Theoretische Informatik I

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

WS 2009/2010

Bitte beachten:

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

Version vom 16. September 2009

Inhaltsverzeichnis

1 Gerichtete Graphen **5**

Datenstruktur Adjazenzmatrix . . . 9
Datenstruktur Adjazenzliste . . . 9
Breitensuche . . . 12
Datenstruktur Schlange . . . 13
Algorithmus Breitensuche (Breadth first search = BFS) . . . 14
Topologische Sortierung . . . 20
Algorithmus Top Sort . . . 21
Algorithmus Verbessertes Top Sort . . . 22

2 Ungerichtete Graphen **24**

Algorithmus Breitensuche (BFS) . . . 28
Algorithmus (Finden von Zusammenhangskomponenten) . . . 34

3 Zeit und Platz **34**

4 Tiefensuche in gerichteten Graphen **40**

Algorithmus Tiefensuche . . . 41

5 Anwendung Tiefensuche: Starke Zusammenhangskomponenten **51**

Algorithmus Starke Komponenten . . . 55

6 Tiefensuche in ungerichteten Graphen: Zweifache Zusammenhangskomponenten **60**

Berechnung von l[v] . . . 69
Algorithmus (l-Werte) . . . 69
Algorithmus (Zweifache Komponenten) . . . 71

7 Minimaler Spannbaum und Datenstrukturen **72**

Algorithmus Minimaler Spannbaum . . . 76
Datenstruktur Union-Find-Struktur . . . 80
Algorithmus Union-by-Size . . . 82
Algorithmus Wegkompression . . . 85
Algorithmus Minimaler Spannbaum nach Prim 1963 . . . 89
Algorithmus (Prim mit Q in heap) . . . 94

8 Kürzeste Wege **95**

Algorithmus (Dijkstra 1959) . . . 97
Algorithmus (Dijkstra ohne mehrfache Berechnung desselben D[w]) . . . 99
Algorithmus Floyd Warshall . . . 105

9 Flüsse in Netzwerken **106**

Algorithmus (Ford Fulkerson) . . . 110

10 Kombinatorische Suche und Rekursionsgleichungen **113**
Algorithmus (Erfüllbarkeitsproblem) 115
Algorithmus (Davis-Putman) 117
Algorithmus (Pure literal rule, unit clause rule) 118
Algorithmus (Monien, Speckenmeyer) 122
Algorithmus 126
Algorithmus (Backtracking für TSP) 129
Algorithmus (Offizielles branch-and-bound) 132
Algorithmus (TSP mit dynamischem Programmieren) 135
Algorithmus (Lokale Suche bei KNF) 139

11 Divide-and-Conquer und Rekursionsgleichungen **144**
Algorithmus: Quicksort 146

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.

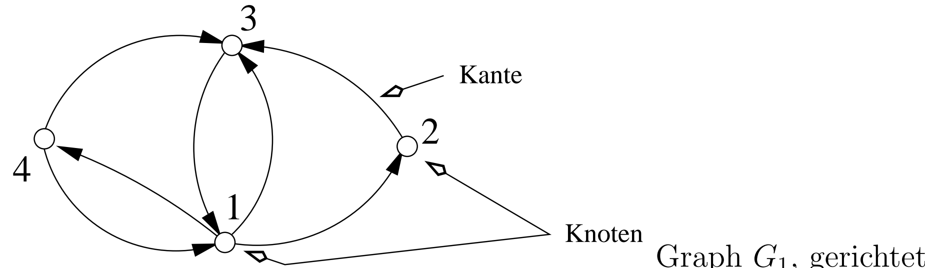
Der Vorlesung liegen die folgenden Lehrbücher zugrunde:

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 (= teilweise durchgerarbeitet) haben.

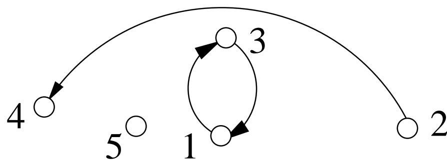
[1] Auf Deutsch im Oldenburg Verlag

1 Gerichtete Graphen

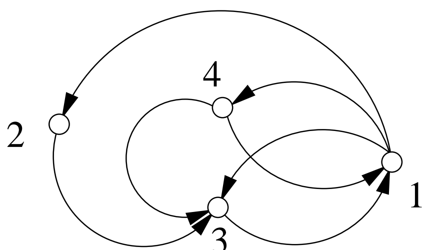


Graph G_1, gerichtet

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 $E \subseteq \{(u,v) \mid u,v \in V, u \neq v\}$

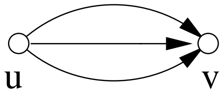
Schreibweise: $G = (V, E)$.

(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 1.2:

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:

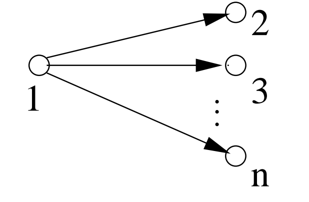
$f(n)$ ist $O(n^2)$ bedeutet: Es gibt eine Konstante $C > 0$, so dass $f(n) \leq C \cdot n^2$ für *alle* hinreichend großen n gilt.

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

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

Es gilt immer:

$$0 \leq Agrad(v) \leq n-1 \text{ und}$$
$$0 \leq Egrad(u) \leq n-1$$

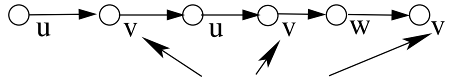


$$Egrad(1) = 0, Agrad(1) = n - 1$$

Definition 1.3 (*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.

Es ist die Länge von $v_0 \to v_1 \to v_2 \to \cdots v_{k-1} \to v_k = k$. Also die Länge $= \#$ (= 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
- $(v_o, v_1, ..., v_{k-1})$ 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}).

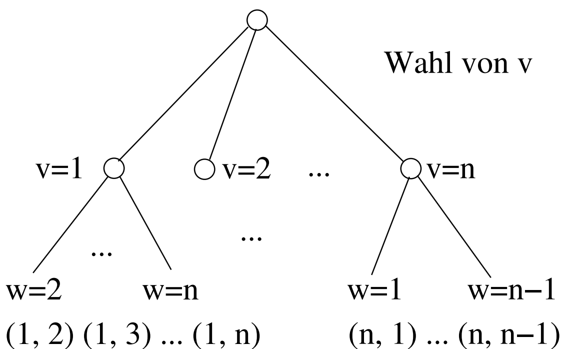
Die Kunst des Zählens ist eine unabdingbare Voraussetzung zum Verständnis von Laufzeitfragen. Wir betrachten zwei einfache Beispiele:

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

   Eine Kante = 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 $n^2 - n$:

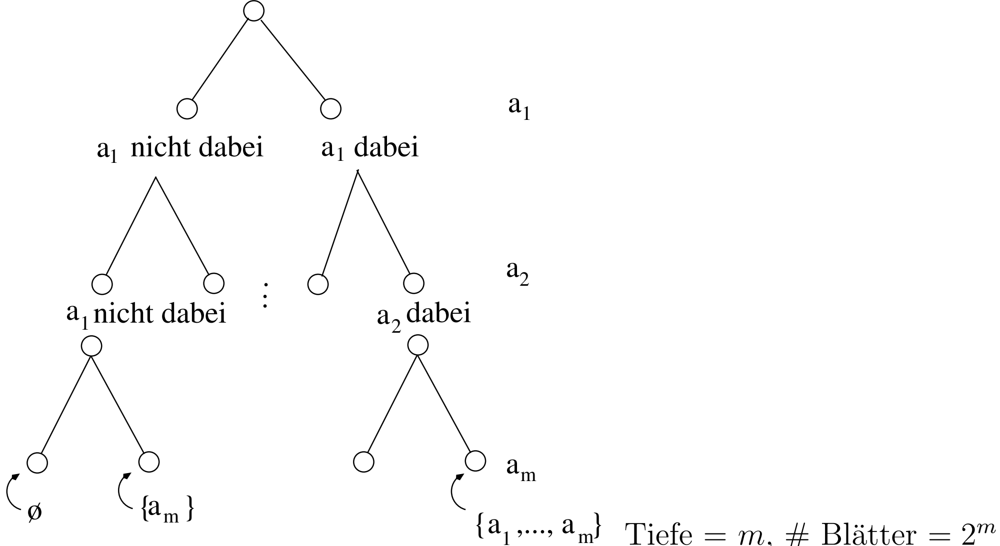
n^2 = # aller Paare (v, w), $v, w \in V$ (auch $v = w$) (Auswahlbaum)
n = # Paare (v, v) $v \in V$. Diese werden von n^2 abgezogen.

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

   $0\ 0\ 0 \ldots 0\ 0 \rightarrow \emptyset$
   $0\ 0\ 0 \ldots 0\ 1 \rightarrow \{a_m\}$
   $0\ 0\ 0 \ldots 1\ 0 \rightarrow \{a_{m-1}\}$
   $0\ 0\ 0 \ldots 1\ 1 \rightarrow \{a_{m-1}, a_m\}$
   $\vdots$
   $1\ 1\ 1 \ldots 1\ 1 \rightarrow \{a_1, a_2, ..., a_m\} = M$

   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:



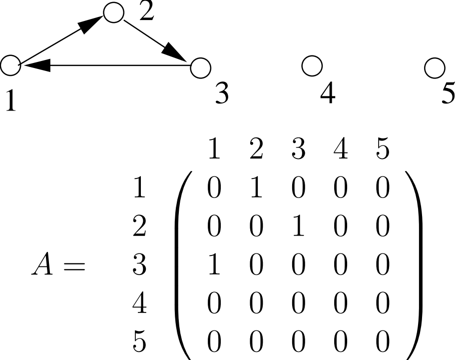
Tiefe $= m$, # Blätter $= 2^m$

Datenstruktur Adjazenzmatrix

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

$$A = (a_{u,v}), 1 \leq u \leq n, 1 \leq v \leq n, a_{u,v} \in \{0, 1\}$$
$$a_{u,v} = 1 \Leftrightarrow (u, v) \in E$$
$$a_{u,v} = 0 \Leftrightarrow (u, v) \notin E.$$

Implementierung durch Array $A[\quad][\quad]$.



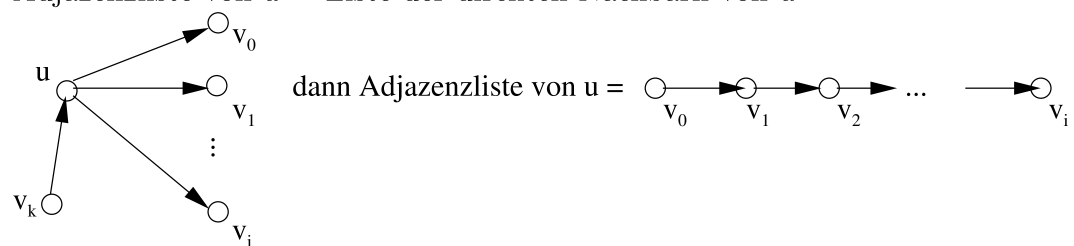
$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

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.

Datenstruktur Adjazenzliste

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

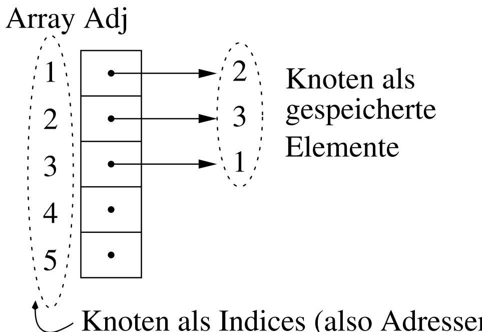
- Adjazenzliste von u = Liste der direkten Nachbarn von u



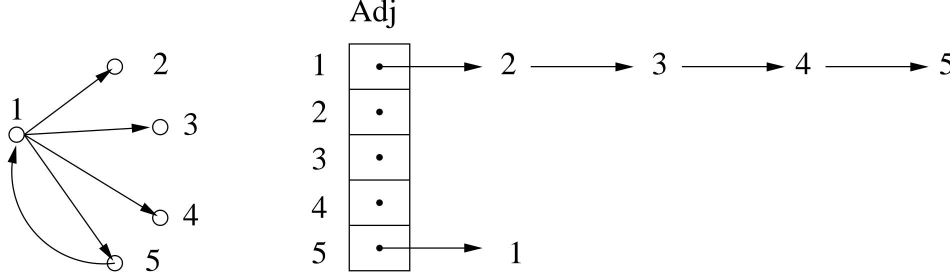
Nicht in der Adjazenzliste ist v_k, jede Reihenfolge von $v_1, ..., v_i$ erlaubt.

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

  Beispiel wie oben:



ein anderes Beispiel:

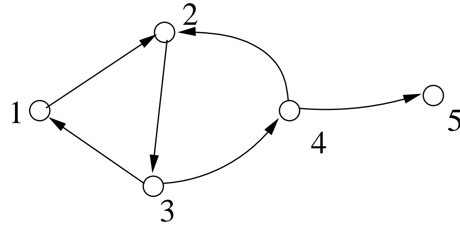


Platz zur Darstellung von $G = (V, E)$: $c + n + d \cdot |E|$ Speicherplätze mit

c : Initialisierung von A (konstant)
n : Größe des Arrays A
$d \cdot |E|$: 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:

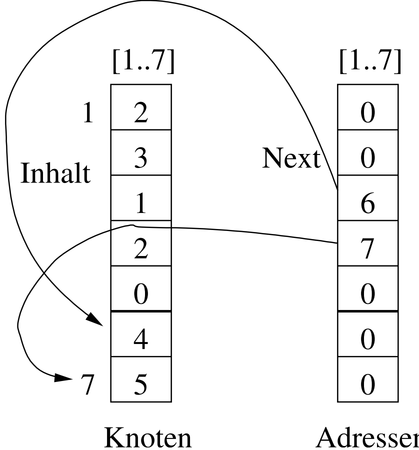


Die Größe des Arrays ist nicht unbedingt gleich der Kantenanzahl, aber sicherlich $\leq |E|+|V|$. Hier ist A[$1 \ldots 6$] falsch. Wir haben im Beispiel A[$1 \ldots 7$]. Die Größe des Arrays ist damit $|E|$ + # Knoten vom $Agrad = 0$.

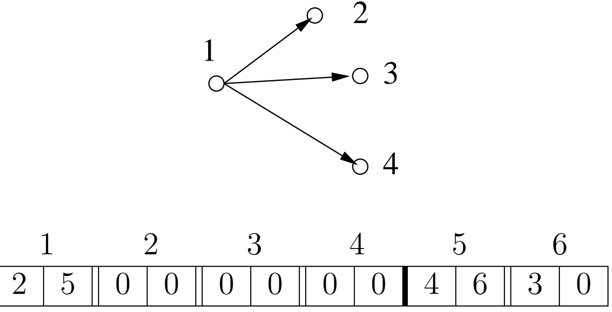
1		2		3		4		5		6		7	
2	0	3	0	1	6	2	7	0	0	4	0	5	0

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:



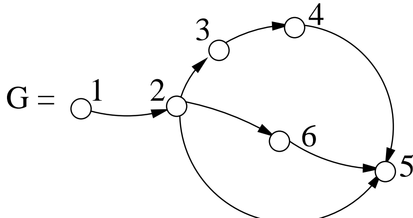
1		2		3		4		5		6	
2	5	0	0	0	0	0	0	4	6	3	0

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

Breitensuche

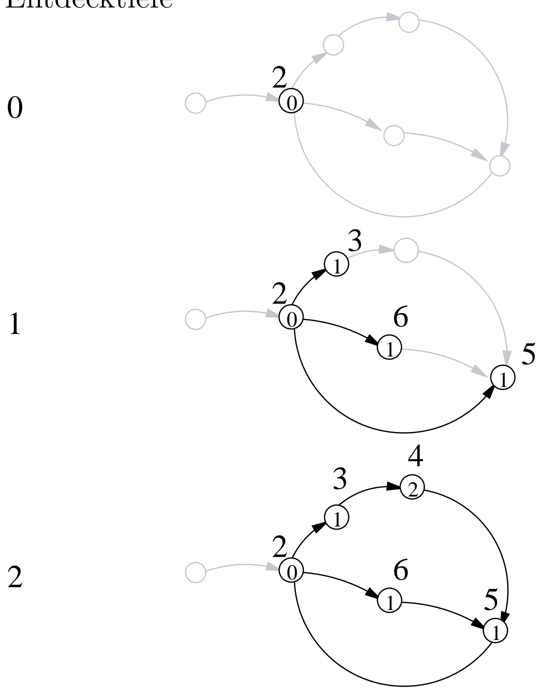
Gibt es einen Weg von u nach v im gerichteten Graphen $G = (V, E)$?
Algorithmische Vorgehensweise: Schrittweises Entdecken der Struktur

Dazu ein Beispiel:

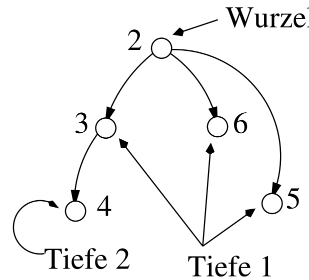


Weg von 2 nach 4 (4 von 2 erreichbar)?

Entdecktiefe



Der Breitensuchbaum ist ein Teilgraph von G und enthält alle Kanten, über die entdeckt wurde:



Implementierung mit einer Schlange:

Datenstruktur Schlange

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

5			
1	2	3	4

head = 1, tail = 2

Einfügen von 7 und 5

5	7	5	
1	2	3	4

head = 1, tail = 4

Löschen und 8 einfügen:

	7	5	8
1	2	3	4

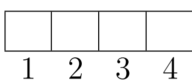
head = 2, tail = 1

Löschen:

		5	8
1	2	3	4

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 = Schlange

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

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

1	2	3	4	5	6
2					

head = 1, tail = 2

3, 6, 5 rein, 2 raus

1	2	3	4	5	6
	3	6	5		

head = 2, tail = 5

4 rein, 3 raus

1	2	3	4	5	6
		6	5	4	

head = 3, tail = 6

5, 6 hat Endecktiefe 1, 4 Endecktiefe 2

6 raus nichts rein

1	2	3	4	5	6
			5	4	

head = 4, tail = 2

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

Algorithmus Breitensuche (Breadth first search = BFS)

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

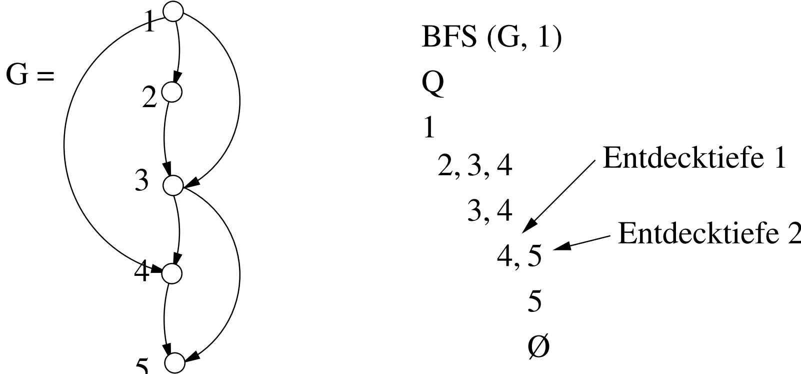
Datenstrukturen:

- Schlange $Q = Q[1..n]$ mit head und tail
- folgende Arrays: col$[1..n]$, um zu merken, ob Knoten bereits entdeckt wurde mit

- col$[u]$ = weiß $\Leftrightarrow$ u noch nicht entdeckt
- col$[u]$ = grau $\Leftrightarrow$ u entdeckt, aber noch nicht abgeschlossen, d.h. u in Schlange
- col$[u]$ = schwarz $\Leftrightarrow$ u entdeckt und abgearbeitet

```
/** Initialisierung */
for each u ∈ V
  col[u] = weiß;
col[s] = grau;                       // s Startknoten
Q = {s};                             // Initialisierung zu Ende.

/** Abarbeitung der Schlange */
while (Q ≠ ()){                      // Testbar mit tail≠ head
  u = Q[head];                       // u wird bearbeitet (expandiert)
  for each v ∈ Adj[u] {
    if (col[v] == weiß) {            // v wird entdeckt
      col[v] = grau;
      v in Schlange einfügen;
    }                                //Schlange immer grau.
  }
  u aus Q entfernen;
  col[u] = schwarz;                  // Knoten u ist fertig abgearbeitet
}
```

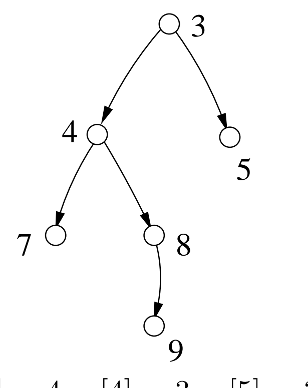
Noch ein Beispiel:



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.

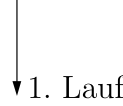


$$\pi[9] = 8,\ \pi[8] = 4,\ \pi[4] = 3,\ \pi[5] = 3,\ \pi[7] = 4$$

$\pi[u] = v \Rightarrow$ Kante (v, u) im Graph.

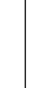
Verifikation? Hauptschleife ist Nummer 3. Wie läuft diese?

$Q_0 = (s), col_0[s] =$ gr, „$col_0 =$ w“ sonst.



$Q_1 = \underbrace{(u_1, \ldots, u_k)}_{Dist=1}, col_1[s] =$ sch, $col_1[u_i] =$ gr, w sonst.
$Dist(s, u_j) = 1$

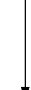
$(u_1, \dots, u_k)$ sind *alle!* mit $Dist = 1$.



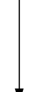
$Q_2 = (\underbrace{u_2, \dots, u_k}_{Dist1}, \underbrace{u_{k+1}, \dots, u_t}_{Dist2}) i \; col[u_1] = \text{sch}$

Alle mit $Dist = 1$ entdeckt.

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



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



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

$\vdots$

$\underbrace{(\dots)}_{Dist=3}$

und *alle* mit $Dist = 3$ erfasst (grau oder schwarz).

$\vdots$

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 1.4 (*Induktion über l*):

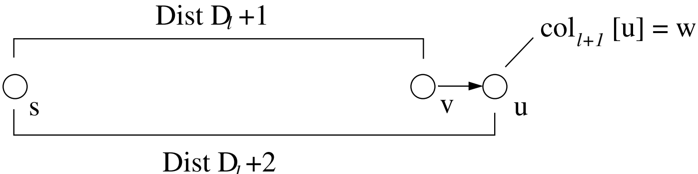
$l = 0$ (vor erstem Lauf), Gilt mit $D_0 = 0, U = (s), V = \emptyset$.
$l = 1$ (nach erstem Lauf), Mit $D_1 = 1, U =$ Nachbarn von $s, V = \emptyset$.
Gilt Invariante nach l-tem Lauf, also
$Q_l = (\overbrace{u_1, \ldots, u_k}^{Dist D_l}, \overbrace{v_1, \ldots, v_r}^{Dist D_l+1})$ mit (*), (**), $D_l = Dist(s, u_1)$.
Finde $l + 1$-ter Lauf statt (also v_1 ex.).
1. Fall: $u_1 = u_k$

- $Q_{l+1} = (v_1, \ldots, v_r \overbrace{\ldots}^{\text{Nachbarn von } u_1}), D_l + 1 = Dist(s, v_1)$

- 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:

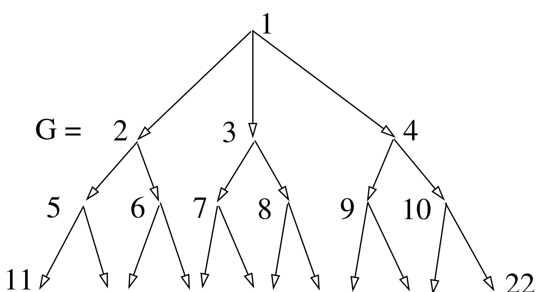


2. Fall: $u_1 \neq u_k$

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

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

Etwa:



$$
\begin{array}{ll}
Q_0 = (1) & D_0 = 0 \\
Q_1 = (2,3,4) & D_1 = 1, \text{ alle dabei!} \\
Q_2 = (\overbrace{(3,4}^{u}, \overbrace{5,6}^{v}) & D_2 = 1 \\
Q_3 = (5,6,7,8,9,10) & D_4 = 2(!) \text{ Alle dabei!}
\end{array}
$$

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

- $Q = (u), D = \Delta$
- 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.

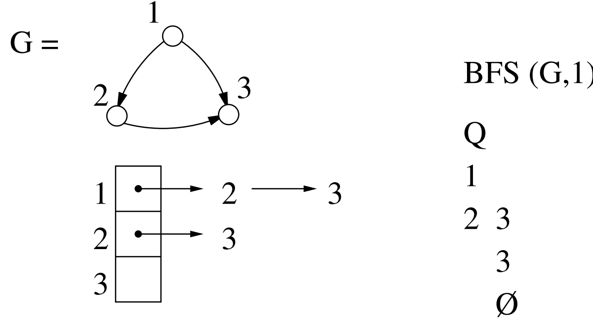
- $d[u] = Dist(s,u)$ für alle $u \in V$. Invariante um Aussage erweitern:
  Für alle erfassten Knoten ist $d[u] = Dist(s,u)$.
- Breitensuchbaum enthält kürzeste Wege. Invariante erweitern gemäß: Für alle erfassten Knoten sind kürzeste Wege im Breitensuchbaum.

Beachte: Dann kürzeste Wege s ○⟶○ u durch

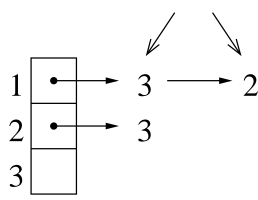
u ○⟵○ π[u] ○⟵○ π[π[u]] ... ○⟵○ π[π[..π[u] ..]] = s.

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○⟶○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.



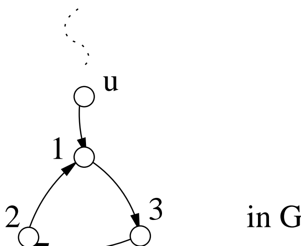
Beim Gang durch $(2,3)$ ist 3 grau.



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

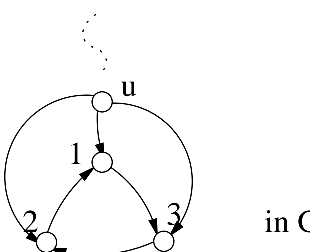
andere Farben: Übung 2

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:



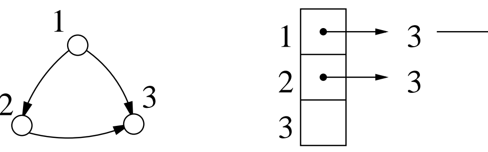
Alle drei Knoten $1, 2, 3$ werden bei der Expansion von u aus das erste Mal entdeckt. Trotzdem: Es gibt eine Kante, die bei Bearbeitung auf einen schwarzen Knoten führt, hier (2,1).

Folgerung 1.5:

> Kreis $\Longrightarrow$ Es gibt eine Kante u○⟶○v, so dass v schwarz ist, wenn u○⟶○v gegangen wird.

Ist ein Kreis daran erkennbar?

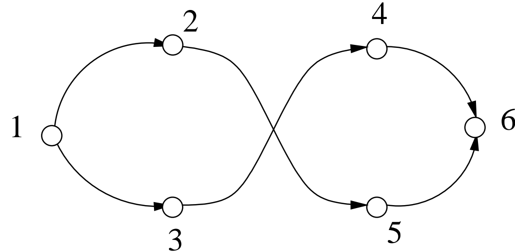
Leider nein:



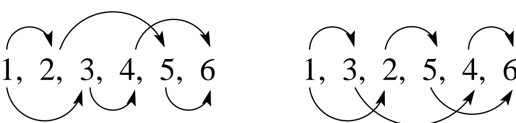
Dann 3 schwarz bei $(2,3)$, trotzdem kein Kreis.

Topologische Sortierung

Wie erkenne ich kreisfreie gerichtete Graphen?

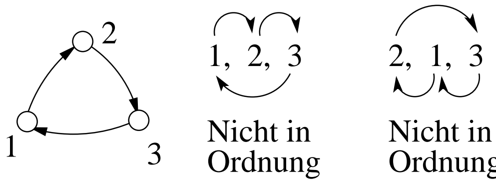


Anordnung der Knoten:



Alle Kanten gemäß der Ordnung.

Aber bei Kreis:



Definition 1.6 (*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.

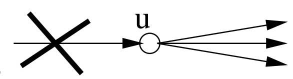
Satz 1.7:

> G hat topologische Sortierung $\iff$ G hat keinen Kreis.

Beweis 1.8:

> „⇒“ 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 $Egrad(u) = 0$. Also 

.

Wieso? Nehme irgendeinen Knoten u_1. Wenn $Egrad(u_1) = 0$, dann ✓.
Sonst $u_2 \rightarrow u_1$, nehme u_2 her. $Egrad(u_2) = 0$ ✓, sonst zu u_3 in G:
$u_3 \rightarrow u_2 \rightarrow u_1$
Falls $Egrad(u_3) = 0$, dann ✓, sonst $u_4 \rightarrow u_3 \rightarrow u_2 \rightarrow u_1$ 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 $|V|$.

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 im neuen Graphen mit $Egrad(u) = 0$
   $\vdots$
   Wie oben.

Wie findet man u gut?

1. Fall:
   Falls Adjazenzmatrix, $A = (a_{v_1,v_2})$, dann alle $a_{v_1,u} = 0$ für $v_1 \in V$.
2. Fall:
   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 $Egrad(u) = 0$.
   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 $Egrad(u) = 0$:

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

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

Algorithmus Verbessertes Top Sort

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

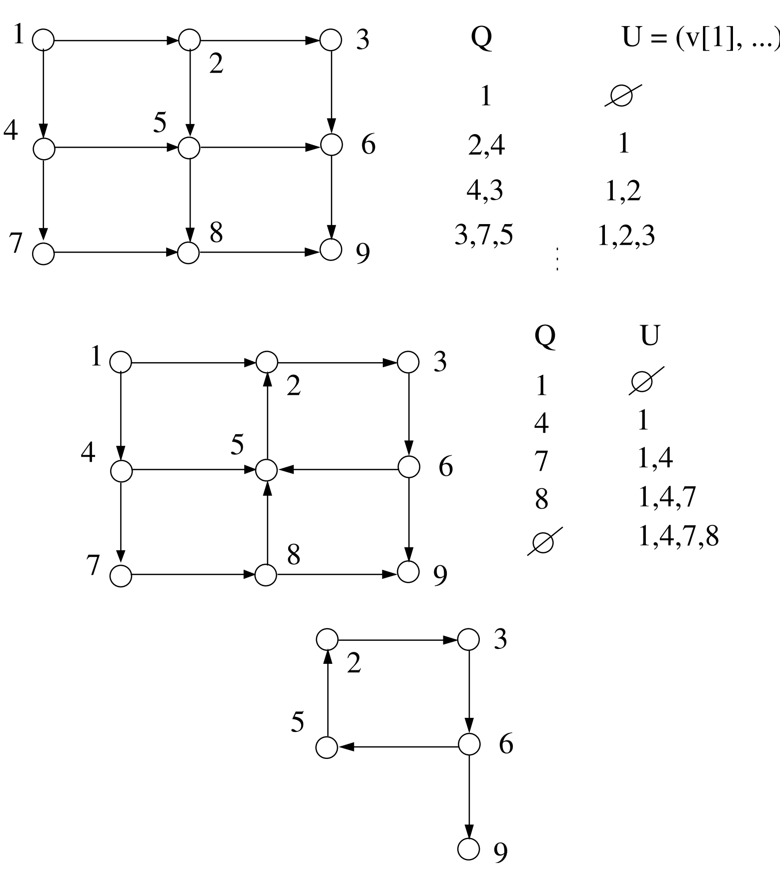
Weitere Datenstrukturen:

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

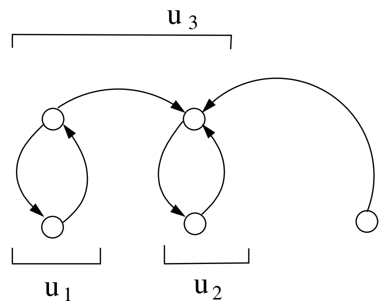
```
1./** Initialisierung */
  Egrad[v] auf 0, Q = ();
  for each v ∈ V {
    Gehe Adj[v] durch,
    zähle für jedes gefundene u
    Egrad[u] = Egrad[u]+1;
  }

2. /** Einfügen in Schlange */
  for each u ∈ V {
    if Egrad[u] = 0
      Q = enqueue(Q,u)
  }

3. /** Array durchlaufen */
  for i = 1 to n {
    if Q = ()
      Ausgabe „Kreis“; return;

4. /** Knoten aus Schlange betrachten*/
    v[i] = Q[head];
    v[i]=dequeue(Q);
```

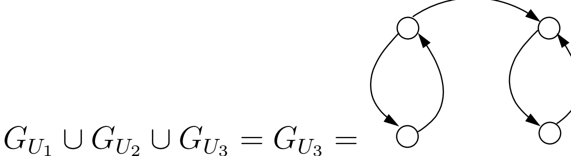
5. /** Adjazenzliste durchlaufen*/

```
for each u ∈ Adj[v[i]] {
   Egrad[u] = Egrad[u]-1;
   if Egrad[u] = 0
      Q = enqueue(Q,u)
   }
}
```

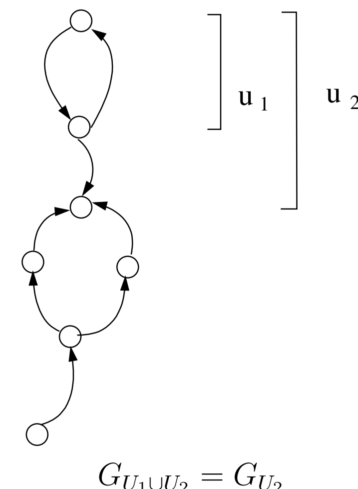


$G = (V, E)$. Für $U \subseteq V$ ist $G_U = (U, F), F = \{(u, v) \in E \mid u, v \in U\}$ der auf U induzierte Graph. Uns interessieren die $U \subseteq V$, so dass in G_U jeder $Egrad \geq 1$ ist. Seien $U_1, \ldots, U_k$ alle U_i, so dass in G_{U_i} jeder $Egrad \geq 1$ ist. Dann gilt auch in $G_{U_1 \cup \ldots \cup U_k} = G_X$ ist jeder $Egrad \geq 1$. Das ist der größte Graph mit dieser Eigenschaft. Am Ende von TopSort bleibt dieser Graph übrig ($X = \emptyset$ bis $X = V$ möglich).



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

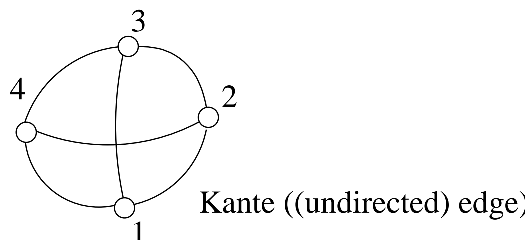
Größter induzierter Teilgraph, wobei jeder $Egrad \geq 1$ ist.



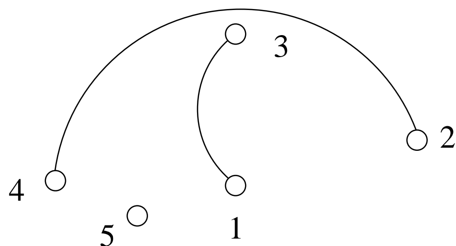
$G_{U_1 \cup U_2} = G_{U_2}$

2 Ungerichtete Graphen

Ein typischer ungerichteter Graph:



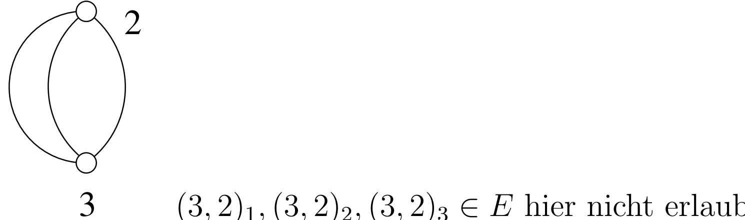
Auch einer:



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

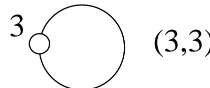
Beachte:

> 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:



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

und keine Schleifen



Definition 2.1 (*ungerichteter Graph*):

Ein ungerichteter Graph besteht aus 2 Mengen:

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

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

Folgerung 2.2:

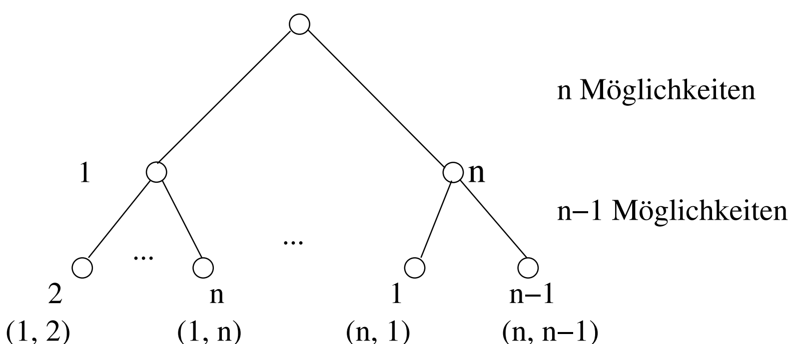
Ist $|V| = n$ fest.

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: $2^{n(n-1)}$)

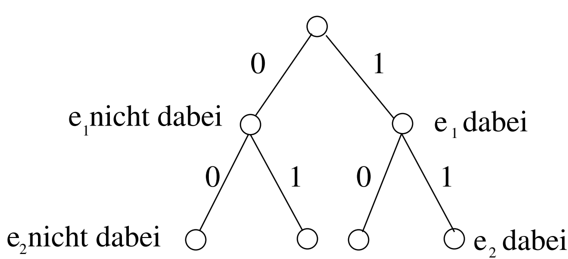
Beweis 2.3:

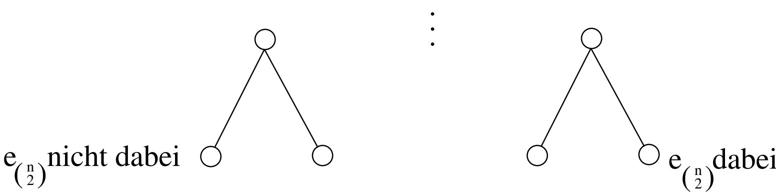
a) „Auswahlbaum "



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

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





Jedes Blatt $\Longleftrightarrow$ 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 $O(n^2)$.

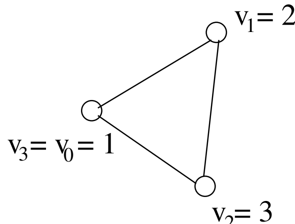
Adjazent, inzident sind analog zum gerichteten Graph zu betrachten.

$Grad(u) = \{\{u, v\} | \{u, v\} \in E\}$, wenn $G = (V, E)$. Es gilt immer $0 \leq Grad(u) \leq n - 1$.
Weg $(v_0, v_1, \dots, v_k)$ wie im gerichteten Graphen.
Länge $(v_0, v_1, \dots, v_k) = k$
Auch ein Weg ist bei $\{u, v\} \in E$ der Weg (u, v, u, v, u, v).
Einfach $\Longleftrightarrow v_0, v_1, \dots, v_k$ alle verschieden, d.h., $|\{v_0, v_1, \dots, v_k\}| = k + 1$
$(v_0, \dots v_k)$ geschlossen $\Longleftrightarrow v_0 = v_k$

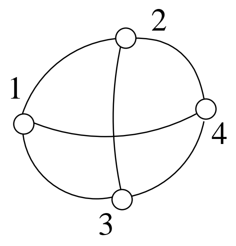
$(v_0, \dots, v_k)$ ist Kreis, genau dann wenn:

- $k \geq 3 (!)$
- $(v_0, \dots, v_{k-1})$ 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 $v_0 = 1, v_1 = 2, v_2 = 3, v_3 = 1 = v_0$.



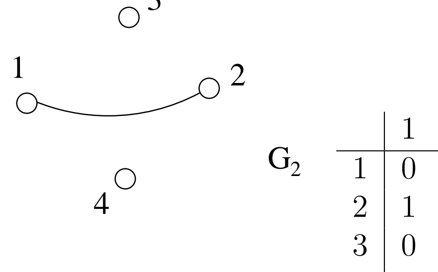
Adjazenzmatrix, symmetrisch (im Unterschied zum gerichteten Fall)



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 $\Longleftrightarrow a_{u,v} = a_{v,u}$



G_2

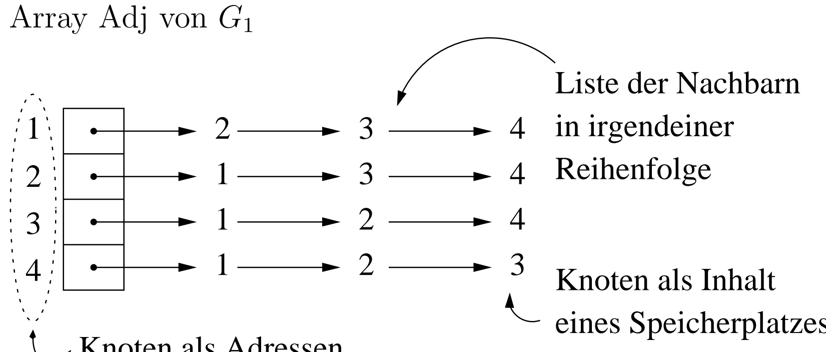
	1	2	3	4
1	0	1	0	0
2	1	0	0	0
3	0	0	0	0
4	0	0	0	0

Darstellung als Array:

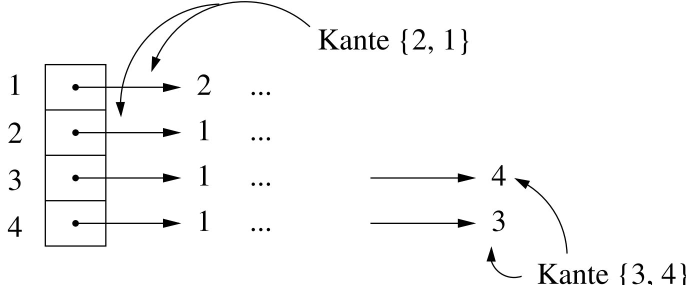
n^2 Speicherplätze

Adjazenzlistendarstellung:

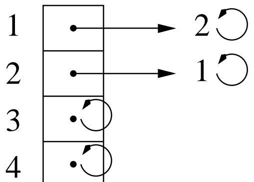
Array Adj von G_1



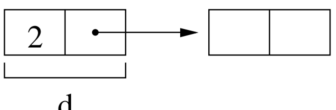
Jede Kante zweimal dargestellt, etwa:



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:



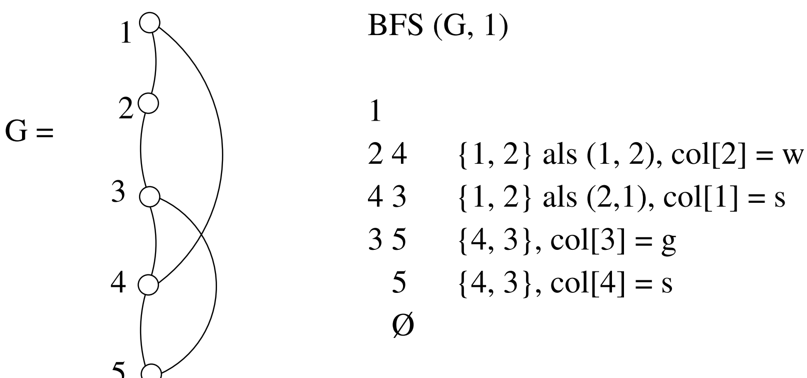
$O(|E|)$, wenn $|E| \geq \frac{n}{2}$.
$|E|$ Kanten sind inzident mit $\leq 2|E|$ Knoten.
Adjazenzlisten in Array wie im gerichteten Fall.

Algorithmus Breitensuche (BFS)

BFS(G, s) genau wie im gerichteten Fall
Eingabe: G in Adjazenzlistendarstellung
Datenstrukturen: Schlange $Q[1 \ldots n]$, Array $col[1 \ldots n]$

```
col[u] weiß für alle u ∈ V
```

```
col[s] = grau;
Q = {s}
while Q ≠ ∅ {
  u = Q[head]                   // u wird expandiert
  for each v∈ Adj[u] {          // {u, v} wird gegangen
    if col[v] = weiß {          // v wird entdeckt
      col[v] =grau;
      v in Schlange setzen; }
  u aus Q raus; col[u] = schwarz.
}
```

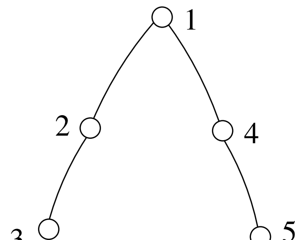


Beobachtung
Sei $\{u, v\}$ eine Kante.

