Theoretische Informatik I

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

WS 2013/2014

Bitte beachten:

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

Version vom 13. Februar 2014

Inhaltsverzeichnis

1 Gerichtete Graphen **8**

1.1 Datenstruktur Adjazenzmatrix 12

1.2 Datenstruktur Adjazenzliste 13

1.3 Breitensuche 15

1.4 Datenstruktur Schlange 16

1.5 Algorithmus Breitensuche (Breadth first search = BFS) 17

1.6 Topologische Sortierung 24

1.7 Algorithmus Top Sort 25

1.8 Algorithmus Verbessertes Top Sort 27

2 Ungerichtete Graphen **30**

2.1 Algorithmus Breitensuche (BFS) 34

2.2 Algorithmus (Finden von Zusammenhangskomponenten) 40

3 Zeit und Platz **41**

4 Tiefensuche in gerichteten Graphen **46**

4.1 Algorithmus Tiefensuche 48

5 Starke Zusammenhangskomponenten **58**

5.1 Finden von starken Zusammenhangskomponenten in $O(|V| + |E|)$ . . 62

6 Zweifache Zusammenhangskomponenten **69**

6.1 Berechnung von l[v] 77

6.2 Algorithmus (l-Werte) 78

6.3 Algorithmus (Zweifache Komponenten) 80

7 Minimaler Spannbaum und Datenstrukturen **83**

7.1 Algorithmus Minimaler Spannbaum (Kruskal 1956) 87

7.2 Union-Find-Struktur 90

7.3 Algorithmus Union-by-Size . . . 94
7.4 Algorithmus Wegkompression . . . 98
7.5 Algorithmus Minimaler Spannbaum nach Prim (1963) . . . 101
7.6 Algorithmus (Prim mit Q in Heap) . . . 109

8 Kürzeste Wege **112**

8.1 Algorithmus (Dijkstra 1959) . . . 114
8.2 Algorithmus Floyd-Warshall . . . 118

9 Flüsse in Netzwerken **122**

9.1 Algorithmus nach Ford-Fulkerson . . . 128
9.2 Algorithmus nach Edmonds-Karp . . . 133

10 Dynamische Programmierung **139**

10.1 Längste gemeinsame Teilfolge . . . 139
10.2 Optimaler statischer binärer Suchbaum . . . 143
10.3 Kürzeste Wege mit negativen Kantengewichten . . . 150

11 Divide-and-Conquer und Rekursionsgleichungen **156**

11.1 Mergesort . . . 156
11.2 Quicksort . . . 159
11.3 Rekursionsgleichungen . . . 162
11.4 Multiplikation großer Zahlen . . . 165
11.5 Schnelle Matrixmultiplikation . . . 171

12 Kombinatorische Suche **179**

12.1 Aussagenlogische Probleme . . . 179
12.2 Aussagenlogisches Erfüllbarkeitsproblem . . . 181
12.2.1 Davis-Putnam-Prozedur . . . 185
12.2.2 Heuristiken . . . 186
12.2.3 Erfüllbarkeitsäquivalente 3-KNF . . . 187

12.2.4 Monien-Speckenmeyer . . . 191
12.2.5 Max-SAT . . . 194
12.3 Maximaler und minimaler Schnitt . . . 196
12.4 Traveling Salesman Problem . . . 198
12.4.1 Branch-and-Bound . . . 202
12.4.2 TSP mit dynamischer Programmierung . . . 209
12.5 Hamilton-Kreis und Eulerscher Kreis . . . 212

Liste der Algorithmen

1 BFS(G, s) . . . 18
2 TopSort(G) . . . 27
3 BFS(G, s) . . . 34
4 Einfache Zusammenhangskomponenten . . . 40
5 DFS(G) . . . 48
- Prozedur DFS-visit(u) . . . 48
6 Test auf starken Zusammenhang (naive Variante) . . . 60
7 Test auf starken Zusammenhang (mit modifizierter Breitensuche) . . . 61
8 Starke-Kompunenten(G) . . . 64
- Prozedur MDFS-visit(u) . . . 78
9 2fache Komponenten(G) . . . 80
- Prozedur NDFS-visit(u) . . . 81
10 Minimaler Spannbaum (Kruskal 1956) . . . 87
- Prozedur union(v, w) . . . 90
- Prozedur union(u, v) . . . 92
- Prozedur find(v) . . . 92
- Prozedur union . . . 94
- Prozedur find(v) . . . 98
11 Minimaler Spannbaum (Prim) . . . 102

- Prozedur Min(Q) . . . 106
- Prozedur DeleteMin(Q) . . . 106
- Prozedur Insert(Q, v, s) . . . 106
- Prozedur Vater(Q, i) . . . 107
- Prozedur linker Sohn(Q, i) . . . 107
- Prozedur rechter Sohn(Q, i) . . . 107

12 Prim mit Heap . . . 109

- Prozedur DecreaseKey(Q, v, s) . . . 110

13 Dijktstras Algorithmus . . . 114

14 Dijktstras Algorithmus (ohne mehrfache Berechnung derselben $D[w]$) . 116

15 Floyd-Warshall-Algorithmus . . . 120

16 Maximaler Fluss (Ford-Fulkerson) . . . 128

17 Maximaler Fluss (Edmonds-Karp) . . . 133

18 lgT(v, w) . . . 141

19 Kürzeste Wege mit Backtracking . . . 150

- Prozedur KW(W, u, v) . . . 151
- Prozedur KW(W, u, v) . . . 152
- Prozedur Setze(W, v, b) . . . 154
- Prozedur KW(V, a, b) . . . 154

20 Mergesort($A[1, \ldots, n]$) . . . 156

21 Quicksort($A[1, \ldots n]$) . . . 159

22 Quicksort(A, l, r) mit Partition . . . 159

- Prozedur Partition($A[1, \ldots n], l, r$) . . . 160

23 Multiplikation großer Zahlen . . . 166

24 Multiplikation großer Zahlen (schnell) . . . 171

25 Erfüllbarkeit . . . 181

26 Davis-Putnam-Prozedur DP(F) . . . 185

27 MoSP(F) . . . 192

28 $|E|/2$-Schnitt 196
29 Backtracking für TSP 201
30 Offizielles branch-and-bound 206
31 TSP mit dynamischer Programmierung 211

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

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

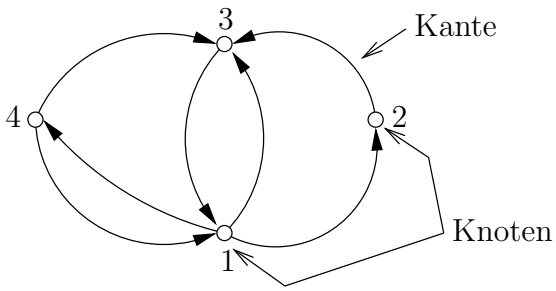
Der Vorlesung liegen die folgenden Lehrbücher zugrunde:

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

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

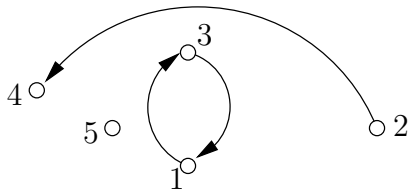
[1] Auf Deutsch im Oldenburg Verlag

1 Gerichtete Graphen

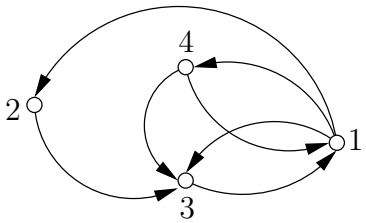


Graph G_1, gerichtet

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



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



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

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

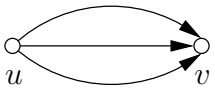
- V*, eine beliebige, endliche Menge von Knoten*
- E*, eine Menge von Kanten wobei* $E \subseteq \{(u, v) \mid u, v \in V, u \neq v\}$

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

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

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

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



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

Ein Beweis der Folgerung folgt auf Seite 11.

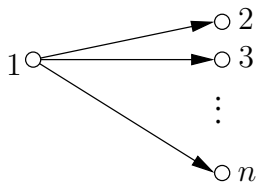
Erinnerung: *$f(n)$ ist $O(n^2)$ bedeutet: Es gibt eine Konstante $C > 0$, so dass $f(n) \leq C \cdot n^2$ für* alle *hinreichend großen n gilt.*

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

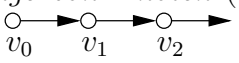
Für eine Menge M ist $\#M = |M|$ = die Anzahl der Elemente von M.

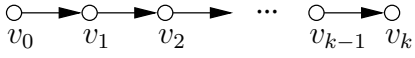
Es gilt immer:

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

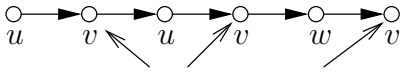


$\text{Egrad}(1) = 0, \text{Agrad}(1) = n - 1$

Definition 1.2(Weg): *Eine Folge von Knoten* $(v_0, v_1, ..., v_k)$ *ist ein Weg in* $G = (V, E)$ *gdw. (genau dann, wenn)*  *Kanten in* E *sind.*

Die Länge von  ist k. Also die Länge ist die Anzahl der Schritte. Die Länge von $v_0 = 0$.

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



ein Weg. Seine Länge ist 5.

- Ein Weg $(v_0, v_1, ..., v_k)$ ist *einfach* genau dann, wenn $|\{v_0, v_1, ..., v_k\}| = k + 1$ (d.h., alle v_i sind verschieden).
- Ein Weg $(v_0, v_1, ..., v_k)$ ist *geschlossen* genau dann, wenn $v_0 = v_k$.
- Ein Weg $(v_0, v_1, ..., v_k)$ ist ein *Kreis* genau dann, wenn:
  - $k \geq 2$ und
  - $(v_0, v_1, ..., v_{k-1})$ einfach und
  - $v_0 = v_k$

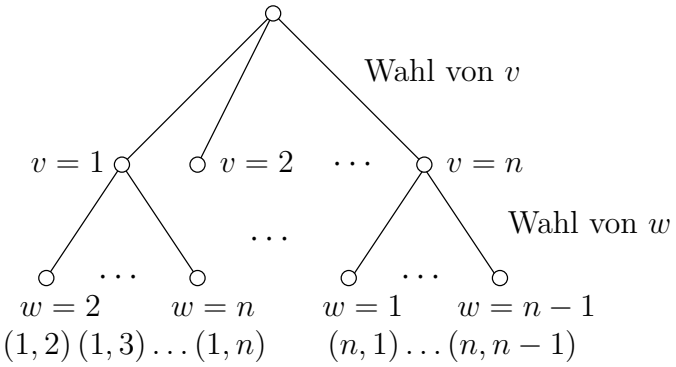
Es ist z.B. $(1, 2, 1)$ ein Kreis der Länge 2 mit $v_0 = 1, v_1 = 2$ und $v_2 = 1 = v_0$. Beachte noch einmal: Im Weg $(v_0, v_1, ..., v_k)$ haben wir $k+1$ v_i's aber nur k Schritte (v_i, v_{i+1}).

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

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

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

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



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

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

Alternative Interpretation von $n^2 - n$:

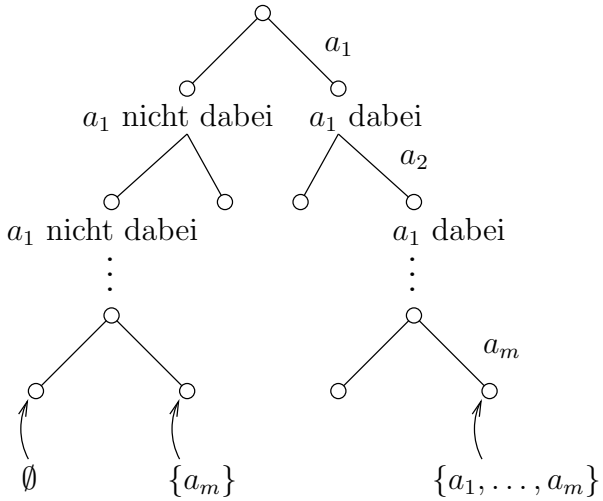
- $n^2 =$ # aller Paare (v, w), $v, w \in V$ (auch $v = w$) (Auswahlbaum)
- $n =$ # Paare (v, v) $v \in V$. Diese werden von n^2 abgezogen.

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

$$
\begin{aligned}
000 \ldots 00 &\rightarrow \emptyset \\
000 \ldots 01 &\rightarrow \{a_m\} \\
000 \ldots 10 &\rightarrow \{a_{m-1}\} \\
000 \ldots 11 &\rightarrow \{a_{m-1}, a_m\} \\
&\vdots \\
111 \ldots 11 &\rightarrow \{a_1, a_2, ..., a_m\} = M
\end{aligned}
$$

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

Noch den Auswahlbaum für die Teilmengen von M:



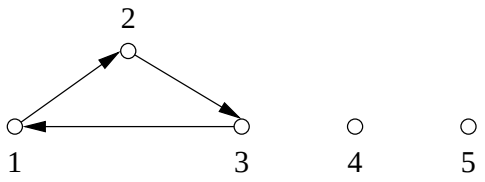
Tiefe $= m$, #Blätter $= 2^m$

1.1 Datenstruktur Adjazenzmatrix

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

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

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

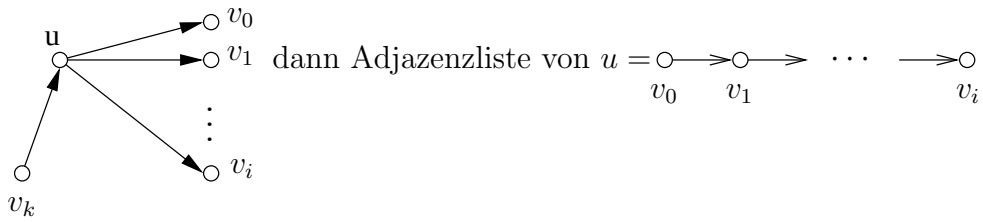


$$
A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\begin{array}{c}
\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\left(\begin{array}{ccccc}
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0
\end{array} \right)
\end{array}
$$

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

1.2 Datenstruktur Adjazenzliste

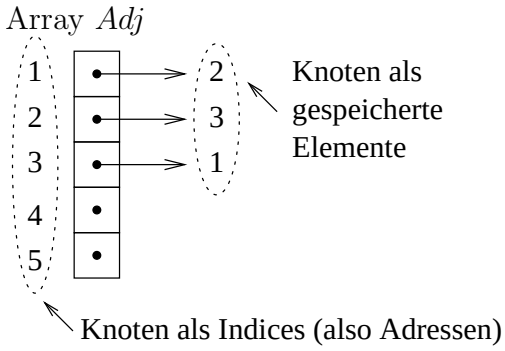
Sei $G = (V, E)$ ein gerichteter Graph. Adjazenzliste von u = Liste der direkten Nachbarn von u



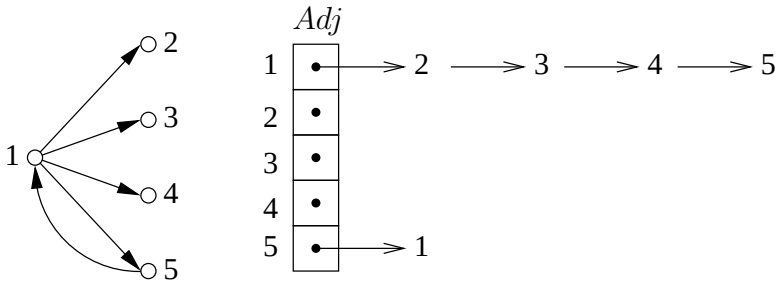
Nicht in der Adjazenzliste ist v_k, jede Reihenfolge von $v_1, ..., v_i$ erlaubt.

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

Beispiel wie oben:



ein anderes Beispiel:

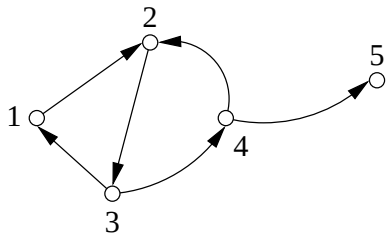


Platz zur Darstellung von $G = (V, E)$: $c + n + d \cdot |E|$ Speicherplätze mit

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

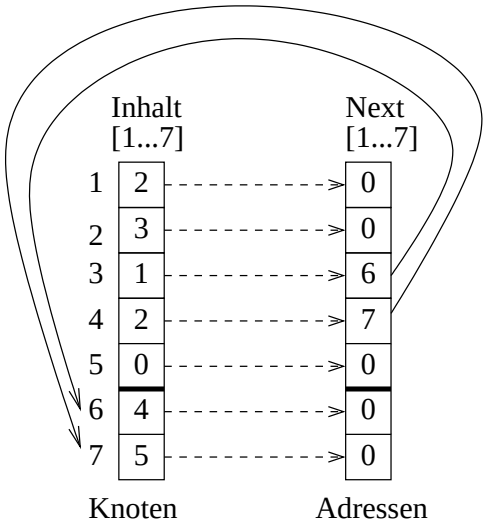


Die Größe des Arrays ist nicht unbedingt gleich der Kantenanzahl, aber sicherlich $\leq |E| + |V|$. Hier ist $A[1 \ldots 6]$ falsch. Wir haben im Beispiel $A[1 \ldots 7]$. Die Größe des Arrays ist damit $|E|$ + #Knoten vom Agrad = 0.

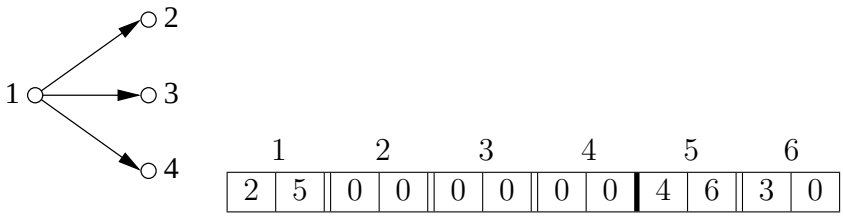
1		2		3		4		5		6		7	
2	0	3	0	1	6	2	7	0	0	4	0	5	0

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:

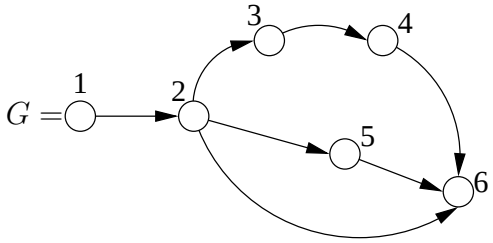


Anzahl Speicherplatz sicherlich immer $\leq 2 \cdot |V| + 2 \cdot |E|$, also $O(|V| + |E|)$ reicht aus.

1.3 Breitensuche

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

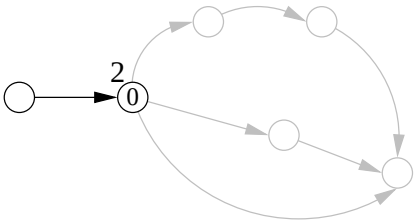
Dazu ein Beispiel:



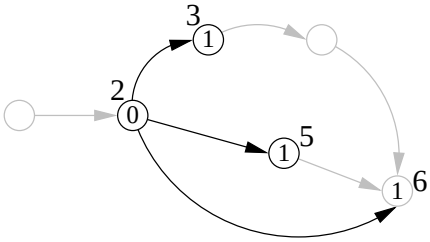
Weg von 2 nach 4 (4 von 2 erreichbar)?

Entdecktiefe

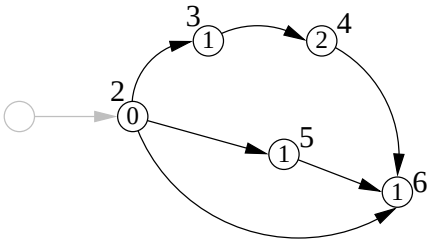
0



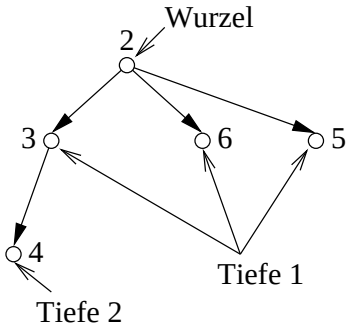
1



2



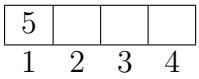
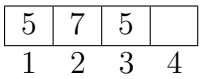
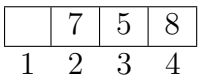
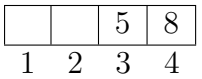
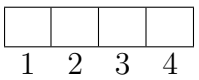
Definition 1.3(Breitensuchbaum): *Der Breitensuchbaum ist ein Teilgraph von G und enthält alle Kanten, über die entdeckt wurde:*



Die Breitensuche implementiert man zweckmäßigerweise mit einer Schlange.

1.4 Datenstruktur Schlange

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

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

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

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

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

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

Startknoten rein:

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 Entdecktiefe 1, 4 Entdecktiefe 2.

6 raus nichts rein:

1	2	3	4	5	6
			5	4	

$head = 4$, $tail = 2$

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

1.5 Algorithmus Breitensuche (Breadth first search = BFS)

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

Datenstrukturen:

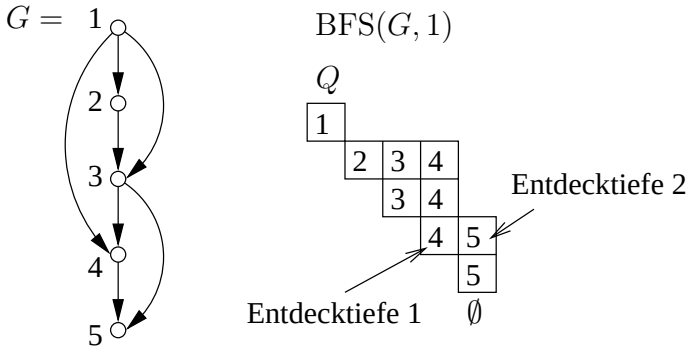
- Schlange $Q = Q[1..n]$ mit *head* und *tail* für entdeckte, aber noch nicht untersuchte Knoten.
- Array $col[1..n]$, um zu merken, ob Knoten bereits entdeckt wurde.

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

Algorithmus 1: BFS(G, s)

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

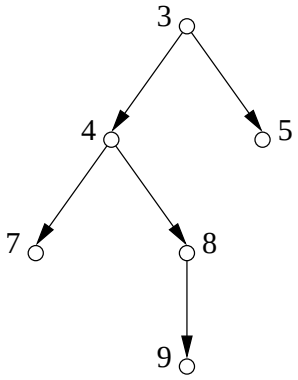
Noch ein Beispiel:



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

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

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



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

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

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

$\downarrow$

1. Lauf

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

$\downarrow$

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

Alle mit $Dist = 1$ entdeckt.

$\downarrow$

$k - 2$ 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).

$\downarrow$

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

$\downarrow$

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

$\downarrow$

Allgemein: Nach l-tem Lauf gilt:

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

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

Beweis. Induktion über l.

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

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

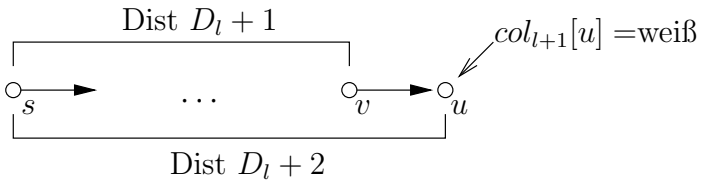
Gilt Invariante nach l-tem Lauf, also

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

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

1. Fall: $u_1 = u_k$

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

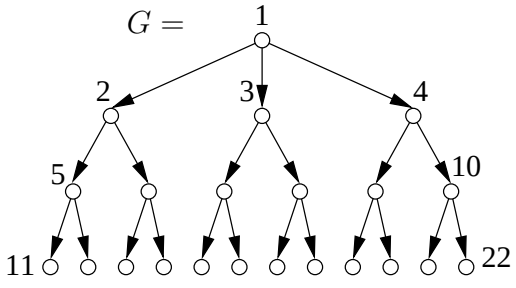


2. Fall: $u_1 \neq u_k$

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

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

Etwa:



$Q_0 = (1)$ $\quad D_0 = 0$

$Q_1 = (2, 3, 4)$ $\quad D_1 = 1$, alle dabei.

$Q_2 = (\overbrace{3, 4}^{u}, \overbrace{5, 6}^{v})$ $\quad D_2 = 1$

$Q_3 = (5, 6, 7, 8, 9, 10)$ $\quad D_4 = 2$(!) Alle dabei.

□

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

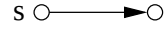
- $Q = (u), D = \Delta$
- Alle mit $Dist \leq \Delta$ erfasst.
- Alle mit $\Delta + 1 =$ weiße Nachbarn von u.

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

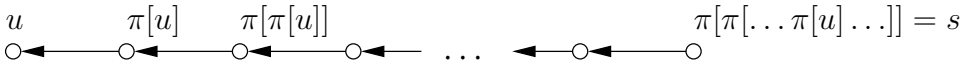
Termination: Pro Lauf ein Knoten aus Q, schwarz, kommt nie mehr in Q. Irgendwann ist Q zwangsläufig leer.

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

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

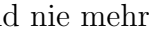
Beachte: Dann ist der kürzeste Wege 

durch



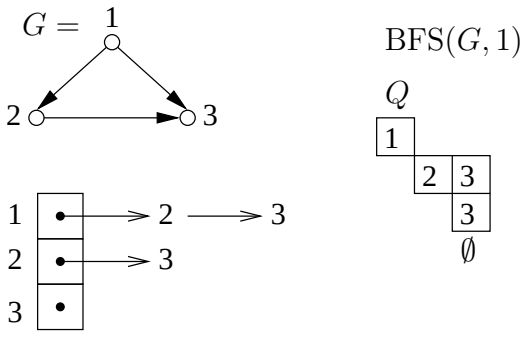
bestimmt.

Beobachtung

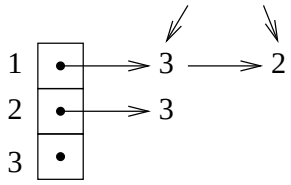
In BFS(G, s) wird jede *„von s erreichbare Kante“* (Kante (u, v), wobei u von s erreichbar ist) genau einmal untersucht. Denn wenn 

, dann ist u grau, v irgendwie, danach ist u schwarz und nie mehr grau.

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

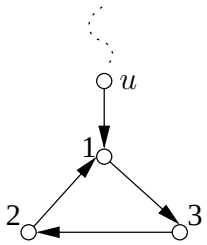


Beim Gang durch $(2, 3)$ ist 3 grau.



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

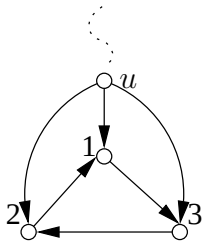
Weiterer Nutzen der Breitensuche? – Kreise finden?



in G

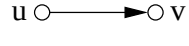
1 ist der erste Knoten, der auf dem Kreis entdeckt wird. Wenn 2 expandiert wird, ist 1 schwarz.

Auch möglich:



in G

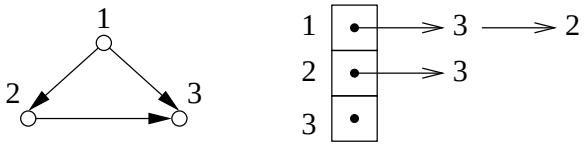
Alle drei Knoten $1, 2, 3$ werden bei der Expansion von u aus das erste Mal entdeckt. Trotzdem: Es gibt eine Kante, die bei Bearbeitung auf einen schwarzen Knoten führt, hier $(2, 1)$.

Folgerung 1.2: *Kreis* $\Longrightarrow$ *Es gibt eine Kante* 

, so dass v schwarz ist, wenn 

gegangen wird.

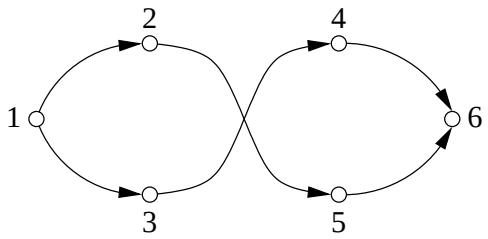
Ist ein Kreis daran erkennbar? – Leider nein, denn



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

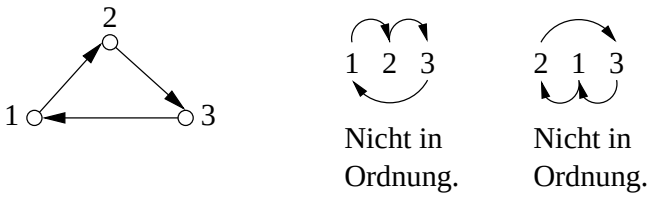
1.6 Topologische Sortierung

Wie erkenne ich kreisfreie gerichtete Graphen?



Anordnung der Knoten:

Alle Kanten gemäß der Ordnung. Aber bei Kreis:



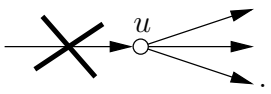
Definition 1.4(Topologische Sortierung): *Ist $G = (V, E)$ ein gerichteter Graph. Eine topologische Sortierung ist eine Anordnung von V als $(v_1, v_2, v_3, \ldots, v_n)$, so dass gilt:*

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

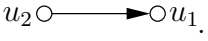
Satz 1.1: *G hat topologische Sortierung $\Longleftrightarrow$ G hat keinen Kreis.*

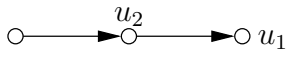
Beweis. „$\Rightarrow$“ Sei also $(v_1, v_2, \ldots, v_n)$ topologische Sortierung von G. Falls Kreis $(w_0, w_1, \ldots, w_0)$, dann mindestens eine Kante der Art v_j ○⟶○ v_i mit $i < j$.

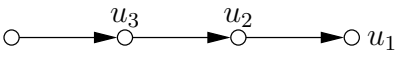
„$\Leftarrow$“ Habe also G keinen Kreis. Dann gibt es Knoten u mit $\text{Egrad}(u) = 0$. Also



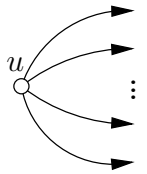
Wieso? Nehme irgendeinen Knoten u_1. Wenn Egrad$(u_1) = 0$, dann ✓.

Sonst 

Nehme u_2 her. Egrad$(u_2) = 0$ ✓, sonst zu u_3 in G: 

Falls $Egrad(u_3) = 0$, dann ✓, sonst 

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 → gibt topologische Sortierung.

Formal: Induktion über $|V|$. □

1.7 Algorithmus Top Sort

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

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

Vorgehensweise:

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

   ... Wie oben.

Wie findet man u gut?

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

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

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

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

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

   ...

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

1.8 Algorithmus Verbessertes Top Sort

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

Ausgabe: Wie bei Top Sort.

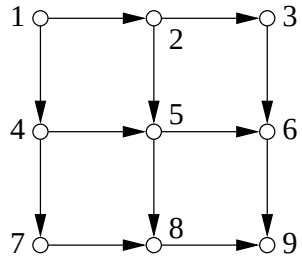
Weitere Datenstrukturen:

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

Algorithmus 2: TopSort(G)

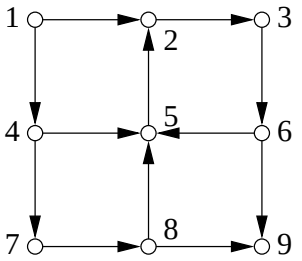
```
   /* 1. Initialisierung */
 1 Egrad[v] auf 0, Q = ();
 2 foreach v ∈ V do
 3 |  Gehe Adj[v] durch;
 4 |  zähle für jedes gefundene u Egrad[u] = Egrad[u] + 1;
 5 end
   /* 2. Einfügen in Schlange */
 6 foreach u ∈ V do
 7 |  if Egrad[u] = 0 then
 8 |  |  Q=enqueue(Q, u);
 9 |  end
10 end
   /* 3.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

Ein Beispiel:



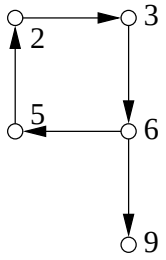
Q	$(v[1], \dots)$
1	$\emptyset$
2, 4	1
4, 3	1, 2
3, 7, 5	1, 2, 3
$\vdots$	

Noch ein Beispiel:



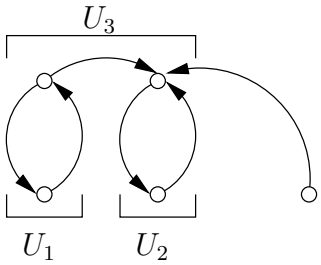
Q	$(v[1], \dots)$
1	$\emptyset$
4	1
7	1, 4
8	1, 4, 7
$\emptyset$	1, 4, 7, 8

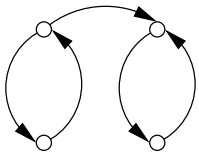
Es verbleibt als Rest:



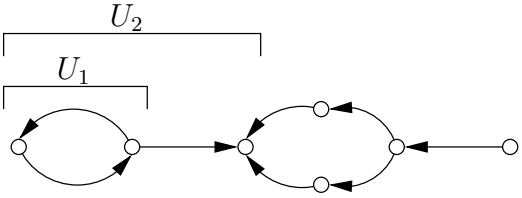
Wir betrachten einen Graphen $G = (V, E)$. Für $U \subseteq V$ ist $G_U = (U, F), F = \{(u, v) \mid u, v \in U \text{ und } (u, v) \in E\}$ der auf U induzierte Graph. Uns interessieren die $U \subseteq V$, so dass in G_U für jeden Knoten Egrad ≥ 1 ist.

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



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

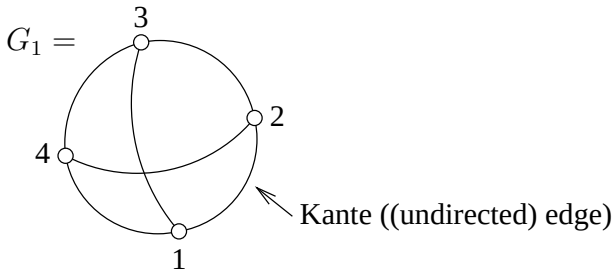
Größter induzierter Teilgraph, wobei jeder $Egrad \geq 1$ ist.



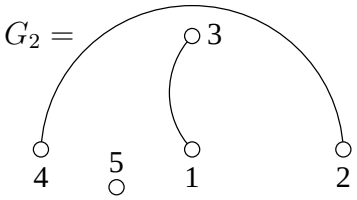
$G_{U_1 \cup U_2} = G_{U_2}$

2 Ungerichtete Graphen

Ein typischer ungerichteter Graph:



Auch einer:

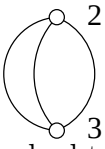


Ungerichteter Graph G_2 mit $V = \{1, 2, 3, 4, 5\}$ und $E = \{\{4, 2\}, \{1, 3\}\}$.

Beachte:

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

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

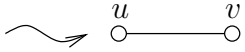


$(3, 2)_1, (3, 2)_2, (3, 2)_3 \in E$ und 

$(3, 3)$ hier nicht erlaubt.

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

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

Schreibweise $G = (V, E)$, $\{u, v\}$ 

.

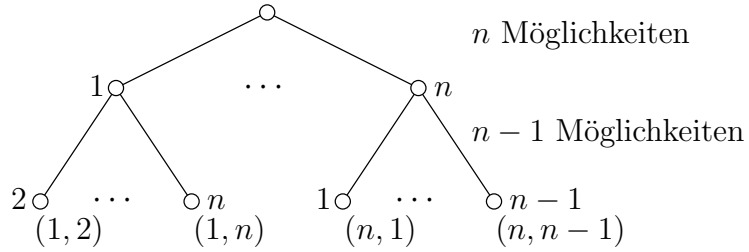
Folgerung 2.1: *Ist* $|V| = n$ *fest, dann:*

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

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

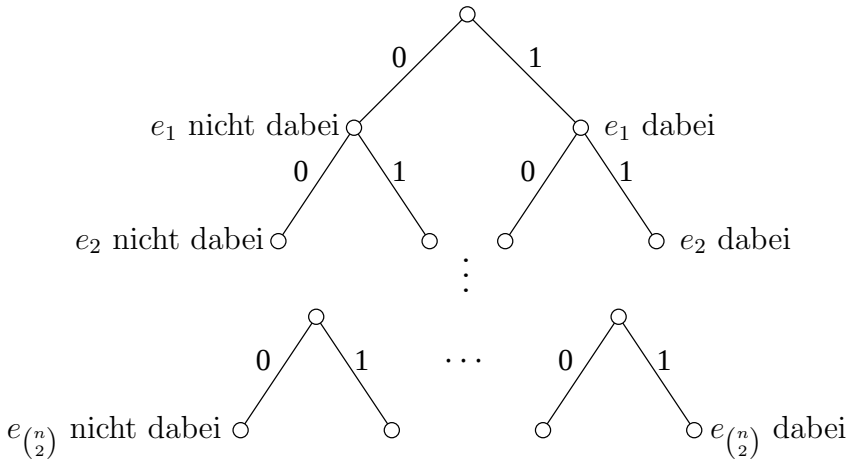
Beweis.

a) „Auswahlbaum“



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

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



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

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

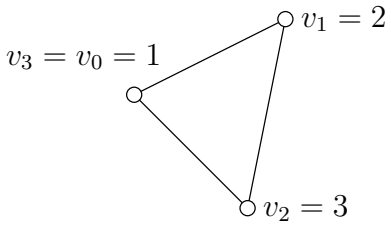
Beachte: $\binom{n}{2}$ ist $O(n^2)$.

□

Adjazent, inzident sind analog zum gerichteten Graph zu betrachten.

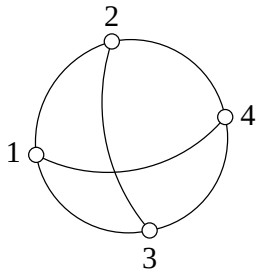
- Grad$(u) = \{\{u, v\} \mid \{u, v\} \in E\}$, wenn $G = (V, E)$. Es gilt immer $0 \leq Grad(u) \leq n - 1$.
- Weg $(v_0, v_1, \ldots, v_k)$ wie im gerichteten Graphen.
- Länge $(v_0, v_1, \ldots, v_k) = k$
- Auch ein Weg ist bei $\{u, v\} \in E$ der Weg (u, v, u, v, u, v). Einfach $\Longleftrightarrow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

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



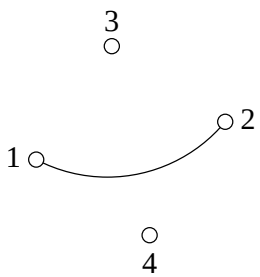
  ein solcher mit $v_0 = 1, v_1 = 2, v_2 = 3, v_3 = 1 = v_0$.

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



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

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

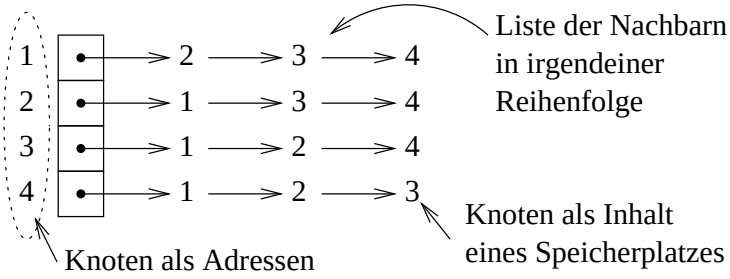


G_2	1	2	3	4
1	0	1	0	0
2	1	0	0	0
3	0	0	0	0
4	0	0	0	0

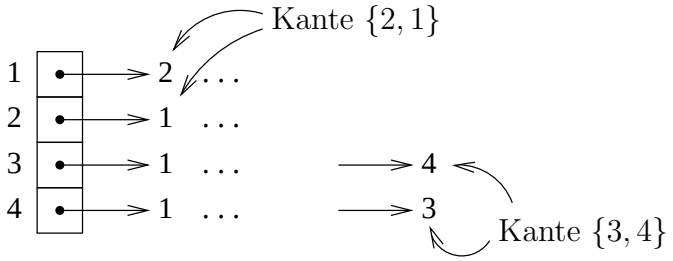
Darstellung als Array: n^2 Speicherplätze.

Adjazenzlistendarstellung:

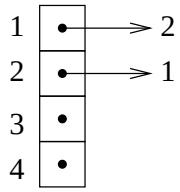
Array Adj von G_1



Jede Kante zweimal dargestellt, etwa:

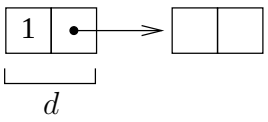


Adjazenzlistendarstellung von G_2:



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.

2.1 Algorithmus Breitensuche (BFS)

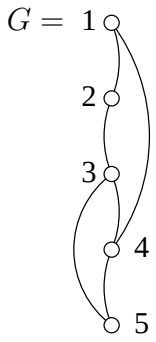
BFS(G, s) genau wie im gerichteten Fall.

Eingabe: G in Adjazenzlistendarstellung

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

Algorithmus 3: BFS(G, s)

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



BFS$(G, 1)$

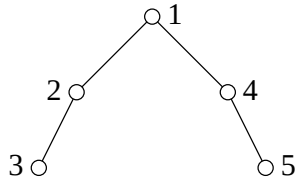
Q				
1				
	2	4		$\{1,2\}$ als $(1,2)$, $col[2]$ =w
		4	3	$\{1,2\}$ als $(2,1)$, $col[1]$ =s
			3	5 $\{4,3\}$, $col[3]$ =g
				5 $\{4,3\}$, $col[4]$ =g
				$\emptyset$

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 $col[u] = s$.

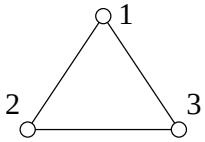
Breitensuchbaum durch $\pi[1, \dots, n]$, Vaterarray. Hier



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

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

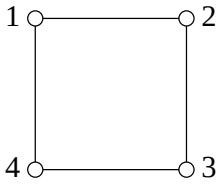
Wir können im ungerichteten Fall erkennen, ob ein Kreis vorliegt:



BFS$(G, 1)$

Q
1
2 3
3

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



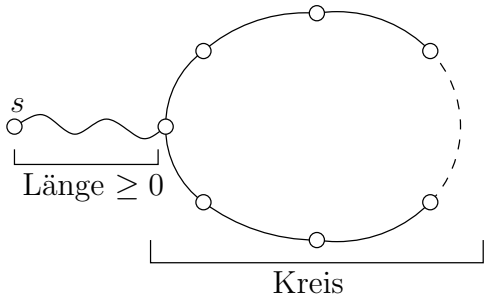
BFS$(G, 2)$

Q
2
1 3
3 4
4

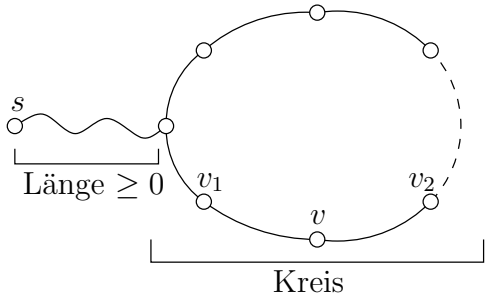
Beim (ersten) Gang durch $\{3, 4\}$ ist $col[4] = grau$, d.h. $4 \in Q$.

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

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



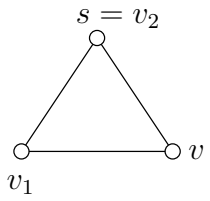
Sei v der Knoten auf dem Kreis, der als *letzter(!)* entdeckt wird.



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

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

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



BFS(G, s)

Q

s

$v_1 \quad v \quad \leftarrow v$ als letzter

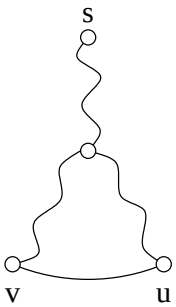
v

$\emptyset$

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

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

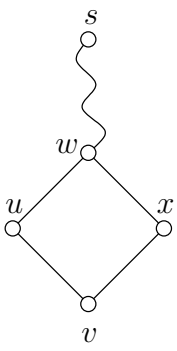
Dann im Breitensuchbaum der Suche:



Also einen Kreis. Etwas formaler:

Haben Wege $(v_0 = s, \ldots, v_k = v)$, $(w_0 = s, \ldots, w_k = u)$ 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 folgenden Fall.



BFS(G, s)

Q
s
$\vdots$
w $\ldots$
$\ddots$ x u
u v $\leftarrow v$ 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.

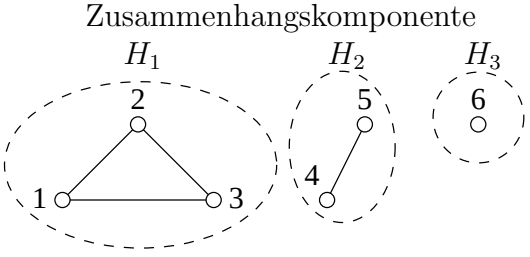
Definition 2.2(Zusammenhang): *Sei $G = (V, E)$ ungerichteter Graph.*

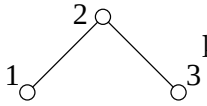
a) G heißt zusammenhängend, genau dann, wenn es für alle $u, v \in V$ einen Weg (u, v) gibt.

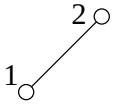
b) Sei $H = (W, F)$, $W \subseteq V, F \subseteq E$ ein Teilgraph. H ist eine Zusammenhangskomponente von G, genau dann, wenn

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

Man sagt dann: H ist ein maximaler zusammenhängender Teilgraph.



Beachte: In obigem Graphen ist 

keine Zusammenhangskomponente, da $\{1, 3\}$ fehlt. Ebenso wenig ist es 

, da nicht maximaler Teilgraph, der zusammenhängend ist.

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

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

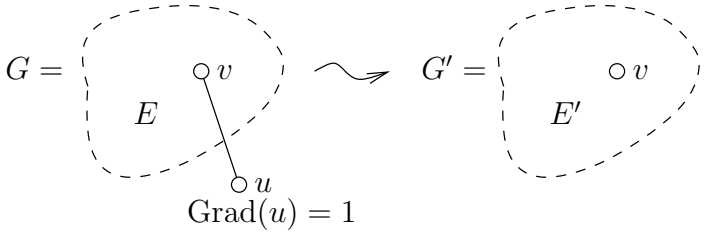
Beweis. „$\Rightarrow$“ Induktion über $n = |V|$.

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

$n = 2$, dann $\circ\!\!-\!\!\!-\!\!\circ$, also ✓.

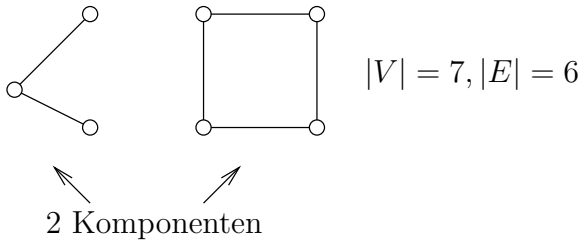
Sei $G = (V, E), |V| = n + 1$ und ohne Kreis. Da G zusammenhängend ist und ohne Kreis, gibt es Knoten vom $Grad = 1$ in G.

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

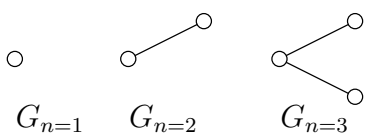


Dann gilt für $G' = (V', E')$: G hat keinen Kreis $\Rightarrow$ G' hat keinen Kreis $\Rightarrow$ Induktionsvoraussetzung $|E'| = |V'| - 1 \Rightarrow |E| = |V| - 1$.

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



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



und die Behauptung gilt.

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

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

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

Folgerung 2.2:

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

b) Besteht $G = (V, E)$ aus k Zusammenhangskomponenten, so gilt:
G ist kreisfrei $\Longleftrightarrow |E| = |V| - k$

Beweis.

a) Baum kreisfrei und zusammenhängend.

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

Beachte eben $G =$  . $|V| = 6, |E| = 3 < |V|$ trotzdem hat G einen Kreis.

□

2.2 Algorithmus (Finden von Zusammenhangskomponenten)

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

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

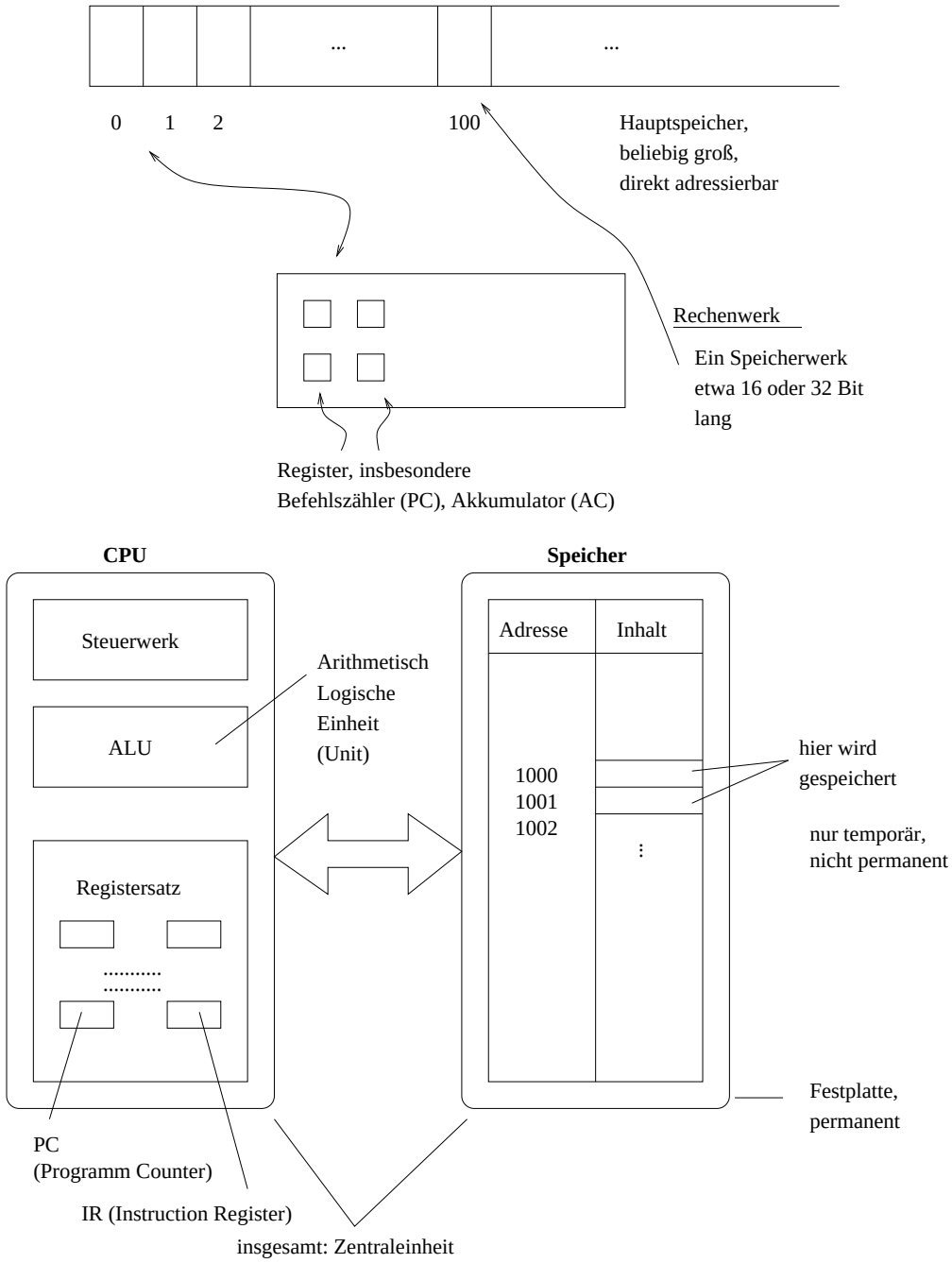
(Frage: Ist u von v erreichbar? In einem Schritt beantwortbar.)

Algorithmus 4: Einfache Zusammenhangskomponenten

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

3 Zeit und Platz

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



Typische Befehle der Art:

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

Programme liegen im Hauptspeicher.

Simulation von Schleifen durch Sprungbefehle, etwa:

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

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

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

Weitere Konventionen sind etwa:

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

Listing 1: Java-Programm zum Primzahltest

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

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

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

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

if (c%d){...}

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

d++

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

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

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

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

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

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

Ausführung eines Maschinenbefehls: Im Nanosekundenbereich

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

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

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

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

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

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

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

Zur Laufzeit unseres Algorithmus BFS(G, s). (Seite 18)

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

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

Also: while-Schleife einmal:

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

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

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

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

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

Regel: Die Multiplikationsregel für geschachtelte Schleifen:

Schleife 1 $\leq n$ Läufe, Schleife 2 $\leq m$ Läufe,

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

Betrachten nun das Topologische Sortieren von Seite 25.

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

	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:	$O(m) + O(n) + O(1)$
$\vdots$		

Nach Seite 27 bekommen wir aber Linearzeit heraus:

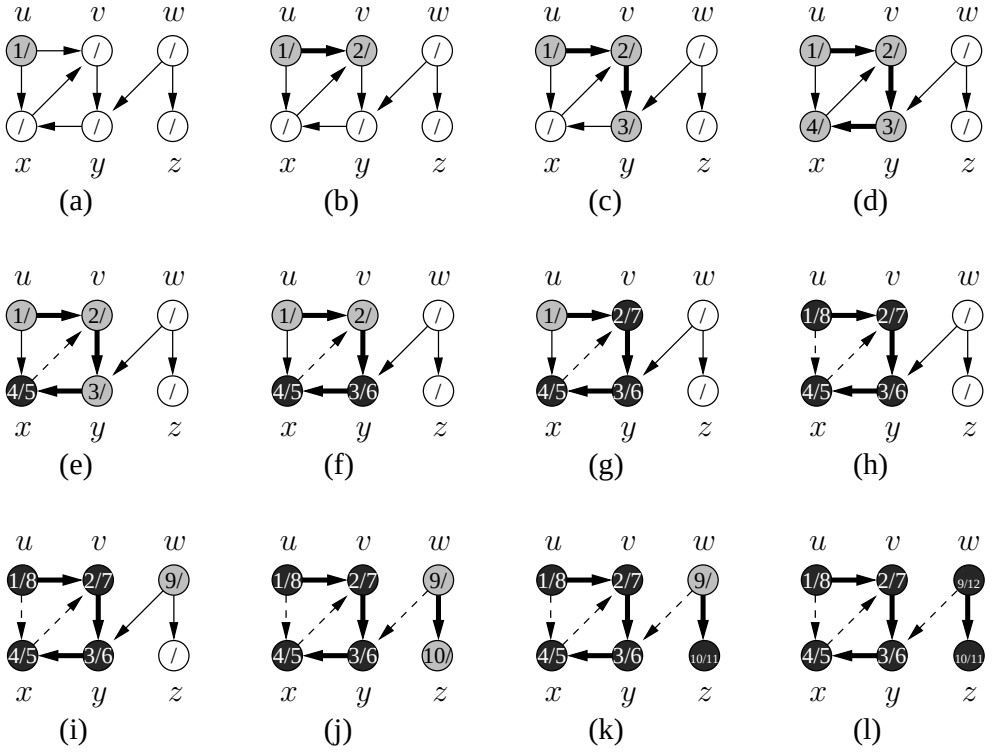
1. und 2. Egrad ermitteln und Q setzen:	$O(m)$
3. Ein Lauf durch 3. $O(1)$! (keine Schleifen), insgesamt n Läufe. Also:	$O(n)$
4. insgesamt	$O(n)$
5. jede Kante einmal, also insgesamt	$O(m)$

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

4 Tiefensuche in gerichteten Graphen

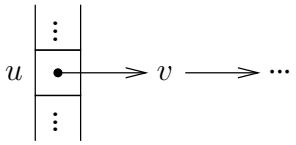
Wir betrachten zunächst das folgende Beispiel.

Beispiel 4.1:



(a) Wir fangen bei Knoten u zum Zeitpunkt 1 an,

(b) gehen eine Kante. Dabei entdecken wir den Knoten v zum Zeitpunkt 2. Der Knoten v ist der erste Knoten auf der Adjazenzliste von u.

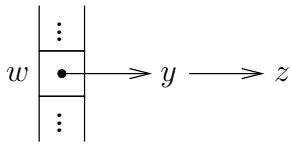


(c) Dann wird y entdeckt. Die Knoten u, v, y sind „offen“ = grau.

(d) Danach entdecken wir das x. Kante (x, v) wird zwar gegangen, aber v nicht darüber entdeckt.

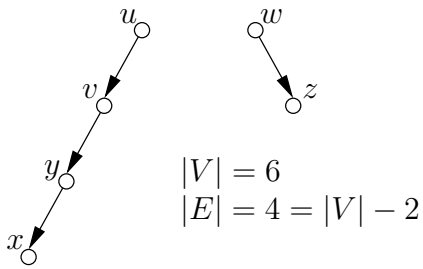
(e) - (h) Die Knoten y, v und u werden „geschlossen“. Es werden keine neuen Knoten mehr darüber entdeckt.

(i) Dann bei w weiter. Adjazenzliste von w ist:



(j) - (l) Wir finden noch den Knoten z.

Der *Tiefensuchwald* besteht aus den Kanten, über die die Knoten entdeckt wurden.



Ein *Wald* ist eine Menge von Bäumen. Die Darstellung erfolgt wie bei der Breitensuche über ein Vaterarray.

$$\pi[x] = y, \pi[y] = v, \pi[v] = u, \pi[u] = u \quad \text{(alternativ } \texttt{nil}\text{)},$$
$$\pi[z] = w, \pi[w] = w \quad \text{(alternativ } \texttt{nil}\text{)}.$$

4.1 Algorithmus Tiefensuche

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

$d[1 \dots n]$	Entdeckzeit (discovery)
	Nicht Entfernung wie vorher. Knoten wird grau.
$f[1 \dots n]$	Beendezeit (finishing), Knoten wird schwarz.
$\pi[1 \dots n]$	Tiefensuchwald, wobei $\pi[u] = v \iff v \circ\!\!\rightarrow\!\!\circ\, u$ im Baum.
	$\pi[u]$ = „Knoten, über den u entdeckt wurde“
$col[1 \dots n]$	aktuelle Farbe

Algorithmus 5: DFS(G)

