, wenn* $u \circ\!\!\longrightarrow\!\circ v$ *gegangen wird.*

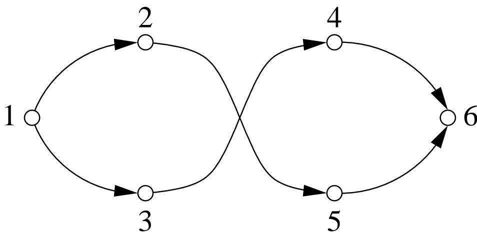
Ist ein Kreis daran erkennbar? – Leider nein, denn



Dann 3 schwarz bei $(2, 3)$, trotzdem kein Kreis.

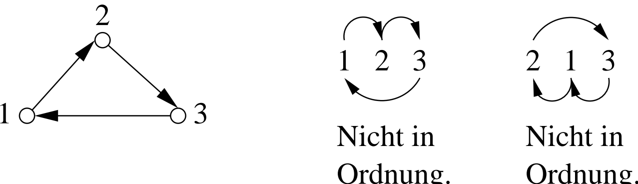
1.6 Topologische Sortierung

Wie erkenne ich kreisfreie gerichtete Graphen?



Anordnung der Knoten:

Alle Kanten gemäß der Ordnung. Aber bei Kreis:



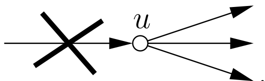
Definition 1.4(Topologische Sortierung): *Ist $G = (V, E)$ ein gerichteter Graph. Eine topologische Sortierung ist eine Anordnung von V als $(v_1, v_2, v_3, \ldots, v_n)$, so dass gilt:*

Ist u ○⟶○ v $\in E$*, so ist* $u = v_i, v = v_j$ *und* $i < j$*. Alle Kanten gehen von links nach rechts in der Ordnung.*

Satz 1.1: *G hat topologische Sortierung $\iff$ G hat keinen Kreis.*

Beweis. „⇒“ Sei also $(v_1, v_2, \ldots, v_n)$ topologische Sortierung von G. Falls Kreis $(w_0, w_1, \ldots, w_0)$, dann mindestens eine Kante der Art v_j ○⟶○ v_i mit $i < j$.

„⇐“ Habe also G keinen Kreis. Dann gibt es Knoten u mit $\text{Egrad}(u) = 0$. Also



Wieso? Nehme irgendeinen Knoten u_1. Wenn $\text{Egrad}(u_1) = 0$, dann ✓.

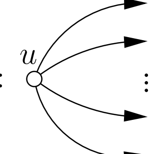
Sonst $u_2 \circ\!\!\longrightarrow\!\!\circ\, u_1$.

Nehme u_2 her. $\text{Egrad}(u_2) = 0$ ✓, sonst zu u_3 in G: $u_3 \circ\!\!\longrightarrow\!\!\overset{u_2}{\circ}\!\!\longrightarrow\!\!\circ\, u_1$

Falls $Egrad(u_3) = 0$, dann ✓, sonst $u_4 \circ\!\!\longrightarrow\!\!\overset{u_3}{\circ}\!\!\longrightarrow\!\!\overset{u_2}{\circ}\!\!\longrightarrow\!\!\circ\, u_1$

in G. Immer so weiter. Spätestens u_n hat $Egrad(u_n) = 0$, da sonst Kreis.

Also: Haben u mit $Egrad(u) = 0$. Also in G sieht es so aus:



$v_1 = u$ erster Knoten der Sortierung. Lösche u aus G. Hier wieder Knoten u' mit $Egrad(u') = 0$ im neuen Graphen, wegen Kreisfreiheit. $v_2 = u'$ zweiter Knoten der Sortierung. Immer so weiter $\rightarrow$ gibt topologische Sortierung.

Formal: Induktion über $|V|$. $\square$

1.7 Algorithmus Top Sort

Eingabe: $G = (V, E)$ beliebiger gerichteter Graph, $V = \{1, \ldots, n\}$.

Ausgabe: Array $v[1..n]$, so dass $(v[1], v[2], \ldots, v[n])$ eine topologische Sortierung darstellt, sofern G kreisfrei. Anderenfalls Meldung, dass G Kreis hat.

Vorgehensweise:

1.
   - Suche Knoten u mit $Egrad(u) = 0$.
   - Falls nicht existent $\Rightarrow$ Kreis.
   - Falls u existiert $\Rightarrow$ u aus G löschen. Dazu Kanten (u, v) löschen.
2.
   - Suche Knoten u mit $Egrad(u) = 0$.

   … Wie oben.

Wie findet man u gut?

1. Falls Adjazenzmatrix, $A = (a_{v_1,v_2})$, dann alle $a_{v_1,u} = 0$ für $v_1 \in V$.
2. G in Adjazenzlisten gegeben. Jedesmal Adjazenzlisten durchsuchen und schauen, ob ein Knoten u nicht vorkommt.

   Dazu ein Array A nehmen, $A[u] = 1 \Leftrightarrow$ u kommt vor. Ein solches u hat $\text{Egrad}(u) = 0$. Dann $Adj[u] = nil$ setzen.

Schleife mit n Durchläufen. Wenn kein u gefunden wird Ausgabe Kreis, sonst beim i-ten Lauf $v[i] = u$.

Verbessern der Suche nach u mit $Egrad(u) = 0$:

1. Ermittle Array Egrad$[1..n]$, mit den Eingangsgraden.
2. Suche u mit Egrad$[u] = 0$, $v[i] = u$, Kreis falls nicht existent. Für alle $v \in Adj[u]$ Egrad$[v]$ = Egrad$[v] - 1$.

   ...

Weitere Verbesserung: Knoten u mit Egrad$[u] = 0$ gleich in Datenstruktur (etwa Schlange) speichern.

1.8 Algorithmus Verbessertes Top Sort

Eingabe: $G = (V, E)$ beliebiger gerichteter Graph in Adjazenzlistendarstellung. Sei $V = \{1, \ldots, n\}$.

Ausgabe: Wie bei Top Sort.