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

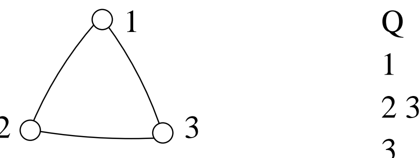
b) Ist der erste Lauf von u aus, so ist zu dem Zeitpunkt $col[v] = w$ oder $col[v] = g$.
Beim zweiten Lauf ist dann jedenfalls $col[u] = s$.

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

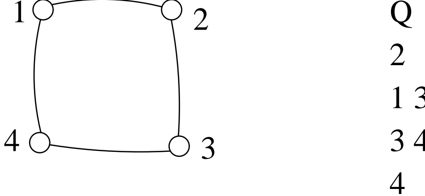


$\pi[3] = 2, \pi[2] = 1, \pi[4] = 1, \pi[5] = 4.$
Entdecktiefe durch $d[1 \dots n].\pi[v], d[v]$ werden gesetzt, wenn v entdeckt wird.

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



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



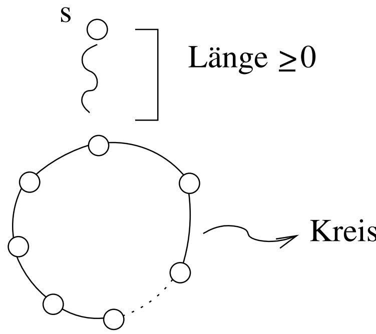
Beim (ersten) Gang durch $\{3,4\}$ ist $col[4] = grau$, d.h. $4 \in Q$.

Satz 2.4:

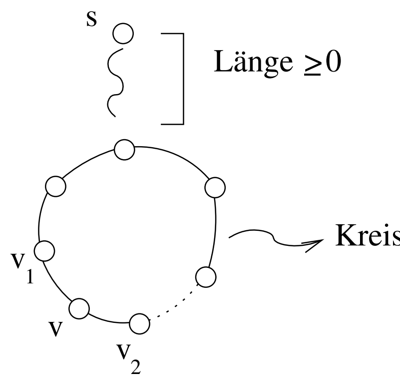
$G = (V, E)$ hat einen Kreis, der von s erreichbar ist $\iff$ Es gibt eine Kante $e = \{u, v\}$, so dass $BFS(G, s)$ beim (ersten) Gang durch e auf einen grauen Knoten stößt.

Beweis 2.5:

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



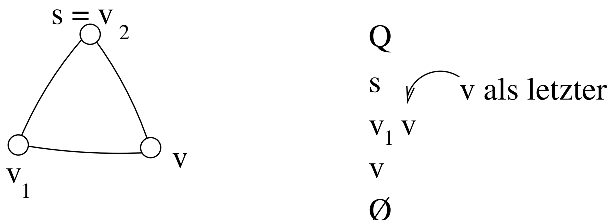
Sei v der Knoten auf dem Kreis, der als letzter (!) entdeckt wird.



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.)

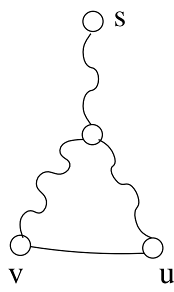
1. Fall: v über v_1 entdeckt, dann $Q = \ldots v_2 \ldots v$ nach Expansion von v_1. (Ebenso v_2.)

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



"⇐" Also bestimmen wir irgendwann während der Breitensuche die Situation $Q = u \dots v \dots$ und beim Gang durch $\{u, v\}$ ist v grau. Dann ist $u \neq s, v \neq s$.
Wir haben Wege s ⟶ u, s ⟶ 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.

□

Beachten nun den Fall

s
w
u x
v

Q
s
:
w ...
... x u
x u
u v
v grau, wenn {u, v}
gegangen wird

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

Definition 2.6 (*Zusammenhang*):

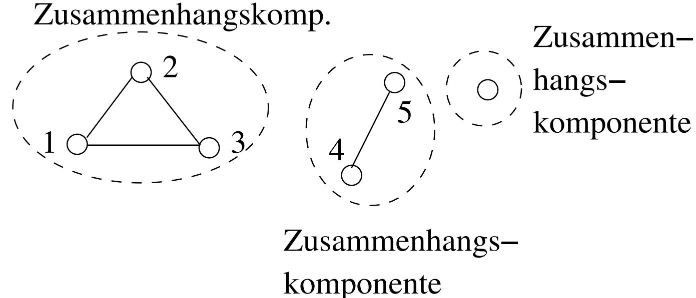
Sei $G = (V, E)$ ungerichteter Graph

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 1 ○—○ 2 —○ 3 keine Zusammenhangskomponente, da $\{1, 3\}$ fehlt. Ebenso wenig ist es 1 ○—○ 2, da nicht maximaler Teilgraph, der zusammenhängend ist.

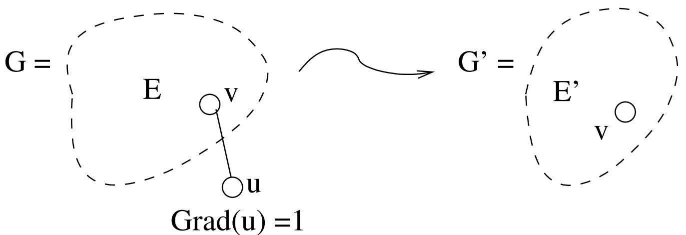
Satz 2.7:

Ist $G = (V, E)$ zusammenhängend. Dann gilt: G hat keinen Kreis $\iff |E| = |V| - 1$

Beweis 2.8:

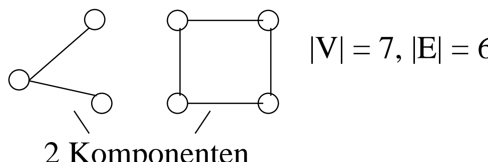
„$\Rightarrow$“ Induktion über $n = |V|$.
$n = 1$, dann ○, also $E = \emptyset$.
$n = 2$, dann ○—○, also ✓. Sei $G = (V, E), |V| = |V| = n + 1$ und ohne Kreis. Da G zusammenhängend ist und ohne Kreis, gibt es Knoten vom $Grad = 1$ in G.

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

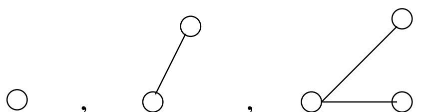


Dann gilt für $G' = (V', E')$:
G hat einen Kreis $\Rightarrow G'$ hat keinen Kreis
$\Rightarrow$ Induktionsvoraussetzung
$|E'| = |V'| - 1$
$\Rightarrow |E| = |V| - 1$

„$\Leftarrow$“ Beachte zunächst, dass die Aussage für G nicht zusammenhängend nicht gilt:



Also wieder Induktion über $n = |V|$. $n = 1, 2, 3$, dann



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 $|E| \geq n + 1 \left((|E| = \frac{1}{2} \sum_v Grad(v) \right)$

Einziehen von Kante und Löschen des Knotens macht Ind.-Vor. anwendbar.

Folgerung 2.9:

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 2.10:

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 $V_1 \dot\cup \ldots \dot\cup V_k = V$, $E_1 \dot\cup \ldots \dot\cup E_k = E$.) Dann gilt G_i ist kreisfrei $\iff |E| = |V_i| - 1$ und die Behauptung folgt.

□

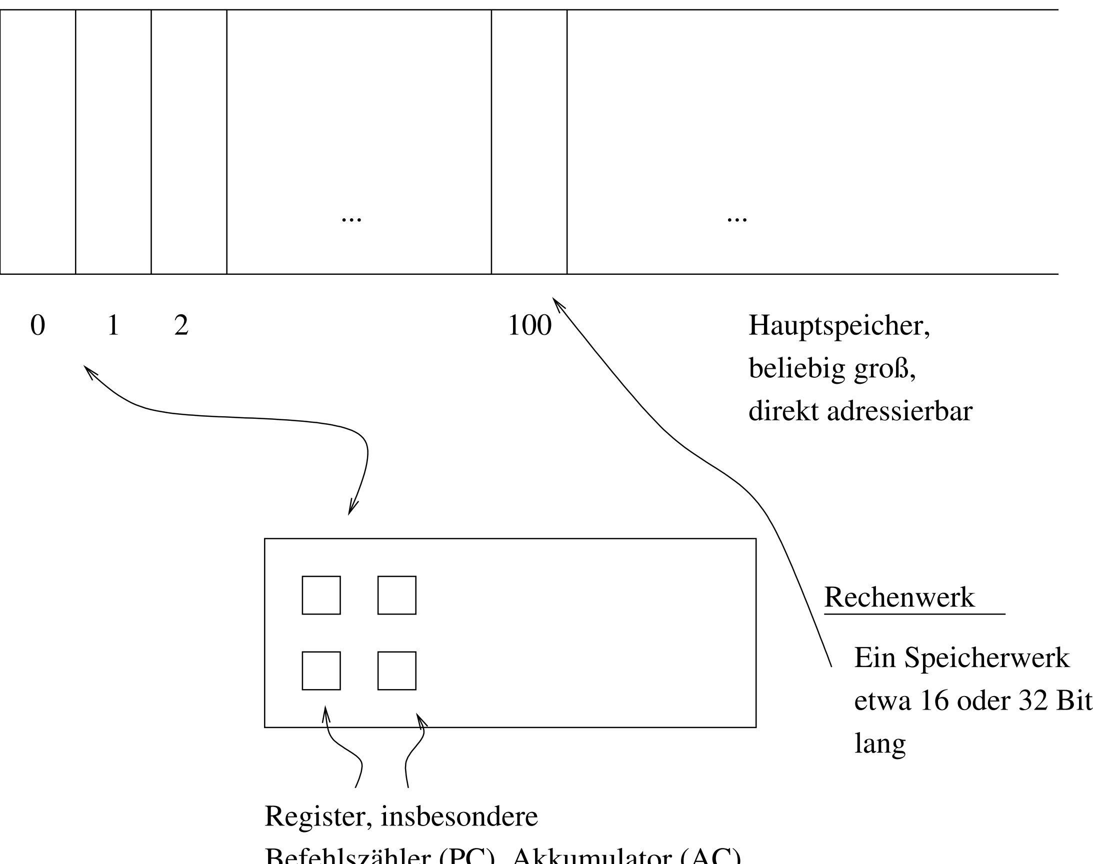
Algorithmus (Finden von Zusammenhangskomponenten)

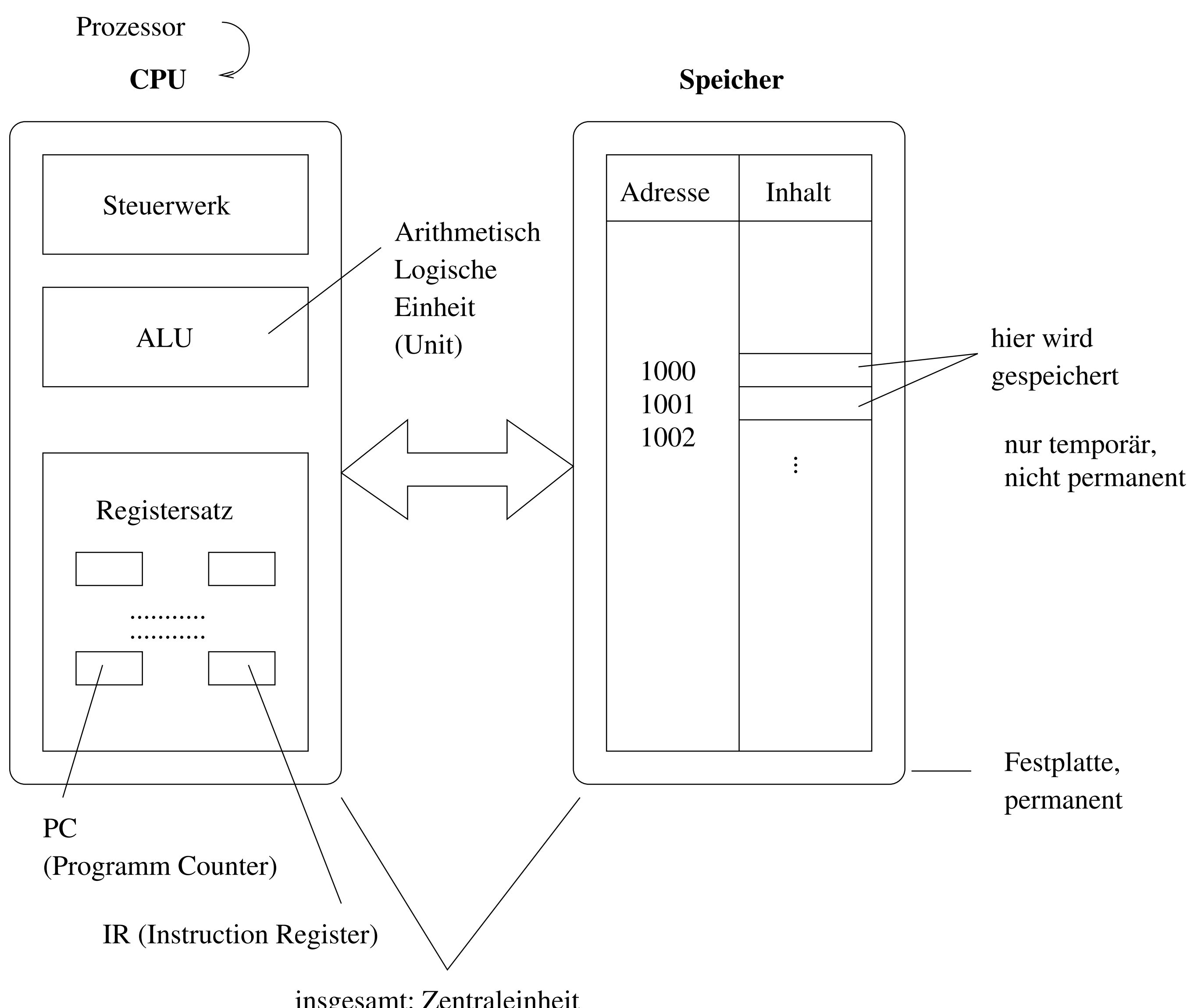
Eingabe: $G = (V, E)$ ungerichtet, Ausgabe: Array A$[1..|V|]$ mit A$[u]$ = A$[v] \iff u, v$ in einer Komponente
(Frage: Ist u von v erreichbar? In einem Schritt beantwortbar.)

```
1.    A auf Null initialisieren
2.    Initialisierung von BFS
3.    for each w ∈ V do
3 a.    if (colour[w] == weiß)
3 b.      then BFS(G,w) modifiziert, so dass Initialsierung,
          aber A[u]=w, falls u im Verlauf entdeckt wird.
```

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:


...
...
0
1
2
100
Hauptspeicher,
beliebig groß,
direkt adressierbar
Rechenwerk
Ein Speicherwerk
etwa 16 oder 32 Bit
lang
Register, insbesondere
Befehlszähler (PC), Akkumulator (AC)



Typische Befehle der Art

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

Programme 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.

```
/** Hier einmal ein Primzahltest.
* Durch „return“wird das Programm beendet-
*„Sprung ans Ende.“
*/
import Prog1Tools.IOTools;
public class PRIM{             // Name muss dem Dateinamen gleich sein!
public static void main(String[ ]args){
  long c, d;
  c = IOTools.readLong(„Einlesen eines langen c >= zum Test:“);
  if (c == 2){
    System.out.print(c +„ist PRIM“);
    return;
  }
  d = 2;
  while(d * d <= c){           //Warum reicht es, bei d*d > c auzuhören?
    if(c % d == 0){
      System.out.println(c +„ist nicht PRIM, denn“+ d +„teilt“+ c);
      return;                  // Hier ist das Programm zuende,
                               Erläuterung siehe oben
    }
    d++;                       //Die Schleife hört auf, nach dem Lauf, wo d
                               d so gesetzt wird, dass d * d > c ist!
                               An diesem Punkt muss immer aufgepasst werden!
  }
  System.out.println(„Die Schleife ist fertig ohne ordentlichen Teiler,
      ist“+ c +„PRIM“);
  }
}
```

Java-Programm zum Primzahltest. Nehmen dabei Bezug zu PRIM.java aus der Vorlesung AuP.

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(...)
               JpE              // Unbedingt ans Ende.

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

Als Flussdiagramm AuP 9.14 folgende.

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

Für das Programm PRIM gilt: # ausgefüllte Java-Zeilen bei Eingabe $c \leq a\sqrt{c} + b$
mit $a \cdot \sqrt{c}$ ... Schleife (Kopf und Rumpf) und b ... Anfang und Ende
a, b konstant, d.h. unabhängig von c!

Also auch # ausgeführte Maschinenbefehle bei Eingabe $c \leq a' \cdot \sqrt{c} + b'$
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 $\Longleftrightarrow$ 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 $BFS(G, s)$. Sei $G = (V, E)$ mit $|V| = n$, $|E| = m$.
Datenstrukturen initialisieren (Speicherplatz reservieren usw.): $O(n)$.

1. Array col setzen:		$O(n)$
2.		O(1)
3. Kopf der while-Schleife		
	Einmal	$O(1)$
4.	Einmal	$O(1)$
5.	Kopf einmal	$O(1)$
	(Gehen $Adj[u]$ durch)	
	Rumpf einmal	$O(1)$
	In einem Lauf der while-Schleife	
	wird 5. bis zu $n-1$-mal durchlaufen.	
	Also 5. in	$O(n)$.
	Rest	$O(1)$
Rest $O(1)$.		

Also: while-Schleife einmal:

$$O(1) + O(n) \text{ also } O(1) + O(1) + O(1) \text{ und } O(n) \text{ für 5.}$$

Durchläufe von 3. $\leq n$, 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 5. jedesmal nur $O(1)$ braucht.

Wie oft wird 5. insgesamt (über das ganze Programm hinweg) betreten? Für jede Kante genau einmal. Also das Programm hat in 5. 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:

$$\begin{array}{c}\text{Schleife } 1 \leq n \text{ Läufe} \\ \text{Schleife } 2 \leq m \text{ Läufe,}\end{array}$$

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 21
$G = (V, E), |V| = n, |E| = m$
Initialisierung: $O(n)$

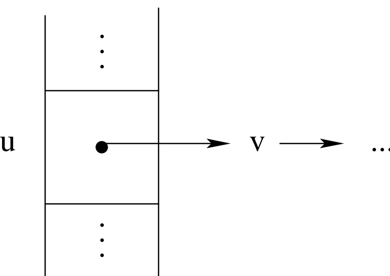
1. Array Egrad bestimmen: $O(m)$
   Knoten mit Grad 0 suchen: $O(n)$
   Knoten in top. Sortierung tun und Adjazenzlisten anpassen: $O(1)$

2. Wieder genauso: $O(m) + O(n) + O(1)$
   ⋮

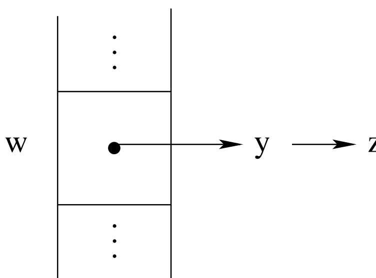
Nach Seite 22 bekommen wir aber Linearzeit heraus:
1. und 2. Egrad ermitteln und Q setzen: $O(m)$
3. Ein Lauf durch 3. $O(1)$! (keine Schleifen), insgesamt n Läufe. Also: $O(n)$
4. insgesamt $O(n)$
5. jede Kante einmal, also insgesamt $O(m)$
Zeit $O(m + n)$, Zeitersparnis durch geschicktes Merken.

4 Tiefensuche in gerichteten Graphen

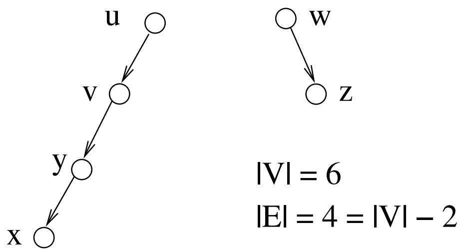
Fangen bei n zum Zeitpunkt 1 an, Gehen eine Kante, entdecken v zum Zeitpunkt 2, (b). Adjazenzliste



v ist der erste Knoten. Dann wird y in (c) entdeckt. u, v, y sind offen = grau. In (d) entdecken wir das x. Kante (x, v) wird zwar gegangen, aber v nicht darüber entdeckt.
Dann bei w weiter. Adjazenzliste ist:



Tiefensuchwald = Kanten, über die entdeckt wurde.



Wald = Menge von Bäumen
$\Pi[x] = y, \Pi[y] = v, \Pi[v] = u, \Pi[u] = u$ (alternativ nil), $\Pi[w] = w$ (alternativ nil)

Algorithmus Tiefensuche

Eingabe $G = (V, E)$ in Adjazenzlistendarstellung, gerichteter Graph, $n = |V|$.

d[1 . . . n] . . . Entdeckzeit(discovery-Zeit)
Nicht(!) Entfernung wie vorher. Knoten wird grau.
f[1 . . . n] . . . Beendezeit (finishing), Knoten wird schwarz.

Π[1 . . . n] . . . Tiefensuchwald, wobei $\Pi[u] = v \Longleftrightarrow$ (v → u)
$\Pi[u]$ = Knoten, über den u entdeckt wurde
col[1 . . . n] . . . aktuelle Farbe

DFS(G)

```
/** 1. Initialisierung */
for each u in V{
  col [u]: = weiß;
  Π[u]: = nil;
}
time: = 0;
/** 2. Hauptschleife,
* Aufruf von DFS-visit
* nur, wenn col[u] = weiß
*/
for each u ∈ V{
  if (col[u] == weiß)
    {DFS-visit}
}
```

DFS-visit(u)

```
1. col[u] = grau;          //Damit ist u entdeckt.
2. d[u] = time; time = time + 1;
3. for each v ∈ Adj[u]{    //u wird bearbeitet
4.   if(col[v] == weiß){   //(u, v) untersucht
5.     Π[v] = u;           //v entdeckt
6.     DFS-visit(v);
     }
   }                       // Die Bearbeitung von u ist hier zuende.
```

```
  /** Sind alle Knoten aus Adj[u] grau oder schwarz,
  * so wird u direkt schwarz.
  */
7. col[u]: = schwarz;
8. f[u]: = time;
  time: = time +1;       //Zeitzähler geht hoch bei Entdecken und Beenden
```

Es ist $\{d[1], \ldots, d[m], f[1], \ldots, f[m]\} = \{1, \ldots, 2n\}$. Man kann über die Reihenfolge nicht viel sagen. Möglich ist:

u → v → w

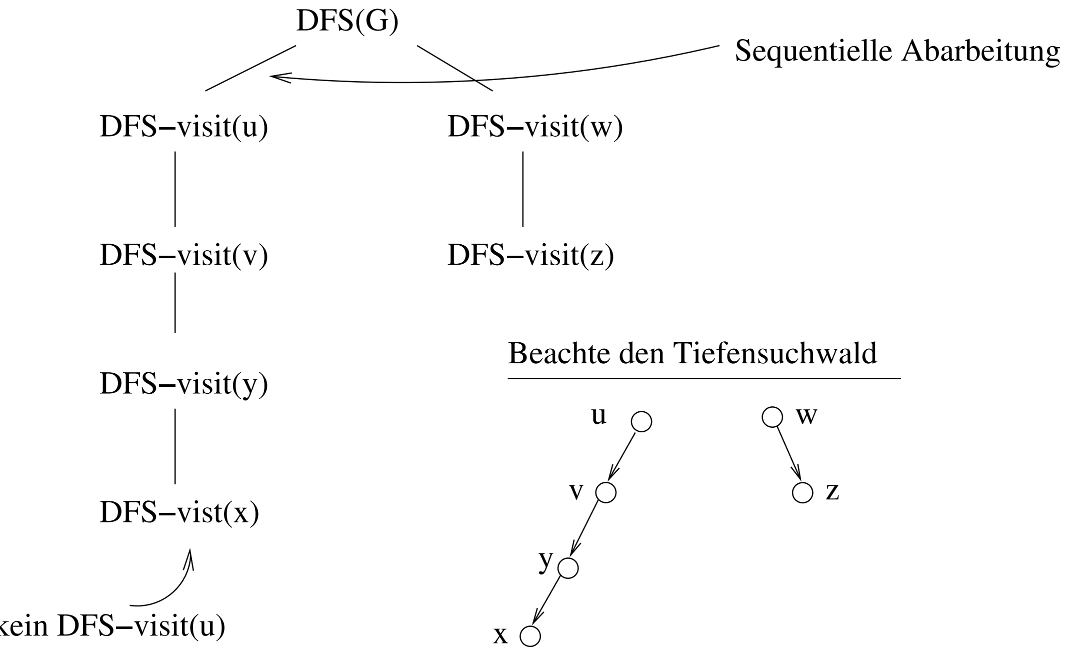
Einmal:
$d[u] = 1, d[v] = 2, d[w] = 3$
und
$f[w] = 4, f[v] = 5, f[u] = 6.$
Aber auch:
$d[w] = 1, f[w] = 2, d[v] = 3, f[v] = 4,$
$d[u] = 5, f[u] = 6$

DFS-visit(u) wird nur dann aufgerufen, wenn col[u] = weiß ist. Die Rekursion geht nur in weiße Knoten.

Nachdem alle von u aus über weiße(!) Knoten erreichbare Knoten besucht sind, wird col[u] = schwarz.
Im Nachhinein war während eines Laufes

- col[u] = weiß, solange time < d[u]
- col[u] = grau, solange d[u] < time < f[u]
- col[u] = schwarz, solange f[u] < time

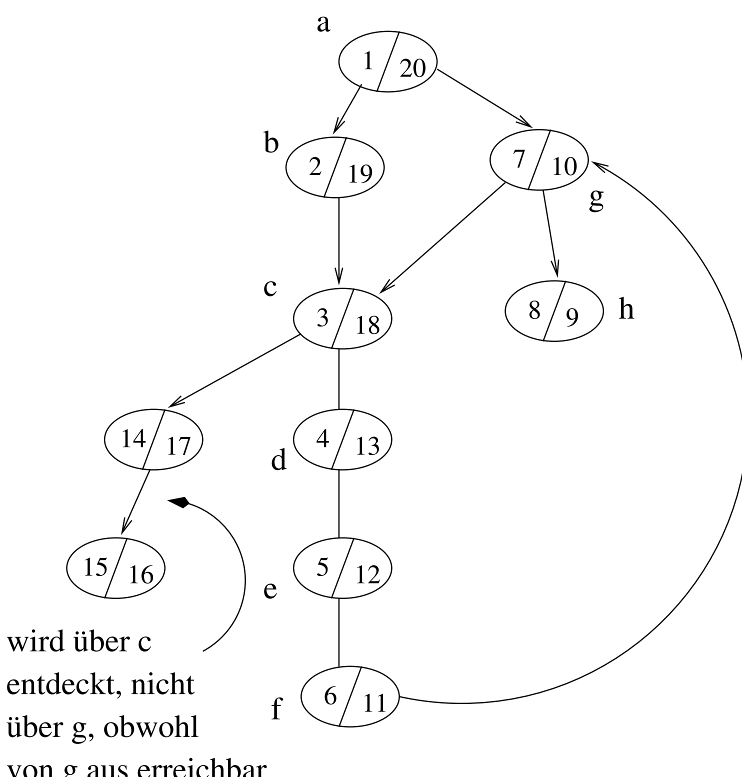
Unser Eingangsbeispiel führt zu folgendem Prozeduraufrufbaum:



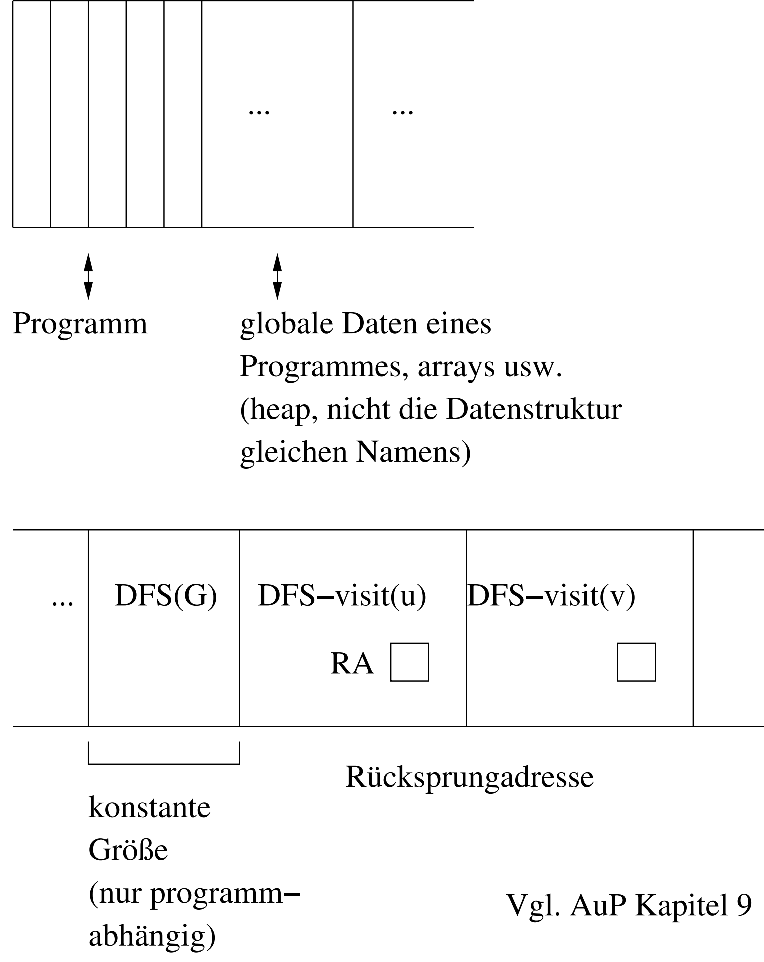
Erzeugung: Präorder(Vater vor Sohn)
Beendigung: Postorder(Sohn vor Vater)

Betrachten wir nun noch einmal folgendes Beispiel:

Beispiel 4.1:



Wie erfolgt die Ausführung der Prozeduraufrufe auf unserem Maschinenmodell der RAM? Organisation des Hauptspeichers als keller von frames:



Verwaltungsaufwand zum Einrichten und Löschen eines frames: $O(1)$, programmabhängig. Hier werden im wesentlichen Adressen umgesetzt.

> **Merkregel** zur Zeitermittlung:
>
> Durchlauf jeder Programmzeile inklusive Prozeduraufruf ist $O(1)$.
> Bei Prozeduraufruf zählen wir so:
> Zeit für den Aufruf selbst (Verwaltungsaufwand) $O(1)$ + Zeit bei der eigentlichen Ausführung.

Satz 4.2:

Für $G = (V, E)$ mit $n = |V|$ und $m = |E|$ braucht DFS(G) eine Zeit $O(n + m)$.

Beweis 4.3:

DFS(G)	1. $O(n)$, 2. $O(n)$ ohne Zeit in DFS-visit(u).
DFS-visit(u)	$O(n)$ für Verwaktungsaufwand insgesamt.

DFS-visit(u) wird nur dann aufgerufen, wenn col[u] = weiß ist. Am Ende des Aufrufs wird col[u] = schwarz.
1. und 2. einmal $O(1)$ insgesamt $O(n)$.
3., 4. 5. 6. ohne Zeit in DFS-visit(v) insgesamt $O(m)$.

(3., 4. für jede Kante einmal, also $O(m)$. 5. und 6. für jedes Entdecken, also $O(n)$).
7. und 8. $O(n)$ insgesamt.
Also tatsächlich $O(n+m)$

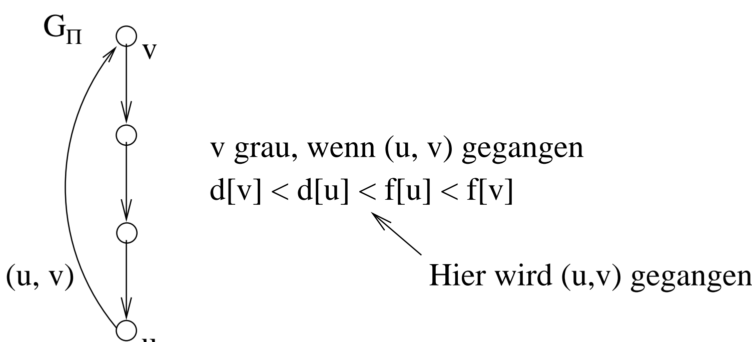
Definition 4.4 (*Tiefensuchwald*):

Sei $G = (V, E)$ und sei DFS(G) gelaufen. Sei $G_\Pi = (V, E_\Pi)$ mit $(u, v) \in E_\Pi \iff \Pi[v] = u$ der Tiefensuchwald der Suche.

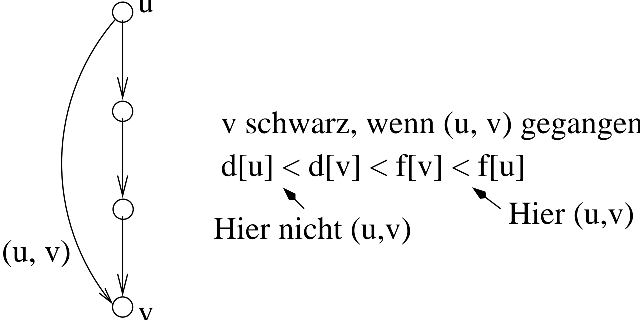
Klassifizieren der Kanten von G.
Sei $(u, v) \in E$.

(a) (u, v) Baumkante $\iff (u, v) \in E_\Pi (\Pi[v] = u)$

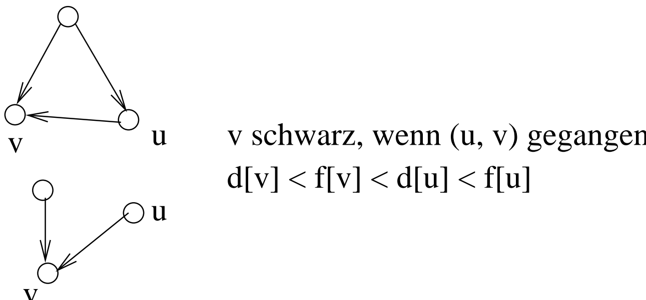
(b) (u, v) Rückwärtskante $\iff (u, v) \notin E_\Pi$ und v Vorgänger von u in G_Π



(c) (u, v) Vorwärtskante $\iff (u, v) \notin E_\Pi$ und v Nachfolger von u in G_Π



(d) (u, v) Kreuzkante $\iff (u, v) \notin E_\Pi$ und u weder Nachfolger noch Vorgänger von v in G_Π



□

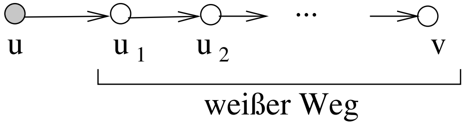
Noch eine Beobachtung für Knoten u, v. Für die Intervalle, in denen die Knoten aktiv (offen, grau) sind. gilt:
Entweder
$d[u] < d[v] < f[v] < f[u]$

DFS-visit(u) ⋮DFS-visit(v)

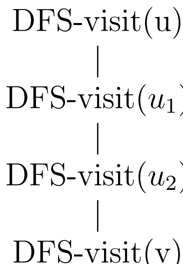
oder
$d[u] < f[u] < d[v] < f[v]$

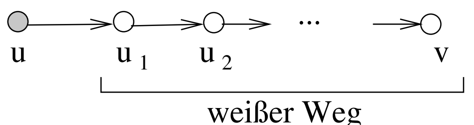
oder umgekehrt. Folgt aus Kellerstruktur des Laufzeitkellers.

Satz 4.5 (*Weißer-Weg-Satz*):

> v wird über u entdeckt (d.h. innerhalb von DFS-visit(u) wird DFS-vist(v) aufgerufen)
> $\Longleftrightarrow$
> Zum Zeitpunkt $d[u]$ gibt es einen Weg
>
> 
>
>
> in G.

Beweis 4.6:

> „⇒" Aufrufstruktur, wenn DFS-visit(v) aufgerufen wird:
>
> 
>



„⇐“ Liege also zu $d[u]$ der weiße Weg u ● → ○ v_1 → ○ v_2 → ··· → ○ v_k → ○ v vor. Das Problem ist, dass dieswe Weg keineswegs von der Suche genommen werden muss. Trotzdem, angenommen v wird nicht über u entdeckt, dann gilt nicht

$$d[u] < d[v] < f[v] < f[u],$$

sondern

$$d[u] < f[u] < d[v] < f[v].$$

Dann aber auch nach Programm

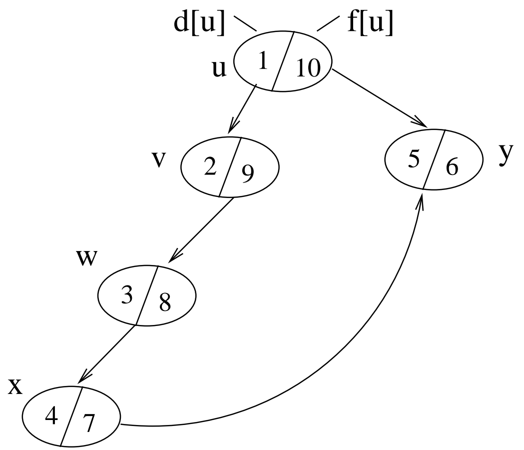
$$d[u] < f[u] < d[v_k] < f[v_k]$$
$$\cdots$$

dann

$$d[u] < f[u] < d[v_1] < f[v_1]$$

was dem Programm widerspricht.

Noch ein Beispiel:
Beispiel 4.7:



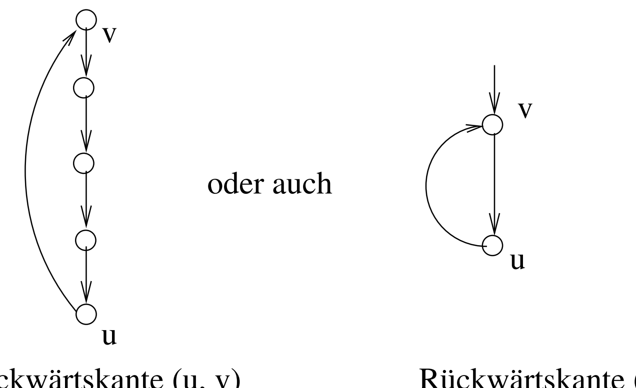
Zum Zeitpunkt 1 ist (u, y) ein weißer Weg. Dieser wird nicht gegangen, sondern ein anderer. Aber y wird in jedem Fall über u entdeckt!

Kreis feststellen!
Satz 4.8:

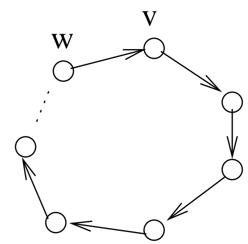
Ist $G = (V, E)$ gerichtet, G hat Kreis $\iff$ DFS(G) ergibt eine Rückwärtskante.

Beweis 4.9:

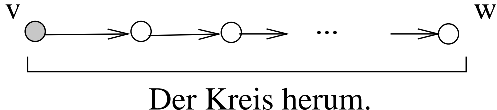
„$\Leftarrow$" Sei (u, v) eine Rückwärtskante. Dann in G_Π, dem Tiefensuchwald



Also Kreis in G.
„$\Rightarrow$" Hat G einen Kreis, dann also



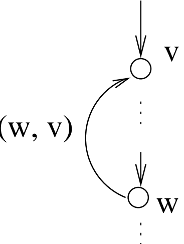
Sei v der erste Knoten auf dem Kreis, den DFS(G) entdeckt. Dann zu dem Zeitpunkt weißer Weg



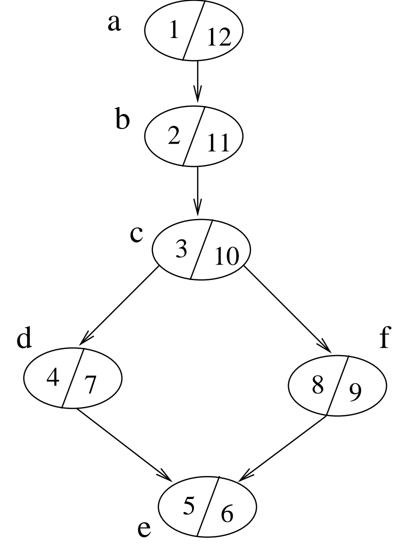
Satz 4.10 (*Weißer-Weg-Satz*):

$$d[v] < d[w] < f[w] < f[v]$$

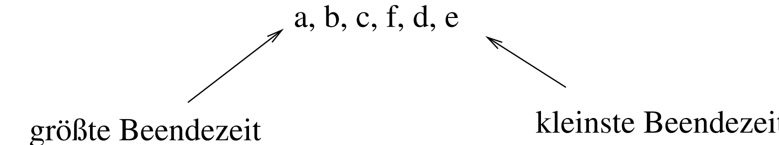
Also wenn (w, v) gegangen und ist $col[v] = grau$ also Rückwärtskante. Alternativ, mit Weißer-Weg-Satz:



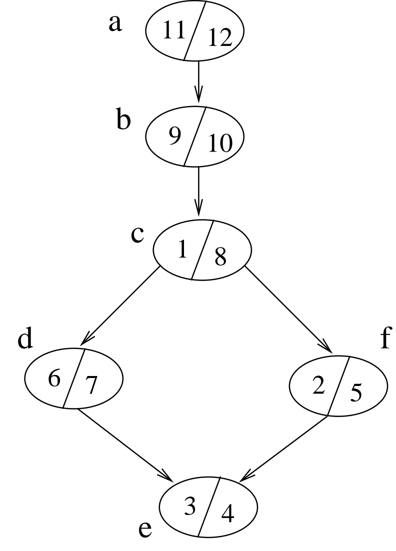
w ist irgendwann unter v und dann ist (w, v) Rückwärtskante. Man vergleiche hier den Satz und den Beweis zu Kreisen in ungerichteten Graphen in Kapitel 2.
Dort betrachte man den zuletzt(!) auf dem Kreis entdeckten Knoten.
Wir können auch topologisch sortieren, wenn kreisfrei.



Nach absteigender Beendezeit sorteieren.



Aber auch



Absteigende Beendezeit: a, b, c, d, f, e. **Satz 4.11**:

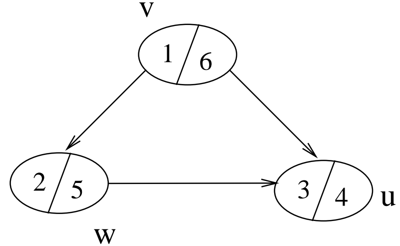
Sei $G = (V, E)$ kreisfrei. Lassen nun DFS(G) laufen, dann gilt für alle $u, v \in V, u \notin v$, dass
$f[u] < f[v] \Rightarrow (u, v) \notin E$ Keine Kante geht von kleinerer nach größerer Beendezeit.

Beweis 4.12:

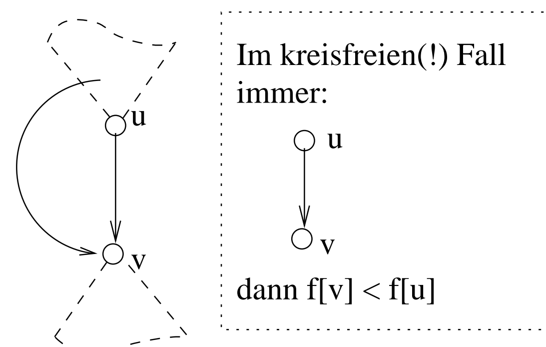
Wir zeigen: $(u, v) \in E \Rightarrow f[u] > f[v]$.

1. Fall:$d[u] < d[v]$
   Dann Weißer Weg (u, v) zu $d[u]$, also $d[u] < d[v] < f[v] < f[u]$ wegen Weißer-Weg-Satz.
2. Fall:$d[u] > d[v]$
   Zum Zeitpunkt $d[v]$ ist $col[u] = wei$ß. Aber da kreisfrei, wird u nicht von v aus entdeckt, da sonst (u, v) Rückwärtskante ist und damit ein Kreis vorliegt. Also kann nur folgendes sein:
   $d[v] < f[v] < d[u] < f[u]$ und es ist $f[v] < f[u]$

Beachte aber



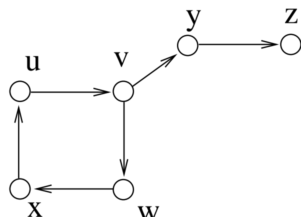
$(u, v) \in E$ zbd $f[u] < f[v]$, der Satz gilt nicht. Aber wir haben ja auch einen Kreis!
$\neg(A \Rightarrow B)$ bedeutet $(A \wedge \neg B)$ Im kreisfreien Fall etwa so:



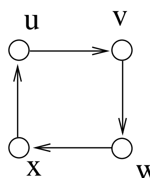
Wird u vor v weiß, dann ist klar $f[v] < f[u]$ Wird aber v vor u weiß, dann wird u nicht nachfolger von v, also auch dann $f[v] < f[u]$, Weg kreisfrei.