```
   /* 1. Initialisierung */
 1 foreach u ∈ V do
 2 |   col[u] = weiß;
 3 |   π[u] = nil;
 4 end
 5 time = 0;
   /* 2. Hauptschleife. */
 6 foreach u ∈ V do
   |   /* Aufruf von DFS-visit nur, wenn col[u] = weiß */
 7 |   if col[u] == weiß then
 8 |   |   DFS-visit(u);
 9 |   end
10 end
```

Prozedur DFS-visit(u)

```
 1 col[u] = grau;                          /* Damit ist u entdeckt */
 2 d[u] = time;
 3 time = time + 1;
 4 foreach v ∈ Adj[u] do                     /* u wird bearbeitet */
 5 |   if col[v] == weiß then                 /* (u,v) untersucht */
 6 |   |   π[v] = u;                              /* v entdeckt */
 7 |   |   DFS-visit(v);
 8 |   end
 9 end
   /* Die Bearbeitung von u ist hier zu Ende. Sind alle Knoten aus
      Adj[u] grau oder schwarz, so wird u direkt schwarz. */
10 col[u] = schwarz;
11 f[u] = time;
   /* Zeitzähler geht hoch bei Entdecken und Beenden */
12 time = time + 1;
```

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

Einmal:

$$d[u] = 1, d[v] = 2, d[w] = 3 \quad \text{und} \quad f[w] = 4, f[v] = 5, f[u] = 6.$$

Aber auch:

$$d[w] = 1, f[w] = 2, d[v] = 3, f[v] = 4, d[u] = 5, f[u] = 6.$$

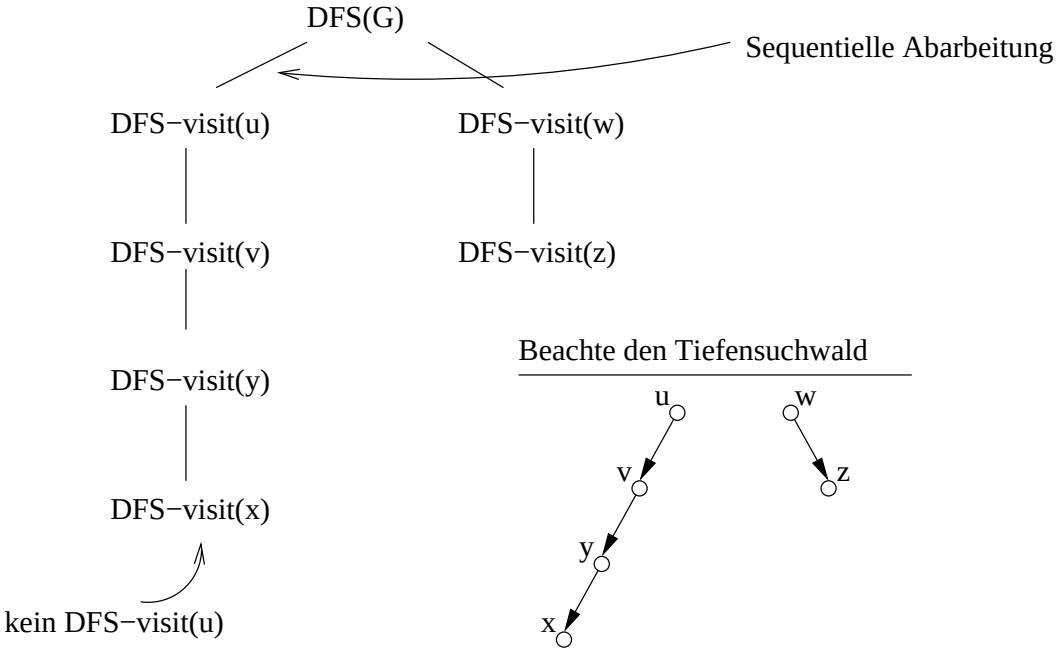
DFS-visit(u) wird nur dann aufgerufen, wenn $col[u]$ = weiß ist. Die Rekursion geht nur in weiße Knoten.

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*:

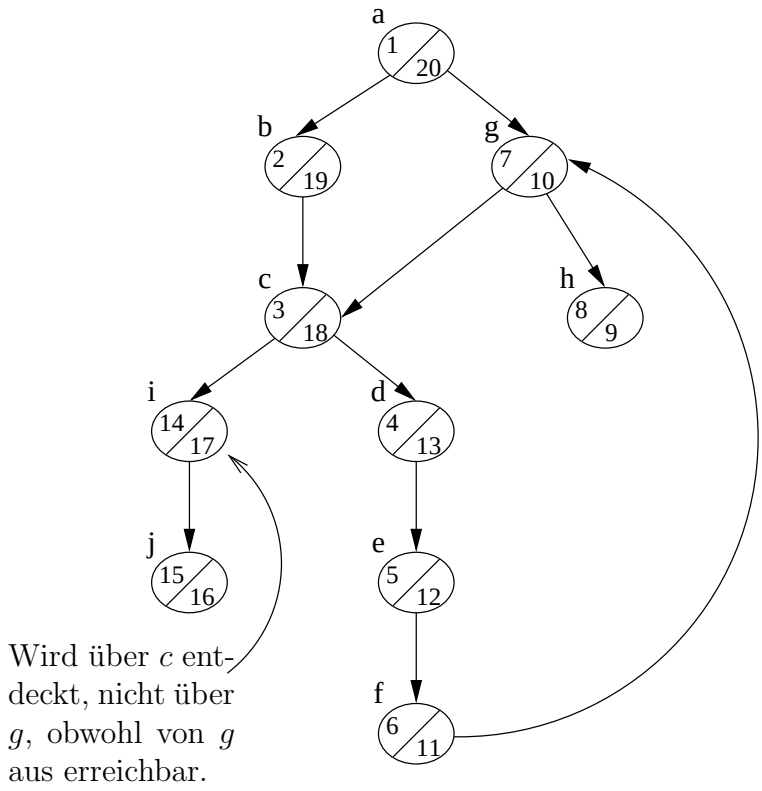


Erzeugung: Präorder(Vater vor Sohn)

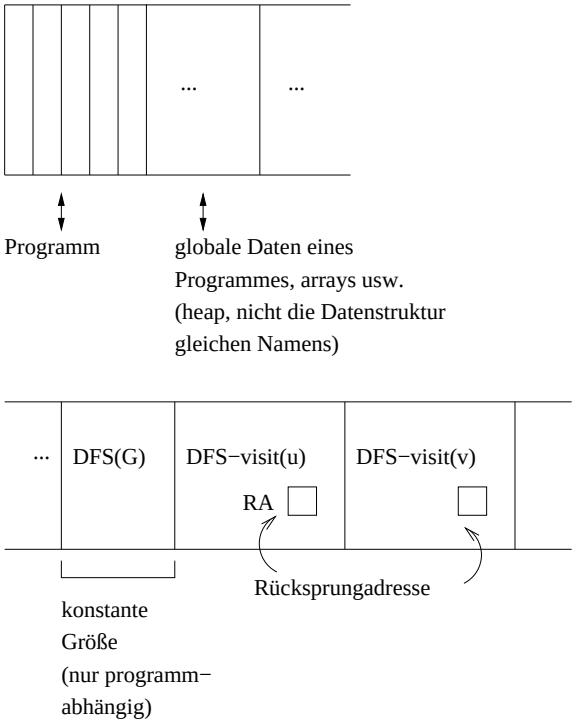
Beendigung: Postorder(Sohn vor Vater)

Wir betrachten wir nun noch einmal folgendes Beispiel:

Beispiel 4.2:



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

Beweis. Für die Hauptprozedur erhalten wir:

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.

Für DFS-visit erhalten wir über alle Aufrufe hinweg gesehen:

Zeilen	Laufzeit
1 - 3	Jeweils $O(1)$, insgesamt $O(n)$.
4 - 9	Ohne Zeit in DFS-visit(v) insgesamt $O(m)$.
5	Für jede Kante einmal $O(1)$ also insgesamt $O(m)$.
6, 7	Für jedes Entdecken $O(1)$, also $O(n)$ insgesamt.
10 - 12	$O(n)$ insgesamt.

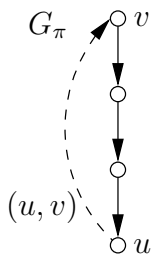
Also tatsächlich $O(n + m)$ □

Definition 4.1(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.*

Definition 4.2(Klassifizieren der Kanten von G): *Sei $(u, v) \in E$ eine Kante im Graphen. Beim Durchlaufen des Graphen mittels Tiefensuche kann diese Kante eine sogenannte* Baumkante, Rückwärtskante, Vorwärtskante *oder* Kreuzkante *sein.*

(a) (u, v) *Baumkante* $\iff (u, v) \in E_\pi$ *(d.h. $\pi[v] = u$)*

(b) (u, v) *Rückwärtskante* $\iff (u, v) \notin E_\pi$ *und v Vorgänger von u in G_π (d.h. $\pi[\pi[\ldots\pi[u]\ldots]] = v$)*

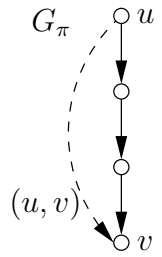


Knoten v ist grau, wenn (u, v) gegangen wird.

$$d[v] < d[u] \quad < \quad f[u] < f[v]$$

↖ *Hier wird (u, v) gegangen.*

(c) (u,v) *Vorwärtskante* $\Longleftrightarrow (u,v) \notin E_\pi$ *und* v *Nachfolger von* u *in* G_π

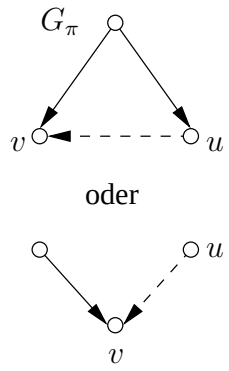


Knoten v *ist schwarz, wenn* (u,v) *gegangen wird.*

$$d[u] \; < d[v] < f[v] < \; f[u]$$

Hier (u,v). *Hier nicht* (u,v).

(d) (u,v) *Kreuzkante* $\Longleftrightarrow (u,v) \notin E_\pi$ *und* u *weder Nachfolger noch Vorgänger von* v *in* G_π



Knoten v *ist schwarz, wenn* (u,v) *gegangen wird.*

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

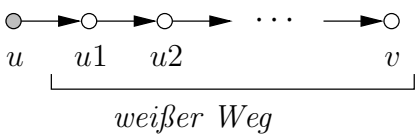
Noch eine Beobachtung für Knoten u, v. Für die Intervalle, in denen die Knoten aktiv (offen, grau) sind. gilt:

Entweder $d[u] < d[v] < f[v] < f[u]$, d.h im Prozeduraufrufbaum ist

$$\text{DFS-visit}(u) \to \cdots \to \text{DFS-visit}(v).$$

Oder es gilt $d[u] < f[u] < d[v] < f[v]$ oder umgekehrt. Das folgt aus Kellerstruktur des Laufzeitkellers.

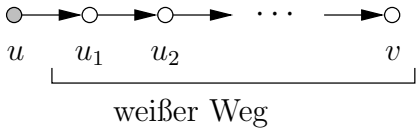
Satz 4.2(Weißer-Weg-Satz): *Der Knoten* v *wird über* u *entdeckt. (d.h. innerhalb von DFS-visit(u) wird DFS-vist(v) aufgerufen)* $\Longleftrightarrow$ *Zum Zeitpunkt* $d[u]$ *gibt es einen Weg*



in G.

Beweis. „⇒“ Aufrufstruktur, wenn DFS-visit(v) aufgerufen wird:

$$\text{DFS-visit}(u) \to \text{DFS-visit}(u_1) \to \text{DFS-visit}(u_2) \to \cdots \to \text{DFS-visit}(v)$$



„⇐“ Liege also zu $d[u]$ der weiße Weg $u \to v_1 \to v_2 \to \cdots \to v_k \to v$ vor. Das Problem ist, dass dieser 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

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

$$\cdots$$

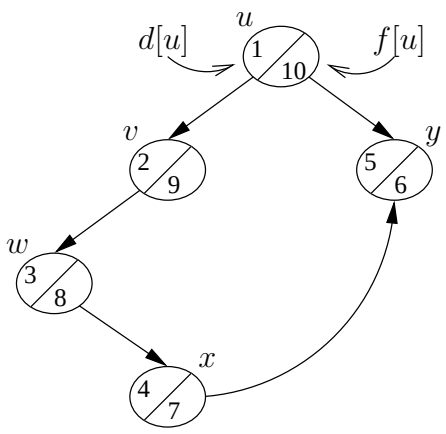
dann

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

was dem Programm widerspricht. □

Noch ein Beispiel:

Beispiel 4.3:

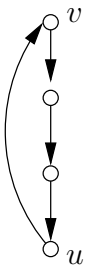


Zum Zeitpunkt 1 ist (u, y) ein *weißer Weg*. Dieser wird nicht gegangen, sondern ein anderer. Aber y wird in jedem Fall über u entdeckt!

Mit der Tiefensuche kann man feststellen, ob ein (gerichteter) Graph einen *Kreis* enthält.

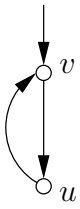
Satz 4.3: *Ist $G = (V, E)$ gerichtet, G hat Kreis $\Longleftrightarrow$ DFS(G) ergibt eine Rückwärtskante.*

Beweis. „⇐“ Sei (u, v) eine Rückwärtskante. Dann herrscht in G_π, dem Tiefensuchwald folgende Situation:



Rückwärtskante (u, v)

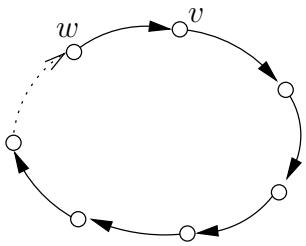
oder auch



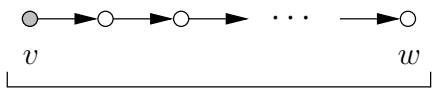
Rückwärtskante (u, v)

Also Kreis in G.

„⇒“ Hat G einen Kreis, dann also



Sei v der erste Knoten auf dem Kreis, den DFS(G) entdeckt. Dann existiert zum Zeitpunkt $d[v]$ der folgende *weiße Weg*.



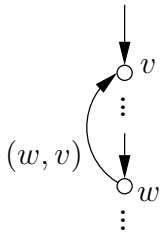
Der Kreis herum.

□

Satz 4.4(Weißer-Weg-Satz):

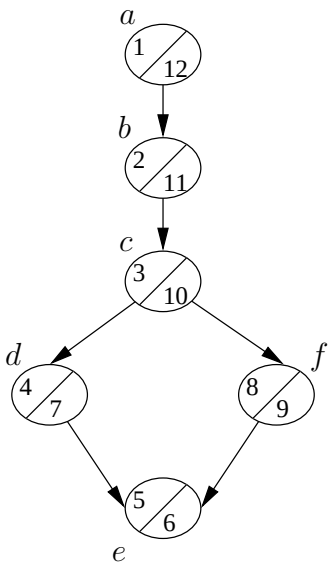
$$d[v] < d[w] < f[w] < f[v]$$

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



Der Knoten 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 der Graph kreisfrei ist.

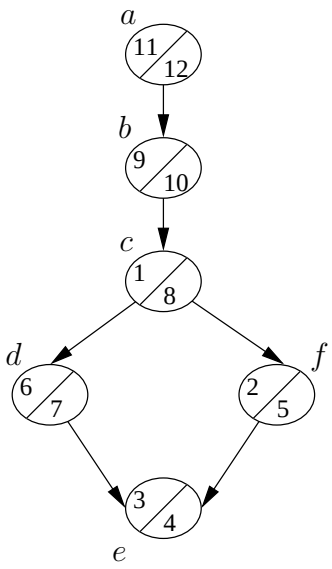


Nach absteigender Beendezeit sorteieren.

$$a, b, c, f, d, e$$

größte Beendezeit ↗ ↖ kleinste Beendezeit

Aber auch



Absteigende Beendezeit: a, b, c, d, f, e.

Satz 4.5: *Sei $G = (V, E)$* kreisfrei. *Lassen wir nun DFS(G) laufen, dann gilt für alle $u, v \in V, u \neq v$, dass*

$f[u] < f[v] \Rightarrow (u, v) \notin E$. *Keine Kante geht von kleinerer nach größerer Beendezeit.*

Beweis. 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 $col[u] = weiß$. 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:

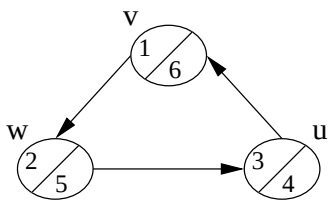
$$d[v] < f[v] < d[u] < f[u] \quad \text{und es ist} \quad f[v] < f[u]$$

□

Bemerkung zum Beweis: Wir wollen eine Aussage der Art $A \Rightarrow B$ zeigen. Wir zeigen aber statt dessen $\neg B \Rightarrow \neg A$. Diese Beiden Aussagen sind äquivalent, denn

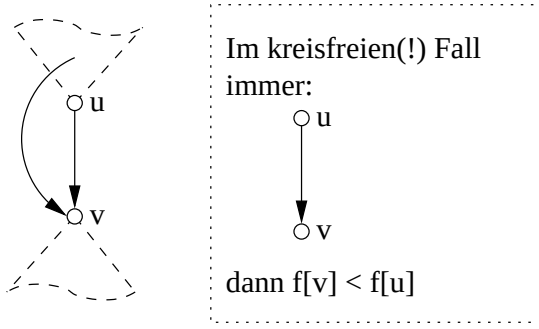
$$A \Rightarrow B \quad \equiv \quad \neg A \vee B \quad \equiv \quad B \vee \neg A \quad \equiv \quad \neg B \Rightarrow \neg A.$$

Beachte aber



Wenn $(u, v) \in E$ und $f[u] < f[v]$, gilt der Satz nicht. Aber wir haben ja auch einen Kreis!

Im kreisfreien Fall etwa so:

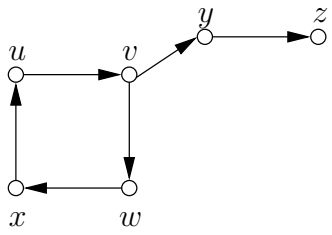


Wird u vor v grau, dann ist klar $f[v] < f[u]$. Wird aber v vor u grau, dann wird u nicht nachfolger von v, also auch dann $f[v] < f[u]$, Weg kreisfrei.

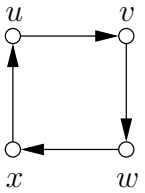
5 Anwendung Tiefensuche: Starke Zusammenhangskomponenten

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.

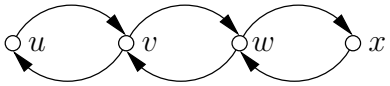
Einige Beispiele:



Starke Zusammenhangskomponente



Der Knoten 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 $\Longleftrightarrow$ für alle $u, v \in V$ gibt es Wege (u, v) und (v, u)*

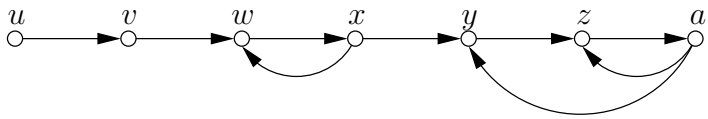
Also noch einmal: Ist (v, u) stark zusammenhängend?

Es bietet sich an, einen Graphen, der nicht stark zusammenhängend ist, in seine stark zusammenhängenden Teile aufzuteilen.

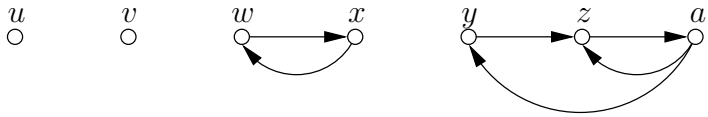
Definition 5.2(Starke Zusammenhangskomponente, starke Komponente): *Ist $G = (V, E)$ gerichteter Graph. Ein Teilgrpah $H = (W, F)$ von G, d.h. $W \subseteq V$ und $F \subseteq E$ ist eine starke Zusammenhangskomponente $\Longleftrightarrow$ H ist ein maximaler induzierter Teilgraph von G, der stark zusammenhängend ist. Man sagt auch:* starke Komponente.

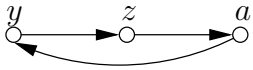
Was soll das das maximal? Wir haben folgende Anordnung auf dem induizierten Teilgraphen von G:

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



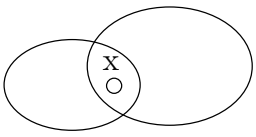
dann sind die starken Komponenten



Nicht 

, 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:

Algorithmus 6: Test auf starken Zusammenhang (naive Variante)

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

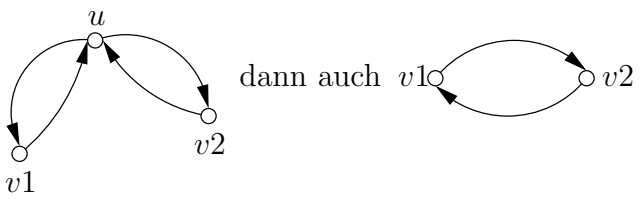
```
1  foreach (u, v) ∈ V × V mit u < v do
2  |  BFS(G, u);
3  |  Speichere die Farbe von v; /* Wenn v erreichbar, dann schwarz. */
4  |  BFS(G, v);
5  |  Speichere die Farbe von u; /* Wenn u erreichbar, dann schwarz. */
6  |  if v erreichbar und u erreichbar then
   |  |  /* Die Wege (u, v) und (v, u) existieren. Hier ist nichts
   |  |     mehr zu tun, weiter mit dem nächsten Paar */
7  |  else
   |  |  /* Mindestens einer der Wege (u, v) und (v, u) existert
   |  |     nicht. Der Graph kann nicht mehr stark zusammenhängend
   |  |     sein. Wir können hier direkt abbrechen. */
8  |  |  return „Nicht stark zusammenhängend.“
9  |  end
10 end
   /* Wenn der Algorithmus bis hierher gekommen ist, existieren
      alle Wege. */
11 return „Stark zusammenhängend.“
```

Zeitabschätzung:

Zeile	Laufzeit (global gezählt)
1	Es gibt $O(\|V\|^2)$ Paare.
2-10	Eine Breitensuche dauert $O(\|V\| + \|E\|)$. Die Tests sind $O(1)$. Also insgesamt $O(\|V\|^2 \cdot (\|V\| + \|E\|) = O(\|V\|^2 \cdot \|E\|)$, sofern $\|E\| \geq \frac{1}{2} \cdot \|V\|$. ($O(\|V\|^2 \cdot \|E\|)$ kann $O(\|V\|^4)$ sein.)
11	$O(1)$

Es geht etwas besser:

Dazu zuerst die folgende Überlegung. Sei u ein beliebiger Knoten des *stark zusammenhängenden Graphen* und v_1, v_2 zwei von u aus erreichbare Knoten. Wenn jetzt außerdem noch ein Weg von v_1 nach u und auch von v_2 nach u existiert, dann existieren auch die Wege (v_1, v_2) und (v_2, v_1). Nämlich mindestens $(v_1, \ldots, u, \ldots, v_2)$ und $(v_2, \ldots, u, \ldots, v_1)$.



Algorithmus 7: Test auf starken Zusammenhang (mit modifizierter Breitensuche)

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

```
/* Die Breitensuche ist derart modifiziert, dass sie eine Liste
   der vom Startknoten aus erreichbaren Knoten liefert. */
```

1 $L = \mathrm{MBFS}(G, u)$; `/* u ist ein beliebiger Knoten z.B. Knoten 1. */`

```
/* Im Prinzip müssen jetzt alle Knoten außer u in L sein. Sonst
   kann der Graph nicht stark zusammenhängend sein. Man kann
   dann bestenfalls noch die Zusammenhangskomponente, in der u
   enthalten ist, bestimmen. */
```

2 **foreach** $v \in L$ **do**
3 | BFS(G, v); `/* normale Breitensuche */`
4 | **if** *u von v aus erreichbar* **then**
| | `/* Weg i(v,u) existiert. Nächten Knoten testen. */`
5 | **else**
6 | | **return** *„Nicht stark zusammenhängend.“*;
| | | `/* bzw. wenn man an der Komponente von u interessiert ist, dann */`
| | | `//` $L = L \setminus \{v\}$;
7 | **end**
8 **end**

Zeitabschätzung:

Zeile	Laufzeit
1	$O(\|V\| + \|E\|)$
2-8	$O(\|V\| \cdot (\|V\| + \|E\|)$ also $O(\|V\| \cdot \|E\|)$ sofern $\|E\| \geq \frac{1}{2} \cdot \|V\|$. $\|V\| \cdot \|E\|$ kann bis $\|V\|^3$ gehen. ($\|V\| = 10$, dann $\|V\|^3 = 1000$)

Satz 5.1(Zusammenhangslemma): *Sei $G = (V, E)$ ein gerichteter Graph und sei $v \in V$. G ist stark zusammenhängend genau dann, wenn für alle $w \in V$ gilt* $v \rightsquigarrow w$ *und* $w \rightsquigarrow v$.

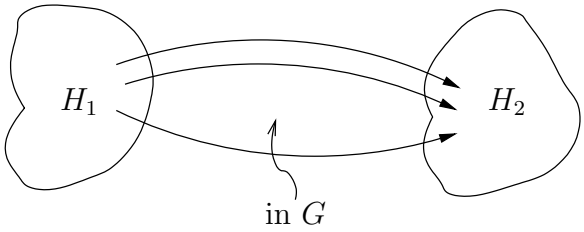
Beweis. „⇒“ Klar nach Definition.

„⇐“ Sind $x, y \in V$, dann $x \overset{v}{\rightsquigarrow} y$ und $y \overset{v}{\rightsquigarrow} x$ nach Voraussetzung. □

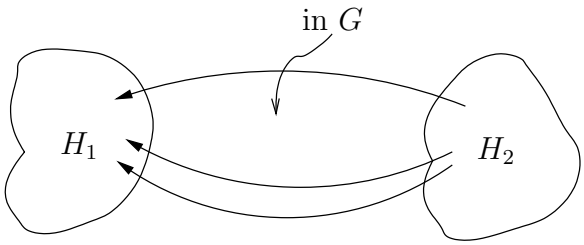
Die mit Hilfe der Breitensuche erreichte Laufzeit von $O(|V| \cdot |E|)$ bzw. $O(|V|^3)$ ist unter praktischen Gesichtspunkten noch nicht besonders gut. Wenn man zum Finden der starken Zusammenhangskomponenten jedoch die Tiefensuche einsetzt, kann man wesentlich besser werden.

5.1 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

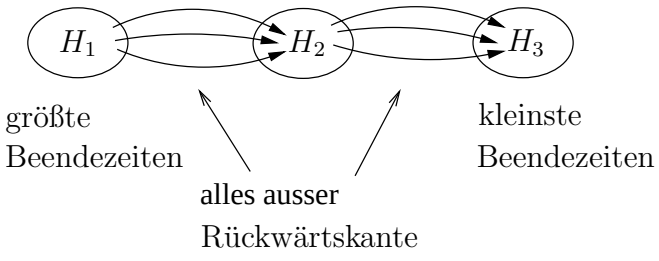


und allgemeiner:

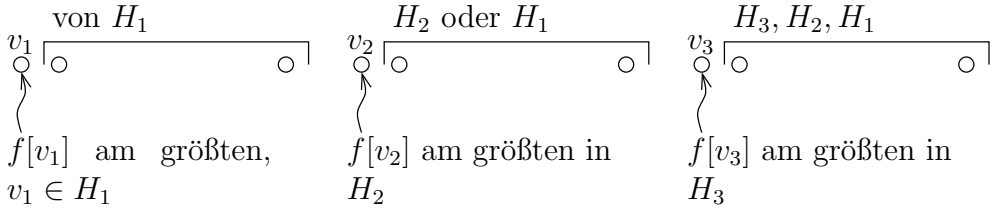
Satz 5.2: *Fassen wir die starken Komponenten als einzelnen Knoten auf und verbinden sie genau dann, wenn sie in G verbunden sind, so ist der entstehende gerichtete Graph kreisfrei.*

Beweis. 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:



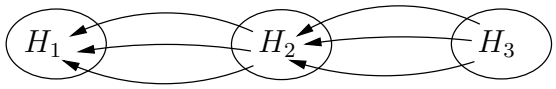
v_1 = Knoten mit kleinster Anfangszeit in H_1
v_2 = Knoten mit kleinster Anfangsteit in H_2
v_3 = Knoten mit kleinster Anfangszeit in H_3

Bemerkung:(Weißer-Weg-Satz) Die maximale Beendezeit in Komponenten lässt die topologische Sortierung erkennen.

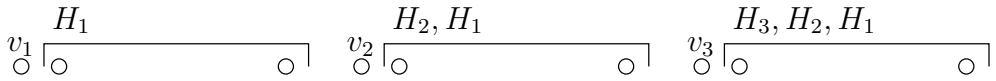
Wie kann man aber diese v_1, v_2, v_3 erkennen?

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:



(Komponenten kommen nach topologischer Sortierung.)

Dann liefert

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

DFS-visit(v_2): Genau H_2.
v_3 vorne auf der Liste.

DFS-visit(v_3): Genau H_3.
Alle Knoten abgearbeitet, wenn nur drei Komponenten vorhanden sind.

Algorithmus 8: Starke-Kompunenten(G)

Input : $G = (V, E)$ gerichteter Graph

```
/* Die Tiefensuche liefert eine Liste der Knoten nach
   absteigender Beendezeit sortiert. */
```

1 $L = \mathrm{DFS}_1(G)$; `/*` $L = (v_1, v_2, \ldots, v_n)$ `mit` $f[v_1] > f[v_2] > \cdots > f[v_n]$ `*/`

2 G^U = „Drehe alle Kanten in G um.“

