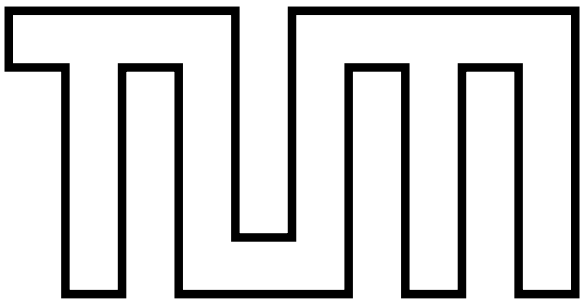


Technische Universität München
Institut für Informatik

Effiziente Algorithmen und Datenstrukturen
WS 1997/98

Stand: 13. März 1998

Dozent:

Prof. Dr. Ernst W. Mayr

Mitschrift:

Martin Stumpf
Thomas Schöpf
Martin Hans
Guido Wimmel

Inhaltsverzeichnis

0 Einleitung und Auffrischung **3**
- 0.1 Was sind „kombinatorische“ Algorithmen? . . . 3
- 0.2 Maschinenmodelle . . . 3
- 0.3 Komplexitätsmaße . . . 3
  - 0.3.1 Berechnungsmodi . . . 3
  - 0.3.2 Probleme und Funktionen . . . 3
  - 0.3.3 Ressourcen . . . 4
  - 0.3.4 Wachstumsverhalten von Funktionen . . . 4
  - 0.3.5 Rekursionsformeln . . . 5
    - 0.3.5.1 Multiplikatoren . . . 5
    - 0.3.5.2 Charakteristisches Polynom . . . 6
    - 0.3.5.3 Erzeugendenfunktionen . . . 6
    - 0.3.5.4 Transformation des Definitions- bzw. Wertebereichs 8

1 Höhere Datenstrukturen **9**
- 1.1 Grundlegende Operationen . . . 9
- 1.2 Suchbäume, balancierte Bäume . . . 10
  - 1.2.1 Höhenbalancierte Bäume . . . 10
    - 1.2.1.1 (a,b)-Bäume . . . 10
    - 1.2.1.2 Rot-Schwarz-Bäume . . . 12
    - 1.2.1.3 AVL-Bäume . . . 14
  - 1.2.2 Gewichtsbalancierte Bäume . . . 14
  - 1.2.3 Hashing . . . 14
- 1.3 Priority Queues . . . 14
  - 1.3.1 Wiederholung . . . 14
    - 1.3.1.1 Binomialbäume . . . 14
    - 1.3.1.2 Binomial Queues (binomial heaps) . . . 15
  - 1.3.2 Fibonacci-Heaps . . . 16
    - 1.3.2.1 Implementierung . . . 17
    - 1.3.2.2 Amortisierte Kostenanalyse für Fibonacci-Heaps . . 18
    - 1.3.2.3 Eine $\mathcal{O}(\log\log N)$ Priority-Queue . . . 19
  - 1.3.3 Sich selbst organisierende Datenstrukturen . . . 22
    - 1.3.3.1 Sich selbst organisierende lineare Listen . . . 22
    - 1.3.3.2 Splay-Trees als Suchbäume . . . 23
      - 1.3.3.2.1 Einleitung . . . 23
      - 1.3.3.2.2 Die Splaying-Operation . . . 24
      - 1.3.3.2.3 Amortisierte Kostenanalyse . . . 25
      - 1.3.3.2.4 Wörterbuchoperationen in Splay-Trees . . 27
  - 1.3.4 Weitere Arten von Datenstrukturen . . . 29

2 Selektion **30**
2.1 Einleitung 30
2.2 Der Blum-Floyd-Pratt-Rivest-Tarjan Selektions-Algorithmus 30
2.3 Der SPP-Algorithmus 32
2.4 Eine untere Schranke für die Medianbestimmung 39
2.5 Eine bessere untere Schranke 42

3 Minimale Spannbäume **46**
3.1 Grundlagen 46
3.2 Kruskals Algorithmus 49
3.3 Prim's Algorithmus, erste Variante 51
3.4 Prim's Algorithmus, zweite Variante 53

4 Kürzeste Pfade **56**
4.1 Grundlegende Begriffe 56
4.2 Dijkstra's Algorithmus 56
4.3 Dikstra's Algorithmus mit Radix-Heaps 58
4.4 Floyd's Algorithmus für das All-pairs-shortest-path-Problem 59
4.5 Min-Plus-Matrix-Produkt und Min-Plus-Transitive Hülle 60
4.6 Boolesche Matrixmuliplikation und Transitive Hülle 61
4.7 Der 4-Russen-Algorithmus für Boolesche Matrixmultiplikation 63
4.8 Transitive Hülle und DFS 65
4.9 Ein besserer Algorithmus für das All-pairs-shortest-distance-Problem 67
4.10 Ein Algorithmus für die transitive Hülle mit lin. erwarteter Laufzeit 69
4.11 Matrizenmultiplikation à la Strassen 72

5 Matchings in Graphen **75**
5.1 Grundlagen 75
5.2 Matchings maximaler Kardinalität in bipartiten Graphen 79

6 Literatur **80**

Kapitel 0

Einleitung und Auffrischung

0.1 Was sind „kombinatorische“ Algorithmen?

Probleme, die, könnten wir alle Fälle aufzählen, trivial wären, aber eine sehr große Anzahl von Fällen haben.
Beispiel: Hamiltonscher Kreis (NP-vollständig), Eulerscher Kreis (P)