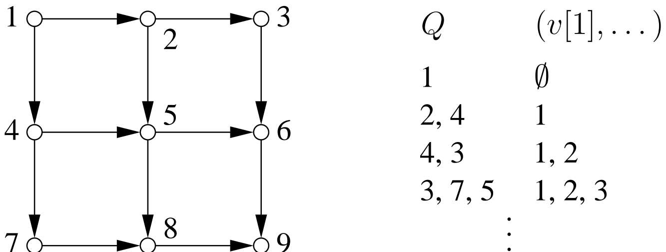
Weitere Datenstrukturen:

- *Array* Egrad$[1..n]$ für aktuelle Eingangsgrade.
- *Schlange* Q, Knoten mit Egrad $= 0$, Q kann auch Liste sein.

Algorithmus 2: TopSort(G)

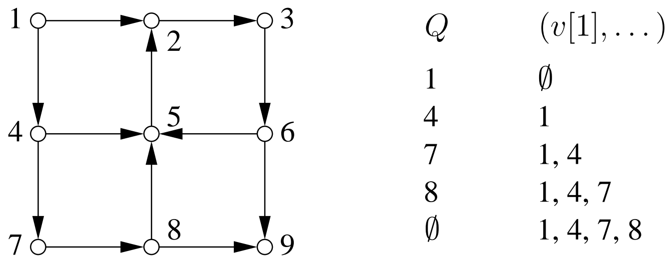
```
   /* 1. Initialisierung */
1  Egrad[v] auf 0, Q = ();
2  foreach v ∈ V do
3  |  Gehe Adj[v] durch;
4  |  zähle für jedes gefundene u Egrad[u] = Egrad[u] + 1;
5  end
   /* 2. Einfügen in Schlange */
6  foreach u ∈ V do
7  |  if Egrad[u] = 0 then
8  |  |  Q=enqueue(Q, u);
9  |  end
10 end
   /* 3. Array durchlaufen */
11 for i=1 to n do
12 |  if Q = () then
13 |  |  Ausgabe „Kreis“;
14 |  |  return;
15 |  end
   |  /* 4. Knoten aus Schlange betrachten */
16 |  v[i] = Q[head];
17 |  v[i] = dequeue(Q);
   |  /* 5. Adjazenzliste durchlaufen */
18 |  foreach u ∈ Adj[v[i]] do
19 |  |  Egrad[u] = Egrad[u] − 1;
20 |  |  if Egrad[u] = 0 then
21 |  |  |  Q = enqueue(Q, u);
22 |  |  end
23 |  end
24 end
```

Ein Beispiel:



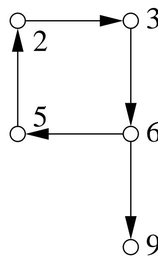
Q	$(v[1], \dots)$
1	$\emptyset$
2, 4	1
4, 3	1, 2
3, 7, 5	1, 2, 3
$\vdots$	

Noch ein Beispiel:



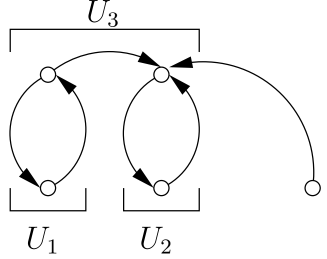
Q	$(v[1], \dots)$
1	$\emptyset$
4	1
7	1, 4
8	1, 4, 7
$\emptyset$	1, 4, 7, 8

Es verbleibt als Rest:



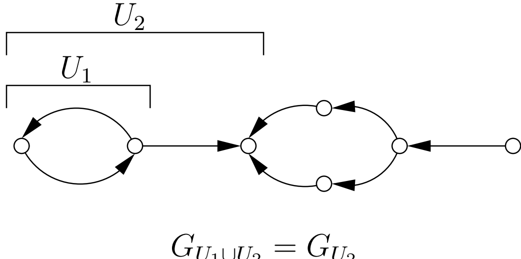
Wir betrachten einen Graphen $G = (V, E)$. Für $U \subseteq V$ ist $G_U = (U, F), F = \{(u, v) \mid u, v \in U \text{ und } (u, v) \in E\}$ der auf U induzierte Graph. Uns interessieren die $U \subseteq V$, so dass in G_U für jeden Knoten Egrad ≥ 1 ist.

Seien $U_1, \dots, U_k$ alle U_i, so dass in G_{U_i} jeder Egrad ≥ 1 ist. Dann gilt auch in $G_{U_1 \cup \ldots \cup U_k} = G_X$ ist jeder Egrad ≥ 1. Das ist der größte Graph mit dieser Eigenschaft. Am Ende von TopSort bleibt dieser Graph übrig ($X = \emptyset$ bis $X = V$ möglich).



$G_{U_1} \cup G_{U_2} \cup G_{U_3} = G_{U_3} =$

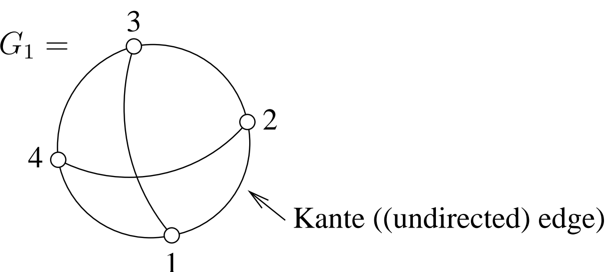
Größter induzierter Teilgraph, wobei jeder $Egrad \geq 1$ ist.



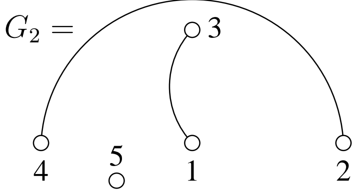
$G_{U_1 \cup U_2} = G_{U_2}$

2 Ungerichtete Graphen

Ein typischer ungerichteter Graph:



Auch einer:



Ungerichteter Graph G_2 mit $V = \{1, 2, 3, 4, 5\}$ und $E = \{\{4, 2\}, \{1, 3\}\}$.