```
/* Die Hauptschleife der Tiefensuche bearbeitet jetzt die
   Knoten in Reihenfolge der Liste L. */
/* Jedes DFS-visit(v) der Hauptschleife ergibt eine starke
   Komponente. */
```

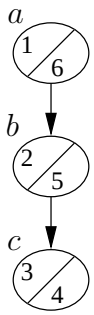
3 $\mathrm{DFS}_2(G^U, \mathrm{L})$;

Die Laufzeit beträgt offensichtlich $O(|V| + |E|)$. Es wird $2\times$ die Tiefensuche ausgeführt. Das umdrehen der Kanten geht auch in $O(|V| + |E|)$.

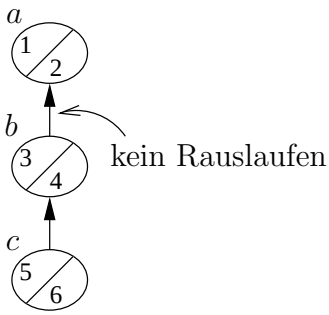
Beispiel 5.1:

$V = \{a, b, c\}$

1. Tiefensuche

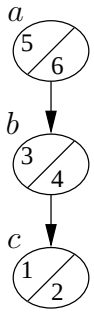


2. Tiefensuche

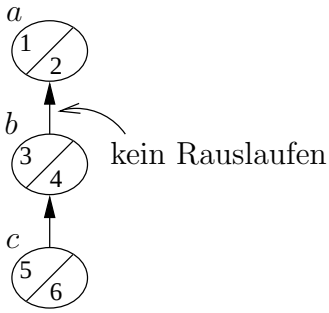


Eine andere Reihenfolge bei der ersten Tiefensuche liefert:

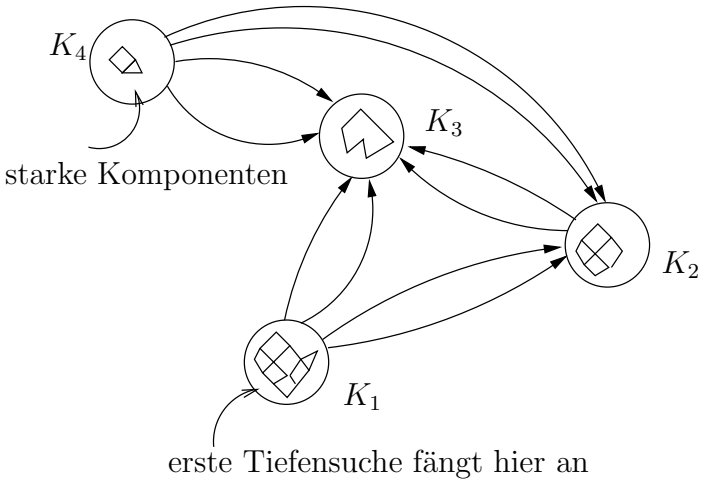
1. Tiefensuche



2. Tiefensuche wie vorher.



Motivation des Algorithmus

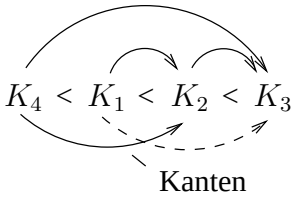


- Eine Tiefensuche entdeckt in jedem Fall die Komponete eines Knotens.
- Die Tiefensuche kann aber aus der Komponente rauslaufen!
- Die Tiefensuche lässt die topologische Sortierung auf den Komponenten erkennen. → Nach maximaler Beendezeit in Komponente:

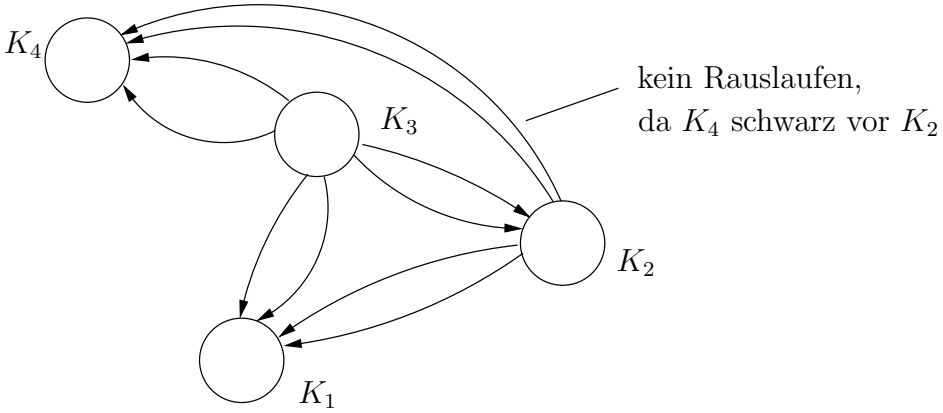
$$\begin{array}{ccccccc} K_4 & < & K_1 & < & K_2 & < & K_3 \\ \uparrow & & & & & & \uparrow \end{array}$$

maximale Beendezeit

egal, ob zuerst nach K_2 oder K_3



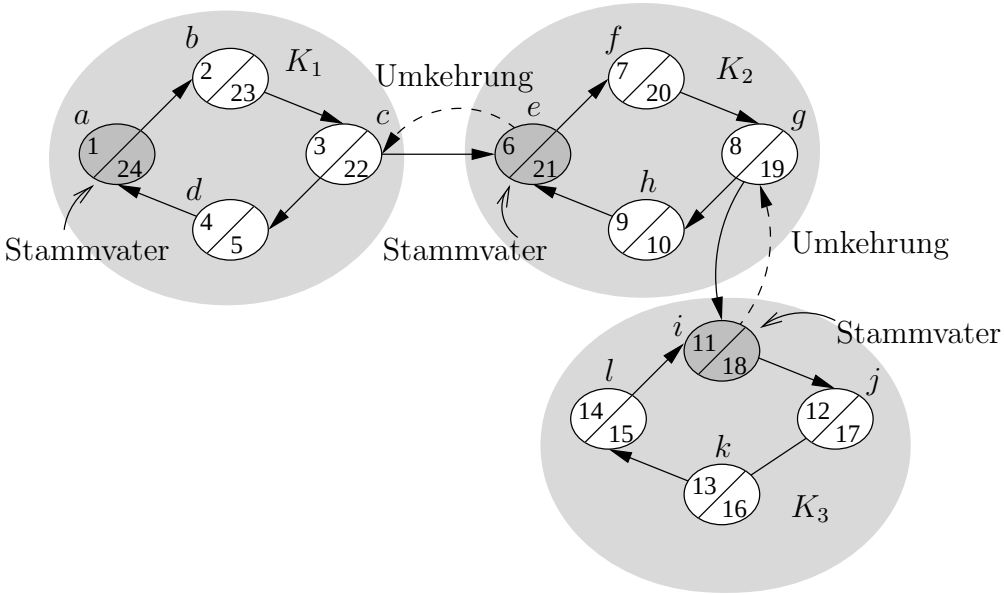
- Bei Beginn in K_4 bekommen wir: $K_1 < K_4 < K_2 < K_3$.
- Wie kann man das Rauslaufen jetzt verhindern?



- Hauptschleife gemäß topologischer Sortierung, d.h. abfallende Beendezeit:

$$K_4, K_1, K_2, K_3$$

Beispiel 5.2:



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

$$(\underbrace{a, b, c}_{K_1}, \underbrace{e, f, g}_{K_2}, \overbrace{i, j, k, l}^{K_3}, \underbrace{h}_{K_2}, \underbrace{d}_{K_1})$$

Stammvater ist der Knoten, über den eine starke Komponente das erste Mal betreten wird. D.h. der Knoten mit der kleinsten Anfangszeit in dieser Komponente.

Einige Überlegungen zur Korrektheit. Welcher Knoten einer Komponente wird zuerst betreten?

Definition 5.3(Stammvater): *Nach dem Lauf von DFS(G) sei für $u \in V$. $\Phi(u)$ ist der Knoten, der unter allen von u erreichbaren Knoten die maximale Beendezeit hat.* $\Phi(u)$ *nennt man* Stammvater *von* u.

Satz 5.3: *Es gilt:*

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

(b) In G gibt es die Wege $\Phi[u] \rightsquigarrow u$ *und* $u \rightsquigarrow \Phi[u]$.

Beweis. (a) Klar bei $u = \Phi(u)$. Sei also $u \neq \Phi(u)$. Wir schließen die anderen Fälle aus.

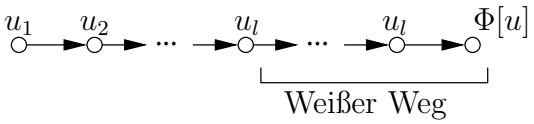
1. Fall: $col[\Phi(u)]$ schwarz bei $d[u]$. Dann $f[\Phi(u)] < d[u] < f[u]$ im Widerspruch zur Definition, da ja 

.

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 

) sieht es so aus:



$$\begin{aligned} col[u_l] &= \text{grau bei } d[u] \\ col[u_l] &= \text{schwarz geht nicht} \end{aligned}$$

   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 

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

□

Folgerung 5.1:

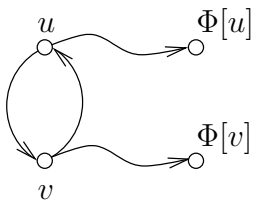
a) u, v *in einer starken Komponente* $\Longleftrightarrow \Phi(u) = \Phi(v)$

Stammväter $\Longleftrightarrow$ *starke Komponenten*

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

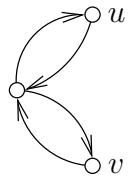
Beweis.

a) Mit letztem Satz und Definition Stammvater:



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

Andererseits

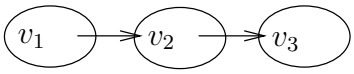
$\Phi[u] = \Phi[v]$ 

Also u, v in einer Komponente.

b) 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:



Die erste Tiefensuche fängt z.B. bei v_2 an.

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

Satz 5.4: *In der Liste $L = (v_1, \ldots, v_n)$ gibt die Position der Stammväter eine topologische Sortierung der starken Komponenten an.*

Beweis. Sei also die Liste $L = (v_1, \ldots, v_k, \ldots, v_l)$.

Sei $k < l$, dann gibt es *keinen* Weg in G. 

, 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 gerichteten Fall und danach betrachten wir in Abbildung 2 den ungerichteten Fall auf dem analogen Graphen (Kanten „⟶—“ sind Baumkanten, über die entdeckt wird.)

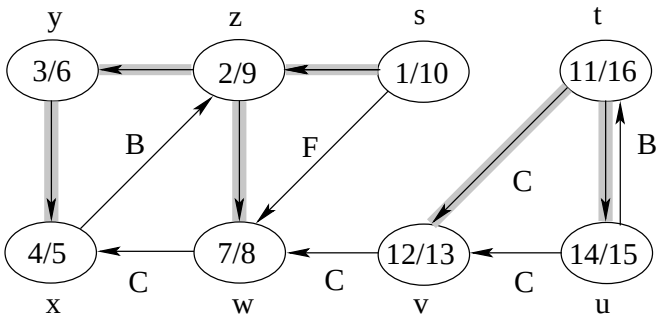


Abbildung 1: Das Ergebnis der Tiefensuche auf einem gerichteten Graphen

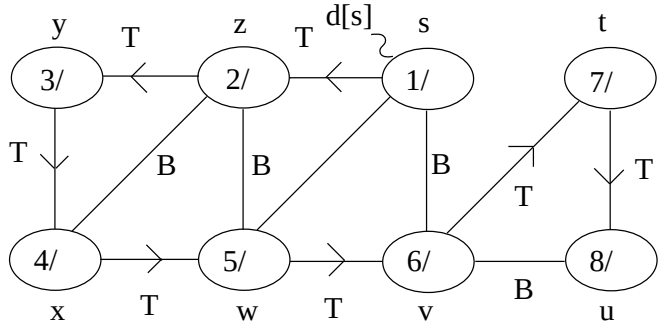
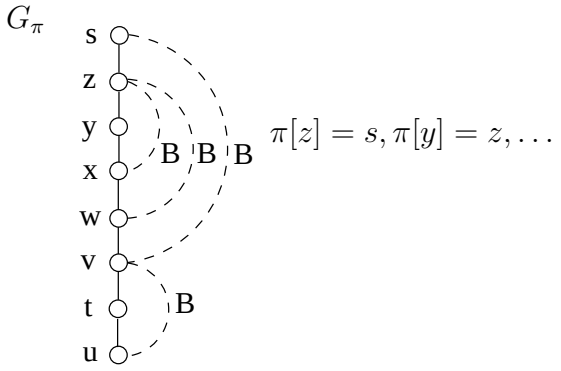


Abbildung 2: vorhergender Graph, ungerichtete Variante

Im ungerichteten Graphen gilt: beim ersten Gang durch eine Kante stößt man

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



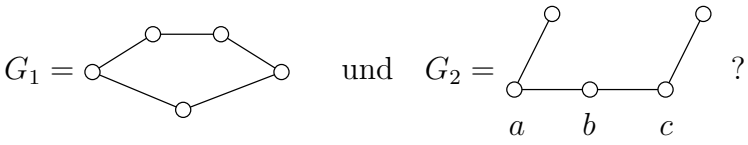
Im ungerichteten Fall der Tiefensuche ist festzuhalten:

- 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 $\Longleftrightarrow$ 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 ein Weg (v, u) ist.

Stattdessen: *zweifach zusammenhängend*. Unterschied zwischen



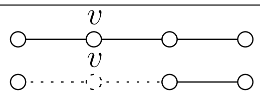
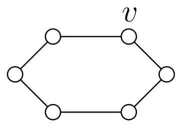
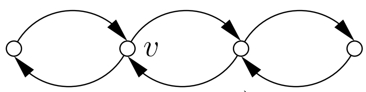
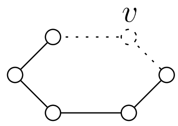
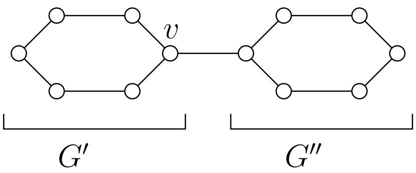
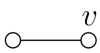
Löschen wir in G_1 einen beliebigen Knoten, hängt der Rest noch zusammen. Für a, b, c im Graphen G_2 gilt dies nicht.

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): *$G = (V, E)$ ist zweifach zusammenhängend $\Longleftrightarrow$ $G \setminus \{v\}$ ist zusammenhängend für alle $v \in V$.*

$G \setminus \{v\}$ bedeutet $V \setminus \{v\}, E \setminus \{\{u, v\} \mid u \in V\}$.

Beispiel 6.1:

G: $G \setminus \{v\}$:		G zusammenhängend, nicht zweifach zusammenhängend.
G:		(Beachte:  ist stark zusammenhängend.)
$G \setminus \{v\}$:		Knoten v und adjazente Kanten fehlen. G ist zweifach zusammenhängend.
G:		G ist nicht zweifach zusammenhängend, da $G \setminus \{v\}$ nicht zusammenhängend ist. Aber G' und G'' sind jeweils zweifach zusammenhängend.
G:	 $G \setminus \{v\}$: 	G ist also zweifach zusammenhängend.

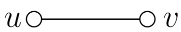
Was ist die Bedeutung zweifach zusammenhängend?

Satz 6.1(Menger 1927): *G zweifach zusammenh. $\Longleftrightarrow$ 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

$$\{u_1, \ldots, u_m\} \cap \{u_1', \ldots, u_m'\} = \emptyset.$$

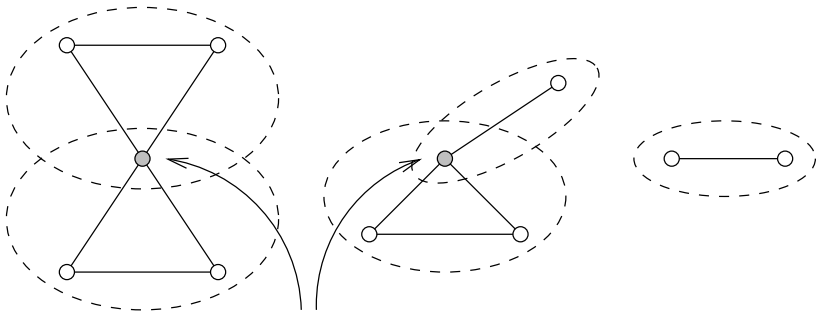
Beweis erfolgt auf überraschende Weise später.

Ein induktiver Beweis im Buch Graphentheorie von Reinhard Diestel. Die Richtung „$\Leftarrow$“ ist einfach. Beachte noch, ist  , so tut es der Weg (u, v), mit leerer Menge von Zwischenknoten (also nur 1 Weg).

Nun gilt es wiederum nicht zweifach zusammenhängende Graphen in ihre zweifach zusammenhängenden Bestandteile zu zerlegen, in die zweifachen Zusammenhangs-

komponenten (auch einfach zweifache Komponenten genannt).

Beispiel 6.2:



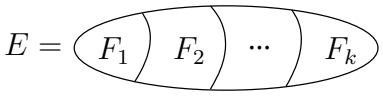
Knoten in zwei zweifachen Zusammenhangskomponenten

Definition 6.2(zweifache Komponenten): *Ein Teilgraph $H = (W, F)$ von G ist eine zweifache Komponente $\Longleftrightarrow$ H ist ein maximaler zweifach zusammenhängender Teilgraph von G (maximal bezüglich Knoten und Kanten).*

Bemerkung:

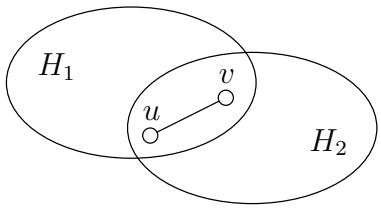
(a) Jede Kante ist in genau einer zweifachen Komponente.

(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 $i \neq j$, $\quad F_1 \cup \cdots \cup F_k = E$.

Beweis. (a) Wir betrachten die Teilgraphen H_1 und H_2 sowie eine Kante $\{u, v\}$, die sowohl in H_1 als auch in H_2 liegt.



Seien also $H_1 \neq H_2$ zweifach zusammenhängend, dann ist $H = H_1 \cup H_2$ zweifach zusammenhängend:

Für alle w aus H_1, $w \neq u, w \neq v$ ist $H \setminus \{w\}$ zusammenhängend (wegen Maximalität). Ebenso für alle w aus H_2.

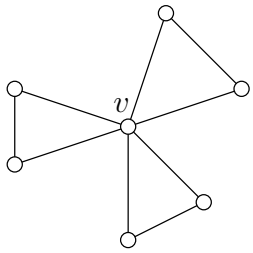
Für $w = u$ ist, da immer noch v da ist, $H \setminus \{w\}$ zweifach zusammenhängend. Also wegen Maximalität zweifache Komponenten $\supseteq H_1 \cup H_2$.

(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_i.

□

Bezeichnung: Eine Kante, die eine zweifache Komponente ist, heißt *Brückenkante.*

Bei Knoten gilt (a) oben nicht:



v gehört zu 3 zweifachen Komponenten. v ist ein typischer Artikulationspunkt.

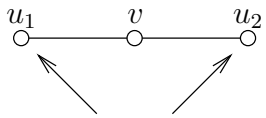
Definition 6.3(Artikulationspunkt): *v ist Artikulationspunkt von $G \iff G \setminus \{v\}$* nicht *zusammenhängend.*

Bemerkung 6.3:

$$v \text{ ist Artikulationspunkt} \iff v \text{ gehört zu} \geq 2 \text{ zweifachen Komponenten.}$$

Beweis. „⇒“ Ist 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)

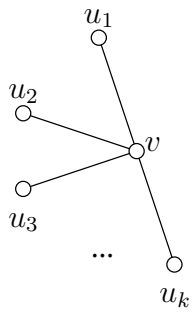
Also sind u, w nicht in einer zweifachen Komponente. Dann ist jeder Weg von der Art $(u, \ldots, u_1, v, u_2, \ldots, w)$, so dass u_1, u_2 nicht in einer zweifachen Komponente sind. Sonst ist in $G \setminus \{v\}$ ein Weg $(u, \ldots, u_1, u_2, \ldots w)$ (ohne v), Widerspruch. Also haben wir:



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. Also v liegt in den beiden Komponenten.

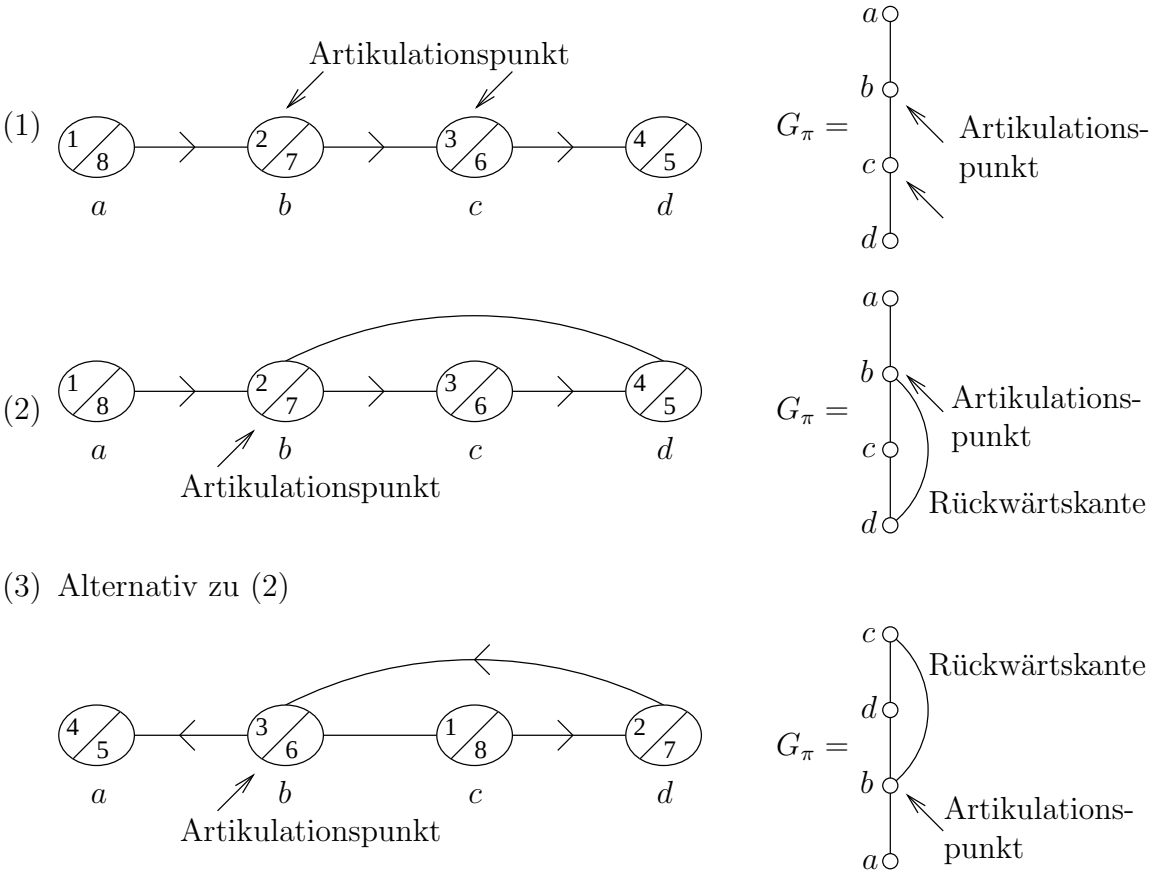
„⇐“ Gehört also v zu > 2 Komponenten. Dann



und ≥ 2 Kanten von $\{u_i, v\}$ liegen in verschiedenen Komponenten. Etwa u_1, u_2. Aber u_1 und u_2 sind in zwei verschiedenen Komponenten. Also gibt es w, so dass in $G \setminus \{w\}$ kein Weg $u_1 \circ\!\!-\!\!\circ u_2$ existiert. Denn wegen $u_1 \circ\!\!-\!\!\overset{v}{\circ}\!\!-\!\!\circ u_2$ muss $w = v$ sein und v ist Artikulationspunkt. □

Graphentheoretische Beweise sind nicht ganz so leicht!

Artikulationspunkte und Tiefensuche?



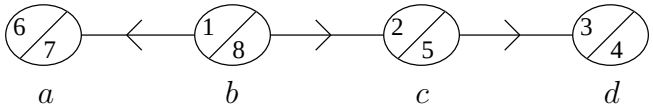
Was haben die Artikulationspunkte in allen genannten Fällen gemeinsam? Der Artikulationspunkt (sofern nicht Wurzel von G_π) hat einen Sohn in G_π, so dass es von dem Sohn oder Nachfolger keine Rückwärtskante zum echten Vorgänger gibt!

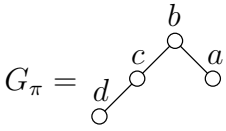
(1) Gilt bei b und c. d ist kein Artikulationspunkt und das Kriterium gilt auch nicht.

(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!

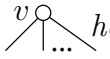
Was ist, wenn der Artikulationspunkt Wurzel von G_π ist?



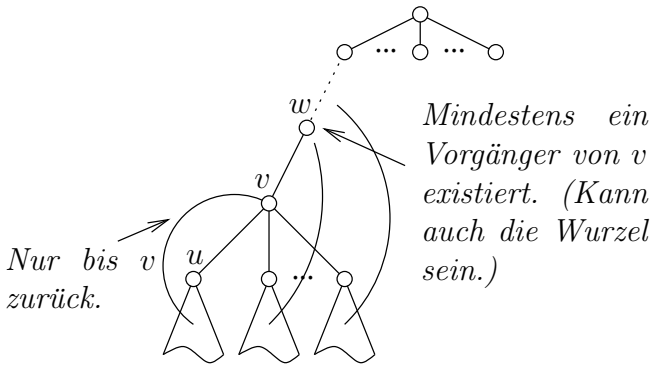


Zwei Söhne.

Satz 6.2(Artikulationspunkte erkennen): *Sei v ein Knoten von G und sei eine Tiefensuche gelaufen. (Erinnerung: G immer zusammenhängend).*

a) Ist v Wurzel von G_π. Dann ist v Arikulationspunkt $\Longleftrightarrow$ v in G_π  *hat ≥ 2 Söhne.*

b) Ist v nicht Wurzel von G_π. Dann ist v Artikulationspunkt $\Longleftrightarrow$ G_π folgendermaßen:



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

Beweis.

a) „$\Leftarrow$“ $G \setminus \{v\}$ nicht zusammenhängend, damit v Artikulationspunkt.

„$\Rightarrow$“ Hat v nur einen Sohn, dann $G_\pi =$

Rückwärts-kanten 

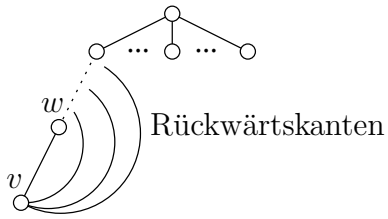
Zeige $A \Rightarrow B$ durch $\neg B \Rightarrow \neg A$.

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

b) „$\Leftarrow$“ In $G \setminus \{v\}$ kein Weg $u \sim w$, also ist v Artikulationspunkt.

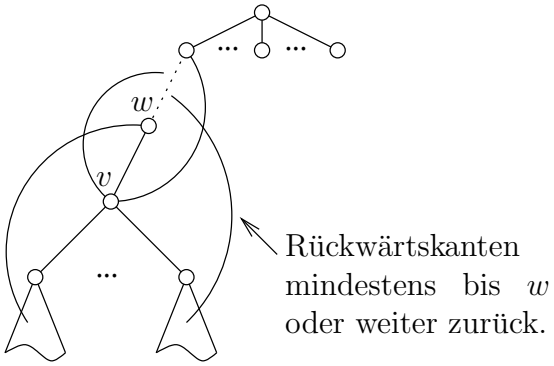
„$\Rightarrow$“ Gelte die Behauptung nicht, also v hat keinen Sohn wie u in G_π.

1. 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 $G_\pi =$



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.

$$\boxed{\text{Zeige } A \Rightarrow B \text{ durch } \neg B \Rightarrow \neg A.}$$

□

Wie kann man Artikulationspunkte berechnen?

Wir müssen für alle Söhne in G_π wissen, wie weit es von dort aus zurück geht.

Definition 6.4(Low-Wert): *Sei DFS(G) gelaufen, also G_π vorliegend.*

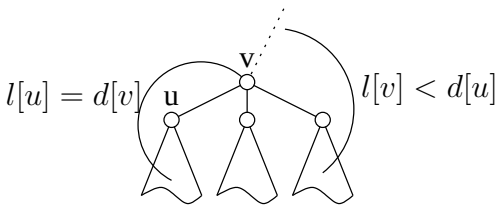
a) Für Knoten v ist die Menge von Knoten $L(v)$ gegeben durch $w \in L(v) \iff$ $w = v$ oder w Vorgänger von v und es gibt eine Rückwärtskante von v oder dem Nachfolger zu w.

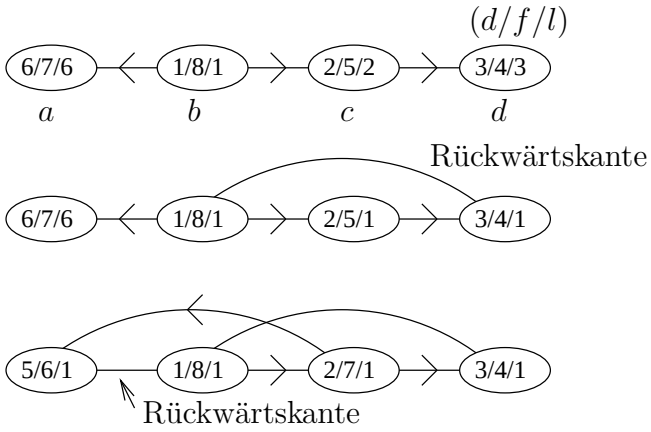
b) $l[v] = \min\{d[w] | w \in L(v)\}$ *ist der Low-Wert von* v*. (*$l[v]$ *hängt von Lauf von DFS(G) ab.)*

Folgerung 6.1: *Sei DFS(G) gelaufen und* v *nicht Wurzel von* G_π*.*

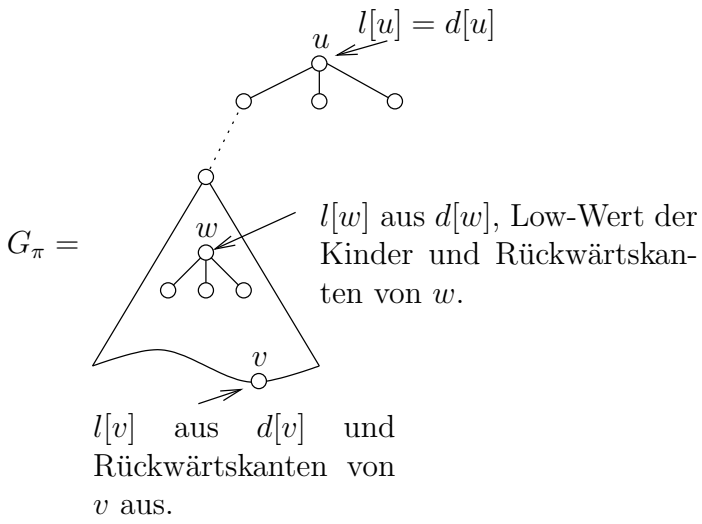
$$\boxed{v \text{ ist Artikulationspunkt} \iff v \text{ hat Sohn } u \text{ in } G_\pi \text{ mit } l[u] \geq d[v].}$$

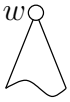
Beachte: $l[v] = d[v] \Longrightarrow v$ Artikulationspunkt. *Keine Äquivalenz!*





6.1 Berechnung von l[v]



Korrektheit: Induktion über die Tiefe des Teilbaumes 

.

6.2 Algorithmus (l-Werte)

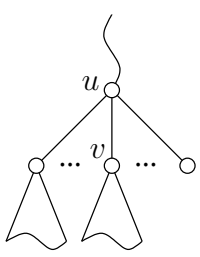
Wir benutzen eine modifizierte Version der Prozedur DFS-visit(u).

Prozedur MDFS-visit(u)

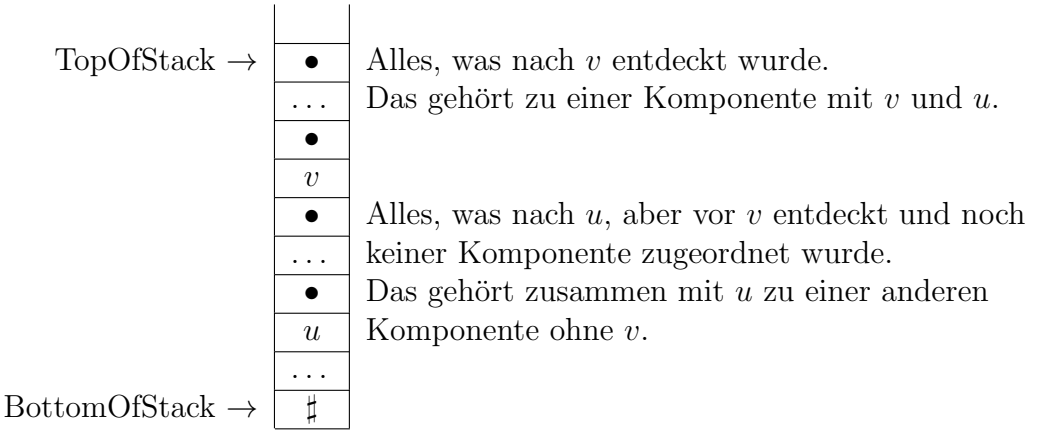
```
 1  col[u] = grau;
 2  d[u] = time;
 3  l[u] = d[u];
 4  time = time + 1;
 5  foreach v ∈ Adj[u] do
 6      if col[v] == weiß then
 7          π[v] = u;
 8          MDFS-visit(v);
            /* Kleineren Low-Wert des Kindes übernehmen. */
 9          l[u] = min{l[u], l[v]};
10      end
11      if col[v] == grau und π[u] ≠ v then          /* Rückwärtskante */
            /* Merken, welche am weitesten zurück reicht. */
12          l[u] = min{l[u], d[v]};
13      end
14  end
15  col[u] = schwarz;
16  f[u] = time;
17  time = time + 1;
```

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

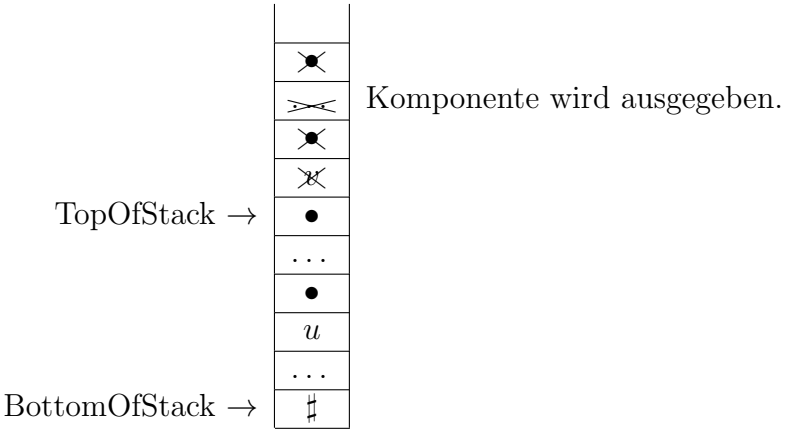
- Weiße Knoten werden vor dem Aufruf von DFS-visit auf einem Keller gespeichert. Wir befinden uns jetzt in DFS-visit(u). Der Tiefensuchbaum aus der ersten Tiefensuche sieht so aus:

$G_\pi =$ 

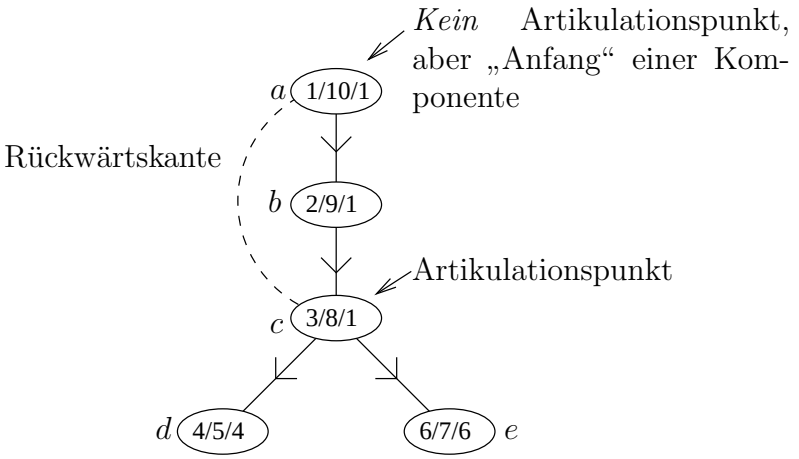
Sei jetzt $l[v] \geq d[u]$. Das heißt u ist ein Artikulationspunkt bezüglich v. Noch in DFS-visit(u), direkt nachdem DFS-visit(v) zurückgekehrt ist sieht unser Keller so aus: (Wenn unter v keine weiteren Artikulationspunkte sind.)



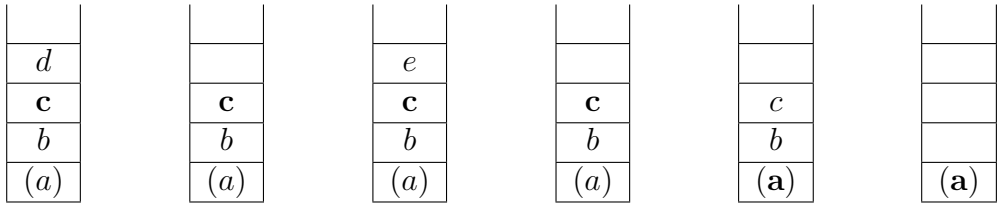
- Nachdem DFS-visit(v) zurückgekehrt ist, geben wir jetzt den Kellerinhalt bis einschließlich v aus. Dann macht die zweite Tiefensuche beim nächsten Kind von u weiter.



Beispiel 6.4:



DFS-visit(c), DFS-visit(d) fertig	Ausgabe: d, c	DFS-visit(c), DFS-visit(e) fertig	Ausgabe: e, c	DFS-visit(a), DFS-visit(b) fertig	Ausgabe: c, b, a



$l[d] \geq d[c]$ $l[e] \geq d[c]$ $l[b] \geq d[a]$

6.3 Algorithmus (Zweifache Komponenten)

Algorithmus 9: 2fache Komponenten(G)

Input : ungerichteter Graph $G = (V, E)$

```
   /* Modifizierte Tiefensuche für l-Werte */
 1 MDFS(G);
   /* Alle Knoten wieder weiß. Die restlicher Werte bleiben
      erhalten. Insbesondere d und l werden noch gebraucht. */
 2 foreach v ∈ V do
 3 |  col[v] =weiß
 4 end
 5 S = ();                                  /* Keller initialisieren */
 6 foreach v ∈ V do              /* Dieselbe Reihenfolge wie bei 1. */
 7 |  if col[v] == weiß then
 8 |  |  NDFS-visit(v);
   |  |  /* Kann bei isolierten Knoten passieren. */
 9 |  |  if Q ≠ () then
10 |  |  |  Knoten in Q ausgeben;
11 |  |  |  Q = ();
12 |  |  end
13 |  end
14 end
```

Prozedur NDFS-visit(u)

```
1  col[u] = grau;
2  foreach v ∈ Adj[u] do
3      if col[v] == weiß then
4          push(Q, v);                                  /* v auf Keller */
5          NDFS-visit(v);
6          if l[v] ≥ d[u] then                /* u ist Artikulationspunkt */
7              Solange Knoten von Q entfernen und Ausgeben, bis v
               ausgegeben;
8              Knoten u auch mit ausgeben;
9          end
10     end
11 end
12 col[u] = schwarz;
```

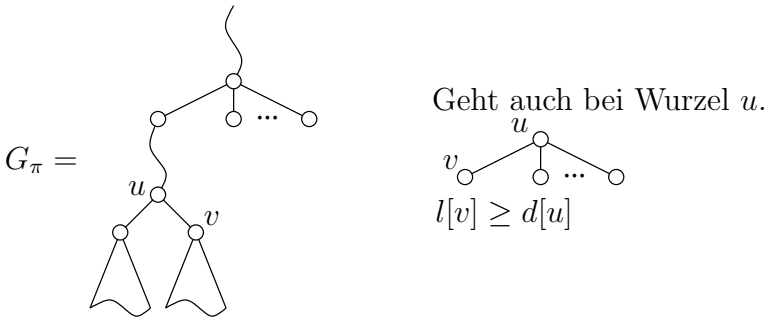
Beweis. Korrektheit induktiv über $s = \#$Zweifache Komponenten von G.

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 Korrektheit 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 dem die Ausgabe erfolgt. Dann ist $l[v] \geq d[u]$.

Ist $l[v] = d[v]$, dann ist v Blatt (sonst vorher Ausgabe) und v, u wird ausgegeben.

Ist $l[v] > d[u]$, dann Ausgabe des Kellers bis v und zusätzlich u. Damit wird eine zweifache Komponente ausgegeben. Alles was nach v entdeckt wurde, fällt

in $G \setminus \{u\}$ ab. (Mehr kann nicht dazu gehören, weniger nicht, da bisher kein Artikulationspunkt).

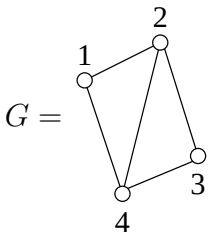
Nach Ausgabe der Komponente liegt eine Situation vor, die in DFS-visit(u) auftritt, wenn wir den Graphen betrachten, in dem die Kante $\{u, v\}$ und die ausgegegenen Knoten außer u gelöscht sind.

Auf diesem ist die Induktionsvoraussetzung anwendbar und der Rest wird richtig ausgegeben.

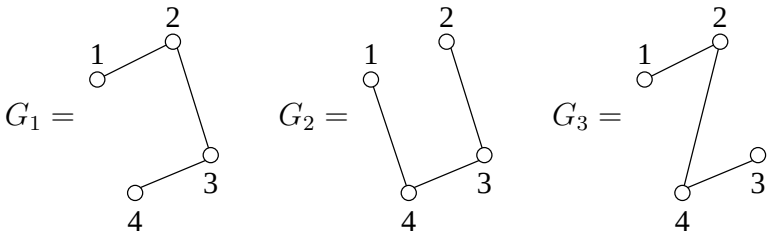
□

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 $f \in F$ $H \setminus \{f\} = (V, F \setminus \{f, \cdot\})$ nicht mehr zusammenhängend ist. (H ist minimal zusammenhängend.)*

Folgerung 7.1: *Sei H zusammenhängend, Teilgraph von G.*

$$H \text{ Spannbaum von } G \iff H \text{ ohne Kreis.}$$

Beweis. „$\Rightarrow$“ Hat H einen Kreis, dann gibt es eine Kante f, so dass $H \setminus \{f\}$ zusammenhängend ist. ($A \Rightarrow B$ durch $\neg B \Rightarrow \neg A$)

„$\Leftarrow$ “ 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 $\neg A \Rightarrow \neg B$) □

Damit hat jeder Spannbaum genau $n - 1$ Kanten, wenn $|V| = n$ ist. Vergleiche Seite 38. Die Baumkanten einer Tiefen- bzw. Breitensuche ergeben einen solchen Spannbaum. Wir suchen Spannbäume minimaler Kosten.

Definition 7.2: *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

$$\binom{m}{n-1} = \frac{m!}{(n-1)! \cdot \underbrace{(m-n+1)!}_{\geq 0 \text{ für } n-1 \leq m}}$$

Wieviel ist das? Sei $m = 2 \cdot n$, dann

$$\begin{aligned} \frac{(2n)!}{(n-1)!(n+1)!} &= \frac{2n \cdot (2n-1) \cdot (2n-2) \cdot \cdots \cdot n}{(n+1) \cdot n \cdot (n-1) \ldots 1} \\ &\geq \underbrace{\frac{2n-2}{n-1}}_{=2} \cdot \underbrace{\frac{2n-3}{n-2}}_{>2} \cdot \cdots \cdot \underbrace{\frac{n}{1}}_{>2} \\ &\geq \underbrace{2 \cdot 2 \cdot \cdots \cdot 2}_{n-1 \text{ mal}} = 2^{n-1} \end{aligned}$$

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

Regel: Sei $c \geq 0$, dann:

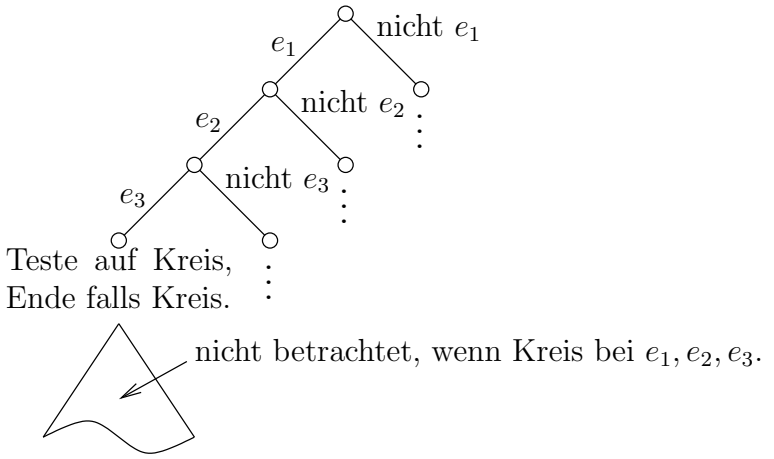
$$\frac{a}{b} \leq \frac{a-c}{b-c} \Leftrightarrow b \leq a.$$

Das c im Nenner hat mehr Gewicht.

Definition 7.3(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 $|f(n)| \geq C \cdot g(n)$ für alle hinreichend großen n. (vergleiche O-Notation, Seite 43)*

Bemerkung: *O-Notation:* Obere Schranke,
Ω-*Notation:* Untere Schranke.

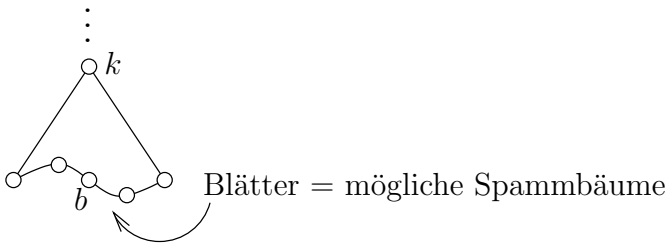
Verbesserung: Backtracking, frühzeitiges Erkennen von Kreisen. Dazu Aufbau eines Baumes (Prozeduraufrufbaum):



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.

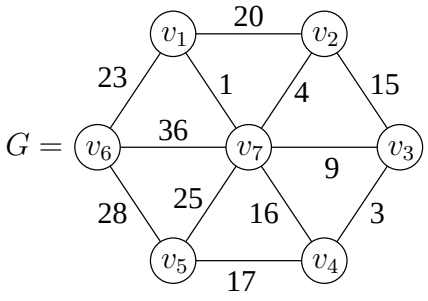
$$\text{Kosten von } k \leq \text{Kosten von } b.$$

Eine Untere Schranke an die Spannbäume unter k.

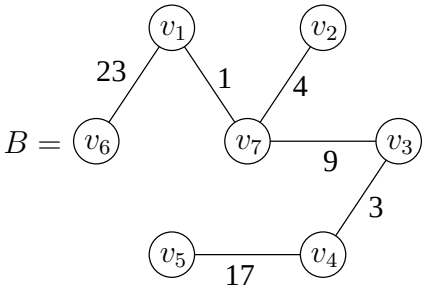
Also weiteres Rausschneiden möglich, wenn Kosten von $k \geq$ Kosten eines bereits gefundenen Baumes. Im allgemeinen ohne weiteres keine bessere Laufzeit nachweisbar.

Besser: Irrtumsfreier Aufbau eines minimalen Spannbaumes.

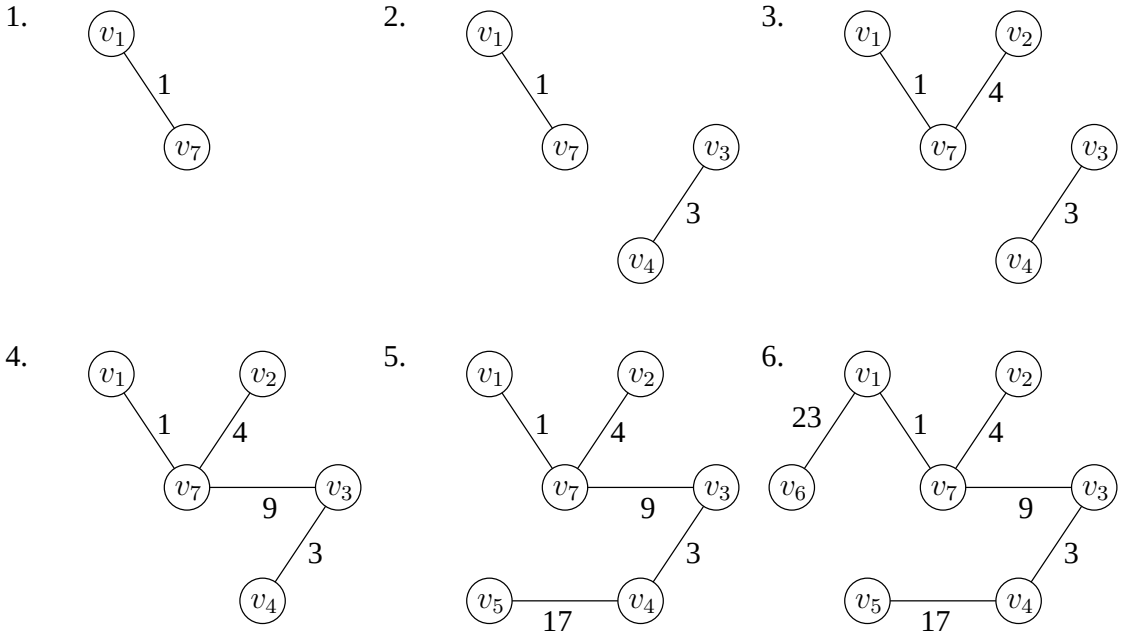
Betrachten wir den folgenden Graphen G mit Kosten an den Kanten.

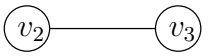
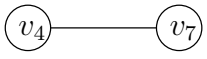


Dazu ist B ein Spannbaum mit minimalen Kosten.



Der Aufbau eines minimalen Spannbaums erfolgt schrittweise wie folgt:



Nimm die günstigste Kante, die keinen Kreis ergibt, also nicht  ,  in Schritt 5 bzw 6.

Implementierung durch Partition von V

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

Problem: Wie Partition darstellen?

7.1 Algorithmus Minimaler Spannbaum (Kruskal 1956)

Algorithmus 10: Minimaler Spannbaum (Kruskal 1956)

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

Output : F = Menge von Kanten eines minimalen Spannbaumes.

```
1  F = ∅;
2  P = {{1}, ..., {n}};                          /* P ist die Partition */
3  E nach Kosten sortieren; /* Geht in O(|E| · log |E|) = O(|E| · log |V|), */
                                  /* |E| ≤ |V|², log |E| = O(log |V|) */
4  while |P| > 1 do                          /* solange P ≠ {{1, 2, ..., n}} */
5  |  {v, w} = kleinstes (erstes) Element von E;
6  |  {v, w} aus E löschen;
7  |  if {v, w} induziert keinen Kreis in F then
8  |  |  W_v = die Menge mit v aus P;
9  |  |  W_w = die Menge mit w aus P;
10 |  |  W_v und W_w in P vereinigen;
11 |  |  F = F ∪ {{v, w}}; /* Die Kante kommt zum Spannbaum dazu. */
12 |  end
13 end
```

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:

Beweis. 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.

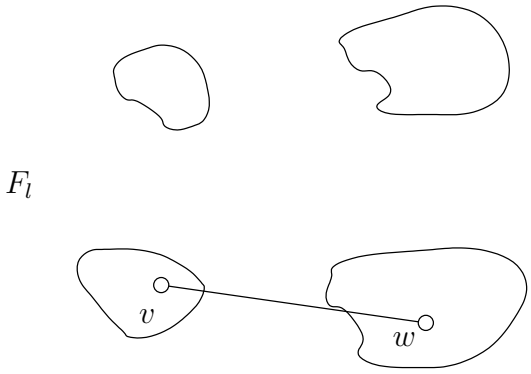
Die Invariante gilt für $l = 0$.

Gelte sie für l und finde ein $l + 1$-ter Lauf statt.

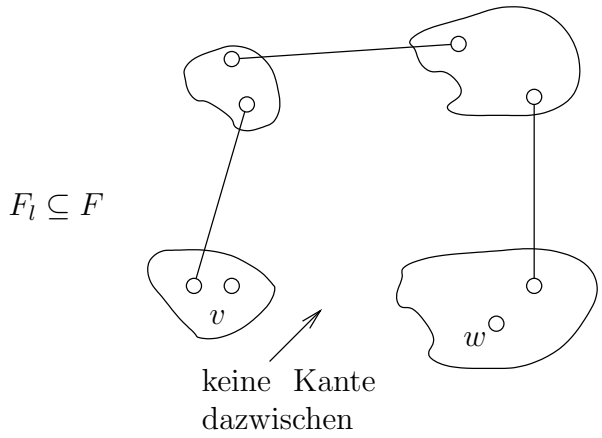
1. Fall Die Kante $\{v, w\}$ wird nicht genommen. Alles bleibt unverändert. Invariante gilt für $l + 1$.

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, muss aber nicht, $\{v, w\}$ enthalten. Zum Beispiel:



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 gehört zu der Liste von Kanten E_l, hat also Kosten, die höchstens größer als die von $\{v, w\}$ sind.

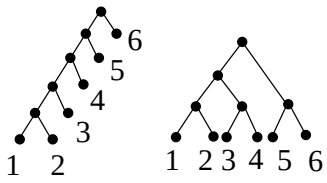
Tauschen wir in F diese Kante und $\{v, w\}$ aus, haben wir einen minimalen Spannbaum mit $\{v, w\}$. Also gilt die Invariante für $F_{l+1} = F_l \cup \{v, w\}$ und P_{l+1} (sofern Operationen richtig implementiert sind.)

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:

- Initialisierung: $O(n)$
- Kanten sortieren: $O(|E| \cdot log|E|)$ (wird später behandelt), also $O(|E| \cdot log|V|)$!
- 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.

    Bis zu $|E|$-mal

  - W_v und W_w in P vereinigen

    Genau $n - 1$-mal.

Die Zeit hängt von der Darstellung von P ab.

Einfache Darstellung: Array $P[1, \ldots, n]$, wobei eine Partition von $\{1, \ldots, n\}$ der Indexmenge dargestellt wird durch:

- u, v in gleicher Menge von $P \longleftrightarrow P[u] = P[v]$
  $(u, v$ sind Elemente der Grundmenge = Indizes)
- 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:

Prozedur union(v, w)

```
1 a = P[v];
2 b = P[w];                                   /* a ≠ b, wenn W_u ≠ W_v */
3 for i = 1 to n do
4 |   if P[i] == a then
5 |   |   P[i] = b;
6 |   end
7 end
```

Damit ist die Laufzeit der Schleife:

7.-11.	Bestimmen der Mengen und Testen auf Kreis geht jeweils in $O(1)$, also insgesamt $O(n)$. Die Hauptarbeit liegt im Vereinigen der Mengen. Es wird $n-1$-mal vereinigt, also insgesamt $n \cdot O(n) = O(n^2)$.
5. und 6.	Insgesamt $O(\|E\|)$.
4.	$O(n)$ insgesamt, wenn $\|P\|$ mitgeführt.

Also $O(n^2)$, selbst wenn E bereits sortiert ist.

Um die Laufzeit zu verbessern, müssen wir das Vereinigen der Mengen schneller machen. Dazu stellen wir P durch eine *Union-Find-Struktur* dar.

7.2 Union-Find-Struktur

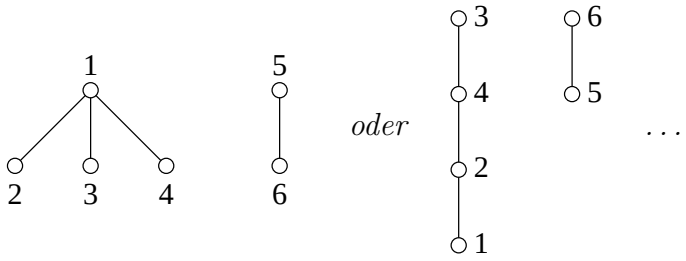
- Darstellung einer Partition P einer Grundmenge (hier klein, d.h. als Indexmenge verwendbar)
- *Union*(U, W): Für $U, W \in P$ soll $U, W \in P$ durch $U \cup W \in P$ ersetzt werden.
- *Find(u)*: Für u aus der Grundmenge soll der Name der Menge $U \in P$ mit $u \in U$ geliefert werden. (Es geht nicht um das Finden von u, sondern darum in welcher Menge u sich befindet!)

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

Definition 7.4(Datenstruktur Union-Find): *Die Mengen werden wie folgt gespeichert:*

(a) Jede Menge von P als ein Baum mit Wurzel.

Sei $P = \{\{1,2,3,4\},\{5,6\}\}$, dann etwa Wurzel = Name der Menge. Die Bäume können dann so aussehen:



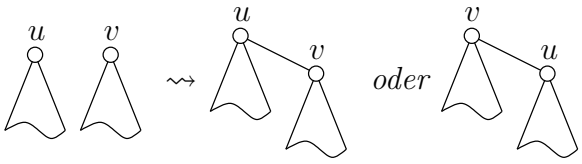
Die Struktur der Bäume ist zunächst egal.

(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.)*

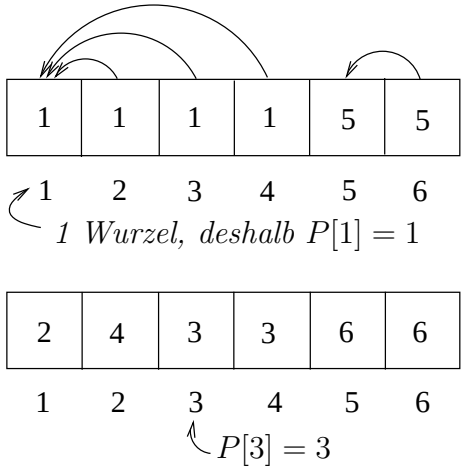
1. *u, v in den Bäumen finden.*
2. *Hänge u unter v (oder umgekehrt).*



(c) Bäume in Array (Vaterarray) speichern:

$$P[1,\ldots,n] \text{ of } \{1,\ldots,n\}, \quad P[v] = \text{Vater von } v.$$

Aus (a) weiter:



Wieder wichtig: Indices = Elemente. Vaterarray kann beliebige Bäume darstellen.

Prozedur union(u, v)

```
   /* u und v müssen Wurzeln sein! */
1  P[u] = v;                                   /* u hängt unter v. */
```

Prozedur find(v)

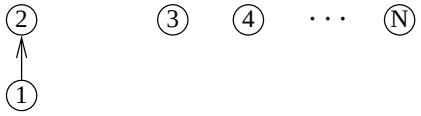
```
1 while P[v] ≠ v do                            /* O(n) im worst case */
2 |  v = P[v];
3 end
4 return v
```

Beispiel 7.1: *Wir betrachten die folgenden Operationen.*

1. ① ② ③ ④ $\cdots$ Ⓝ $\quad P =$

1	*2*	*3*	$\ldots$	*N*
1	*2*	*3*	$\ldots$	*N*

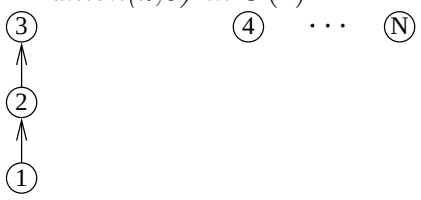
2. $\curvearrowright$ *union(1,2) in* $O(1)$



$P =$

2	*2*	*3*	$\ldots$	*N*
1	*2*	*3*	$\ldots$	*N*

3. $\curvearrowright$ *union(2,3) in* $O(1)$



$P =$

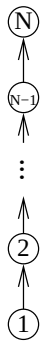
2	*3*	*3*	$\ldots$	*N*
1	*2*	*3*	$\ldots$	*N*

4. $\curvearrowright$ *union(3,4) in* $O(1)$

Das Entstehen langer dünner Bäume.

$\vdots$

N. $\curvearrowright$ *union(N-1,N) in* $O(1)$



$P =$

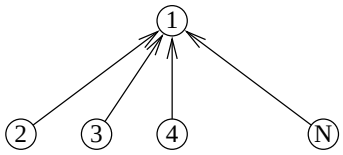
1	*2*	*3*	...	*N*	*N*
1	*2*	*3*	...	*N-1*	*N*

Find in $\Omega(N)$.

Beispiel 7.2: *Wir Vertauschen die Argumente bei der union-Operation.*

$P =$ ① ② ③ ④ $\cdots$ Ⓝ *und union(2,1); union(3,1); ... union(N,1).*

Dann erhalten wir den folgenden Baum



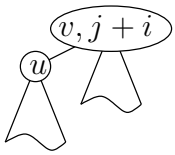
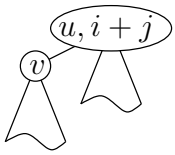
und die Operation find(2) geht in $O(1)$.

$\Rightarrow$ Union-by-Size; kleinere Anzahl nach unten.

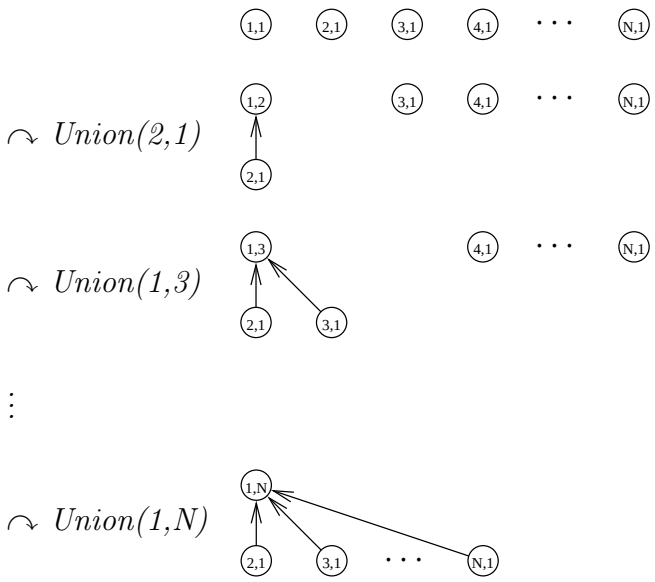
$\Rightarrow$ Maximale Tiefe (längster Weg von Wurzel zu Blatt) $= log_2 N$.
Beachte: Das Verhältnis von $log_2 N$ zu $N = 2^{log_2 N}$ ist wie N zu 2^N.

7.3 Algorithmus Union-by-Size

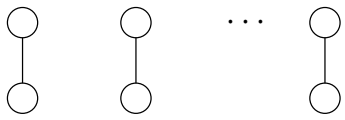
Prozedur union$((u,i),\,(v,j))$