5 Anwendung Tiefensuche: Starke Zusammenhangskomponenten

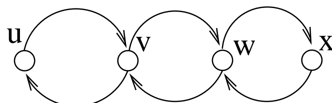
Starke Zusammenhangskomponenten beziehen sich immer nur auf gerichtete Graphen. Der Begriff der Zusammenhangskomponente im ungerichteten Graph besagt, dass zwei Knoten zu einer solchen Komponente genau dann gehören, wenn man zwischen ihnen hin und her gehen kann. Die analoge Suche führt bei gerichteten Graphen auf starke Zusammenhangskomponenten.
Einige Beispiele:



Starke Zusammenhangskomponente



y kann nicht dabei sein. (y, z) ist keine starke Zusammenhangskomponente, also sind y und z alleine, Interessante Beobachtung: Scheinbar gehört nicht jede Kante zu einer starken Zusammenhangskomponente.



Der ganze Graph ist eine starke Zusammenhangskomponente. Zu u, w haben wir z.B. die Wege (u, v, w) und (w, v, u). Die Kanten der Wege sind alle verschieden, die Knoten nicht!
Damit sind die Vorbereitungen für folgende offizielle Definition getroffen: **Definition 5.1** (*stark zusammenhängend*):

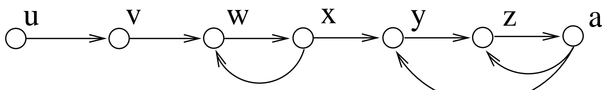
> Ist $G = (V, E)$ gerichtet, so ist G stark zusammenhängend $\iff$ Für alle $u, v \in V$ gibt es Wege (u, v) und (v, u)

Also noch einmal: Ist (v, u) stark zusammenhängend?
Es bietet sich an, einen Graphen, der nicht stark zusammenhängend ist, in seine stark zusammenhängenden Teile aufzuteilen. **Definition 5.2** (*Starke Zusammenhangskomponente, starke Komponente*):

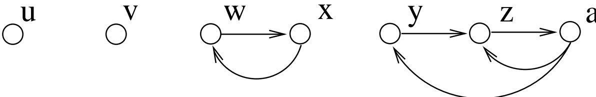
> Ist $G = (V, E)$ gerichteter Graph. Ein Teilgrpah $H = (W, F)$ von G, d.h. $W \subseteq V$ und $F \subseteq E$ ist eine starke Zusammenhangskomponente $\iff$ H ist ein maximaler indizierter Teilgraph von G, der stark zusammenhängend ist. Man sagt auch: *starke Komponente.*

Was soll das das maximal? Wir haben folgende Anordnung auf dem induzierten Teilgraphen von G:

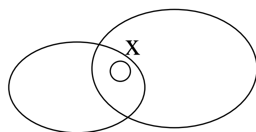
Sei$H = (W, F), H' = (W', F')$, dann $H \leq H'$, genau dann, wenn H Teilgraph von H' ist, d.h. $W \subseteq W'$. Ein maximaler stark zusammenhängender Teilgraph ist einer, der keinen weiteren stark zusammenhängenden über sich hat. Also



dann sind die starken Komponenten



Nicht y → z → a , obwohl stark zusammenhängend, wegen der Maximalität.
Man sieht noch einmal: Jeder Knoten gehört zu genau einer starken Komponente. Warum kann ein Knoten nicht zu zwei verschiedenen starken Komponenten gehören? Etwa in:



Wir gehen das Problem an zu testen, ob ein Graph stark zusammenhängend ist, bzw, die starken Komponenten zu finden. Ein einfacher Algorithmus wäre etwa: Prüfe für je zwei Knoten (u, v), ob es einen Weg von u nach v und von v nach u gibt. Etwas genauer:
Eingabe $G = (V, E), V = \{1, \ldots, n\}$

1. Generiere alle Paare (u, v) von Knoten mit $u < v$.

2. Für jedes (u, v) aus 1. führe aus:

   BFS(G, u), BFS(G,v)
   Teste, ob v und u gefunden werden. Fall nicht, ist die Ausgabe „Nicht stark zusammenhängend (Weg (u, v))“

3. Ausgabe: „stark zusammenhängend“

Zeitabschätzung:

1. $O(|V|^2)$

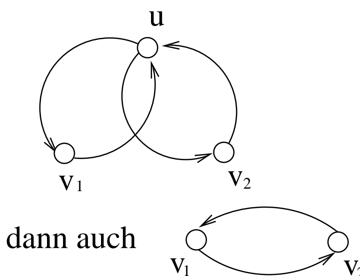
2. $O(|V|^2 \cdot (|V| + |E|)$ also $O(|V|^2 \cdot |E|)$,
   sofern $|E| \geq \frac{1}{2} \cdot |V|$.
   $O(|V|^2 \cdot |E|)$ kann $O(|V|^4)$ sein.

Es geht etwas besser:

1. BFS(G, u) für einen Knoten u, so modifiziert, dass alle gefundenen v auf Liste gespeichert.

2. Für jedes in 1. gefundene v wird folgendes ausgeführt:

   BFS(G, V). Teste, ob u hier gefunden wird.



Zeitabschätzung:

1. $O(|V| + |E|)$
2. $O(|V| \cdot (|V| + |E|)$ also $O(|V| \cdot |E|)$
   sofern $|E| \geq \frac{1}{2} \cdot |V|$.
   $|V| \cdot |E|$ kann bis $|V|^3$ gehen ($|V| = 10$, dann $|V|^3 = 1000$!)

Bemerkung

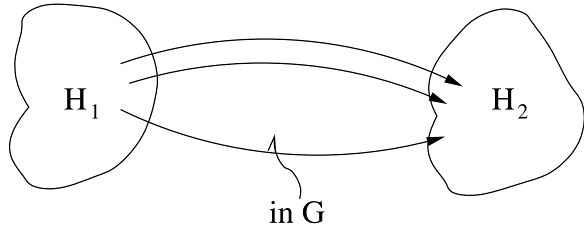
Sei $G = (V, E)$ ein gerichteter Graph uns sei $v \in V$. G ist stark zusammenhängend genau dann, wenn für alle $w \in V$ gilt $u \rightsquigarrow w$, $w \rightsquigarrow v$

Beweis 5.3:

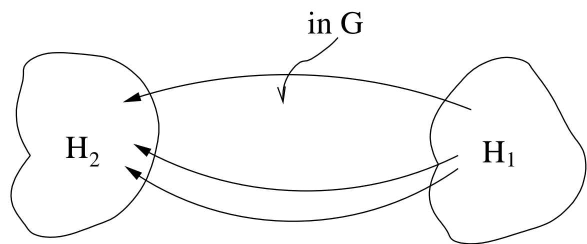
„⇒“ klar nach Definition

„⇐“ Sind $x, y \in V$, dann $x \to v \to y$ und $y \to v \to x$ nach Voraussetzung.

Finden von starken Zusammenhangskomponenten in $O(|V| + |E|)$! Schlüsselbeobachtung ist: Sind $H_1 = (W_1, F_1), H_2 = (W_2, F_2)$ zwei starke Komponenten von $G = (V, E)$, dann



oder



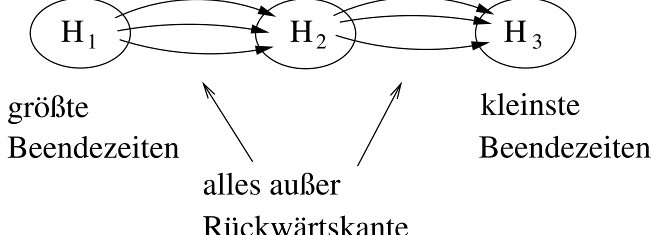
und allgemeiner:
Satz 5.4:

> Fassen wir die starken Komponenten als einzelnen Knoten auf und verbinden sie genau dann, wenn sie in G verbunden sind, so ist der entstehende gerichtete Graph kreisfrei.

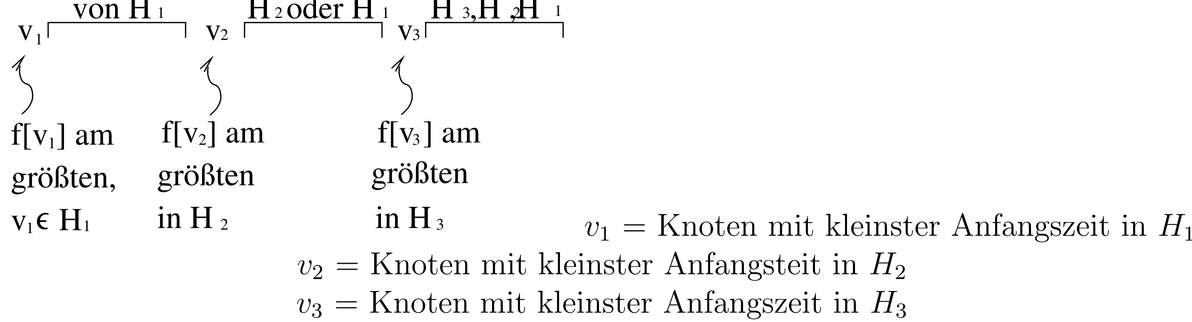
Beweis 5.5:

> Ist $(H_1, H_2, \ldots, H_k, H_1)$ ein Kreis auf der Ebene der Komponenten, so gehören alle H_i zu einer Komponente.

Wir können die Komponenten topologisch sortieren!
Immer ist nach Tiefensuche:

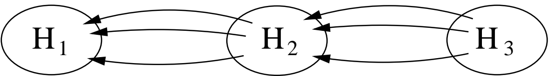


Ordnen wir die Knoten einmal nach absteigender Beendezeit, dann immer:

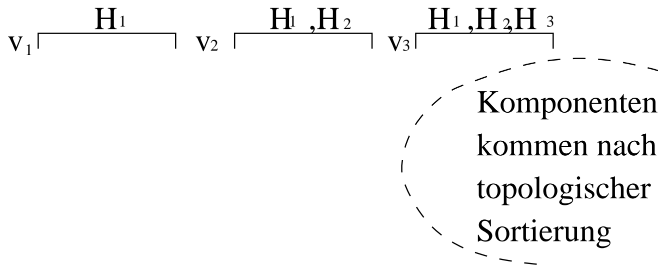


v_1 = Knoten mit kleinster Anfangszeit in H_1
v_2 = Knoten mit kleinster Anfangsteit in H_2
v_3 = Knoten mit kleinster Anfangszeit in H_3

(Weißer-Weg-Satz) Bem.: Die maximale Beendezeit in Komponenten lässt die topologische Sortierung erkennen.
Wie kann man aber diese v_1, v_2, v_3 erkennen?
Fangen wir in H_3 an, dann H_2, dann H_1, dann sind v_3, v_2, v_1 die Wurzeln der Bäume. Aber nicht, wenn die Reihenfolge H_1, H_2, H_3 gewählt wird. Rauslaufen tritt auf.
Deshalb: Eine weitere Tiefensuche mit dem Umkehrgraph:



Hauptschleife von DFS(G) in obiger Reihenfolge:



Dann

DFS-visit(v_1): Genau H_1
v_2 steht vorne auf der Liste.
DFS-visit(v_2): Genau H_2
v_3 vorne auf der Liste
DFS-visit(v_3): Genau H_3.
v_3 vorne auf der Liste

Algorithmus Starke Komponenten

Starke-Komponenten(G)
Eingabe: $G = (V, E)$ gerichteter Graph.

1. DFS(G). Mit Liste nach absteigender Beendezeit $V = (v_1, v_2, \ldots, v_n)$, $f[v_1] > f[v_2] > \cdots > f[v_n]$.
   v_1 ... zuletzt beendet
   v_n ... zuerst beendet

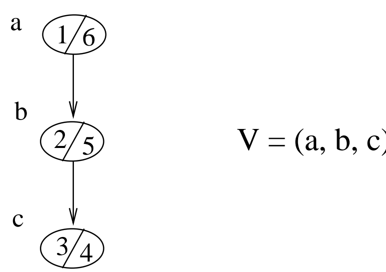
2. Drehe Kanten in G um. Graph G^U.

3. DFS(G^U) mit Hauptschleife nach Liste V. Jedes DFS-visit(v) in der Hauptschleife ergibt eine starke Komponente

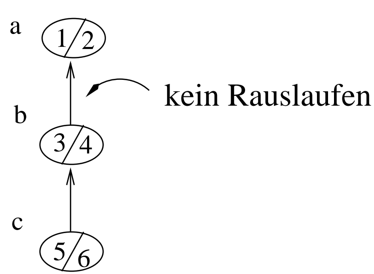
Zeit:$O(|V| + |E|)$! □

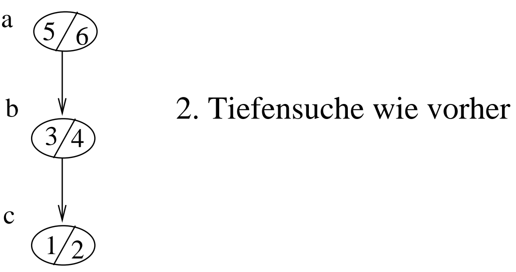
Beispiel 5.6:

1. Tiefensuche

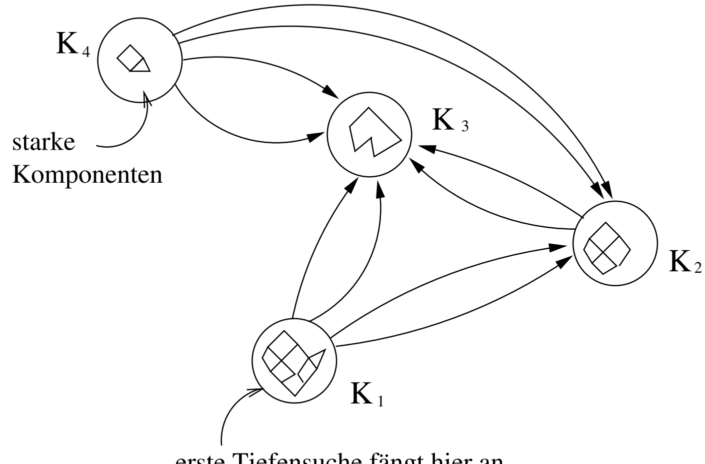


2. Tiefensuche

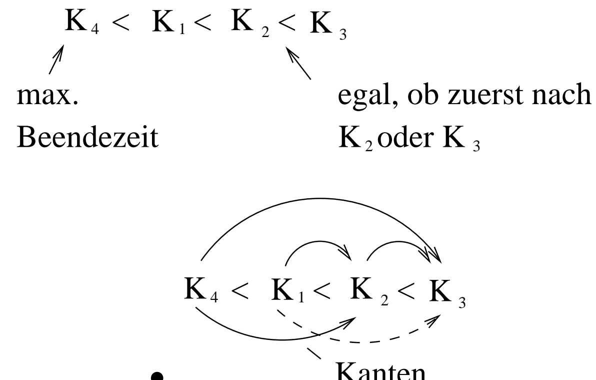




Motivation des Algorithmus

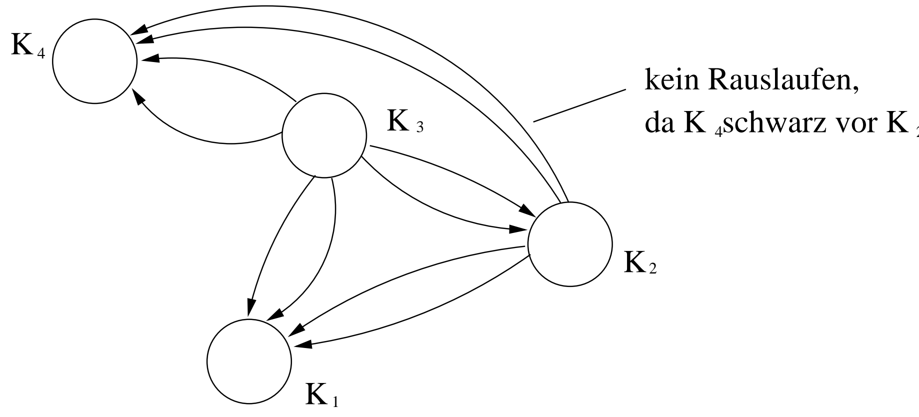


- Eine Tiefensuche entdeckt in jedem Fall die Komponete eines Knotens
- Die Tiefensuche kann aber rauslaufen!
- Die Tiefensuche lässt die topologische Sortierung auf den Komponenten erkennen: nach maximaler Beendezeit in Komponente:

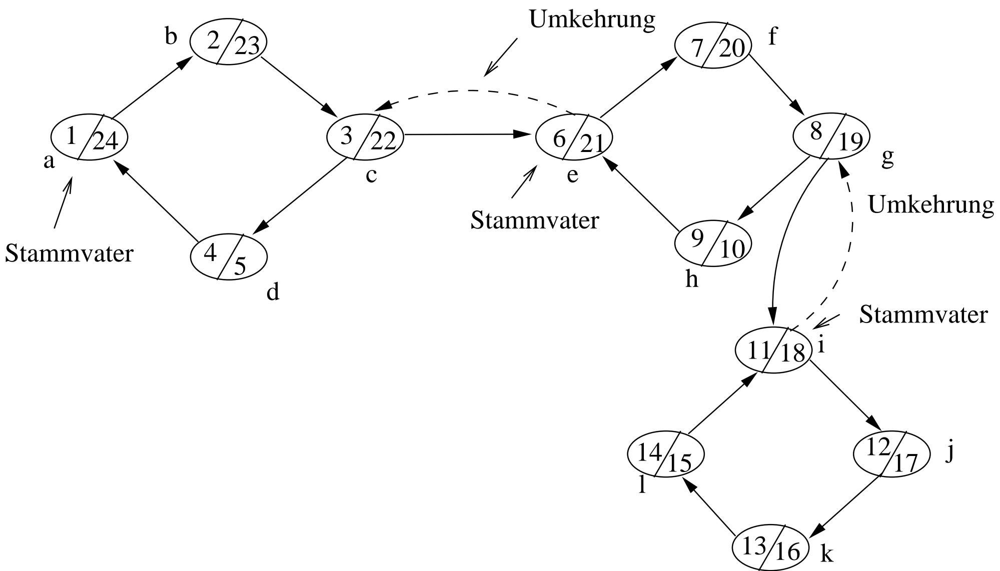


-

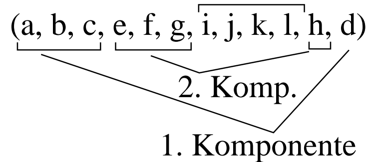
- Bei Beginn in K_4 bekommen wir: $K_1 < K_4 < K_2 < K_3$.
- Wie kann man das Rauslaufen jetzt verhindern?



- Hauptschleife gemäß topologischer Sortierung, d.h. abfallende Beendezeit: K_4, K_1, K_2, K_3



1. Tiefensuche von Starke-Komponenten(g)
Knotenliste:



Stammvater = der Knoten, über den eine starke Komponente das erste Mal betreten wird.
erstes Mal = kleinste Anfangszeit
Einige Überlegungen zur Korrektheit. Welcher Knoten einer Komponente wird zuerst betreten?

Definition 5.7 (*Stammvater*):

Nach dem Lauf von DFS(G) sei für $u \in V$
$\Phi(u) =$ der Knoten, der unter allen von u erreichbaren Knoten die maximale Beendezeit hat.
$\Phi(u)$ Stammvater von u.

Satz 5.8:

(a) Es ist $col[\Phi(u)] = grau$ bei $d[u]$.

(b) In G $\Phi(u) \rightsquigarrow u$, $u \rightsquigarrow \Phi(u)$

Beweis 5.9:

(a) Klar bei $u = \Phi(u)$. Sei also $u \neq \Phi(u)$. Wir schließen die anderen Fälle aus.
1. Fall
$col[\Phi(u)]$ schwarz bei $d[u]$. Dann $f[\Phi(u)] < d[u] < f[u]$ im Widerspruch zur Definition, da ja $u \rightsquigarrow u$.
2. Fall
$col[\Phi(u)]$ weiß bei $d[u]$. Gibt es zu $d[u]$ Weißen Weg von u nach $\Phi(u)$, dann $d[u] < d[\Phi(u)] < f[\Phi(u)] < f[u]$ im Widerspruch zur Definition von $\Phi(u)$.

Gibt es zu $d[u]$ keinen Weißen Weg von u nach $\Phi(u)$ dann (da ja $u \rightsquigarrow \Phi(u)$) sieht es so aus:

$$u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow \ldots \rightarrow u_k \rightarrow \Phi(u)$$

Weißer Weg
col[u_l] = grau bei d[u]
col[u_l] = schwarz geht nicht

Es ist u_l der letzte Knoten auf dem Weg mit $col[u_l] = grau$. Dann aber wegen Weißem Weg $d[u_l] < d[\Phi(u)] < f[\Phi(u)] < f[u_l]$ im Widerspruch zur Definition von $\Phi(u)$, da $u \rightsquigarrow u_l$

(b) Mit (a) ist $d[\Phi(u)] < d[u] < f[u] < f[\Phi(u)]$

Folgerung 5.10:

a) u, v in einer starken Komponente $\iff \Phi(u) = \Phi(v)$
Stammväter $\iff$ starke Komponenten

b) $\Phi(u) =$ der Knoten mit der kleinsten Anfangszeit in der Komponente von u.

Beweis 5.11:

a) Mit letztem Satz und Definition Stammvater:

$u \rightsquigarrow \Phi(u)$
$v \rightsquigarrow \Phi(v)$

Also $\Phi(v) = \Phi(u)$.

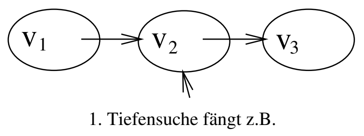
Andererseits

$\Phi(u) = \Phi(v)$ u v

Also u, v in einer Komponente.

1. Sei v mit kleinster Anfangszeit in starker Komponente. Dann $d[v] < d[u] < f[u] < f[v]$ für alle u in der Komponente. Alle von u erreichbaren werden vor $f[u]$ entdeckt, also vor $f[v]$ beendet. Also $v = \Phi(u)$

Beachte noch einmal



$V = (v_1, \ldots, v_2, \ldots, v_3)$ wegen Beendezeiten.
Die Korrektheit des Algorithmus ergibt sich jetzt aus: **Satz 5.12**:

In der Liste $V = (v_1, \ldots, v_n)$ gibt die Position der Stammväter eine topologische Sortierung der starken Komponenten an.

Beweis 5.13:

Sei also die Liste

$V = (v_1, \ldots, v_n)$

v_1 Stammvater v_l Stammvater v_k Stamm– vater

Sei $k < l$, dann gibt es keinen Weg in G. v_l → v_k, denn sonst wäre v_l kein Stammvater, da $f[v_l] > f[v_k]$.

Im Umkehrgraph: Gleiche Komponenten wie vorher. Aber Kanten zwischen Komponenten Komponenten gegen topologische Sortierung. Also gibt die 2. Tiefensuche die Komponenten aus.

6 Tiefensuche in ungerichteten Graphen: Zweifache Zusammenhangskomponenten

Der Algorithmus ist ganz genau derselbe wie im gerichteten Fall!
Abbildung 1 zeigt noch einmal den gerichtete Fall und danach betrachten wir in Abbildung 2 den ungerichteten Fall auf dem analogen Graphen (Kanten " ⟶ " sind Baumkanten, über die entdeckt wird.)
Im ungerichteten Graphen gilt: beim ersten Gang durch eine Kante stößt man

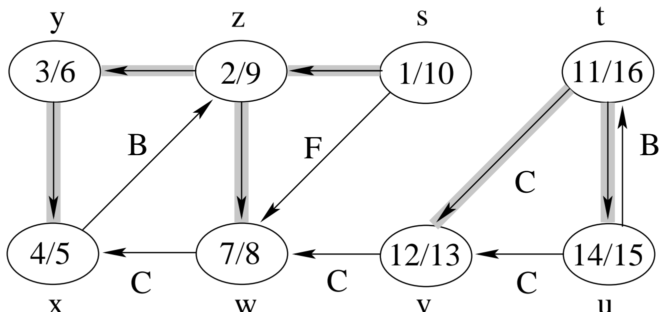


Abb. 1: Das Ergebnis der Tiefensuche auf einem gerichteten Graphen

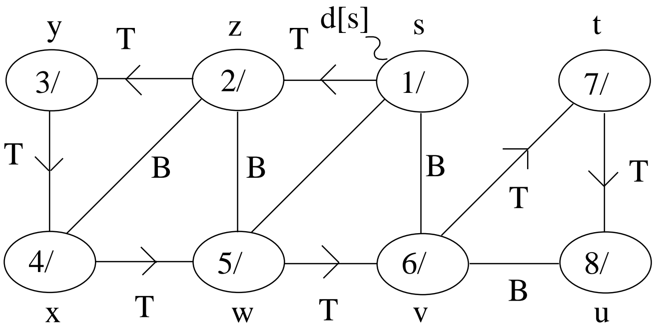
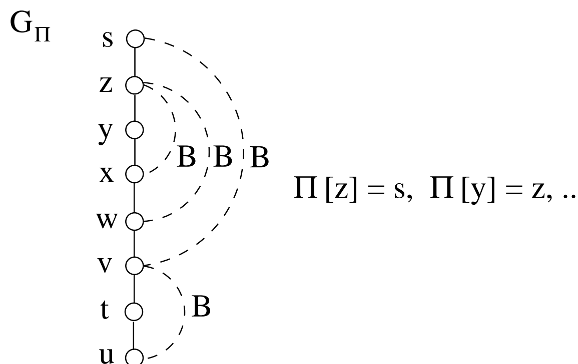


Abb. 2: vorhergender Graph, ungerichtete Variante

- auf einen weißen Knoten (Kante: T)
- auf einen grauen Knoten (Kante: B)
- nicht auf einen schwarzen Knoten (Kante: F oder C)



Im ungerichteten Fall der Tiefensuche ist festzuhalten:

- Jede Kante wird zweimal betrachtet: $\{u, v\}$ bei DFS-visit(u) und bei DFS-visit(v).
- Maßgeblich für den Kantentyp (Baumkante, Rückwärts-, Vorwärts-, Kreuzkante) ist die Betrachtung:

  (i) $\{u, v\}$ Baumkante $\iff$ Beim ersten Betrachten von $\{u, v\}$ findet sich ein weißer Knoten.

(ii) $\{u, v\}$ Rückwärtskante $\Longleftrightarrow$ Beim ersten Gehen von $\{u, v\}$ findet sich ein bereits grauer Knoten

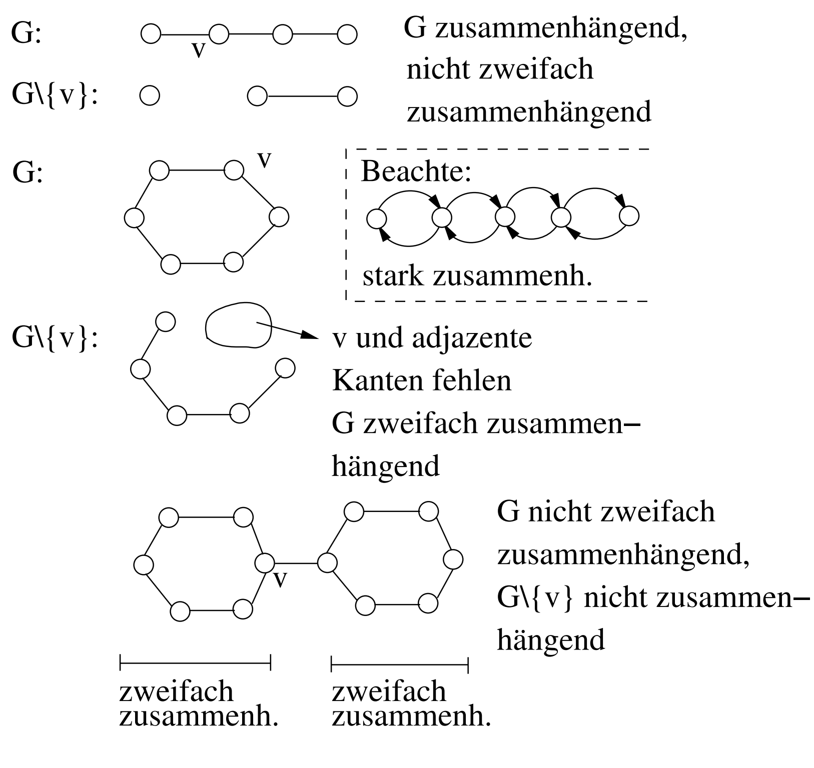
(iii) Beim ersten Gehen kann sich kein schwarzer Knoten finden. Deshalb gibt es weder Kreuz- noch Vorwärtskanten.

Der Weiße-Weg-Satz gilt hier vollkommen analog.
Kreise erkennen ist ganz ebenfalls analog mit Tiefensuche möglich.
Der Begriff der starken Zusammenhangskomponente ist nicht sinnvoll, da Weg (u, v) im ungerichteten Fall ebenso (v, u) ist.

Stattdessen: *zweifach zusammenhängend*. Unterschied zwischen $G_1 =$ und $G_2 =$?
Löschen wir in G_1 einen beliebigen Knoten, hängt der Rest noch zusammen. Für a, b, c im Graphen G_2 gilt dies nicht!
Für den Rest dieses Kapitels gilt nun folgende *Konvention*:
Ab jetzt gehen wir nur immer von zusammenhängenden, gerichteten Graphen aus.

Definition 6.1 (*zweifach zusammenhängend*):

$G \setminus \{v\}$ ist zweifach zusammenhängend $\Leftrightarrow$ G ist zusammenhängend für alle $v \in V$.
$G \setminus \{v\} = (V \setminus \{v\}, E \setminus \{\{u, v\} | u \in V\})$.

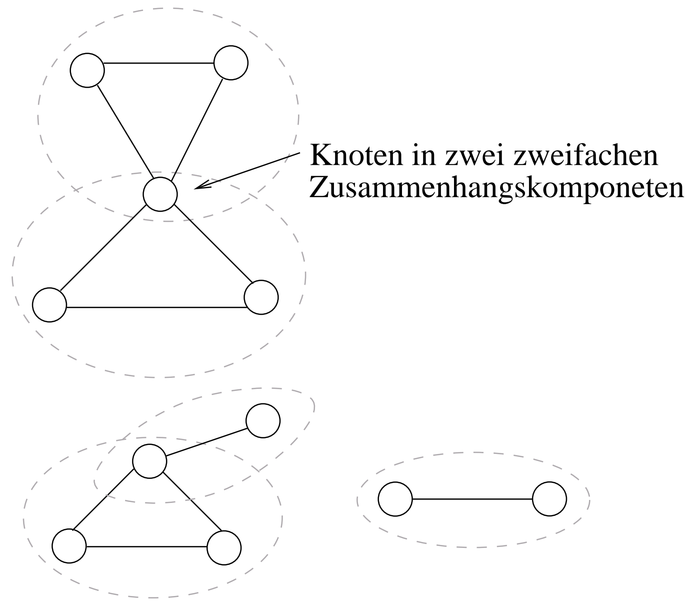


G: (v) G\{v}: also G zweifach zusammenh.

Was ist die Bedeutung zweifach zusammenhängend?
Satz 6.2 (*Menger 1927*):

> G zweifach zusammenh. $\iff$ Für alle $u, v \in V, u \neq v$ gibt es zwei disjunkte Wege zwischen u und v in G.
> Das heißt, Wege $(u, u_1, u2, \ldots, u_m, v)$, $(u, u'_1, u'_2, \ldots, u'_m, v)$ mit $\{u_1, \ldots, u_m\} \cap \{u'_1, \ldots, u'_m\} = \emptyset$ $\square$

Beweis erfolgt auf überraschende Weise später.
Ein induktiver Beweis im Buch Graphentheorie von Reinhard Diestel. Die Richtung „$\Leftarrow$“ ist einfach. Beachte noch, ist u ○——○ v, so tut es der Weg (u, v), mit leerer Menge von Zwischenknoten (also nur 1 Weg).
Nun gilt es wiederm nicht zweifach zusammenhängende Graphen in seine zweifach zusammenhängenden Bestandteile zu zerlegen, in die zweifachen (Zusammenhangs-)Komponenten.
Beispiel 6.3:



Formal
Definition 6.4 (*zweifache Komponenten*):

> Ein Teilgraph $H = (W, F)$ von G ist eine zweifache Komponente $\iff$ H ist ein maximaler zweifach zusammenhängender Teilgraph von G (maximal bezüglich Knoten und Kanten). □

(a) Jede Kante ist in genau einer zweifachen Komponente.

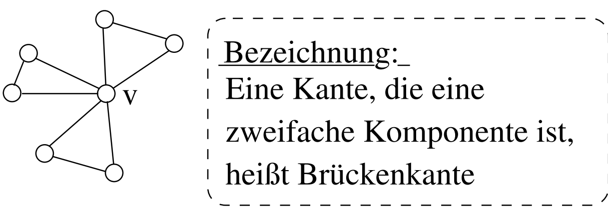
(b) Sind $H_1 = (W_1, F_1), \ldots, H_k = (W_k, F_k)$ die zweifachen Komponeten von $G = (V, E)$, so ist $F_1, \ldots, F_k$ eine Partition (Einteilung) von E.

$F_i \cap F_j = \emptyset$ für $i \neq j, F_1 \cup \cdots \cup F_k = E$.
Beweis 6.5:

(a) Sei also $H_1 \neq H_2$ zweifach zusammenhängend, dann ist $H = H_1 \cup H_2$ zweifach zusammenhängend:
Für w aus $H_1, w \neq u, w \neq v$ ist $H \setminus \{w\}$ zusammenhängend (wegen Maximalität). Ebenso w aus H_2. Für $w = u$ ist, da immer noch v da ist, $H \setminus \{w\}$ zweifach zusammenhängend. Also wegen Maximalität zweifache Komponenten $\supseteq H_1 \cup H_2$.

(b) $F_i \cap F_j = \emptyset$ wegen (a). Da Kante (u, v) zweifach zusammenhängend ist, ist nach Definition jede Kante von E in einem F_1.

Bei Knoten gilt (a) oben nicht:



v gehört zu 3 zweifachen Komponenten. v ist ein typischer Artikulationspunkt. **Definition 6.6** (*Artikulationspunkt*):

> v ist Artikulationspunkt von $G \iff G \setminus \{v\}$ nicht zusammenhängend.

Bemerkung 6.7 (v):

> st Artikulationspunkt $\iff$ v gehört zu ≥ 2 zweifachen Komponenten

Beweis 6.8 („$\Rightarrow$“):

st v Artikulationspunkt. Dann gibt es Knoten $u, w, u \neq v, w \neq v$, so dass jeder Weg von u nach w von der Art $(u, \ldots, v, \ldots, w)$ ist. (Sonst $G \setminus \{v\}$ zusammenhängend) Also u, w nicht in einer zweifachen Komponente. Dann ist jeder Weg von der Art $(u, \ldots, u_1, v, u_2, \ldots, 2)$, so dass u_1, u_2 nicht in einer zweifachen Komponente sind. Sonst ist $G \setminus \{v\}$ Weg $(u, \ldots, u_1, u_2, \ldots w)$ (ohne v), Widerspruch. Also haben wir

u1 — v — u2

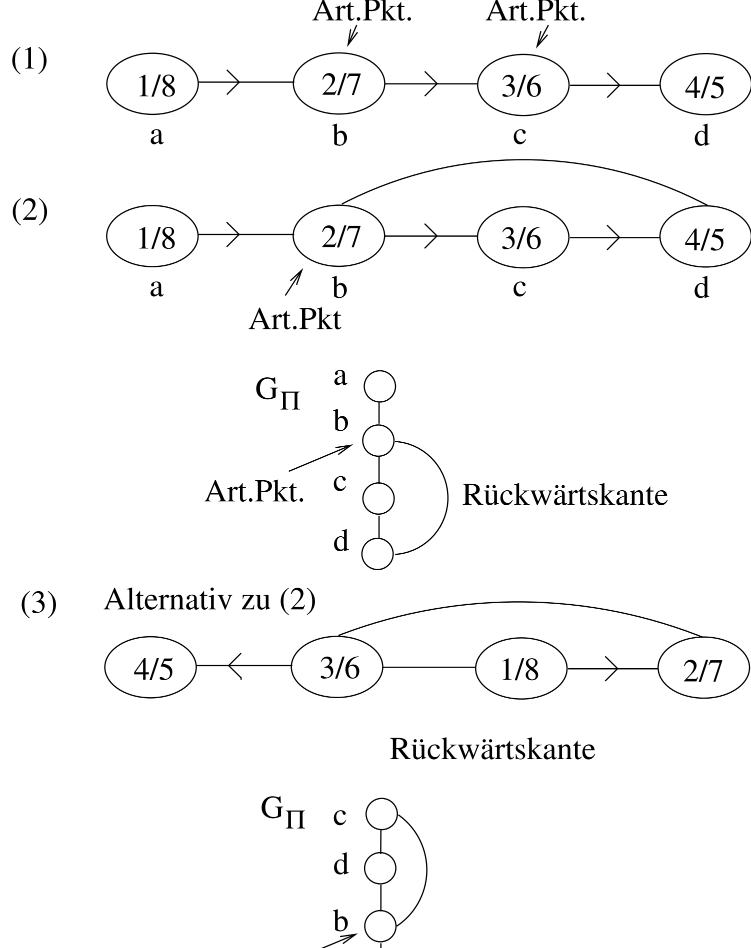
verschiedene Komponenten.

Die Kante $\{u_1, v\}$ gehört zu einer Komponente (eventuell ist sie selber eine), ebenso $\{v, u_2\}$. Die Komponenten der Kanten sind verschieden, da u_1, u_2 in verschiedenen Komponenten liegen. Alos v liegt in den beiden Komponenten. „$\Leftarrow$“ Gehört also v zu > 2 Komponenten. Dann

u1, u2, v, u3 ..., uk

und ≥ 2 Kanten von $\{u_i, v\}$ in verschiedenen Komponenten. Etwa u_1, u_2. Aber u_1 und u_2 in 2 verschiedenen Komponenten. Also gibt es w, so dass in $G \setminus \{w\}$ kein Weg u1 — u2 existiert. Denn wegen u1 — v — u2 muss $w = v$ sein und v Artikulationspunkt.

Graphentheoretische Beweise sind nicht ganz so leicht!
Artikulationspunkte und Tiefensuche?



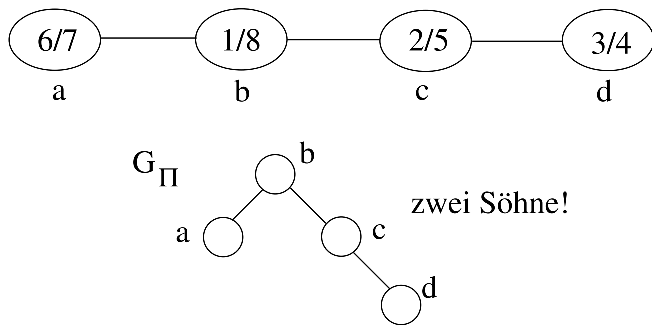
Seite 6.18 Was haben die Artikulationspunkte in allen genannten Fällen gemeinsam? Der Artikulationspunkt (sofern nicht Wurzel von G_Π) hat einen Sohn in G_Π, so dass es von dem Sohn oder Nachfolger keine Rückwärtskante zum echten Vorgänger gibt!

(1) Gilt bei b, c. d ist kein Artikulationspunkt und das Kriterium gilt auch nicht.

(2) b ist Artikulationspunkt, Sohn c erfüllt das Kriterium. c, d erfüllen es nicht, sind auch keine Artikulationspunkte.

(3) Hier zeigt der Sohn a von b an, dass b ein Artikulationspunkt ist. Also keineswegs immer derselbe Sohn!

Was ist, wenn der Artikulationspunkt Wurzel von G_Π ist?

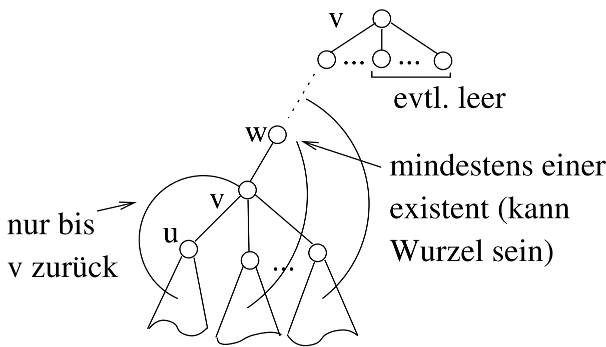


Satz 6.9 (*Artikulationspunkte erkennen*):

Sei v ein Knoten von G und sei eine Tiefensuche gelaufen. (Erinnerung: G immer zusammenhängend).

(a) Ist v Wurzel von G_Π. v ist Arikulationspunkt $\iff G_\Pi$ v ... $\leq$ 2Söhne.

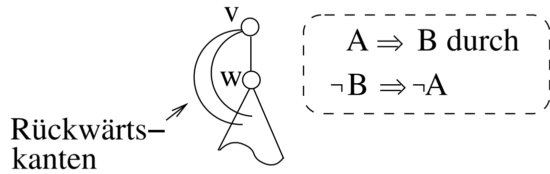
(b) Ist v nicht Wurzel von G_Π.
v ist Artikulationspunkt
$\iff$



Das heißt, v hat einen Sohn (hier u), so dass von dem und allen Nachfolgern in G_Pi keine (Rückwärts-)Kanten zu echtem Vorgänger von v.

Beweis von Satz 7.2:

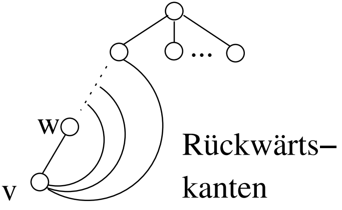
a „$\Leftarrow$“ $G \setminus \{v\}$ nicht zusammenhängend, damit v Artikulationspunkt. „$\Rightarrow$ “ Hat v nur einen Sohn, dann



dann $G \setminus \{v\}$ zusammenhängend. (Können in $G \setminus \{v\}$ über w statt v gehen.) Also ist v kein Artikulationspunkte. Ist v ohne Sohn, dann ist er kein Artikulationspunkt.

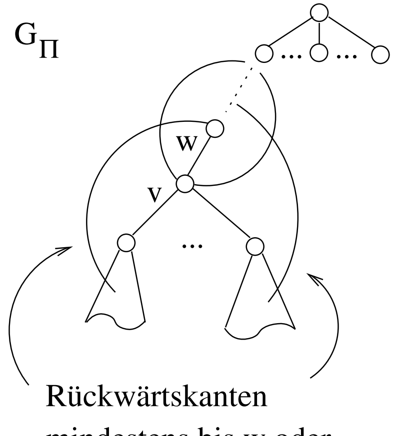
(b) „$\iff$ “ In $G\{v\}$ kein Weg u ○——○ w , also ist v Artikulationspunkt.
„$\Rightarrow$ “ Gelte die Behauptung nicht, also v hat keinen Sohn wie u in G_Π.

a. Fall: v hat keinen Sohn. Dann in G_Π



In $G \setminus \{v\}$ fehlen die Rückwärtskanten und $\{w, v\}$. Also bleibt der Rest zusammenhängend. v ist kein Artikulationspunkt.

2. Fall: v hat Söhne, aber keinen wie u in der Behauptung
Dann:



In $G \setminus \{v\}$ bleiben die eingetragenen Rückwärtskanten, die nicht mit v inzident sind, stehen. Also ist $G \setminus \{v\}$ zusammenhängend. Also ist v kein Artikulationspunkt

$A \Rightarrow B$ durch $\neg B \Rightarrow \neg A$.

□

Wie kann man Artikulationspunkte berechnen?
Wir müssen für alle Söhne in G_Π wissen, wie weit es von dort aus zurück geht.
Definition 6.10:

Sei DFS(G) gelaufen, also G_Π vorliegend.

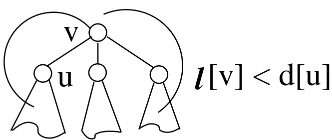
a Für Knoten v ist die Menge von Knoten $L(v)$ gegeben durch $w \in L(u) \iff w = v$ oder w Vorgänger von v und es gibt eine Rückwärtskante von v oder dem Nachfolger zu w.