Beachte:

Ungerichtet, deshalb Kante als Menge, $\{4, 2\} = \{2, 4\}$. (Manchmal auch im ungerichteten Fall Schreibweise $(4, 2) = \{4, 2\}$, dann natürlich $(4, 2) = (2, 4)$. *Nicht* im gerichteten Fall!)

Wie im gerichteten Fall gibt es keine Mehrfachkanten und keine Schleifen.

2, 3 $\quad (3, 2)_1, (3, 2)_2, (3, 2)_3 \in E$ und 3 $\quad (3, 3)$ hier nicht erlaubt.

Definition 2.1(ungerichteter Graph): *Ein ungerichteter Graph besteht aus 2 Mengen:*

- *V, beliebige endliche Menge von Knoten*
- *$E \subseteq \big\{\{u, v\} \mid u, v \in V, u \neq v\big\}$, Kanten*

Schreibweise $G = (V, E)$, $\quad \{u, v\} \rightsquigarrow$ u —— v .

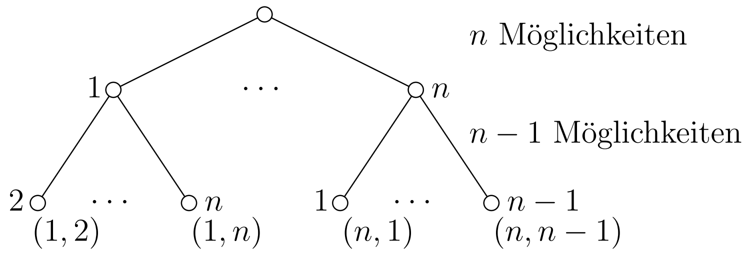
Folgerung 2.1: *Ist $|V| = n$ fest, dann:*

a) Jeder ungerichtete Graph mit Knotenmenge V hat $\leq \binom{n}{2} = \frac{n \cdot (n-1)}{2}$ Kanten.

b) #ungerichtete Graphen über $V = 2^{\binom{n}{2}}$. (#gerichtete Graphen: $2^{n(n-1)}$)

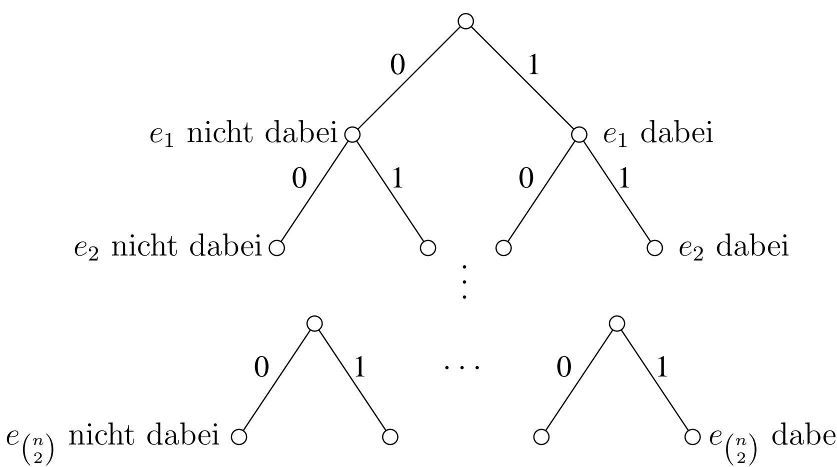
Beweis.

a) „Auswahlbaum“



- $n(n-1)$ Blätter,
- Blatt $\Longleftrightarrow (u,v), u \neq v$.
- Kante $\{u,v\} \Longleftrightarrow$ 2 Blätter, (u,v) und (v,u). $\Rightarrow$ Also $\binom{n}{2} = \frac{n \cdot (n-1)}{2}$ Möglichkeiten.

b) Graph = Teilmenge von Kanten. Alle Kanten: $e_1, e_2, \ldots, e_{\binom{n}{2}}$.



Jedes Blatt $\Leftrightarrow$ 1 ungerichteter Graph. Schließlich $2^{\binom{n}{2}}$ Blätter.

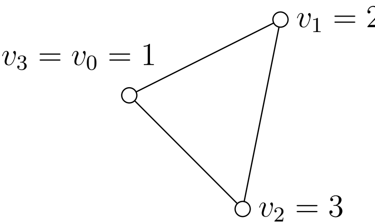
Alternativ: Blatt $\Longleftrightarrow$ 1 Bitfolge der Länge $\binom{n}{2}$. Es gibt $2^{\binom{n}{2}}$ Bitfolgen der Länge $\binom{n}{2}$.
Beachte: $\binom{n}{2}$ ist $O(n^2)$.

$\square$

Adjazent, inzident sind analog zum gerichteten Graph zu betrachten.

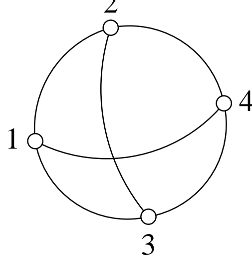
- Grad(u) = $\{\{u, v\} \mid \{u, v\} \in E\}$, wenn $G = (V, E)$. Es gilt immer $0 \leq Grad(u) \leq n - 1$.
- Weg $(v_0, v_1, \ldots, v_k)$ wie im gerichteten Graphen.
- Länge $(v_0, v_1, \ldots, v_k) = k$
- Auch ein Weg ist bei $\{u, v\} \in E$ der Weg (u, v, u, v, u, v). Einfach $\iff v_0, v_1, \ldots, v_k$ alle verschieden, d.h., $|\{v_0, v_1, \ldots, v_k\}| = k + 1$
- $(v_0, \ldots v_k)$ geschlossen $\iff v_0 = v_k$
- $(v_0, \ldots, v_k)$ ist Kreis, genau dann wenn:
  - $k \geq 3(!)$
  - $(v_0, \ldots, v_{k-1})$ einfach
  - $v_0 = v_k$

  $(1, 2, 1)$ wäre kein Kreis, denn sonst hätte man immer einen Kreis. $(1, 2, 3, 1)$ jedoch ist



  ein solcher mit $v_0 = 1, v_1 = 2, v_2 = 3, v_3 = 1 = v_0$.