```
   /* i, j sind die Anzahlen der Elemente in dem Mengen. Muss
      mitgeführt werden. */
1  if i ≤ j then
2  |   
   |                                                    /* u unter v. */
3  else
4  |   
   |                                                    /* v unter u. */
5  end
```

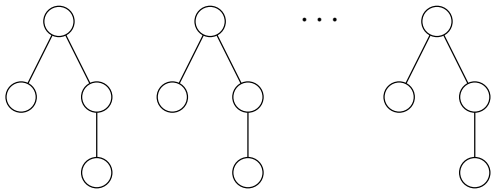

Beispiel 7.3: *Union by Size:*



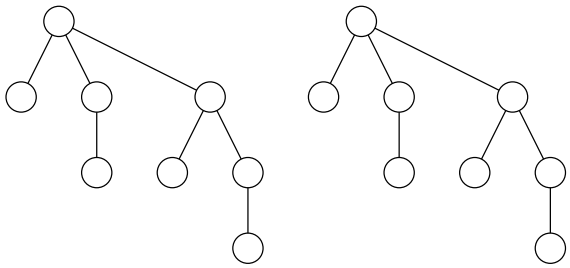
Wie können tiefe Bäume durch Union-by-Size entstehen?

1. 
2. 

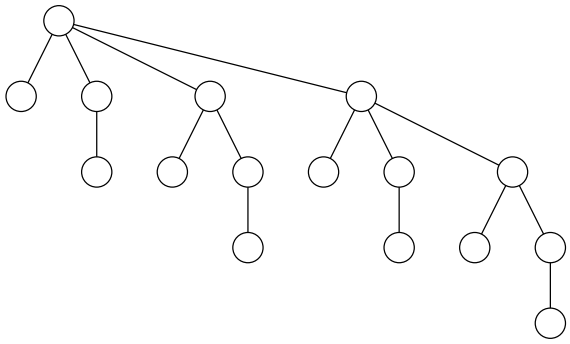
3.



4.



5.



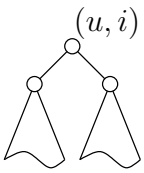
Vermutung: N Elemente $\Rightarrow$ Tiefe $\leq log_2 N$.

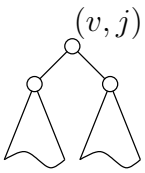
Satz 7.1: *Beginnend mit* $P = \{\{1\}, \{2\}, \ldots, \{n\}\}$ *gilt, dass mit Union-by-Size für jeden entstehenden Baum* T *gilt: Tiefe(*T*)* $\leq \log_2 |T|$. *(*$|T|$ *= #Elemente von* T *= #Knoten von* T*)*

Beweis. Induktion über die Anzahl der ausgeführten Operationen union(u, v), um T zu bekommen.

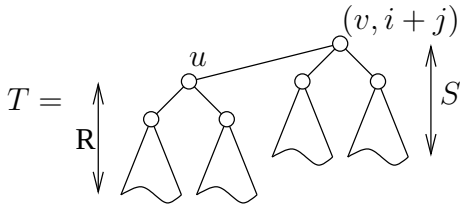
Induktionsanfang: kein union(u,v) ✓ Tiefe$(u) = 0 = \log_2 1$ $(2^0 = 1)$

Induktionsschluss: T durch union(u, v).

$R =$ 

$S =$ 

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

$$\text{Tiefe}(T) = \text{Tiefe}(R) + 1 \leq (\log_2 |R|) + 1 = \log_2(2|R|) \leq |T|$$

$(2^x = |R| \Rightarrow 2 \cdot 2^x = 2^{x+1} = 2|R|)$

Die Tiefe wird immer nur um 1 größer. Dann mindestens Verdopplung der #Elemnte.

□

Mit Bäumen und Union-by-Size Laufzeit der Schleife:

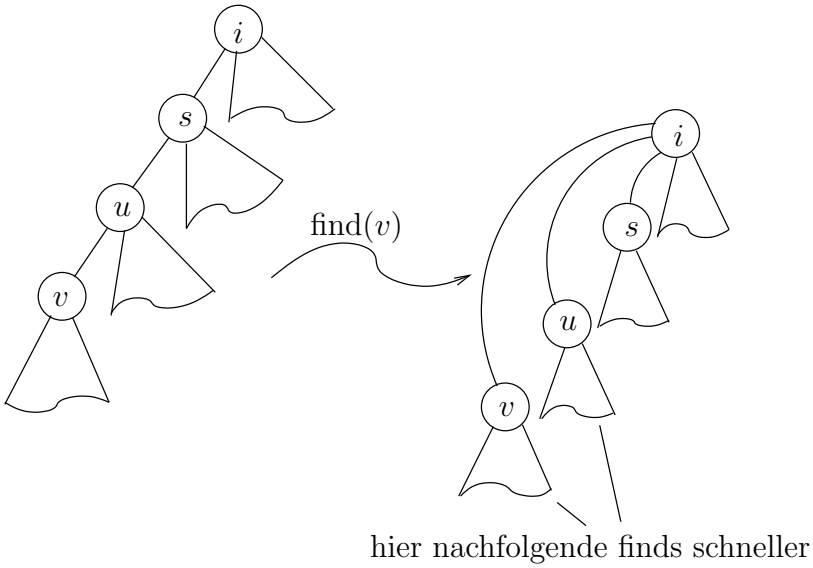
$\leq 2|E|$-mal Find(u): $O(|E| \log |V|)$ $\quad$ ($\log |V|$ ... Tiefe der Bäume)

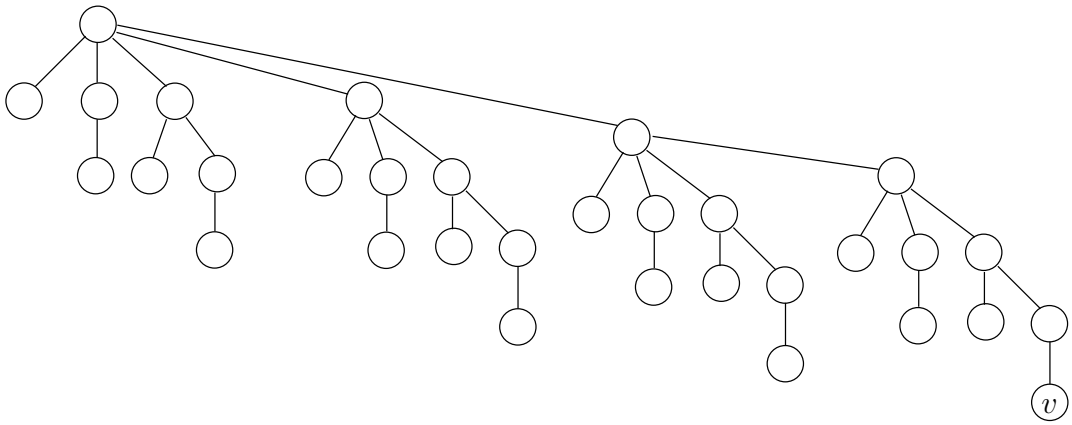
$n-1$-mal Union(u, v): $O(n)$

Rest $O(|E|)$. Also insgesamt $O(|E| \cdot \log |V|)$

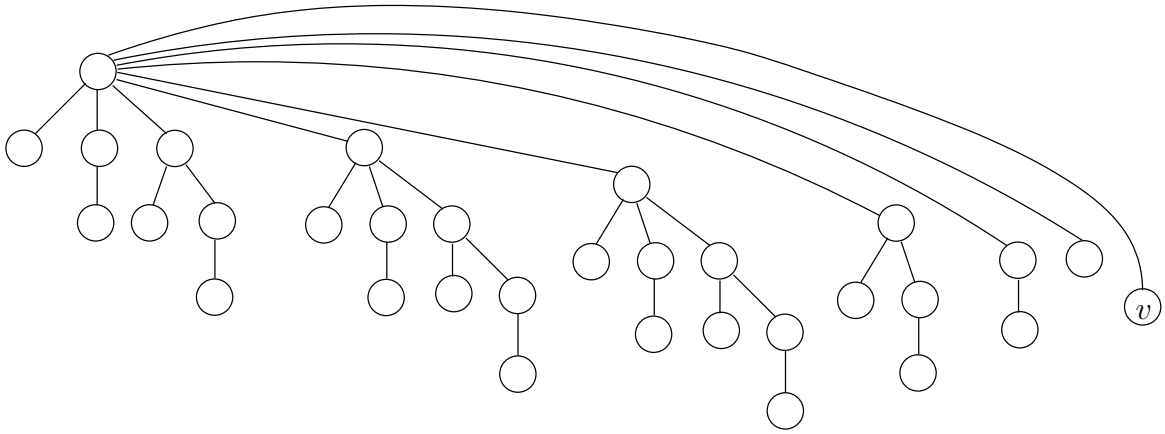
Vorne: $O(|V|^2)$. Für dichte Graphen ist $|E| = \Omega(\frac{|V|^2}{\log |V|})$. → So keine Verbesserung erkennbar.

Beispiel 7.1 (*Wegkompression*):





$\curvearrowright$ find(v)



Im Allgemeinen $\Omega(\log N)$ und $O(\log N)$

7.4 Algorithmus Wegkompression

Prozedur find(v)

```
1  S = ∅;          /* leerer Keller, etwa als Array implementieren */
2  push(S, v);
3  while P[v] ≠ v do            /* In v steht am Ende die Wurzel. */
4  |  v = P[v];
5  |  push(S, v);
6  end
   /* Auch Schlange oder Liste als Datenstruktur ist möglich. Die
      Reihenfolge ist egal. */
7  foreach w ∈ S do
   |  /* Alle unterwegs gefundenen Knoten hängen jetzt unter der
   |     Wurzel. */
8  |  P[w] = v;
9  end
10 return v
```

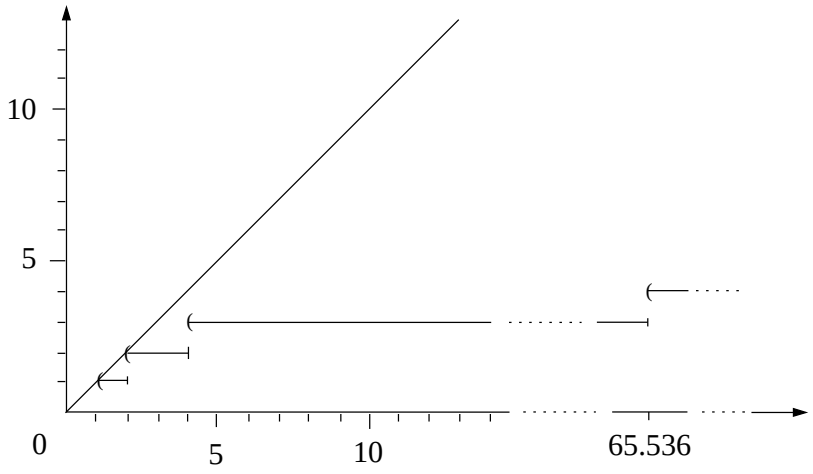
Laufzeit $O(\log |V|)$ reicht immer, falls Union-by-Size dabei.

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 $\log^*$.

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

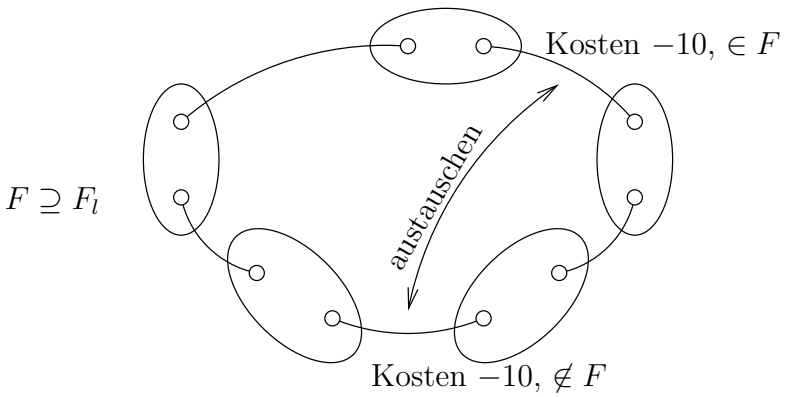
$$
\begin{aligned}
\log^* n &= \min\{s | \log^{(s)}(n) \leq 1\} \\
\log^* 2^{(2^{(2^{(2^{(2^2)})})})} &= 7 \\
\log^{(s)}(n) &= \underbrace{\log(\log(\ldots(\log}_{s\text{-mal}}(n))\ldots))
\end{aligned}
$$

Kruskal bekommt dann eine Zeit von $O(|V|+|E|\cdot\log^*|V|)$. Das ist 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.



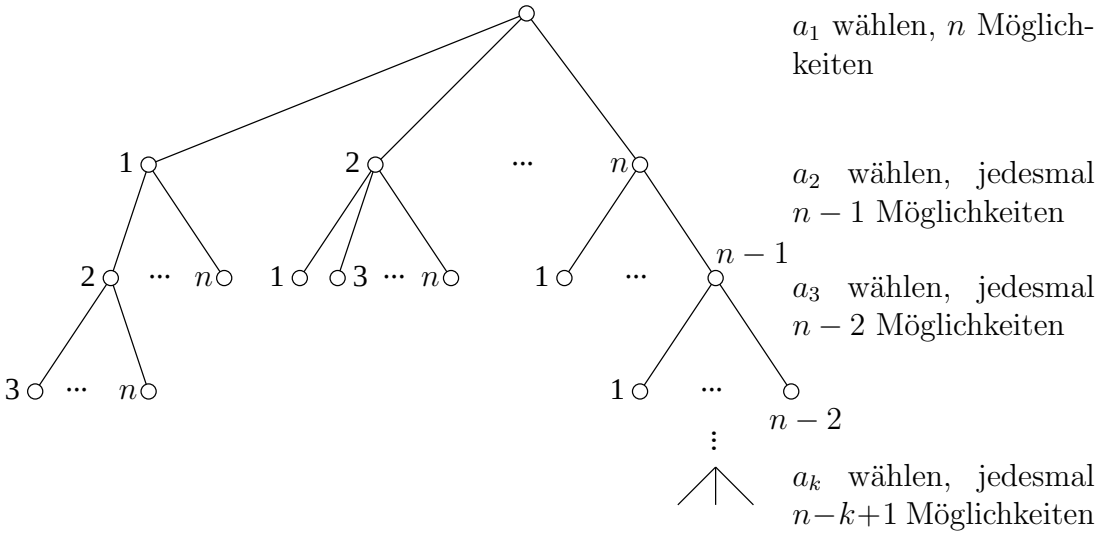
Alle Kanten, die zwei Komponenten verbinden, haben Kosten ≥ -10.

Beachte: Negative Kosten sind bei Kruskal kein Problem, nur bei branch-and-bound (vorher vergrößern)!

Binomialkoeffizient $\binom{n}{k}$ für $n \geq k \geq 0$

$$\begin{aligned}
\binom{n}{k} &= \frac{n!}{(n \cdot k) \cdot k!} = \frac{\overbrace{n(n-1)\dots(n-k+1)}^{\text{oberste } k \text{ Faktoren von } n!}}{k!} \qquad 0! = 1! = 1 \\
\binom{n}{k} &= \#\text{Teilmengen von } \{1,\dots,n\} \text{ mit genau } k \text{ Elementen} \\
\binom{n}{1} &= n, \quad \binom{n}{2} = \frac{n(n-1)}{2}, \quad \binom{n}{k} \leq n^k
\end{aligned}$$

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



Der Baum hat $\underbrace{n(n-1)\dots\overbrace{(n-k+1)}^{=n-(k-1)}}_{\substack{k \text{ Faktoren (nicht } k-1\text{, da} \\ 0,1,\dots,k+1 \text{ genau } k \text{ Zahlen)}}} = \frac{n!}{(n-k)!}$ Blätter.

Jedes Blatt entspricht genau einer Folge $(a_1, \dots, a_k)$. Jede Menge aus k Elementen kommt $k!$-mal vor: $\{a_1, \dots, a_k\}$ ungeordnet, geordnet als

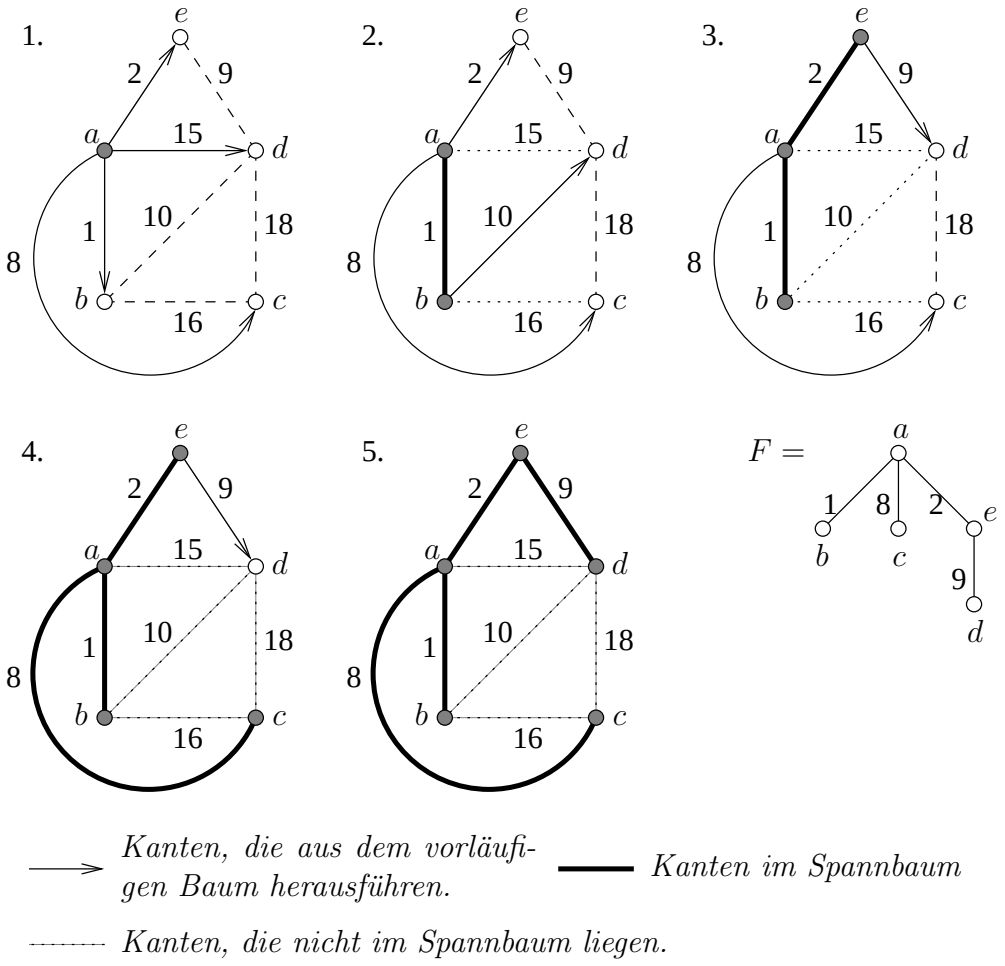
$$(a_1, \dots, a_k), (a_2, a_1, \dots, a_k), \dots, (a_k, a_{k-1}, \dots, a_2, a_1).$$

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

7.5 Algorithmus Minimaler Spannbaum nach Prim (1963)

Dieser Algorithmus verfolgt einen etwas anderen Ansatz als der von Kruskal. Hier wird der Spannbaum schrittweise von einem Startknoten aus aufgebaut. Dazu werden in jedem Schritt die Kanten betrachtet, die aus dem bereits konstruierten Baum herausführen. Von allen diesen Kanten wählen wir dann eine mit *minimalem Gewicht* und fügen sie dem Baum hinzu. Man betrachte das folgende Beispiel.

Beispiel 7.4:



Algorithmus 11: Minimaler Spannbaum (Prim)

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

Output : $F =$ Menge von Kanten eines minimalen Spannbaumes.

```
1 Wähle einen beliebigen Startknoten s ∈ V;
2 F = ∅;
  /* Q enthält immer die Knoten, die noch bearbeitet werden
     müssen. */
3 Q = V \ {s};
4 while Q ≠ ∅ do
  |  /* V \ Q sind die Knoten im Baum. In M sind alle Kanten,
  |     die einen Knoten in V \ Q und einen in Q haben. Also alle
  |     Kanten, die aus dem Baum herausführen. */
5 |  M = {{v, w} ∈ E | v ∈ V \ Q, w ∈ Q};
6 |  {v, w} = eine Kante mit minimalen Kosten in M;
7 |  F = F ∪ {{v, w}};
8 |  Q = Q \ {w};                    /* w ist jetzt Teil des Baumes. */
9 end
```

Korrektheit mit der Invariante:

- Es gibt einen minimalen Spannbaum $F \supseteq F_l$.

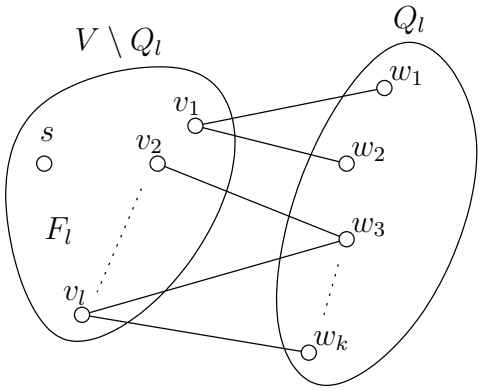
$$F_0 = \emptyset$$

- Q_l ist die Menge Q des Algorithmus nach dem l-ten Schritt. Also die Menge der Knoten, die nach l Schritten noch nicht im Spannbaum sind.

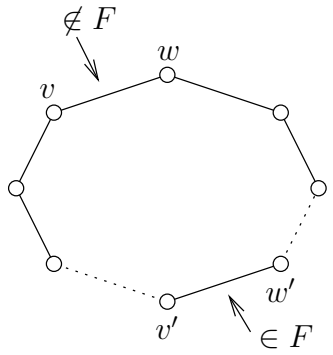
$$\begin{aligned} Q_0 &= V \setminus \{s\} \\ Q_l &= V \setminus \{u \mid u \in F_l \text{ oder } u = s\} \end{aligned}$$

Beweis. $l = 0$ ✓. Jeder Spannbaum enthält den Startknoten. Das ergibt sich aus der Definition des Spannbaumes.

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 $K(\{v, w\} = K(\{v', w'\})$. Wir können in F die Kante $\{v', w'\}$ durch $\{v, w\}$ ersetzen und haben immernoch einen minimalen Spannbaum. Damit gilt die Invariante für $l + 1$. □

Laufzeit von Prim:

- $n - 1$ Läufe durch 4-8.
- 5. und 6. 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 $v \notin Q$.
- Array $key[1 \ldots n]$ of real mit der Intention: Für alle $w \in Q$ ist $key[w] =$ minimale Kosten einer Kante $\{v, w\}$, wobei $v \notin Q$. ($key[w] = \infty$, gdw. keine solche Kante existiert)

- Außerdem Array $kante[1, \ldots, n]$ of $\{1, \ldots, n\}$. Für alle $w \in Q$ gilt:

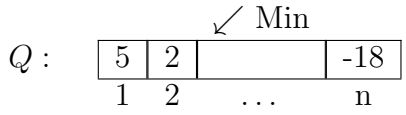
$$kante[w] = v \Longleftrightarrow \{v, w\} \text{ ist eine Kante minimaler Kosten mit } v \notin Q.$$

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



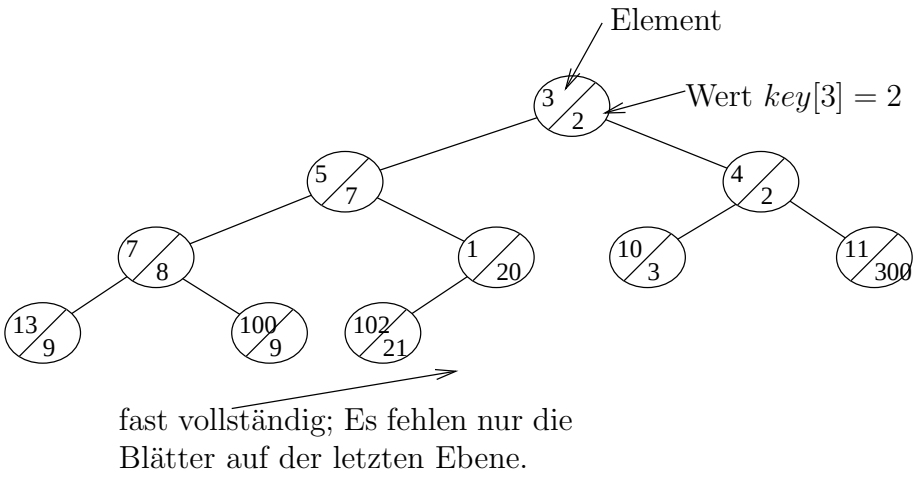
- Indices = Elemente
- Einträge in Q = Schlüsselwerte, Sonderwert (etwa $-\infty$) bei „nicht vorhanden“.

Zeiten:

- `Insert(`v, s`)` in $O(1)$ (sofern keine gleichen Elemente mehrfach auftreten)
- `Min` in $O(1)$
- `DeleteMin` in $O(1)$ 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.

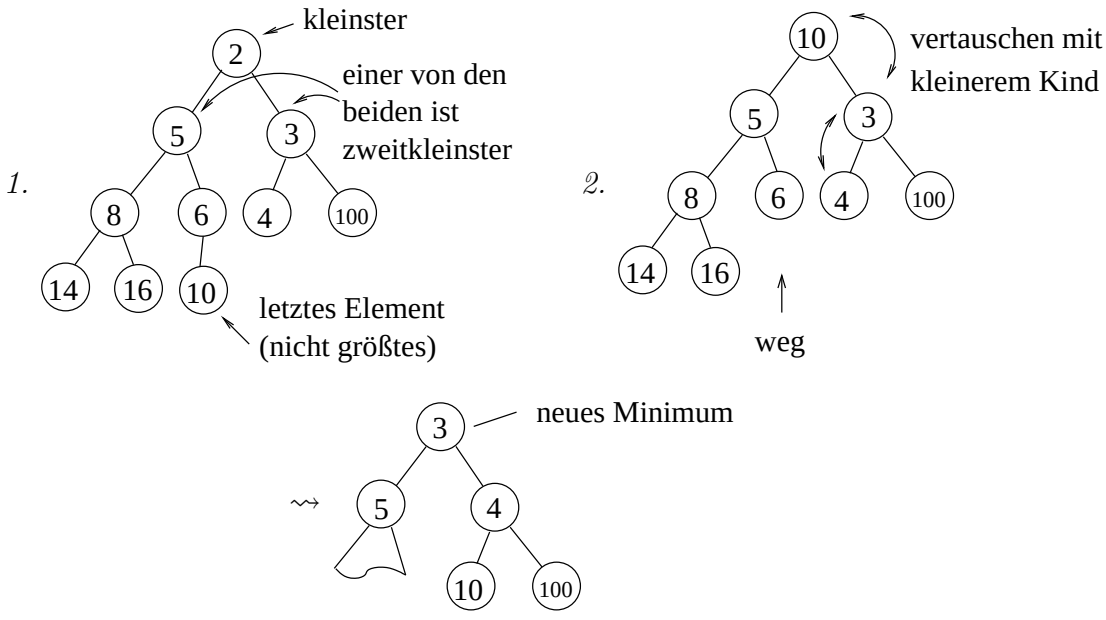


Tiefe n, dann $2^n - 1$ innere Knoten und 2^n Blätter. Insgesamt $2^{n+1} - 1$ Elemente.

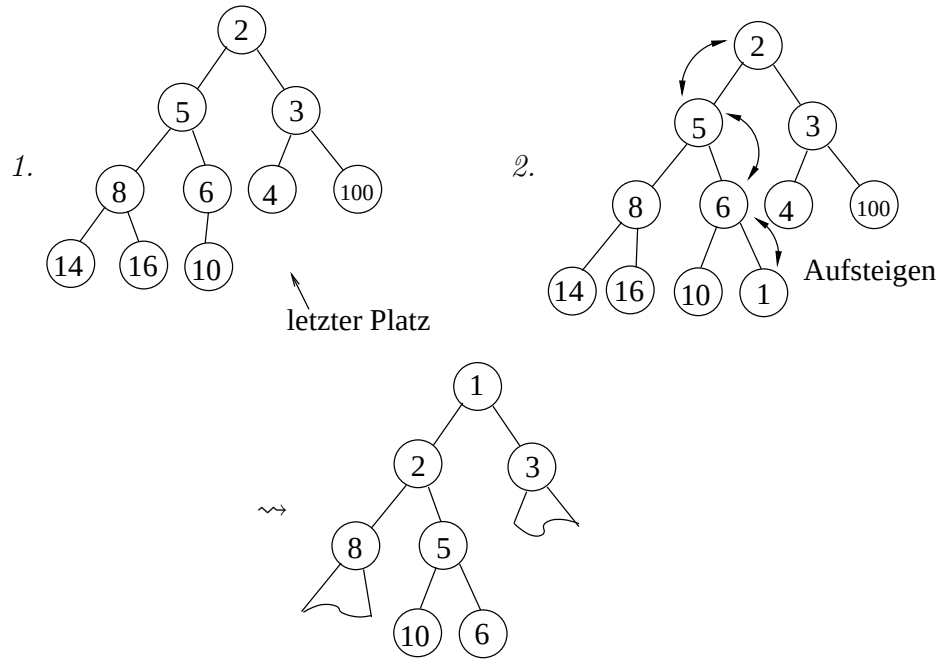
Heapeigenschaft: Für alle u, v gilt:

$$u \text{ Vorfahr von } v \Longrightarrow key[u] \leq key[v]$$

Beispiel 7.5(Minimum löschen): *Es sind nur die Schlüsselwerte key[i] eingezeichnet.*



Beispiel 7.6(Einfügen): *Element mit Wert 1 einfügen.*



Die Operationen der Priority Queue mit Heap, sagen wir Q.

Prozedur Min(Q)

```
1 return Element der Wurzel;
  /* Laufzeit: O(1) */
```

Prozedur DeleteMin(Q)

```
1 Nimm „letztes“ Element, setze es auf Wurzel;
2 x = Wurzel(-element);
3 while key[x] ≥ key[linker Sohn] oder key[x] ≥ key[rechter Sohn] do
4 |  Tausche x mit kleinerem Sohn;
5 end
  /* Laufzeit: O(log n), da bei n Elementen maximal log n
     Durchläufe. */
```

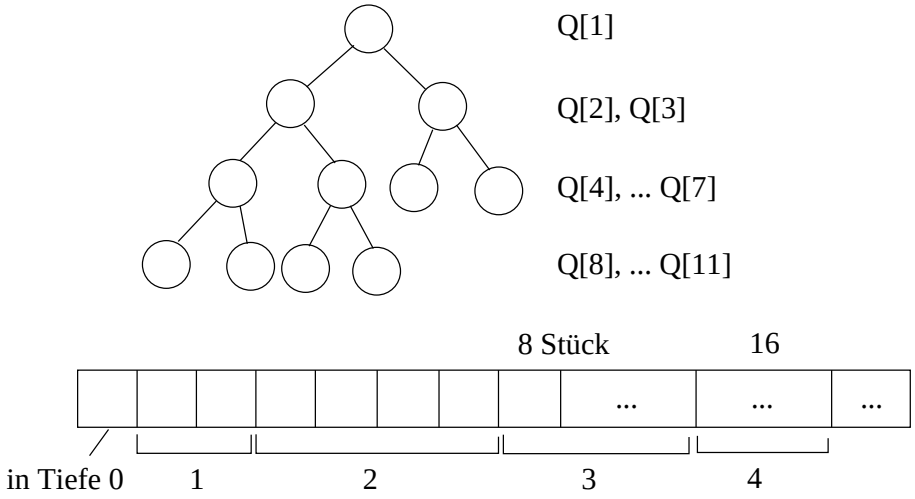
Prozedur Insert(Q, v, s)

```
1 key[v] = s;
2 Setze v als letztes Element in Q;
3 while key[Vater von v] > key[v] do
4 |  Vertausche v mit Vater;
5 end
  /* Laufzeit: O(log n) bei n Elementen. */
```

Vergleich der Möglichkeiten der Priority Queue:

	Min	DeleteMin	Insert	Element finden
Heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n)$
Array oder Liste	$O(1)$	$O(n)$	$O(1)$	$O(1)$ (Array), $O(n)$ (Liste)

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



Prozedur Vater(Q, i)

```
   /* i ist Index aus 1,...,n */
1  if i ≠ 1 then
2  |  return ⌊i/2⌋;
3  else
4  |  return i;
5  end
```

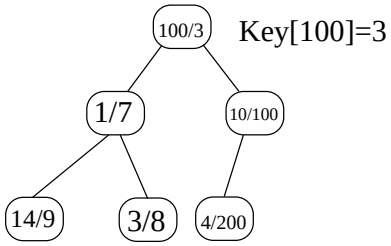
Prozedur linker Sohn(Q, i)

```
1  return 2i;
```

Prozedur rechter Sohn(Q, i)

```
1  return 2i + 1;
```

Beispiel 7.7: *Ein abschließendes Beispiel.*



$Q[1] = 100$ $key[100] = 3$
$Q[2] = 1$ $key[2] = 7$
$Q[3] = 10$ $key[10] = 100$
$Q[4] = 14$ $key[14] = 9$
$Q[5] = 3$ $key[3] = 3$
$Q[6] = 4$ $key[4] = 200$

... wenn die Elemente nur einmal auftreten, kann man key auch direkt als Array implementieren.

Suchen nach Element oder Schlüssel wird nicht unterstützt.

7.6 Algorithmus (Prim mit Q in Heap)

Algorithmus 12: Prim mit Heap

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

Output : F = Menge von Kanten eines minimalen Spannbaumes.

```
1  F = ∅;
2  Wähle einen beliebigen Startknoten s ∈ V;
3  foreach v ∈ Adj[s] do
4  |  key[v] = K({s, v});
5  |  kante[v] = s;
6  end
7  foreach v ∈ V \ ({s} ∪ Adj[s]) do
8  |  key[v] = ∞;
9  |  kante[v] = nil;
10 end
11 Füge alle Knoten v ∈ V \ {s} mit Schlüsselwerten key[v] in Heap Q ein;
12 while Q ≠ ∅ do
13 |  w = Min(Q);
14 |  DeleteMin(Q);
15 |  F = F ∪ {{kante[w], w}};
16 |  foreach u ∈ (Adj[w] ∩ Q) do
17 |  |  if K({u, w}) < key[u] then
18 |  |  |  key[u] = K({u, w});
19 |  |  |  kante[u] = w;
20 |  |  |  Q anpassen;
21 |  |  end
22 |  end
23 end
```

Korrektheit mit zusätzlicher Invariante. Für alle $w \in Q_l$ gilt:

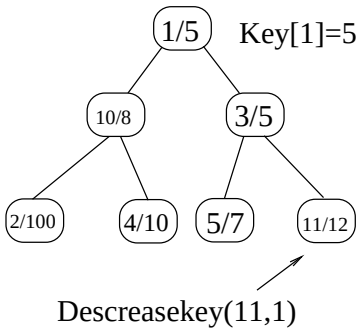
$$\begin{aligned} key_l[w] &= \text{minimale Kosten einer Kante } \{v, w\} \text{ mit } v \notin Q_l \\ key_l[w] &= \infty, \text{ wenn eine solche Kante nicht existiert.} \end{aligned}$$

Laufzeit:

1. - 11.	$O(n) + O(n \cdot logn)$ für das Füllen von Q. (Füllen von Q in $O(n)$ – interessante Übungsaufgabe)
12. - 22.	$n - 1$ Läufe
13. - 15.	Einmal $O(logn)$, insgesamt $O(n \cdot \log n)$
16. - 22.	Insgesamt $O(\|E\|) + \|E\|$-mal Anpassen von Q.

Anpassen von Heap Q

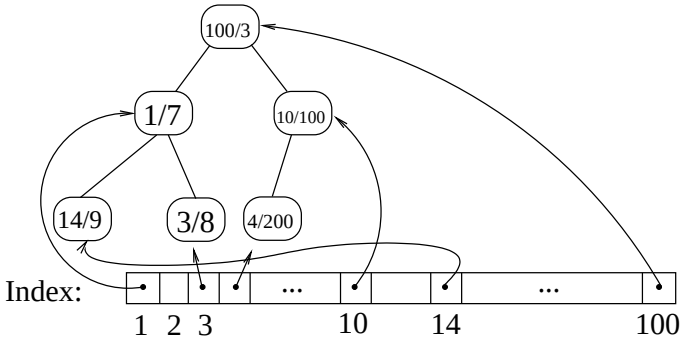
Beispiel 7.8(Operation DecreaseKey(v,s)): *$s < key[v]$, neuer Schlüssel*



Prozedur DecreaseKey(Q, v, s)

```
1 Finde Element v in Q;          /* Wird von Heap nicht unterstützt! */
2 key[v] = s;
3 while key[Vater von v] > key[v] do
4 |   Tausche v mit Vater;
5 end
  /* Laufzeit: O(n) + O(log n) */
```

zum Finden: „Index drüberlegen“



Direkte Adressen:

$$Index[1] = 2, Index[3] = 5, Index[4] = 6, \ldots, Index[100] = 1$$

Mit direkten Adressen geht finden in $O(1)$. Das $Index$-Array muss beim Vertauschen mit aktualisiert werden. Das geht auch in $O(1)$ für jeden Tauschvorgang.

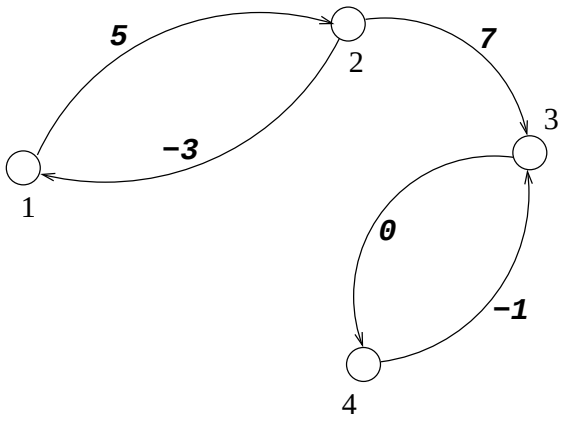
Falls die Grundmenge zu groß ist, kann man einen Suchbaum verwenden. Damit geht das Finden dann in $O(\log n)$.

Falls direkte Adressen dabei: Einmaliges Anpassen von Q (und Index) $O(\log |V|)$. Dann hat Prim insgesamt die gleiche Laufzeit, $O(|E| \log |V|)$, wie Kruskal mit Union-by-Size. Beachte vorher $O(|E| \cdot |V|)$.

8 Kürzeste Wege

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

Also etwa:



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

Ist $W = (v_0, v_1, \ldots, v_k)$ 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

$$\begin{aligned} K(3,4) &= 0, \\ K(4,3) &= -1 \quad \text{und} \\ K(3,4,3,4) &= -1, \\ K(3,4,3,4,3,4) &= -2, \\ &\ldots \end{aligned}$$

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*

- $\mathrm{Dist}(u,v) = \min\{K(W) \mid W$ *ist einfacher Weg von u nach v* $\}$, *sofern es einen Weg $u \circ\!\!\longrightarrow\!\!\circ\, v$ gibt.*
- $\mathrm{Dist}(u,v) = \infty$, *wenn es keinen Weg $u \circ\!\!\longrightarrow\!\!\circ\, v$ gibt.*
- $Dist(v,v) = 0$

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

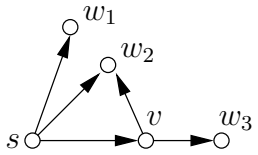
Ist $u = v_0 \to v_1 \to v_2 \cdots v_{k-1} \to v_k = v$ ein kürzester Weg von u nach v, so sind alle Wege $v_0 \to v_i$ oben auch kürzeste Wege. Deshalb ist es sinnvoll, das *single source shortest path-Problem* zu betrachten, also alle kürzesten Wege von einem Ausgangspunkt aus zu suchen. Ist s unser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

$$s \to v$$

Gibt uns diese Kante einen kürzesten Weg von s nach v? Im Allgemeinen nicht! Nur unter der Einschränkung, dass die Kostenfunktion $K : E \to \mathbb{R}^{\geq 0}$ ist, also keine Kosten < 0 erlaubt sind. Diese Bedingung treffen wir zunächst einmal bis auf weiteres.

Also, warum ist der Weg $s \to v$ ein kürzester Weg? (Die nachfolgende Aussage gilt bei negativen Kosten so nicht!) Jeder andere Weg von s aus hat durch seine erste Kante $s \to w$ mindestens die Kosten $K(s, v)$.

Der *Greedy-Ansatz* funktioniert für den ersten Schritt. Wie bekommen wir einen weiteren kürzesten Weg von s aus?

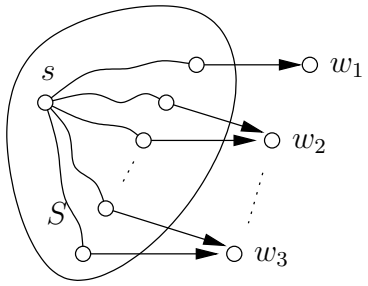


Wir schauen uns die zu s und v adjazenten Knoten an. Oben w_1, w_2, w_3. Wir ermitteln

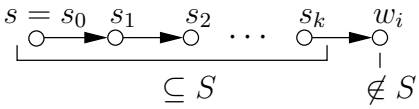
- zu w_1 $K(s, w_1)$
- zu w_2 $\min\{K(s, w_2), K(s, v) + K(v, w_2)\}$
- zu w_3 $K(s, v) + K(v, w_3)$.

Ein minimaler dieser Werte gibt uns einen weiteren kürzesten Weg. Ist etwa $K(s, v) + K(v, w_2)$ minimal, dann gibt es keinen kürzeren Weg $s \to w_2$. 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 alle Knoten 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 $s \circ\!\!\longrightarrow\!\!\circ\, w_i$. Wie vorher sieht man, dass es wirklich keinen kürzeren Weg $s \circ\!\!\longrightarrow\!\!\circ\, w_i$ gibt.

Das Prinzip: Nimm immer den nächstkürzeren Weg von S ausgehend.

8.1 Algorithmus (Dijkstra 1959)

Algorithmus 13: Dijktstras Algorithmus

Input : $G = (V, E),\ K : E \to \mathbb{R}^{\geq 0},\ |V| = n, s \in V$
Output : Array $D[1 \ldots n]$ of real mit $D[v] = \text{Dist}(s, v)$ für $v \in V$
Data : $S \to$ Menge der Knoten, zu denen ein kürzester Weg gefunden ist

```
1  D[s] = 0;
2  S = {s};
3  Q = V \ S;
4  for i = 1 to n − 1 do     /* Wir suchen noch n − 1 kürzeste Wege. */
5  |  foreach w ∈ Q do
   |  |  /* Betrachte alle Kanten (v, w) mit v ∈ S */
6  |  |  D[w] = min{D[v] + K(v, w) | v ∈ S und (v, w) ∈ E};
7  |  end
8  |  w = ein w ∈ Q mit D[w] minimal;
9  |  S = S ∪ {w};
10 |  Q = Q \ {w};
11 end
```

Korrektheit mit der Invariante: Für alle $w \in S_l$ ist $D[w]$ = Kosten eines kürzesten Weges.

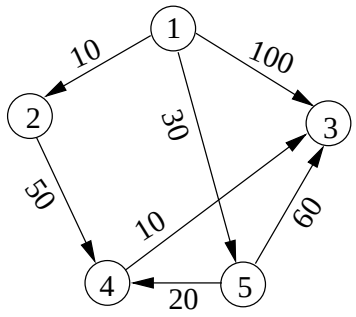
Das Argument gilt wie oben bereits vorgeführt.

Zwecks Merken der Wege das Array $\pi[1,\ldots,n]$ of $\{1,\ldots,n\}$ mit

$$\pi[u] = v \iff \text{Ein kürzester Weg } s \circ\!\!\longrightarrow\!\!\circ\, v \quad \text{ist } (s,\ldots,u,v).$$

π kann leicht im Algorithmus mitgeführt werden. Es ist das Vaterarray des *Kürzesten-Wege-Baumes* mit Wurzel s.

Beispiel 8.1: *Bestimmung der kürzesten Wege von s aus in folgendem Graphen.*

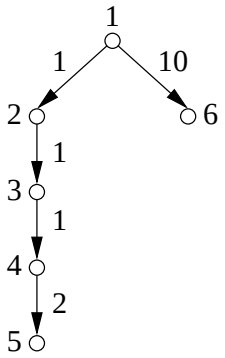


S	w	$D[1]$	$D[2]$	$D[3]$	$D[4]$	$D[5]$
$\{1\}$	/	0	10	100	∞	30
$\{1,2\}$	2	0	10	100	60	30
$\{1,2,5\}$	5	0	10	90	50	30
$\{1,2,5,4\}$	4	0	10	60	50	30
$\{1,2,5,4,3\}$	3	0	10	60	50	30

Die Wege werden der Länge nach gefunden:

Beispiel 8.2:

$G =$



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

Kürzester-Wege-Baum:

$$\begin{aligned}\pi[1] &= 1,\\ \pi[2] &= 1,\\ \pi[3] &= 2,\\ \pi[4] &= 5,\\ \pi[5] &= 4,\\ \pi[6] &= 1\end{aligned}$$

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 zwischen S und Q verlaufen. Q und S sind als boolesche Arrays mit

$$Q[v] = \mathit{true} \iff v \in Q$$

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, dann Hashing oder Suchbaum verwenden.

In Q ist ein Dictionary implementiert. Jede Operation erfolgt in $O(1)$. Das Ziel ist eine bessere Implementierung.

Für welche $w \in Q$ kann sich $D[w]$ in 6. nur ändern? Nur für diejenigen, die adjazent zu dem w des *vorherigen* Laufes sind. Dann sieht es etwa so aus:

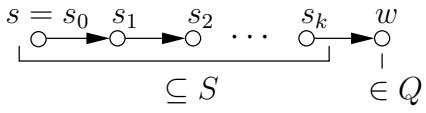
Algorithmus 14: Dijktstras Algorithmus (ohne mehrfache Berechnung derselben $D[w]$)

Input : $G = (V, E),\ K : E \to \mathbb{R}^{\geq 0},\ |V| = n, s \in V$
Output : Array $D[1 \ldots n]$ of real mit $D[v] = \text{Dist}(s, v)$ für $v \in V$
Data : $S \to$ Menge der Knoten, zu denen ein kürzester Weg gefunden ist