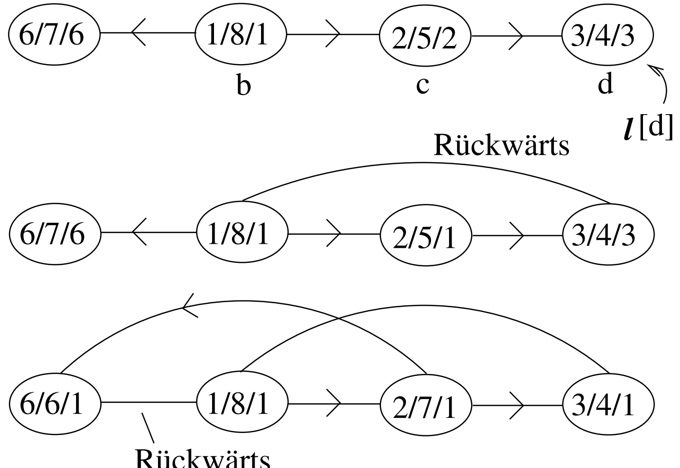
b $l[v] = Min\{d[w] | w \in L(v)\}$ ist der Low-Wert von v. ($l[v]$ hängt von Lauf von DFS(G) ab.)

Folgerung 6.11:

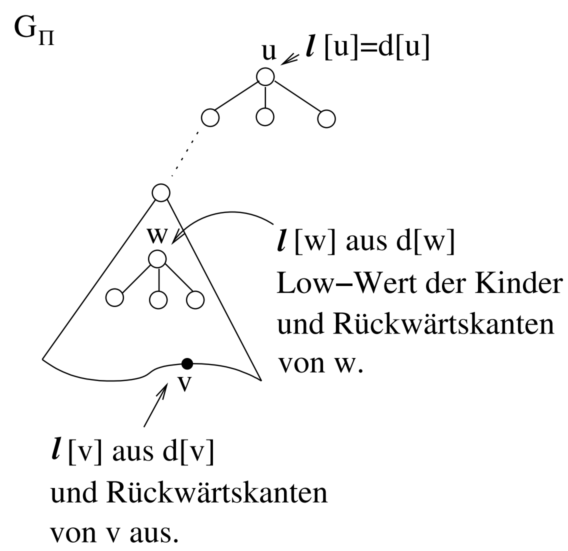
Sei DFS(G) gelaufen und v nicht Wurzel von G_Π.
v ist Artikulationspunkt $\iff v$ hat Sohn u in G_Π mit $l[u] \leq d[v]$.

Beachte: $l[v] = d[v] \Rightarrow v$ Artikulationspunkt. Keine Äquivalenz.





Berechnung von l[v]



Korrektheit: Induktion über die Tiefe des Teilbaumes

Algorithmus (l-Werte)

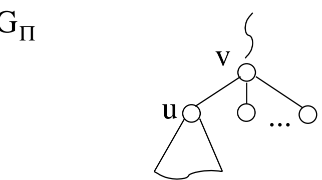
Modifikation von DFS-visit(u):

```
MDFS-visit(u)                      //hier col[u] = weiß
⋮
l[u]=d[u]
⋮
for each v ∈ Adj[u] do{
   if (col[v] == weiß){
      Π[v] = u; MDFS-visit(v);
      l[u] = MIN{l[u], l[v]}
   }
   if (col[v] == grau) and (Π[u] ≠ v){
      l[u] = MIN{l[u], d[v]}       //d[v]! nicht l[v], keine Iteration.
```

    }
}

Damit können wir Artikulationspunkte in Linearzeit erkennen. Es bleiben die zweifachen Komponenten zu finden. Seien die l-Werte gegeben. Nun 2. Tiefensuche folgendermaßen:

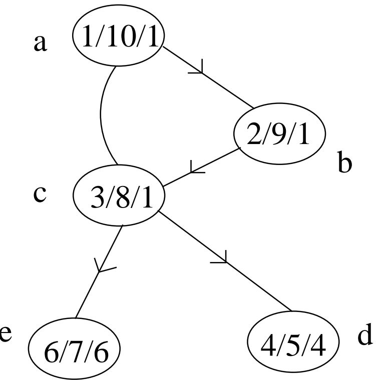
- Knoten, der entdeckt wird, auf einem Keller speichern.



  Sei jetzt $l[u] \leq d[v]$.

- Am Ende von DFS-visit(u) Keller bis v(!) ausgeben. Ist zweifache Komponente. v wieder oben drauf schreiben.

Beispiel 6.12:



Kellerinhalt:

			↓
		↓	a
	↓	b	a
↓	c	b	a
d	c	b	a

$l[d] \geq d[c]$ Ausgabe d, c

	c	b	a
e	c	b	a

Ausgabe e, c

c b a

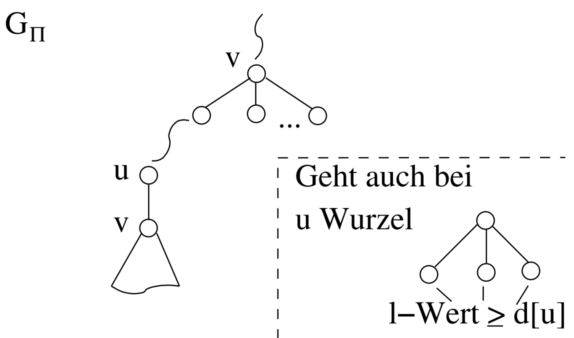
Ausgabe c, b, a am Ende.

Algorithmus (Zweifache Komponenten)

```
1. DFS(G)                          //modifiziert für l-Werte
2. DFS(G)                          //in derselben(!) Reihenfolge wie 1. mit
   NDFS-visit(u)                   //col[u] = w
   ⋮
   for each v ∈ Adj[v]{
     if (col[v] == w){
       Π[v] = u, v auf Keller;
       NDFS-visit(v);
       if (l[v] ≥ d[v]){
         Ausgabe bis v inklusive,
         u noch hinzu ausgeben
       }
     }
   }
```

Beweis 6.13:

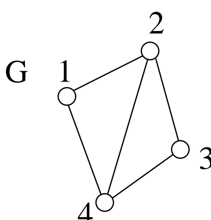
Korrektheit induktiv über $s = \#$ Zweifache Komponenten von G.
Induktionsanfang: $s = 1$
Ausgabe nur am Ende, da kein Artikulationspunkt. Also korrekt. Induktionsschluss Gelte Behauptung für alle(!) Graphen mit $s \geq 1$ zweifachen Komponenten. Zeige dies für G mit $s + 1$ Komponenten.
Wir betrachten die 2. Tiefensuche des Algorithmus. Die l-Werte stimmen also bereits, wegen der Koorektheit der 1. Tiefensuche.
Wir betrachten den Baum G_Π, welcher der gleiche wie bei der 1. Tiefensuche und 2. Tiefensuche ist.



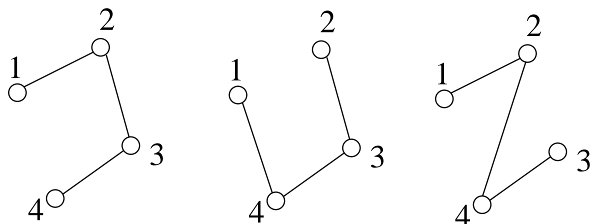
Sei NDFS-visit(v) der erste Aufruf, nach dm die Ausgabe erfolgt. Dann ist $l[v] \geq [u]$.
Ist l[v] = d[v], dann ist v Blatt (sonst vorher Ausgabe) und v, u wird ausgegeben und u kommt auf die Kellerspitze.
Ist l[v] = d[u], dann Ausgabe des Kellers bis u und u kommt wieder auf die Kellerspitze.
Damit wird eine zweifache Komponente ausgegeben. (Mehr kann nicht dazu gehören, weniger nicht, da bisher kein Artikulationspunkt). Nach Ausgabe der Komponente liegt eine Situation vor, die in DFS-visit(u) auftritt, wenn wir den Graphen betrachten, in dem $\{u, v\}$ und die anderen ausgegegenen Knoten gelöscht sind.
Auf diesem ist die Induktionsvoraussetzung anwendbar und der Rest wird richtig ausgegeben.

7 Minimaler Spannbaum und Datenstrukturen

Hier betrachten wir als Ausgangspunkt stets zusammenhängende, ungerichtete Graphen.



So sind



Spannbäume (aufspannende Bäume) von G. Beachte immer 3 Kanten bei 4 Knoten.
Definition 7.1 (*Spannbaum*):

> Ist $G = (V, E)$ zusammenhängend, so ist ein Teilgraph $H = (V, F)$ ein Spannbaum von G, genau dann, wenn H zusammenhängend ist und für alle $f \in FH \setminus \{f\} = (V, F \setminus \{f\})$ nicht mehr zusammenhängend ist (H ist maximal zusammenhängend).

Folgerung 7.2:

> Sei H zusammenhängend, Teilgraph von G.
> H Spannbaum von $G \iff H$ ohne Kreis.

Beweis 7.3:

> „⇒" Hat H einen Kreis, dann gibt es eine Kante f, so dass $H \setminus \{f\}$ zusammenhängend ist. ($A \Rightarrow B$ durch $\neg B \wedge \neg A$)
> „⇐ " Ist H kein Spannbaum, dann gibt es eine Kante f, so dass $H \setminus \{f\}$ zusammenhängend ist. Dann haben wir einen Kreis in H, der f enthält. ($B \Rightarrow A$ durch $\neg A \wedge \neg B$)

Damit hat jeder Spannbaum genau $n - 1$ Kanten, wenn $|V| = n$ ist. Vergleiche Seite 32. Die Baumkanten einer Suche ergeben einen solchen Spannbaum. Wir suchen Spannbäume minimaler Kosten. **Definition 7.4**:

> Ist $G = (V, E)$ und $K : E \to \mathbb{R}$ (eine Kostenfunktion) gegeben. Dann ist $H = (V, F)$ ein *minimaler Spannbaum* genau dann, wenn:
>
> - H ist Spannbaum von G
> - $K(H) = \sum_{f \in F} K(f)$ ist minimal unter den Knoten aller Spannbäume.
>   ($K(H) =$ Kosten von H).

Wie finde ich einen minimalen Spannbaum? Systematisches Durchprobieren. Verbesserung durch branch-and-bound.
Eingabe $G = (V, E), K :\to \mathbb{R}, E = \{e_1, \ldots, e_n\}$ (also Kanten)
Systematisches Durchgehen aller Mengen von $n - 1$ Kanten (alle Bitvektoren $b_1, \ldots, b_n$ mit genau $n - 1$ Einsen), z.B, rekursiv.
Testen, ob kein Kreis. Kosten ermitteln. Den mit den kleinsten Kosten ausgeben.
Zeit: Mindestens $\binom{m}{n-1} = \frac{m!}{(n-1)! \cdot \underbrace{(m-1+1)!}_{\geq 0 \text{ für } n-1 \leq m}}$
Wieviel ist das? $m = 2 \cdot n$, dann

$$\frac{(2n)!}{(n-1)!(n+1)!} = \frac{2n(2n-1)\cdot(2n-2)\cdot\ldots\cdot n}{(n+1)n(n-1)\ldots 1}$$

$$\geq \underbrace{\frac{2n-2}{n-1}}_{=2}\cdot\underbrace{\frac{2n-3}{n-1}}_{>2}\cdot\ldots\cdot\underbrace{\frac{n}{1}}_{>2}$$

$$\geq \underbrace{2\ \ 2\ldots 2}_{n-1-\text{mal}} = 2^{n-1}$$

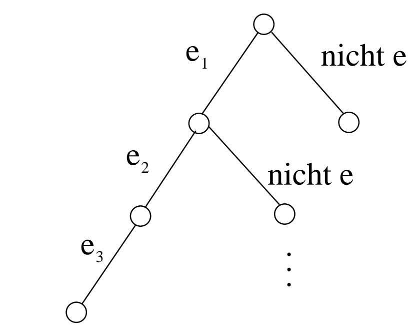
Also Zeit $\Omega(2^n)$.

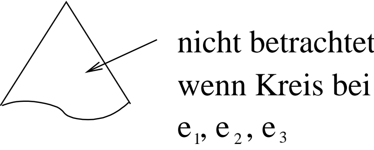
Regel: $c \geq 0$ $\frac{a}{b} \leq \frac{a-c}{b-c}$ $\Leftrightarrow b \leq a$. Das c im Nenner hat mehr Gewicht.

Definition 7.5 (Ω-*Notation*):

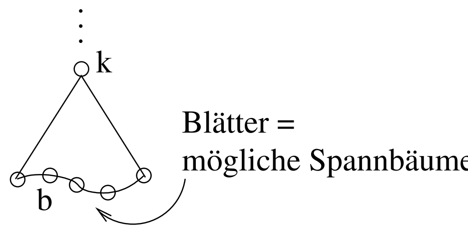
> Ist $f, g : \mathbb{N} \to \mathbb{R}$, so ist $f(u) = \Omega(g(u))$ genau dann, wenn es eine Konstante $C > 0$ gibt, mit $|f(n)| \geq C - g(n)$ für alle hinreichend großen n. (vergleiche O-Notation)

O-Notation: Obere Schranke
Ω-Notation: Untere Schranke.
Verbesserung: Backtracking, frühzeitiges Erkennen von Kreisen: Aufbau eines Baumes (Prozeduraufrufbaum):





Alle Teilmengen mit e_1, e_2, e_3 sind in einem Schritt erledigt, wenn ein Kreis vorliegt.
Einbau einer Kostenschranke in den Backtracking-Algorithmus: *Branch-and-bound.*
Kosten des Knotens $k = \sum_{\substack{f \text{ in k} \\ \text{gewählt}}} K(f)$.



Es ist für Blätter wie b unter k.
Kosten* von $k \leq$ Kosten von b.
*Untere Schranke an die Spannbäume unter k.
Also weiteres Rausschneiden möglich, wenn Kosten von $k \geq$ Kosten eines bereits gefundenen Baumes. Im allgemeinen ohne weiteres keine bessere Laufzeit nachweisbar. Besser: Irrtumsfreier Aufbau eines minimalen Spannbaumes.

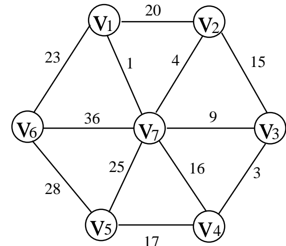


Abb. 3: ungerichteter Graph mit Kosten an den Kanten

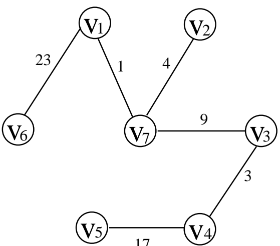
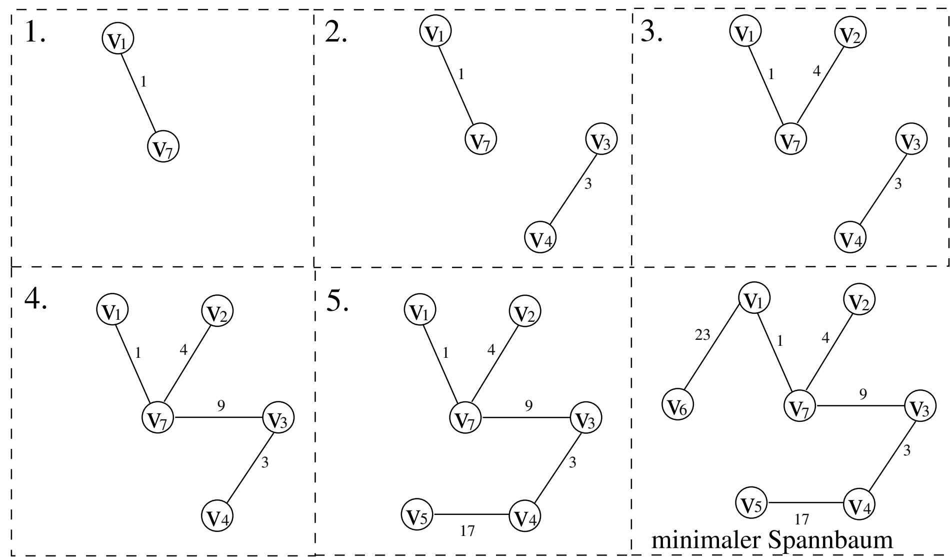


Abb. 4: Spannbaum mit minimalen Kosten

Aufbau eines minimalen Spannbaums



Nimm die günstigste Kante, die keinen Kreis ergibt, also nicht $v_2 — v_3$, $v_4 — v_7$

Abb. 5: schrittweiser Aufbau eines Spannbaumes

Implementierung durch Partition von V

$\{v_1\}, \{v_2\}, \ldots, \{v_7\}$	keine Kante
$\{v_1, v_7\}, v_2, v_3, v_4, v_5$	$\{v_1, v_7\}$
$\{v_1, v_7\}, \{v_3, v_4\}, \ldots$	$\{v_1, v_7\}, \{v_3, v_4\}$
$\{v_1, v_7, v_2\}, \{v_3, v_4\}, \ldots$	$\{v_1, v_7\}, \{v_3, v_4\}, \{v_7, v_2\}$
$\{v_1, v_7, v_2, v_3, v_4\}, \ldots$	$\{v_1, v_7\}, \{v_3, v_4\}, \{v_7, v_2\}, \{v_7, v_3\}$
v_5 hinzu	$\vdots$
v_6 hinzu	

Problem: Wie Partition darstellen?

Algorithmus Minimaler Spannbaum

(Kruskal 1956)
Eingabe:

$G = (V, E)$, zusammenhängend, $V = \{1, \ldots, n\}$, Kostenfunktion $K : E \to \mathbb{R}$

Ausgabe:

$F =$ Mege von Kanten eines minimalen Spannbaumes.

```
1. F = ∅, P = {{1}, ..., {n}} // P ist die Partition
2. E nach Kosten sortieren     // geht in O(E log |E|)=O(|E| log|V|),
                               // E≤V², log|E|=Olog|V|
3. while(|P|≥1){               // solange P≠{{1, ..., n}}
```

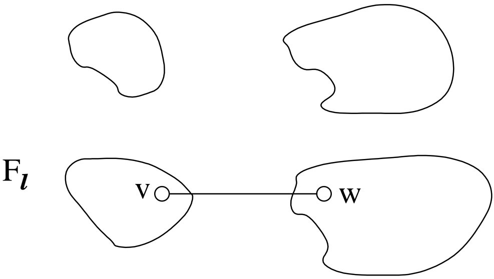
```
4.    {v, w} = kleinstes (erstes) Element von E
      {v, w} aus E löschen
5.    Testen, ob F ∪ {v, w} einen Kreis hat
6.    if({v, w} induziert keinen Kreis){
         W_v = die Menge mit v aus P;
         W_w = die Menge mit w aus P;
         W_v und W_w in P vereinigen
         F = F ∪ {{v, w}}
      }
   }
```

□

Der Algorithmus arbeitet nach dem Greedy-Prinzip (*greedy* = *gierig*):
Es wird zu jedem Zeitpunkt die günstigste Wahl getroffen (= Kante mit den kleinsten Kosten, die möglich ist) und diese genommen. Eine einmal getroffene Wahl bleibt bestehen. Die lokal günstige Wahl führt zum globalen Optimum.
Zur Korrektheit des Algorithmus:
Sei F_l = der Inhalt von F nach dem l-ten Lauf der Schleife.
P_l = der Inhalt von P nach dem l-ten Lauf der Schleife.
Wir verwenden die Invariante:
F_l lässt sich zu einem minimalen Spannbaum fortsetzen. (D.h., es gibt $F \supseteq F_l$, so dass F ein minimaler Spannbaum ist.)
P_l stellt die Zusammenhangskomponenten von F_l dar.
Invariante gilt für $l = 0$.
Gelte sie für l und finde ein $l + 1$-ter Lauf statt.

1. Fall:
   Die Kante $\{v, w\}$ wird nicht genommen. Alles bleibt unverändert.
   Invariante gilt für $l + 1$.
2. Fall:
   $\{v, w\}$ wird genommen. $\{v, w\}$ = Kante von minimalen Kosten, die zwei Zusammenhangskomponenten von F_l verbindet.

Also liegt folgende Situation vor:



Zu zeigen: Es gibt einen minimalen Spannbaum, der $F_{l+1} = F_l \cup \{v, w\}$ enthält. Nach Invariante für F_l gibt es mindestens Spannbaum $F \supseteq F_l$. Halten wir ein solches F fest. Dieses F kann,

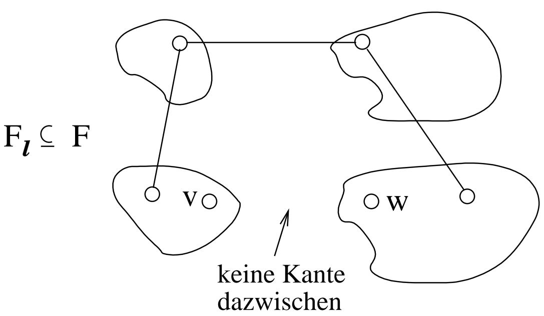
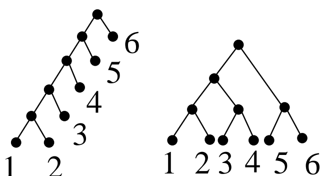


Abb. 6: ein Beispiel für F

muss aber nicht, $\{v,w\}$ enthalten. Falls $F\{v,w\}$ enthält, gilt die Invariante für $l+1$ (sofern die Operation auf P richtig implementiert ist). Falls aber $F\{v,w\}$ nicht enthält, argumentieren wir so:
$F \cup \{v,w\}$ enthält einen Kreis (sonst F nicht zusammenhängend). Dieser Kreis muss mindestens eine weitere Kante haben, die zwei verschiedene Komponenten von F_l verbindet, also nicht zu F_l gehört. Diese Kante ghört z der Liste von Kanten E_l, hat also Kosten, die höchstens größer als die von $\{v,w\}$ sind.
Tauschen wir in F diese Kante und {v, w} aus, haben wir einen minimalen Spannbaum mit $\{v,w\}$. Also gilt die Invariante für $F_{l+1} = F_l \cup \{v,w\}$ und P_{l+1} (sofern Operationen richtig implementiert sind.)
Quintessenz: Am Ende ist oder hat F_l nur noch eine Komponente, da $|P_l| = 1$. Also minimaler Spannbaum wegen Invariante.
Termination: Entweder wird die Liste E_l kleiner oder P_l wird kleiner. Laufzeit:

1. $O(n)$
2. Kanten sortieren:
   $O(|E| \cdot log|E|)$ (wird später behandelt), also $O(|E| \cdot log|V|)$!
3. die Schleife

- $n-1 \leq \#$ Läufe $\leq |E|$
  Müssen $\{\{1\},\ldots,\{n\}\}$ zu $\{\{1,\ldots,n\}\}$ machen. Jedesmal eine Menge weniger, da zwei vereinigt werden. Also $n-1$ Vereinigungen, egal wie vereinigt wird

  Binärer Baum mit n Blättern hat immer genau $n-1$ Nicht-Blätter.

- Für $\{v,w\}$ suchen, ob es Kreis in F induziert, d.h. ob v,w innerhalb einer Menge von P oder nicht.

$$\text{Bis zu } |E|\text{-mal}$$

- W_v und W_w in P vereinigen

$$\text{Genau } n-1\text{-mal.}$$

Zeit hängt an der Darstellung von P.
Einfache Darstellung: $arrayP[1,\ldots,n]$, wobei eine Partition von $\{1,\ldots,n\}$ der Indexmenge dargestellt wird durch:
u, v in gleicher Menge von $P \longleftrightarrow P[u] = P[v]$
(u, v ... Elemente der Grundmenge = Indizes)
Seite 7.22 Die Kante $\{u, v\}$ induziert Kreis $\longleftrightarrow u, v$ in derselben Zusammenhangskomponente von $F \longleftrightarrow P[u] = P[v]$
W_u und W_v vereinigen:

```
1. a = P[v]; b = P[w]          // a+b, wenn W_u ≠ W_v
2. for i = 1, ..., n{
     if (P[i] = a){
       P[i] = b
     }
   }
```

Damit Laufzeit der Schleife:
7. insgesamt $n \cdot O(n) = O(n^2)$ (Es wird $n-1$-mal vereinigt.)
4. und 5. insgesamt $O(|E|)$
3. $O(n)$ insgesamt, wenn $|P|$ mitgeführt.
Also $O(n^2)$, wenn E bereits sortiert ist.
Darstellung von P durh eine Union-Find-Struktur:

- Darstellung einer Partition P einer Grundmenge (hier klein, d.h. als Indexmenge verwendbar)
- Für $U, W \in P$

$$\text{Union}(U, W)$$

  soll $U, W \in P$ durch $U \cup W \in P$ ersetzen

- Für u aus der Grundmenge

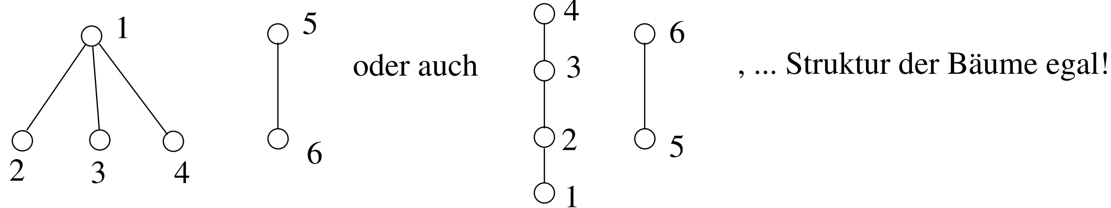
$$\text{Find(u)}$$

  soll Namen der Menge $U \in P$ mit $u \in U$ geben. (Also es geht nicht ums Finden von u!)

Für Kruskal $\leq 2 \cdot |E|$-mal Find(u) + $|V| - 1$-mal Union(u,w)
$\longrightarrow$ Zeit $\Omega(|E| + |V|)$, sogar falls $|E|$ bereits sortiert ist.

Datenstruktur Union-Find-Struktur

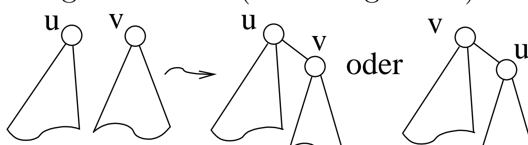
(a) Jede Menge von P als ein Baum mit Wurzel.
$P = \{\{1,2,3,4\},\{5,6\}\}$, dann etwa Wurzel = Name der Menge.



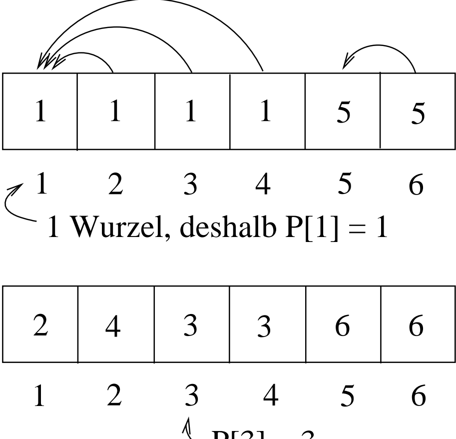
(b)

Find(u)
1. u in den Bäumen finden // Kein Problem, solange Grundmenge
//Indexmenge sein kann
2. Gehe von u aus hoch bis zur Wurzel
3. (Namen der) Wurzel ausgeben.

Union (u,v) // u, v sind Namen von Mengen, also Wurzeln
1. u, v in den Bäumen finden
2. Hänge u unter v (oder umgekehrt)



(c) Bäume in array(Vaterarray): $P[1,\ldots,n]$ of $1\ldots n$
$P[v]$ = Vater von v.
Aus (a) weiter:



Wieder wichtig: Indices = Elemente.
Vaterarray für beliebige Bäume

```
Union (u,v)                          //u, v Wurzeln
  P[u]=v;                            // u hängt unter v
Find(v)
  while (P[v]≠v){                    // O(n) im worst case
    v=P[v];
  }
  Ausgabe von v.
```

1.

(1) (2) (3) (4) ... (N)

P = | 1 | 2 | 3 | ... | N |
1 2 3 ... N

↷ Union(1,2) in $O(1)$

2.

(2) (3) (4) ... (N)
↑
(1)

P = | 2 | 2 | 3 | ... | N |
1 2 3 ... N

↷ Union(2,3) in $O(1)$

3.

(3) 4 ... (N)
↑
(2)
↑
(1)

P = | 2 | 3 | 3 | ... | N |
1 2 3 ... N

↷ Union(3,4) in $O(1)$

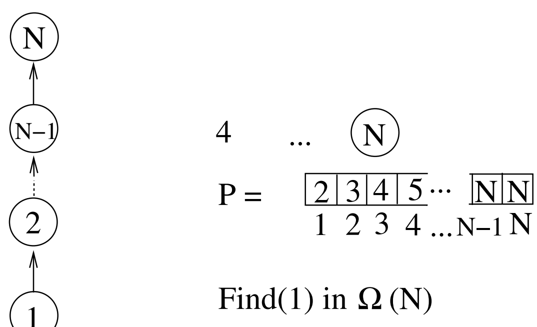
4.

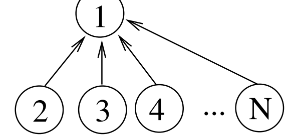
(4)
↑
(3)
↑
(2)
↑
(1)

das Entstehen
langer, dünner Bäume

⋮

N.





Union(2,1); Union(3,1); Union(1,1); ..., Union (N,1)

Vertauschen der Argumente

Find(2) in O(1).

⇒ Union by Size; kleinere Anzahl nach unten.
⇒ maximale Tiefe (längster Weg von Wurzel zu Blatt) = $log_2 N$

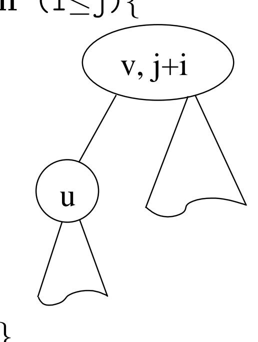
Beachte:

$log_2 N$ zu $N = 2^{log_2 N}$ wie N zu 2^N.

Algorithmus Union-by-Size

```
Union((u,i), (v,j))            // i, j # Elemente in den Mengen.
                               // Muss mitgeführt werden
```

if (i≤j){

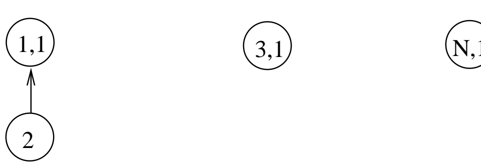


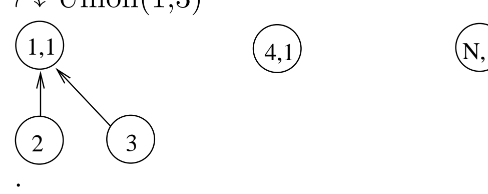
}
else {„umgekehrt"}

Union by Size:

(1,1) (2,1) (3,1) (N,1)

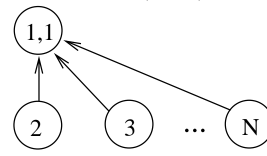
↷ Union(2,1)



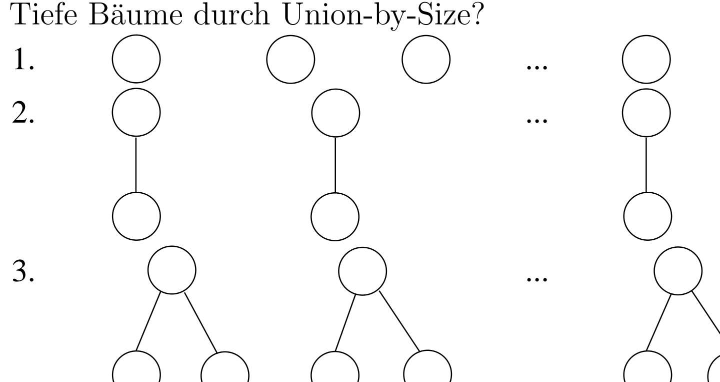


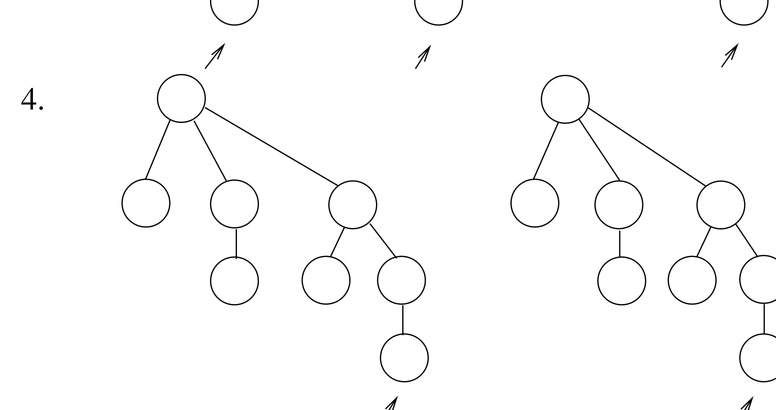
⋮

↷ Union(1,N)



Tiefe Bäume durch Union-by-Size?





5.

Vermutung: N Elemente $\Rightarrow$ Tiefe $\leq log_2 N$.

Satz 7.6:

Beginnend mit $P = \{\{1\}, \{2\}, \ldots, \{n\}\}$ gilt, dass mit Union-by-Size für jeden entstehenden Baum T gilt: Tiefe(T) $\leq log_2|T|$
($|T|$ = # Elemente von T = # Knoten von T)

Beweis 7.7:

Induktion über die # der ausgeführten Operationen Union(u,v), um T zu bekommen.
Induktionsanfang, kein Union(u,v)✓
Tiefe(u)$= 0 = log_2 1$ $(2^0 = 1)$
Induktionsschluss:

T durch Union (u,v) und R = (u, i), S = (v, j)

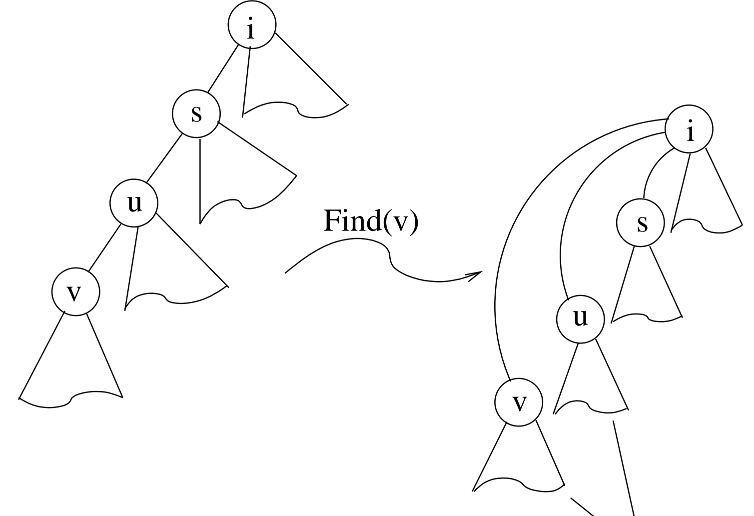
T = (v, i+j), u, R, S

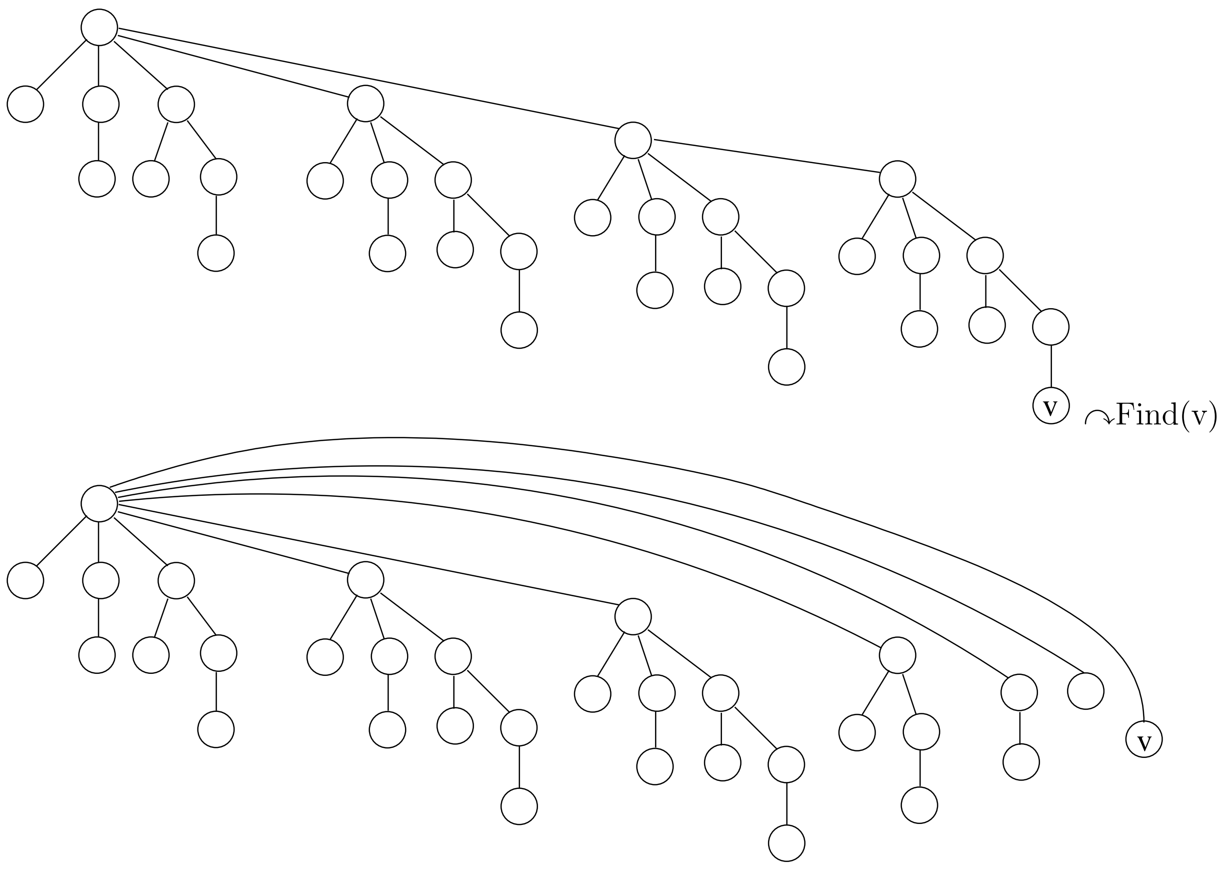
Sei $i \leq j$, dann
1. Fall: keine größere Tiefe als vorher. Die Behauptung gilt nach Induktionsvoraussetzung, da T mehr Elemente hat erst recht. 2. Fall: Tiefe vergrößert sich echt. Dann aber $Tiefe(T) = Tiefe(R) + 1 \leq (log_2|R|) + 1 = log_2(2|R|) \leq |T|$
Union by Size. $(2^x = |R| \Rightarrow 2 \cdot 2^x = 2^{x+1} = 2|R|)$!
Tiefe wird immer nur um 1 größer. Dann mindestens Verdopplung der # Elemnte.

Mit Bäumen und Union-by-Size Laufzeit der Schleife:
$\leq 2|E|$-mal Find(u): $O(|E| log|V|)$ ($log|V| \ldots$ Tiefe der Bäume)
$n - 1$-mal Union(u,v): $O(n)$
Rest $O(|E|)$ Also insgesamt $O(|E| \cdot log|V|)$
Vorne: $O(|V|^2)$. Für dichte Graphen ist $|E| = \Omega(\frac{|V|^2}{log|V|})$ so keine Verbesserung erkennbar.

Beispiel 7.8 (*Wegkompression*):

Wegkompression





Im Allgemeinen $\Omega(logN)$ und $O(logN)$

Algorithmus Wegkompression

```
Find(v)
1. v auf Keller tun                    // etwa als array implemntieren
2. while (P[v]≠v){                     (wie Schlange)
```

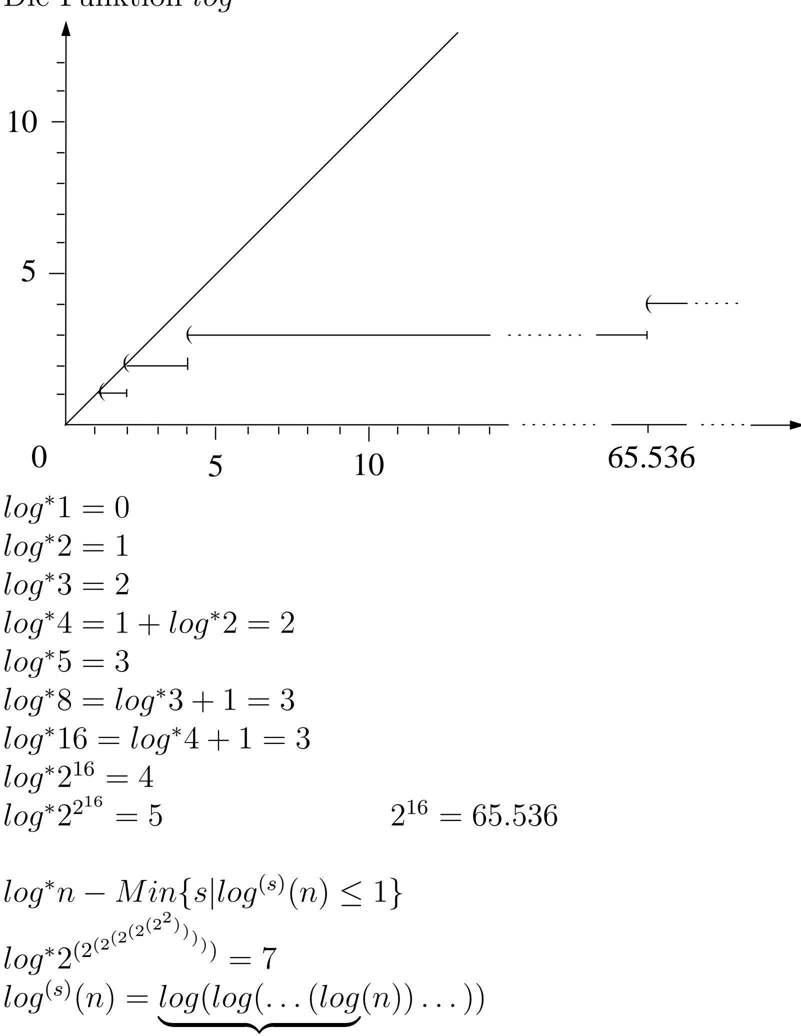
```
3.    v = P[v];
      v auf Keller tun
   }
4. v ausgeben;
5. for (w auf dem Keller){
      P[w] = v;                          //Können auch Schlange, Liste oder
   }                                     //sonstwas nehmen, da Reihenfolge egal
```

Laufzeit $O(log|V|)$ reicht immer, falls Union-by-Size dabei.
Es gilt sogar (einer der frühen Höhepunkte der Theorie der Datenstrukturen):
$n-1$ Unions, m Finds mit Union-by-Size und Wegkompression in Zeit

$$O(n + (n+m) \cdot log^*(n))$$

bei anfangs $P = \{\{1\}, \{2\}, \ldots, \{n\}\}$.
Falls $m \geq \Omega(n)$ ist das $O(n + m \cdot log^*(n))$.
(Beachten Sie die Abhängigkeit der Konstanten: O-Konstante hängt von Ω-Konstante ab!)
Was ist $log^*(n)$?

Die Funktion log^*

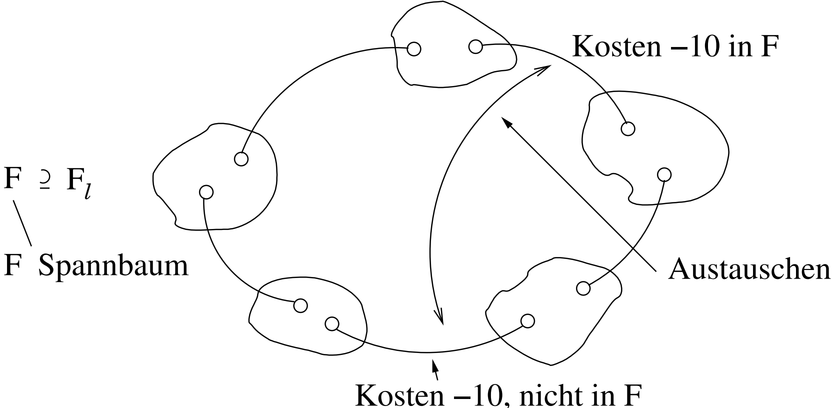


$log^*1 = 0$
$log^*2 = 1$
$log^*3 = 2$
$log^*4 = 1 + log^*2 = 2$
$log^*5 = 3$
$log^*8 = log^*3 + 1 = 3$
$log^*16 = log^*4 + 1 = 3$
$log^*2^{16} = 4$
$log^*2^{2^{16}} = 5$ $\qquad$ $2^{16} = 65.536$

$log^*n - Min\{s|log^{(s)}(n) \leq 1\}$
$log^*2^{(2^{(2^{(2^{(2^{2})})})})} = 7$
$log^{(s)}(n) = \underbrace{log(log(\ldots(log}_{s-\text{ mal}}(n))\ldots))$

Kruskal bekommt dann eine Zeit von $O(|V| + |E| \cdot log^*|V|)$
Fast $O(|V| + |E|)$ bei vorsortierten Kanten natürlich nur $\Omega(|V| + |E|)$ in jedem Fall.