Die Adjazenzmatrix ist symmetrisch (im Unterschied zum gerichteten Fall).



G_1	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

symmetrisch $\iff a_{u,v} = a_{v,u}$

3

2

1

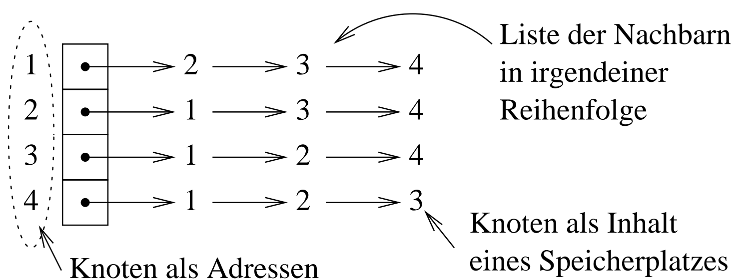
4

G_2	1	2	3	4
1	0	1	0	0
2	1	0	0	0
3	0	0	0	0
4	0	0	0	0

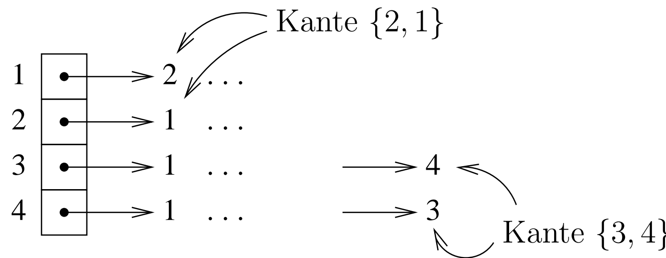
Darstellung als Array: n^2 Speicherplätze.

Adjazenzlistendarstellung:

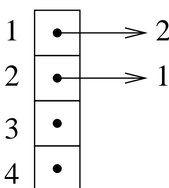
Array *Adj* von G_1



Jede Kante zweimal dargestellt, etwa:

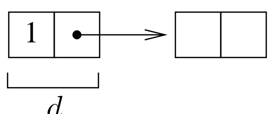


Adjazenzlistendarstellung von G_2:



Platz zur Darstellung von $G = (V, E) : c + n + d \cdot 2 \cdot |E|$

- $c \ldots$ Initialisierung,
- $n \ldots$ Array Adj,
- $d \cdot 2 \cdot |E| \ldots$ jede Kante zweimal, d etwa 2:



$O(|E|)$, wenn $|E| \geq \frac{n}{2}$. $|E|$ Kanten sind inzident mit $\leq 2|E|$ Knoten. Adjazenzlisten in Array wie im gerichteten Fall.

2.1 Algorithmus Breitensuche (BFS)

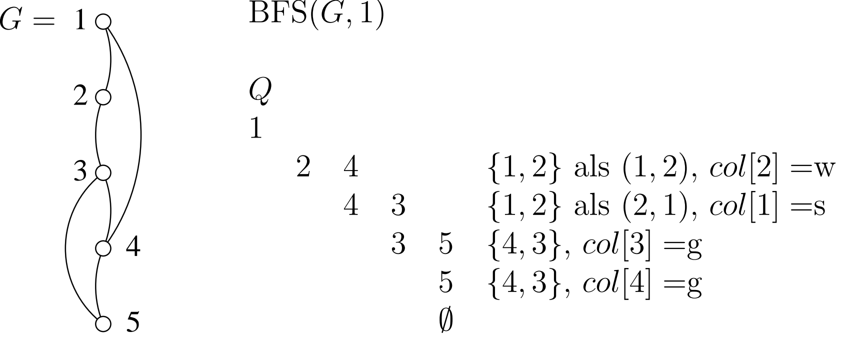
BFS(G, s) genau wie im gerichteten Fall.

Eingabe: G in Adjazenzlistendarstellung

Datenstrukturen: Schlange $Q[1 \dots n]$, Array $col[1 \dots n]$

Algorithmus 3: BFS(G, s)

```
 1 foreach u ∈ V do
 2 |   col[u] = weiß;
 3 end
 4 col[s] = grau;
 5 Q = (s);
 6 while Q ≠ () do
 7 |   u = Q[head] ;                              /* u wird expandiert */
 8 |   foreach v ∈ Adj[u] do                  /* {u, v} wird expandiert */
 9 |   |   if col[v] == weiß then                 /* v wird expandiert */
10 |   |   |   col[v] = grau;
11 |   |   |   v in Schlange setzen;
12 |   |   end
13 |   end
14 |   u aus Q raus;
15 |   col[u] = schwarz;
16 end
```

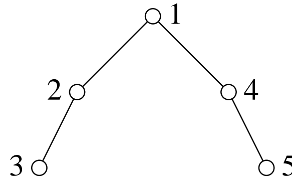


Beobachtung: Sei $\{u, v\}$ eine Kante.

a) Die Kante u, v wird genau zweimal durchlaufen, einmal von u aus, einmal von v aus.

b) Ist der erste Lauf von u aus, so ist zu dem Zeitpunkt $col[v] = w$ oder $col[v] = g$. Beim zweiten Lauf ist dann jedenfalls $col[u] = s$.

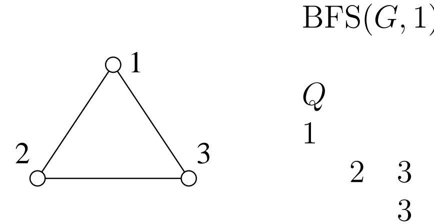
Breitensuchbaum durch $\pi[1, \dots, n]$, Vaterarray. Hier



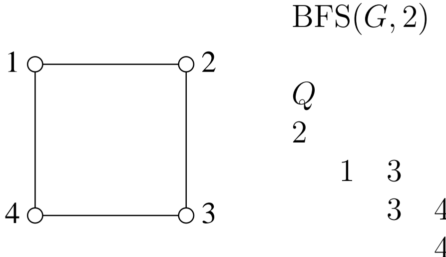
$\pi[3] = 2, \pi[2] = 1, \pi[4] = 1, \pi[5] = 4.$

Entdecktiefe durch $d[1 \dots n]$. $\pi[v], d[v]$ werden gesetzt, wenn v entdeckt wird.