```
1  D[s] = 0;
2  D[w] = K(s,w) für w ∈ Adj[s];
3  D[v] = ∞ für v ∈ V \ ({s} ∪ Adj[s]);
4  S = {s};
5  Q = V \ S;
6  for i = 1 to n − 1 do     /* Wir suchen noch n − 1 kürzeste Wege. */
7  |  w = ein w ∈ Q mit D[w] minimal;
8  |  S = S ∪ {w};
9  |  Q = Q \ {w};
10 |  foreach v ∈ Adj[w] do
   |  |  /* D[v] anpassen für v adjazent zu w */
11 |  |  if D[w] + K(w,v) < D[v] then
12 |  |  |  D[v] = D[w] + K(w,v);
13 |  |  end
14 |  end
15 end
```

Korrektheit mit zusätzlicher Invariante:

Für $w \in Q_l$ ist $D_l[w]$ = minimale Kosten eines Weges der Art



Man kann auch leicht zeigen:

> Für alle $v \in S_l$ ist $D_l[v] \leq D_l[w_l]$ mit w_l = das Minimum der l-ten Runde. Damit ändert 12. nichts mehr an $D[v]$ für $v \in S_l$.

Laufzeit:

7.	insgesamt $n-1$-mal Minimum finden
8., 9.	insgesamt $n-1$-mal Minimum löschen
10.-14.	insgesamt $O(\|E\|)$, da dort jede Adjazenzliste nur einmal durchlaufen wird.

Mit Q als boolesches Array:

7.	$O(n^2)$ insgesamt. Das Finden eines neuen Minimums nach dem Löschen dauert $O(n)$.
8., 9.	$O(n)$ insgesamt, einzelnes Löschen dauert $O(1)$.
10.-14.	Bleibt bei $O(\|E\|)$.

Also alles in allem $O(n^2)$.

Aber mit Q benötigen wir die klassischen Operationen der Priority Queue, also Q als Heap, Schlüsselwert aus D.

7.	$O(n)$ insgesamt.
8., 9.	$O(n \cdot \log n)$ insgesamt.
10.-14.	$\|E\|$-mal Heap anpassen, mit DecreaseKey(Q, v, s) (vergleiche Prim, Seite 110) $O(\|E\| \cdot \log n)$. Zum Finden Index mitführen.

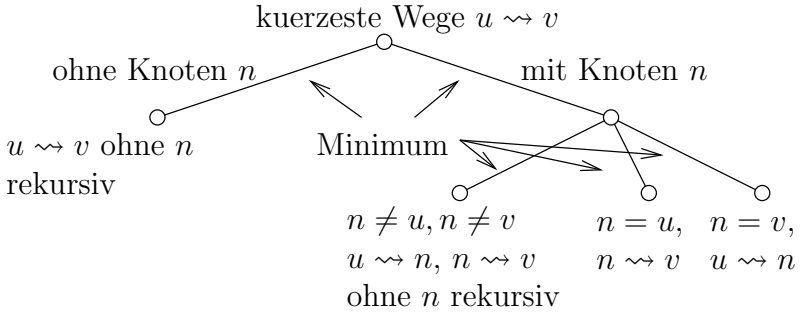
Also:

Q als boolesches Array:	$O(n^2)$ (Zeit fällt beim Finden der Minima an).
Q als heap mit Index:	$O(\|E\| \cdot \log n)$ (Zeit fällt beim DecreaseKey(Q, v, s) an).

Ist $|E| \cdot \log n \geq n^2$, also ist der Graph sehr dicht, dann ist ein Array besser (vergleiche Prim).

8.2 Algorithmus Floyd-Warshall

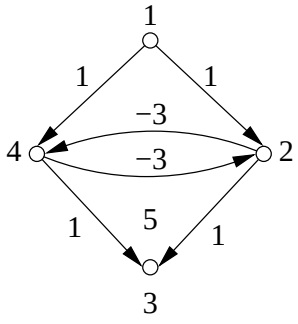
Wir beginnen einmal mit einer anderen Fallunterscheidung beim Backtracking:



Korrektheit: Ist $u, vneqn$ und ist ein kürzester Weg $u \circ\!\!\longrightarrow\!\!\circ v$ ohne n, dann wird dieser nach Induktionsvoraussetzung links gefunden. Enthalte jetzt jeder kürzeste Weg $u \circ\!\!\longrightarrow\!\!\circ v$ den Knoten n. Wird ein solcher unbedingt rechts gefunden?

Nur dann, wenn die kürzesten Wege $u \circ\!\!\longrightarrow\!\!\circ n$ und $n \circ\!\!\longrightarrow\!\!\circ v$ keinen weiteren gemeinsamen Knoten haben. Wenn sie einen gemeinsamen Knoten w haben, so $u \rightsquigarrow w \rightsquigarrow n \rightsquigarrow w \rightsquigarrow v$. Da dieser Weg kürzer ist als jeder Weg ohne n, muss $w \rightsquigarrow n \rightsquigarrow w$ ein Kreis der Länge < 0 sein.

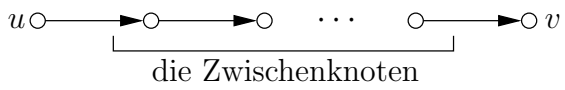
Beispiel 8.3(negative Kreise):

$G =$ 

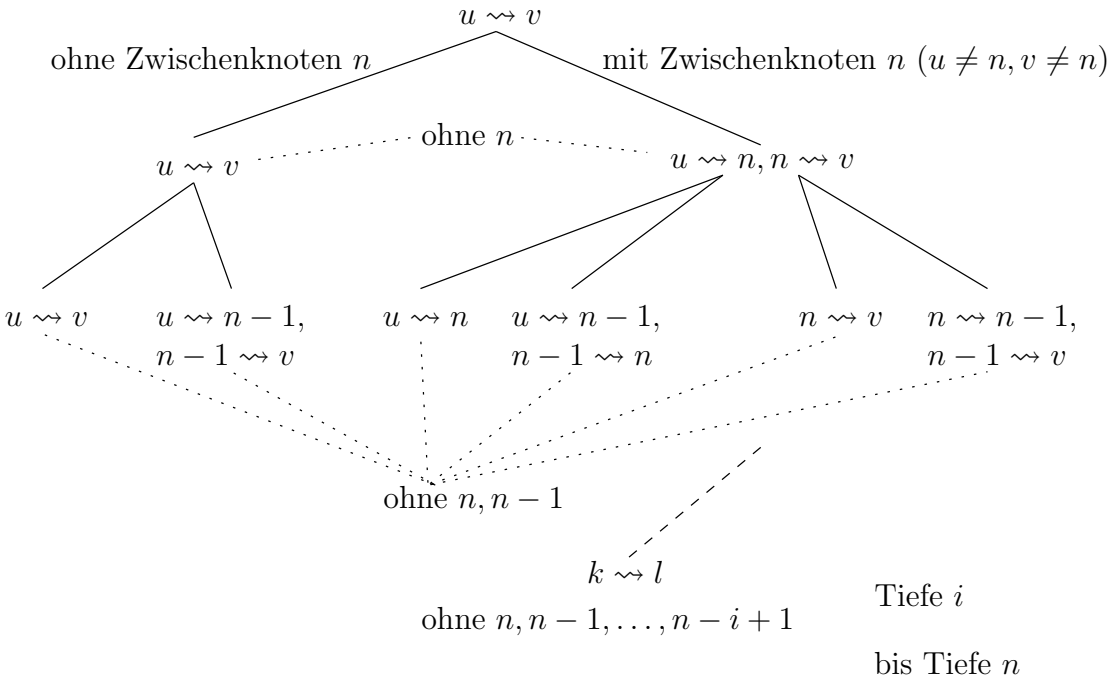
- *Kürzester Weg* $1 \circ\!\!\longrightarrow\!\!\circ 3$ *ohne* 4 *hat Kosten* 2,
- *kürzester Weg* $1 \circ\!\!\longrightarrow\!\!\circ 4$ *über* 2 *hat Kosten* -2,
- *kürzester Weg* $4 \circ\!\!\longrightarrow\!\!\circ 3$ *ebenfalls über* 2 *hat Kosten* -2
- *und* $(2, 4, 2)$ *ist Kreis der Kosten* -6.

Betrachten wir also jetzt $K : E \to \mathbb{R}$, aber so, dass keine Kreise < 0 existieren.

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



Also Backtracking:



Die rekursive Prozedur:

KW(u, v, k) für kürzeste Wege $u \circ\!\!\longrightarrow\!\!\circ\, v$ ohne Zwischenknoten $> k$. Kürzester Weg ergibt sich durch KW(u, v, n).

Rekursionsanfang KW$(u, v, 0)$ gibt Kante $u \circ\!\!\longrightarrow\!\!\circ\, v$. Wieviele verschiedene Aufrufe? $\Rightarrow O(n^3)$

Also dynamisches Programmieren:

$T[u, v, k]$ = kürzester Weg $u \circ\!\!\longrightarrow\!\!\circ\, v$ wobei Zwischenknoten $\subseteq \{1, \ldots, k\}$. (Also nicht unter $k + 1, \ldots, n$)

$$
\begin{aligned}
T[u, v, 0] &= K(u, v) \quad \text{für alle } u, v \\
T[u, v, 1] &= \min\{T[u, 1, 0] + T[1, v, 0], T[u, v, 0]\} \quad \text{für alle } u, v \\
&\vdots \\
T[u, v, n] &= \min\{T[u, n, n-1] + T[n, v, n-1], T[u, v, n-1]\} \quad \text{für alle } u, v
\end{aligned}
$$

Dies ist das *all pairs shortest path*-Problem. Das heißt es werden die kürzesten Wege zwischen *allen* Knoten bestimmt. *Wichtig:* G ohne Kreise mit Länge < 0.

Algorithmus 15: Floyd-Warshall-Algorithmus

Input : $G = (V, E)$ gerichteter Graph, $|V| = n$, $K : E \to \mathbb{R}$
Output : Ein kürzester Weg $u \rightsquigarrow v$ für alle Paare $u, v \in V \times V$

`/* Initialisierung */`

1 $$T[u,v] = \begin{cases} 0 & \text{für } u = v \\ K(u,v) & \text{für } (u,v) \in E \\ \infty & \text{für } u \neq v,\ (u,v) \notin E \end{cases}$$

`/* Wege über Zwischenknoten` $\leq i$ `*/`
2 **for** $k = 1$ **to** n **do**
3 | **foreach** $(u, v) \in V \times V$ **do** `/* alle geordneten Paare */`
4 | | $T[u,v] = \min\{T[u,v],\ T[u,k] + T[k,v]\}$;
5 | **end**
6 **end**

Invariante: Nach $l-$tem Lauf der Schleife enthält $T_l[u, v]$ die Länge eines kürzesten Weges $u \circ\!\!\longrightarrow\!\!\circ\, v$ mit Zwischenknoten $\subseteq \{1, \ldots, l\}$.

Wichtig: Keine negativen Kreise, denn ist in $l + 1$-ter Runde

$$T_l[u, l + 1] + T_l[l + 1, v] < T_l[u, v]$$

dann ist der Weg hinter $T_l[u, l + 1] + T_l[l + 1, v]$ ein einfacher!

Ist $u \rightsquigarrow l{+}1 \rightsquigarrow v$ kürzer als die kürzesten Wege $u \rightsquigarrow v$ ohne $l{+}1$, dann tritt oben die Situation $u \rightsquigarrow w \rightsquigarrow l{+}1 \rightsquigarrow w \rightsquigarrow v$ *nicht* ein, da sonst $K(w \rightsquigarrow l{+}1 \rightsquigarrow w) < 0$ sein müsste.

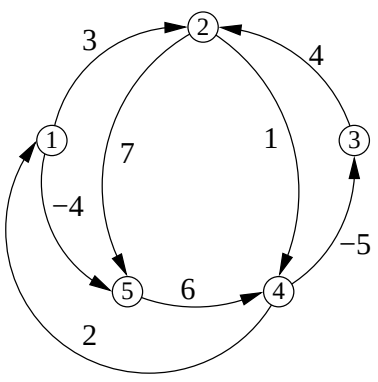
Erkennen von negativen Kreisen: Immer gilt, also bei beliebiger Kostenfunktion, dass Floyd-Warshall die Kosten eines Weges von $u \circ\!\!\longrightarrow\!\!\circ\, v$ in $T[u, v]$ liefert. Dann ist

$$T[u, u] < 0 \iff G \text{ hat Kreis } < 0.$$

Laufzeit: $O(n^3)$ (Vergleiche Dijkstra $O(|E| \log |V|)$ oder $O(|V|^2)$.)

Beispiel 8.4: *Die ersten Schritte des Algorithmus auf dem folgenden Graphen.*

$G =$



Am Anfang:

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

Bevor *k auf 2 geht:*

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

Es ist $A[i,j] = \min\{A[i,j], A[i,\mathbf{1}] + A[\mathbf{1},j]\}$

Direkte Wege, d.h. hier Wege mit Zwischenknoten 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 $K : E \to \mathbb{R}^{\geq 0}$. Hier nun $K(u, v) =$ die Kapazität von (u, v). Wir stellen uns vor, (u, v) stellt eine Verbindung dar, durch die etwas fließt (Wasser, Strom, Fahrzeuge, ...).

Dann besagt $K(u, v) = 20$ zum Beispiel

- ≤ 20 Liter Wasser pro Sekunde
- ≤ 10 produzierte Waren pro Tag
- ...

Definition 9.1(Flussnetzwerk): *Ein Flussnetzwerk ist ein gerichteter Graph* $G = (V, E)$ *mit* $K : V \times V \to \mathbb{R}^{\geq 0}$, $K(u, v) = 0$ *falls* $(u, v) \notin E$.
Außerdem gibt es zwei ausgezeichnete Knoten

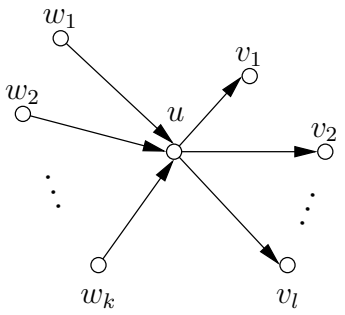
- $s \in V$, *Quelle (source)*
- $t \in V$, *Ziel (target, sink)*

Wir verlangen außerdem noch: Jeder Knoten $v \in V$ *ist auf einem Weg* $s \rightsquigarrow v \rightsquigarrow t$.

Ziel: Ein möglichst starker Fluss pro Zeiteinheit von s nach t.

Den Fluss durch $u \circ\!\!\longrightarrow\!\!\circ\, v \in E$ bezeichnen wir mit $f(u, v) \in \mathbb{R}^{\geq 0}$. Für einen Fluss f müssen die folgenden Bedingungen gelten:

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

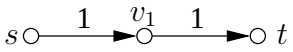


dann $\sum f(w_i, u) = \sum (u, v_i)$.

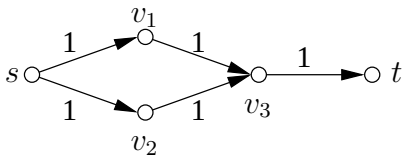
Prinzip: Methode der Erweiterungspfade (Ford - Fulkerson 1950er Jahre)

1. Beginne mit dem Fluss 0, $f(u,v) = 0$ für alle (u,v)
2. Suche einen Weg (Erweiterungspfad) $(s = v_0 \to v_1 \to v_2 \dots v_{k-1} \to v_k = t)$ 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.

Dieses Problem ist von ganz anderem Charakter als bisher, da die Anzahl der zunächst zulässigen Flüsse unendlich ist. Wir betrachten folgendes Flussnetzwerk:

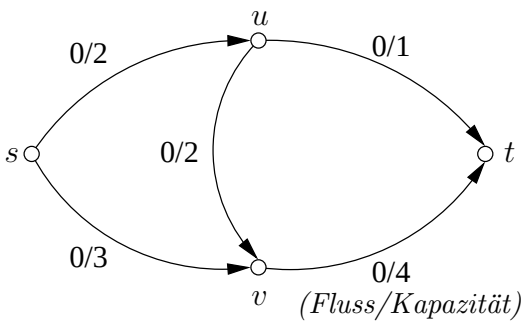


Ist $f(s,v_1) = f(v_1,t) < 1$ so haben wir einen erlaubten Fluss. Aber sogar maximale Flüsse gibt es unendlich viele:

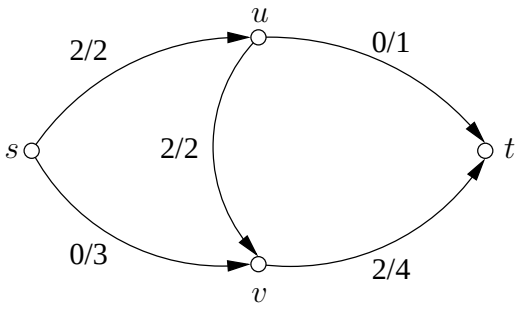


Jeder Fluss mit $f(s,v_1) + f(s,v_2) = 1$ ist maximal.

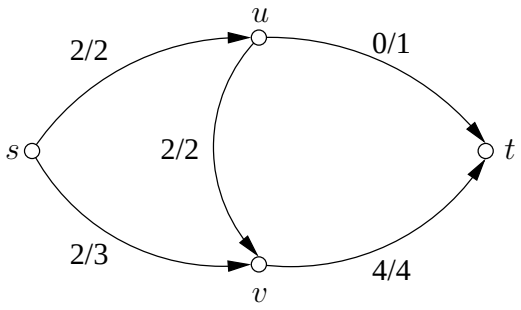
Beispiel 9.1(negative Flüsse): *Wir betrachten das folgende Flussnetzwerk:*



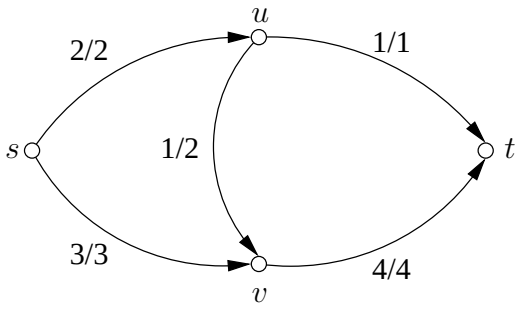
Weg (s,u,v,t) $+2$



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

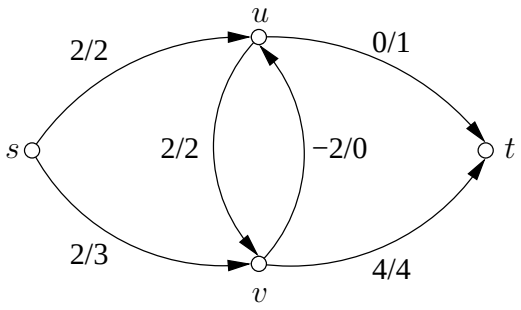


Jetzt gibt es keinen Erweiterungspfad mehr, aber der Fluss in



ist größer!

Mit negativen Flüssen: Immer ist $f(u, v) = -f(v, u)$.



Weg (s, v, u, t) $+1$ *gibt den maximalen Fluss. Wir lassen auch* $f(u, v) < 0$ *zu.*

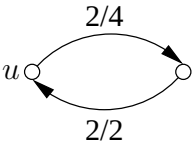
Definition 9.2(Fluss): *Ein Fluss ist eine Funktion* $f : V \times V \to \mathbb{R}$ *mit*

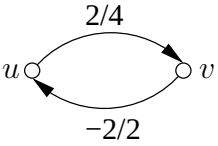
- $f(u, v) \leq K(u, v)$ *für alle* u, v *(Kapazitätsbedingung)*
- $f(u, v) = -f(v, u)$ *für alle* u, v *(Symmetrie)*
- *Für alle* $u \neq s,\ u \neq t$ *gilt* $\sum_{v \in V} f(u, v) = 0$ *(Kirchhoffsches Gesetz)*

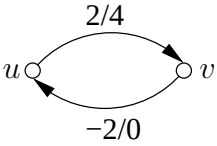
- $|f| = \sum_{v \in V} f(s,v)$ *ist der Wert von* f.

Problem: Wie groß ist der maximale Fluss $|f|$ in einem gegebenen Flussnetzwerk?

Noch einige Anmerkungen:

- 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 einer der Werte > 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 $f(u,v) = f(v,u) = 0$. Wir können (müssen) kürzen, aber 

  oder 

  geht.

Definition 9.3(Restnetzwerk): *Gegeben ist das Flussnetzwerk* $G = (V,E)$ *mit Kapazität* $K : V \times V \to \mathbb{R}^{\geq 0}$ *und ein zulässiger Fluss* $f : V \times V \to \mathbb{R}$.

- *Das Restnetzwerk* G_f *mit Restkapazität* K_f *ist gegeben durch:*

$$\begin{aligned} K_f(u,v) &= K(u,v) - f(u,v) \geq 0 \\ E_f &= \{(u,v) \mid K_f(u,v) > 0\} \\ G_f &= (V, E_f) \end{aligned}$$

  Es ist $K_f(u,v) \geq 0$, *da* $f(u,v) \leq K(u,v)$.

- *Ein Erweiterungspfad von* G *und* f *ist ein einfacher Weg in* G_f

$$W = (s = v_0 \to v_1 \to v_2 \ldots v_{k-1} \to v_k = t)$$

  Die Restkapazität von W *ist*

$$K_f(W) = \min\{K_f(v_i, v_{i+1})\},$$

  nach Definition gilt $K_f(v_i, v_{i+1}) \gneq 0$).

Es ist G_f mit der Restkapazität K_f ein Flussnetzwerk. *Beachte:* $K_f(u,v)$ ist für alle Paare $(u,v) \in V \times V$ definiert.

Wir betrachten jetzt $g : V \times V \to \mathbb{R}$ mit

$$\begin{aligned} g(v_i, v_{i+1}) &= K_f(W) > 0 \\ g(v_{i+1}, v_i) &= -K_f(W) = -g(v_i, v_{i+1}) \\ g(u, v) &= 0 \quad \text{für} (u, v) \notin W. \end{aligned}$$

Das ist ein zulässiger Fluss auf G_f. Das heißt: $|g| = K_f(W)$.

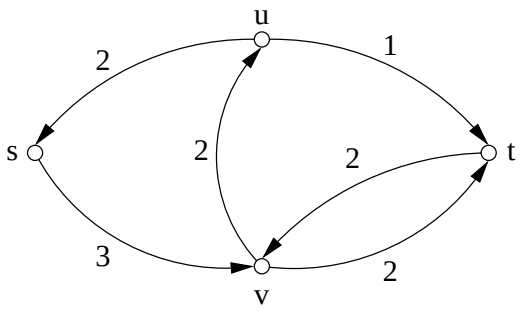
Tatsächlich gilt nun sogar folgendes:

Lemma 9.1: *Sei G, K, f Flussnetzwerk mit zulässigem Fluss f. Und sei G_f das zugehörige Restnetzwerk.*

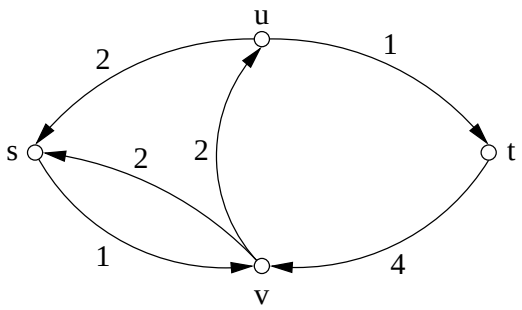
Sei jetzt g irgendein zulässiger Fluss auf G_f. Dann ist $f + g$ ein gültiger Fluss auf G und $|f + g| = |f| + |g|$.

Beispiel 9.2(Restnetzwerke): *Zunächst noch ein Beispiel. Wir betrachten die Restnetzwerke zu dem Flussnetzwerk aus Beispiel 9.1.*

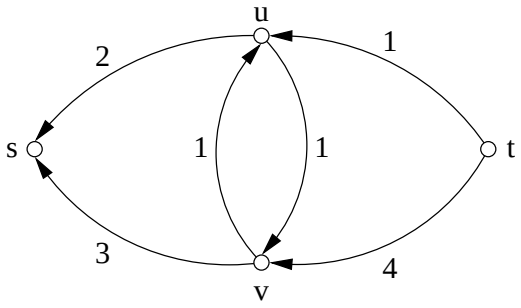
Weg (s, u, v, t):



Weg (s, v, t):



Weg (s, v, u, t)*:*



kein Erweiterungspfad. Kapazitäten sind immer ≥ 0*.*

Beweis. Wir müssen also überlegen, dass $f + g$ ein zulässiger Fluss in G ist.

- Kapazitätsbedingung: Es ist $g(u, v) \leq K_f(u, v) = K(u, v) - f(u, v)$. Also

$$\begin{aligned}(f+g)(u,v) &= f(u,v) + g(u,v) \\ &\leq f(u,v) + K(u,v) - f(u,v) = K(u,v)\end{aligned}$$

- Symmetrie:

$$\begin{aligned}(f+g)(u,v) &= f(u,v) + g(u,v) \\ &= (-f(v,u)) + (-g(v,u)) \\ &= -(f+g)(v,u)\end{aligned}$$

- Kirchhoff: Sei $u \in V \setminus \{s, t\}$, dann

$$\begin{aligned}\sum_{v\in V}(f+g)(u,v) &= \sum_{v\in V}\big(f(u,v) + g(u,v)\big) \\ &= \sum_{v\in V} f(u,v) + \sum_{u\in V} g(u,v) \\ &= 0.\end{aligned}$$

Schließlich ist $|f + g| = \sum_{v\in V}(f + g)(s, v) = |f| + |g|$. □

9.1 Algorithmus nach Ford-Fulkerson

Algorithmus 16: Maximaler Fluss (Ford-Fulkerson)

```
   /* Starte beim Nullfluss */
 1 foreach (u, v) ∈ V × V do
 2 |  f(u, v) = 0;
 3 end
 4 while Es gibt einen Weg s ⟶ t  in G_f do
 5 |  W = ein Erweiterungspfad in G_f;
 6 |  g = Fluss in G_f mit g(u, v) = K_f(W) für alle (u, v) ∈ W;
 7 |  f = f + g;
 8 end
   /* Gib f als maximalen Fluss aus */
 9 return f;
```

Warum liefert dieses Vorgehen einen *maximalen Fluss*? Dass auf diese Weise ein korrekter Fluss erzeugt wird, ist klar. Aber ist dieser auch *maximal*? Dazu müssen wir etwas weiter ausholen. Zunächst noch eine Definition.

Definition 9.4(Schnitt): *Ein Schnitt eines Flussnetzwerkes $G = (V, E)$ ist eine Partition S, T von V, d.h. $V = S \cup T$ mit $s \in S$, $t \in T$ und $S \cap T = \emptyset$.*

- *Kapazität von S, T*

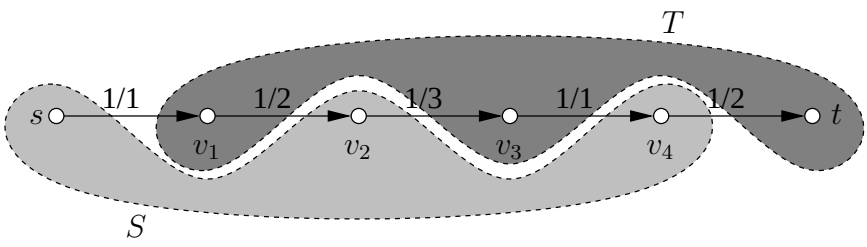
$$K(S,T) = \sum_{u \in S, v \in T} K(u,v) \qquad (K(u,v) \geq 0, u \in S, v \in T)$$

- *Ist f ein Fluss, so ist der Fluss durch den Schnitt*

$$f(S,T) = \sum_{u \in S, v \in T} f(u,v).$$

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

Beispiel 9.3(Schnitt): *Wir betrachten ein Flussnetzwerk:*



$$\begin{aligned} S &= \{s, v_2, v_4\} \\ T &= \{v_1, v_3, t\} \end{aligned}$$

$$\begin{aligned} K(S,T) &= \underbrace{K(s,v_1)}_{=1} + \underbrace{K(v_2,v_3)}_{=3} + \underbrace{K(v_4,t)}_{=2} = 6 \\ F(S,T) &= 1 - 1 + 1 - 1 + 1 = 1 \quad \textit{(Betrag des Flusses)} \end{aligned}$$

S, T in mehreren Stücken $\longrightarrow$ kein minimaler Schnitt.

Tatsächlich gilt sogar für jeden Schnitt S, T, dass $f(S,T) = |f|$ ist.

Beweis. Induktion über $|S|$.

- Induktionsanfang: $|S| = 1$, dann $S = \{s\}$, und $f(S,T) = |f|$ nach Definition.
- Induktionsschritt: Sei $|S| = l + 1$, $v \in S$, $v \neq s$. Wir betrachten den Zustand aus dem dieses S entstanden ist:

  $S' = S \setminus \{v\}, \quad |S'| = l$ (Behauptung gilt nach Induktionsvoraussetzung!)
  $T' = T \cup \{v\} = V \setminus S'$.

  Dann ist der neue Fluss durch den Schnitt

$$f(S,T) = f(S',T') - \underbrace{\sum_{u \in S} f(u,v)}_{=-\sum_{u\in S} f(v,u)} + \sum_{u \in T} f(v,u) = |f| + \underbrace{\sum_{u \in V} f(v,u)}_{=0} = |f|.$$

Also: Für jeden Schnitt $|f| = f(S,T) \leq K(S,T)$. □

Aus dieder Überlegung folgt auch, dass

$$\max\{|f| \mid f \text{ Fluss}\} \leq \min\{K(S,T) \mid S, T \text{ Schnitt}\}.$$

Wir zeigen im folgenden: Es gibt einen Fluss f^*, der diese obere Schranke erreicht, das heißt

$$f^* = \min\{K(S,T) \mid S, T \text{ Schnitt}\}.$$

Satz 9.1(Min-Cut-Max-Flow): *Ist f ein zulässiger Fluss in G, so sind die folgenden Aussagen äquivalent.*

1. *f ist ein 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.

- *(1)* $\rightarrow$ *(2)* gilt, da f maximal. Gälte *(2)* nicht, dann gälte auch *(1)* nicht.
- *(2)* $\rightarrow$ *(3)*: Setze

$$\begin{aligned} S &= \{u \in V \mid \text{Es gibt Weg } (s,u) \text{ in } G_f\} \\ T &= V \setminus S \end{aligned}$$

  Dann $s \in S$ und $t \in T$, da kein Erweiterungspfad nach *(2)*. Also ist S, T ein ordentlicher Schnitt.

  Für $u \in S$, $v \in T$ gilt:

$$0 = K_f(u,v) = K(u,v) - f(u,v)$$

  Also $K(u,v) = f(u,v)$. Dann

$$\begin{aligned} |f| = f(S,T) &= \sum_{u \in S, v \in T} f(u,v) \\ &= \sum_{u \in S, v \in T} K(u,v) \\ &= K(S,T). \end{aligned}$$

- *(3)* $\rightarrow$ *(1)* gilt, da immer $|f| \leq K(S,T)$.

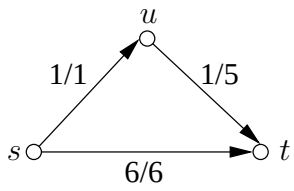
□

Korrektheit von Ford-Fulkerson. Invariante: f_l ist zulässiger Fluss

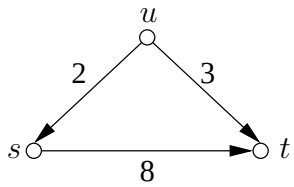
Quintessenz: Am Ende ist f_l maximaler Fluss (Min-Cut-Max-Flow).

Termination: Bei ganzzahligen Kapazitäten terminiert das Verfahren auf jeden Fall. Der Fluss wird in jedem Schritt um eine ganze Zahl ≥ 1 erhöht. Also ist irgendwann nach *endlich* vielen Schritten der maximale Fluss erreicht.

Beispiel 9.4: *Minimaler Schnitt = Maximaler Fluss*



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



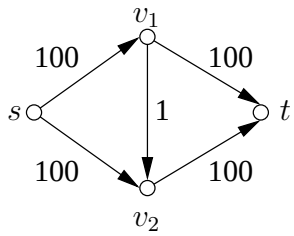
$$\begin{aligned} S = \{s\},\ T = \{u,t\},\ K(S,T) &= 8 \\ S = \{s,u\},\ T = \{t\},\ K(S,T) &= 11 \end{aligned}$$

$$|f| \leq 8.$$

Vergleiche hierzu Beispiel 9.3 auf Seite 128.

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

Beispiel 9.5(lange Laufzeit bei Ford-Fulkerson): *Wir betrachten das folgende Flussnetzwerk:*



Der maximale Fluss in diesen Netzwerk beträgt offensichtlich $|f| = 200$. *Wenn die Erweiterungswege „schlecht“ gewählt werden, kann sich folgendes ergeben:*

$$\begin{aligned}(s, v_1, v_2, t) &\rightarrow f + 1\\ (s, v_2, v_1, t) &\rightarrow f + 1\\ (s, v_1, v_2, t) &\rightarrow f + 1\\ &\vdots\\ (s, v_2, v_1, t) &\rightarrow f + 1\end{aligned}$$

Also 200 *einzelne Erhöhungen. Beachte im Restnetzwerk das „oszillieren“ von* (v_1, v_2).

Man sieht, dass Laufzeiten der Größe $\Omega(|E| \cdot |f^*|)$ durchaus auftreten können. Die Kapazitäten lassen sich so wählen, dass $|f^*|$ exponentiell in der Länge (in Bits) von G und der Kapazitätsfunktion ist.

Abhilfe schafft hier das folgende vorgehen:

> Wähle im Restnetzwerk immer den kürzesten Erweiterungspfad. Das heißt den Erweiterungspfad mit der minimalen Kantenanzahl nehmen.

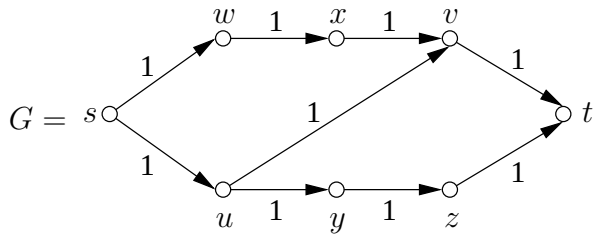
9.2 Algorithmus nach Edmonds-Karp

Algorithmus 17: Maximaler Fluss (Edmonds-Karp)

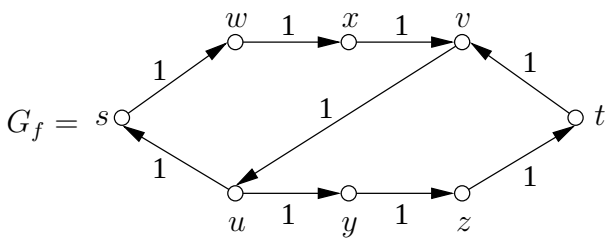
```
1  foreach (u, v) ∈ V × V do
2  |  f(u, v) = 0;
3  end
4  G_f = G;
5  while t in G_f von s aus erreichbar do
6  |  BFS(G_f, s);                                  /* Breitensuche */
7  |  v_k = t;
8  |  v_{k-1} = π[v_k];                          /* π Breitensuchbaum */
9  |  ...
10 |  v_1 = π[v_2];
11 |  s = v_0 = π[v_1];
   |  /* W ist ein kürzester Erweiterungspfad bezogen auf die
   |     Anzahl der Kanten. */
12 |  W = (s = v_0, v_1, ..., v_k = t);
13 |  g(v_i, v_{i+1}) = K_f(W);        /* Flusserhöhung für diesen E-Pfad. */
14 |  f = f + g;
15 |  G_f berechnen;
16 end
   /* Ausgabe: maximaler Fluss f */
17 return f;
```

Beachte: Gemäß Algorithmus ist ein Erweiterungsweg ein Weg ($s \rightsquigarrow t$) mit minimaler Kantenanzahl. Vergleiche das Beispiel 9.5, dort wird gerade kein solcher Weg gewählt.

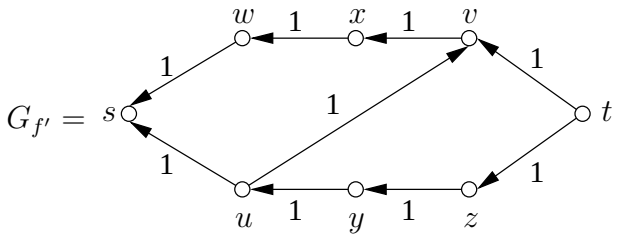
Beispiel 9.6: *Auch bei Edmonds-Karp sind die Umkehrkanten weiterhin notwendig. (Negativer Fluss $\Longleftrightarrow$ Fluss umlenken.)*



Kürzester Erweiterungsweg:
$(s, u, v, t) \rightarrow +1$

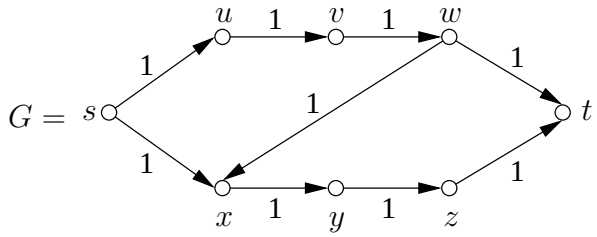


Im Restnetzwerk kürzester Erweiterungsweg:
$(s, w, x, v, u, y, z, t) \rightarrow +1$



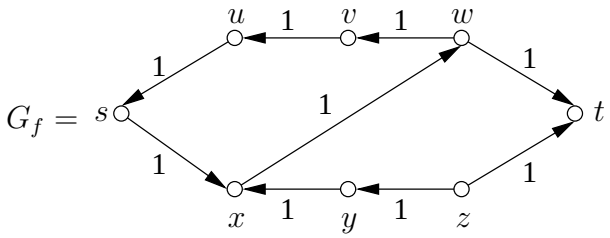
Kein weiterer Erweiterungsweg → maximaler Fluss gefunden.

Beispiel 9.7: *Wenn* nicht *die kürzesten Wege als Erweiterungspfade gewählt werden, können sich die Wege im Restnetzwerk verkürzen.*



$$\begin{aligned} \text{Dist}(s,w) &= 3 \\ \text{Dist}(s,t) &= 4 \end{aligned}$$

Erweiterungsweg
$(s,u,v,w,x,y,z,t) \to +1$



$$\begin{aligned} \text{Dist}(s,w) &= 2 \\ \text{Dist}(s,t) &= 3 \end{aligned}$$

Erweiterungsweg
$(s,x,w,t) \to +1$

Wir zeigen im restlichen Verlauf dieses Abschnitts den folgenden Satz.

Satz 9.2: *Sind* $W_1, W_2, W_3, \ldots W_k$ *die von der* Edmonds-Karp-*Strategie betrachteten Erweiterungswege, dann gilt:*

(a) $\text{Länge}(W_1) \leq \text{Länge}(W_2) \leq \cdots \leq \text{Länge}(W_k) \leq |V|$, *(*$|V|$ = *#Knoten des betrachteten Flussnetzes)*

(b) $\text{Länge}(W_1) \lneq \text{Länge}(W_{|E|+1}) \lneq \text{Länge}(W_{2|E|+1}) \lneq \text{Länge}(W_{3|E|+1}) \lneq \ldots$
Das heißt: Es gibt maximal $|E|$ *viele Erweiterungswege von gleicher Länge.*

Aus diesem Satz folgt dann:

Folgerung 9.1: *Bei Verwendung der* Edmonds-Karp-*Strategie gilt:*

(a) Es werden maximal $|E| \cdot |V|$ Runden der Schleife (5-15) ausgeführt.

(b) Die Laufzeit beträgt $O(|E|^2 \cdot |V|)$, sofern die Kapazitäten in $O(1)$ zu bearbeiten sind.

Wir zeigen zunächst, dass die Folgerung gilt. Danach kommt noch der Beweis des obigen Satzes (der Voraussetzung).

Beweis. (a) Mit der Aussage (b) im Satz gilt für die Wege folgendes:

$$\underbrace{W_1, W_2, \ldots,}_{\text{Länge}\geq 1} \underbrace{W_{|E|+1}, W_{|E|+2}, \ldots,}_{\text{Länge}\geq 2} \underbrace{W_{2|E|+1}, W_{|E|+2}, \cdots}_{\text{Länge}\geq 3} \cdots$$

Also ist sicher beim Index $(|V|-1) \cdot |E| + 1$ die Länge $|V|$ erreicht. Wir haben maximal $|E|$ Wege der Länge $|V|$ und bei $|E| \cdot |V|$ ist ganz sicher der letzte Weg erreicht. Damit hat die Schleife maximal $|E| \cdot |V|$ viele Durchläufe.

(b) Ein Durchlauf durch die Schleife braucht eine Zeit von $O(|V| + |E|)$ für die Breitensuche, Berechnung des Restnetzes etc. Unter der vernünftigen Annahme $|E| \geq |V|$ gilt die Behauptung.

□

Betrachten wir nun einmal einen einzelnen Lauf der Schleife. Zunächst vom allgemeinen *Ford-Fulkerson*. Wir benutzen die folgenden Bezeichnungen:

$$\begin{aligned} f &= \text{Fluss vor dem Lauf der Schleife,} \\ f' &= \text{Fluss nach dem Lauf der Schleife,} \\ G_f, G_{f'} &\quad \text{sind die zugehörigen Restnetzwerke.} \end{aligned}$$

Es gilt: $G_{f'}$ entsteht aus G_f, indem eine (oder mehrehre) Kanten $u \circ\!\!\longrightarrow\!\!\circ\, v$ gelöscht und durch $v \circ\!\!\longrightarrow\!\!\circ\, u$ ersetzt werden. Dabei gilt für $u \circ\!\!\longrightarrow\!\!\circ\, v$:

- Die Kanten $u \circ\!\!\longrightarrow\!\!\circ\, v$ liegen auf dem gewählten Erweiterungsweg.
- Es ist möglich, dass eine Umlenkung $v \circ\!\!\longrightarrow\!\!\circ\, u$ bereits in G_f vorhanden ist. (Das macht nichts.)

Bei *Edmonds-Karp* gilt zusätzlich:

- Die Kanten $u \circ\!\!\longrightarrow\!\!\circ\, v$ wie oben liegen auf einem *kürzesten* Weg $s \circ\!\!\longrightarrow\!\!\circ\, t$.

Wir betrachetn einmal alles ganz allgemein.

Lemma 9.2: *Sei G ein gerichteter Graph mit Knoten s und t. Sei $k = \mathrm{Dist}(s,t)$ die Anzahl der Kanten eines kürzesten Weges von s nach t.*

Ist $u \circ\!\!\longrightarrow\!\!\circ\, v$ eine Kante auf irgendeinem kürzesten Weg $s \circ\!\!\longrightarrow\!\!\circ\, t$, so sei auch die Umkehrkante $v \circ\!\!\longrightarrow\!\!\circ\, u$ in G enthalten.

(a) Ist $u \circ\!\!\longrightarrow\!\!\circ\, v$ Kante eines kürzesten Weges, so kommt nie $v \circ\!\!\longrightarrow\!\!\circ\, u$ auf einem kürzesten Weg vor.

(b) Enthält ein einfacher *kürzester Weg W der Länge k mit l Umkehrkanten, dann gilt für die Weglänge*

$$\text{Länge}(W) \geq k + 2l.$$

Beweis. (a) Angenommen auf den kürestem Wegen kommen irgendwelche dieser Umkehrkanten vor. Wir betrachten einen kürzesten Weg auf dem eine dieser Umkehrkanten am weitesten vorne steht. Dieser sei

$$W = (s = v_0, v_1, \ldots, v_k = t)$$

und (v_i, v_{i+1}) sei die erste Umkehrkante. Außerdem benutzt kein kürzester Weg auf seinen ersten i Schritten eine Umkehrkante.

Angenommen, es gibt einen kürzesten Weg U der Art

$$U = (s = u_0, \ldots, u_j, u_{j+1}, \ldots, u_m = t),$$

mit $u_{j+1} = v_i$ und $u_j = v_{i+1}$. Dann ist $j \geq i$, da sonst die Umkehrkante auf U unter den ersten i Schritten wäre.

Dann ist aber

$$U' = (\underbrace{s = v_0, v_1, \ldots, v_i}_{\text{von } W} = \underbrace{u_{j+1}, \ldots, u_k = t}_{\text{von } U})$$

ein Weg mit

$$\text{Länge}(U') = i + k - (j+1) \leq k - 1,$$

da $j \geq i$. Das kann aber nicht sein, da wir von $\mathrm{Dist}(s,t) = k$ ausgegangen sind!

(b) Zunächst einige Spezialfälle:

- $l = 0$: ✓, da $\text{Dist}(s,t) = k$.
- $l = 1$: Wir betrachten den Weg U.

$$U = (\underbrace{s = u_0, u_1, \ldots, u_i}_{=U_1}, \underbrace{u_{i+1}, \ldots, u_m = t}_{=U_2})$$

Wobei (u_i, u_{i+1}) die Umkehrkante aus einem kürzesten Weg W ist.

$$W = (s = v_0, v_1, \ldots, v_j, v_{j+1}, \ldots, v_k = t), \quad v_j = u_{i+1},\ v_{j+1} = u_i$$

Dann ist

$$\text{Länge}(U_1) = i \geq j + 1,$$

da W ein kürzester Weg ist. Ebenso ist

$$\text{Länge}(U_2) = m - i \geq k - j.$$

Dann ist

$$\text{Länge}(U) \geq (j+1) + 1 + (k-j) = k + 2.$$

- Für ein beliebiges $l \geq 2$ gehen wir analog zum Fall $l = 1$ induktiv vor. Gelte die Behauptung für alle Wege mit $l-1$ Umkehrkanten. Wir betrachten jetzt einen Weg U mit *genau* l Umkehrkanten.

$$U = (\underbrace{s = u_0, u_1, \ldots, u_i}_{U_1}, \underbrace{u_{i+1}, \ldots, u_m = t}_{U_2})$$

Sei (u_i, u_{i+1}) die erste Umlenkung auf U und W ein kürzester Weg.

$$W = (\underbrace{s = v_0, v_1, \ldots, v_j}_{=W_1}, \underbrace{v_{j+1}, \ldots, v_k = t}_{=W_2}), \quad v_j = u_{i+1},\ v_{j+1} = u_i$$

Wir sehen uns jetzt wieder einen Weg U' ohne die Umkehrung an.

$$U' = (\overbrace{\underbrace{s = v_0, \ldots, v_j}_{\substack{\text{ohne Um-}\\\text{kehrung}\\\text{wegen (a)}}}}^{W_1 \text{ von } W} = \overbrace{\underbrace{u_{i+1}, \ldots, u_m = t}_{\substack{l-1 \text{ Um-}\\\text{kehrungen}}}}^{U_2 \text{ von } U})$$

Für die Weglänge ergibt sich nach Induktionsvoraussetzung

$$\text{Länge}(U') \geq k + 2(l-1),$$

da W ein kürzester Weg ist.

$$\text{Länge}(U_1) = i \geq j + 1$$

Damit ist schließlich

$$\begin{aligned}
\text{Länge}(U) &= \text{Länge}(U') - \underbrace{\text{Länge}(W_1)}_{=j} + \underbrace{\text{Länge}(U_1)}_{=i} + 1 \\
&\geq k + 2(l-1) - j + \underbrace{(j+1)}_{\geq i} + 1 \\
&= k + 2l.
\end{aligned}$$

□

Aus diesem Lemma folgt nun direkt der Satz 9.2.

Beweis. In jedem Lauf der Schleife löscht Edmonds-Karp mindestens eine Kante auf einem kürzesten Weg $s \circ\!\!\longrightarrow\!\!\circ\, t$ und trägt die Umkehrkante ein.

(a) Fangen wir mit W_1 mit Länge$(W_1) = k$ an. Zunächst bearbeitet Edmonds-Karp alle Erweiterungswege der Länge k. Solange es solche Wege gibt, wird nach dem Lemma keine neue Unkehrkante betreten. Irgendwann sind die Wege der Länge k erschöpft. Dann haben wir nur noch Kürzeste Wege der Länge $k' \geq k+1$. Wir bearbeiten diese Wege und machen so weiter.

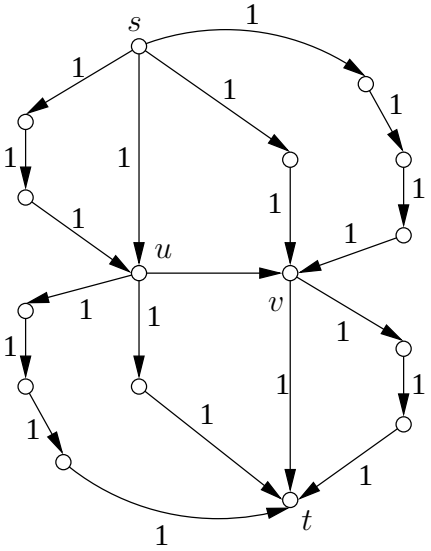
Prinzip: Umkehrkanten von kürzesten Wegen bringen *keine kürzeren* Wege.

(b) Seien einmal $W_1, \ldots, W_l$ die kürzesten Wege in einem Restnetzwerk. In jeder Runde nach Edmonds-Karp verschwindet mindestens eine Kante auf den Kürzesten Wegen.

Nach $|E|$ Runden sind alle Kanten auf den $W_1, \ldots, W_l$ sicherlich fort. Nach dem Lemma enthalten die W_i untereinander keine Umkehrkanten. Also muss in Runde $|E|+1$ eine Umkehrkante vorkommen. Dann vergrößert sich die Länge.

□

Beispiel 9.8(Oszilation bei Edmonds-Karp): *Wir betrachten das folgende Flussnetzwerk.*



1. *Erweiterungsweg* $(s, u, v, t) \to +1$
2. *Erweiterungsweg* $(s, \cdot, v, u, \cdot, t) \to +1$
3. *Erweiterungsweg* $(s, \cdot, \cdot, u, v, \cdot, \cdot, t) \to +1$
4. *Erweiterungsweg* $(s, \cdot, \cdot, \cdot, v, u, \cdot, \cdot, \cdot, t) \to +1$

Die Wege mit der oszillierenden Kante werden immer länger.

10 Dynamische Programmierung

Das Prinzip der *Dynamischen Programmierung* wird häufig bei Fragestellungen auf Worten angewendet.

10.1 Längste gemeinsame Teilfolge

Wir betrachten Worte der Art $w = a_1 a_2 \dots a_m$, wobei a_i ein Zeichen aus einem Alphabet Σ ist. Das Alphabet kann zum Beispiel $\Sigma = \{\text{a, b, } \dots, \text{ z}\}$ sein.

Ein Wort lässt sich als einfaches Array

$$w[1, \dots, m] \quad \text{mit } w[i] = a_i$$

darstellen. So können wir direkt auf die einzelnen Zeichen zugreifen. Zunächst müssen wir noch klären, was wir unter einer Teilfolge verstehen wollen.

Definition 10.1(Teilfolge):

Das Wort v ist eine Teilfolge von $w = a_1 a_2 \dots a_m \iff$ es gibt eine Folge von Indizes $1 \leq i_1 \lneq i_2 \lneq \dots \lneq i_k \leq m$ so dass $v = a_{i_1} a_{i_2} \dots a_{i_k}$.

Oder anders: Eine Teilfolge v von w entsteht dadurch, dass wir aus w einzelne Zeichen(positionen) auswählen und dann in der *gleichen Reihenfolge*, wie sie in w auftreten, als v hinschreiben.

Beispiel 10.1: *Ist $w = abcd$, so sind*

$$a, b, c, d, \quad ab, ac, ad, bc, bd, cd, \quad abc, acd$$

und abcd sowie das leere Wort ε alles Teilworte von w.

Ist $w = a_1 a_2 \dots a_m$, dann entspricht eine Teilfolge von w einer Folge über $\{0, 1\}$ der Länge m. Die Anzahl der Teilworte von w ist $\leq 2^m$.

Beispiel 10.2:

$$\begin{aligned} w = abcd &\Rightarrow \#\,Teilfolgen = 16, \quad aber \\ w = aaaa &\Rightarrow \#\,Teilfolgen = 5. \end{aligned}$$

Beim Problem der längsten gemeinsamen Teilfolge haben wir zwei Worte v und w gegeben und suchen ein Wort u so dass:

- Das Wort u Teilfolge von v *und* w ist.

- Es gibt kein s mit $|s| \gneq |u|$ und s ist Teilfolge von v und w.

Hier ist $|w|$ = #Zeichen von w, die Länge von w. Wir setzen $|\varepsilon| = 0$. Bezeichnung $u = \mathrm{lgT}(v, w)$.

Beispiel 10.3:

1. *Ist etwa* $v = abab$ *und* $w = baba$*, so sind* aba *und* bab *jeweils* $\mathrm{lgT}(v, w)$.
2. *Ist* $w = aba$ *und* $v = acacac$*, so ist* aa *eine* $\mathrm{lgT}(v, w)$.

Also: Die Zeichen der längsten gemeinsamen Teilfolge brauchen in v und w nicht direkt hintereinander stehen.

Man kann $\mathrm{lgt}(v, w)$ auf $\mathrm{lgT}(v', w')$ mit $|v'| + |w'| \lneq |v| + |w|$ zurückführen.

Satz 10.1: *Ist* $v = a_1 \dots a_m$ *und* $w = b_1 \dots b_n$*, so gilt:*

1. *Ist* $a_m = b_n = a$*, so ist jede* $\mathrm{lgT}(v, w)$ *von der Form* ua*. Und* u *ist eine* $\mathrm{lgT}(a_1 \dots a_{m-1}, b_1 \dots b_{m-1})$.
2. *Ist* $a_m \neq b_n$*, so ist jede* $\mathrm{lgT}(v, w)$ *der Art*

$$u_1 = \mathrm{lgT}(a_1 \dots a_{m-1}, w) \quad \textit{oder} \quad u_2 = \mathrm{lgT}(v, b_1 \dots b_{n-1}).$$

Beweis. 1. Ist r eine Teilfolge von v und w ohne a am Ende, so ist ra eine längere gemeinsame Teilfolge. (Beispiel: $v = aa$, $w = aaa \to \mathrm{lgT}(v, w) = \mathrm{lgT}(a, aa)a$.)

Ist ua eine gemeinsame Teilfolge von v und w aber u keine $\mathrm{lgT}(a_1 \dots a_{m-1}, b_1 \dots b_{n-1})$, so ist ua auch keine $\mathrm{lgT}(v, w)$.

2. Für jede gemeinsame Teilfolge u von v und w gilt eine der Möglichkeiten:
   - u ist Teilfolge von $a_1 \dots a_m$ und $b_1 \dots b_{n-1}$
   - u ist Teilfolge von $a_1 \dots a_{m-1}$ und $b_1 \dots b_n$
   - u ist Teilfolge von $a_1 \dots a_{m-1}$ und $b_1 \dots b_{n-1}$

   (Beispiel: $v = abba, w = baab \quad \Rightarrow \quad ab$ ist $\mathrm{lgT}(abb, baab)$, ba ist $\mathrm{lgT}(abba, baa)$.)

□

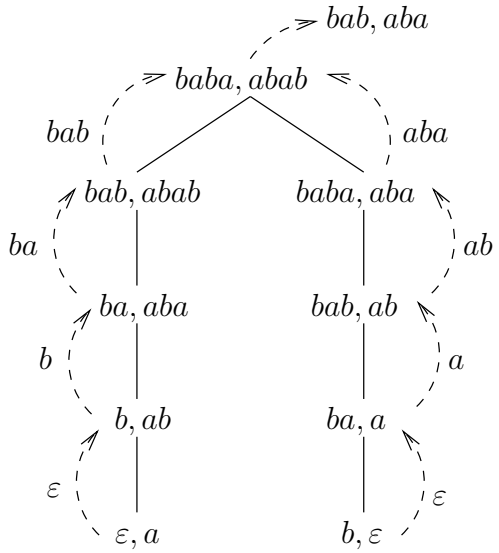
Hier wird das Prinzip der *Reduktion auf optimale Lösungen von Teilproblemen* genutzt. Wir betrachten zunächst den rekursiven Algorithmus.

Algorithmus 18: $\mathrm{lgT}(v, w)$

Input : $v = a_1 \dots a_m,\ w = b_1 \dots b_n$
Output : längste gemeinsame Teilfolge von v und w

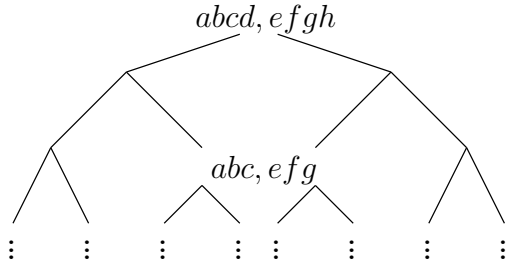
```
1  if m == 0 oder n == 0 then
2  |  return ε
3  end
4  if a_m == b_n then
5  |  d = lgT(a_1 ... a_{m-1}, b_1 ... b_{n-1});
6  |  return „d verlängert um a_m“;
7  else
8  |  d = lgT(a_1 ... a_{m-1}, b_1 ... b_n);
9  |  e = lgT(a_1 ... a_m, b_1 ... b_{n-1});
10 |  return „Längeres von d, e“ ;        /* bei |d| = |e| beliebig */
11 end
```

Betrachten wir nun den Aufrufbaum für $v = baba$ und $w = abab$.

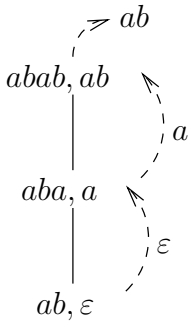


Die längste gemeinsame Teilfolge ist also *aba* uder *bab*. Die Struktur des Baumes ist *nicht* statisch. Sie hängt von der Eingabe ab.

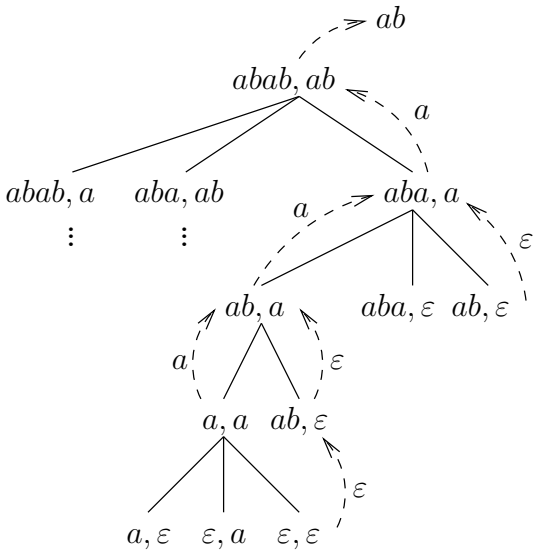
Die Größe des Baumes ist $\geq 2^n$ bei $|v| = |w| = n$.



Beispiel 10.4:



So finden wir den Teil ab $\underline{ab}$ *und* $\underline{ab}$ *als längste gemeinsame Teilfolge. Wie bekommen wir alle Möglichkeiten?*



⇒ *Immer alle Wege gehen. Bei gleicher maximalen Länge beide merken.*

Sei jetzt wieder $|v| = m$, $|w| = n$. Wieviele *verschiedene* haben wir?

$$\text{Aufruf} \Longleftrightarrow \text{zwei Anfangsstücke von } v \text{ und } w$$

Das Wort v hat $m+1$ Anfangsstücke (inklusive v und ε), w hat $n+1$ Anfangsstücke. Also #Anfangsstücke $\leq (m+1) \cdot (n+1)$.

Wir merken uns die Ergebnisse der verschiedenen Aufrufe in einer Tabelle $T[k, h]$, $0 \leq k \leq m$, $0 \leq h \leq n$.

$$T[k, h] = \text{Länge einer lgT}(a_1 \ldots a_k, b_1 \ldots b_h)$$

Dann ist

$$\begin{aligned} T[0,h] &= 0 \quad \text{für } 0 \le h \le m \\ T[k,0] &= 0 \quad \text{für } 0 \le k \le n \end{aligned}$$

und weiter für $h, k \gneq 0$

$$T[k,h] = \begin{cases} T[k-1,h-1]+1 & \text{falls } a_k = b_h \\ \max\{T[k,h-1], T[k-1,h]\} & \text{falls } a_k \neq b_h. \end{cases}$$

Das Mitführen von Pointern zu den Maxima erlaubt die Ermittlung aller $\mathrm{lgT}(v,w)$ mit ihren Positionen.

Laufzeit: $\Theta(m \cdot n)$, da pro Eintrag $O(1)$ anfällt.

Beispiel 10.5: *$v = abc$, $w = abcd$* $\quad T[k,h]$ *mit* $0 \le k \le 3$, $0 \le h \le 4$

$k \backslash h$ T		0	a 1	b 2	c 3	d 4
	0	0	0	0	0	0
a	1	0	1	1	1	1
b	2	0	1	2	2	2
c	3	0	1	2	3	**3**
			a	b	c	

$k \backslash h$ B		0	a 1	b 2	c 3	d 4
	0	•	←	←	←	←
a	1	↑	↖	←	←	←
b	2	↑	↑	↖	←	←
c	3	↑	↑	↑	↖	←
			a	b	c	

$$\nwarrow \Leftrightarrow +1 \qquad \uparrow, \leftarrow \Leftrightarrow +0$$

Die Tabelle füllen wir spaltenweise von links nach rechts *oder zeilenweise von* oben nach unten *aus. Zusätzlich haben wir noch eine Tabelle $B[k,h]$ gleicher Dimension für die $\nwarrow, \uparrow, \leftarrow$.*

10.2 Optimaler statischer binärer Suchbaum

Wir betrachten im folgenden das Problem des *optimalen statischen binären Suchbaumes*. Uns interessieren dabei nur die Suchoperationen, Einfügen und Löschen von Elementen findet nicht statt.

Gegeben sind n verschiedene (Schlüssel-)Werte, $a_1 \lneq a_2 \lneq \cdots \lneq a_n$, mit den zugehörigen zugriffswahrscheinlichkeiten. Auf den Wert a_1 wird mit Wahrscheinlichkeit p_1, auf a_2 mit p_2, usw., zugegriffen. Dabei gilt

$$\sum_{i=1}^{n} p_i = 1.$$

Gesucht ist ein *binärer Suchbaum* T, der die Werte $a_1, \ldots, a_n$ enthält. Die Kosten für das Suchen in T mit den Häufigkeiten p_i sollen *minimal* sein.

Was sind die Kosten von T?

Definition 10.2: *(a)* $\text{Tiefe}_T(a_i)$ *bezeichnet die Tiefe von* a_i *in* T*, also die Anzahl der Kanten von der Wurzel von* T *bis zum Knoten* a_i*. Bei der Suche nach* a_i *sind demnach* $\text{Tiefe}_T(a_i) + 1$ *viele Vergleichsoperationen nötig. Wir summieren die Kosten für die einzelnen* a_i *gewichtet mit der Wahrscheinlichkeit, dass dieser Wert gesucht wird.*

$$K(T) := \sum_{i=1}^{n} p_i \cdot \big(\text{Tiefe}_T(a_i) + 1\big)$$

(b) T *ist optimal genau dann, wenn gilt:*

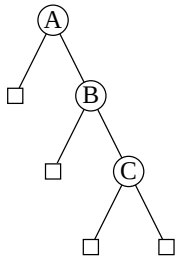
$$K(T) = \min\{K(S) \mid S \text{ ist ein binärer Suchbaum für } a_1, \ldots, a_n\}$$

Beachte: Ein optimaler Suchbaum existiert immer, da es nur endlich viele Suchbäume zu $a_1, \ldots, a_n$ gibt.

Beispiel 10.6:

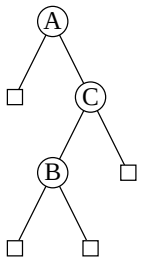
$$a_1 = A, \quad a_2 = B, \quad a_3 = C, \qquad p_1 = \frac{4}{10}, \quad p_2 = \frac{3}{10}, \quad p_3 = \frac{3}{10}$$

1.



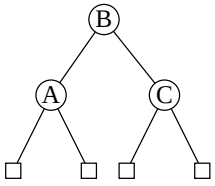
$$K = \frac{\overbrace{1 \cdot 4}^{A} + \overbrace{2 \cdot 3}^{B} + \overbrace{3 \cdot 3}^{C}}{10} = \frac{19}{10}$$

2.



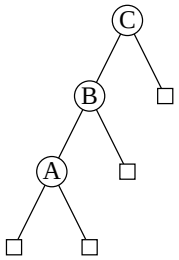
$$K = \frac{\overbrace{1 \cdot 4}^{A} + \overbrace{3 \cdot 3}^{B} + \overbrace{2 \cdot 3}^{C}}{10} = \frac{19}{10}$$

3.



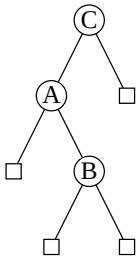
$$K = \frac{\overbrace{2 \cdot 4}^{A} + \overbrace{1 \cdot 3}^{B} + \overbrace{2 \cdot 3}^{C}}{10} = \frac{\mathbf{17}}{\mathbf{10}}$$

4.



$$K = \frac{\overbrace{3 \cdot 4}^{A} + \overbrace{2 \cdot 3}^{B} + \overbrace{1 \cdot 3}^{C}}{10} = \frac{21}{10}$$

5.

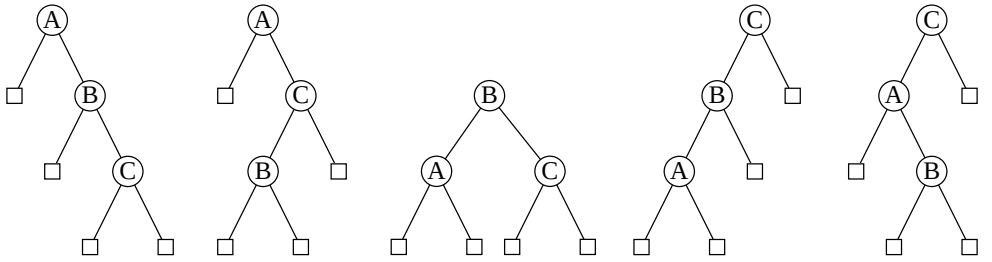


$$K = \frac{\overbrace{2 \cdot 4}^{A} + \overbrace{3 \cdot 3}^{B} + \overbrace{1 \cdot 3}^{C}}{10} = \frac{20}{10}$$

Beobachtung: Es reicht nicht aus, einfach das häufigste Element an die Wurzel zu setzen.

Beispiel 10.7: *Es müssen auch nicht zwangsläufig ausgeglichene Bäume entstehen. Wie man hier sieht.*

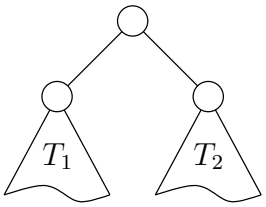
$$a_1 = A, \quad a_2 = B, \quad a_3 = C, \qquad p_1 = \frac{5}{9}, \quad p_2 = \frac{2}{9}, \quad p_3 = \frac{2}{9}$$



Kosten: $\frac{5+4+6}{9} = \mathbf{\frac{15}{9}}$ $\frac{5+6+4}{9} = \mathbf{\frac{15}{9}}$ $\frac{10+2+4}{9} = \frac{16}{9}$ $\frac{15+4+2}{9} = \frac{21}{9}$ $\frac{10+6+2}{9} = \frac{18}{9}$

Unser Ziel ist es, den optimalen Suchbaum in einer Laufzeit von $O(n^3)$ zu bestimmen.

Haben wir einen beliebigen binären Suchbaum für $a_1 \lneq a_2 \lneq \cdots \lneq a_n$ gegeben, so lassen sich seine Kosten bei Häufigkeiten p_i folgendermaßen ermitteln:



Wurzel betreten: $1 = \sum_{a_j \in T} p_j$

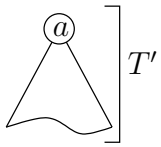
Linker Sohn der Wurzel: $\sum_{a_i \in T_1} p_i$

Rechter Sohn der Wurzel: $\sum_{a_k \in T_2} p_k$

...

Allgemein sagen wir die Kosten, die ein Teilbaum von T mit der Wurzel a zu den Gesamtkosten beiträgt, ist $K_T(a)$ mit

$$K_T(a) = \sum_{a_j \in T'} p_j$$

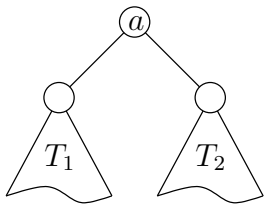
T' = Teilbaum von T mit Wurzel a. $\quad a_i \in$ 

Folgerung 10.1: *Für den binären Suchbaum S auf $a_1, \ldots, a_n$ gilt:*

$$K(S) = \sum_{i=1}^{n} K_S(a_i)$$

Beweis. Induktion über n. Für $n = 1$ ist $p_1 = 1$ und damit $K(T) = 1$ ✓.

Induktionsschluß: Sei $n \geq 1$. Wir betracheten $a_1, \ldots, a_{n+1}$ mit beliebigen p_i. Es ist

$S =$ 

$a = a_j$ für ein a_j

Dann ist

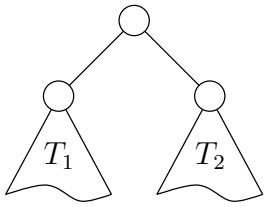
$$\begin{aligned} K(S) &= \sum_{i=1}^{n+1} p_i \cdot \big(\text{Tiefe}_S(a_i) + 1\big) \quad \text{nach Definition} \\ &= \underbrace{\sum_{i=1}^{n+1} p_i}_{=1} + \sum_{i=1}^{n+1} p_i \cdot \text{Tiefe}_S(a_i) \end{aligned}$$

$\text{Tiefe}_S(a_j) = 0$ Tiefe der Wurzel

$$\begin{aligned}
K(S) &= 1 + \sum_{a_i \in T_1} p_i \cdot \text{Tiefe}_S(a_i) + \sum_{a_k \in T_2} p_k \cdot \text{Tiefe}_S(a_k) \\
&= 1 + \sum_{a_i \in T_1} p_i \cdot \big(1 + \text{Tiefe}_{T_1}(a_i)\big) + \sum_{a_k \in T_2} p_k \cdot \big(1 + \text{Tiefe}_{T_2}(a_k)\big) \\
&\quad \text{mit der Induktionsvoraussetzung für } T_1 \text{ und } T_2 \\
&= 1 + \sum_{a_i \in T_1} K_{T_1}(a_i) + \sum_{a_k \in T_2} K_{T_2}(a_k) \\
&= K_S(a_j) + \sum_{a_i \in T_1} K_S(a_i) + \sum_{a_k \in T_2} K_S(a_k) \\
&= \sum_{i=1}^{n+1} K_S(a_i)
\end{aligned}$$

□

Folgerung 10.2: *Ist für eine Menge $b_1, \ldots, b_l$ mit p_i, $\sum p_i \leq 1$*

$S =$ 

ein Baum, so dass $\sum K_S(b_i)$ minimal ist, so ist für T_1 und T_2

$$\sum_{b_i \in T_1} K_{T_1}(b_i), \quad \sum_{b_i \in T_2} K_{T_2}(b_i)$$

minimal.

Beweis.

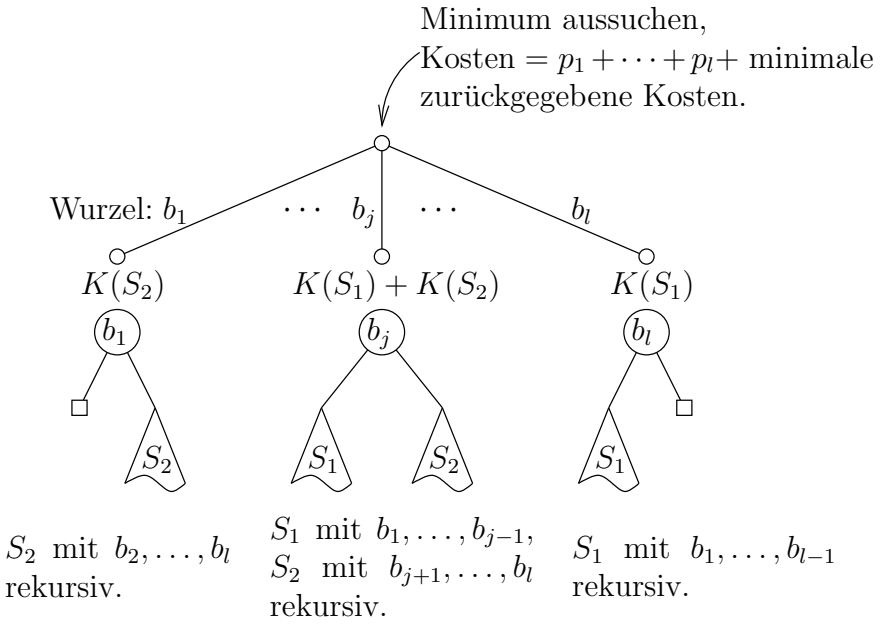
$$\sum_{i=1}^{l} K_S(b_i) = \sum_{i=1}^{l} p_i + \sum_{b_i \in T_1} K_{T_1}(b_i) + \sum_{b_i \in T_2} K_{T_2}(b_i)$$

□

Prinzip: Teile optimaler Lösungen sind optimal. Diese Eigenschaft wird beim dynamischen Programmieren immer genutzt.

Bei der Eingabe von $b_1, \ldots, b_l$, $b_1 \lneq \cdots \lneq b_l$ mit Wahrscheinlichkeiten p_i, $\sum p_i \leq 1$ ermitteln wir rekursiv einen Baum S mit

$$\sum_{i=1}^{l} K_S(b_i) \quad \text{minimal.}$$



Laufzeit: $\gg \Omega(2^l)$, 2^l Aufrufe werden alleine für ganz links und ganz rechts benötigt. Die Struktur des Aufrufbaumes ist statisch. Wieviele verschiedene Aufrufe sind darin enthalten?

$$
\begin{aligned}
\#\text{verschiedene Aufrufe} &\leq \#\text{verschiedene Aufrufe der Art} \\
&\quad b_k \lneq b_{k+1} \lneq \cdots \lneq b_{h-1} \lneq b_h, \text{ wobei } 1 \leq k \leq h \leq l \\
&= \big|\{(k,h) \mid 1 \leq k \leq h \leq l\}\big| \\
&= \underbrace{l}_{k=1} + \underbrace{(l-1)}_{k=2} + \underbrace{(l-2)}_{k=3} + \cdots + \underbrace{1}_{k=l} \quad \#\text{Möglichkeiten für } h \\
&= \frac{l(l+1)}{2} = O(l^2).
\end{aligned}
$$

Bei $a_1, \ldots, a_n$ mit p_i, $\sum p_i = 1$ Tabelle $T[k,h]$, $1 \leq k \leq h \leq n$ mit der Bedeutung

$$T[k,h] \Leftrightarrow \text{ optimale Kosten für einen Baum } S_{k,h} \text{ mit den Werten } a_k, \ldots, a_h.$$

Beispiel 10.8: *Für $n = 4$ ergeben sich die Tabelleneinträge ergeben wie folgt:*

$k \backslash h$	1	2	3	4
1				
2				
3				
4				

Für $k > h$ keine Einträge.

$$
\begin{aligned}
T[1,1] &= p_1, \qquad T[2,2] = p_2, \qquad T[3,3] = p_3, \qquad T[4,4] = p_4 \\
\\
T[1,2] &= \min\Big\{ \underbrace{(p_1 + p_2 + T[2,2])}_{a_1 \text{ als Wurzel}},\ \underbrace{(p_2 + p_1 + T[1,1])}_{a_2 \text{ als Wurzel}} \Big\} \\
T[2,3] &= \min\big\{(p_2 + p_3 + T[3,3]),\ (p_3 + p_2 + T[2,2])\big\} \\
T[3,4] &= \min\big\{(p_3 + p_4 + T[4,4]),\ (p_4 + p_3 + T[3,3])\big\} \\
\\
T[1,3] &= p_1 + p_2 + p_3 + \min\{T[2,3],\ (T[1,1] + T[3,3]),\ T[1,2]\} \\
T[2,4] &= p_1 + p_2 + p_3 + \min\{T[3,4],\ (T[2,2] + T[4,4]),\ T[2,3]\} \\
\\
T[1,4] &= p_1 + p_2 + p_3 + p_4 + \\
&\qquad + \min\{T[2,4],\ (T[1,1] + T[3,4]),\ (T[1,2] + T[4,4]),\ T[1,3]\}
\end{aligned}
$$

Allgemein: Wir haben $O(n^2)$ Einträge in T. Pro Eintrag fätllt eine Zeit von $O(n)$ an. (Das lässt sich induktiv über $h - k$ zeigen.) Also lässt sich das Problem in der Zeit $O(n^3)$ lösen.

Für alle Tabellenenträge gilt:

$$
\begin{aligned}
T[k,h] &= p_k + \cdots + p_h + \min\Big\{ \big\{T[k+1,h]\big\} \cup \big\{T[k,h-1]\big\} \cup \\
&\qquad \cup \big\{T[k,i-1] + T[i+1,h] \mid k+1 \le i \le h-1\big\} \Big\}
\end{aligned}
$$

Beispiel 10.9:

$$
a_1 = A, \quad a_2 = B, \quad a_3 = C, \qquad p_1 = \frac{4}{10}, \quad p_2 = \frac{3}{10}, \quad p_3 = \frac{3}{10}
$$

$K(S)$	A	B	C
A	$\frac{4}{10}$	$\frac{10}{10}$	$\mathbf{\frac{17}{10}}$
B	–	$\frac{3}{10}$	$\frac{9}{10}$
C	–	–	$\frac{3}{10}$

Wurzel	A	B	C
A	A	A	**B**
B	–	B	B *oder* C
C	–	–	C

$$
T[A,C] = \frac{10}{10} + \min\left\{ \underbrace{T[B,C]}_{=\frac{9}{10}},\ \underbrace{T[A,A] + T[C,C]}_{=\frac{7}{10}},\ \underbrace{T[A,B]}_{=\frac{10}{10}} \right\}
$$

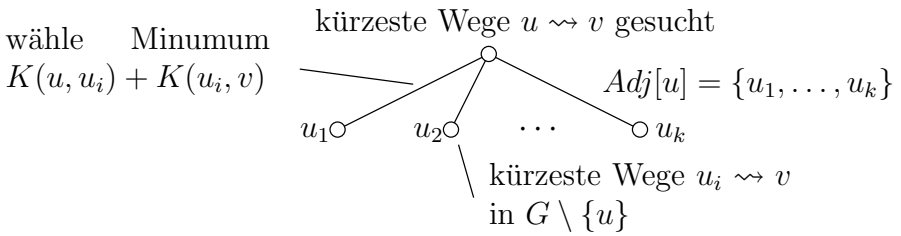
10.3 Kürzeste Wege mit negativen Kantengewichten

Ab jetzt betrachten wir wieder eine beliebige Kostenfunktion $K : E \to \mathbb{R}$.

Der Greedy-Ansatz wie bei Dijkstra funktioniert nicht mehr. Kürzeste Wege $u \circ\!\!\longrightarrow\!\!\circ\, v$ findet man durch Probieren.

Die Laufzeit ist dann $> (n-2)! \geq 2^{\Omega(n \cdot \log n)} \gg 2^n$ (!)

Es werden im Allgemeinen viele Permutationen generiert, die gar keinen Weg ergeben. Das kann mit Backtracking vermieden werden:



Dies ist korrekt, da Teilwege kürzester Wege wieder kürzeste Wege sind und wir mit kürzesten Wegen immer nur einfache Wege meinen.

Algorithmus 19: Kürzeste Wege mit Backtracking

Input : $G = (V, E)$ gerichteter Graph, $K : E \to \mathbb{R}$ beliebige Kantengewichte, u Startknoten, v Zielknoten