Nachtrag zur Korrektheit von Kruskal:
1. Durchlauf: Die Kante mit minimalen Kosten wird genommen. Gibt es mehrere, so ist egal, welche.
$l + 1$-ter Lauf: Die Kante mit minimalen Kosten, die zwei Komponenten verbindet, wird genommen.



Alle Kanten, die zwei Komponenten verbinden, haben Kosten $\geq$ -10!
Beachte: Negative Kosten sind bei Kruskal kein Problem, nur bei branch-and-bound (vorher vergrößern)!
Binomialkoeffizient $\binom{n}{k}$ für $n \geq k \geq 0$.

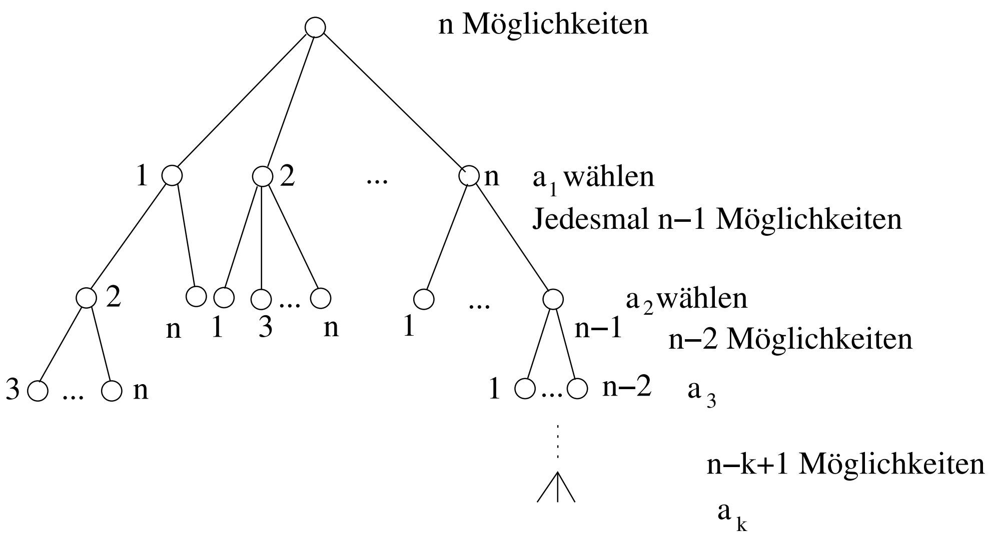
$$\binom{n}{k} = \frac{n!}{(n \cdot k) \cdot k!} = \frac{\overbrace{n(n-1) \cdot \ldots \cdot (n-k+1)}^{\text{oberste k Faktoren von n!}}}{k!}$$

$0! = 1! = 1.$

$\binom{n}{k} = \#$ Teilmengen von $\{1, \ldots, n\}$ mit genau k Elementen

$\binom{n}{1} = n, \binom{n}{2} = \frac{n(n-1)}{2}, \binom{n}{k} \leq n^k.$ **Beweis 7.9**:

Auswahlbaum für alle Folgen $(a_q, \ldots, a_k)$ mit $a_i \in \{1, \ldots, n\}$, alle a_i verschieden.



Der Baum hat $\underbrace{n(n-1) \cdot \ldots \cdot \overbrace{(n-k+1)}^{n-(k-1)}}_{\text{k Faktoren (nicht k-1, da 0, 1, \ldots, k+1 genau k Zahlen)}} = \frac{n!}{(n-k)!}$ Jedes Blatt $\Longleftrightarrow$ genau eine Folge $(a_1, \ldots, a_k)$

Jede Menge aus k Elementen kommt $k!$-mal vor: $\{a_1, \ldots, a_k\}$ ungeordnet, geordnet als $(a_1, \ldots, a_k), (a_2, a_1, \ldots, a_k), \ldots (a_k, a_{k-1}, \ldots, a_2, a_1) k!$ Permutationen.

Also $\frac{n!}{(n-k)!k!} = \binom{n}{k}$ Mengen mit k verschiedenen Elementen.

Algorithmus Minimaler Spannbaum nach Prim 1963

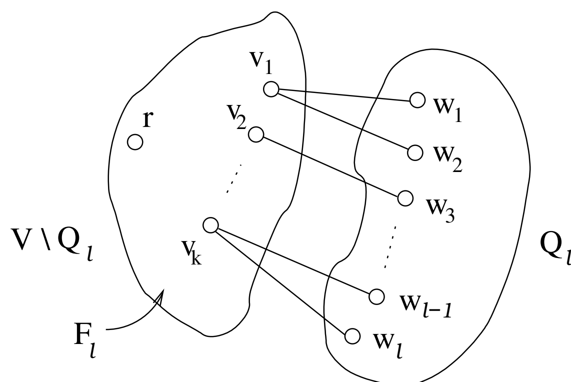
Eingabe: Wie bei Kruskal
Ausgabe: Menge von Kanten F eines minimalen Spannbaumes

```
1. Wähle irgendeinen Startknoten r
   Q = V\{r}                         // Q enthält immer die Knoten,
   F = ∅                             // die noch bearbeitet werden müssen.
2. while (Q ≠ ∅ {
3.    M = {{v, w}| v ∈ V\ Q, w ∈ Q}
                                     // M = Menge der Kanten mit
                                     // genau einem Knoten in Q
4.    {v, w} = eine Kante von minimalen Kosten in M
      F = F ∪{{v, w}};
      Q = Q ohne den einen Knoten aus {v, w}, der auch zu Q gehört
```

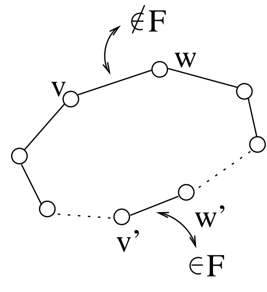
Korrektheit mit der Invariante:

- Es gibt einen minimalen Spannbaum $F \supseteq F_l$
- $Q_l =$
  $l = 0$ ✓.

Gelte die Invariante für l und finde ein $l+1$-ter Lauf der Schleife statt. Sei $F \supseteq F_l$ ein minimaler Spannbaum, der nach Induktionsvoraussetzung existiert.



Werde $\{v, w\}$ im $l+1$-ten Lauf genommen. Falls $\{v, w\} \in F$, dann gilt die Invariante auch für l+1. Sonst gilt $F \cup \{\{v, w\}\}$ enthält einen Kreis, der $\{v, w\}$ enthält. Dieser enthält mindestens eine weitere Kante $\{v, w\}$ mit $v' \in V \setminus Q_l, w' \in Q_l$.



Es ist $K(\{v,w\}) \leq K(\{v',w'\})$ gemäß Prim. Also, da F minimal, ist $K(\{v,w\} = K(\{v',w'\})$. Können in F die Kante $\{v',w'\}$ durch $\{v,w\}$ ersetzen und haben immernoch einen minimalen Spannbaum. Damit Invariante für l+1.
Öaufzeit von Prim:

- $n-1$ Läufe durch 2.
- 3. und 4. einmal $O(|E|)$ reicht sicherlich, da $|E| \geq |V|-1$.

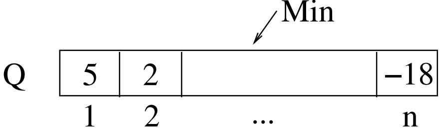
Also $O(|V| \cdot |E|)$, bis $O(n^3)$ bei $|V| = n$.
Bessere Laufzeit durch bessere Verwaltung von Q.

- Maßgeblich dafür, ob $w \in Q$ in den Baum aufgenommen wird, sind die minimalen Kosten einer Kante(!) $\{v,w\}$ für $v \notin Q$.
- Also array key[1 ... n] of real mit der Intention: Für alle $w \in Q$ ist key[w] = min. Kosten einer Kante $\{v,w\}$, wobei $v \notin Q$. (key[w] = ∞, gdw. keine solche Kante existiert.)
- Außerdem array ka[1, ..., n] of {1, ..., n} mit: Für alle $w \in Q$
  ka[w] = v $\iff$ $\{v,w\}$ ist keine Kante minimaler Kosten mit $v \notin Q$

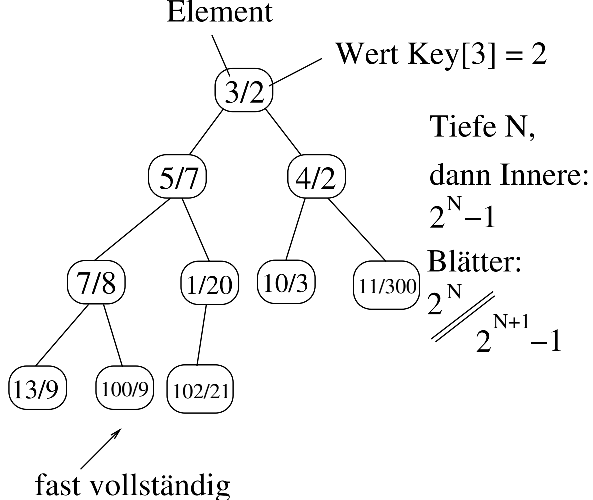
Wir werden Q mit einer Datenstruktur für die *priority queue* (Vorrangwartesschlange) implementieren:

- Speichert Menge von Elementen, von denen jedes einen Schlüsselwert hat (keine Eindeutigkeitsforderung).
- Operation Min gibt uns (ein) Elemnet mit minimalem Schlüsselwert.
- DeleteMin löscht ein Element mit minimalem Schlüsselwert.
- Insert(v, s) = Element v mit Schlüsselwert s einfügen.

Wie kann man eine solche Datenstruktur implementieren?
1. Möglichkeit: etwa als Array

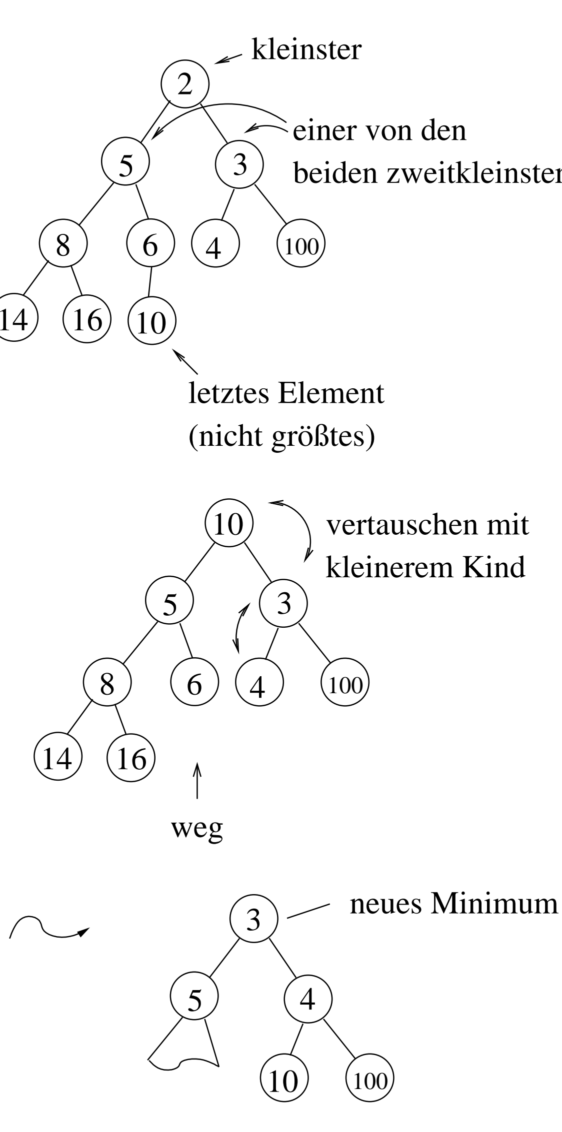


Indices = Elemente,
Einträge in Q = Schlüsselwerte, Sonderwert (etwa -∞) bei „nicht vorhanden“.
Zeiten:
Insert(v,s) in $O(1)$ (sofern keine gleichen Elemente mehrfach auftreten)
Min in $O(1)$
DeleteMin - $O(1)$ fürs Finden des zu löschenden Elementes, dann aber $O(n)$, um ein neues Minimum zu ermitteln.
2. Möglichkeit:
Darstellung als Heap.
Heap = „fast vollständiger“ binärer Baum, dessen Elemente (= Knoten) bezüglich Funktionswerten key[j] nach oben hin kleiner werden.

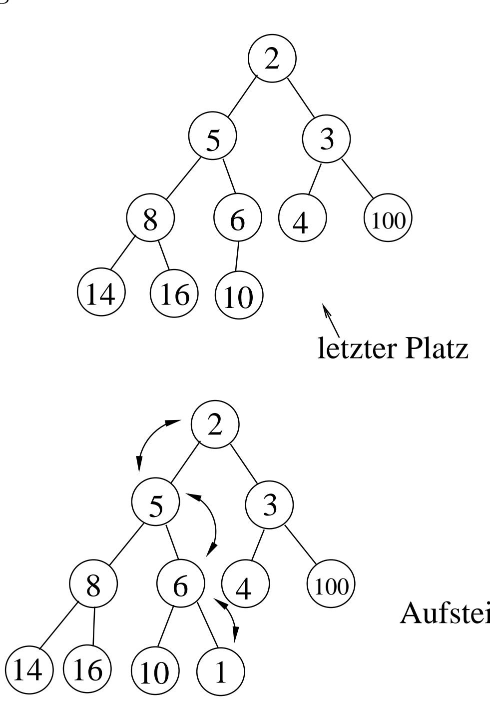


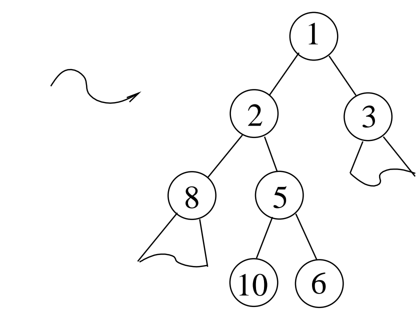
Heapeigenschaft: für alle u, v, u Vorfahr von $v \Longrightarrow key[u] \leq key[v]$

Beispiel Minimum löschen:
Nur Werte: Key[I]:



Beispiel Einfügen
Element mit Wert 1 einfügen

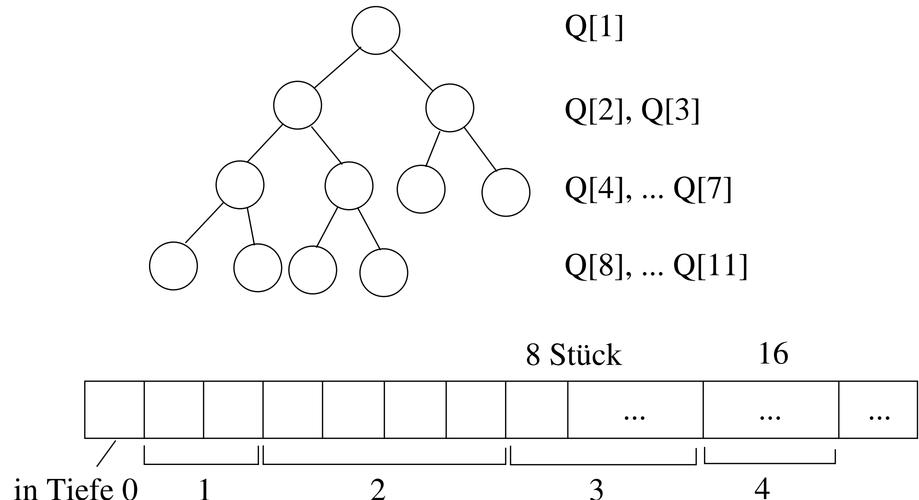




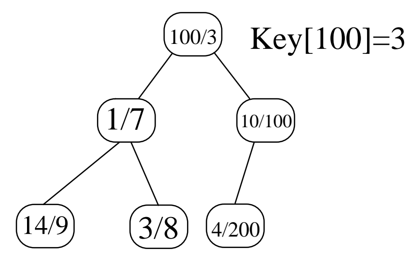
Priority Queue mit Heap, sagen wir Q.
Min {
1. Gib Element der Wurzel aus
} O(1)
Deletemin{
1. Nimm „letztes“ Element, setze es auf Wurzel x:=Wurzel(-Element)
2. While Key[x]≥Key[linker Sohn von x] oder rechter Sohn
{tausche x mit kleinerem Sohn}
}
N Elemente, $O(logN)$, da maximal $O(logN)$ Durchläufe
Insert(v,s){//Key[v]=s.
1. Tue v als letztes Element in den Heap.
2. While Key[Vater von v]¿Key[v]
{vertausche v mit seinem Vater}
}
N Elemente $O(logN)$
Vergleich der Möglichkeiten der Priority Queue

	Min	Deletemin	Insert	Element finden
Heap	1	log N	logN	N
Array o. Liste	1	N(!)	1	1(Array), N(Liste)

Heap als Array $Q[1, \ldots, N]$



Vater I{// I Index aus 1, ..., N
if (I+1){
Gebe $\lfloor \frac{1}{2} \rfloor$ aus}
Linker Sohn I 2I
Rechter Sohn 2I+1
Abschließendes Beispiel



Array Q
Q[1] = 100 Key[100] = 3
Q[2] = 1 Key[2] = 7
Q[3] = 10 Key[10 = 100
Q[4] = 14
Q[5] = 3
Q[6] = 4 ⋮
wenn Elemente nur einmal

Suchen nach Element oder Schlüssel wird nicht (!) unterstützt.

Algorithmus (Prim mit Q in heap)

```
1. Wähle Startknoten r.
   for each v∈ Adj[r]{
     key[v]=k({r,v});
     kante[v]=r}
   Für alle übrigen v∈V{
     key[v]=∞; kante[v]=∞}
   Füge V\{r} mit Schlüsselwerten key[v] in heap Q ein.
2. while Q≠ ∅{
3.   w = Min; DeleteMin_Q;
     F = F∪{{kante[w],w}}
4. for each u∈ Adj[w]∩ Q{
     if K({u,w})<key[u]{
       key[u] = K({u,w});
       Q anpassen}
     }
   }
```

Korrektheit mit zusätzlicher Invariante:

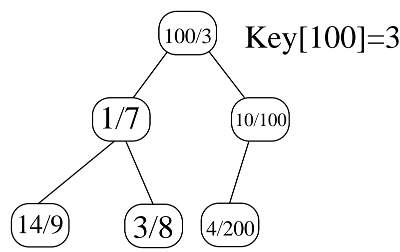
Für alle $w \in Q_l$

key$_l$[w] = minimale Kosten einer Kante {v, w} für $v \notin Q_l$}
key$_l$[w] = ∞, wenn eine solche Kante nicht existiert.

Laufzeit

1. $O(n) + O(n \cdot logn)$ für das Füllen von Q.
(Füllen von Q in $O(n)$ - interessante Übungsaufgabe)
2. $n-1$ Läufe
3. Einmal $O(logn)$, insgesamt $O(n \cdot logn)$
4. Insgesamt $O(|E|)$ + $|E|$-mal Anpassen von Q.

Anpassen von Heap Q
Operation Decreasekey(v,s)
$s < key[v]$, neuer Schlüssel



Decreasekey(v,s)

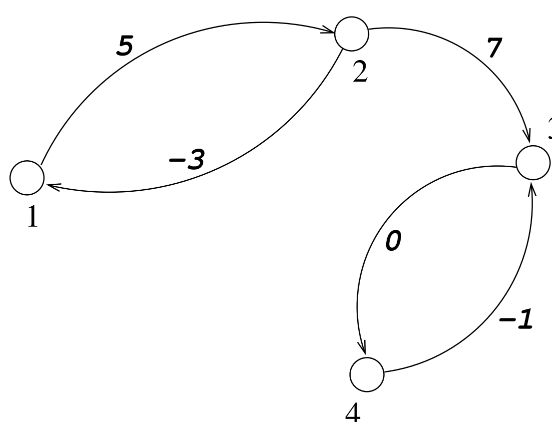
1. Finde Element v in Q //nicht von Q unterstützt!
2. while key[Vater von v]¿s tausche v mit Vater

Laufzeit Deckey(v,s):
2. $O(logN)$ 1. $O(N)$(!!).
zum Finden: „Index drüberlegen“
Direkte Adressen:
Ind[1] = 3, Ind[3] = 5, Ind[4]= 6, Ind[100] = 1
Finden in $O(1)$
Falls Grundmenge zu groß, dann Suchbaum: Finden $O(logN)$
Falls direkte Adressen dabei:
Einmaliges Anpassen von Q (und Index) $O(log|V|)$
Dann Prim insgesamt $O(|E|log|V|)$ (Beachte vorher $O(|E| \cdot |V|)$)
(wie Kruskal mit Union by Size.)

8 Kürzeste Wege

Hier sind alle Graphen gerichtet und gewichtet, d.h. wir haben eine Kostenfunktion $K : E \longrightarrow \mathbb{R}$ dabei.

Also etwa:



$K(1, 2) = 5, K(2, 1) = -3, K(2, 3) = -7, K(3, 4) = 0$

Ist $W = (v_0, v_1, \dots, v_k)$ irgendein Weg im Graphen, so bezeichnet $K(W) = K(v_0, v_1) + K(v_1, v_2) + \cdots + K(v_{k-1}, v_k)$ die Kosten von W. $K(v_0) = 0$
Betrachten wir die Knoten 3 und 4, so ist $K(3,4) = 0, K(4,3) = -1$ und $K(3,4,3,4) = -1, K(3,4,3,4,3,4) = -2, \dots$
Negative Kreise erlauben beliebig kurze Wege. Wir beschränken uns auf einfache Wege, d.h. noch einmal, dass alle Kanten verschieden sind.

Definition 8.1 (*Distanz*):

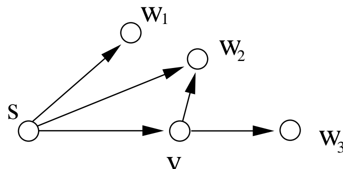
> Für $u, v \in V$ ist $Dist(u,v) = Min\{K(W) | W = (v_0 = u, v_1, \dots, v_k = v)$ ist einfacher Weg von u nach v $\}$, sofern es einen Weg u ○⟶○ v gibt. $Dist(u,v) = \infty$, wenn es keinen Weg u ○⟶○ v gibt.
> $Dist(v,v) = 0$

Uns geht es jetzt darum, kürzeste Wege zu finden. Dazu machen wir zunächst folgende Beobachtung:

Ist $u = v_0 \to v_1 \to v_2 \to \dots \to v_{k-1} \to v_k = v$ ein kürzester Weg von z nach v, so sind alle Wege v_0 ○⟶○ v_i oben auch kürzeste Wege. Deshalb ist es sinnvoll, das single source shortest path-Problem zu betrachten, also alle kürzesten Wege von einem Ausgangspunkt zu suchen. Ist s unser Ausgangspunkt, so bietet es sich an, eine Kante minimaler Kosten von s aus zu betrachten:

s ○⟶○ v

Gibt uns diese Kante einen kürzesten Weg von s nach v? Im Allgemeinen nicht! Nur unter der Einschränkung, dass die Kostenfunktion $K : E \longrightarrow \mathbb{R}^{\geq 0}$ ist, also keine Kosten < 0 erlaubt. Diese Bedingung treffen wir zunächst einmal bis auf weiteres.
Also, warum ist der Weg s ○⟶○ v ein kürzester Weg? (Nachfolgende Aussage gilt bei negativen Kosten so nicht:) Jeder andere Weg von s aus hat durch seine erste Kante s ○⟶○ w mindestens die Kosten $K(s,v)$.
Der *Greedy-Ansatz* tut es am Anfang. Wie bekommen wir einen weiteren kürzesten Weg von s aus?

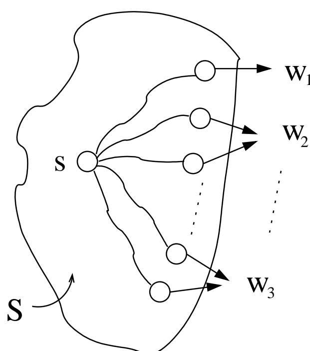


Wir schauen uns die zu s, v adjazenten Knoten an. Oben w_1, w_2, w_3. Wir ermitteln

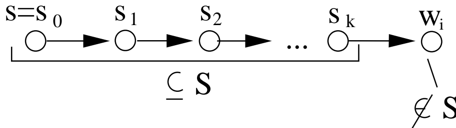
- zu $w_1 K(s, w_1)$
- zu $w_2 Min\{K(s, w_2), K(s,v) + K(v, w_2)\}$
- zu $w_3 K(s,v) + K(v, w_3)$.

Ein minimaler dieser Werte gibt uns einen weiteren kürzesten Weg. Ist etwa $K(s,v) + K(v,w_2)$ minimal, dann gibt es keinen kürzesten Weg s ○⟶○ w_2. Ein solcher Weg müsste ja die

Menge s, v irgendwann verlassen. Dazu müssen aber die eingezeichneten Wege genommen werden und der Weg zu w_2 wird höchstens länger. (wieder wichtig: Kosten ≥ 0).
So geht es allgemein weiter:
Ist S eine Menge von Knoten, zu denen ein kürzester Weg von s aus gefunden ist, so betrachten wir alleKnoten w_i adjazent zu S aber nicht in S.



Für jeden Knoten w_i berechnen wir die minimalen Kosten eines Weges der Art



Für ein w_i werden diese Kosten minimal und wir haben einen kürzesten Weg s ○⟶○ wᵢ.
Wie vorher sieht man, dass es wirklich keinen kürzeren Weg s ○⟶○ wᵢ gibt.
Das Prinzip: Nimm immer den nächstkürzeren Weg von s ausgehend.

Algorithmus (Dijkstra 1959)

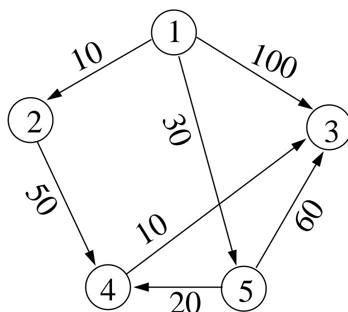
Eingabe: $G = (V, E), K : E \longrightarrow \mathbb{R}^{\geq 0}(!)$
$|V| = n, s \in V$
Ausgabe: array $D[1, \ldots, n]$ of real mit $D[v] = Dist(s, v)$ für $v \in V$.
Datenstrukturen:
S = Menge der Knoten, zu denen kürzester Weg gefunden ist
$Q = V \setminus S$

```
1. D[s] = 0, S = {s}, Q=V\{s}
2. for (i = 1 to n - 1){                 //Suchen noch n-1 kürzeste Wege
3.    M = {{v, w}|v ∈ S, w ∈ Q}
4.    for each w ∈ Q adjazent zu S{
5.       D[w] = Min{D[v] + K(v, w)|(v,w) ∈ M}}
6.    w = ein w ∈ Q mit D[w] minimal;
7.    S = S ∪, Q = Q\{w}}
```

Korrektheit mit Invariante:

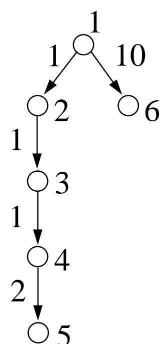
Für alle $w \in S_l$ ist $D[w]$ = Kosten eines kürzesten Weges. Das Argument gilt wie oben bereits

vorgeführt.
Zwecks Merken der Wege das Array $\Pi[1,\dots,n]$ of $1,\dots,n$ mit $\Pi[u]=v \iff$ Ein kürzester s ○⟶○ v ist $(s,\dots,u,v)$.
Π kann leicht in 6. mitgeführt werden. Es ist das Vaterarray des kürzesten-Wege-Baumes mit Wurzel s.



S	W	D[1]	D[2]	D[3]	D[4]	D[5]
1	/	0	10	∞	30	100
{1, 2}	2	0	10	60	30	100
{1, 2, 4}	4	0	10	50	30	80
{1, 2, 4, 3}	3	0	10	50	30	60
{1, 2, 4, 3, 5}	5	0	10	50	30	60

Die Wege werden der Länge nach gefunden:



S
{1}
{1, 2}
{1, 2, 3}
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5, 6}

Kürzester-Wege-Baum

$\Pi[6]=1, \Pi[2]=1, \Pi[3]=2$
$\Pi[4]=5, \Pi[5]=4$

Die Laufzeit ist bei direkter Implementierung etwa $O(|V|\cdot|E|)$, da es $O(|V|)$ Schleifendurchläufe gibt und jedesmal die Kanten durchsucht werden, ob sie zu M gehören. Q und S sind als boolesche Arrays mit $Q[v]=true \iff v\in Q$ zu implementieren.
Die Datenstruktur des dictionary (Wörterbuch) speichert eine Menge von Elementen und unterstützt die Operationen

- Find(u) = Finden des Elementes u innerhalb der Struktur

- Insert(u) = Einfügen
- Delete(u) = Löschen

(Ist die Grundmenge nicht als Indexmenge geeignet: Hashing oder Suchbaum.)
In Q ist ein dictionary implementiert. Jede Operation erfolgt in $O(1)$.
Ziel: bessere Implementierung
Für welche $v \in Q$ kann sich $D[v]$ in 5. nur ändern? Nur für diejenigen, die adjazent zu dem w des vorherigen Laufes sind. Dann sieht es etwa so aus:

Algorithmus (Dijkstra ohne mehrfache Berechnung desselben D[w])

```
1. D[1] = 0; D[v] = ∞ für v ∈ V\{s}; Q = V\{s}; S = {s};
2. for i = 1 to n - 1 {
3.    w = ein w ∈ Q mit D[w] minimal;
4.    S = S ∪ {w}; Q = Q\{w};
5.    for each v ∈ Adj[w]{        //D[v] aussparen für v adjazent zu w
6.       if D[w] + K(w, v) < D[v]{
            D[v] = D[w] + K(w, v) }}
```

Korrektheit mit zusätzlicher Invariante:
Für $w \in Q_l$ ist $D_l[w]$ = minimale Kosten eines Weges der Art

$$\underbrace{s = s_0 \to s_1 \to s_2 \to \dots s_k}_{\subseteq S} \to \underbrace{w}_{\in Q}$$

Man kann auch leicht zeigen:
Für alle $v \in S_l$ ist $D_l[v] \leq D_l[w_l]$,
w_l = das Minimum der l-ten Runde. Damit ändert 6. nichts mehr an $D[v]$ für $v \in S_l$.

Laufzeit:

3. insgesamt $n - 1$ - mal Minimum finden
4. insgesamt $n - 1$ - mal Minimum löschen
5., 6. insgesamt $O(|E|)$, da dort jede Adjazenzliste nur einmal durchlaufen wird.

Mit Q als boolesches Array:

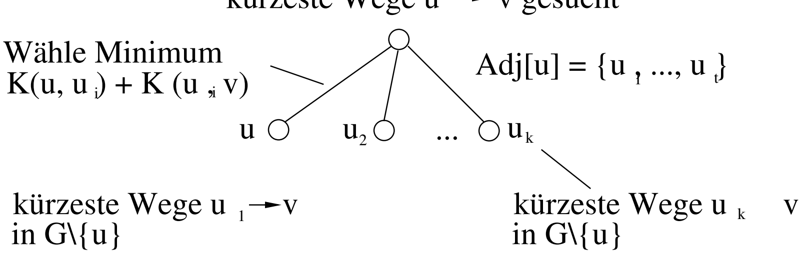
3. $O(n^2)$ insgesamt. Das Finden eines neuen Minimums nach dem Löschen dauert $O(n)$.
4. $O(n)$ insgesamt, 1. Löschen $O(1)$
5., 6. Bleibt bei $O(|E|)$.
Also alles in allem $O(n^2)$.

Aber mit Q benötigen wird die klassischen Operationen der priority queue, also Q als heap, den Schlüsselwert aus D.
3. $O(n)$ insgesamt. 4. $O(n \cdot logn)$ insgesamt.
5., 6. $|E|$-mal heap anpassen, mit Decreasekey(v, t) (vgl. Prim) $O(|E| \cdot logn)$. Zum Finden Index mitführen!

Also: Q als boolesches Array:
$O(n^2)$ (Zeit fällt beim Finden der Minima an).

Q als heap mit Index:
$O(|E|logn)$ (Zeit fällt beim Decreasekey(v, t) an).

Ist $|E|logn \geq n^2$, also ist der Graph sehr dicht, dann ist ein Array besser (vgl. Prim).
Ab jetzt betrachten wir wieder eine beliebige Kostenfunktion $K : E \longrightarrow \mathbb{R}$.
Der Greedy-Ansatz wie bei Dijkstra geht nicht mehr. Kürzeste Wege u○———►○v findet man durch Durchprobieren.
Die Zeit ist $(n-2)! \geq 2^{\Omega(logn+n)} \gg 2^n$!
Es werden im Allgemeinen viele Permutationen generiert, die gar keinen Weg ergeben. Das vermeidet backtracking:



Dies ist korrekt, da Teilwege kürzester Wege wieder kürzeste Wege sind und wir mit kürzesten Wegen immer nur einfache Wege meinen.
Das Ganze ist leicht durch Rekursion umzusetzen:

Eingabe: $G = (V, E), K : E \Longrightarrow \mathbb{R}$ beliebig.

```
KW (W, u, v)                     //u, v ∈ W, W = nach b betrachtete Knotenmenge
1. if (u == v)
       return 0;
2. l = ∞;
3. for each w ∈ Adj[u] ∩ W{
       l' = KW(W\ {u}, w, v);
       l' = K(u, w) + l'_j;
```

```
      if(l' < l)
        l = l';                   //Hier Π[w] = u gibt kürzeste Wege selbst
   }
4. return l                       //Ausgabe ∞, wenn Adj[u] ∩ W = Ø.
```

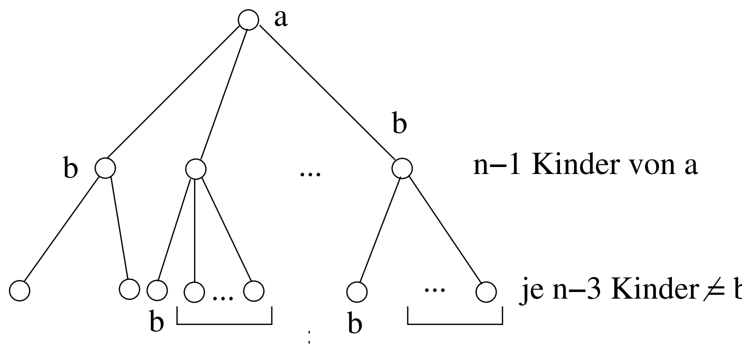
Aufruf: KW(V, a, b)
Korrektheit: Induktion über $|W|$.
Etwas einfacher ist es, alle einfachen Wege u○——►○v systematisch zu erzeugen und den Längenvergleich nur am Ende durchzuführen.
Datenstruktur: L
L = Liste von Knoten als Keller implementiert
Intention: L enthält den Weg vom Startknoten a bis zum aktuellen Knoten.
G = Liste von Knoten. Der aktuell kürzeste Weg a○——►○b
L, G = globale Arrays
KW(!, u, v){

```
KW(!, u, v){ 1. if (u == v){
      if (K(L) < K(G)) G = L;   K(L), K(G) = Kosten von L, G
   }
2. for each w ∈ Adj[u] ∪ W{
      w auf Keller L ablegen;
      w vom Keller L löschen
      KW(W\{u}, wm v);
   }} // Aufruf mit K(L) = ∞, K(G) = ∞, L = (a), KW(v, a, b)
```

Korrektheit mit der Aussage: Beim Aufruf von KW(W, w, v) ist L = w. . . a.
Am Ende des Aufrufs KW(W, w, v) enthält G einen kürzesten Weg der Art $G = b \dots L$. (L ist das L von oben)
Das Ganze induktiv über $|W|$.

Laufzeit der rekursiven Verfahren?

Enthält der Graph alle $n(n-1)$ Knoten, so sieht der Aufrufbaum folgendermaßen aus:



Also $\geq (n-2)(n-3) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 \geq 2^{\Omega(n \cdot log n)}$.

Wieviele verschiedene Aufrufe haben wir höchstens? Also Aufrufe KW(W, c, d)?

- $W \subseteq V \leq 2^n$ Möglichkeiten.
- $c, d \leq n$ Möglichkeiten, da Endpunkt immer gleich.

Also $n \cdot 2^n = 2^{logn} \cdot 2^n = 2^{O(n)}$.
Also bei $2^{\Omega(nlogn)}$ Aufrufen wie oben viele doppelt. Es ist ja $\frac{2^{O(n)}}{2^{\Omega(nlogn)}} = \frac{1}{2^{\Omega(logn)}} = \frac{1}{n^\epsilon} \longrightarrow 0$ (wobei $n^\epsilon = n^{\Omega(1)}$) für ein $\epsilon > 0$ konstant.
Also: Vermeiden der doppelten Berechnung durch Merken (Tabellieren).

Dazu das Array $T \overbrace{[1 \ldots 2^n]}^{\in\{0,1\}^n}[1 \ldots n]$ of int mit der Interpretation: Für $W \subseteq V$ mit $b, v \in W$ ist $T[W, v] =$ Länge eines kürzesten Weges v ○⟶○ b nur durch W.

Füllen $T[W, v]$ für $|W| = 1, 2, \ldots$
1. $|W| = 1$, dann $v = b \in W, T[\{b\}, b] = 0$ 2. $|W| = 2$, dann $T[W, b] = 0, T[W, v] = K(v, b)$, wenn $v \neq b$
3. $|W| = 3, T[W, b] = 0$
T[W, v] ergibt sich aus:

$l = \infty, W = W \setminus \{v\}$
for each $u \in$ Adj[v]∩W{
  $l' = K(v, u) + T[W', u]$
  if $(l' < l)$
    $l = l'$
}
$T[W, v] = l$

Also, das heißt:

$T[W, v] = Min\{Q(v, u) + T[W', u] | W' = W \setminus \{v\}, u \in W\}$
$T[W, v] = \infty$, wenn keine Kante v ○⟶○ u mit $u \in W$.
Das Mitführen des Weges durch $\Pi[W, v] = u, u$ vom Minimum ermöglicht eine leichte Ermittlung der Wege. Also, der Eintrag $T[W, v]$ wird so gesetzt:

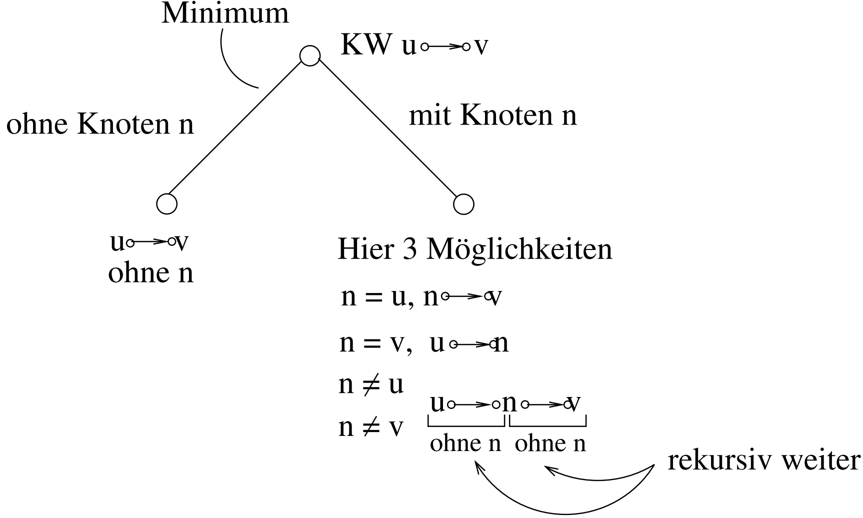
```
Setze(W, v){
1. if (v == b) {
      T[W, v] = 0;
      Π[W, v] = u;
```

Dann insgesamt

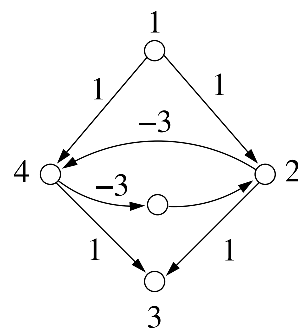
```
KW(V, a, b){

1. for i = 1 to n {
2.    for each W ⊆ V, b ∈ W, |W| = i{
3.       for each v ∈ W{
4.          Setze(W, v);
         }}}}
```

Dann enthält $T[V, a]$ das Ergebnis. Weg ist $a_0 = a, a_1 = \Pi[V, a], a_2 = \Pi[V \setminus \{a\}, a_1], \ldots, a_i = \Pi[V \setminus \{a_0, \ldots, a_{i-2}\}, a_{i-1}], \ldots$
Invariante für 1.
Nach dem $l-$ten Lauf ist $T_l[W, v]$ korrekt für alle W mit $|W| \leq l$.
Laufzeit: $O(n^2 \cdot 2^n)$, da ein Lauf von *Setze(W, v)* in $O(n)$ möglich ist.

Das hier verwendete Prinzip:
Mehr rekursive Aufrufe als Möglichkeiten $\Rightarrow$ Tabellieren der Aufrufe heißt:
Dynamisches Programmieren (Auffüllen einer Tabelle, 1950er)
Wir beginnen einmal mit einer anderen Fallunterscheidung beim backtracking:



Korrektheit: Ist $u, v + n$ und ist ein kürzester Weg u○⟶○v ohne n, dann wird dieser nach Induktionsvoraussetzung links gefunden. Erhalte jetzt jeder kürzeste Weg u○⟶○v den Knoten n. Wird ein solcher unbedingt rechts gefunden? Nur dann, wenn die kürzesten Wege u○⟶○n, n○⟶○v keinen weiteren gemeinsamen Knoten haben. Wenn sie einen gemeinsamen Knoten w haben, so u○⟶○w⟶○n⟶○w⟶○v. Da dieser Weg kürzer ist als jeder Weg ohne n, muss w○⟶○n⟶○w ein Kreis der Länge < 0 sein.
Betrachten wir also jetzt $K : E \longrightarrow \mathbb{R}$, aber so, dass keine Kreise < 0 existieren.



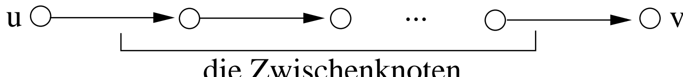
kürzester Weg 1 ○→○ 3 ohne 4, Kosten 2

kürzester Weg 1 ○→○→○ 4 Kosten −2 (über 2)

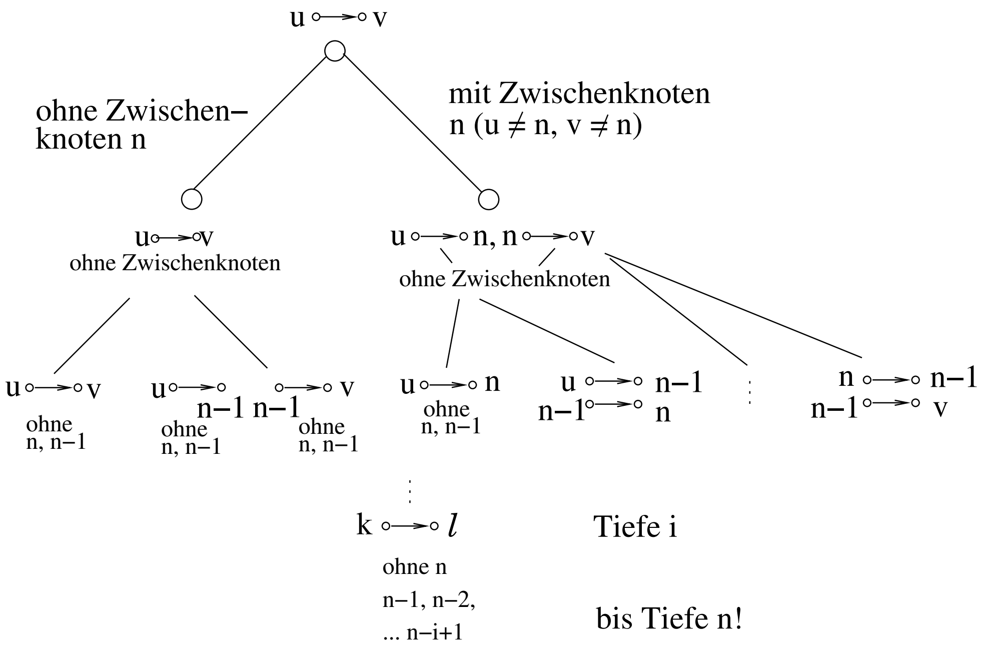
kürzester Weg 4 ○→○→○ 3 Kosten −2 (über 2)

und (2, 4, ○, 2) Kreis der Kosten −6

Es ist günstiger, die Fallunterscheidung nach den echten Zwischenknoten zu machen



Also Backtracking:



Also rekursive Prozedur:
KW(u, v, i) für kürzeste Wege u○→○v ohne Zwischenknoten $n, n-1, n-1, \ldots, n-i+1$. Kürzester Weg ergibt sich durch KW(u, v, 0). Rekursionsanfang KW(u, v, n) gibt Kante u○→○v. Wieviele verschiedene Aufrufe? $O(n^3)$!