Wir können im ungerichteten Fall erkennen, ob ein Kreis vorliegt:



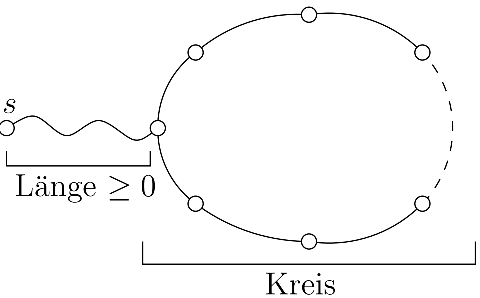
Beim (ersten) Gang durch $\{2, 3\}$ ist $col[3] = grau$, d.h. $3 \in Q$.



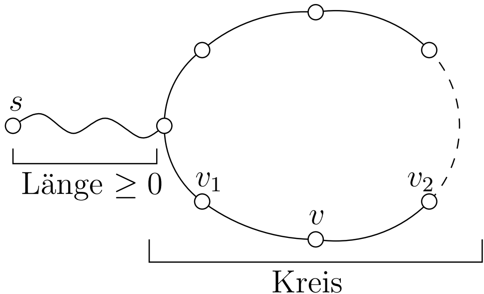
Beim (ersten) Gang durch $\{3, 4\}$ ist $col[4] = grau$, d.h. $4 \in Q$.

Satz 2.1: *$G = (V, E)$ hat einen Kreis, der von s erreichbar ist $\Leftrightarrow$ Es gibt eine Kante $e = \{u, v\}$, so dass BFS(G, s) beim (ersten) Gang durch e auf einen grauen Knoten stößt.*

Beweis. „$\Rightarrow$“ Also haben wir folgende Situation in G:



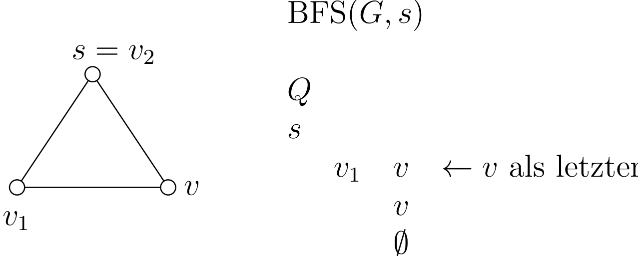
Sei v der Knoten auf dem Kreis, der als *letzter(!)* entdeckt wird.



v_1, v_2 die beiden Nachbarn von v auf dem Kreis, $v_1 \neq v_2$ (Länge ≥ 3). In dem Moment, wo v entdeckt wird, ist $col[v_1] = col[v_2] = grau$. (Schwarz kann nicht sein, da sonst v vorher bereits entdeckt, weiß nicht, da v letzter.)

1. Fall: v wird über v_1 entdeckt, dann $Q = (\ldots v_2 \ldots v)$ nach Expansion von v_1. (Ebenso für v_2.)

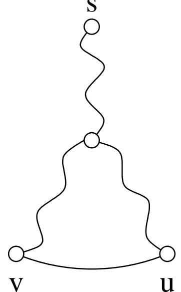
2. Fall: v wird über $u \neq v_1, u \neq v_2$ entdeckt. Dann $Q = (\ldots v_1 \ldots v_2 \ldots v)$ nach Expansion von u.



"$\Leftarrow$" Also bestimmen wir irgendwann während der Breitensuche die Situation $Q = (\ldots u \ldots v \ldots)$ und beim Gang durch $\{u, v\}$ ist v grau. Dann ist $u \neq s, v \neq s$.

Wir haben Wege $s \rightsquigarrow u$ und $s \rightsquigarrow v$.

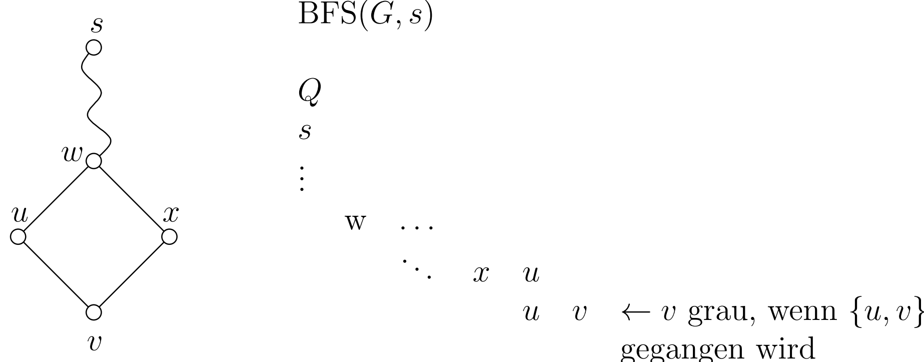
Dann im Breitensuchbaum der Suche:



Also einen Kreis. Etwas formaler:

Haben Wege $(v_0 = s, \dots, v_k = v)$, $(w_0 = s, \dots, w_k = u)$ und $v \neq u, v \neq s$. Gehen die Wege von v und u zurück. Irgendwann der erste Knoten gleich. Da Kante $\{u, v\}$ haben wir einen Kreis. $\square$

Beachten nun den folgenden Fall.



Also $d[v] = d[u] + 1$. Also nicht immer $d[v] = d[u]$, wenn v grau ist beim Gang durch u.