```
1 KW(G, u, v);
```

Das Ganze ist leicht durch Rekursion umzusetzen:

Prozedur KW(W, u, v)

Input : W noch zu betrachtende Knotenmenge, u aktueller Knoten, v Zielknoten
Output : Länge eines kürzesten Weges von u nach v

```
1  if u == v then
2  |  return 0;
3  end
4  l = ∞;
5  foreach w ∈ Adj[u] ∩ W do
6  |  l' = KW(W \ {u}, w, v);          /* Ein kürzester Weg (w, ..., v) */
7  |  l' = K(u, w) + l';              /* Die länge des Weges (u, w, ..., v) */
8  |  if l' < l then
9  |  |  l = l';                      /* Neuen kürzesten Weg merken. */
10 |  end
11 end
12 return l;                          /* Ausgabe ∞, wenn Adj[u] ∩ W = ∅ */
```

Die Korrektheit läßt sich mittels Induktion über $|W|$ zeigen.

Etwas einfacher ist es, alle einfachen Wege $a \circ\!\!\longrightarrow\!\!\circ\, b$ systematisch zu erzeugen und den Längenvergleich nur am Ende durchzuführen.

Idee: Wir speichern den Weg vom Startknoten u bis zum aktuellen Knoten. Wenn ein neuer kürzester Weg gefunden wird, dann abspeichern.

Prozedur KW(W, u, v)

Data : L, M als globale Arrays. L ist als Stack implementiert enthält den aktuell betrachteten Weg. In M steht der aktuell kürzeste Weg. Vorher initialisiert mit $L = (a)$, a Startknoten und $M = \emptyset$.

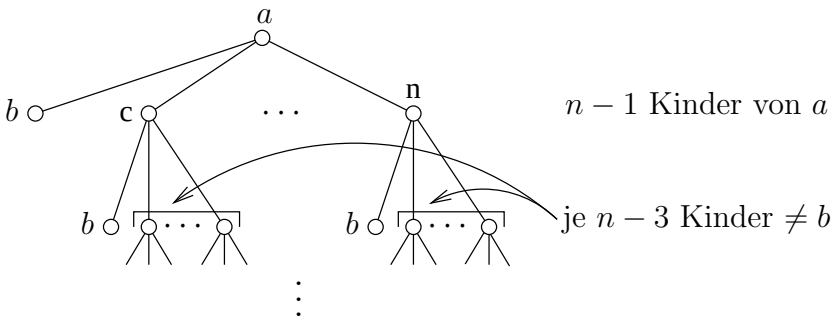
```
1  if u == v then
   |  /* K(L), K(M) Kosten des Weges in L bzw. M */
2  |  if K(L) < K(M) then
3  |  |  M = L;
4  |  end
5  else
6  |  foreach w ∈ Adj[u] ∩ W do
7  |  |  push(L, w);
8  |  |  KW(W \ {u}, w, v);
9  |  |  pop(L);
10 |  end
11 end
```

Korrektheit mit der Aussage:

> Beim Aufruf von KW(W, w, v) ist $L = (w \dots a)$. Am Ende des Aufrufs KW(W, w, v) enthält M einen kürzesten Weg der Art $M = (b \dots L)$. (L ist das L von oben)

Das Ganze wieder induktiv über $|W|$.

Laufzeit der rekursiven Verfahren? Enthält der Graph alle $n(n-1)$ Kanten, so sieht der Aufrufbaum folgendermaßen aus:



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

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

$$\begin{aligned} W \subseteq V &\leq 2^n \text{ Möglichkeiten} \\ c, d &\leq n \text{ Möglichkeiten, da Endpunkt immer gleich} \end{aligned}$$

Also $n \cdot 2^n = 2^{\log n} \cdot 2^n = 2^{n+\log n} = 2^{O(n)}$ viele verschiedene Aufrufe.

Bei $2^{\Omega(n \log n)}$ Aufrufen wie oben sind viele doppelt. Es ist ja

$$\frac{2^{O(n)}}{2^{\Omega(n \log n)}} = \frac{1}{2^{\Omega(n \log n) - O(n)}} \geq \frac{1}{2^{\varepsilon n \cdot \log n - c \cdot n}} \geq \frac{1}{2^{\Omega(n \log n)}}.$$

Die Anzahl verschiedener Aufrufe ist nur ein verschwindend kleiner Anteil an allen Aufrufen.

Vermeiden der doppelten Berechnung durch Merken (Tabellieren). Dazu benutzen wir das zweidimensionale Array $T \underbrace{[1 \ldots 2^n]}_{\in \{0,1\}^n}[1 \ldots n]$ mit der Interpretation:

> Für $W \subseteq V$ mit $b, v \in W$ ist $T[W, v]$ = Länge eines kürzesten Weges $v \circ\!\!\longrightarrow\!\!\circ\, b$ 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. $|W| = 3$, dann

$$\begin{aligned} T[W, b] &= 0 \\ T[W, v] &= \min\{K(v, u) + T[W', u] \mid W' = W \setminus \{v\}, u \in W\} \\ T[W, v] &= \infty, \text{ wenn keine Kante } v \circ\!\!\longrightarrow\!\!\circ\, u \quad \text{mit } u \in W \end{aligned}$$

Das Mitführen des Weges durch $\pi[W, v] = u, \quad u$ vom Minimum ermöglicht eine leichte Ermittlung der Wege.

Der Eintrag $T[W, v]$ wird so gesetzt:

Prozedur Setze(W, v, b)

```
1  if v == b then
2  |  T[W, v] = 0;
3  |  π[W, v] = b;
4  else
5  |  W' = W \ {v};                          /* Hier ist |W| ≥ 2 */
6  |  T[W, v] = ∞;
7  |  foreach u ∈ Adj[v] ∩ W' do
8  |  |  l = K(u, v) + T[W', v];
9  |  |  if l < T[W, v] then
10 |  |  |  T[W, v] = l;
11 |  |  |  π[W, v] = u;
12 |  |  end
13 |  end
14 end
```

Dann insgesamt:

Prozedur KW(V, a, b)

```
   /* Gehe alle Teilmengen von V, die den Zielknoten b enthalten,
      der größe nach durch. */
1  for i = 1 to n do
2  |  foreach W ⊆ V mit b ∈ W, |W| = i do
3  |  |  foreach v ∈ W do
4  |  |  |  Setze(W, v, b)
5  |  |  end
6  |  end
7  end
```

Dann enthält $T[V, a]$ das Ergebnis. Der Weg ist

$$\begin{aligned} a_0 &= a, \\ a_1 &= \pi[V, a], \\ a_2 &= \pi[V \setminus \{a\}, a_1], \\ &\dots, \\ a_i &= \pi[V \setminus \{a_0, \dots, a_{i-2}\}, a_{i-1}], \\ &\dots \end{aligned}$$

Die Korrektheit ergibt sich mit der folgenden Invariante:

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 ⇒ Tabellieren der Aufrufe heißt:

Dynamisches Programmieren (Auffüllen einer Tabelle, 1950er)

11 Divide-and-Conquer und Rekursionsgleichungen

Der Ansatz des *Divide-and-Conquer* lässt sich auf Probleme anwenden, die sich auf die folgende Weise lösen lassen:

1. Das Problem in Teilprobleme aufteilen. (*divide*)
2. Die einzelnen Teilprobleme (in der Regel rekursiv) unabhängig voneinander lösen.
3. Die erhaltenen Lösungen der Teilprobleme zu einer Gesamtlösung zusammensetzen. (*conquer*)

Typische Beispiele sind *Binäre Suche*, *Meregesort* und *Quicksort*.

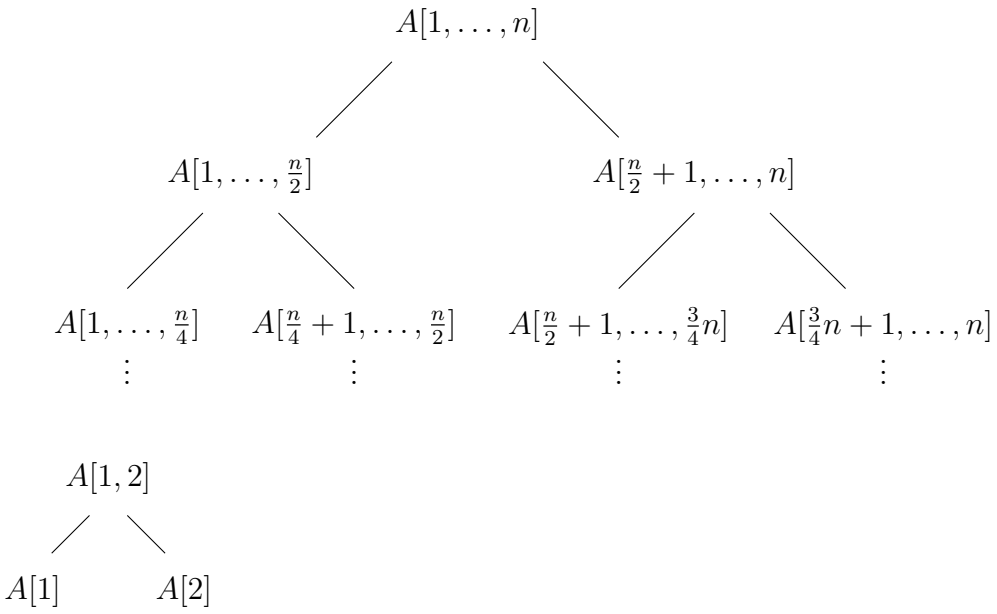
11.1 Mergesort

Algorithmus 20: Mergesort($A[1, \ldots, n]$)

Input : Array $A[1..n]$ mit n Elementen
Output : Das Array A sortiert.

```
   /* Ein- bzw. nullelementige Folge ist sortiert. */
```
1 **if** $|A| == 1$ *oder* $|A| == 0$ **then**
2 | **return** A;
3 **end**
```
   /* Zwei Teilfelder der Größe n/2 bilden und diese sortieren. */
```
4 B_1 =Mergesort($A[1, \ldots, \lfloor \frac{n}{2} \rfloor]$);
5 B_2 =Mergesort($A[\lfloor \frac{n}{2} \rfloor + 1, \ldots, n]$);
```
   /* Neues, sortiertes, Feld aus B1 und B2 bilden. Das ist
      einfach, da die beiden Teilfelder bereits sortiert sind. */
```
6 **return** *„Mischung“* von B_1 und B_2;

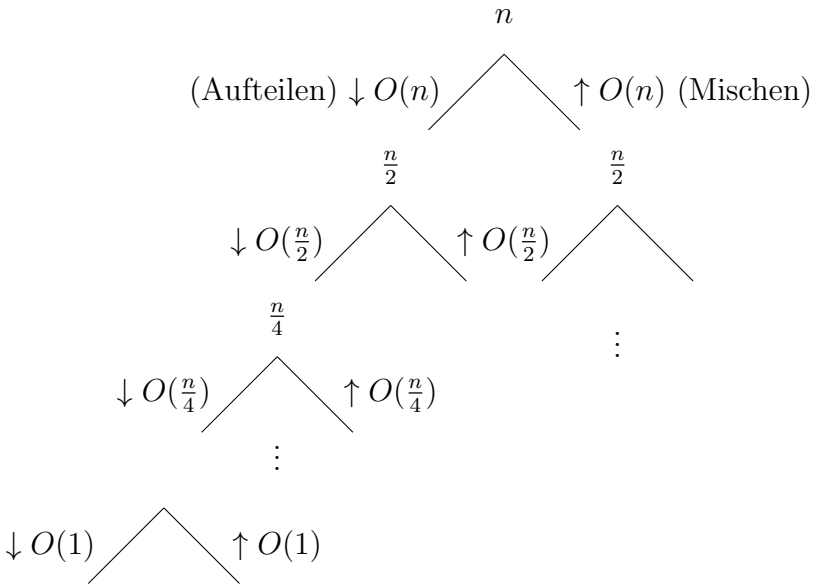
Betrachten wir den Aufrufbaum von *Mergesort* der Einfachheit halber für $n = 2^k$, einer Zweierpotenz:



Die Tiefe des Aufrufbaumes ist hier genau $\log_2 n$. Betrachten wir die Laufzeit:

- Blätter: $O(n)$
- Aufteilen: bei m Elementen $O(m)$
- Zusammensetzen von zweimal $\frac{m}{2}$ Elementen: $O(m)$

Damit erhalten wir insgesamt den folgenden Aufwand:



Für die Blätter: $n \cdot O(1) = O(n)$

Für das Aufteilen: $c \cdot n + c \cdot \frac{n}{2} + c \cdot \frac{n}{4} + \ldots + c \cdot 1 + c \cdot \frac{n}{2} + \ldots$

Wo soll das enden? Wir addieren einmal in einer anderen Reihenfolge:

$$
\begin{array}{ll}
\text{Stufe:} & \\
1 & c \cdot n + \\
2 & \quad + c \cdot \frac{n}{2} + c \cdot \frac{n}{2} + \\
3 & \quad\quad + c \cdot (\frac{n}{4} + \frac{n}{4} + \frac{n}{4} + \frac{n}{4}) + \\
4 & \quad\quad\quad + c \cdot (\underbrace{\frac{n}{8} + \cdots + \frac{n}{8}}_{8\times}) + \\
\vdots & \\
\log_2 n & \quad\quad\quad + c \cdot (\underbrace{1 + \cdots + 1}_{n\times}) \\
& = \log_2 n \cdot c \cdot n = O(n \cdot \log n)
\end{array}
$$

Ebenso für das Mischen: $O(n \cdot \log n)$

Insgesamt haben wir dann $O(n \cdot \log n) + O(n) = O(n \cdot \log n)$.

- *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 lässt sich auch einfach bottom-up ohne Rekursion lösen, da der Aufrufbaum unabhängig von der Eingabe *immer* die gleiche Struktur hat.

  1. $A[1,2], A[3,4], \ldots, A[n-1,n]$ Sortieren, dann
  2. $A[1,\ldots,4], A[5,\ldots,8], \ldots$ Sortieren
  3. $\ldots$

  Auch $O(n \cdot \log n)$.

- Heapsort sortiert ebenfalls in $O(n \cdot \log n)$ und
- Bubblesort, Insertion Sort, Selection Sort in $O(n^2)$. Das sind aber keine divide-and-conquer Algorithmen.

11.2 Quicksort

Eingabe: Array $A[1,\ldots,n]$ of „geordneter Datentyp“

Algorithmus 21: Quicksort($A[1,\ldots n]$)

Input : Array $A[1..n]$ mit n Elementen
Output : Das Array A sortiert.

```
1  if |A| == 1 oder |A| == 0 then
2  |  return A;
3  end
4  a = A[1];                 /* Erstes Element aus A als Pivotelement. */
5  B1 = [ ]; B2 = [ ];            /* Zwei leere Felder der Größe n − 1. */
6  for j = 2 to n do
7  |  if A[j] ≤ a then
8  |  |  A[j] als nächstes Element zu B1;
9  |  else /* A[j] > a */
10 |  |  A[j] als nächstes Element zu B2;
11 |  end
12 end
   /* Elemente ≤ a und > a getrennt sortieren. */
13 B1 = Quicksort(B1);
14 B2 = Quicksort(B2);
   /* Die sortierten Teilfelder zum Gesamtfeld zusammensetzen. */
15 return B1.a.B2;
```

Beachte: Die Prozedur Partition erlaubt es, mit dem Array A alleine auszukommen. Das heisst es muß nichts kopiert werden und an *Quicksort* werden nur die *Grenzen* der Teilarrays übergeben. Dann sieht das so aus:

Algorithmus 22: Quicksort(A, l, r) mit Partition

Input : Array A per Referenz; l, r linke und rechte Grenze.
Output : $A[l,\ldots,r]$ sortiert.

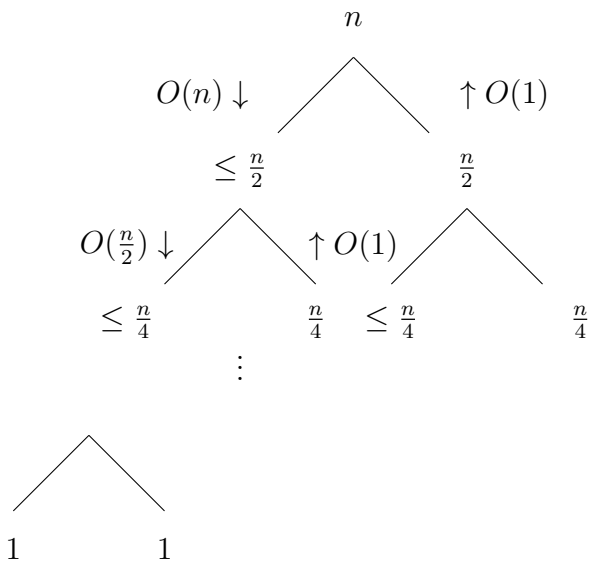
```
1  if l < r then
2  |  j = Partition(A, l, r);
3  |  Quicksort(A, l, j − 1);
4  |  Quicksort(A, j + 1, r);
5  end
```

Prozedur Partition($A[1,\dots n], l, r$)

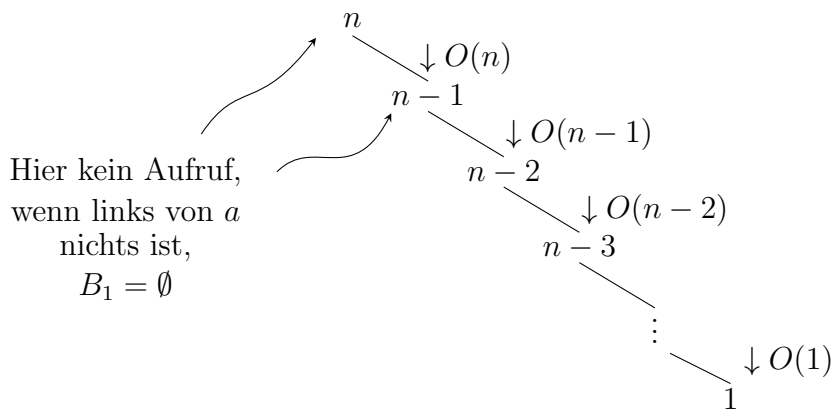
Input : Array A als *Referenz*, es wird direkt auf A gearbeitet. Indizes l, r linke und rechte Grenze des Bereichs, der partitioniert werden soll.
Output : Index j des letzten Elementes von A, das $\leq$ dem Pivotelement ist.