Also dynamisches Programmieren:
$T[u, v, i]$ = kürzester Weg u○→○v wobei Zwischenknoten $\subseteq \{1, \ldots, i\}$ (Also nicht unter

i+1, ..., n.)
0. $T[u, v, 0] = K(u, v)$ für alle u, v!
1. $T[u, v, 1] = Min\{T[u, 1] + T[1, v], T[u, v, 0]\}$ für alle u, v
$\vdots$
n. $T[u, v, n] = Min\{T[v, n, n-1] + T[n, v, n-1], T[u, v, n-1]$ für alle u, v
All pairs shortest path.

Algorithmus Floyd Warshall

G ohne Kreise ¡ 0, V = {1, ..., n}

```
1. T[u, v] = 0, T[u, v] = K(u, v) für (u, v) ∈ E
   T[u, v] = ∞ für u ≠ v, (u, v) ∉ E
2. for i = 1 to n{
     for each (u, v) ∈ V{        Alle geordneten Paare.
       T[u, v] = Min{T[u, v], T[u, i] + T[i, v]}
   }
```

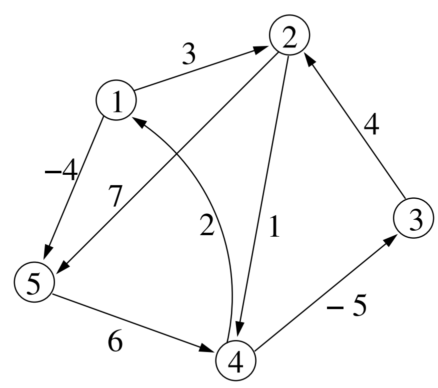
Invariante: Nach $l-$tem Lauf der Schleife in 1. entsteht $T_l[u, v] =$ Länge eines kürzesten Weges u○ ⟶ ○v mit Zwischenknoten $\subseteq \{1, \ldots, l\}$.
Wichtig: Keine negativen Kreise, denn in $l+1$-ter Runde gilt $T_l[u, l+1] + T_l[l+1, v] < T_l[u, v]$, dann ist der Weg hinter $T_l[u, l+1] + T_l[l+1, v]$ ein einfacher!

u ○ ⟶ ○ (l+1) ⟶ ○ v

kürzer als kürzeste Wege u○ ⟶ ○v ohne $l + 1$, dann oben nicht

u○ ⟶ ○w ⟶ ○ (l+1) ⟶ ○w ⟶ ○v

da sonst K(u ○ ⟶ ○ (l+1) ⟶ ○ v) <0 sein müsste.
Erkennennegative Kreise: Immer gilt, also bei beliebiger Kostenfunktion, dass Floyd Warshall die Kosten eines Weges von u○ ⟶ ○v in $T[u, v]$ leitet. Dann $T[u, v] < 0 \iff G$ hat Kreis < 0.
Laufzeit: $O(n^3)$ (Vergleiche Dijkstra $O(|E| log |V|)$ oder $O(|V|^2)$.)



Am Anfang

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

Bevor k auf 2 geht:

0	3	∞	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	**5**	-5	0	∞
∞	∞	∞	6	0

Es ist A[i, j] = Min{A[i, j], A[i, 1] + A[1, j]}
Direkte Wege.
Wege mit Zwischenkomponenten 1.
In 1 bevor k auf 3 geht:
Direkte Wege + Wege mit Zwischenknoten 1 + Wege mit Zwischenknoten 1, 2.
Nicht: mit 2 Zwischenknoten!

9 Flüsse in Netzwerken

$G = (V, E)$ gerichtet, Kostenfunktion $K : E \longrightarrow \mathbb{R}^{\geq 0}$. Hier nun $K(u, v)$ = die Kapazität von (u, v). Stellen uns vor, (u, v) stellt eine Verbindung dar, durch die etwas fließt (Wasser, Strom, Autos, ...).
Dann besagt $K(u, v) = 20$ zum Beispiel

- $\leq$ 20 Liter Wasser pro Sekunde
- $\leq$ 10 produzierte Waren pro Tag ...

Definition 9.1 (*Flussnetzwerk*):

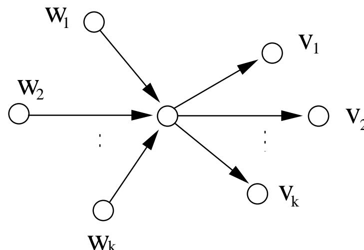
in Flussnetzwerk ist ein gerichteter Graph $G = (V, E)$ mit $K : V \times V \longrightarrow \mathbb{R}^{\geq 0}, K(u, v) = 0$ falls $(u, v) \notin E$.
Außerdem gibt es zwei ausgezeichnete Knoten

- $s \in V$, Quelle (source)
- $t \in V$, Ziel (target, sink)

Wir verlangen noch: Jeder Knoten $v \in V$ ist auf einem Weg s ○ ⟶ ○ (v) ⟶ ○ t.

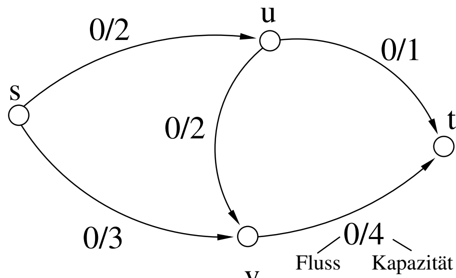
Ziel: Ein möglichst starker Fluss pro Zeiteinheit von s nach t. Fluss durch u ○──────► ○ v $\in E$ ist $f(u, v) \in \mathbb{R}^+$. Es muss für einen Fluss f gelten:

- $f(u, v) \leq K(u, v)$
- Was zu u hinfließt, muss auch wegfließen (sofern $u \neq s, u \neq t$). Also

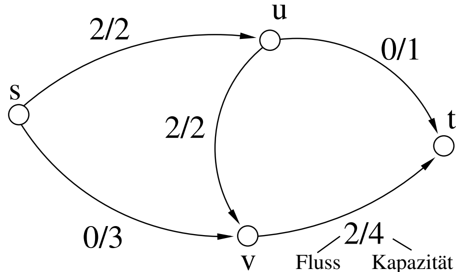


dann $\sum f(w_i, u) = \sum (w, v_i)$.

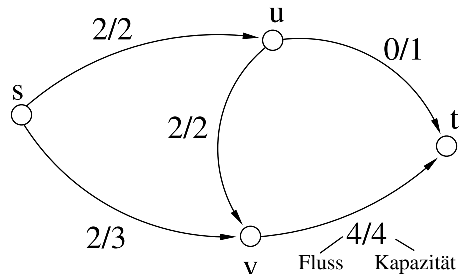
Prinzip: Methode der Erweiterungspfade (Ford - Fulkerson 1950er Jahre)
1. Beginne mit dem Fluss 0, f(u, v) = 0 für alle (u, v)
2. Suche einen Weg (Erweiterungspfad) $s = v_0 \rightarrow v_1 \rightarrow v_2 \cdots \rightarrow v_k = t$ so dass für alle $f(v_i, v_{i+1}) < K(v_i, v_{i+1})$.
3. Erhöhe den Fluss entlang des Weges so weit es geht. Dann bei 2. weiter.
Wieso negative Flüsse?



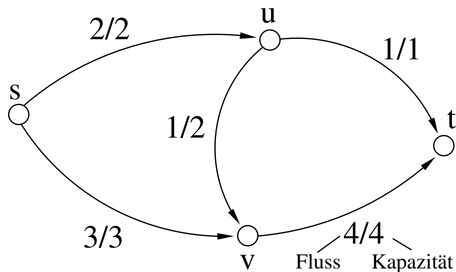
Weg (s, u, v, t) + 2



Weg (s, v, t) + 1

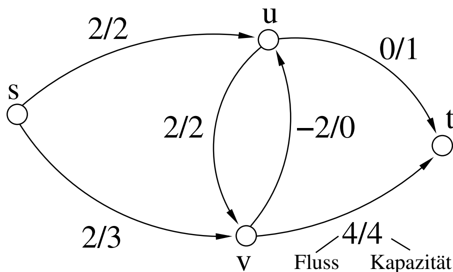


kein Erweiterungspfad, aber



ist größer!

Mit negativen Flüssen:

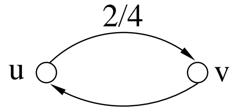


Immer ist $f(u,v) = -f(v,u)$.
Weg (s, v, u, t) + 1 gibt den minimalen Fluss.
Wir lassen auch $f(u,v) < 0$ zu.

Definition 9.2 (E):

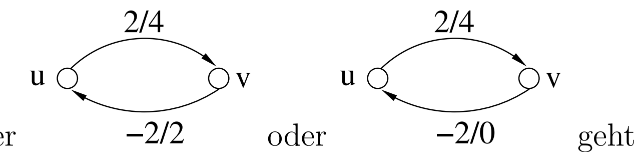
n Fluss ist eine Funktion $f : V \times V \longrightarrow \mathbb{R}$ mit

- $f(u,v) \leq K(u,v)$ für alle u, v (Kapazitätsbedingung)
- $f(u,v) = -f(v,u)$ für alle u, v (Symmetrie)
- Für alle $u \neq s, u \neq t$ gilt $\sum\limits_{v \in V} f(u,v) = 0$ (Kirchhoffsches Gesetz)
  $|f| = \sum\limits_{v \in V} f(s,v)$ ist der Wert von f.
  Problem: Maximaler Fluss $|f|$.

- Immer ist $f(u,u) = -f(u,u) = 0$, da $K(u,u) = 0$, da nie $(u,u) \in E$.
- Auch $f(u,v) = f(v,u) = 0$, wenn $(u,v), (v,u) \notin E$. Denn es ist $f(u,v) = -f(v,u)$, also ein Wert > 0.
- Ist $f(u,v) \neq 0$, so u○⟶○v $\in E$ oder v○⟶○u $\in E$.
- Nicht sein kann 

wegen $f(u,v) = -f(v,u)$. Hier würden wir setzen $f(u,v) = f(v,0) = 0$.

Wir können (müssen) kürzen, aber



geht.

Definition 9.3:

Gegeben ist das Flussnetzwerk $G = (V, E)$ mit Kapazität $K : V \times V \longrightarrow \mathbb{R}^{\geq 0}$ und ein zulässiger Fluss $f : V \times V \longrightarrow \mathbb{R}$.

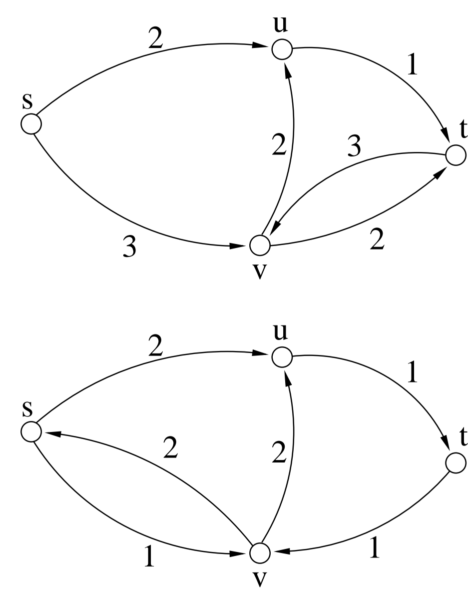
- Das Restnetzwerk G_f mit Restkapazität K_f ist gegeben durch:
  $K_f = (u,v) = K(u,v) - f(u,v) \geq 0, E_f = \{(u,v) | K_f(u,v) > 0\}$. (Es ist $K_f(u,v) \geq 0$, da $f(u,v) \leq K(u,v)$)
  $G_F = (V, E_f)$
- Ein Erweiterungspfad von G und f ist ein einfacher Weg in G_f
  $W = $ s = v_0 ⟶ v_1 ⟶ v_2 ... ⟶ v_k= t
  Die Restkapazität von W ist $K_f(W) = Min\{K_f(v_i, v_{i+1})\}$ (Nach Definition gilt $K_f(v_i, v_{i+1}) > 0$))

Es ist G_f ein Flussnetzwerk und es ist $g : V \times V \longrightarrow \mathbb{R}$ mit $g(v_i, v_{i+1}) = K_f(W) > 0$, $g(v_{i+1}, v_i) = -K_f(W) = -G(v_i, v_{i+1})$, $g(u,v) = 0$ für $(u,v) \notin W$, ein zulässiger Fluss auf $G_f \cdot |g| = K_f(W)$.
Tatsächlich gilt nun sogar:

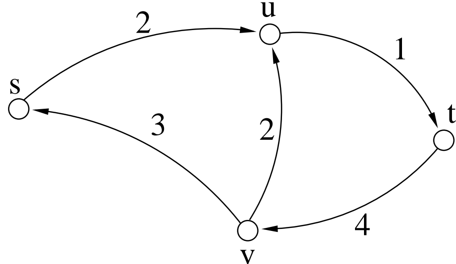
Lemma 9.4:

Sei G, K, f Flussnetzwerk mit zulässigem Fluss f. Sei G_f das Restnetzwerk. Sei g irgendein zulässiger Fluss auf G_f. Dann ist $f + g$ Fluss auf $G, |f + g| = |f| + |g|$.

Restnetzwerke zu vorangegangenem Netzwerk



Kapazitäten immer ≥ 0



kein Erweiterungspfad.

Beweis 9.5:

Wir müssen also überlegen, dass $f+g$ zulässiger Fluss von G ist. Kapazitätsbedingung: s ist $g(u,v) \leq K_f(u,v) = K(u,v) - f(u,v)$.
Also $(f+g)(u,v) = f(u,v) + g(u,v) \leq f(u,v) + K(u,v) - f(u,v) = K(u,v)$
Symmetrie
$(f+g)(u,v) = f(u,v) + g(u,v) = -f(v,u) - g(v,u) = -(f+g)(v,u)$ Kirchhoff: Sei $u \in V \setminus \{s,t\}$, dann $\sum\limits_{v \in V} (f+g)(u,v) = \sum\limits_{v \in V} (f(u,v) + g(u,v))$
$= \sum\limits_{v \in V} f(u,v) + \sum\limits_{u \in V} g(u,v) = 0.$
Schließlich ist $|f+g| = \sum\limits_{v \in V} (f+g)(s,v) = |f| + |g|$.

Algorithmus (Ford Fulkerson)

1. $f(u,v) = 0$ für alle $u, v \in V$
2. while Es gibt Weg s ○———► ○ t in G_f{
3. W = ein Erweiterungspfad in G_f
4. g = Fluss in G_f mit $g(u,v) = K_f(W)$ für alle u○———►○v $\in W$, wie oben
5. $f = f + g$ }

Gib f als maximalen Fluss aus.

Definition 9.6:

Ein Schnitt eines Flussnetzwerkes $G = (V,E)$ ist eine Partition S, T von V, d.h. $V = S \cup T$ mit $s \in S$ und $t \in T$.

- Kapazität von S,T, $K(S,T) = \sum\limits_{u \in S, v \in T} K(u,v)$ mit $K(u,v) \geq 0, u \in S, v \in T$
- Ist f ein Fluss, so ist $f(S,T) = \sum\limits_{u \in S, v \in T} f(u,v)$.

Immer ist $f(S,T) \leq K(S,T)$ und $|f| = f(\{s\}, V \setminus \{s\})$.
Ein Flussnetzwerk

s —1/1→ v_1 —1/2→ v_2 —1/3→ v_3 —1/1→ v_4 —1/2→ t

S = { s, v_2, v_4}

T = { v_1, v_3, t }

K(S, T) = 1 + 3 + 2 = 6

s → v_1 v_2 → v_3 v_4 → t

$F(S,T) = 1 - 1 + 1 - 1 + 1 = 1$ (Betrag des Flusses)
S,T in mehreren Stücken $\longrightarrow$ kein minimaler Schnitt.
Tatsächlich gilt sogar für jeden Schnitt S,T, dass $f(S,T) = |f|$
Induktion über $|S|$. $|S| = 1$, dann $S = \{s\}$, dann gilt es. Sei $|S| = l+1, v \in S \setminus \{s\}$, sei $S' = S \setminus \{v\}, T' = \cup\{v\}$. Dann $f(S,T) = f(S',T') + \sum\limits_{u \in S} f(u,v) + \sum\limits_{u \in T} f(v,u) = |f| + \sum\limits_{u \in V} f(v,u) = |f|$.
Also: Für jeden Schnitt $|f| \leq K(S,T)$.
Also auch $Max\{|f||fFluss\} \leq Min\{K(S,T)|S,TSchnitt\}$ Es gibt einen Fluss f^*, der diese obere Schranke erreicht, $f^* = Min\{K(S,T)|S,TSchnitt\}$:

Satz 9.7 (*Min-Cut-Max-Flow*):

Ist f ein zulässiger Fluss in G. Äquivalent sind (1), (2) und (3).
(1) f ist maximaler Fluss.
(2) G_f hat keinen Erweiterungspfad.
(3) Es gibt einen Schnitt S,T, so dass $|f| = K(S,T)$.
(Immer $f(S,T) = |f|, K(S,T) \geq |f|$)

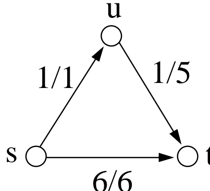
Beweis 9.8:

(1) $\longrightarrow$ (2) gilt, da f maximal. Gälte (2) nicht, dann gälte auch (1) nicht. (2) $\longrightarrow$ (3) Setze $S = \{u \in V|$ Es gibt Weg (s, u) in $G_f\}$
$T = V \setminus S$
Dann $s \in S$ und $t \in T$, da kein Erweiterungspfad nach (2). Also S,T ist ordentlicher Schnitt. Für $u \in S, v \in T$ gilt:
$O = K_f(u,v) = K(u,v) - f(u,v)$
Also $K(u,v) = f(u,v)$.
Dann $|f| = f(S,T) = \sum_(u \in S, v \in T)f(u,v) = \sum_(u \in S, v \in T)K(u,v) = K(S,T)$.
(3) $\longrightarrow$ (1) gilt, da immer $|f| \leq K(A,B)$.

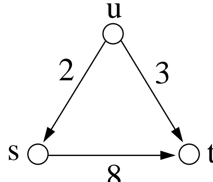
Korrektheit von Ford-Fulkerson:
Invariante:
f_l ist zulässiger Fluss Quintessenz: Am Ende ist f_l maximaler Fluss (Mincut-max-flow).
Termination: $f_l(S,T)$ erhöht sich jedesmal.

Laufzeit von Ford Fulkerson?
Bei ganzzahligen Kapazitäten reicht $O(|E| \cdot |f^*|)$, wobei f^* ein maximaler Fluss ist. Rationale Zahlen: Normieren!

Min. Schnitt = Max. Fluss



$S = \{s, u\} T = \{t\}$
$K(S, T) = f(S, T) = 7$



$S = \{s\}, T = \{u, t\}, K(S, T) = 8$
$S = \{s, u\}, T = \{t\}, K(S, T) = 11$
$|f| \leq 8$. Vergleiche hierzu Seite 110

10 Kombinatorische Suche und Rekursionsgleichungen

Bisher meistens Probleme in polynomieller Zeit (und damit auch Platz). Obwohl es exponentiell viele mögliche Wege von $u \circ\!\!\longrightarrow\!\circ\, v$ gibt, gelingt es mit Dijkstra oder Floyd-Warshall, einen kürzesten Weg systematisch aufzubauen, ohne alles zu durchsuchen. Das liegt daran, dass man die richtige Wahl lokal erkennen kann. Bei Ford-Fulkerson haben wir prinzipiell unendlich viele Flüsse, trotzdem können wir einen maximalen systematisch aufbauen, ohne blind durchzuprobieren. Dadurch erreichen wir polynomiale Zeit.
Jetzt: Probleme, bei denen man die Lösung nicht mehr zielgeirchtet aufbauen kann, sondern im Wesentlichen exponentiell viele Lösungskandidaten durchsuchen muss. (Das bezeichnet man als kombinatorische Suche.)

Zunächst: Aussagenlogische Probleme.

Aussagenlogische Probleme, wie zum Beispiel $x \wedge y \vee (\neg((u \wedge \neg(v \wedge \neg x)) \rightarrow y), x \vee y, x \wedge y, (x \vee y) \wedge (x \vee \neg y)$.
Also wir haben eine Menge von aussagenlogischen Variablen zur Verfügung, wie x, y, v, u, ...
Formeln werden mittels der üblichen aussagenlogischen Operationen $\wedge$ (und), v (oder, lat. vel), $\neg, ^-$ (nicht), $\Rightarrow$ (Implikation), $\Leftrightarrow$ (Äquivalenz) aufgebaut.

Variablen stehen für die Wahrheitswerte 1 (= wahr) und 0 (= falsch). Die Implikation hat folgende Bedeutung:

x	y	x ⇒ y
0	0	1
0	1	1
1	0	0
1	1	1

D.h., das Ergebnis ist nur dann falsch, wenn aus Wahrem Falsches folgen soll.
Die Äquivalenz $x \Rightarrow y$ ist genau dann wahr, wenn x und y beide den gleichen Wahrheitswert haben, also beide gleich 0 oder gleich 1 sind.
Damit ist $x \Leftrightarrow y$ gleichbedeutend zu $(x \Rightarrow y) \wedge (y \Rightarrow x)$.
Ein Beispiel verdeutlicht die Relevanz der Aussagenlogik:

Worin besteht das Geheimnis Ihres langen Lebens? Folgender Diätplan wird eingehalten:

- Falls es kein Bier zum Essen gibt, dann wird in jedem Fall Fisch gegessen.
- Falls aber Fisch, gibt es auch Bier, dann aber keinesfalls Eis
- Falls Eis oder auch kein Bier, dann gibts auch keinen Fisch

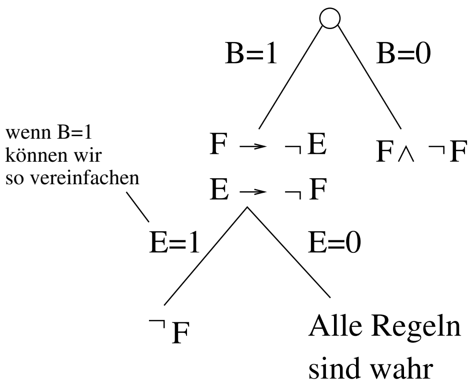
Dazu Aussagenlogik:

B = Bier beim Essen
F = Fisch
E = Eis

Ausagenlogisch werden obige Aussagen nun zu:

$\neg B \Rightarrow F$
$F \wedge B \Rightarrow \neg E$
$E \vee \neg B \Rightarrow \neg F$

Aussagenlogik erlaubt die direkte Darstellung von Wissen (Wissensrepräsentation - ein eigenes Fach). Wir wollen nun feststellen, was verzehrt wird:



Also: Immer Bier und falls Eis, dann kein Fisch. $B \wedge (E \Rightarrow \neg F)$.
Das ist gleichbedeutend zu $B \wedge (\neg E \vee \neg F)$ und gleichbedeutend zu $B \wedge (\neg(E \wedge F))$.

Immer Bier, Eis und Fisch nicht zusammen.

Die Syntax der aussagenlogischen Formeln sollte soweit klar sein. Die Semantik (Bedeutung) kann erst dann erklärt werden, wenn die Variablen einen Wahrscheinlichkeitswert haben, d.h. wir haben eine Abbildung $a : Variablen \rightarrow \{0, 1\}$, d.h. eine Belgung der Variablen mit Wahrheitswerten ist gegeben.

Dann ist $a(F)$ = Wahrheitswert von von F bei Belegung a.
Ist $a(x) = a(y) = a(z) = 1$, dann $a(\neg x \vee \neg y) = 0, a(\neg x \vee \neg y \vee z) = 1, a(\neg x \wedge (y \vee z)) = 0$.
Für eine Formel F der Art $F = G \wedge \neg G$ gilt $a(F) = 0$ für jedes a, für $F = G \vee \neg G$ ist $a(F) = 1$ für jedes a.
Einige Bezeichnungen:

- F ist erfüllbar, genau dann, wenn es eine Belgung a mit $a(F) = 1$ gibt.
- F ist unerfüllbar (widersprüchlich), genau dann wenn für alle a $a(F) = 0$ ist.
- F ist tautologisch, genau dann, wenn für alle a $a(F) = 1$ ist.

Beachte: Ist F nicht tautologisch, so heißt das im Allgemeinen nicht, dass F unerfüllbar ist.

Algorithmus (Erfüllbarkeitsproblem)

Eingabe. F

1. Erzeuge hintereinander alle Belegungen $a(0\dots 0, 0\dots 1.0\dots 10, \dots)$. Ermittle $a(F)$. Ist $a(F) = 1$, return „F erfüllbar duch a."
2. return „F unerfüllbar."

Laufzeit:

Bei n Variablen $O(2^n \cdot |F|)$, wobei $|F| =$ Größe von F ist.

Dabei muss F in einer geeigneten Datenstruktur vorliegen. Wenn F erfüllbar ist, kann die Zeit wesentlich geringer sein! Reiner worst-case. Verbesserung durch backtracking:

Schrittweises Einsetzen der Belegung und Vereinfachen von F (Davis-Putman-Prozedur, siehe später).
Die Semantik einer Formel F mit n Variablen lässt sich auch verstehen als eine Funktion $F : \{0,1\}^n \to \{0,1\}$.

Frage: Wieviele derartigen Funktionen gibt es?

Jede derartige Funktion lässt sich in konjunktiver Normalform (KNF) darstellen. Auch in disjunktiver Normalform.

Konjunktive Normalform ist:
$x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee x_2)$
Disjunktive Normalform ist:
$(x_1 \wedge x_2 \wedge \neg x_3 \wedge x_4) \vee (x_2 \wedge x_3 \wedge \neg x_4) \wedge (\neg x_1 \wedge x_5 \wedge \dots \wedge x_n) \wedge \dots$

Im Prinzip reichen DNF oder KNF aus. Das heißt, ist $F : \{0,1\}^n \to \{0,1\}$ eine boolsche Funktion, so lässt dich F als KNF oder auch DNF darstellen.
KNF: Wir gehen alle $(b_1, \dots, b_n) \in \{0,1\}^n$, für die $F(b_1, \dots, b_n) = 0$ ist, durch.
Wir schreiben Klauseln nach folgendem Prinzip:

$$F(0-0) = 0 \to x_1 \vee x_2 \vee \dots \vee x_n$$
$$F(110-0) = 0 \to \neg x_1 \vee \neg x_2 \vee \dots \vee x_n$$

Die Konjunktion dieser Klauseln gibt die Formel, die F darstellt.
DNF: Alle $(b_1, \dots, b_n) \in \{0,1\}^n$ für die $F(b_1, \dots, b_n) = 1$ Klauseln analog oben:

$$F(0,0,\dots,0) = 1 \to \neg x_1 \wedge \neg x_2 \wedge \dots \wedge x_n$$
$$F(1,1,0\dots,0) = 1 \to x_1 \wedge x_2 \wedge \neg x_3 \wedge \dots \neg x_n$$

$\vdots$

Beachte:

Erfüllbarkeitsproblem bei KNF $\iff$ Jede (!) Klausel muss ein wahres Literal haben.
Erfüllbarkeitsproblem bei DNF $\iff$ Es gibt eine (!) Klausel, die wahr gemacht werden kann.
Erfüllbarkeitsproblem bei DNF leicht: Gibt es eine Klausel, die nicht x und $\neg x$ enthält.

Schwierig bei DNF:

Gibt es eine Belegung, so dass 0 rauskommt. Das ist nun wieder leicht bei KNF (Wieso?). Eine weitere Einschränkung ist die k-KNF, k-DNF, $k = 1, 2, 3 \cdots$: Klauselgröße (= # Literale) $\leq k$ pro Klausel.
1-KNF: $x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \ldots$
2-KNF: $(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_1 \vee \neg x_1)$ $\quad x_1 \vee \neg x_1$ - tautologische Klausel, immer wahr, eigentlich unnötig
3-KNF: $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_3) \wedge x_1 \wedge \ldots$
Eine unerfüllbare 1-KNF ist $x_1 \wedge \neg x_1$,
eine unerfüllbare 2-KNF $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$

Interessant ist, dass sich unerfüllbare Formeln auch durch geeignete Darstellung mathematischer Aussagen ergeben können. Betrachten wir den Satz:

Ist $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$ injektive Abbildung, dann auch surjektiv. Dazu nehmen wir n^2 viele Variablen. Diese stellen wir uns so vor:

$x_{1,1}, x_{1,2}, x_{1,3}, \ldots x_{1,n}$
$x_{2,1}, \ldots$
$\vdots$
$\ldots, x_{n,n}$

Jede Belegung a der Variablen entspricht einer Menge von Paaren M :
$a(x_{i,j}) = 1 \iff (i, j) \in M$.
Eine Abbilding ist eine spezielle Menge von Paaren. Wir bekommen eine widersprüchliche Formel nach folgendem Prinzip:

1. a stellt eine Abbildung dar
   $\wedge$
2. a ist eine injektive Abbildung
   $\wedge$
3. a ist nicht surjektive Abbildung

$$\left.\begin{array}{l}(x_{1,1} \vee x_{1,2} \vee \cdots \vee x_{1,n}) \\ \wedge \\ (x_{2,1} \vee x_{2,2} \vee \cdots \vee x_{2,n}) \\ \wedge \\ \ldots \\ \wedge \\ (x_{n,1} \vee x_{n,2} \vee \cdots \vee x_{n,n})\end{array}\right\} \begin{array}{l}\text{Jedem Element aus } 1, \cdot, n \\ \text{ist eines zugeordnet,} \\ n \text{ Klauseln}\end{array}$$

$$\underbrace{(\neg x_{1,1} \vee \neg x_{1,2}) \wedge (\neg x_{1,1} \vee x_{1,3}) \wedge \cdots \wedge (\neg x_{1,n-1} \vee \neg x_{1,n})}_{\text{1 ist höchstens 1 Element zugeordnet, } \binom{n}{k} \text{ Klauseln}}$$

Ebenso für $2, \ldots, n$.
Damit haben wir gezeigt, dass a eine Abbildung ist.

$$\underbrace{(\neg x_{1,1} \vee \neg x_{2,1}) \wedge (\neg x_{1,1} \vee x_{3,1}) \wedge (\neg x_{1,1} \vee \neg x_{4,1}) \wedge \cdots \wedge (\neg x_{1,1} \vee \neg x_{n,1})}_{\text{1 wird höchstens von einem Element getroffen.}}$$

$\vdots$
Ebenso für $2, \ldots, n$.
Haben jetzt: a injektive Abbildung

$$\left.\begin{array}{l} \wedge \\ ((\neg x_{1,1} \wedge x_{2,1} \wedge x_{3,1} \wedge \cdots \wedge \neg x_{n,1}) \\ \vee \\ (\neg x_{1,2} \wedge \neg x_{2,2} \ldots) \\ \vdots \\ \vee \\ (\neg x_{1,n} \wedge \cdots \wedge \neg x_{n,n}) \end{array}\right\} a \text{ nicht surjektiv}$$

Im Falle des Erfüllungsproblems für KNF lässt sich der Backtracking-Algorithmus etwas verbessern. Dazu eine Bezeichnung: $F_{x=1}$ = x auf 1 setzen und F vereinfachen.
= Klauseln mit x löschen (da wahr); in Klauseln mit $\neg x$ das $\neg x$ löschen (da es falsch ist)
Analog $F_{x=0}$.
Es gilt F erfüllbar $\iff$ $F_{x=1}$ oder $F_{x=0}$ erfüllbar.
Der Backtracking-Algorithmus für KNF führt zur *Davis-Putman-Prozedur*, die so aussieht:

Algorithmus (Davis-Putman)

Eingabe: F in KNF, Variablen $x_1, \ldots x_n$
Array $a[1, \ldots, n]$ of boolean //Für die Belegung

```
DP(F){
1. if F offensichtlich wahr            // leere Formel
      return „F erfüllbar„             // ohne Klausel
2. if F offensichtlich unerfüllbar
      return „unerfüllbar„             // Enthält leere Klausel
                                       oder Klausel x und ¬ x
3. Wähle eine Variable x von F gemäß einer Heuristik.
   Wähle (b1, b2) mit b1, b2 = 0 oder b1=0, b2=1
4. H:=F—x=b1; a[x]:=b1;
   if (DP(H) == „H) erfüllbar„
```

return „F erfüllbar„
5. H:=F—x=b2; a[x]:=b2
// Nur, wenn in 4. „unerfüllbar„
return DP(H)}

Ist DP(F) = „F erfüllbar", so ist in a die gefundene erfüllende Belegung.
Korrektheit:
Induktiv über die #Variablen in F, wobei das a noch einzubauen ist.

Algorithmus (Pure literal rule, unit clause rule)

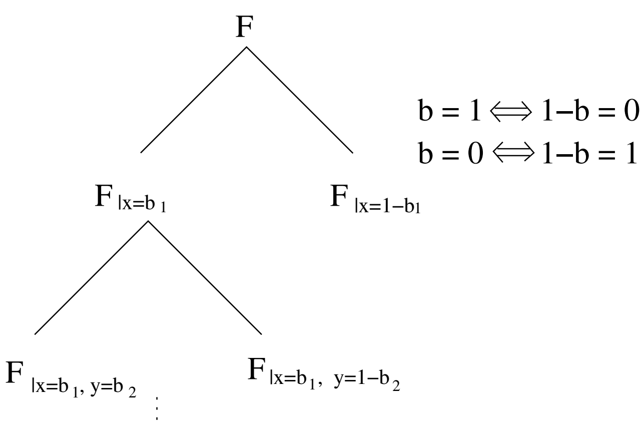
In die einfache Davis-Putman-Prozedur wurden noch folgedne Heuristiken in 3. eingebaut:

- Pure literal rule (x ist ein „pures Literal")
  if es gibt ein x in F, so dass $\neg x$ nicht in $F\{$
  $H := F|x = 1, a[x] := 1$; return DP(H);}
  if $\neg x$ in F aber x nicht in $f\{$
  $H := F|x = 0; a[x] = 0$; return DP(H); // Hier kein Backtracking
- Unit clause rule
  if es gibt eine Einerklausel (x) in $F\{$
  $H := F \ll x = 1; a[x] := 1$; return DP(H);}
  if $(\neg x)$ in $F\{$
  $H := F|x = 0; a[x] = 0$; return DP(H);}// kein Backtracking

Erst danach geht das normale 3. des Davis-Putman-Algorithmus weiter.
Korrektheit:

- Pure literal: Es ist $F_{|x=1} \subseteq F$, heißt jede Klausel in $F_{x=1}$ tritt auch in F auf. Damit gilt:
  $F_{x=1}$ erfüllbar $\Longleftarrow F$ erfüllbar
  $F_{x=1}$ unerfüllbar $\Longleftarrow F$ unerfüllbar
  (wegen $F_{x=1} \subseteq F$)
  Also $F_{x=1}$ erfüllbar $\Longleftrightarrow F$ erfüllbar, backtracking nicht nötig.
- Unit clause: $F_{x=0}$ ist offensichtlich unerfüllbar,1 wegen leerer Klausel, also kein backtracking nötig.

Laufzeit: Prozedurbaum



Sei T(n) = maximale #Blätter bei F mit n Variablen, dann gilt:

$T(n) \leq T(n-1) + T(n-1)$ für $n > 1$
$T(1) = 2.$

Dann das „neue T(n)“($\geq$ „altes T(n)“)

$T(n) = T(n-1) + T(n-1)$ für $n > 1$
$T(1) = 2$

Hier sieht man direkt $T(n) = 2^n$.

Eine allgemeine Methode läuft so:
Machen wir den Ansatz (=eine Annahme), dass $T(n) = 2^n$ für ein $\alpha \geq 1$ ist. Dann muss für $n > 1$ $\alpha^n = \alpha^{n-1} + \alpha^{n-1} = 2\alpha^{n-1}$ sein. Also teilen wir durch α^{n-1} :

$\alpha = 1 + 1 = 2.$
Muss durch Induktion verifiziert werden, da der Ansatz nicht stimmen muss.

Damit Laufzeit:
$O((2^n - 1 + 2^n) \cdot |F|) = O(2^n \cdot |F|)$, $F \ldots$ Zeit fürs Einsetzen

Also: Obwohl backtracking ganze Stücke des Lösungsraums, also der Menge $\{0, 1\}^n$ rausschneidet, können wir zunächst nichts Besseres als beim einfachen Durchgehen aller Belegungen zeigen.
Bei k-KNF's für ein festes k lässt sich der Davis-Putman-Ansatz verbessern. Interessant ist, dass in gewissem Sinne k=3 reicht (konkret beim Erfüllbarkeitsproblem). Damit Laufzeit:

Satz 10.1:

Sei F eine beliebige aussagenlogische Formel. Wir können zu F eine 3-KNF G konstruieren mit:

- F erfüllbar $\Longleftrightarrow$ G erfüllbar (F, G erfüllbarkeitsäquivalent)
- Die Konstruktion lässt sich in der Zeit $O(|F|)$ implementieren.
  (Insbesondere $|G| = O(|F|)$)

Beachte:

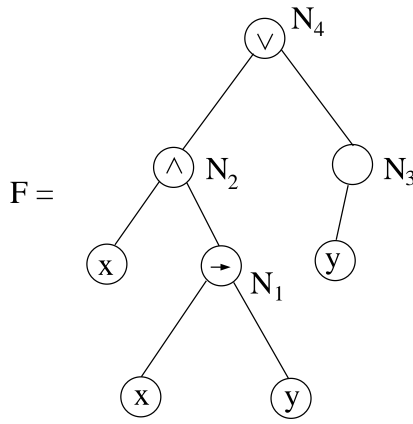
Durch Ausmultiplizieren lässt sich jedes F in ein äquivalentes F in KNF transformieren. Aber

- Exponentielle Vergrößerung ist möglich.
- keine 3-KNF

Beweis 10.2 (*des Satzes*):

Am Beispiel wird eigentlich alles klar. Der Trick besteht in der Einfürhung neuer Variablen!
$F = (x \wedge (x \Leftarrow y)) \vee \neg y)$

Schreiben F als Baum: Variablen = Blätter, Operationen = innere Knoten



N_1, N_2, N_3, N_4 : neue Variablen, für jeden inneren Knoten eine neue Variable

Idee: N_i = Wert an den Knoten, bei gegebener Belegung der alten Variablen x, y.

Das drückt F' aus:
$F' = (N_1 \Leftrightarrow (x \Rightarrow y)) \wedge (N_2 \Leftrightarrow (x \wedge N_1)) \wedge (N_3 \Leftrightarrow \neg y) \wedge (H_4 \Leftrightarrow (N_3 \vee N_2))$ Es ist F' immer erfüllbar. Brauchen nur die N_i von unten noch eben zu setzen.

Aber:
$F - F' \wedge N_4$, $N_4 \ldots$ Variable der Wurzel
erfordert, dass alle $N - i$ richtig und $N_4 = 1$ ist. Es gilt: Ist $a(F) = 1$, so können wir eine Belegung b konstruieren, in der die $b(N - i)$ richtig stehen, so dass $b(F'') = 1$ ist. Ist andererseits $b(F'') = 1$, so $b(N_4) = 1$ und eine kleine Überlegung zeigt uns, dass die b(x), b(x) eine erfüllende Belegug von F sind.
3-KNF ist gemäß Prinzip auf Seite 116 zu bekommen. Anwendung auf die Äquivalenzen.

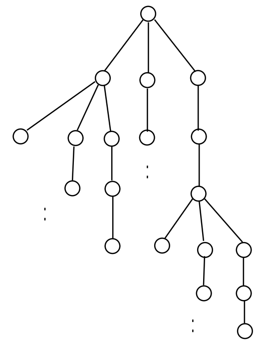
Frage: Warum kann man so keine 2-KNF bekommen?

Davis-Putman basiert auf dem Prinzip F erfüllbar $\Longleftrightarrow$ $F_{|x=1} oder F_{x=0}$ erfüllbar. (Verzweigen nach dem Wert von x) Haben wir eine 3-KNF, etwa $F = \cdots \wedge (x \vee \neg y \vee z) \wedge \ldots$, dann ist F erfüllbar, gdw. $F_{|x=1}$ oder $F_{|y=0}$ oder $F_{z=1}$ erfüllbar (Verzweigen nach der Klausel).

Ein backtracking-Algorithmus nach diesem Muster führt zu einer Aufrufstruktur der Art

Tiefe n
$\Rightarrow$ 3^n Blätter (?!)

Aber es gilt auch: F erfüllbar $\Longleftrightarrow$ $F_{|x=1}$ erfüllbar oder $F_{|x=0,y=0}$ erfüllbar oder $F_{x=0,y=1,z=1}$ erfüllbar.
Damit Aufrufstruktur im backtracking:



Sei wieder $T(n) :=$ maximale # Blätter bei b Variablen, dann gilt:

$T(n) \geq T(n-1) + (T(n-1) + T(n-3)$ für $n \geq n_0, n_o$ eine Konstante $T(n) \leq d = O(1)$ für $n < n_0$. Was ist das?

Lösen $T(n) = T(n-1) + T(n-2) + T(n-3)$.
Ansatz: $T(n) = c \cdot \alpha^n$ für alle n groß genug. Dabb $c \cdot \alpha^n = c \cdot^{n-1} + c \cdot \alpha^{n-2} + c \cdot \alpha^{n-3}$, also$\alpha^3 = \alpha^2 + \alpha + 1$.
Wir zeigen: Für $\alpha > 0$ mit$\alpha^3 = \alpha^2 + \alpha + 1$ gilt $T(n) = O(\alpha^n)$.

Ist $n > n_0$, dann $T(n) \leq d \cdot \alpha^n$ für geeignete Konstante d, da $\alpha > 0$! Sei $n \geq n_0$.

Dann:
$T(n) = T(n-1) + T(n-2) + T(n-3) \leq d \cdot \alpha^{n-1} + d \cdot \alpha n - 2 + d \cdot \alpha^{n-3}$

$= d(\alpha^{n-3} \cdot \overbrace{(\alpha^2 + \alpha + 1)}^{=\alpha \text{ nach Wahl} \alpha})$
$= d \cdot \alpha^n = O(\alpha^n)$.

Genauso $T(n) = \Omega(\alpha^n)$. Da sie angegebene Argumentation für jedes(!) $\alpha > 0$ mit $\alpha^3 = \alpha^2 + \alpha + 1$ gilt, darf es nur ein solches α geben! Es gibt ein solches $\alpha < 1.8393$.
Dann ergibt sich, dass die Laufzeit $O(|F| \cdot 1.8393^n)$ ist, da die inneren Knoten durch den konstanten Faktor mit erfasst werden können.

Frage: Wie genau geht das?

Bevor wir weiter verbessern, diskutieren wir, was derartige Verbesserungen bringen. Zunächst ist der exponentielle Teil maßgeblich (zumindest für die Asymptotik (n groß) der Theorie):
Es ist für $c > 1$ konstant und $\epsilon > 0$ konstant (klein) und k konstant (groß) $n^k \cdot c^n \leq^{(1+\epsilon)n}$, sofern n groß genug ist, denn $c^{\epsilon \cdot n} \geq c^{log_c n \cdot k} = n^k$ für $\epsilon > 0$ konstant und n groß genug (früher schon gezeigt).

Haben wir jetzt zwei Algorithmen

- A_1 Zeit 2^n
- A_2 Zeit $2^{\frac{1}{2} \cdot n} = (\sqrt{2})^n = (1.4\dots)^n$

für dasselbe Problem. Betrachten wir eine vorgegebene Zeit x (fest). Mit A_1 können wir alle Eingaben der Größe n mit $2^n \leq x$, d.h. $n \leq log_2 x$ in Zeit x sicher bearbeiten. Mit A_2 dagegen alle mit $2^{\frac{1}{2}\cdot n} \leq x$, d.h. $n \leq 2 \cdot log_2$, also Vergrößerung um einen konstanten Faktor! Lassen wir dagegen A_1 auf einem neuen Rechner laufen, auf dem jede Instruktion 10-mal so schnell abläuft, so $\frac{1}{10} \cdot 2^n \leq x$, d.h. $n \leq log_2 10x + log_2 10$ nur die Addition einer Konstanten.

Fazit:

Bei exponentiellen Algorithmen schlägt eine Vergrößerung der Rechengeschwindigkeit weniger durch als eine Verbesserung dr Konstante c in c^n der Laufzeit.