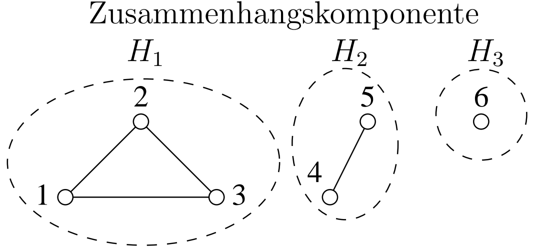
Definition 2.2(Zusammenhang): *Sei $G = (V, E)$ ungerichteter Graph.*

a) *G heißt zusammenhängend, genau dann, wenn es für alle $u, v \in V$ einen Weg (u, v) gibt.*

b) *Sei $H = (W, F)$, $W \subseteq V, F \subseteq E$ ein Teilgraph. H ist eine Zusammenhangskomponente von G, genau dann, wenn*

   - *F enthält alle Kanten $\{u, v\} \in E$ mit $u, v \in W$ (H ist der auf W induzierte Teilgraph)*
   - *H ist zusammenhängend.*
   - *Es gibt keine Kante $\{u, v\} \in E$ mit $u \in W, v \notin W$.*

Man sagt dann: H ist ein maximaler zusammenhängender Teilgraph.



Beachte: In obigem Graphen ist [Graph: 1 — 2 — 3] keine Zusammenhangskomponente, da $\{1,3\}$ fehlt. Ebenso wenig ist es [Graph: 1 — 2], da nicht maximaler Teilgraph, der zusammenhängend ist.

Satz 2.2: *Ist $G = (V, E)$ zusammenhängend. Dann gilt:*

G hat keinen Kreis $\iff |E| = |V| - 1$

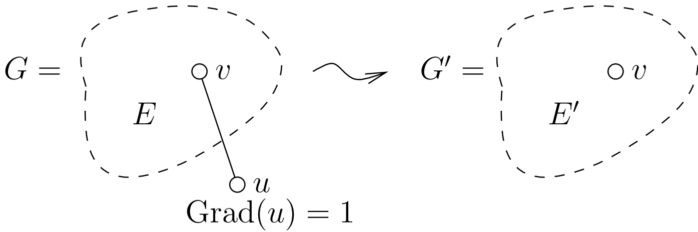
Beweis. „⇒" Induktion über $n = |V|$.

$n = 1$, dann ○, also $E = \emptyset$.

$n = 2$, dann ○——○, also ✓.

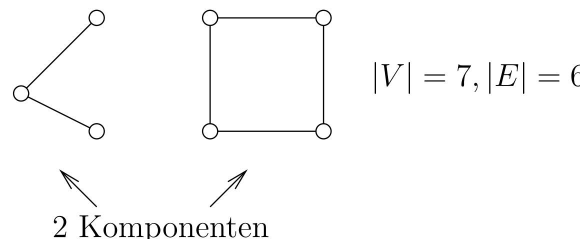
Sei $G = (V, E), |V| = n + 1$ und ohne Kreis. Da G zusammenhängend ist und ohne Kreis, gibt es Knoten vom $Grad = 1$ in G.

Wären alle Grade ≥ 2, so hätten wir einen Kreis. Also:

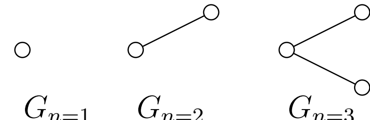


Dann gilt für $G' = (V', E')$: G hat keinen Kreis $\Rightarrow G'$ hat keinen Kreis $\Rightarrow$ Induktionsvoraussetzung $|E'| = |V'| - 1 \Rightarrow |E| = |V| - 1$.

„⇐" Beachte zunächst, dass die Aussage für G nicht zusammenhängend nicht gilt:



Also wieder Induktion über $n = |V|$. $n = 1, 2, 3$, dann haben wir die Graphen



und die Behauptung gilt.

Sei jetzt $G = (V, E)$ mit $|V| = n + 1$ zusammenhängend und $|E| = |V|$. Dann gibt es Knoten vom Grad $= 1$. Sonst

$$\sum_v Grad(v) \geq 2(n+1) = 2n+2,$$

also $|E| \geq n+1$ $\left(|E| = \frac{1}{2}\sum_v \text{Grad}(v)\right)$. Einziehen von Kante und Löschen des Knotens macht Induktionsvoraussetzung anwendbar. □

Folgerung 2.2:

a) Ist $G = (V, E)$ *Baum, so ist* $|E| = |V| - 1$.

b) Besteht $G = (V, E)$ *aus* k *Zusammenhangskomponenten, so gilt:*

G *ist kreisfrei* $\Longleftrightarrow |E| = |V| - k$

Beweis.

a) Baum kreisfrei und zusammenhängend.

b) Sind $G_1 = (V_1, E_1), \ldots, G_k = (V_k, E_k)$ die Zusammenhangskomponenten von G. (Also insbesondere $V_1 \mathbin{\dot\cup} \ldots \mathbin{\dot\cup} V_k = V$, $E_1 \mathbin{\dot\cup} \ldots \mathbin{\dot\cup} E_k = E$.) Dann gilt G_i ist kreisfrei $\Leftrightarrow |E_i| = |V_i| - 1$ und die Behauptung folgt.

Beachte eben $G =$ [Dreieck mit drei weiteren isolierten Knoten] . $|V| = 6, |E| = 3 < |V|$ trotzdem hat G einen Kreis.

□

2.2 Algorithmus (Finden von Zusammenhangskomponenten)

Eingabe: $G = (V, E)$ ungerichtet,

Ausgabe: Array $A[1 \dots |V|]$ mit $A[u] = A[v] \Longleftrightarrow u$ und v sind in einer Komponente.