```
1 pivot = A[l]; i = l; j = r;
2 while i < j do
  |  /* Alle, die bezüglich des Pivotelementes richtig stehen,
  |     übergehen. */
3 |  while A[i] < pivot do  i = i + 1;
4 |  while A[j] > pivot do  j = j − 1;
5 |  if i < j then
6 |  |  t = A[i]; A[i] = A[j]; A[j] = t;   /* A[i] und A[j] vertauschen. */
7 |  end
8 end
9 return j;
```

Sehen wir uns einige mögliche Prozedurbäume an:



Hier ist die Laufzeit $O(n \cdot \log n) = O(n) + 2 \cdot O(\frac{n}{2}) + 4 \cdot O(\frac{n}{4}) + \ldots + n \cdot O(1) + \underbrace{O(n)}_{\text{Blätter}}$.
Das stellt praktisch den Idealfall dar, den wir uns normalerweise wünschen. Es ist aber auch folgendes möglich, wenn die Eingabe „ungünstig“ ist.



Jetzt erhalten wir für die Laufzeit:

$$\begin{aligned} n + n - 1 + \ldots + 3 + 2 + 1 = \frac{n \cdot (n-1)}{2} &= O(n^2) \quad \text{(Aufteilen)} \\ 1 + 1 + \ldots + 1 + 1 + 1 &= O(n) \quad \text{(Zusammensetzen)} \end{aligned}$$

Also: $O(n^2)$ und auch $\Omega(n^2)$.

Wieso wird Quicksort trotzdem in der Praxis häufig eingesetzt? In der Regel tritt ein „gutartiger“ Baum auf, und wir haben $O(n \cdot \log n)$. Vergleiche innerhalb eines Aufrufes finden immer mit einem festem a statt. Dieser Wert kann in einem Register gehalten werden. Damit erreicht man ein schnelleres Vergleichen. Dagegen sind beim Mischen von Mergesort öfter beide Elemente des Vergleiches neu.

Der Aufrufbaum von Quicksort ist vorher nicht zu erkennen. Dadurch ist keine nicht-rekursive Implementierung wie bei Mergesort möglich. Nur mit Hilfe eines (Rekursions-)Kellers.

11.3 Rekursionsgleichungen

Noch einmal zu Mergesort. Wir betrachten den Fall, dass n eine Zweierpotenz ist. Dann sind $\frac{n}{2}, \frac{n}{4}, \ldots, 1 = 2^0$ ebenfalls Zweierpotenzen. Sei

$$T(n) = \text{ worst-case-Zeit bei } A[1, \ldots, n],$$

dann gilt für ein geeignetes c:

$$\begin{aligned} T(1) &\leq c \\ T(n) &\leq c \cdot n + 2 \cdot T\left(\frac{n}{2}\right) \end{aligned}$$

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

Betrachten wir nun den Fall

$$\begin{aligned} T(1) &= c \\ T(n) &= c \cdot n + 2 \cdot T\left(\frac{n}{2}\right). \end{aligned}$$

Dann erhalten wir durch Abwickeln der rekursiven Gleichung:

$$\begin{aligned} T(n) &= c \cdot n + 2 \cdot T\left(\frac{n}{2}\right) \\ &= c \cdot n + 2 \cdot c \cdot \frac{n}{2} + 2 \cdot 2 \cdot T\left(\frac{n}{4}\right) \\ &= c \cdot n + c \cdot n + 4 \cdot c \cdot \frac{n}{2} + 2 \cdot 2 \cdot 2 \cdot T\left(\frac{n}{8}\right) \\ &\vdots \\ &= c \cdot n \cdot \log n + 2 \cdot 2 \cdot \cdots \cdot 2 \cdot T(1) \\ &= c \cdot n \cdot \log n + c \cdot n \\ &= O(c \cdot n \log n) = O(n \cdot \log n) \end{aligned}$$

Schließlich noch ein strenger Induktionsbeweis, dass $T(n) \leq d \cdot n \cdot \log n = O(n \cdot \log n)$ ist.

Induktionsanfang:

$$\begin{array}{rcll} T(1) & = & c \neq O(1 \cdot \log 1) = 0, \text{ das stimmt so nicht. Also fangen wir bei } T(2) \text{ an.} \\ T(2) & = & c \cdot 2 + 2 \cdot T(1) = 4 \cdot c & \\ & \leq & d \cdot 2 \cdot \log 2 \ = 2 \cdot d & \text{(Gilt für } d \geq 2 \cdot c\text{.)} \\ & = & O(2 \cdot \log 2) \quad \checkmark & \end{array}$$

Die Behauptung gilt also für $n = 2$.

Induktionsschluss:

$$\begin{array}{rcll} T(2n) & = & c \cdot 2n + 2 \cdot T(n) & \text{(nach Definition)} \\ & \leq & c \cdot 2n + 2 \cdot d \cdot n \cdot \log n & \text{(nach Induktionsvoraussetzung)} \\ & \leq & d \cdot n + 2 \cdot d \cdot n \cdot \log n & \text{(mit } d \geq 2 \cdot c \text{ von oben)} \\ & = & d \cdot 2n - d \cdot n + 2 \cdot d \cdot n \cdot \log n & \\ & = & d \cdot 2n \cdot (1 + \log n) - d \cdot n & \\ & \leq & d \cdot 2n \cdot \log 2n & \\ & = & O(2n \cdot \log 2n) \quad \square & \end{array}$$

Wie sieht das bei Quicksort aus? Wir betrachten

$$T(n) = \text{ worst-case-Zeit von Quicksort auf } A[1, \ldots, n].$$

Dann gelten die folgenden Ungleichungen. Damit sind alle Möglichkeiten beschrieben, wie das Feld aufgeteilt werden kann. Einer dieser Fälle muß der worst-case sein.

$$\begin{array}{rcl} T(n) & \leq & c \cdot n + T(n-1) \\ T(n) & \leq & c \cdot n + T(1) + T(n-2) \\ T(n) & \leq & c \cdot n + T(2) + T(n-3) \\ & \vdots & \\ T(n) & \leq & c \cdot n + T(n-2) + T(1) \\ T(n) & \leq & c \cdot n + T(n-1) \end{array}$$

Also haben wir

$$\begin{array}{rcl} T(1) & \leq & c \\ T(n) & \leq & c \cdot n + \max\Big\{\big\{T(n-1)\big\} \cup \big\{T(i) + T(n-i-1) \mid 1 \leq i \leq n-2\big\}\Big\}. \end{array}$$

Dann gilt $T(n) \leq d \cdot n^2$ für ein geeignetes d.

Induktionsanfang:

$$\begin{array}{rcll} T(1) & \leq & c \cdot 1 \leq d \cdot 1^2 & \text{gilt für } d \geq c. \checkmark \\ T(2) & \leq & c \cdot 2 + T(1) & \\ & \leq & 3c \leq d \cdot 2^2 & \text{gilt für } d \geq \frac{3}{4}c. \checkmark \end{array}$$

Wir wählen $d \geq c$, dann gilt die Behauptung für alle $n \leq 2$.

Indukstionsschluss:

$$\begin{array}{rcl} & & \text{(nach Definition)} \\ T(n+1) & \leq & c(n+1) + \max\Big\{\big\{T(n)\big\} \cup \big\{T(i) + T((n+1) - i - 1) \mid 1 \leq i \leq (n+1) - 2\big\}\Big\} \\ & = & c(n+1) + \max\Big\{\big\{T(n)\big\} \cup \big\{T(i) + T(n-i) \mid 1 \leq i \leq n-1\big\}\Big\} \\ & & \\ & & \text{(nach Induktionsvoraussetzung)} \\ & \leq & c(n+1) + \max\Big\{\big\{dn^2\big\} \cup \big\{di^2 + d(n-i)^2 \mid 1 \leq i \leq n-1\big\}\Big\} \\ & = & c(n+1) + \max\Big\{\big\{dn^2\big\} \cup \underbrace{\big\{ dn^2 + 2di\underbrace{(i-n)}_{<0} \mid 1 \leq i \leq n-1\big\}}_{<dn^2}\Big\} \\ & \leq & c(n+1) + dn^2 \\ & & \\ & & \text{(mit } d \geq c\text{)} \\ & \leq & d(n^2+n+1) \leq d(n+1)^2 \qquad \square \end{array}$$

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

11.4 Multiplikation großer Zahlen

Im Folgenden behandeln wir das Problem der Multiplikation großer Zahlen. Bisher haben wir angenommen:

$$\text{Zahlen passen in ein Speicherwort} \Longrightarrow \text{arithmetische Ausdrücke in } O(1).$$

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 $n \geq 1$ werden $\lfloor \log_2 n \rfloor + 1$ Bits benötigt. Man misst nicht in der Zahl selbst!

Die normale Methode, zwei Zahlen zu *addieren*, lässt sich in $O(n)$ Bitoperationen implementieren, wenn die Zahlen die Länge n haben.

$$\begin{array}{rccccc} & a_1 & \ldots & & a_n \\ + & b_1 & \ldots & & b_n \\ \hline & & \ldots & a_n + b_n & \end{array}$$

Multiplikation in $O(n^2)$:

$$\begin{array}{rll} (a_1 \ldots a_n) \cdot (b_1 \ldots b_n) = & (a_1 \ldots a_n) \cdot (b_1\ 0\ \ldots\ 0) & O(n) \\ + & (a_1 \ldots a_n) \cdot (b_2 0 \ldots\ 0) & O(n) \\ & \ddots & \\ + & (a_1 \ldots a_n) \cdot (b_n) & O(n) \\ \hline = & \textit{(Summenbildung)} & \end{array}$$

Zur Berechnung des Ergebnissen sind noch $n-1$ Additionen mit Zahlen der Länge $\leq 2n$ notwendig. Insgesammt haben wir dann $O(n^2)$ viele Operationen.

Mit divide-and-conquer geht es besser! Nehmen wir zunächst einmal an, n ist eine Zweierpotenz. Dann können wir die Zahlen auch so schreiben:

$$\begin{array}{ccccc} \overbrace{a_1 \ldots a_n}^{a:=} & = & \overbrace{a_1 \ldots a_{\frac{n}{2}}}^{a':=} & \overbrace{a_{\frac{n}{2}+1} \ldots a_n}^{a'':=} \\ \underbrace{b_1 \ldots b_n}_{b:=} & = & \underbrace{b_1 \ldots b_{\frac{n}{2}}}_{b':=} & \underbrace{b_{\frac{n}{2}+1} \ldots b_n}_{b'':=} \end{array}$$

Nun schreiben wir $a \cdot b$ als:

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

Mit den vier Produkten, bei denen die Faktoren nur noch die Länge $\frac{n}{2}$ haben, machen wir rekursiv weiter. Das Programm sieht in etwa so aus:

Algorithmus 23: Multiplikation großer Zahlen

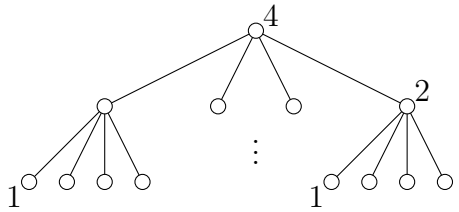
Input : a, b zwei Zahlen mit n Bits
Output : Produkt $a \cdot b$

1 **if** $n == 1$ **then return** $a \cdot b$;

`/* Divide */`
2 $a' = a_1 \dots a_{\frac{n}{2}}$; $a'' = a_{\frac{n}{2}+1} \dots a_n$;
3 $b' = b_1 \dots b_{\frac{n}{2}}$; $b'' = b_{\frac{n}{2}+1} \dots b_n$;
4 $m_1 = \text{Mult}(a', b')$; $m_2 = \text{Mult}(a', b'')$;
5 $m_3 = \text{Mult}(a'', b')$; $m_4 = \text{Mult}(a'', b'')$;

`/* Conquer */`
6 **return** $m_1 \cdot 2^n + (m_2 + m_3) \cdot 2^{n/2} + m_4$;

Der Aufrufbaum für $n = 4$ sieht dann wie unten aus. Die Tiefe ist $\log_2 4 = 2$.



Laufzeit? Bei $\text{Mult}(a, b)$ 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:	$4 \cdot 4 \cdot d \cdot \frac{n}{4}$	
$\vdots$		
$\log_2 n$-te Stufe:	$4^{\log_2 n - 1} \cdot d \cdot \frac{n}{2^{\log_2 n - 1}}$	
#Blätter:	$4^{\log_2 n} \cdot d$	

Das gibt:

$$\begin{aligned} T(n) &\leq \sum_{i=0}^{\log_2 n} 4^i \cdot d \cdot \frac{n}{2^i} \\ &= d \cdot n \cdot \sum_{i=0}^{log_2 n} 2^i \\ &= d \cdot n \cdot \frac{2^{\log_2 n+1} - 1}{2-1} \\ &= O(n^2) \qquad \text{mit } \sum_{i=0}^{k} x^i = \frac{x^{k+1}-1}{x-1} \quad \text{für alle } x \neq 1, \textit{ geometrische Reihe.} \end{aligned}$$

Betrachten wir die induzierte Rekursionsgleichung. Annahme: $n = 2^k$ Zweierpotenz. (*Beachte:* $2^{\lfloor \log_2 n \rfloor} \leq n \leq 2^{\lfloor \log_2 n \rfloor + 1}$, das heißt zwischen n und $2n$ existiert eine Zweierpotenz.)

$$\begin{aligned} T(1) &= d \\ T(n) &= dn + 4 \cdot T\left(\frac{n}{2}\right) \end{aligned}$$

Versuchen $T(n) = O(n^2)$ durch Induktion zu zeigen.

1. Versuch: $T(n) \leq d \cdot n^2$

Induktionsanfang: ✓

Induktionsschluss:

$$\begin{aligned} T(n) &= dn + 4 \cdot T\left(\frac{n}{2}\right) \\ &\leq dn + dn^2 \quad > dn^2 \qquad \text{Induktion geht nicht!} \end{aligned}$$

Wir müssen irgendwie das dn unterbringen.

2. Versuch: $T(n) \leq 2dn^2$ gibt im Induktionsschluss:

$$T(n) \leq dn + 2dn^2 > 2dn^2$$

3. Versuch: $T(n) \leq 2dn^2 - dn$ „Überraschenderweise“ ist das ganau das, was wir uns oben für die Laufzeit überlegt haben.

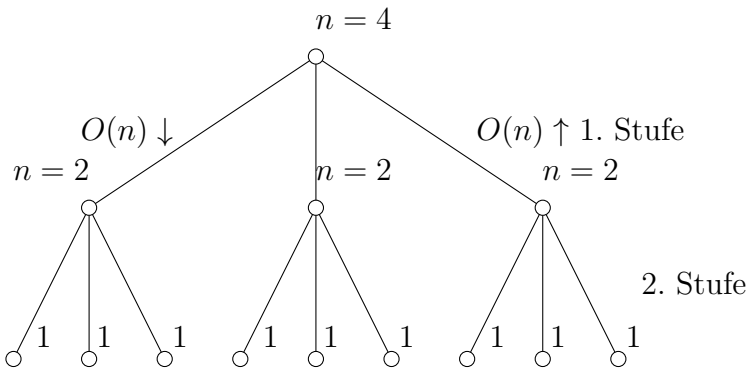
Induktionsanfang: $T(1) \leq 2d - d = d$ ✓

Induktionsschluss:

$$\begin{aligned} T(n) &= dn + 4 \cdot T\left(\frac{n}{2}\right) \\ &\leq dn + 4\left(2d\left(\frac{n}{2}\right)^2 - d\frac{n}{2}\right) \\ &= dn + 2dn^2 - 2dn \\ &= 2dn^2 - dn \qquad \square \end{aligned}$$

Vergleiche auch Seite 163, wo $\log_2 \frac{n}{2} = \log_2 n - 1$ wichtig ist.

Bisher haben wir mit dem *divide-and-conquer*-Ansatz also noch nichts gegenüber der einfachen Methode gewonnen. Unser nächstes Ziel ist die Verbesserung der Multiplikation durch nur noch *drei rekursive Aufrufe* mit jeweils $\frac{n}{2}$ vielen Bits. Analysieren wir zunächst die Laufzeit.



Wir haben wieder $\log_2 n$ viele divide-and-conquer-Schritte und die Zeit an den Blättern.

1. Stufe:	$d \cdot n$
2. Stufe:	$3 \cdot d \cdot \frac{n}{2}$
3. Stufe:	$3 \cdot 3 \cdot d \cdot \frac{n}{4}$
⋮	
$\log_2 n$-te Stufe:	$3^{\log_2 n - 1} \cdot d \cdot 2$
#Blätter:	$3^{\log_2 n} \cdot d \cdot 1$

Dann ist die Laufzeit

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_2 n} 3^i \cdot d \cdot \frac{n}{2^i} \\
&= d \cdot n \cdot \sum_{i=0}^{\log_2 n} \left(\frac{3}{2}\right)^i \\
&= d \cdot n \cdot \frac{(\frac{3}{2})^{\log_2 n+1} - 1}{\frac{3}{2} - 1} \\
&\leq 2 \cdot d \cdot n \cdot \frac{3}{2} \cdot \left(\frac{3}{2}\right)^{\log_2 n} \\
&= 3 \cdot d \cdot n \cdot 2^{\log_2(3/2) \cdot \log_2 n} \\
&= 3 \cdot d \cdot n \cdot n^{\overbrace{\log_2(3/2)}^{<1}} \\
&= 3 \cdot d \cdot n \cdot n^{\log_2 3 - \overbrace{\log_2 2}^{=1}} \\
&= 3 \cdot d \cdot n^{log_2 3} = O(n^{1.59}) \qquad \square
\end{aligned}$$

also tatsächlich besser als $O(n^2)$. Fassen wir zusammen:

- Zwei Aufrufe mit $\frac{n}{2}$, linearer Zusatzaufwand:

  (Auf jeder Stufe $d \cdot n +$ 2 Aufrufe mit halber Größe.)

$$\begin{aligned}
T(n) &= d \cdot n \cdot \sum_{i=0}^{\log_2 n} \underbrace{\frac{2^i}{2^i}}_{=1} \\
&= O(n \cdot \log n)
\end{aligned}$$

- Drei Aufrufe:

$$\begin{aligned}
T(n) &= d \cdot n \cdot \sum \left(\frac{3}{2}\right)^i \qquad \text{(3 Aufrufe, halbe Größe)} \\
&= O(n^{\log_2 3})
\end{aligned}$$

- Vier Aufrufe:

$$\begin{aligned}
T(n) &= d \cdot n \cdot \sum \left(\frac{4}{2}\right)^i \qquad \text{(4 Aufrufe, halbe Größe)} \\
&= d \cdot n \cdot n \\
&= O(n^2)
\end{aligned}$$

Aufgabe: Stellen Sie für den Fall der drei Aufrufe die Rekursionsgleichungen auf und beweisen Sie $T(n) = O(n^{log_2 3})$ durch eine ordnungsgemäße Induktion.

Zurück zur Multiplikation.

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

Wie können wir einen Aufruf einsparen? Erlauben uns zusätzliche Additionen. Wir beobachten zunächst allgemein:

$$(x-y)\cdot(u-v) = x(u-v) + y\cdot(u-v) = x\cdot u - x\cdot v + y\cdot u - y\cdot v$$

Wir haben *eine explizite* und *vier implizite Multiplikationen*, die in der Summe verborgen sind, durchgeführt.

Wir versuchen jetzt $a'b'' + a''b'$ auf einen Schlag zu ermitteln:

$$(a'-a'')\cdot(b''-b') = a'b'' - a'b' - a''b'' + a''b'$$

Die Produkte $a'b'$ und $a''b''$ berechnen wir normal. Damit bekommen wir

$$a'b'' + a''b' = (a'-a'')\cdot(b''-b') + a'b' + a''b''.$$

Die Terme $(a'-a'')$ und $(b''-b')$ können < 0 sein. Diese Fälle müssen wir in unsere Rechnung mit einbeziehen.

Algorithmus 24: Multiplikation großer Zahlen (schnell)

Input : a, b zwei Zahlen mit n Bits
Output : Produkt $a \cdot b$

1 **if** $n == 1$ **then return** $a \cdot b$;

`/* Divide */`
2 $a' = a_1 \dots a_{\frac{n}{2}}$; $a'' = a_{\frac{n}{2}+1} \dots a_n$; $b' = b_1 \dots b_{\frac{n}{2}}$; $b'' = b_{\frac{n}{2}+1} \dots b_n$;
3 $m_1 = \text{Mult}(a', b')$; $m_2 = \text{Mult}(a'', b'')$;

`/* Beachte:` $|a'' - a'|$ `und` $|b' - b''|$ `ist immer` $\leq 2^{n/2} - 1$`, es reichen also immer` $\frac{n}{2}$ `viele Bits. */`
4 **if** $(a' - a'') > 0$ *und* $(b'' - b') > 0$ **then** $m_3 = \text{Mult}(a' - a'', b'' - b')$;
5 **if** $(a' - a'') < 0$ *und* $(b'' - b') < 0$ **then** $m_3 = \text{Mult}(a'' - a', b' - b'')$;
6 **if** $(a' - a'') < 0$ *und* $(b'' - b') > 0$ **then** $m_3 = -\text{Mult}(a'' - a', b'' - b')$;
7 **if** $(a' - a'') > 0$ *und* $(b'' - b') < 0$ **then** $m_3 = -\text{Mult}(a' - a'', b' - b'')$;
8 **if** $(a' - a'') == 0$ *oder* $(b'' - b') == 0$ **then** $m_3 = 0$;

`/* Conquer */`
9 **return** $m_1 \cdot 2^n + \underbrace{(m_3 + m_1 + m_2)}_{\text{Zeit } O(n/2)} \cdot 2^{n/2} + m_2$;

Damit bekommen wir für die Laufzeit:

$$\begin{aligned} T(1) &\leq d \\ T(n) &\leq d \cdot n + 3 \cdot T\left(\frac{n}{2}\right) \quad \Rightarrow O(n^{log_2 3}). \end{aligned}$$

Die Addition zweier Zahlen geht linear in der Anzahl der Bits. Für die Multiplikation ist kein Linearzeitverfahren bekannt.

11.5 Schnelle Matrixmultiplikation

Wir gehen jetzt wieder davon aus, dass die Basisoperationen in der Zeit $O(1)$ durchführbar sind.

Erinnern wir uns an die Matrizenmultiplikation. Dabei betrachten wir zunächst nur *quadratische Matrizen*.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

Im allgemeinen haben wir bei zwei $n \times n$-Matrizen:

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nn} \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^{n} a_{1k}b_{k1} & \dots & \sum_{k=1}^{n} a_{1k}b_{kn} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} a_{nk}b_{k1} & \dots & \sum_{k=1}^{n} a_{nk}b_{kn} \end{pmatrix}$$

Der Eintrag c_{ij} der Ergebnismatrix berechnet sich als $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$.

Laufzeit:

- Pro Eintrag des Ergebnisses n Multiplikationen und $n-1$ Additionen, also $O(n)$.
- Bei n^2 Einträgen $O(n^3)$ $(= O(n^2)^{3/2})$.

Überraschend war seinerzeit, dass es mit divide-and-conquer besser geht. Wie könnte ein divide-and-conquer-Ansatz funktionieren? Teilen wir die Matrizen einmal auf:

$$\left(\begin{array}{ccc|ccc} a_{1,1} & \dots & a_{1,\frac{n}{2}} & a_{1,\frac{n}{2}+1} & \dots & a_{1,n} \\ \vdots & & \ddots & & & \vdots \\ a_{\frac{n}{2},1} & \dots & a_{\frac{n}{2},\frac{n}{2}} & a_{\frac{n}{2},\frac{n}{2}+1} & \dots & a_{\frac{n}{2},n} \\ \hline a_{\frac{n}{2}+1,1} & \dots & a_{\frac{n}{2}+1,\frac{n}{2}} & a_{\frac{n}{2}+1,\frac{n}{2}+1} & \dots & a_{\frac{n}{2}+1,n} \\ \vdots & & & \ddots & & \vdots \\ a_{n,1} & \dots & a_{n,\frac{n}{2}} & a_{n,\frac{n}{2}+1} & \dots & a_{n,n} \end{array}\right) = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

Die $A_{i,j}$ sind $\frac{n}{2} \times \frac{n}{2}$-Matrizen. Wir nehmen wieder an, dass n eine Zweierpotenz ist.

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad \text{ist dann analog aufgeteilt.}$$

Das Ergebnis schreiben wir dann so:

$$A \cdot B = \begin{pmatrix} \sum_{k=1}^{n} a_{1k} b_{k1} & \dots & \sum_{k=1}^{n} a_{1k} b_{kn} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} a_{nk} b_{k1} & \dots & \sum_{k=1}^{n} a_{nk} b_{kn} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Betrachten wir einmal die Submatrix C_{11} des Ergebnisses.

$$\begin{aligned} C_{11} &= \begin{pmatrix} \sum_{k=1}^{n} a_{1,k} b_{k,1} & \dots & \sum_{k=1}^{n} a_{1,k} b_{k,\frac{n}{2}} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} a_{\frac{n}{2},k} b_{k,1} & \dots & \sum_{k=1}^{n} a_{\frac{n}{2},k} b_{k,\frac{n}{2}} \end{pmatrix} \\ &= \begin{pmatrix} \sum_{k=1}^{\frac{n}{2}} a_{1,k} b_{k,1} + \sum_{k=\frac{n}{2}+1}^{n} a_{1,k} b_{k,1} & \dots \\ \vdots & \ddots \quad \vdots \\ \sum_{k=1}^{\frac{n}{2}} a_{\frac{n}{2},k} b_{k,1} + \sum_{k=\frac{n}{2}+1}^{n} a_{\frac{n}{2},k} b_{k,1} & \dots \end{pmatrix} \end{aligned}$$

Bei genauem hinsehen, ergibt sich $C_{11} = A_{11}B_{11} + A_{12}B_{21}$, denn

$$A_{11}B_{11} = \begin{pmatrix} \sum_{k=1}^{\frac{n}{2}} a_{1,k}b_{k,1} & \cdots & \sum_{k=1}^{\frac{n}{2}} a_{1,k}b_{k,\frac{n}{2}} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{\frac{n}{2}} a_{\frac{n}{2},k}b_{k,1} & \cdots & \sum_{k=1}^{\frac{n}{2}} a_{\frac{n}{2},k}b_{k,\frac{n}{2}} \end{pmatrix} \quad \text{und}$$

$$A_{12}B_{21} = \begin{pmatrix} \sum_{k=\frac{n}{2}+1}^{n} a_{1,k}b_{k,1} & \cdots & \sum_{k=\frac{n}{2}+1}^{n} a_{1,k}b_{k,\frac{n}{2}} \\ \vdots & \ddots & \vdots \\ \sum_{k=\frac{n}{2}+1}^{n} a_{\frac{n}{2},k}b_{k,1} & \cdots & \sum_{k=\frac{n}{2}+1}^{n} a_{\frac{n}{2},k}b_{k,\frac{n}{2}} \end{pmatrix}$$

Für die restlichen Submatrizen ergibt sich ganz analog

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22}. \end{aligned}$$

Das sieht genauso aus wie wenn die A_{ij}, B_{ij} und C_{ij} ganz normale Skalare wären.

Für die Anzahl der Operationen ergibt das die folgende Rekurrenz für ein geeignetes d.

$$\begin{aligned} T(1) &\leq d \\ T(n) &\leq \underbrace{d \cdot n^2}_{\substack{\text{linear in der} \\ \#\text{Einträge}}} + 8 \cdot \underbrace{T\left(\frac{n}{2}\right)}_{\substack{(n/2)^2 \,=\, n^2/4 \\ \text{Einträge}}} \end{aligned}$$

Das führt (analog zur Rechnung auf Seite 168) zu

$$T(n) \leq d \cdot n^2 \cdot \sum_{i=0}^{\log_2 n} \left(\frac{8}{4}\right)^i = O(n^3).$$

Angenommen, wir könnten, ähnlich wie bei der Multiplikation großer Zahlen, eine der acht Multiplikationen der $\frac{n}{2} \times \frac{n}{2}$-Matrizen einsparen. Wir bekämen dann nur noch sieben rekursive Aufrufe. Das führt auf die Rekurrenz:

$$\begin{aligned} T(1) &\leq d \\ T(n) &\leq d \cdot n^2 + 7 \cdot T\left(\frac{n}{2}\right) \end{aligned}$$

Das führt auf

$$\begin{aligned} T(n) &\leq d \cdot n^2 \cdot \sum_{i=0}^{\log_2 n} \left(\frac{7}{2^2}\right)^i \\ &= d \cdot n^2 \cdot \frac{(7/4)^{\log_2 n+1} - 1}{1 - 7/4} \\ &= \frac{7}{3} d \cdot n^{\log_2 n} - \frac{4}{3} d \cdot n^2 \\ &= O(n^{\log_2 7}), \end{aligned}$$

da $\log_2 7 < 2.81 < 3$. Die entsprechenden Rechnungen sind eine gute Übung zur Induktion.

Wir brauchen immernoch

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22}. \end{aligned}$$

Wir suchen jetzt sieben Produkte von $\frac{n}{2} \times \frac{n}{2}$-Matrizen, $P_1, \ldots, P_7$, so dass die C_{ij} ohne weitere Multiplikationen erzeugt werden können. Wir betrachten Produkte der Art

$$(A_{11} - A_{21})(B_{11} + B_{21})$$

und ähnlich. Dazu führen wir zur übersichtlichen Darstellung noch die folgende Notation ein.

Wir schreiben C_{11} noch einmal etwas anders hin. Dabei betrachten wir die A_{ij}, B_{ij} jetzt einfach nur als Symbole.

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ &= (A_{11}, A_{12}, A_{21}, A_{22}) \cdot \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} B_{11} \\ B_{21} \\ B_{12} \\ B_{22} \end{pmatrix} \end{aligned}$$

Dafür schreiben wir jetzt:

$$\begin{array}{c} \\ A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{array} \begin{array}{c} \begin{array}{cccc} B_{11} & B_{21} & B_{12} & B_{22} \end{array} \\ \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{array} = C_{11}$$

Wenn wir Minuszeichen eintragen, erhalten wir zum Beispiel folgendes:

$$\begin{pmatrix} - & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = (-A_{11} - A_{12}) \cdot B_{11}$$

Beachte: Das Matrixprodukt ist assoziativ, da $(AB)C = A(BC)$ für alle A, B, C $n \times n$-Matrizen. Aber es ist *nicht kommutativ*, da im allgemeinen $AB \neq BA$ ist. Etwa bei

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 2 & 1 \\ 2 & 3 \end{pmatrix}.$$

In unserer Schreibweise suchen wir jetzt:

$$C_{11} = \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = \begin{pmatrix} \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \end{pmatrix} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix} = A_{21}B_{12} + A_{22}B_{22}$$

Wir versuchen, die Produkte $P_1, \ldots, P_7$ mit jeweils einer Multiplikation zu ermitteln. Wir beginnen einmal mit $C_{12} = P_1 + P_2$.

P_1 könnte $\begin{pmatrix} \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$ und P_2 könnte $\begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$ sein.

Dann ist $C_{12} = A_{11}B_{12} + A_{12}B_{22} = P_1 + P_2$. Damit gewinnen wir allerdings nichts. Eine andere Möglichkeit wäre

$$P_1 = \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = (A_{11} + A_{12})B_{22}, \quad P_2 = \begin{pmatrix} \cdot & \cdot & + & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = A_{11}(B_{12} - B_{22}).$$

Gibt es noch andere Matrizen für ein einzelnes Produkt?

$$\begin{pmatrix} \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = A_{11}(B_{12} + B_{22}),$$

$$\begin{pmatrix} \cdot & + & \cdot & \cdot \\ \cdot & - & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & - & \cdot & \cdot \end{pmatrix} = (A_{11} - A_{12} + A_{21} - A_{22})B_{21}$$

Beachte: Die A_{ij} stehen immer links!

Wir nehmen jetzt die folgenden offiziellen P_1, P_2 für C_{12}.

$$C_{12} = \begin{pmatrix} \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = \underbrace{\begin{pmatrix} \cdot & \cdot & + & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}}_{:=P_1} + \underbrace{\begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}}_{:=P_2}$$

$$\begin{aligned} P_1 &= A_{11}(B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12})B_{22} \end{aligned}$$

Ebenso P_3, P_4 für C_{21}.

$$C_{21} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \end{pmatrix} = \underbrace{\begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}}_{:=P_3} + \underbrace{\begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & + & \cdot & \cdot \end{pmatrix}}_{:=P_4}$$

$$\begin{aligned} P_3 &= (A_{21} + A_{22})B_{11} \\ P_4 &= A_{22}(-B_{11} + B_{21}) \end{aligned}$$

Bis jetzt haben wir vier Multiplikationen verbraucht und C_{12} und C_{21} erzielt. Das ist noch nichts neues! Wir haben noch drei Multiplikationen für

$$C_{11} = \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}, \quad C_{22} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}$$

übrig. Beachte $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$, die Hauptdiagonale fehlt noch.

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22}. \end{aligned}$$

Versuchen wir einmal zwei Produkte auf der rechten Seite gleichzeitig zu berechnen. Wir nehmen P_5 wie unten, das ist auch ein einzelnes Produkt von Matrizen.

$$P_5 = \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix} = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$P_5 = \underbrace{A_{11}B_{11}}_{\text{in } C_{11}} + \overbrace{\underbrace{A_{11}B_{22}}_{\text{in } P_1,P_2} + \underbrace{A_{22}B_{11}}_{\text{in } P_3,P_4}}^{\text{Das ist zuviel.}} + \underbrace{A_{22}B_{22}}_{\text{in } C_{22}}$$

Abhilfe schafft hier zunächst einmal

$$\begin{aligned} P_5 + P_4 - P_2 &= \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & + & \cdot & \cdot \end{pmatrix} - \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \\ &= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \end{pmatrix} = A_{11}B_{11} + A_{22}B_{21} + A_{22}B_{22} - A_{12}B_{22}. \end{aligned}$$

Mit der Addition von P_6 bekommen wir dann endlich C_{11}.

$$\begin{aligned} P_6 &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & - & \cdot & - \end{pmatrix} = (A_{12} - A_{22})(B_{21} + B_{22}) \\ &= A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22} \\ C_{11} &= (P_5 + P_4 - P_2) + P_6 = A_{11}B_{11} + A_{12}B_{21} \end{aligned}$$

Für C_{22} benutzen wir auch wieder P_5 und berechnen

$$\begin{aligned} P_5 + P_1 - P_3 &= \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix} + \begin{pmatrix} \cdot & \cdot & + & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} - \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix} \\ &= \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix} = A_{11}B_{11} + A_{11}B_{12} - A_{21}B_{11} + A_{22}B_{22}. \end{aligned}$$

Mit der letzten Multiplikation berechnen wir P_7 und C_{22}.

$$\begin{aligned} P_7 &= \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & - & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} = (A_{11} - A_{21})(B_{11} + B_{12}) \\ &= A_{11}B_{11} + A_{11}B_{12} - A_{21}B_{11} - A_{21}B_{12} \\ C_{22} &= (P_5 + P_1 - P_3) - P_7 = A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Mit diesen $P_1, \ldots, P_7$ kommen wir dann insgesammt auf $O(n^{\log_2 7})$.

12 Kombinatorische Suche

Bisher haben wir meistens Probleme in polynomieller Zeit (und damit auch Platz) betrachtet. 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 Fluß systematisch aufbauen, ohne blind durchzuprobieren. Dadurch erreichen wir polynomiale Zeit.

Jetzt betrachten wir Probleme, bei denen man die Lösung nicht mehr zielgerichtet aufbauen kann, sondern im Wesentlichen exponentiell viele Lösungskandidaten durchsuchen muss. Das bezeichnet man auch als *kombinatorische Suche*.

12.1 Aussagenlogische Probleme

Aussagenlogische Probleme, wie zum Beispiel

$$x \wedge y \vee \Big(\neg\big(u \wedge \neg(v \wedge \neg x)\big) \to y\Big), \quad x \vee y, \quad x \wedge y, \quad (x \vee y) \wedge (x \vee \neg y).$$

Also wir haben eine Menge von aussagenlogischen Variablen zur Verfügung, wie x, y, v, u. Formeln werden mittels der üblichen aussagenlogischen Operationen $\wedge$ (und), $\vee$ (oder, lat. vel), $\neg, \bar{}$ (nicht), $\to$ (Implikation), $\Leftrightarrow$ (Äquivalenz) aufgebaut.

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

x	y	$x \to y$
0	0	1
0	1	1
1	0	0
1	1	1

Das heißt, das Ergebnis ist nur dann falsch, wenn aus Wahrem Falsches folgen soll.

Die Äquivalenz $x \Leftrightarrow y$ 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 $(x \to y) \wedge (y \to x)$.

Ein Beispiel verdeutlicht die Relevanz der Aussagenlogik:

Beispiel 12.1: Frage: *Worin besteht das Geheimnis Ihres langen Lebens?*

Antwort: *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.*

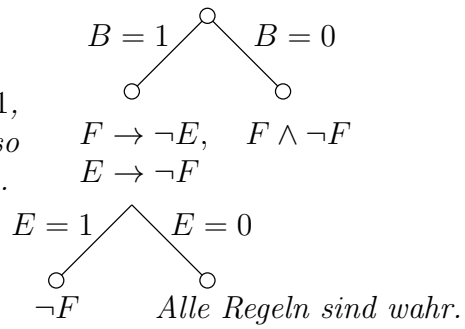
Wir wählen die folgenden aussagenlogischen Variablen:

$$\begin{aligned} B &= \textit{Bier beim Essen} \\ F &= \textit{Fisch} \\ E &= \textit{Eis} \end{aligned}$$

Ausagenlogisch werden obige Aussagen nun zu:

$$\begin{aligned} \neg B &\rightarrow F \\ F \wedge B &\rightarrow \neg E \\ E \vee \neg B &\rightarrow \neg F \end{aligned}$$

Die 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 Wahrheitswert haben. Das heißt wir haben eine Abbildung

$$a : \text{Variablen} \to \{0, 1\}.$$

Das heißt eine Belegung der Variablen mit Wahrheitswerten ist gegeben. Dann ist $a(F)$ = Wahrheitswert von F bei Belegung a.

Ist $a(x) = a(y) = a(z) = 1$, dann

$$a(\neg x \vee \neg y) = 0, \quad a(\neg x \vee \neg y \vee z) = 1, \quad a(\neg x \wedge (y \vee z)) = 0.$$

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.

Definition 12.1: *Wir bezeichnen F als*

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

12.2 Aussagenlogisches Erfüllbarkeitsproblem

Wir betrachten zunächst einen einfachen Algorithmus zur Lösung des aussagenlogischen Erfüllbarkeitsproblems.

Algorithmus 25: Erfüllbarkeit

Input : Eine aussagenlogische Formel F.
Output : „erfüllbar“ falls F erfüllbar ist, sonst „unerfüllbar“.

```
1 foreach Belegung a ∈ {0,1}^n do        /* a(0...0),...,a(1...1) */
2 |  if a(F) = 1 then
3 |  |  return „erfüllbar durch a“;
4 |  end
5 end
6 return „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! Wir betrachten hier den reinen worst-case-Fall.

Eine Verbesserung kann durch Backtracking erreicht werden: Schrittweises Einsetzen der Belegung und Vereinfachen von F (*Davis-Putnam-Prozedur*, siehe später).

Die Semantik einer Formel F mit n Variablen lässt sich auch verstehen als eine Funktion $F : \{0,1\}^n \to \{0,1\}$.

Wieviele derartige Funktionen gibt es? Jede derartige Funktion lässt sich in *konjunktiver Normalform (KNF)* oder auch in *disjunktiver Normalform (DNF)* darstellen.

Eine Formel in *konjunktiver Normalform ist zum Beispiel*

$$F_{\text{KNF}} = (x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee x_2),$$

und eine Formel in *disjunktiver Normalform ist*

$$F_{\text{DNF}} = (x_1 \wedge x_2 \wedge \neg x_3 \wedge x_4) \vee (x_2 \wedge x_3 \wedge \neg x_4) \vee (\neg x_1 \wedge x_5 \wedge \ldots \wedge x_n).$$

Im Prinzip reichen DNF oder KNF aus. Das heißt, ist $F : \{0,1\}^n \to \{0,1\}$ eine boolesche Funktion, so lässt sich F als KNF oder auch DNF darstellen.

KNF: Wir gehen alle $(b_1, \ldots, b_n) \in \{0,1\}^n$, für die $F(b_1, \ldots, b_n) = 0$ ist, durch.

Wir schreiben Klauseln nach folgendem Prinzip:

$$\begin{aligned} F(0\ldots0) = 0 &\quad\to\quad x_1 \vee x_2 \vee \cdots \vee x_n \\ F(110\ldots0) = 0 &\quad\to\quad \neg x_1 \vee \neg x_2 \vee x_3 \vee \ldots \vee x_n \\ &\quad\vdots \end{aligned}$$

Die *Konjunktion* dieser Klauseln gibt eine Formel, die F darstellt.

DNF: Alle $(b_1, \ldots, b_n) \in \{0,1\}^n$ für die $F(b_1, \ldots, b_n) = 1$ Klauseln analog zu oben:

$$\begin{aligned} F(0\ldots0) = 1 &\quad\to\quad \neg x_1 \wedge \neg x_2 \wedge \ldots \wedge \neg x_n \\ F(110\ldots0) = 1 &\quad\to\quad x_1 \wedge x_2 \wedge \neg x_3 \wedge \ldots \wedge \neg x_n \\ &\quad\vdots \end{aligned}$$

Die *Disjunktion* dieser Klauseln gibt eine Formel, die F darstellt.

Beachte:

- Erfüllbarkeitsproblem bei KNF $\iff$ *Jede* Klausel muss ein wahres Literal haben.
- Erfüllbarkeitsproblem bei DNF $\iff$ Es gibt *mindestens 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 ist die folgende Frage: Gibt es eine Belegung, so dass 0 rauskommt? Das ist nun wieder leicht bei KNF. (Wieso?)

Eine weitere Einschränkung ist die k-KNF bzw. k-DNF, $k = 1, 2, 3, \ldots$. Hier ist die Klauselgröße auf $\leq k$ Literale pro Klausel beschränkt.

Beispiel 12.2:

$$\begin{array}{ll} 1\text{-}KNF: & x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \ldots \\ 2\text{-}KNF: & (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_1 \vee \neg x_1) \wedge \ldots \\ 3\text{-}KNF: & (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_3) \wedge x_1 \wedge \ldots \end{array}$$

Die Klausel $(x_1 \vee \neg x_1)$ *oben ist eine tautologische Klausel. Solche Klauseln sind immer wahr und bei KNF eigentlich unnötig.*

Eine unerfüllbare 1*-KNF ist*

$$(x_1) \wedge (\neg x_1),$$

eine unerfüllbare 2*-KNF ist*

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2).$$

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

> Ist $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$ eine injektive Abbildung, dann ist diese Abbildung auch surjektiv.

Dazu nehmen wir n^2 viele Variablen. Diese stellen wir uns so vor:

$$\begin{array}{cccccc} x_{1,1}, & x_{1,2}, & x_{1,3}, & \ldots & x_{1,n} \\ x_{2,1}, & x_{2,2}, & \ldots & & x_{2,n} \\ \vdots & & & & \vdots \\ x_{n,1}, & & \ldots & & x_{n,n} \end{array}$$

Jede Belegung a der Variablen entspricht einer Menge von geordneten Paaren M:

$$a(x_{i,j}) = 1 \Longleftrightarrow (i, j) \in M.$$

Eine Abbilding ist eine spezielle Menge von Paaren. Wir bekommen eine widersprüchliche Formel nach folgendem Prinzip:

1) a stellt eine Abbildung dar **und**

2) a ist eine injektive Abbildung **und**

3) a ist *keine* surjektive Abbildung. (Im Widerspruch zum Satz oben!)

Zu 1):

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

Das Element 1 auf der linken Seite ist höchstens einem Element zugeordnet.

$$(\neg x_{1,1} \vee \neg x_{1,2}) \wedge (\neg x_{1,1} \vee \neg x_{1,3}) \wedge \ldots \wedge (\neg x_{1,2} \vee \neg x_{1,3}) \wedge \ldots \wedge (\neg x_{1,n-1} \vee \neg x_{1,n})$$

Das ergibt $\binom{n}{2}$ Klauseln. (Sobald zum Beispiel $f : 1 \to 2$ und $f : 1 \to 3$ gilt, ist dieser Teil der Formel falsch!)

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

Zu 2): Die 1 auf der rechten Seite wird höchstens von einem Element getroffen. Ebenso für $2, \ldots, n$.

$$(\neg x_{1,1} \vee \neg x_{2,1}) \wedge (\neg x_{1,1} \vee \neg x_{3,1}) \wedge (\neg x_{1,1} \vee \neg x_{4,1}) \wedge \ldots \wedge (\neg x_{1,1} \vee \neg x_{n,1})$$

Haben jetzt: a ist eine injektive Abbildung.

Zu 3):

$$\left.\begin{array}{l}\wedge \\ \Big((\neg x_{1,1} \wedge \neg x_{2,1} \wedge \neg x_{3,1} \wedge \ldots \wedge \neg x_{n,1}) \\ \vee \\ (\neg x_{1,2} \wedge \neg x_{2,2} \wedge \ldots \wedge \neg x_{n,2}) \\ \vdots \\ \vee \\ (\neg x_{1,n} \wedge \ldots \wedge \neg x_{n,n})\Big)\end{array}\right\} \begin{array}{l}a \text{ nicht surjektiv. Das heißt: 1 wir nicht} \\ \text{getroffen oder 2 wird nicht getroffen} \\ \text{oder 3 usw.}\end{array}$$

Die so zusammengesetzte Formel ist unerfüllbar. Der zugehörige Beweis ist sehr lang und soll hier keine weitere Rolle spielen.

Kommen wir zurück zum Erfüllbarkeitsproblem bei KNF-Formeln.

12.2.1 Davis-Putnam-Prozedur

Im Falle des Erfüllungsproblems für KNF lässt sich der Backtracking-Algorithmus etwas verbessern. Dazu eine Bezeichnung:

$$F_{|x=1} \Rightarrow x \text{ auf 1 setzen und } F \text{ vereinfachen.}$$

Das führt dazu, dass Klauseln mit x gelöscht werden, da sie wahr sind. In Klauseln, in denen $\neg x$ vorkommt, kann das $\neg x$ entfernt werden, da es falsch ist und die Klausel nicht wahr machen kann. Analog für $F_{|x=0}$.

Es gilt:

$$F \text{ erfüllbar } \iff F_{|x=1} \text{ oder } F_{|x=0} \text{ erfüllbar.}$$

Der Backtracking-Algorithmus für KNF führt zur *Davis-Putman-Prozedur*, die so aussieht:

Algorithmus 26: Davis-Putnam-Prozedur DP(F)

Input : Formel F in KNF mit Variablen $x_1, \ldots x_n$
Output : „erfüllbar“ mit erfüllender Belegung in a, falls F erfüllbar ist, sonst „unerüllbar“.
Data : Array $a[1, \ldots, n]$ für die Belegung

```
/* F = ∅, leere Formel ohne Klausel */
```
if *F ist offensichtlich wahr* **then return** *„erfüllbar“*;
```
/* {} ∈ F, enthält leere Klausel oder die Klauseln x und ¬x. */
```
if *F ist offensichtlich falsch* **then return** *„unerfüllbar“*;
Wähle eine Variable x von F gemäß einer Heuristik;
Wähle $(b_1, b_2) = (1, 0)$ oder $(b_1, b_2) = (0, 1)$;
$H = F_{|x=b_1}$;
$a[x] = b_1$;
if *DP(H)==„erfüllbar“* **then**
  return *„erfüllbar“*;
else
  $H = F_{|x=b_2}$;
  $a[x] = b_2$;
  return DP(H);
end

Liefert DP(F) „erfüllbar“, so ist in a eine erfüllende Belegung gespeichert.

Korrektheit: Induktiv über die #Variablen in F, wobei das a noch einzubauen ist.

12.2.2 Heuristiken

In die einfache Davis-Putman-Prozedur wurden noch die folgednen Heuristiken eingebaut. Man wählt in den Schritten sieben und acht, die nächste Variable geschickter aus.

pure literal rule: Wenn es in F ein Literal x gibt, so dass $\neg x$ *nicht* in F vorkommt, dann setze $x = 1$ und mache ohne Backtracking weiter. Analog für $\neg x$.

unit clause rule: Wenn es in F eine Einerklausel (x) gibt, dann setze $x = 1$ und mache ohne Backtracking weiter. Analog für $\neg x$.

Erst danach geht die normale Davis-Putman-Algorithmus weiter.

Zur *Korrektheit:*

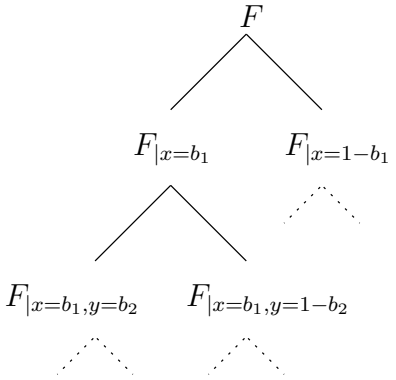
pure literal: Es ist $F_{|x=1} \subseteq F$, das heißt jede Klausel in $F_{|x=1}$ tritt auch in F auf. Damit gilt:

$$\begin{aligned} F_{|x=1} \text{ erfüllbar} &\Rightarrow F \text{ erfüllbar und} \\ F_{|x=1} \text{ unerfüllbar} &\Rightarrow F \text{ unerfüllbar (wegen } F_{|x=1} \subseteq F). \end{aligned}$$

Also ist $F_{|x=1}$ erfüllbar $\Longleftrightarrow$ F erfüllbar. Backtracking ist hier nicht nötig.

unit clause: $F_{|x=0}$ ist offensichtlich unerfüllbar, wegen leerer Klausel. Also ist kein backtracking nötig.

Laufzeit: Prozedurbaum



$$\begin{aligned} b = 0 &\Leftrightarrow 1 - b = 1 \\ b = 1 &\Leftrightarrow 1 - b = 0 \end{aligned}$$

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

$$\begin{aligned} T(1) &= 2 \\ T(n) &\leq T(n-1) + T(n-1) \quad \text{für } n \gneq 1 \end{aligned}$$

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

$$\begin{aligned} T(1) &= 2 \\ T(n) &= T(n-1) + T(n-1) \quad \text{für } n \gneq 1 \end{aligned}$$

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

Eine allgemeine Methode läuft so: Machen wir den Ansatz (das heißt eine Annahme), dass

$$T(n) = \alpha^n \text{ für ein } \alpha \geq 1$$

ist. Dann muss für $n > 1$

$$\alpha^n = \alpha^{n-1} + \alpha^{n-1} = 2 \cdot \alpha^{n-1}$$

sein. Also teilen wir durch α^{n-1} und erhalten

$$\alpha = 1 + 1 = 2.$$

Diese Annahme muss durch Induktion verifiziert werden, da der Ansatz nicht unbedingt stimmen muss.

Damit haben wir die Laufzeit $O(2^n \cdot |F|)$. $|F|$ ist die 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-KNFs für ein festes k lässt sich der Davis-Putman-Ansatz jedoch verbessern. Interessant ist, dass in gewissem Sinne $k = 3$ reicht, wenn man sich auf die Erfüllbarkeit von Formeln beschränkt.

12.2.3 Erfüllbarkeitsäquivalente 3-KNF

Satz 12.1: *Sei F eine beliebige aussagenlogische Formel. Wir können zu F eine 3-KNF G konstruieren mit:*

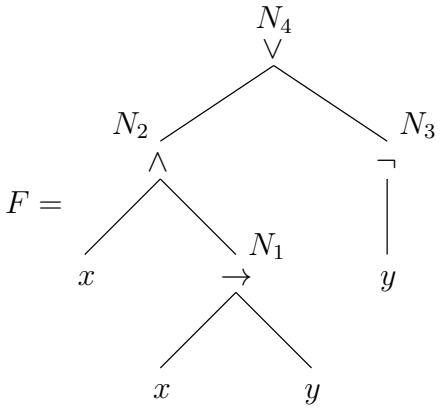
- *F erfüllbar $\Longleftrightarrow$ G erfüllbar. Man sagt: F und G sind erfüllbarkeitsäquivalent.*
- *Die Konstruktion lässt sich in der Zeit $O(|F|)$ implementieren. Insbesondere gilt $|G| = O(|F|)$.*

Beachte: Durch „Ausmultiplizieren“ lässt sich jedes F in ein *äquivalentes* G in KNF transformieren. Aber exponentielle Vergrößerung ist möglich und es entsteht keine 3-KNF.

Beweis. Am Beispiel wird eigentlich alles klar. Der Trick besteht in der Einfürhung neuer Variablen.

$$F = \big(x \wedge (x \to y) \big) \vee \neg y$$

Schreiben F als Baum: Variablen $\to$ Blätter, Operatoren $\to$ innere Knoten.



N_1, N_2, N_3, N_4 sind die neuen Variablen. Für jeden inneren Knoten eine.

Die Idee ist, dass $N_i \Leftrightarrow$ Wert an den Knoten, bei gegebener Belegung der alten Variablen x und y.

Das drückt F' aus:

$$F' = \big(N_1 \Leftrightarrow (x \to y)\big) \wedge \big(N_2 \Leftrightarrow (x \wedge N_1)\big) \wedge \big(N_3 \Leftrightarrow \neg y\big) \wedge \big(N_4 \Leftrightarrow (N_3 \vee N_2)\big)$$

Die Formel F' ist immer erfüllbar. Wir brauchen nur die N_i von unten nach oben passend zu setzen. Aber

$$F'' = F' \wedge N_4, \quad N_4 \text{ Variable der Wurzel,}$$

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, dass die $b(x)$, $b(y)$ eine erfüllende Belegug von F sind.

Eine 3-KNF zu F'' ist gemäß Prinzip auf Seite 182 zu bekommen, indem man es auf die einzelnen Äquivalenzen anwendet. □

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

Davis-Putman basiert auf dem Prinzip: Verzweigen nach dem Wert von x.

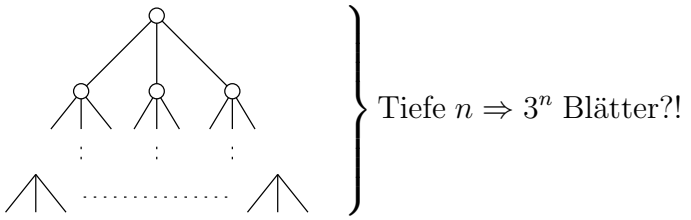
$$F \text{ erfüllbar } \Longleftrightarrow F_{|x=1} \text{ oder } F_{|x=0} \text{ erfüllbar.}$$

Haben wir eine 3-KNF, etwa $F = \ldots \wedge (x \vee \neg y \vee z) \wedge \ldots$, dann ist

$$F \text{ erfüllbar} \Longleftrightarrow F_{|x=1} \text{ oder } F_{|y=0} \text{ oder } F_{|z=1} \text{ erfüllbar.}$$

Wir können bei k-KNF Formeln auch nach den Klauseln verzweigen.

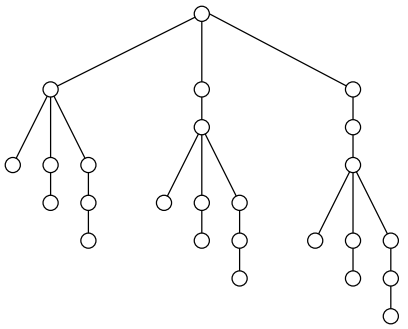
Ein Backtracking-Algorithmus nach diesem Muster führt zu einer Aufrufstruktur der Art:



Aber es gilt auch: (für die obige Beispielklausel)

F erfüllbar $\iff F_{|x=1}$ erfüllbar oder $F_{|x=0,y=0}$ erfüllbar oder $F_{|x=0,y=1,z=1}$ erfüllbar.

Damit ergibt sich die folgende Aufrufstruktur im Backtracking:



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

$$\begin{aligned} T(n) &\leq d = O(1) \quad \text{für } n < n_0 \\ T(n) &\leq T(n-1) + T(n-2) + T(n-3) \quad \text{für } n \geq n_0,\ n_0 \text{ eine Konstante} \end{aligned}$$

Was ergibt das? Wir lösen $T(n) = T(n-1) + T(n-2) + T(n-3)$.