Aufgabe:

Führen Sie die Betrachtung des schnelleren Rechners für polynomielle Laufzeiten n^k durch. Haben wir das literal x pur in F (d.h. $\bar{a}$ ist nicht dabei), so reicht es, $F_{|z=1}$ zu betrachten. Analoges gilt, wenn wir mehrere Variablen gleichzeitig ersetzen. Betrachten wir die Variablen $x_1, \ldots, x_k$ und Wahrheitswerte $b_1, \ldots, b_k$ für diese Variablen und gilt $H = F_{|x_1=b1,x_2=b_2,\ldots,x=b_k} \subseteq F$, d.h. jede Klausel C von H ist schon in F, d.h., wenn das Setzen von einem $x_i = b_j$ wahr wird, so gilt

$$F \text{ erfüllbar} \Longleftrightarrow H \text{ erfüllbar}$$

wie beim Korrektheitsbeweis der *pure literal rule*. Kein backtracking nötig!
Wir sagen, die Belegung $x_1 = b_1, \ldots, x_k = b_k$ ist autark für F. Etwa $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee x_4) \wedge G, G$ ohne $x_3, x_1, \bar{x}_2$.
Dann $F_{|x_1=0,x_2=0,x_3=0} \subseteq F$, also $x_1 = 0, x_2 = 0, x_3 = 0$ autark für F. (In G fallen höchstens Klauseln weg.)

Algorithmus (Monien, Speckenmeyer)

```
1. if F offensichtlich wahr return „erfüllt„
2. if F offensichtlich unerfüllbar return „unerfüllbar„
3. Wähle eine kleinste Klausel C,
   C=l_1 ∨ ··· ∨ l_i in F                     //l_i Literal, x oder ¬ x
4. Betrachte die Belegungen l_1 = 1, l_1 = 0, l_2 = 1, ... l_1 = 0, l_2 = 0, ..., l_s = 1
5. if (eine der Belegungen autark in F){
      b := eine autarke Belegung; return MoSP(F_|b)}
6. Teste = MoSP(F_i) für i=1, ..., s          // Hier ist keine der
   und F_i jeweils durch Setzen einer der     // Belegungen autark
   obigen Belegungen; return „erfüllbar„, wenn ein Aufruf
   „erfüllbar„ ergibt, „unerfüllbar„ sonst.
```

Wir beschränken uns bei der Analyse der Laufzeit auf den 3-KNF-Fall. Sei wieder $T(n) = \#$ Blätter bei kleinster Klausel mit 3 Literalen und $T'(n) = \#$ Blätter bei kleinster Klausel mit ≤ 2 Literalen.

Für $T(n)$ gilt:

$T(n) \leq T(n-1)$ (Autarke Belegung gefunden, minestens 1 Variable weniger)
$T(n) \leq T'(n-1)+T'(n-2)+T'(n-3)$ (Keine autarke Belegung gefunden, dann aber Klauseln der Größe ≤ 2, Algorithmus nimmt immer die kleinste Klausel)

Ebenso bekommen wir $T(n) \leq T(n-1)$
$T'(n) \leq T'(n-1) + T'(n-2)$ (Haben $\leq$ 2er Klausel ohne autarke Belegung)

Zewcks Verschwinden des $\leq$-Zeichens neue $T(n), T'(n) \leq$ die alten $T(n), T'(n)$.

$T(n) = d$ für $n \leq n_0, d$ Konstante
$T(n) = \text{Max}\{T(n-1), T'(n-1) + T'(n-2) + T'(n-3)\}$ für $n > n_0$
$T'(n) = d$ für $n \leq n_0, d$ Konstante
$T'(n) = \text{Max}\{T(n-1), T'(n-1) + T'(n-2)\}$

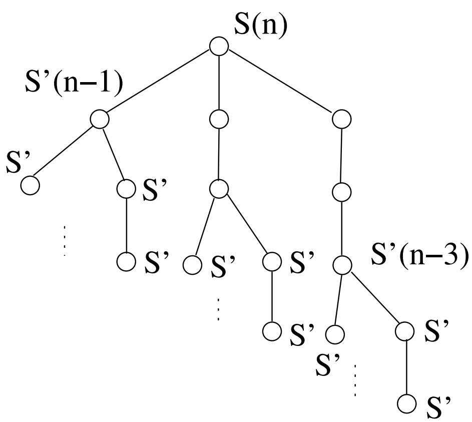
Zunächst müssen wir das Maximum loswerden. Dazu zeigen wir, dass für $S(n) = d$ für $n \leq n_0$

$S(n) = S'(n-1) + S'(n-2) + S'(n-3)$ für $n > n_0, S'(n) = d$ für $n \leq n_0$
$S'(n) = S'(n-1) + S'(n-2)$ für $n > n_0$ gilt, dass $T(n) \leq S(n), T'(n) \leq S'(n)$.

Ind.-Anfang ✓
Induktionsschluss

$T(n) = \text{Max}\{T'(n-1) + T'(n-2) + T'(n-3)\} \leq$ Ind.-Voraussetzung
$\text{Max}\{S(n-1), S'(n-1) + S'(n-2) + S'(n-3)\}$
$\leq$ Monotonie
$S'(n-1) + S'(n-2) + S'(n-3)\} = S(n)$.
$T'(n) = \text{Max}\{T(n-1), T'(n-1) + T'(n-2)$
$\leq$ Ind.-Voraussetzung
$\text{Max}\{S(n-1), S'(n-1) + S'(n-2)\}$
$= \text{Max}\{\overbrace{S'(n-2) + S'(n-3)}^{=S'(n-1)} + S'(n-4), S'(n-1) + S'(n-2)\}$
$=$ Monotonie
$S'(n-1) + S'(n-2) = S'(n)$.

Der Rekursionsbaum für $S(n)$ hat nun folgende Struktur:



Ansatz: $S'(n) = \alpha^n, \alpha^{n-1} + \alpha^{n-2} \Longrightarrow \alpha^2 = \alpha + 1$
Sei $\alpha > 0$ Lösung der Gleichung, dann $S'(n) = O(\alpha^n)$.

Ind.-Anfanfang: $n \leq n_0 \vee d_0 d_0 > 0$.
Induktionsschluss:
$S'(n+1) = S'(n) + S'(n-1)$
Induktionsvoraussetzung
$\leq d_0 \cdot \alpha^n + c \cdot \alpha^{n-1}$
Wahl von α
$= c \cdot \alpha^{n-1}(\alpha + 1)$
$= c \cdot \alpha^{n-1} \cdot \alpha^2$
$= c \cdot \alpha^{n+1} = O(\alpha^{n+1})$.

Dann auch $S(n) = O(\alpha^n)$ und Laufzeit von Monien-Speckenmayer ist $O(\alpha^n \cdot |F|)$. Für F in 3-KNF ist $|F| \leq (2n)^3$, also Zeit $O(\alpha^{(1+\epsilon)n})$ für n groß genug. Es ist $\alpha < 1.681$.

Die Rekursionsgleichung von $S'(n)$ ist wichtig.

Definition 10.3:

> Die Zahlen $(F_n)_{n \geq 0}$ mit $F_0 = 1, F_1 = 1, \ldots, F_n = F_{n-1} + F_{n-2}$ für alle $n \geq 2$ heißen Fibonacci-Zahlen.
> Es ist $F_n = O(1.681^n)$.

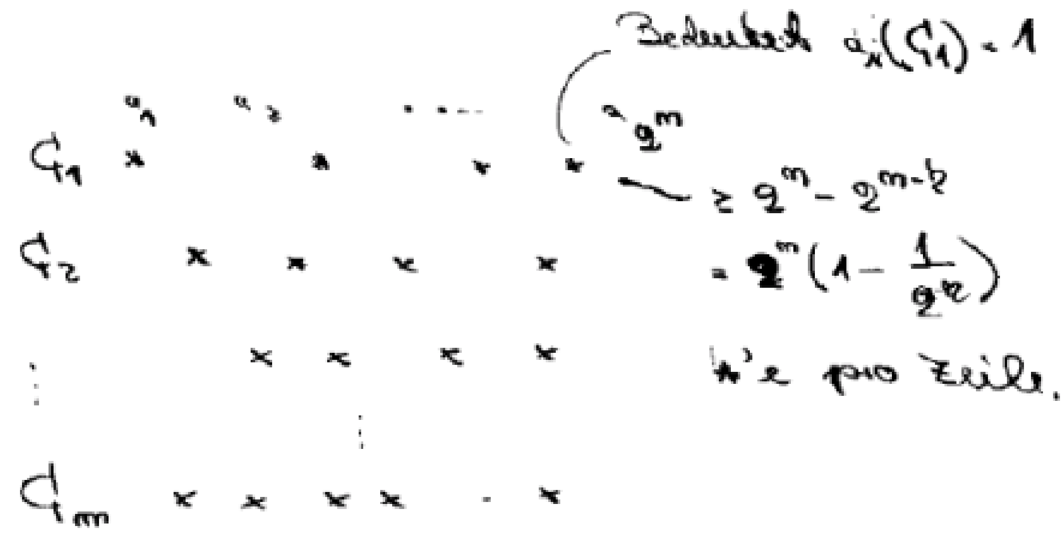
Verwandt mit dem Erfüllbarkeitsproblem ist das Max-Sat- und das Max-k-Sat-Problem. Dort hat man als Eingabe eine Formel $F = C_1 \wedge \cdots \wedge C_m$ in KNF und sucht eine Belegung, so dass $|\{i | a(C) = 1\}|$ maximal ist. Für das Max-Ek-Sat-Problem, d.h. jede Klausel besteht aus k verschiedenen Literalen, hat man folgenden Zusammenhang.

Satz 10.4:

Sei $m = \#F = C_1 \wedge \cdots \wedge C_m$
Formel gemäß Max-Ek-Sat. ($C_i = C_j$ ist zugelassen.) Es gibt eine Belegung a, so dass $|\{j|a(C_j) = 1\}| \geq (1 - \frac{1}{2^k}) \cdot m$.

Beweis 10.5:

Eine feste Klausel $C \subseteq \{C_1, \ldots, C_m\}$ wird von wievielen Belegungen falsch gemacht? $2^{n-k}, n = \#$ Variablen, denn die k Literale der Klausel müssen alle auf 0 stehen. Also haben wir folgende Situation vorliegen:



Zeilenweise Summation ergibt:
$m \cdot 2^n \cdot (1 - \frac{1}{2}) \leq \sum_{j=1}^{m} \{a_i | a_i(C_j) = 1\}| = \#$ der $*$ insgesamt.

Spaltenweise Summation ergibt:
$\sum_{i=1}^{2^n} |\{j|a_i(C_j) = 1\}| = \#$ der $*$ insgesamt, $\geq 2^n \cdot (m \cdot (1 - \frac{1}{2^k}))$

Da wir 2^n Summanden haben, muss es ein a_i geben mit $|\{j|a_i(C_j) = 1\}| \leq m \cdot (1 - \frac{1}{2^k})$.

Wie findet man eine Belegung, so dass $\{j|a(C_j) = 1\}$ maximal ist? Letztlich durch Ausprobieren, sofern $k \geq 2$, also Zeit $O(|F| \cdot 2^n)$. (Für $k = 2$ geht es in $O(c^n)$ für ein $c < 2$.)

Wir haben also 2 ungleiche Brüder:

2-Sat: polynomiale Zeit
Max-2-Sat: nur exponentielle Algorithmen vermutet

Vergleiche bei Graphen mit Kantenlängen ≥ 0 : kürzester Weg polynomial, längster kreisfreier Weg nur exponentiell bekannt.
auch 2-färbbar: poly

3-färbbar: nur exponentiell

Ein ähnliches Phäomen liefert das Problem des maximalen Schnittes:
Für einen Graphen $G = (V, E)$ ist ein Paar (S_1, S_2) mit $(S_1 \cup S_2 = V)$ ein Schnitt $(S_1 \cap S_2 = \emptyset, S_1 \cup S_2 = V)$. Eine Kante $\{v, w\}, v \neq w$,liegt im Schnitt (S_1, S_2), genau dann, wenn $v \in S_1, w \in S_2$ (oder auch umgekehrt).
Es gilt: G 2-färbbar $\iff$ Es gibt einen Schnitt, so dass jede Kante in diesem Schnitt liegt.
Beim Problem des maximalen Schnittes geht es darum, einen Schnitt (S_1, S_2) zu finden, so dass $|\{e \in E | e \text{ liegt in } (S_1, S_2)\}|$ maximal ist. Wie beim Max-Ek-Sat gilt:

Satz 10.6:

> zu $G = (V, E)$ gibt es einen Schnitt (S_1, S_2) so, dass $|\{e \in E | e \text{ in } (S_1, S_2)\}| \geq \frac{|E|}{2}$
> Beweis: Übungsaufgabe.

Algorithmus
(findet den Scnitt, in dem $\geq \frac{|E|}{2}$ Kanten liegen)

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

```
1. for v=1 to n{
2. if Knoten v hat mehr direkte Nachbarn in S_2 als in S_1{
      S_1 = S_1 ∪ {v}; break;
   }
3. S_2 = S_2 ∪ {v}}
```

Laufzeit: $O(n^2), S_1, S_2$ als boolsches Array

Wieviele Kanten liegen im Schnitt (S_1, S_2)?

Invariante:

Nach dem j-ten Lauf gilt: Von den Kanten $v, w \in E$ mit $v, w \in S_{1,j} \cup S_{2,j}$ liegt mindestens die Hälfte im Schnitt $(S_{1,j}, S_{2,j})$. Dann folgt die Korrektheit. So findet man aber nicht immer einen Schnitt mit einer maximalen Anzahl Kanten.

Frage: Beispiel dafür.

Das Problem minimaler Schnitt dagegen fragt nach einem Schnitt (S_1, S_2), so dass $|\{e \in E | e \text{ in } (S_1, S_2)\}|$ minimal ist. Dieses Problem löst Ford-Fulkerson in polynomialer Zeit.

Frage: Wie? $\rightsquigarrow$ Übungsaufgabe.

Für den maximalen Schnitt ist ein solcher Algorithmus nicht bekannt. Wir haben also wieder:

Minimaler Schnitt: polynomial
Maximaler Schnitt: nur exponentiell bekannt

Die eben betrachteten Probleme sind kombinatorische Optimierungsprobleme. Deren bekanntestes ist das Problem des Handlungsreisenden *TSP- Travelling Salesperson*
Definition 10.7 (*TSP, Problem des Handlungsreisenden*):

gegeben: gerichteter, gewichteter Graph
gesucht: eine Rundreise (einfacher Kreis) mit folgenden Kriterien:

- enthält alle Knoten
- Summe der Kosten der betretenen Kanten minimal

Der Graph ist durch eine Distanzmatrix gegeben. ($\infty \Leftrightarrow$ keine Kante). Etwa $M = (M(u,v)), 1 \leq u, v \leq 4$

$$\begin{array}{ccc} & \mathit{1 \leq u,v \leq 4} & \mathit{Zeile} \\ M = & \begin{pmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{pmatrix} & \begin{array}{c} \mathit{1} \\ \mathit{2} \\ \mathit{3} \\ \mathit{4} \end{array} \\ \mathit{Spalte} & \begin{array}{cccc} \mathit{1} & \mathit{2} & \mathit{3} & \mathit{4} \end{array} & \end{array}$$

Die Kosten der Rundreise $R = (1,2,3,4,1)$ sind $K(R) = M(1,2)+M(2,3)+M(3,4)+M(4,1) = 10+9+12+8 = 39$ Für $R' = (1,2,4,3,1)$ ist $K(R') = 35$ Ist das minimal?
1. Versuch:
Knoten 1 festhalten, alle $(n-1)!$ Permutationen auf $2, \dots n$ aufzählen, jeweils Kosten ermitteln.
Zeit:

$$\Omega((n-1)! \cdot n) = \Omega(n!) \geq \{(\frac{n}{2}\})^{(\frac{n}{2})}$$
$$= 2^{((logn)-1)\cdot \frac{n}{2}} = 2^{\frac{(logn)-1}{2} \cdot n} = 2^{\Omega(logn)\cdot n} \gg 2^n$$

Beachte:

$$n^n = 2^{(logn)\cdot n}, \qquad n! \geq 2^{\frac{logn-1}{2}\cdot n}, \qquad n! \ll n^n$$

2. Versuch:

Greedy. Gehe zum jeweils nächsten Knoten, der noch nicht vorkommt (analog Prim). Im Beispiel mit Startknoten 1 $R = (1,2,3,4,1), K(R) = 39$ nicht optimal. Mit Startknoten 4 $R' = (4,2,1,3,4), K(R') = 35$ optimal.
Aber: Greedy geht es zwingend in die Irre.

Greedy	
(2,3,4,1,2)	Kosten = 105
(3,4,1,2,3)	Kosten = 105
(4,3,2,1,4)	Kosten = 105
(1,4,3,2,1)	Kosten = 105

Immer ist $1 \circ\!\!\longleftrightarrow\!\!\circ\, 2$ dabei. Optimal ist (1,4,2,3,1) mit Kosten von 14. Hier ist $4 \circ\!\!\longrightarrow\!\!\circ\, 2$ nicht greedy, sondern mit Voraussicht(!) gewählt. Tatsächlich sind für TSP nur Exponentialzeitalgorithmen bekannt.
Weiter führt unser backtracking. Wir verzweigen nach dem Vorkommen einer Kante in der Rundreise. Das ist korrekt, denn: Entweder eine optimale Rundreise enthält diese Kante $4 \circ\!\!\longrightarrow\!\!\circ\, 2$ oder eben nicht.

Gehen wir auf unser Beispiel von Seite 127 Wir haben die aktuelle Matrix

$$M = \begin{pmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{pmatrix}$$

Wählen wir jetzt zum Beispiel die Kante $2 \circ\!\!\longrightarrow\!\!\circ\, 3$, dann gilt:

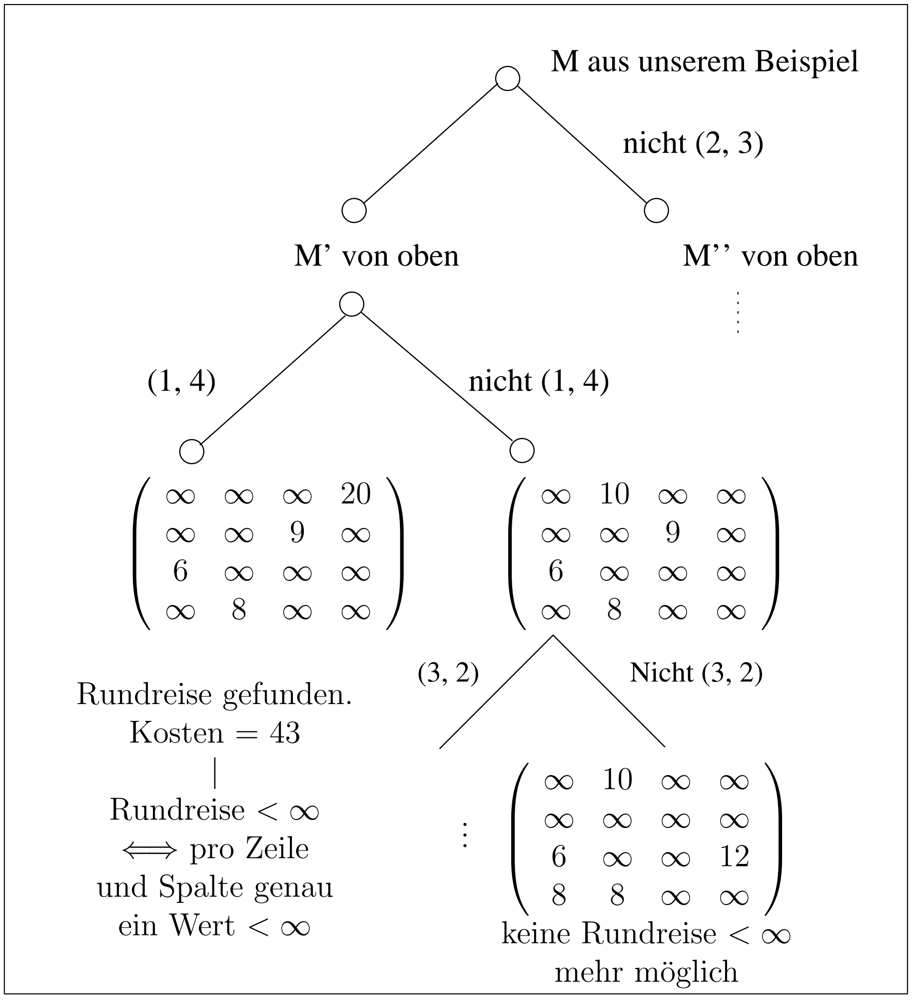
- keine Kante $u \circ\!\!\longrightarrow\!\!\circ\, 3$, $u \neq 2$ muss noch betrachtet werden
- keine $2 \circ\!\!\longrightarrow\!\!\circ\, v$, $v \neq 3$
- nicht $3 \circ\!\!\longrightarrow\!\!\circ\, 2$

Das heißt, wir suchen eine optimale Reise in

$$M' = \begin{pmatrix} \infty & 10 & \infty & 20 \\ \infty & \infty & 9 & \infty \\ 6 & \infty & \infty & 12 \\ 8 & \infty & \infty & \infty \end{pmatrix}$$

Alle nicht mehr benötigten Kanten stehen auf ∞. Ist dagegen $2 \circ\!\!\longrightarrow\!\!\circ\, 3$ nicht gewählt, dann

$$M' = \begin{pmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & \infty & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{pmatrix}$$



Algorithmus (Backtracking für TSP)

Eingabe: M=(M(u,v)), $1 \leq$ u,v $\leq$ n Ausgabe: Eine optimale Rundreise dargstellt als Modifikation von M.

```
1. if M stellt Rundreise dar
     return (M, Kosten von M)
2. if M hat keine Rundreise ≤ ∞
     return (M, ∞)                 // Etwa eine Zeile voller ∞,
                                   // eine Spalte voller ∞
3. Wähle (u,v), u≠v mit M(u,v) ≤ ∞,
  wobei in Zeile von u oder Spalte von v mindestens ein Wert ≠ ∞
4. M':=M modifiziert, so dass u○──→○v gewählt
                                   //vgl. Seite 128
5. M'':= M modifiziert, dass M(u,v) = ∞
6. Führe TSP(M') aus
  Führe TSP(M'') aus
7. Vergleiche die Kosten;
  return (M, K), wobei M die Rundreise der kleineren Kosten ist.
```

Korrektheit:

Induktion über die # Einträge in $M = \infty$ (nimmt jedesmal zu). Zeit zunächst $O(2^{n(n-1)})$, $2^{n(n-1)} = 2^{\Omega(n^2)} \gg n^n$!

Verbesserung des backtracking durch branch-and-bound.
Dazu:
Für Matrix M(=Knoten im Baum): $S(M)$ = untere(!) Schranke an die Kosten aller Rundreisen zu M, d.h. $S(M) \leq$ Kosten aller Rundreisen unter M im Baum.

Prinzip: Haben wir eine Rundreise der Kosten K gefunden, dann keine Auswertung (Expansion) der Kosten M auf $S(M) \geq K$ mehr.
Allgemein ist es immer so:
$S(k)$ untere Schranke bei Minimierungsproblemen
$S(k)$ obere Schranke bei Maximierungsproblemen (etwa Max-k-Sat),
wobei k Knoten im backtracking-Baum.
Es folgen vier konkrete $S(M)$s für unser TSP-Backtracking:

1. $S_1(M)$ = minimale Kosten einer Rundreise unter M. (Das ist die beste (d.h. größte) Schranke, aber nicht polynomial zu ermitteln)

2. $S_2(M)$ = Summe der Kosten der bei M gewählten Kanten.
   Ist untere Schranke bei $M(u, v) \geq 0$ (Das können wir aber hier annehmen, im Unterschied zu den kürzesten Wegen (Wieso?))
   Ist M = Wurzel des backtracking-Baumes, dann im allgemeinen $S_2(M) = 0$, sofern nicht $M = (\infty\infty\infty m\infty\infty)$ oder analog für Spalte.

3. $S_3(M) = 12 \cdot \sum_v \text{Min}\{M(u, v) + M(v, \infty) | u, v \in V\}$ (jedesmal das Minimum, um durch den Knoten v zu gehen) Wieso ist $S_2(M)$ untere Schranke?
   Sei R eine Rundreise unter k. Dann gilt:
   $K(R)$ $\quad (R = (1, v_1, v_{n-1}, 1))$
   $= M(1, v_1) + M(v_1, v_2) + \cdots + M(v_{n-1}, 1)$
   $= \frac{1}{2}(M(1, v_1) + M(v_1, v_2) + \cdots + M(v_{n-1}, 1) \cdots + M(v_{n-1}, 1) + M(v_{n-1}, 1))$
   shift nur 1 nach rechts.
   $= \frac{1}{2}(M(v_{n-1}, 1) + M(1, v_2) + M(1, v_1) + M(v_1, v_2) + M(v_1, v_2) + M(v_2, v_3) + \cdots + M(v_{n-2}, v_{n-1}) + M(v_{n-1}, 1)) \geq 12 \cdot \sum_v \text{Min}\{M(u, v) + M(v, w) | u, w \in V\}$
   Also ist $S_3(M)$ korrekte Schranke. Es ist $S_3(M) = \infty \iff$ Eine Zeile oder Spalte voller ∞. (Ebenso für $S_2(M)$).
   Betrachten wir M von Seite 127 Wir ermitteln $S_2(M)$.

v=1	5+10	$2 \circ \longrightarrow \overset{1}{\circ} \longrightarrow \circ\ 2$		
v=2	8+5	$4 \circ \longrightarrow \overset{2}{\circ} \longrightarrow \circ\ 1$		
v=3	9+6	$2 \circ \longrightarrow \overset{3}{\circ} \longrightarrow \circ\ 1$	oder	$4 \circ \longrightarrow \overset{3}{\circ} \longrightarrow \circ\ 1$
v=4	10+8	$2 \circ \longrightarrow \overset{4}{\circ} \longrightarrow \circ\ 1$	oder	$2 \circ \longrightarrow \overset{4}{\circ} \longrightarrow \circ\ 2$

Also $S_3(M) = \frac{1}{2} \cdot 61 = 30,5$.

Wegen ganzzahliger Kosten sogar $S_3(M) \geq 31$.
Also heißt das $K(R) \geq 31$ für jede Reise R.
Ist $2 \circ\!\!\longrightarrow\!\circ 3$ bei M gewählt, dann

$$M' = \begin{pmatrix} \infty & 10 & \infty & 20 \\ \infty & \infty & 9 & \infty \\ 6 & \infty & \infty & 12 \\ 8 & 8 & \infty & \infty \end{pmatrix}$$

v=1 6+10 $3 \circ\!\longrightarrow\!\circ 1 \longrightarrow\!\circ 2$
v=2 8+9 $4 \circ\!\longrightarrow\!\circ 2 \longrightarrow\!\circ 3$
v=3 9+6 $2 \circ\!\longrightarrow\!\circ 3 \longrightarrow\!\circ 1$
v=4 12+8 $3 \circ\!\longrightarrow\!\circ 4 \longrightarrow\!\circ 2$ oder $3 \circ\!\longrightarrow\!\circ 4 \longrightarrow\!\circ 1$

$S_3(M') = \frac{1}{2}(16 + 17 + 15 + 20) = 34$ wogegen $S_2(M') = 9$.

4. Betrachten wir eine allgemeine Matrix $M = (M(u, v))_{1 \leq u,v \leq n} \ldots$ alle Einträge ≥ 0!
Ist R eine Rundreise zu M, dann ergeben sich die Kosten von R als $K(R) = z_1 + z_2 + \cdots + z_n$, wobei z_n geeignete Werte aus Zeile n darstellt, alternativ ist $K(R) = s_1 + s_2 + \cdots + s_n$, wobei s_n einen geeigneten Wert aus Spalte n darstellt. Dann gilt
$S_4'(M) = \text{Min}\{M(1, 1), M(1, 2), \ldots, M(1, n)\} + \min\{M(2, 1), M(2, 2), \ldots, M(2, n)\} +$
$\vdots$
$+$
$\text{Min}\{M(n, 1), M(n, 2), \ldots, M(n, n)\}$ nimmt aus jeder Zeile den kleinsten Wert. Ist korrekte Schranke. Wenn jetzt noch nicht alle Spalten vorkommen, können wir noch besser werden. Und zwar so: Sei also für $1 \leq u \leq n$
$M_u = \text{Min}\{M(u, 1), M(u, 2), \ldots M(u, n)\}$ = kleinster Wert der Zeile n. Wir setzen

$$\hat{M} = \begin{pmatrix} M(1,1) - M_1 & \ldots & M(1,n) - M_1 \\ M(2,1) - M_2 & \ldots & M(2,n) - M_2 \\ & \vdots & \\ M(n,1) - M_n & \ldots & M(n,n) - M_n \end{pmatrix}$$

Alles ≥ 0. Pro Zeile ein Eintrag $= 0$.
Es gilt für jede Rundreise R zu M, $K_M(R) = K_{\hat{M}}(R) + S_4'(M) \geq S_4'(M)$. mit K_M = Kosten in M und $K_{\hat{M}}$ = Kosten in $\hat{M}$, in $\hat{M}$ alles ≥ 0.
Ist in $\hat{M}$ in jeder Spalte eine 0, so $S_4'(M)$ = die minimalen Kosten einer Rundreise. Ist das nicht der Fall, iterieren wir den Schritt mit den Spalten von $\hat{M}_u$.
Ist also

$S_u = \text{Min}\{\overbrace{\hat{M}(1, u), \hat{M}(2, u), \ldots, \hat{M}(n, u)}^{\text{Spalte von } \hat{M}}\}$, dann

$$\hat{\hat{M}} = \begin{pmatrix} \hat{M}(1,1) - S_1 & \dots & \hat{M}(1,n) - S_n \\ \hat{M}(2,1) - S_1 & \dots & \hat{M}(2,n) - S_n \\ & \vdots & \\ \hat{M}(n,1) - S_1 & \dots & \hat{M}(n,n) - S_n \end{pmatrix}$$

Alles ≥ 0. Pro Spalte eine 0, pro Zeile eine 0.
Für jede Rundreise R durch $\hat{M}$ gilt $K_{\hat{M}}(R) = K_{\hat{\hat{M}}}(R) + S_1 + \cdots + S_n$ Also haben wir insgesamt:

$$K_M(R) = K_{\hat{M}}(R) + M_1 + \cdots + M_n = K_{\hat{\hat{M}}}(R) + \overbrace{S_1 + \cdots + S_n}^{\text{aus}\hat{M}} + \underbrace{M_1 + \cdots + M_n}_{\text{aus M}} \geq M_1 +$$

$\cdots + M_n + S_1 + \cdots + S_n$. (in $\hat{\hat{M}}$ alles ≥ 0)
Wir definieren offiziell:
$S_4(M) = M_1 + \cdots + M_n + S_1 + \dots S_n$
Aufgabe:
$S_1(M) \geq S_4(M) \geq S_3(M) \geq S_2(M)$. ($S_4(M) \geq S_3(M)$ ist nicht ganz so offensichtlich.)

In unserem Eingangsbeispiel von 127

$$M = \begin{pmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{pmatrix}$$

$$M_1 = 10, M_2 = 5, M_3 = 6, M_4 = 8$$

$$\hat{M} = \begin{pmatrix} \infty & 0 & 5 & 10 \\ 0 & \infty & 4 & 5 \\ 0 & 7 & \infty & 6 \\ 0 & 0 & 1 & \infty \end{pmatrix}$$

$$S_1 = 0, S_2 = 0, S_3 = 1, S_4 = 5$$

$$\hat{\hat{M}} = \begin{pmatrix} \infty & 0 & 4 & 10 \\ 0 & \infty & 3 & 0 \\ 0 & 7 & \infty & 1 \\ 0 & 0 & 0 & \infty \end{pmatrix}$$

$$\text{Also} S_4(M) = 35 > S_3(M) = 31.$$

Die Schranken lassen sich für jede Durchlaufreihenfolge des backtracking-Baumes zum Rausschneidem weiterer Teile verwenden.

Algorithmus (Offizielles branch-and-bound)

Frot des aktuellen Teils des backtracking-Baumes im Heap geordnet nach $S(M)$ ($S(M)$ = die verwendete Schranke).
Immer an dem M mit $S(M)$ minimal weitermachen. Minimale gefundene Rundreise vermerken.
Anhalten, wenn $S(M) \geq$ Kosten der minimalen Reise für $S(M)$ minimal im Heap.
Offizielles Branch-and-Bound mit Beispiel von Seite 127 und Schranke S_4.

$$\begin{pmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{pmatrix}$$

Schranke = 35

nicht (2,1) | (2,1)

$$\begin{pmatrix} \infty & & & \\ \infty & \infty & & \\ & & & \\ & & & \end{pmatrix} \qquad \begin{pmatrix} \infty & \infty & & \\ 5 & \infty & \infty & \infty \\ \infty & & & \\ \infty & & & \end{pmatrix}$$

Schranke = 34
Nehmen weiter
Schranke 35!

Schranke = 40

nicht (3,1) | (3,1)

$$\begin{pmatrix} \infty & & & \\ \infty & & & \\ \infty & & & \\ & & & \end{pmatrix} \qquad \begin{pmatrix} \infty & & \infty & \\ \infty & \infty & & \\ 6 & \infty & \infty & \infty \\ \infty & & & \end{pmatrix}$$

Schranke = 39

Schranke = 34
weiter 35

nicht (4,2) | (4,2)

$$\begin{pmatrix} \infty & 10 & \infty & 20 \\ \infty & \infty & 9 & 10 \\ 6 & \infty & \infty & \infty \\ \infty & \infty & 9 & \infty \end{pmatrix} \qquad \begin{pmatrix} \infty & \infty & \infty & 20 \\ \infty & \infty & 9 & 10 \\ 6 & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty \end{pmatrix}$$

Schranke = 35
(10+9+6+9+1 = 35)

Schranke = 43
(20+9+6+8 = 43)

nicht (2,4) | (2,4)

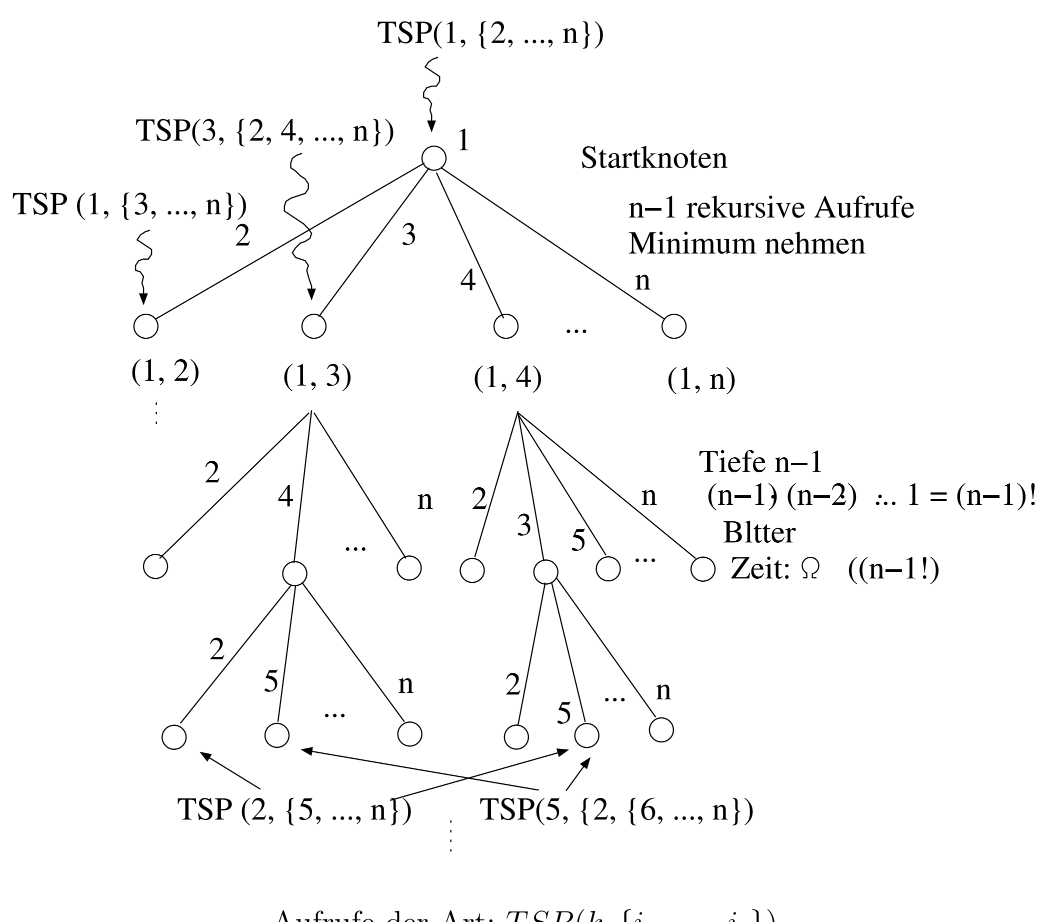
$$\begin{pmatrix} \infty & 10 & \infty & 20 \\ \infty & \infty & 9 & \infty \\ 6 & \infty & \infty & \infty \\ \infty & \infty & 9 & \infty \end{pmatrix} \qquad \begin{pmatrix} \infty & 10 & \infty & \infty \\ \infty & \infty & \infty & 10 \\ 6 & \infty & \infty & \infty \\ \infty & \infty & 9 & \infty \end{pmatrix}$$

Schranke = 45

Schranke = 43

Es ist wegen der Schranke keine bessere Lösung mehr möglich.

Aufgabe: Zeigen Sie für M und M' mit $\begin{array}{c}\mathrm{M}\\ |\\ \mathrm{M'}\end{array}$ im Baum, dass $S_2(M') \geq S_(M), S_3(M') \geq S_3(M)$. Das backtracking mit Verzweigen nach dem Vorkommen einer Kante ist praktisch wichtig und führt ja auch zum branch-and-bound. Jedoch, viel Beweisbares ist dort bisher nicht herausgekommen. Wir fangen mit einer anderen Art des backtracking an: Verzweigen nach dem nächsten Knoten, zu dem eine Rundreise gehen soll. Der Prozeduraufrufbaum hat dann folgende Struktur:

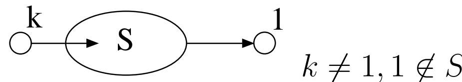


Aufrufe der Art: $TSP(k, \underbrace{\{i_a, \ldots, i_e\}}_{\subseteq\{2,\ldots,n\}})$

Wieviele verschiedene rekursive Aufrufe TSP(k, S) gibt esprinzipiell?

- Wähle k: n Möglichkeiten
- Wähle S: $\leq 2^{n-1}$ Möglichkeiten

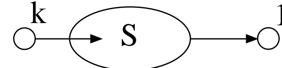
$2^{n-1} \cdot n = 2^{n-1+logn} = 2^{O(n)}$ Wogegen $(n-1)! = 2^{\overbrace{\Omega(logn) \cdot n}^{\gg n}}$.
Wir tabellieren wieder die Ergebnisse der rekursiven Aufrufe, in der Reihenfolge, wobei TSP(k, S) = kürzeste Reise:



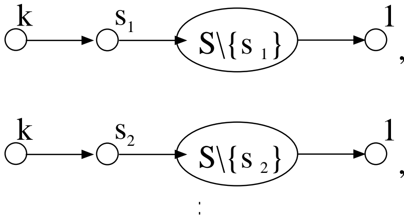
Zunächst $S = \emptyset$

TSP$(2, \emptyset)$ = M(2, 1) // M=(M(u, v)) Eingangsmatrix
TSP$(n, \emptyset)$ = M(n, 1)
Dann $|S| = 1$
TSP(2, {S}) = M(2, 3) + TSP(3, S\{3})
⋮
TSP(2, {n}) = M(2, n) + TSP(n, S\{n})
⋮
Dann $|S|$=2
TSP(2, {3, 4}) = Min {M(2, 3) + TSP(3, {4}) + M(2, 4) + TSP(4, {3})}
⋮
Allgemein
TSP(k, S) = Min {M(k, s) + TSP(s, S\{s}) | s ∈ S}

also: Minimum über den Einstiegspunkt in S:



= Minimum von:



Alle Elemente aus S probieren

Algorithmus (TSP mit dynamischem Programmieren)

Datenstruktur: array $TSP[1, \ldots n, \overbrace{0 \ldots 0}^{\text{n-1 Bits}}, \ldots \overbrace{1 \ldots 1}^{\text{n-1 Bits}}]$ of integer

```
1. for i=2 to n TSP(k, 0 ..., 0) := M(k,1)
2. for i=2 to n-2 {
     for all S⊆{2, ..., n}, |S| = i {
     for all k ∈ {2, ..., n}\S{
       TSP(k,S) = Min{M(k,s) + TSP(s, S\{s})}
     } } }
```

Ergebnis ist TSP(1, {2, …, n-1})=Min{M(1,s)+TSP(s,{2,…,n-1}\s)}
Laufzeit: $O(n \cdot 2^n)$ Einträge im array TSP. Pro Eintrag das minimum ermitteln, also Zeit $O(n)$. Also $O(n^2 \cdot 2^n)$.

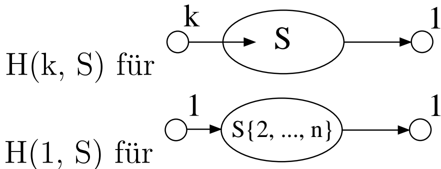
Es ist $n^2 \cdot 2^n = 2^{n+2logn} \leq 2^{n(1+\varepsilon)}$
$= 2^{(1+\varepsilon)\cdot n} = (2(1+\varepsilon)!)^n \ll (n-1)!$
für $\varepsilon, \varepsilon' \geq 0$ geeignet.

$2^\varepsilon \to 1$ für $\varepsilon \to 0$, da $2^0 = 1$
Verwandt mit dem Problem des Handlungsreisenden ist das Problem des Hamiltonschen Kreises.

Definition 10.8 (*Hamilton-Kreis*):

> Sei $G = (V, E)$ ein ungerichteter Graph. Ein Hamilton-Kreis ist ein einfacher Kreis der Art $(1, v_1, v_2, \ldots, v_{n-1}, 1)$. (Also ist dies ein Kreis, in dem alle Knoten genau einmal auftreten.)

Mit dynamischem Programmieren



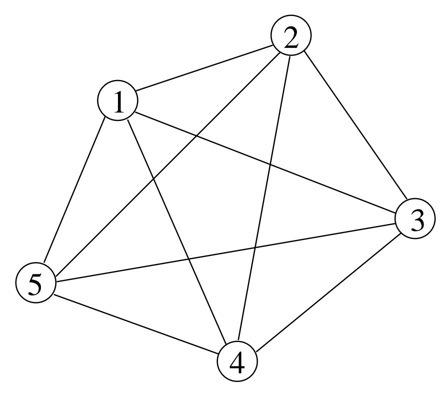
in Zeit $O(n^2 \cdot 2^n)$ lösbar. Besser als $\Omega((n-1)!)$ druch einfaches backtracking.

Definition 10.9 (*Eulerscher Kreis*):

> Sei $G = (V, E)$ ein ungerichteter Graph. Ein Eulerscher Kreis ist ein geschlossener Weg, in dem jede Kante genau einmal vorkommt.

1 S{2, ..., n} 1

Dann sollte $(1, 5, 4, 3, 2, 1, 4, 2, 5, 3, 1)$ ein Eulerscher Kreis sein.



Scheint nicht zu gehen. Dynamisches Programmieren? $O(m \cdot n \cdot 2^m)$, $|V| = n, |E| = m$ sollte klappen. Es geht aber in polyninomialer Zeit und wir haben wieder:

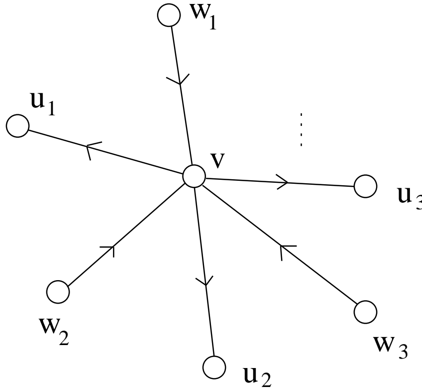
- Hamilton-Kreis: polynomiale Zeit, nicht bekannt
- Eulerscher Kreis: polynomiale Zeit

Dazu der **Satz 10.10**:

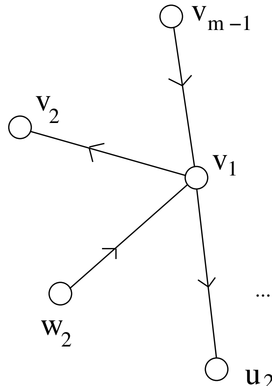
Sei G ohne Knoten vom Grad 0 (Grad(v) = # direkter Nachbarn), G hat Eulerschen Kreis $\Longleftrightarrow$ G zusammenhängend und Grad(v) = gerade für alle $v \in V$.