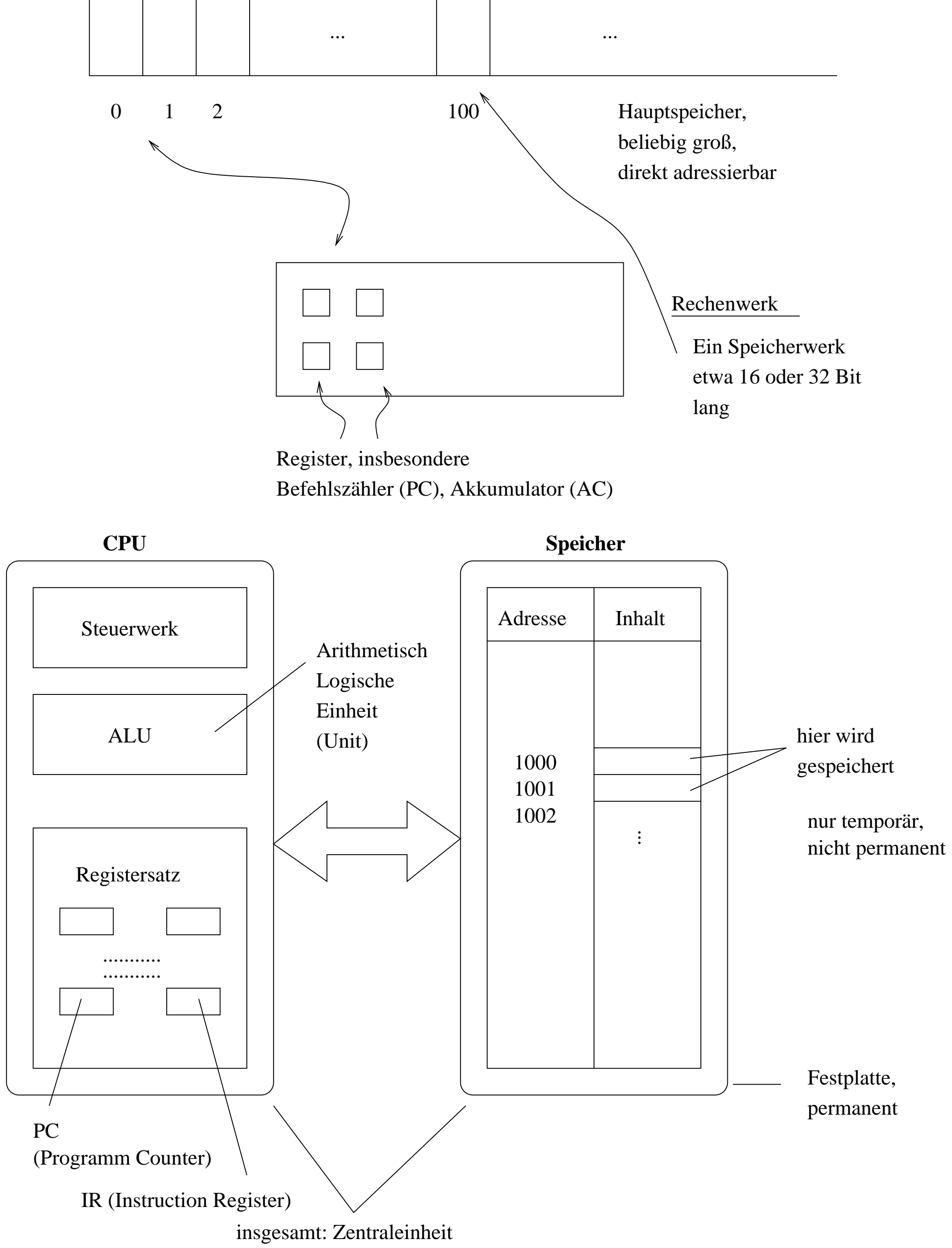
(Frage: Ist u von v erreichbar? In einem Schritt beantwortbar.)

Algorithmus 4: Einfache Zusammenhangskomponenten

```
1 A auf Null initialisieren;
2 Initialisierung von BFS;
3 foreach w ∈ V do
4 |  if col[w] == weiß then
5 |  |  BFS(G, w) modifiziert, so dass keine erneute Initialisierung. Bereits
  |  |  in einem früheren Lauf entdeckte Knoten bleiben also schwarz. Und
  |  |  A[u] = w wird gesetzt, falls u im Verlauf entdeckt wird.
6 |  end
7 end
```

3 Zeit und Platz

Unsere Programme laufen letztlich immer auf einem *Von-Neumann-Rechner* (wie nachfolgend beschrieben) ab. Man spricht auch von einer Random Access Machine (RAM), die dann vereinfacht so aussieht:



Typische Befehle der Art:

`Load 5000`	Inhalt von Speicherplatz 5000 in Akkumulator
`Add 1005`	Inhalt von 1005 hinzuaddieren
`Store 2000`	Inhalt des Akkumulators in Platz 2000 speichern

Programme liegen im Hauptspeicher.

Simulation von Schleifen durch Sprungbefehle, etwa:

`JP E`	unbedingter Sprung nach E
`JPZ E`	Ist ein bestimmtes Register = 0, dann Sprung nach E (Jump if Zero)

Zur Realisierung von Pointern müssen Speicherinhalte als Adressen interpretiert werden. Deshalb auch folgende Befehle:

`Load ↑x`	Inhalt des Platzes, dessen Adresse in Platz x steht wird geladen
`Store ↑x, Add ↑x, ...`	...

Weitere Konventionen sind etwa:

- Eingabe in einem speziell reservierten Bereich des Hauptspeichers.
- Ebenso Ausgabe.

Listing 1: Java-Programm zum Primzahltest

```
1  // Hier einmal ein Primzahltest.
2
3  import Prog1Tools.IOTools;
4  public class PRIM{
5  public static void main(String[] args){
6    long c, d;
7    c = IOTools.readLong("Einlesen eins langen c zum Test:");
8    if (c == 2){
9      System.out.print(c +"ist PRIM");
10     return;
11   }
12   d = 2;
13   while(d * d <=c){ // Warum reicht es, bei D*D > c aufzuhören?
14     if (c % d == 0){
15       System.out.println(c + "ist nicht PRIM, denn" + d + "teilt"
           + c);
16       return;
17     }
18     d++;
19   }
20   System.out.println("Die Schleife ist fertig ohne ordentlichen
        Teiler, ist" + c + "PRIM");
21 }
22 }
```

Das obige Programm wird etwa folgendermaßen für die RAM übersetzt. Zunächst die while-Schleife:

```
Load d    // Speicherplatz von d in Register (Akkumulator)
Mult d    // Multiplikation von Speicherplatz d mit Akkumulator
Store e   // Ergebnis in e
...       // Analog in f e-c berechnen
Load f    // Haben d·d − c im Register
JGZ E     // Sprung ans Ende wenn d·d − c > 0, d.h. d·d > c
```

Das war nur der Kopf der while-Schleife. Jetzt der Rumpf:

```
                 ...       // c % d in Speicherplatz m ausrechnen.
                 Load m
if (c%d){...}    JGZ F     // Sprung wenn > 0
                 ...       // Übersetzung von System.out.println(...)
                 JP E      // Unbedingt ans Ende

          F:  Load d
              Inc      // Register erhöhen
d++           JP A     // Zum Anfang
          E:  Stop
```

Wichtige Beobachtung: Jede einzelne Java-Zeile führt zur Abarbeitung einer *konstanten* Anzahl von Maschinenbefehlen (konstant $\Leftrightarrow$ unabhängig von der Eingabe c, wohl abhängig von der eigentlichen Programmzeile.)