Ansatz:

$$T(n) = c \cdot \alpha^n \quad \text{für alle } n \text{ groß genug.}$$

Dann

$$c \cdot \alpha^n = c \cdot \alpha^{n-1} + c \cdot \alpha^{n-2} + c \cdot \alpha^{n-3},$$

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 eine geeignete Konstante d, da $\alpha > 0$. Sei $n \geq n_0$.

Dann:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + T(n-3) \\ &\leq d \cdot \alpha^{n-1} + d \cdot \alpha^{n-2} + d \cdot \alpha^{n-3} \\ &= d(\alpha^{n-3} \cdot \underbrace{(\alpha^2 + \alpha + 1)}_{=\alpha^3 \text{ nach Wahl von } \alpha}) \\ &= d \cdot \alpha^n = O(\alpha^n). \end{aligned}$$

Genauso $T(n) = \Omega(\alpha^n)$, da die 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 $\varepsilon > 0$ konstant (klein) und k konstant (groß)

$$n^k \cdot c^n \leq c^{(1+\varepsilon)n},$$

sofern n groß genug ist. Denn

$$c^{\varepsilon \cdot n} \geq c^{\log_c n \cdot k} = n^k$$

für $\varepsilon > 0$ konstant und n groß genug (früher schon gezeigt).

Haben wir jetzt zwei Algorithmen

- A_1 Zeit 2^n
- A_2 Zeit $2^{n/2} = \left(\sqrt{2}\right)^n = (1.4\ldots)^n$

für dasselbe Problem. Betrachten wir eine vorgegebene Zeit x (fest). Mit A_1 können wir alle Eingaben der Größe n mit $2^n \leq x$, das heißt $n \leq \log_2 x$ in Zeit x sicher bearbeiten.

Mit A_2 dagegen alle mit $2^{n/2} \leq x$, das heißt $n \leq 2 \cdot \log_2 x$, 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 $\frac{1}{10} \cdot 2^n \leq x$, das heißt

$$n \leq log_2 10x = \log_2 x + log_2 10$$

nur die Addition einer Konstanten.

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

Aufgabe: Führen Sie die Betrachtung des schnelleren Rechners für polynomielle Laufzeiten n^k durch.

12.2.4 Monien-Speckenmeyer

Haben wir das Literal x *pure* in F (das heißt $\neg x$ ist nicht dabei), so reicht es, $F_{|x=1}$ 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. Falls nun gilt

$$H = F_{|x_1=b_1, x_2=b_2, \ldots, x_k=b_k} \subseteq F,$$

das heißt jede Klausel C von H ist schon in F enthalten, dann gilt

$$F \text{ erfüllbar} \iff H \text{ erfüllbar}.$$

Oder anders: Für jede Setzung $x_i = b_i$, die eine Klausel verkleinert, gibt es ein $x_j = b_j$, das diese Klausel wahr macht. Das ist wie beim Korrektheitsbeweis der *pure literal rule*. In diesem Fall ist kein Backtracking nötig.

Wir sagen, die Belegung $x_1 = b_1, \ldots, x_k = b_k$ ist *autark* für F. Etwa

$$F = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge G, \quad G \text{ ohne } x_1, x_2, x_3.$$

Dann $F_{|x_1=0, x_2=0, x_3=0} \subseteq F$, also ist $x_1 = 0, x_2 = 0, x_3 = 0$ autark für F. In G fallen höchstens Klauseln weg.

Algorithmus 27: MoSP(F)

Input : F in KNF.
Output : „erfüllbar“ / „unerfüllbar“

```
1  if F offensichtlich wahr then return „erfüllbar“;
2  if F offensichtlich falsch then return „unerfüllbar“;
3  C = (l_1 ∨ l_2 ∨ ··· ∨ l_s), eine kleinste Klausel in F;      /* l_i Literal */
   /* Betrachte die Belegungen */
4  b_1 = (l_1 = 1);                                            /* nur l_1 gesetzt */
5  b_2 = (l_1 = 0, l_2 = 1);                                   /* nur l_1 und l_2 gesetzt */
6  ...;
7  b_s = (l_1 = 0, l_2 = 0, ..., l_s = 1);
8  if eine der Belegungen autark in F then
9  |  b = eine autarke Belegung;
10 |  return MoSp(F_|b);
11 else
   |  /* Hier ist keine der Belegungen autark. */
12 |  foreach b_i do
13 |  |  if MoSp(F_|b_i) == „erfüllbar“ then return „erfüllbar“;
14 |  end
   |  /* Mit keiner der Belegungen erfüllbar. */
15 |  return „unerfüllbar“;
16 end
```

Wir beschränken uns bei der Analyse der Laufzeit auf den 3-KNF-Fall. Sei wieder

$$\begin{aligned} T(n) &= \#\text{Blätter, wenn in der kleinsten Klausel 3 Literale sind, und} \\ T'(n) &= \#\text{Blätter bei kleinster Klausel mit} \leq 2 \text{ Literalen.} \end{aligned}$$

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, der Algorithmus nimmt immer die kleinste Klausel.

Ebenso bekommen wir

$T'(n) \leq T(n-1)$ wenn autarke Belegung.
$T'(n) \leq T'(n-1) + T'(n-2)$ Haben $\leq$ 2er Klausel ohne autarke Belegung.

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

$$\begin{aligned} T(n) &= d \quad \text{für } n \leq n_0,\ d \text{ Konstante} \\ T(n) &= \max\{T(n-1),\ T'(n-1) + T'(n-2) + T'(n-3)\} \quad \text{für } n > n_0 \\ T'(n) &= d \quad \text{für } n \leq n_0,\ d \text{ Konstante} \\ T'(n) &= \max\{T(n-1),\ T'(n-1) + T'(n-2)\} \quad \text{für } n > n_0. \end{aligned}$$

Zunächst müssen wir das Maximum loswerden. Dazu zeigen wir, dass für

$$\begin{aligned} S(n) &= d \quad \text{für } n \leq n_0, \\ S(n) &= S'(n-1) + S'(n-2) + S'(n-3) \quad \text{für } n > n_0, \\ S'(n) &= d \quad \text{für } n \leq n_0, \\ S'(n) &= S'(n-1) + S'(n-2) \quad \text{für } n > n_0 \end{aligned}$$

gilt, dass $T(n) \leq S(n)$, $T'(n) \leq S'(n)$.

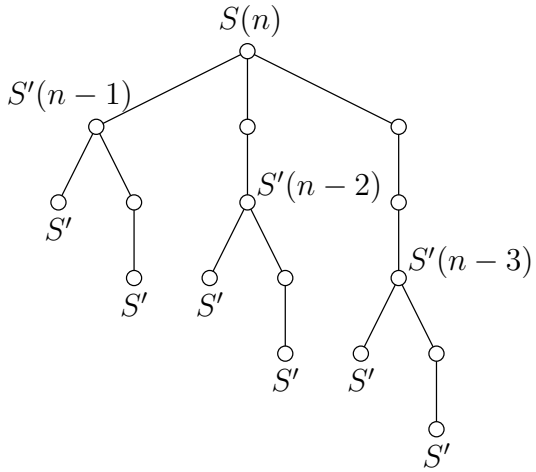
Induktionsanfang: ✓

Induktionsschluss:

$$\begin{aligned} T(n) &= \max\{T'(n-1) + T'(n-2) + T'(n-3)\} \\ &\quad \text{(nach Induktionsvoraussetzung)} \\ &\leq \max\{S(n-1),\ S'(n-1) + S'(n-2) + S'(n-3)\} \\ &\quad \text{(Monotonie)} \\ &\leq S'(n-1) + S'(n-2) + S'(n-3)\} = S(n). \end{aligned}$$

$$\begin{aligned} T'(n) &= \max\{T(n-1),\ T'(n-1) + T'(n-2)\} \\ &\quad \text{(nach Induktionsvoraussetzung)} \\ &\leq \max\{S(n-1),\ S'(n-1) + S'(n-2)\} \\ &= \max\{\overbrace{S'(n-2) + S'(n-3)}^{=S'(n-1)} + S'(n-4),\ S'(n-1) + S'(n-2)\} \\ &\quad \text{(Monotonie)} \\ &= S'(n-1) + S'(n-2) = S'(n). \end{aligned}$$

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



Ansatz: $S'(n) = \alpha^n$, $\alpha^n = \alpha^{n-1} + \alpha^{n-2} \Longrightarrow \alpha^2 = \alpha + 1$.

Sei $\alpha > 0$ Lösung der Gleichung, dann $S'(n) = O(\alpha^n)$.

Induktionsanfanfang: $n \leq n_0$ ✓, da $\alpha > 0$.

Induktionsschluss:

$$\begin{aligned} S'(n+1) &= S'(n) + S'(n-1) \\ &\quad \text{(nach Induktionsvoraussetzung)} \\ &\leq c \cdot \alpha^n + c \cdot \alpha^{n-1} \\ &= c \cdot \alpha^{n-1}(\alpha + 1) \\ &\quad \text{(nach Wahl von } \alpha) \\ &= c \cdot \alpha^{n-1} \cdot \alpha^2 \\ &= c \cdot \alpha^{n+1} = O(\alpha^{n+1}). \end{aligned}$$

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

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

Definition 12.2: *Die Zahlen* $(F_n)_{n\geq 0}$ *mit*

$$\begin{aligned} F_0 &= 1, \\ F_1 &= 1, \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

für alle $n \geq 2$ *heißen Fibonacci-Zahlen. Es ist* $F_n = O(1.681^n)$.

12.2.5 Max-SAT

Verwandt mit dem Erfüllbarkeitsproblem ist das Max-SAT- und das Max-k-SAT-Problem. Dort hat man als Eingabe eine Formel $F = C_1 \wedge \ldots \wedge C_m$ in KNF und sucht eine Belegung, so dass

$$\big|\{i \mid a(C_i) = 1\}\big|$$

maximal ist. Für das Max-Ek-SAT-Problem, d.h. jede Klausel besteht aus genau k verschiedenen Literalen, hat man folgenden Zusammenhang.

Satz 12.2: *Sei* $F = C_1 \wedge \ldots \wedge C_m$ *eine Formel gemäß Max-Ek-SAT. (*$C_i = C_j$ *ist zugelassen.) Es gibt eine Belegung* a*, so dass*

$$\big|\{j \mid a(C_j) = 1\}\big| \geq \left(1 - \frac{1}{2^k}\right) \cdot m.$$

Beweis. Eine feste Klausel $C \subseteq \{C_1, \ldots, C_m\}$ wird von wievielen Belegungen *falsch* gemacht?

$$2^{n-k}, \quad n = \#\text{Variablen},$$

denn die k Literale der Klausel müssen alle auf 0 stehen. Wir haben wir folgende Situation vorliegen:

	a_1	a_2	a_3	$\ldots$	a_{2^n}
C_1	$*$		$*$		$*$
C_2		$*$			
$\vdots$			$\vdots$		
C_m		$*$		$*$	$*$

$* \Leftrightarrow a_i(C_j) = 1$, das heißt Klausel C_j wird unter der Belegung a_i wahr.
Pro Zeile gibt es $\geq 2^n - 2^{n-k} = 2^n(1 - 1/2^k)$ viele Sternchen.

Zeilenweise Summation ergibt:

$$m \cdot 2^n \cdot \left(1 - \frac{1}{2^k}\right) \leq \sum_{j=1}^{m} \left|\{a_i \mid a_i(C_j) = 1\}\right| = \#\text{ der } * \text{ insgesamt.}$$

Spaltenweise Summation ergibt:

$$\begin{aligned} \sum_{i=1}^{2^n} \left|\{j \mid a_i(C_j) = 1\}\right| &= \#\text{ der } * \text{ insgesamt} \\ &\geq 2^n \cdot m \cdot \left(1 - \frac{1}{2^k}\right) \end{aligned}$$

Da wir 2^n Summanden haben, muss es ein a_i geben mit

$$\left|\{j \mid a_i(C_j) = 1\}\right| \geq m \cdot \left(1 - \frac{1}{2^k}\right).$$

□

Wie findet man eine Belegung, so dass $\left|\{j \mid a(C_j) = 1\}\right|$ 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 zwei ungleiche Brüder:

1. 2-SAT: polynomiale Zeit
2. Max-2-SAT: nur exponentielle Algorithmen vermutet.

Vergleiche bei Graphen mit Kantenlängen ≥ 0: kürzester Weg polynomial, längster kreisfreier Weg nur exponentiell bekannt.

Ebenfalls *Zweifärbbarkeit* ist polynomiell, für 3-Färbbarkeit sind nur exponentielle Algorithmen bekannt.

12.3 Maximaler und minimaler Schnitt

Ein ähnliches Phäomen liefert das Problem des maximalen Schnittes:

Für einen Graphen $G = (V, E)$ ist ein Paar (S_1, S_2) mit $(S_1 \dot\cup S_2 = V)$ 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 umgekehrt.

Es gilt: G ist 2-färbbar $\Longleftrightarrow$ 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

$$\big|\{e \in E \mid e \text{ liegt in } (S_1, S_2)\}\big|$$

maximal ist. Wie beim Max-Ek-SAT gilt:

Satz 12.3: *Zu $G = (V, E)$ gibt es einen Schnitt (S_1, S_2) so, dass*

$$\big|\{e \in E \mid e \textit{ liegt in } (S_1, S_2)\}\big| \geq \frac{|E|}{2}.$$

Beweis: Übungsaufgabe.

Algorithmus 28: $|E|/2$-Schnitt

Input : $G = (V, E)$, $V = \{1, \ldots, n\}$
Output : Findet den Schnitt, in dem $\geq |E|/2$ Kanten liegen.

```
1 S_1 = ∅;
2 S_2 = ∅;
3 for v = 1 to n do
4 |   if v hat mehr direkte Nachbarn in S_2 als in S_1 then
5 |   |   S_1 = S_1 ∪ {v};
6 |   else
7 |   |   S_2 = S_2 ∪ {v};
8 |   end
9 end
```

Laufzeit: $O(n^2)$, S_1 und S_2 als boolesches 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

$$\big|\{e \in E \mid e \text{ liegt in } (S_1, S_2)\}\big|$$

minimal ist. Dieses Problem löst *Ford-Fulkerson* in polynomialer Zeit.

Frage: Wie? 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 Salesmen Problem*)

12.4 Traveling Salesman Problem

Definition 12.3(TSP, Problem des Handlungsreisenden): *Wir betrachten einen gerichteten, gewichteten Graphen $G = (V, E)$ mit $K : V^2 \to \mathbb{R}$. $K(\{u, v\})$ sind die Kosten der Kante, ∞ falls $\{u, v\} \notin E$.*

Wir suchen eine Rundreise *(einfacher Kreis) mit folgenden Kriterien:*

- *Die Rundreise enthält alle Knoten und*
- *die Summe der Kosten der betretenen Kanten ist* minimal.

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

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

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?

Ein erster Versuch. Wir halten Knoten 1 fest und zählen alle $(n-1)!$ Permutationen der Knoten $\{2, \ldots, n\}$ auf. Für jede, der sich so ergebenden möglichen Rundreise, ermitteln wir die Kosten.

Laufzeit:

$$\begin{aligned} \Omega((n-1)! \cdot n) = \Omega(n!) &\ge \left(\frac{n}{2}\right)^{\frac{n}{2}} \\ &= 2^{((\log n)-1)\cdot\frac{n}{2}} \\ &= 2^{\frac{(\log n)-1}{2}\cdot n} \\ &= 2^{\Omega(\log n)\cdot n} \\ &\gg 2^n \end{aligned}$$

Beachte:

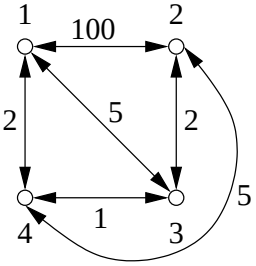
$$n^n = 2^{(\log n)\cdot n}, \quad n! \ge 2^{\frac{(\log n)-1}{2}\cdot n}, \quad n! \ll n^n$$

Zweiter Versuch. Wir versuchen es mit dem Greedy-Prinzip.

Gehe zum jeweils nächsten Knoten, der noch nicht vorkommt (analog Prim). Im Beispiel mit Startknoten 1 bekommen wir $R = (1, 2, 3, 4, 1)$, $K(R) = 39$, was nicht optimal ist. Wählen wir hingegen 4 als Startknoten, ergibt sich $R' = (4, 2, 1, 3, 4)$, $K(R') = 35$ optimal.

Aber: Das muss nicht immer klappen!

Beispiel 12.3: *Betrachten wir einmal den folgenden Graphen. Mit Greedy geht es zwingend in die Irre.*

$G =$ 

mit greedy gewählte Rundreisen:

$R_2 = (2, 3, 4, 1, 2)$ *Kosten* $= 105$
$R_3 = (3, 4, 1, 2, 3)$ *Kosten* $= 105$
$R_4 = (4, 3, 2, 1, 4)$ *Kosten* $= 105$
$R_1 = (1, 4, 3, 2, 1)$ *Kosten* $= 105$

Immer ist $1 \overset{100}{\longleftrightarrow} 2$ *dabei. Optimal ist* $R = (1, 4, 2, 3, 1)$ *mit Kosten von* 14. *Hier ist* $4 \to 2$ *nicht greedy, sondern mit Voraussicht gewählt. Tatsächlich sind für das 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 die Kante (v, w) oder eben nicht.

Gehen wir auf unser Beispiel von Seite 198 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 $2 \to 3$, dann gilt:

- Keine Kante $u \to 3$, $u \neq 2$ muss noch betrachtet werden,
- keine Kante $2 \to v$, $v \neq 3$ muss noch betrachtet werden und
- die Kante $3 \to 2$ kann auch nicht mehr vorkommen.

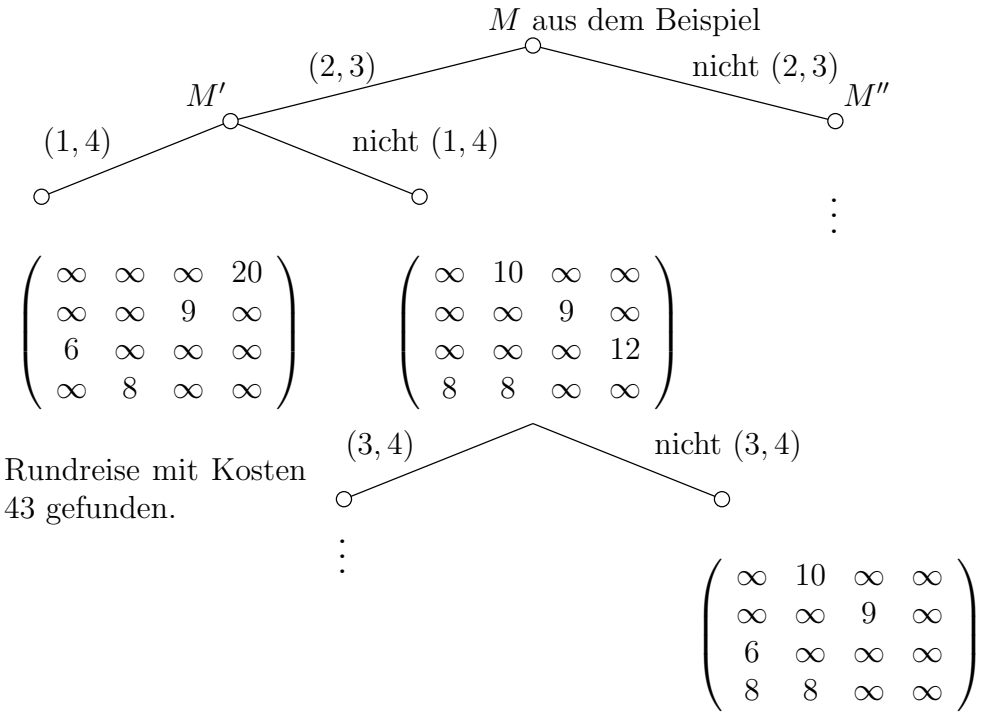
Das heißt, wir suchen eine optimale Reise in M'. Alle nicht mehr benötigten Kanten stehen auf ∞.

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

Ist dagegen $2 \to 3$ nicht gewählt, dann sieht M'' so aus. Nur $M(2,3)$ wird zu ∞.

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

Für das Beispiel ergibt sich dann folgender Bachtrackingbaum.



Bemerkung: Wir haben eine Rundreise gefunden, wenn in *jeder Zeile* und in *jeder Spalte* noch *genau ein* Wert $\neq \infty$ steht und die n verbleibenden Kanten *einen einzigen* Kreis bilden. Falls in einer Zeile oder einer Spalte *nur noch* ∞ stehen, dann ist offensichtlich keine Rundreise mehr möglich.

Algorithmus 29: Backtracking für TSP

Input : Distanzmatrix M
Output : Minimale Rundreise in M, dargestellt als modifikation von M.

if *M stellt Rundreise dar* **then return** $(M, K(M))$;
if *M hat keine Rundreise* $< \infty$ **then**
  return (M, ∞); `/* z.B. Zeile/Spalte voller ∞ */`
end
Wähle eine Kante (u, v), $u \neq v$ mit $M(u, v) \neq \infty$, wobei in der Zeile von u oder der Spalte von v mindestens ein Wert $\neq \infty$ ist;
$M' = M$ modifiziert, so dass $u \to v$ gewählt; `/* (vgl. Seite 199) */`
$M'' = M$ modifiziert, so dass $M(u, v) = \infty$;
$(M_1, K_1) = \text{TSP}(M')$;
$(M_2, K_2) = \text{TSP}(M'')$;
if $K_1 \leq K_2$ **then** `/* Kleinere Rundreise zurückgeben. */`
  return (M_1, K_1);
else
  return (M_2, K_2);
end

Korrektheit: Induktion über die #Einträge in M, die den Wert ∞ haben. Solche Einträge kommen in jedem Schritt dazu.

Die Laufzeit ist zunächst einmal $O(2^{n(n-1)})$, $2^{n(n-1)} = 2^{\Omega(n^2)} \gg n^n$.

12.4.1 Branch-and-Bound

Um die Laufzeit von Backtracking-Algorithmen zu verbessern, versucht man im Backtrackingbaum ganze Zweige, die für die Lösung nicht mehr relevant sind, abzuschneiden. Das führt in der Praxis oft zu einer Beschleunigung. Im worst-case kann man aber nicht viel sagen.

Wir betrachten weiterhin das Problem des Handlungstreisenden. Welche Zweige im obigen Backtracking-Algorithmus können wir gefahrlos abschneiden?

Bei Optimierungsproblemen sind das die Zweige, die mit Sicherheit keine bessere Lösung als die bereits gefundenen Lösungen liefen *können*. Das Problem ist nur, wie soll man die erkennen?

Dazu definiert man eine *untere* bzw. *obere* Schranke für die ab dem betrachteten Punkt noch zu erwartenden Lösungen. Je nachdem, ob es sich um ein *Maximierungs*- oder ein *Minimierungs*-Problem handelt.

Für das TSP wählen wir also eine *untere Schranke* $S(M)$. Zu einer Matrix M – das ist ein Knoten im Backtrackingbaum – ist $S(M)$ *kleiner oder gleich* der Kosten *jeder* Rundreise, die in M möglist ist. Damit gilt auch $S(M) \leq$ Kosten einer minimalen Rundreise.

Wenn wir ein solches $S(M)$ haben, können wir das folgende Prinzip während des Backtrackings anwenden.

> Haben wir eine Rundreise der Kosten K gefunden, dann brauchen wir keine Auswertung (Expansion) der Kosten an Knoten mit $S(M) \geq K$ mehr durchführen.

Zunächst einmal zwei „offensichtliche“ Schranken.

$$S_1(M) = \text{ minimale Kosten einer Rundreise unter } M$$

Das ist die *beste*, das heißt größte Schranke, aber nicht polynomial zu ermitteln. Dazu müssten wir schließlich das Problem selbst lösen.

$$S_2(M) = \text{ Summe der Kosten der bei } M \text{ gewählten Kanten}$$

Ist eine untere Schranke bei $M(u, v) \geq 0$. Das können wir aber hier im Unterschied zu den kürzesten Wegen annehmen. (Wieso?)

Ist M die Wurzel des Backtrackingbaumes, dann ist im Allgemeinen $S_2(M) = 0$, sofern nicht die Zeile $(\infty\, \infty\, \infty\, m\, \infty\, \infty)$ oder analoges für eine Spalte auftritt. S_2 ist auch einfach zu berechnen, das ist schonmal ein guter Anfang.

Die folgenden Schranken sind etwas aufwendiger zu berechnen, liefern dafür aber auch bessere Ergebnisse.

$$S_3(M) = \frac{1}{2} \cdot \sum_v \min\left\{M(u,v) + M(v,w) \mid u,w \in V\right\}$$

Das ist jedesmal das Minimum, um durch den Knoten v zu gehen. ($u \overset{\min}{\to} v \overset{\min}{\to} w$) Wieso ist $S_3(M)$ eine untere Schranke?

Sei $R = (1, v_1, \ldots, v_{n-1}, 1)$ eine Rundreise unter M. Dann gilt:

$$\begin{aligned}
K(R) &= M(1,v_1) + M(v_1,v_2) + \cdots + M(v_{n-1},1) \\
&\quad \text{(In der Summe alles verdoppelt.)} \\
&= \frac{1}{2}\Big(M(1,v_1) + M(1,v_1) + M(v_1,v_2) + M(v_1,v_2) + \cdots + \\
&\qquad + M(v_{n-1},1) + M(v_{n-1},1)\Big) \\
&\quad \text{(Shift um 1 nach rechts.)} \\
&= \frac{1}{2}\Big(M(v_{n-1},\underbrace{1}_{v=1}) + M(\underbrace{1}_{v=1},v_1) + M(1,\underbrace{v_1}_{v=v_1}) + M(\underbrace{v_1}_{v=v_1},v_2) + \\
&\qquad + M(v_1,\underbrace{v_2}_{v=v_2}) + M(\underbrace{v_2}_{v=v_2},v_3) + \ldots \\
&\qquad + M(v_{n-2},\underbrace{v_{n-1}}_{v=v_{n-1}}) + M(\underbrace{v_{n-1}}_{v=v_{n-1}},1)\Big) \\
&\geq \frac{1}{2} \cdot \sum_v \min\left\{M(u,v) + M(v,w) \mid u,w \in V\right\}
\end{aligned}$$

Also ist $S_3(M)$ eine korrekte untere Schranke für eine Rundreise in M.

Es ist $S_3(M) = \infty \Longleftrightarrow$ Eine Zeile oder Spalte voller ∞. Ebenso für $S_2(M)$.

Betrachten wir M von Seite 198 Wir ermitteln $S_2(M)$.

$$\begin{aligned}
v = 1 &\Rightarrow 2 \xrightarrow{5} 1 \xrightarrow{10} 2 && \text{Das ist in einer Rundreise nicht möglich,} \\
& && \text{bei der Schranke aber egal.} \\
v = 2 &\Rightarrow 4 \xrightarrow{8} 2 \xrightarrow{5} 1 \\
v = 3 &\Rightarrow 2 \xrightarrow{9} 3 \xrightarrow{6} 1 && \text{oder} \quad 4 \xrightarrow{9} 3 \xrightarrow{6} 1 \\
v = 4 &\Rightarrow 2 \xrightarrow{10} 4 \xrightarrow{8} 1 && \text{oder} \quad 2 \xrightarrow{10} 4 \xrightarrow{8} 2
\end{aligned}$$

Also $S_3(M) = \frac{1}{2} \cdot 61 = 30,5$. Wegen ganzzahliger Kosten sogar $S_3(M) = 31$. Somit gilt, dass $K(R) \geq 31$ für jede Rundreise R.

Beispiel 12.4: *Ist* $2 \to 3$ *bei* M *gewählt, dann*

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

und

$$\begin{array}{lcl} v=1 & \Rightarrow & 3 \xrightarrow{6} 1 \xrightarrow{10} 2 \\ v=2 & \Rightarrow & 4 \xrightarrow{8} 2 \xrightarrow{9} 3 \\ v=3 & \Rightarrow & 2 \xrightarrow{9} 3 \xrightarrow{6} 1 \\ v=4 & \Rightarrow & 3 \xrightarrow{12} 4 \xrightarrow{8} 2 \quad \textit{oder} \quad 3 \xrightarrow{12} 4 \xrightarrow{8} 1. \end{array}$$

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

Kommen wir nun zur letzten Schranke. Betrachten wir eine allgemeine Matrix $M = \big(M(u,v)\big)_{1 \le u,v \le n}$. Alle Einträge $M(u,v) \ge 0$.

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_i einen geeigneten Wert aus Zeile i der Matrix darstellt. Alternativ ist

$$K(R) = s_1 + s_2 + \cdots + s_n,$$

wobei s_j einen geeigneten Wert aus Spalte j darstellt. Dann gilt

$$\begin{array}{rcl} S'_4(M) & = & \min\big\{\overbrace{M(1,1), M(1,2), \ldots, M(1,n)}^{\text{Zeile 1}}\big\} + \\ & & \min\big\{\overbrace{M(2,1), M(2,2), \ldots, M(2,n)}^{\text{Zeile 2}}\big\} + \\ & & \cdots + \\ & & \min\big\{\overbrace{M(n,1), M(n,2), \ldots, M(n,n)}^{\text{Zeile } n}\big\} \end{array}$$

nimmt aus jeder Zeile den kleinsten Wert. $S'_4(M)$ ist eine korrekte Schranke.

Wenn jetzt noch nicht alle Spalten vorkommen, können wir noch besser werden. Und zwar so:

Sei also für $1 \le u \le n$ $\quad M_u = \min\big\{M(u,1), M(u,2), \ldots, M(u,n)\big\}$ der kleinste Wert der Zeile u. Wir setzen

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

Alles ≥ 0. Pro Zeile ist mindestens ein Eintrag 0.

Es gilt für jede Rundreise R zu M,

$$K_M(R) = K_{\widehat{M}}(R) + S_4'(M) \geq S_4'(M).$$

mit K_M = Kosten in M und $K_{\widehat{M}}$ = Kosten in $\widehat{M}$, in $\widehat{M}$ alles ≥ 0.

Ist in $\widehat{M}$ auch 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 $\widehat{M}_u$.

Ist also $S_u = \min\{\overbrace{\widehat{M}(1,u), \widehat{M}(2,u), \ldots, \widehat{M}(n,u)}^{\text{Spalte } u \text{ von } \widehat{M}}\}$, dann

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

Alles ≥ 0. Pro Spalte eine 0, pro Zeile eine 0.

Für jede Rundreise R durch $\widehat{M}$ gilt

$$K_{\widehat{M}}(R) = K_{\widehat{\widehat{M}}}(R) + S_1 + \cdots + S_n.$$

Also haben wir insgesamt:

$$\begin{aligned} K_M(R) &= K_{\widehat{M}}(R) + M_1 + \cdots + M_n \\ &= K_{\widehat{\widehat{M}}}(R) + \overbrace{S_1 + \cdots + S_n}^{\text{aus } \widehat{M}} + \underbrace{M_1 + \cdots + M_n}_{\text{aus } M} \\ &\quad (\text{In } \widehat{\widehat{M}} \text{ alles} \geq 0) \\ &\geq M_1 + \cdots + M_n + S_1 + \cdots + S_n. \end{aligned}$$

Wir definieren jetzt offiziell:

$$S_4(M) = M_1 + \cdots + M_n + S_1 + \ldots S_n$$

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.

Beispiel 12.5: *In unserem Eingangsbeispiel von Seite 198 ist*

$$M = \begin{pmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{pmatrix} \quad \begin{matrix} M_1 = 10 \\ M_2 = 5 \\ M_3 = 6 \\ M_4 = 8 \end{matrix}$$

$$\widehat{M} = \begin{pmatrix} \infty & 0 & 5 & 10 \\ 0 & \infty & 4 & 5 \\ 0 & 7 & \infty & 6 \\ 0 & 0 & 1 & \infty \end{pmatrix} \quad \begin{matrix} S_1 = 0 \\ S_2 = 0 \\ S_3 = 1 \\ S_4 = 5 \end{matrix}$$

$$\widehat{\widehat{M}} = \begin{pmatrix} \infty & 0 & 4 & 5 \\ 0 & \infty & 3 & 0 \\ 0 & 7 & \infty & 1 \\ 0 & 0 & 0 & \infty \end{pmatrix} \quad S_4(M) = 35 \; > \; S_3(M) = 31$$

Die Schranken lassen sich für jede Durchlaufreihenfolge des Backtracking-Baumes zum Herausschneiden weiterer Teile verwenden.

Zusammenfassend noch einmal das prinzipielle Vorgehen.

Algorithmus 30: Offizielles branch-and-bound

1. Die Front des aktuellen Teils des Backtrackingbaumes steht im Heap, geordnet nach $S(M)$ ($S(M)$ = die verwendete Schranke);
2. Immer an dem M mit $S(M)$ minimal weitermachen;
3. Minimale gefundene Rundreise merken;
4. Anhalten, wenn minimales $S(M)$ im Heap $\geq$ Kosten der minimalen bisher gefundenen Rundreise;

Beispiel 12.6: *Zum Abschluß noch das Beispiel von Seite 198 mit branch-and-bound und der Schranke S_4.*

Hier noch eine etwas andere Darstellung. Wir geben nur noch die Suchfront Q an. In Q sind alle Matrizen, die noch eine Rundreise enthalten können. Den Test, ob wir eine Rundreise haben, machen wir beim Herausnehmen der M aus Q.

1. *Am Anfang:*

$$M = \begin{pmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{pmatrix} \quad S_4(M) = 35$$

$$Q = \big[(M, 35)\big]$$

2. Kante $2 \to 1$*:*

$$M_{(2,1)} = \begin{pmatrix} \infty & \infty & 15 & 20 \\ 5 & \infty & \infty & \infty \\ \infty & 13 & \infty & 12 \\ \infty & 8 & 9 & \infty \end{pmatrix} \quad S_4(M_{(2,1)}) = 40$$

$$M_{\neg(2,1)} = \begin{pmatrix} \infty & 10 & 15 & 20 \\ \infty & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{pmatrix} \quad \begin{array}{l} S_4(M_{\neg(2,1)}) = 34 < S_4(M) = 35 \\ \textit{Wir nehmen weiterhin } 35 \textit{ als Schranke,} \\ \textit{da } M_{\neg(2,1)} \textit{ aus } M \textit{ entstanden ist.} \end{array}$$

$$Q = \left[(M_{\neg(2,1)}, 35), (M_{(2,1)}, 40)\right]$$

3. Kante $(3 \to 1)$*:*

$$M_{\neg(2,1),(3,1)} = \begin{pmatrix} \infty & 10 & \infty & 20 \\ \infty & \infty & 9 & 10 \\ 6 & \infty & \infty & \infty \\ \infty & 8 & 9 & \infty \end{pmatrix} \quad \begin{array}{l} S_4(M_{\neg(2,1),(3,1)}) = 34 = 35 \\ \textit{(siehe oben)} \end{array}$$

$$M_{\neg(2,1),\neg(3,1)} = \begin{pmatrix} \infty & 10 & 15 & 20 \\ \infty & \infty & 9 & 10 \\ \infty & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{pmatrix} \quad S_4(M_{\neg(2,1),\neg(3,1)}) = 39$$

$$Q = \left[(M_{\neg(2,1),(3,1)}, 35), (M_{\neg(2,1),\neg(3,1)}, 39), (M_{(2,1)}, 40)\right]$$

4. Kante $(4 \to 2)$*:*

$$M_{\neg(2,1),(3,1),(4,2)} = \begin{pmatrix} \infty & \infty & \infty & 20 \\ \infty & \infty & 9 & \infty \\ 6 & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty \end{pmatrix} \quad S_4(M_{\neg(2,1),(3,1),(4,2)}) = 43$$

$$M_{\neg(2,1),(3,1),\neg(4,2)} = \begin{pmatrix} \infty & 10 & \infty & 20 \\ \infty & \infty & 9 & 10 \\ 6 & \infty & \infty & \infty \\ \infty & \infty & 9 & \infty \end{pmatrix} \quad S_4(M_{\neg(2,1),(3,1),\neg(4,2)}) = 35$$

$$Q = \left[(M_{\neg(2,1),(3,1),\neg(4,2)}, 35), (M_{\neg(2,1),\neg(3,1)}, 39), (M_{(2,1)}, 40), (M_{\neg(2,1),(3,1),(4,2)}, 43)\right]$$

$M_{\neg(2,1),(3,1),(4,2)}$ *ist bereits eine Rundreise.* $(1 \to 4 \to 2 \to 3 \to 1)$ *Wir fügen sie trotzdem in den Heap ein. Wenn es noch Matrizen gibt, die bessere Rundreisen liefern können, werden die vorher bearbeitet.*

5. *Kante* $(2 \to 4)$*:*

$$M_{\neg(2,1),(3,1),\neg(4,2),(2,4)} = \begin{pmatrix} \infty & 10 & \infty & \infty \\ \infty & \infty & \infty & 10 \\ 6 & \infty & \infty & \infty \\ \infty & \infty & 9 & \infty \end{pmatrix} \quad S_4(M_{\neg(2,1),(3,1),\neg(4,2),(2,4)}) = 35$$

$$M_{\neg(2,1),(3,1),(4,2),\neg(2,4)} = \begin{pmatrix} \infty & 10 & \infty & 20 \\ \infty & \infty & 9 & \infty \\ 6 & \infty & \infty & \infty \\ \infty & \infty & 9 & \infty \end{pmatrix} \quad S_4(M_{\neg(2,1),(3,1),(4,2),\neg(2,4)}) = 44$$

$$Q = \big[(M_{\neg(2,1),(3,1),\neg(4,2),(2,4)}, 35), (M_{\neg(2,1),\neg(3,1)}, 39), (M_{(2,1)}, 40), (M_{\neg(2,1),\neg(3,1),(4,2)}, 43), (M_{\neg(2,1),(3,1),(4,2),\neg(2,4)}, 44)\big]$$

6. *Als nächstes ist* $M_{\neg(2,1),(3,1),\neg(4,2),(2,4)}$ *an der Reihe. Das ist eine fertige Rundreise mit Kosten* 35*. Alle anderen* unteren Schranken *sind größer. Die verbleibenden Matrizen im Heap können also keine bessere Rundreise mehr liefern.*

   Damit haben wir mit

$$(1 \to 2 \to 4 \to 3 \to 1)$$

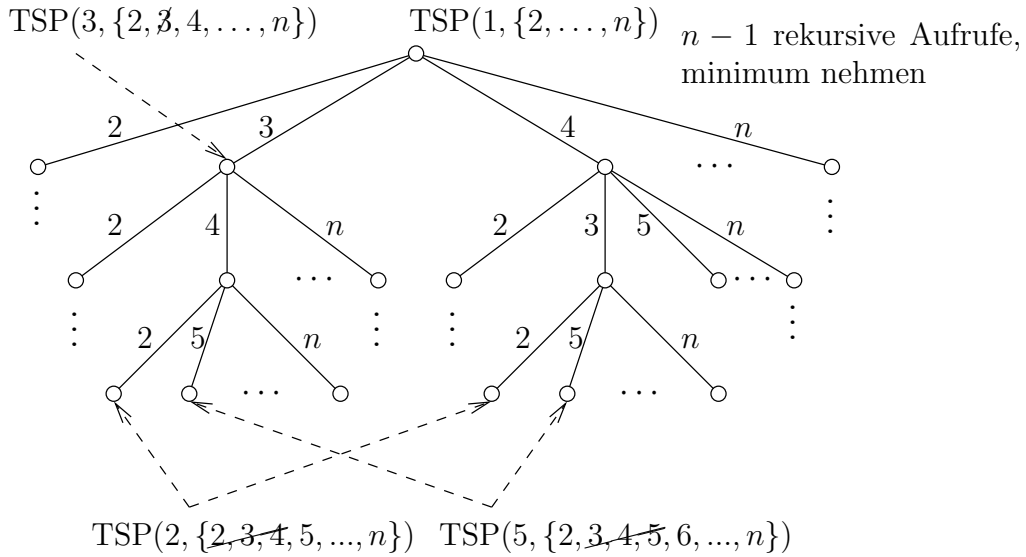
   eine minimale Rundreise im M *gefunden und sind fertig.*

Aufgabe: Zeigen Sie für M und M', wobei M' unter M im Backtrackingbaum hängt Baum, dass $S_2(M') \geq S_2(M), S_3(M') \geq S_3(M)$. (Für S_4 gilt das, wie oben gesehen, so nicht.)

12.4.2 TSP mit dynamischer Programmierung

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:

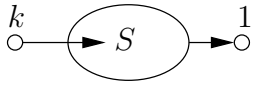


Die Tiefe des Baumes ist $n-1$, wir haben $(n-1)(n-2)\ldots 1 = (n-1)!$ Blätter. Wir bekommen Aufrufe der Art $\mathrm{TSP}(k, \underbrace{\{i_1, \ldots, i_l\}}_{\subseteq\{2,\ldots,n\}})$.

Wieviele verschiedene rekursive Aufrufe $\mathrm{TSP}(k, S)$ gibt es prinzipiell?

- Wähle k: n Möglichkeiten
- Wähle S: $\leq 2^{n-1}$ Möglichkeiten
- $2^{n-1} \cdot n = 2^{n-1+\log n} = 2^{O(n)}$, wogegen $(n-1)! = 2^{\overbrace{\Omega(\log n) \cdot n}^{\gg n}}$

Wir tabellieren wieder die Ergebnisse der rekursiven Aufrufe, wobei $TSP(k, S) =$ kürzeste Reise der Art:



$k \neq 1,\ 1, k \notin S$

Wir suchen einen Weg beginnend beim Knoten k, der über *alle* Knoten in S zum Knoten 1 führt.

- Zunächst $S = \emptyset$: (Zwischen k und 1 liegen *keine* Knoten.)

$$\begin{aligned} TSP(2,\emptyset) &= M(2,1) \qquad M = \big(M(u,v)\big) \text{ Eingangsmatrix} \\ &\vdots \\ TSP(n,\emptyset) &= M(n,1) \end{aligned}$$

- Dann $|S| = 1$:

$$\begin{aligned} TSP(2,\{3\}) &= M(2,3) + TSP(3, \underbrace{S \setminus \{3\}}_{=\emptyset}) \\ &\vdots \\ TSP(2,\{n\}) &= M(2,n) + TSP(n, \underbrace{S \setminus \{n\}}_{=\emptyset}) \\ &\vdots \end{aligned}$$

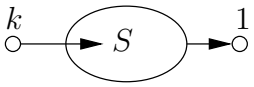
- Dann $|S| = 2$

$$\begin{aligned} TSP(2,\{3,4\}) &= \min\big\{M(2,3) + TSP(3,\{4\}),\ M(2,4) + TSP(4,\{3\})\big\} \\ TSP(2,\{3,5\}) &= \min\big\{M(2,3) + TSP(3,\{5\}),\ M(2,5) + TSP(5,\{3\})\big\} \\ &\vdots \end{aligned}$$

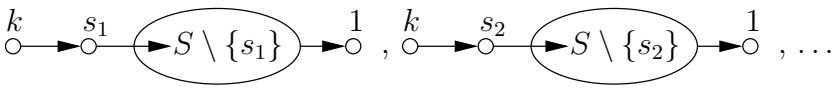
- Allgemein:

$$TSP(k,S) = \min\big\{M(k,s) + TSP(s, S \setminus \{s\}) \mid s \in S\big\}$$

In Worten: Wir suchen jeweils das *minimale Reststück* einer Rundreise. Das Reststück beginnt mit $k \to s$ und durchläuft alle Knoten der Restmenge S. Also bilden wir das Minimum über alle Einstiegspunkte $s \in S$.



ist das Minimum von



für alle Nachbarn $s_i \in S$ von k. Falls k keine Nachbarn in S hat, ∞. Dann gibt es keine Rundreise dieser Form.

Algorithmus 31: TSP mit dynamischer Programmierung

Input : Gerichteter Graph G dargestellt als Distanzmatrix M.
Output : Minimale Rundreise in G, falls eine existiert.
Data : 2-dimensionales Array TSP indiziert mit [1, . . . , n] für den Startpunkt und $[0 = \emptyset, \ldots, 2^{n-1} - 1 = \{2, \ldots, n\}]$, allen möglichem Mengen von Restknoten

```
1  for k = 1 to n do
2  |  TSP(k, ∅) = M(k, 1);
3  end
4  for i = 1 to n − 1 do
   |  /* Alle Teilmengen der Restknoten der Größe nach. */
5  |  foreach S ⊆ {2, . . . , n}, |S| = i do
   |  |  /* Alle Knoten, in denen wir vor betreten der Menge S
   |  |     sein können. */
6  |  |  foreach k ∉ S do
7  |  |  |  TSP(k, S) = min{M(k, s) + TSP(s, S \ {s}) | s ∈ S};
8  |  |  end
9  |  end
10 end
```

Das Ergebnis ist

$$TSP(1, \{2, \ldots, n\}) = \min\big\{M(1, s) + TSP(s, \{2, \ldots, n\} \setminus s) \mid s \in \{2, \ldots, n\}\big\}.$$

Laufzeit: $O(n \cdot 2^n)$ Einträge im Array TSP. Pro Eintrag das Minimum ermitteln, also Zeit $O(n)$. Ergibt insgesammt $O(n^2 \cdot 2^n)$.

Es ist

$$n^2 \cdot 2^n = 2^{n+2\log n} \leq 2^{n(1+\varepsilon)} = 2^{(1+\varepsilon)\cdot n} = \big(2(1+\varepsilon')\big)^n \ll (n-1)!$$

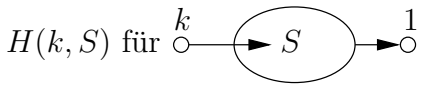
für $\varepsilon, \varepsilon' \geq 0$ geeignet. ($2^\varepsilon \to 1$ für $\varepsilon \to 0$, da $2^0 = 1$.)

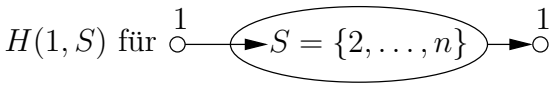
12.5 Hamilton-Kreis und Eulerscher Kreis

Verwandt mit dem Problem des Handlungsreisenden ist das Problem des Hamiltonschen Kreises.

Definition 12.4(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

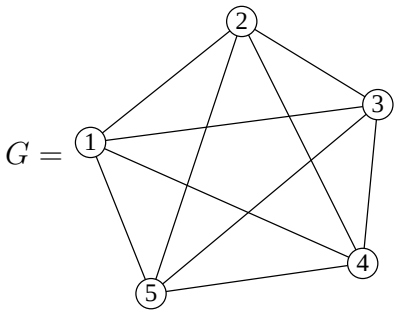




in Zeit $O(n^2 \cdot 2^n)$ lösbar. Besser als $\Omega((n-1)!)$ druch einfaches Backtracking. Geht genauso wie TSP mit Kosten überall $= 1$.

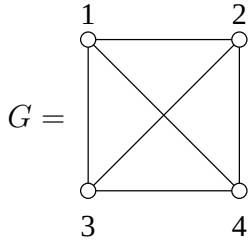
Definition 12.5(Eulerscher Kreis): *Sei $G = (V, E)$ ein ungerichteter Graph. Ein Eulerscher Kreis ist ein geschlossener Pfad, in dem jede Kante genau einmal vorkommt.*

Beispiel 12.7:



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

Beispiel 12.8: *Wie sieht das hier aus?*



$(1, 2, 3, 4, 1, 3, ?)$, $\{2, 4\}$ *fehlt*

$(1, 2, 4, 1, 3, 4, ?)$, $\{3, 4\}$ *fehlt*

Scheint nicht zu gehen. Es fehlt immer eine Kante, außerdem kein geschlossener Pfad möglich?!

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 den Gegensatz

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

Wie findet man jetzt in Polynomialzeit heraus, ob ein Graph einen Eulerschen Kreis entält? Und wie findet man ihn?

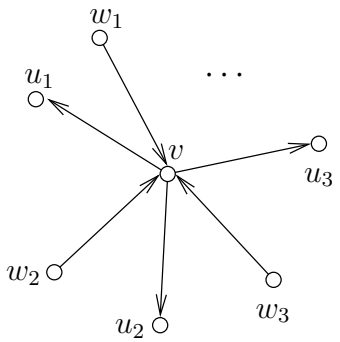
Der Beweis des folgenden Satzes liefert praktischerweise gleich einen Algorithmus dazu.

Satz 12.4: *Sei G ohne Knoten vom Grad 0 (*$\mathrm{Grad}(v)$ *= #direkter Nachbarn von v).*

G hat Eulerschen Kreis $\Longleftrightarrow$ G ist zusammenhängend und $Grad(v)$ ist gerade für alle $v \in V$.

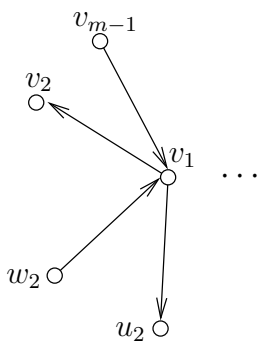
Beweis. „$\Rightarrow$“ Sei also $G = (V, E), |V| = n, |E| = m$. Sei $K = (v_1, v_2, v_3, \ldots, v_m = v_1)$ ein Eulerscher Kreis von G.

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.

„⇐“ Das folgende ist eine Schlüsselbeobachtung für Graphen, die zusammenhängend sind und geraden Grad haben.

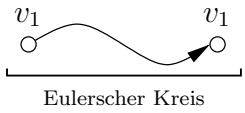
Wir beginnen einen Weg an einem beliebigen Knoten v_1 und benutzen jede vorkommende Kante nur einmal. Irgendwann landen wir wieder an v_1:

$$W = v_1 \to v_2 \to v_3 \to v_4 \to v_5 \to v_6 \to \cdots \to v_1$$

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.

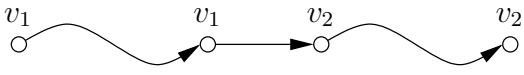
Nehmen wir nun W aus G heraus, haben alle Knoten wieder geraden Grad und wir können mit den Startknoten $v_1, v_2, \ldots$ so weitermachen und am Ende alles zu einem Eulerschen Kreis zusammensetzen. Konkret sieht das so aus:

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 noch von v_1 erreichbar ist. Schreibe diesen Eulerschen Kreis hin. Wir bekommen:



3. Erweitere den Eulerschen Kreis auf W zu

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* liefert.