Beweis 10.11:

„$\Rightarrow$“ Sei also $G = (V, E), |V| = n, |E| = m$. Sei $K = (v_1, v_2, v_3, \ldots, v_m = v_1)$ ein Eulerscher Kreis von G. Betrachten wir einen beliebigen Knoten $v \neq v_1$ der etwa $k-$mal auf K vorkommt, dann haben wir eine Situation wie



alle u_i, w_i verschieden, also $Grad(v) = 2k$. Für $v = v_1$ haben wir



und $Grad(v_1)$ ist gerade

„$\Leftarrow$“ Das folgende ist eine Schlüsselbeobachtung für Graphen, die zusammenhängend sind und geraden Grad haben: Wir beginnen einen Weg an einem beliebigen Knoten v_1 und benutzen jede vorkommende Kante nur einmal. Irgendwann landen wir wieder an v_1 :

$$W = v_1 \rightarrow \circ \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow \cdots \rightarrow v_1$$

Ist etwa $v_3 = v_5 = u$, so hat u mindestens 3, also ≥ 4 Nachbarn. Wir können also von v_5 wieder durch eine neue kante weggehen. Erst, wenn wir bei v_1 gelandet sind, ist das nicht mehr sicher, und wir machen dort Schluss.

Nehmen wir nun W aus G heraus, haben alle Knoten wieder geraden Grad und wir können mit den Startknoten $v_1, v_2, \dots$ so weitermachen und am Ende alles zu einem Eulerschen Kreis zusammensetzen. konkret sieht das so aus:

1. Gehe einen Weg W wie oben und streiche die Kanten von W aus G. (W gehört nicht direkt zum Eulerschen Kreis)
2. Nach Induktionsvoraussetzung haben wir einen Eulerschen Kreis auf dem Stück, das jetzt von v_1 erreichbar ist. Schreibe diesen Eulerschen Kreis hin. Wir bekommen

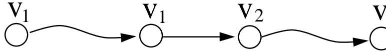
   v1 v1

   Eulerscher Kreis

3. Erweitere den Eulerschen Kreis auf W zu

   v1 v1 v2

   Mache von v_2 aus dasselbe wie bei v_1 :

   So geht es weiter bis zum Ende von W

Aufgabe: Formulieren Sie mit dem Beweis oben einen (rekursiven) Algorithmus, der in $O(|V|+|E|)$ auf einen Eulerschen Kreis testet und im positiven Fall einen Eulerschen Kreis gibt.

Eine weitere Möglichkeit, die kombinatorische Suche zu gestalten, ist das Prinzip der lokalen Suche.

Beim aussagenlogischen Erfüllbarkeitsproblem fpr KNF gestaltet es sich etwa folgendermaßen:

Algorithmus (Lokale Suche bei KNF)

Eingabe: F in KNF

1. Wähle eine Belegung $a = (a_1, \dots, a_n)$ der Variablen.
2. if $a(F) = 1$ return „$a(F) = 1$“
3. Wähle Klausel C von F mit $a(C) = 0$
4. Ändere a so, dass $a(C) = 1$ ist, indem ein (oder mehrere) Werte von a geändert werden.
5. Mache bei 1. weiter.

Wie Nicht-Erfüllbarkeit erkennen?
Eine Modifikation. 2 lokale Suchen, von $a = (0, \dots, 0)$ und $b = (1, \dots, 1)$.
Nun gilt: Jede Belegung unterscheidet sich auf $\leq \frac{\lceil n \rceil}{2}$ Positionen von a ($\leq \frac{\lceil n \rceil}{2}$ Einsen) oder b

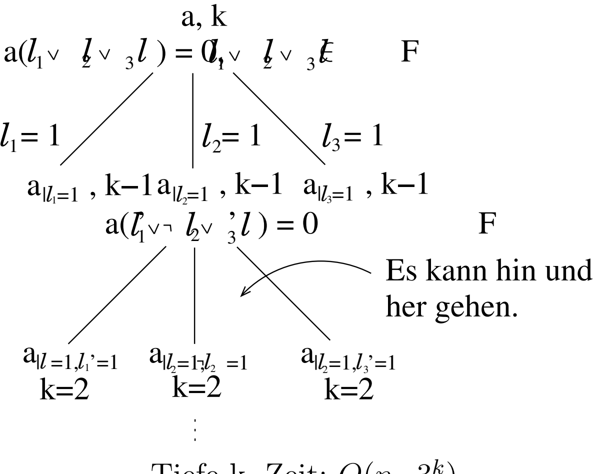
($\geq \frac{\lceil n \rceil}{2}$ Einsen).

Bezeichnung:
Für zwei Belegungen $a = (a_1, \dots, a_n), b = (b_1, \dots, b_n)$ ist $D(a, b) = |\{i | 1 \leq i \leq n, a_i \neq b_i\}|$. also die # Positionen, auf denen a und b verschieden sind, die Distanz von a und b.

Für F und Belegung a gilt: Es gibt b mit $b(F) = 1$ und $D(a, b) \leq k$
$\iff$

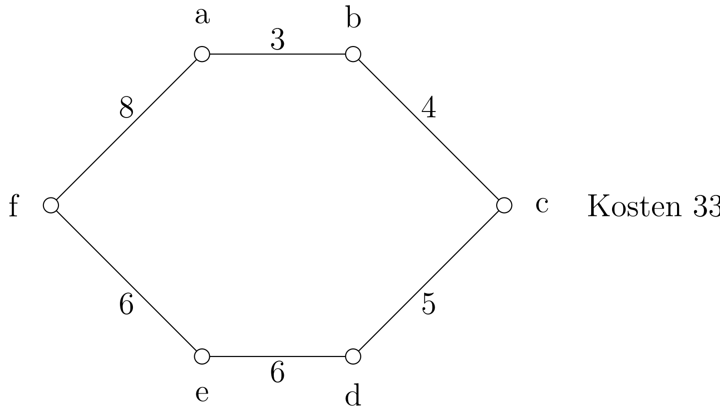
- $a(F) = 1$ oder
- Für jedes $C = l_1 \vee \dots \vee l_m \in F$ mit $a(C) = 0$ gibt es ein l_j, so, dass für alle $a'(l_j) = 0$ und a' sonst wie a gilt: $D(b, a) \leq k - 1$.

Mit diesem Prinzip rekursiv für 3-KNF F:

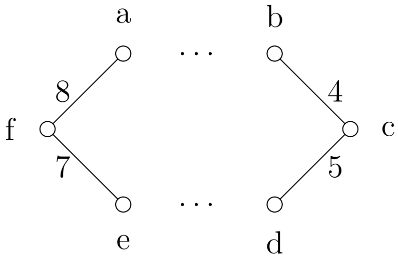


Einmal rekursiv mit $F, a = (0, \dots, 0), \frac{\lceil n \rceil}{2}$ noch einmal mit $F, b = (1, \dots, 1), \frac{\lceil n \rceil}{2}$.
Dann Zeit $O(n \cdot 3^{\frac{n}{2}} = O(\underbrace{1,7}_{<2!} \dots^n)$ (3 wegen 3-KNF)
Auf die Art kann bis zu $O(1,5^n)$ erreicht werden.

Die lokale Suche basiert auf einem Distanzbegriff zwischen möglichen Lösungen. Bei Belegungen ist dieser klar gegeben. Wie bei Rundreisen für das TSP? Wir betrachten hier nur den Fall, dass der zugrundeliegende Graph ungerichtet ist.
Haben nun also eine Rundreise wie

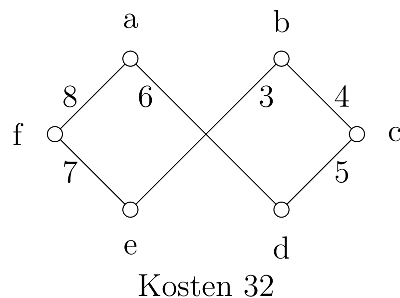


gegeben. Duch Ändern einer Kante bekommen wir keine neue Rundreise hin. An zwei benachbarten Kanten können wir nichts ändern. Löschen wir einmal zwei nicht benachtbarte Kanten a—b, c—d



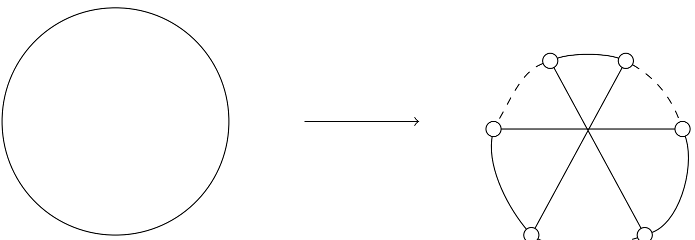
Wie können wir eine neue Rundreise zusammensetzen?

d—e (und a—b) gibt die Alte.
Also d—a (und b—e). Das gibt:



Kosten 32

Man kann in $O(n^2)$ Schritten testen, ob es so zu einer Verbesserung kommt. Ein solcher Verbesserungsschritt wird dann gemacht. Aber es gilt (leider) nicht:
Keine Verbesserung möglich $\Longleftrightarrow$ Minimale Rundreise gefunden
Allgemein kann man auch drei Kanten löschen und den Rest zusammenbauen.

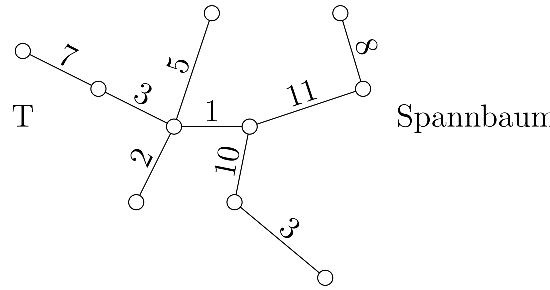


Man findet jedoch auch nicht unbedingt eine minimale Rundreise.
Bemerkung:

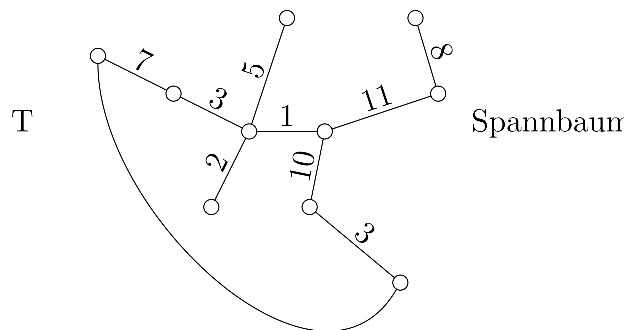
- Für all die Probleme, für die wir keine Polynomialzeit-Algorithmen angegeben haben, sind auch keine bekannt.
- Das bedeutet, für diese Probleme kann das weitgehend blinde Ausprobieren von (exponentiell) vielen Lösungsmöglichkeiten nicht vermieden werden.
- Es ist kein Beweis bekannt, dass es nicht doch in polynomialer Zeit geht. (Wird aber nicht erwartet)
- Die hier betrachteten Exponentialzeit-Probleme sind ineinander übersetzbar (Theoretische Informatik 2)

Die lokale Suche erlaubt es dagegen, in Polynomialzeit einen minimalen Spannbaum zu finden. Erinnerung: Können minimalen Spannbaum in $O(|E|log|V|)$ finden (sogar $O(|E|log*|V|)$, wenn E nach den Kosten vorsortiert ist.)

Wir definieren zunächst den grundlegenden Transformationsschritt der lokalen Suche beim minimalen Spannbaum:



Wähle eine Kante e, die nicht im Baum ist:



Diese Kante induziert genau einen einfachen Kreis mit dem Spannbaum. Wir prüfen für jede Kante auf diesem Kreis, ob ihre Kosten > Kosten von e sind. Haben wir eine solche Kante gefunden, löschen wir sie und bekommen einen Spannbaum mit geringeren Kosten.

Satz 10.12:

Haben wir einen nicht-minimalen Spannbaum, so gibt es immer eine Kante e, mit der oben angegebener Transformationsschritt zu einer Verbesserung führt.

Beweis 10.13:

Sei also T ein Spannbaum nicht minimal, sei S ein minimaler Spannbaum, dann $S \neq T$. Wir betrachten einmal die Kantenmenge $S \setminus T = \{e_1, \ldots, e_k\}$

Falls der oben angegebene Transformationsschritt mit T und e_1, T und $e_2, \ldots, T$ und e_k jedesmal zu keiner Verringerung der Kosten führt, transformieren wir so:

$T_1 = T \cup \{e_1\}$
$R1 := T_1 \setminus \{\text{Kante nicht aus S, auf Kreis durch} e_1\}$ //keine Verbesserung
$T_2 := R_1 \cup \{e_2\}$ //noch Annahme
$R_1 := T_2 \setminus \{$ Kante nicht aus S auf Kreis durch $e_2\}$ //Keine Verbesserung, da e_1 nicht kleiner als alle Kanten auf dem vorherigen Kreis
$\vdots$
$T_k := R_{k-1} \cup \{e_k\}$
$R_k := T_k \setminus \{$Kante nicht aus S auf Kreis durch $e_k\}$ //keine Verbesserung, da vorher keiner Verbesserungen

Also, es sind die Kosten von T minimal, im Widerspruch zur Annahme.

Laufzeit eines Algorithmus mit Transformationsschritt:

- Maximal $|E|^2$ Schritte
- Pro Schritt $|E|$ Kanten ausprobieren
- Pro Kante $O(|V| + |E|)$

Insgesamt $O(|E|^4 + |V|)$.

11 Divide-and-Conquer und Rekursionsgleichungen

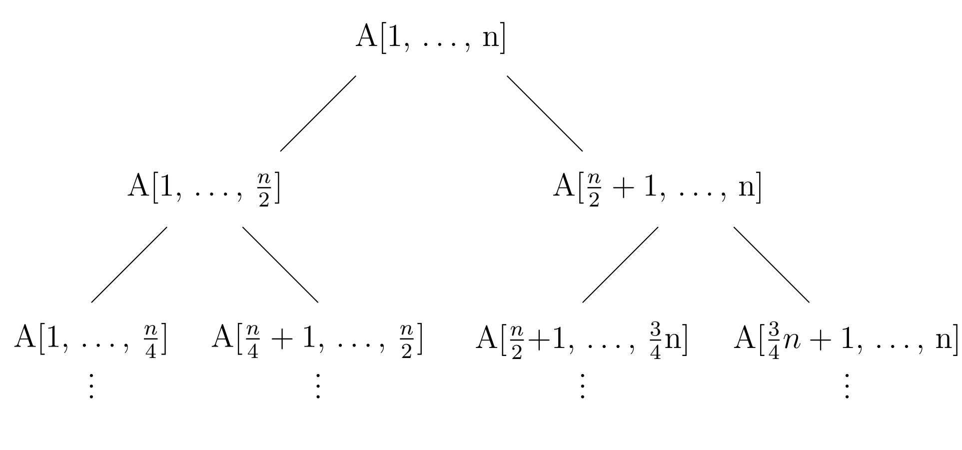
Divide-and-Conquer

- Problem aufteilen in Teilprobleme
- Teilproblem (rekursiv) lösen
- Lösungen der Teilprobleme zusammensetzen

Typische Beispiele: Binäre Suche, Meregesort, Quicksort

```
Mergesort(A[1, ..., n]){
1. if (n==1) oder (n==0) return A;
2. B1 = Mergesort (A[1, ..., n/⌊2⌋]);
3. B2 = Mergesort (A[n/⌊2⌋+1, ..., n]); //2. + 3. divide-Schritte
4. return "Mischung von B1 und B2"   //Mischung bilden, conquer-Schritt
}
```

Aufrufbaum bei n = Zweierpotenz



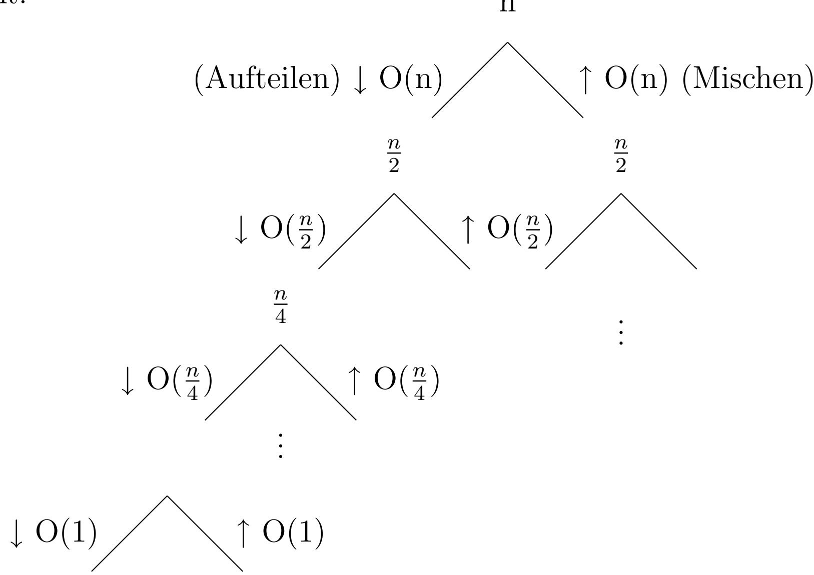
Tiefe: genau $log_2 n$

Laufzeit:

- Blätter: $O(n)$
- Aufteilen: beim m Elementen $O(m)$

- Zusammensetzen von 2-mal $\frac{n}{2}$ Elementen: $O(m)$

Damit insgesamt:



Für die Blätter: $n \cdot O(1) = O(n)$
Für das Aufteilen: $c \cdot n + c \cdot \frac{n}{2} + c \cdot \frac{n}{4} + \cdots + c \cdot 1 + c \cdot \frac{n}{2} + \ldots$ Wo soll das enden?

Wir addieren in einer anderen Reihenfolge:

1	$c \cdot n$
2	$+c \cdot \frac{n}{2} + c \cdot \frac{n}{2}$
3	$+c \cdot (\frac{n}{4} + \frac{n}{4} + \frac{n}{4} + \frac{n}{4})$
4	$+c \cdot (\frac{n}{8} + \cdots + \frac{n}{8}$
$\vdots$	
log n	$+c \cdot \underbrace{(1 + \cdots + 1)}_{\text{n-mal}}$

$= logn \cdot c \cdot n = O(n \cdot logn)$ Ebenso für das Mischen: $O(n \cdot logn)$
Insgesamt $O(n \cdot logn) + O(n) = O(n \cdot logn)$.

- Wichtig: Richtige Reihenfolge beim Zusammenaddieren. Bei Bäumen oft stufenweise!
- Die eigentliche Arbeit beim divide-and-conquer geschieht im Aufteilen und im Zusammenfügen. Der Rest ist rekusiv.
- Mergesort einfach bottom-up ohne Rekursion, da starrer Aufrufbaum.
  $A[1,2], A[3,4], \ldots, A[n-1,n]$ Sortieren
  $A[1,\ldots,4], A[5,\ldots,8], \ldots$ Sortieren
  $\vdots$
  Auch $O(n \cdot logn)$.
- Ebenfalls Heapsort sortiert in $O(n \cdot logn)$.
- Bubblesort, Insertion Sort, Selection Sort $O(n^2)$

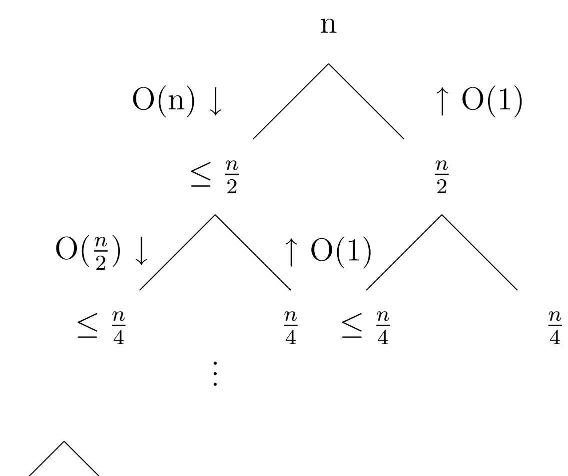
zu Quicksort:
array A[1, ..., n] of „geordneter Datentyp“

Algorithmus: Quicksort

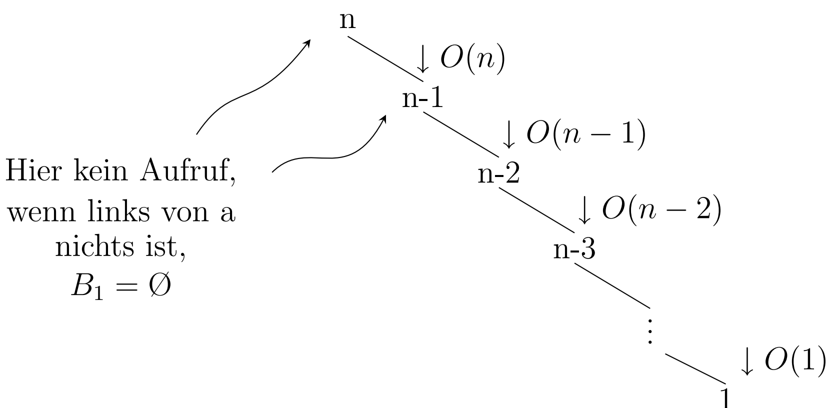
```
1. if (n == m) return
2. a = ein A[i];
3. Lösche A[i] aus A.
4. for j=1 to n {
5.   if (A[j] ≤ a){                A[j] als nächstes Element zu B1;
       break;
       }
6. if (A[j] > a){
     A[j] zu B2}
   }
7. A = B1 a B2
8. if (B1 ≠ Ø) Quicksort (B1)        // B1 als Teil von A
9. if (B2 ≠ Ø) Quicksort (B2)        // B2 als Teil von A
```

Beachte:
Die Prozedur Partition aus dem 1. Semester erlaubt es, mit dem array A alleine auszukommen.

Prozedurbäume:



Hier $O(n \cdot logn)$:
$O(n) + 2 \cdot O(\frac{n}{2} + 4 \cdot O(\frac{n}{4} + \cdots + n \cdot O(1) + O(n)$
(O(n) ... Blätter)
Aber auch:



Laufzeit:

$n + n - 1 + \cdots + 3 + 2 + 1 = \dfrac{n \cdot (n-1)}{2} = O(n^2)$ (Aufteilen) $1 + 1 + \cdots + 1 + 1 + 1 = O(n)$ (Zusammensetzen)
Also: $O(n^2)$ und auch $\Omega(n^2)$. Wieso Quicksort trotzdem in der Praxis häufig? In der Regel tritt ein gutartiger Baum auf, und wir haben $O(n \cdot logn)$. Vergleiche immer mit festem a, a in einem Register $\Longrightarrow$ schnelleres Vergleichen. Dagegen sind beim Mischen von Mergesort öfter beide Elemente des Vergleiches neu.

Aufrufbaum von Quicksort vorher nicht zu erkennen. Keine nicht-rekursive Implementierung wie bei Mergesort. Nur mit Hilfe eines (Rekursions-) Kellers.
Noch einmal zu Mergesort. Wir betrachten den Fall, dass n eine Zweierpotenz ust, dann $\frac{n}{2}, \frac{n}{4}, \ldots, 1 = 2^0$ ebenflls. Sei $T(n)$ = worst-case-Zeit bei A[1, …, n]. Dann gilt für ein geeignetes c:
$T(n) \leq c \cdot n + 2 \cdot T(\frac{n}{2})$
$T(1) \leq c$

- Einmaliges Teilen O(n)
- Mischen O(n)
- Aufrufverwaltung hier O(n)

Betrachten nun $T(n) = c \cdot n + 2 \cdot T(\frac{n}{2})$
$T(1) = c$

Dann durch Abwickeln der rekursiven Gleichung:
$T(n) = c \cdot n + 2 \cdot T(\frac{n}{2})$
$= c \cdot n + 2 \cdot c \cdot \frac{n}{2} + 2 \cdot 2 \cdot T(\frac{n}{4})$
$= c \cdot n + c \cdot n + 4 \cdot c \cdot \frac{n}{2} + 2 \cdot 2 \cdot 2 \cdot T(\frac{n}{8})$
$\vdots$

$= c \cdot n \cdot logn + 2 \cdot 2 \cdot \dots \cdot 2 \cdot T(1)$
$= c \cdot n \cdot logn + c \cdot n$
$= O(c \cdot nlogn) = O(n \cdot logn)$! Schließlich noch ein strenger Induktionsbeweis, dass $(T(n) = O(n \cdot logn)$. Induktionsanfang: $T(1) = O(1 \cdot log1) = 0$!
stimmt so nicht. Also fangen wir bei $T(2)$ an:
$T(2) \leq d \cdot 2$ für ein d geht.
Induktionsschluss:
$T(n) = c \cdot n + 2 \cdot T(\frac{n}{2})$
$\leq c \cdot n + 2 \cdot d \cdot \frac{n}{2}$
$\leq c \cdot n + 2 \cdot d \cdot \frac{n}{2} \cdot (logn - 1)$
$= c \cdot n + d \cdot nlogn - d \cdot n$
$\leq d \cdot nlogn$
geht, wenn $c \geq d$ gewählt ist.

Wie sieht das bei Quicksort aus?
$T(n) =$ worst-case-Zeit von Quicksort auf $A[1, \dots, n]$.

Dann

$T(n) \leq c \cdot n + T(n-1)$,
$T(n) \leq c \cdot n + T(1) + T(n-2)$,
$T(n) \leq c \cdot n + T(2) + T(n-3)$,
$\vdots$
$T(n) \leq c \cdot n + T(n-2) + T(1)$
$T(n) \leq c \cdot n + T(n-1)$

Also

$T(1) \leq c$
$T(n) \leq c \cdot n + Max(\{T(n-1)\} \cup \{T(i) + T(n-i-1) | 1 \leq i \leq n-1\})$

Dann $T(n) \leq dn^2$ für d geeignet.
Induktionsanfang: ✓
Indukstionsschluss:
$T(n) \leq c \cdot n + Max(\{d(n-1)^a\} \cup \{\underbrace{d \cdot i^2 + d(n-i-1)^2}_{i+n-i-1-n-1} | 1 \leq i \leq n-1\})$
$\leq c \cdot n + d(n-1)^2$
$\leq \cdot n^2$ für $d \geq c$

Aufgabe: Stellen Sie die Rekursionsgleichung für eine rekursive Version der binären Suche auf und schätzen sie diese bestmöglich ab.

Im Folgenden behandeln wir das Problem der Multiplikation großer Zahlen. Bisher haben wir angenommen: Zahlen in ein Speicherwort $\Longrightarrow$ arithmetische Ausdrücke in O(1).
Man spricht vom uniformen Kostenmaß. Bei größeren Zahlen kommt es auf den Multiplikationsalgorithmus an. Man misst die Laufzeit in Abhängigkeit von der # Bits, die der Rechner

verarbeiten muss. Das ist bei einer Zahl n etwa $log_2 n$ (genauer für $n \geq 1 \llcorner log_2 n \lrcorner + 1$).
Man misst nicht in der Zahl selbst!
Die normale Methode, zwei Zahlen zu addieren, lässt sich in O(n) Bitoperationen implementieren bei Zahlen der Länge n:

$$\begin{array}{rccc} & a_1 & \dots & a_n \\ + & b_1 & \dots & b_n \\ \hline \\ & & \dots & a_n + b_n \end{array}$$

Multiplikation in $O(n^2)$:

$(a_1 \dots a_n) \cdot (b_1 \dots b_n)$

$$\begin{array}{rr} (a_1 \dots a_n) \cdot b_1 & O(n) \\ (a_1 \dots a_n) \cdot b_2 & O(n) \\ & \vdots \\ (a_1 \dots a_n) \cdot b_n & O(n) \\ \hline \end{array}$$

Summenbildung

$n - 1$ Additionen mit Zahlen der Länge $\leq 2 \cdot n \implies O(n^2)$! Mit divide-and-conquer geht es besser! Nehmen wir zunächst einmal an, n ist eine Zweierpotenz. Dann

$$\begin{array}{c} \overbrace{a_1 \dots a_n}^{a:=} = \overbrace{a_1 \dots a_{\frac{n}{2}}}^{a':=} \overbrace{a_{\frac{n}{2}-1} \dots a_n}^{a'':=} \\ \underbrace{b_1 \dots b_n}_{b:=} = \underbrace{b_1 \dots b_{\frac{n}{2}}}_{b':=} \underbrace{b_{\frac{n}{2}-1} \dots b_n}_{b'':=} \end{array}$$

Nun ist

$$a \cdot b = \underbrace{(a' \cdot 2^{\frac{n}{2}} + a'')}_{a=} \cdot \underbrace{b' \cdot 2^n 2 + b''}_{b=}$$

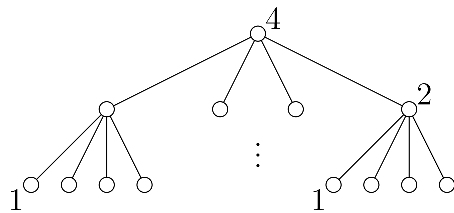
$$= \underbrace{a'}_{\frac{n}{2}} \cdot \underbrace{b'}_{\frac{n}{2}} \cdot 2^n + \underbrace{a'}_{\frac{n}{2}} \cdot \underbrace{b''}_{\frac{n}{2}} \cdot 2^{\frac{n}{2}} + a'' \cdot b' \cdot 2^{\frac{n}{2}} + a'' \cdot b''$$

Mit den 4 Produkten rekursiv weiter.
Das Programm sieht in etwa so aus:

```
Mult(a, b, n){                           //a=a1, ...an, b=b1, ..., bn

  1. if (n == 1) return a · b            // n Zweierpotenz
  2. a', a'', b', b'' wie oben.          // Divide
  3. m1:= Mult(a', b', n/2);
  4. m2:= Mult(a', b'', n/2);
  5. m3:= Mult(a'', b, n/2);
```

```
   6. m4:= Mult(a'', b'', n/2);
   7. return (m1 · 2^n + (m2+m3) · 2^(n/2) + m4)  //Conquer
}
```

Aufrufbaum, n = 4:



Tiefe $\log_2 4$
Laufzeit? Bei Mult(a,b,n) braucht der divide-Schritt und die Zeit für die Aufrufe selbst zusammen $O(n)$. Der conquer-Schritt erfolgt auch in $O(n)$. Zählen wir wieder pro Stufe:

1. Stufe $d \cdot n$ (von $O(n)$)
2. Stufe $4 \cdot d \cdot \frac{n}{2}$
3. Stufe $4 \cdot 4 \cdot d \cdot \frac{n}{4}$

$log_2 n-$te Stufe $4^{log_2 n-1} \cdot \cdot \dfrac{n}{4^{log_2 n} \cdot d}$

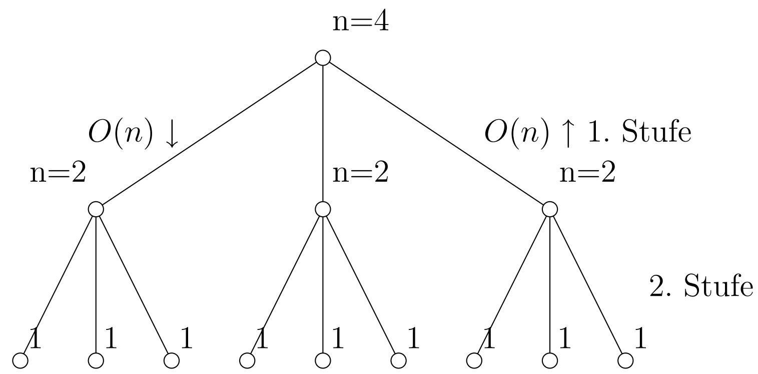
Blätter $4^{log_2 n} \cdot d$
Das gibt

$T(n)$
$\leq \sum_{i=0}^{log_2 n} 4^i \cdot d \cdot \dfrac{n}{2^i}$
$= d \cdot n \cdot \sum_{i=0}^{log_2 n} 2^i$
$= d \cdot n \cdot \dfrac{2^{log_2 n+1} - 1}{2 - 1}$
$= O(n^2)$ mit $\sum_{i=0}^{k} x^i = \dfrac{x^{k+1} - 1}{x - 1}$
für alle $x \neq 1$ (x>0, x<0 egal!), geometrische Reihe. Betrachten wir die induzierte Rekursionsgleichung. Annahme: n = Zweierpotenz. (Beachte: $2^{\lfloor log_2 n \rfloor} \leq n \leq 2^{\lfloor log_2 n \rfloor + 1}$, d.h. zwischen n und $2n$ existiert eine Zweierpotenz.)
$T(1) = d$
$T(n) = dn + 4 \cdot T(\frac{n}{2})$
Versuchen $T(n) = O(n^2)$ durch Induktion zu zeigen.
1. Versuch: $T(n) \leq d \cdot n^2$
Induktionsanfang: ✓
Induktionsschluss:
$T(n) = dn + 4 \cdot T(\frac{n}{2})$
$\leq d \cdot n + dn^2 > dn^2$ Induktion geht nicht!

Müssen irgendwie das dn unterbringen.
2. Versuch $T(n) \leq 2dn^2$ gibt ein Induktionsschluss.
$T(n) \leq dn + 2dn^2 > 2dn^2$ 3. Versuch $T(n) \leq 2dn^2 - dn$

Induktionsanfang: $T(1) \leq 2d - d = d$✓
Induktionsschluss:
$T(n) = dn + 4 \cdot T(\frac{n}{2})$
$\leq dn + 4(2d(\frac{n^2}{4} - d\frac{n}{2})$
$= dn + 2dn^2 - 2dn$
$= 2dn^2 - dn$
Vergleiche auch Seite 148, wo $log_2 \frac{n}{2} = log_2 n - 1$ wichtig ist.

Nächstes Ziel: Verbesserung der Multiplikation durch nur noch drei rekursive Aufrufe mit jeweils $\frac{n}{2}$ vielen Bits. Analysieren wir zunächst die Laufzeit.



$log_2 n$ divide-and-conquer-Schritte
+ Zeit an n Blättern. 1. Stufe $d \cdot n$,
2. Stufe $3 \cdot d \cdot \frac{n}{2}$,
3. Stufe $3 \cdot 3 \cdot d \cdot \frac{n}{4}, \ldots,$
$log_2 n - te$ Stufe Blätter $3^{log_2 n} \cdot d \cdot 1$
Dann ist $T(n) = \sum_{i=0}^{log_2 n} 3^i \cdot d \cdot \frac{n}{2^i}$
$= d \cdot n \cdot \sum (\frac{3}{2})^i$
$$= d \cdot n \cdot \frac{(\frac{3}{2})^{log_2 n+1} - 1}{\underbrace{\frac{3}{2} - 1}_{=\frac{1}{2}}}$$
$\leq \not{2} \cdot d \cdot n \cdot \frac{3}{\not{2}} \cdot (\frac{3}{2})^{log_2 n}$
$= 3 \cdot d \cdot n \cdot 2^{log_2(\frac{3}{2}) \cdot log_2 n}$
$$= 3 \cdot d \cdot n \cdot n^{\overbrace{log_2(\frac{3}{2})}^{<1}}$$
$= 3 \cdot d \cdot n^{log_2 3} = O(n^{1.59})$
Also tatsächlich besser als $O(n^2)$. Fassen wir zusammen:
2 Aufrufe mit $\frac{n}{2}$, linearer Zusatzaufwand

$T(n) = d \cdot n \cdot \sum_{i=0}^{log_2 n} \underbrace{\frac{2^i}{2^i}}_{=1}$
$= O(n \cdot logn)$ (Auf jeder Stufe, d.h. 2 Aufrufe, halbe Größe)

3 Aufrufe
$T(n) = d \cdot n \sum (\frac{3}{2})^i$ (3 Aufrufe, halbe Größe)
$= O(d \cdot n^{log_2 3})$

4 Aufrufe
$T(n) = d \cdot n \cdot \sum (\frac{4}{2})^i$ (4 Aufrufe, halbe Größe)
$= d \cdot n \cdot n$
$= O(n^2)$.

Aufgabe: Stellen Sie für den Fall der drei Aufrufe die Rekursionsgleichungen auf und beweisen Sie $T(n) = O(n^{log_2 3})$ duch eine ordnungsgemäße Induktion.

Zurück zur Multiplikation.
$\overbrace{a_1 \dots a_n}^{a:=} = \overbrace{a_1 \dots a_{\frac{n}{2}}}^{a':=} \cdot \overbrace{a_{\frac{n}{2}+1} \dots a_n}^{a'':=}$
$\overbrace{b_1 \dots b_n}^{b:=} = \overbrace{b_1 \dots b_{\frac{n}{2}}}^{b':=} \cdot \overbrace{b_{\frac{n}{2}+1} \dots b_n}^{b'':=}$

Wie können wir einen Aufruf einsparen? Erlauben uns zusätzliche Additionen: Wir beobachten zunächst allgemein: $(x - y) \cdot (u - v) = x(u - v) + y \cdot (u - v) = x \cdot u - x \cdot v + y \cdot u - y \cdot v$ (eine explizite und 4 implizite Multiplikationen (aber in Summe)) Wir versuchen jetzt $a'b'' + a''b'$ auf einen Schlag zu ermitteln:
$(a' - a'') \cdot (b'' - b')$
$= a'b'' - a'b' + a''b'' + a''b'$
Und: $a'b', a''b''$ brauchen wir sowieso und
$a'b'' + a''b' = (a' - a'') \cdot (b'' - b') + a'b' + a''b''$.

Das Programm: $\vdots$

```
m₁ = Mult(a', b', n/2)
m₃ = Mult(a'', b'', n/2)
if (a' - a'' < 0, b''-b' < 0)
    m₂ = Mult(a''-a', b'-b'', n/2)//|a'' - a'|, |b' - b''| < 2^(n/2 - 1)
if (a' - a'' ≥ 0, b''-b' < 0)
⋮
return (m₁ · 2ⁿ + (m₂ + m₁ + m₃) · 2^(n/2) + m₃
                  └── Zeit O(n/2) ──┘
```

Damit für die Zeit $T(1) \leq d$
$T(n) \leq d \cdot n + 3 \cdot T(\frac{n}{2}) \Leftarrow O(n^{log_2 3})$.

Also Addition linear, nicht bekannt für Multiplikation.
Wir gehen jetzt wieder davon aus, dass die Basisoperationen in $O(1)$ Zeit durchzuführen sind.
Erinnern wir uns an die Matrizenmultiplikation: quadratische Matrizen.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$= \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

und allgemein haben wir bei 2 $n \times n$-Matrizen:

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & & \vdots \\ b_{n1} & \dots & b_{nn} \end{pmatrix}$$

$$= \begin{pmatrix} \sum_{i=1}^n a_{1i}b_{i1} & \sum_{i=1}^n a_{1i}b_{i2} \dots & \sum_{i=1}^n a_{1i}b_{in} \\ \vdots & & \\ \sum_{i=1}^n a_{ni}b_{i1} & \dots & \sum_{i=1}^n a_{ni}b_{in} \end{pmatrix}$$

Laufzeit ist, pro Eintrag des Ergebnisses: n Multiplikationen, $n-1$ Additionen, also $O(n)$. Bei n^2 Einträgen $O(n^3)(= O(n^2)^{\frac{3}{2}})$. Überraschend war seinerzeit, dass es besser geht mit divide-and-conquer.
Wie könnte ein divide-and-conquer-Ansatz funktionieren? Teilen wir die Matrizen einmal auf:

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ a_{21} & \dots & a_{2n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

$A_{i,j}$ sind $\frac{n}{2} \times \frac{n}{2}$ - Matrizen (wir nehmen wieder an, n 2-er-Potenz)

$$\begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & & \vdots \\ b_{n1} & \dots & b_{nn} \end{pmatrix} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$\begin{pmatrix} \sum_{i=1}^n a_{1i}b_{i1} & \dots & \sum_{i=1}^n a_{1i}b_{in} \\ \vdots & & \\ \sum_{i=1}^n a_{ni}b_{i1} & \dots & \sum_{i=1}^n a_{ni}b_{in} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Es ist

$$C_1 1 = \begin{pmatrix} \sum_{i=1}^n a_{1i}b_{i1} & \dots & \sum_{i=1}^n a_{1i}b_{i\frac{n}{2}} \\ \vdots & & \vdots \\ \sum_{i=1}^n a_{\frac{n}{2}i}b_{i1} & \dots & \sum_{i=1}^n a_{\frac{n}{2}i}b_{i\frac{n}{2}} \end{pmatrix}$$

Es ist

$$A_{11} \cdot B_{11} = \begin{pmatrix} \sum_{i=1}^{n} a_{1i} b_{i1} & \dots & \sum_{i=1}^{n} a_{1i} b_{i\frac{n}{2}} \\ \vdots & & \vdots \\ \sum_{i=1}^{n} a_{\frac{n}{2}i} b_{i1} & \dots & \sum_{i=1}^{n} a_{\frac{n}{2}i} b_{i\frac{n}{2}} \end{pmatrix}$$

Verifikation? Hauptschleife ist Nummer 3. Wie läuft diese?

$Q_0 = (s), col_0[s] =$ gr, „$col_0 =$ w“ sonst.

$\downarrow$ 1. Lauf

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

$\downarrow$

$Q_2 = (\underbrace{u_2, \ldots, u_k}_{Dist1}, \underbrace{u_{k+1}, \ldots, u_t}_{Dist2}) i\ col[u_1] =$ sch
Alle mit $Dist = 1$ entdeckt.

$\downarrow$ $k - 2$ Läufe weiter

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

$\downarrow$

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

$\downarrow$

$Q_{k+2}(\underbrace{\ldots}_{Dist=2}, \underbrace{\ldots}_{Dist=3})$
$\vdots$
$\underbrace{(\ldots)}_{Dist=3}$
und *alle* mit $Dist = 3$ erfasst (grau oder schwarz).
$\vdots$

Allgemein: Nach l-tem Lauf gilt:

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

- $Q_l = (\overbrace{\underbrace{u_1, \ldots,}_{Dist D}}^{U} \overbrace{\underbrace{\ldots}_{Dist D+1}}^{V})$

  $D_l = Dist(s, u_1), u_1$ 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 11.1 (*Induktion über l*):

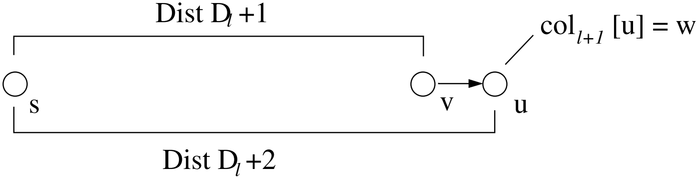
$l = 0$ (vor erstem Lauf), Gilt mit $D_0 = 0, U = (s), V = \emptyset$.
$l = 1$ (nach erstem Lauf), Mit $D_1 = 1, U =$ Nachbarn von $s, V = \emptyset$.
Gilt Invariante nach l-tem Lauf, also
$Q_l = (\overbrace{u_1, \ldots, u_k}^{Dist D_l}, \overbrace{v_1, \ldots, v_r}^{Dist D_l+1})$ mit (*), (**), $D_l = Dist(s, u_1)$.
Finde $l + 1$-ter Lauf statt (also v_1 ex.).
1. Fall: $u_1 = u_k$

- $Q_{l+1} = (v_1, \ldots, v_r \overbrace{\ldots}^{\text{Nachbarn von } u_1}), D_l + 1 = Dist(s, v_1)$

- 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:

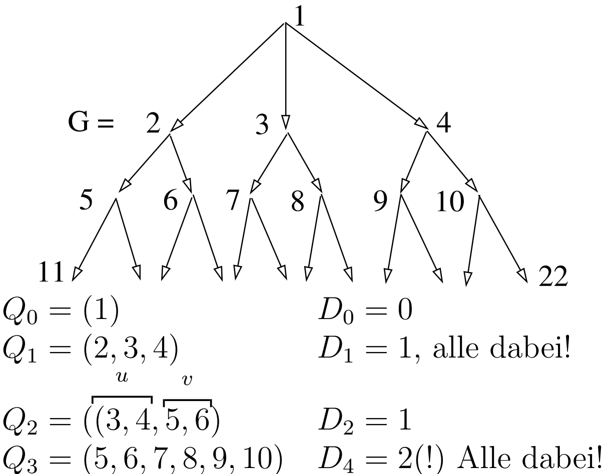


2. Fall: $u_1 \neq u_k$

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

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

Etwa:



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

- $Q = (u), D = \Delta$
- 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 kürzeste Wege s ○——►○ u durch

u ○◄—— ○ Π[u] ◄—— ○ Π [Π [u]] ◄—— ○ ... ○◄—— ○ Π [Π[..Π[u] ..]] = s.