Für das Programm PRIM gilt: Die Anzahl ausgeführter Java-Zeilen bei Eingabe c ist $\leq a\sqrt{c} + b$ mit

$a \cdot \sqrt{c}$ . . . Schleife (Kopf und Rumpf) und

b . . . Anfang und Ende a, b konstant, d.h. unabhängig von c!

Also auch die Anzahl der ausgeführten Maschinenbefehle bei Eingabe c ist $\leq a' \cdot \sqrt{c} + b'$.

Ausführung eines Maschinenbefehls: Im Nanosekundenbereich

#Nanosekunden bei Eingabe $c \leq a'' \cdot \sqrt{c} + b''$.

In jedem Fall gibt es eine Konstante k, so dass der „Verbrauch“ $\leq k\sqrt{c}$ ist ($k = a + b, a' + b', a'' + b''$).

Fazit: Laufzeit unterscheidet sich nur um einen konstanten Faktor von der Anzahl ausgeführter Programmzeilen. Schnellere Rechner $\Rightarrow$ Maschinenbefehle schneller $\Rightarrow$ Laufzeit nur um einen konstanten Faktor schneller. Bestimmen Laufzeit nur bis auf einen konstanten Faktor.

Definition 3.1(O-Notation): *Ist $f, g : \mathbb{N} \to \mathbb{R}$. Dann ist $f(n) = O(g(n))$ genau dann, wenn es eine Konstante $C > 0$ mit $|f(n)| \leq C \cdot g(n)$ gibt, für alle hinreichend großen n.*

Das Programm PRIM hat Zeit $O(\sqrt{c})$ bei Eingabe c. Platzbedarf (= #belegte Plätze) etwa 3, also $O(1)$.

Eine Java-Zeile $\Leftrightarrow$ endliche Anzahl Maschinenbefehle trifft nur bedingt zu:

Solange Operanden nicht zu groß (d.h. etwa in einem oder einer endlichen Zahl von Speicherplätzen). D.h., wir verwenden das *uniforme Kostenmaß*. Größe der Operanden ist uniform 1 oder $O(1)$.

Zur Laufzeit unseres Algorithmus BFS(G, s). (Seite 15)

Sei $G = (V, E)$ mit $|V| = n$, $|E| = m$. Datenstrukturen initialisieren (Speicherplatz reservieren usw.): $O(n)$.

1-3	Array col setzen	$O(n)$
4-5	Startknoten grau, in Schlange	$O(1)$
6	Kopf der while-Schleife	$O(1)$
7	Knoten aus Schlange	$O(1)$
8-13	Kopf einmal	$O(1)$
	(Gehen $Adj[u]$ durch)	
	Rumpf einmal	$O(1)$
	In einem Lauf der while-Schleife	
	wird 8-13 bis zu $n-1$-mal durchlaufen.	
	Also 8-13 in	$O(n)$.
14-15	Rest	$O(1)$.

Also: while-Schleife einmal:

$$O(1) + O(n) \text{ also } O(1) + O(1) + O(1) \text{ und } O(n) \text{ für } 8-13.$$

#Durchläufe von 6-15 $\leq n$, denn schwarz bleibt schwarz. Also Zeit $O(1) + O(n^2) = O(n^2)$.

Aber das ist nicht gut genug. Bei $E = \emptyset$, also keine Kante, haben wir nur eine Zeit von $O(n)$, da 8-13. jedesmal nur $O(1)$ braucht.

Wie oft wird 8-13. insgesamt (über das ganze Programm hinweg) betreten? Für jede Kante genau einmal. Also das Programm hat in 8-13 die Zeit $O(m+n)$, n für die Betrachtung von $Adj[u]$ an sich, auch wenn $m = 0$.

Der Rest des Programmes hat Zeit von $O(n)$ und wir bekommen $O(m+n) \leq O(n^2)$. Beachte $O(m+n)$ ist bestmöglich, da allein das Lesen des ganzen Graphen $O(m+n)$ erfordert.

Regel: Die Multiplikationsregel für geschachtelte Schleifen:

$$\text{Schleife } 1 \leq n \text{ Läufe, Schleife } 2 \leq m \text{ Läufe,}$$

also Gesamtzahl Läufe von Schleife 1 und Schleife 2 ist $m \cdot n$ gilt nur bedingt. Besser ist es (oft), Schleife 2 insgesamt zu zählen. Globales Zählen.

Betrachten nun das Topologische Sortieren von Seite 22.

$G = (V, E), |V| = n, |E| = m$

	Initialisierung:	$O(n)$
1.	Array Egrad bestimmen:	$O(m)$
	Knoten mit Grad 0 suchen:	$O(n)$
	Knoten in top. Sortierung tun und Adjazenzlisten anpassen:	$O(1)$
2.	Wieder genauso:	$O(m) + O(n) + O(1)$
⋮		

Nach Seite 24 bekommen wir aber Linearzeit heraus:

1. und 2. Egrad ermitteln und Q setzen:	$O(m)$
3. Ein Lauf durch 3. $O(1)$! (keine Schleifen), insgesamt n Läufe. Also:	$O(n)$
4. insgesamt	$O(n)$
5. jede Kante einmal, also insgesamt	$O(m)$

Zeit $O(m + n)$, Zeitersparnis durch geschicktes Merken.