0.2 Maschinenmodelle

- Turingmaschine (TM)
- Random Access Machine (RAM)
- Schaltkreise
- (Quantencomputer)
- (DNA-Computer)

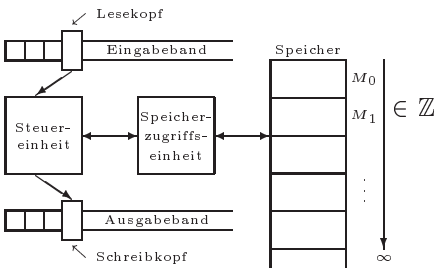


Abb. 1: **Random Access Machine**

0.3 Komplexitätsmaße

Terminologie:
Problem $\equiv$ Sprache $= \{x \in \Sigma^*,\ x$ hat eine bestimmte Eigenschaft$\}$
Maschine M $\left\{ \begin{array}{l} \text{entscheidet/erkennt} \\ \text{akzeptiert} \end{array} \right\}$ eine (formale) Sprache $L \subseteq \Sigma^*$

0.3.1 Berechnungsmodi

a) erkennt: $(\forall x \in \Sigma^*)$ [M hält auf Eingabe x und gibt die richtige Antwort]

b) akzeptiert: $(\forall x \in L)$ [M hält auf x und akzeptiert] und $(\forall x \in \overline{L})$ [M hält auf x und weist zurück oder M hält nicht]

0.3.2 Probleme und Funktionen

- Probleme sind Teilmengen $\subseteq \Sigma^*$
- Berechnung von Funktionen:

Definition: Seien Σ, Γ Alphabete. Eine TM (bzw. RAM) M berechnet eine (partielle) Funktion $f : \Sigma^* \to \Gamma^*$ gdw. $(\forall x \in \Sigma^*)$ [falls M mit Eingabe x hält, dann ist die Ausgabe von M gleich $f(x) \equiv x \in$ Definitionsbereich von f]

0.3.3 Ressourcen

- welche Kosten: Zeit, Platz (Speicher), Schaltkreisgröße, Schaltkreistiefe, Programmgröße
- wie werden die Kosten berechnet: Betrachte Eingaben x der Länge n
  - worst-case Komplexität:

    $$C(n) := \max\{\text{Komplexität}(x);\ |x| = n\}$$

  - durchschnittliche Komplexität (average complexity):

    $$C(n) := \frac{1}{|\Sigma^n|} \sum_{|x|=n} \text{Komplexität}(x)$$

    allgemeiner: Wahrscheinlichkeitsmaß μ

    $$C(n) := \sum_{x \in \Sigma^n} \mu(x) \cdot \text{Komplexität}(x)$$

  - amortisierte Komplexität:
    durchschnittliche Kosten der Operationen in Folgen
    worst-case über alle Folgen
  - probabilistische oder randomisierte Komplexität:
    Algorithmus hat Zufallsbits zur Verfügung. Erwartete Laufzeit (über alle Realisierungen (mit Wahrscheinlichkeit) der Zufallsbits) für feste Eingabe x, dann worst-case für alle $|x| = n$.
- Kosteneinheiten:
  - uniformes Kostenmodell
  - logarithmisches Kostenmodell: Kosten $\approx$ Länge der Operanden

Beispiel: $f(n) := 2^{2^n}$ (1 Schritt pro Quadrierung $\Rightarrow n$ Schritte)

0.3.4 Wachstumsverhalten von Funktionen

Groß-$\mathcal{O}$-Notation (Bachmann/Landau)
$f(n) = 3n^2 + n$ (Asymptotische Komplexität)
$f_2(n) = 5n^2 - 3n$

Definition: f, g seien Funktionen von $\mathbb{N}$ nach $\mathbb{R}$

- $g(n)$ ist $\mathcal{O}(f(n))$ [auch: g ist $\mathcal{O}(f)$, $g(n) = \mathcal{O}(f(n))$, $g(n) \in \mathcal{O}(f(n))$] gdw. $(\exists C \in \mathbb{R}, \exists n_0 \in \mathbb{N})[(\forall n \geq n_0)[|g(n)| \leq C \cdot |f(n)|]]$
  n^3 ist <u>nicht</u> $\mathcal{O}\left(\frac{n^3}{\log n}\right)$
- $g(n)$ ist $\Omega(f(n))$ [...] gdw. $(\exists C > 0, \exists n_0 \in \mathbb{N})[(\forall n \geq n_0)[|g(n)| \geq C \cdot |f(n)|]]$
- $g(n)$ ist $\Theta(f(n))$ gdw. g ist $\mathcal{O}(f)$ und g ist $\Omega(f)$
- $g(n)$ ist $o(f(n))$ gdw. $(\forall C > 0\ \exists n_0)[(\forall n \geq n_0)[|g(n)| \leq C \cdot |f(n)|]]$
- $g(n)$ ist $\omega(f(n))$ gdw. $(\forall C > 0\ \exists n_0)[(\forall n \geq n_0)[|g(n) \geq C \cdot |f(n)|]]$
- $g(n)$ ist $\Omega_\infty(f(n))$ gdw. $(\exists C > 0)[|g(n)| \geq C \cdot |f(n)|$ für unendlich viele $n]$
  Beispiel: $n^2 \sin\frac{\pi}{2}n = \Omega_\infty(n)$
- $g(n)$ ist $\omega_\infty(f(n))$ gdw. $(\forall C > 0)[|g(n)| \geq C \cdot |f(n)|]$ für unendlich viele $n]$

0.3.5 Rekursionsformeln

Mergesort:

$$\begin{aligned}
\mathrm{T}(n) &= 2\mathrm{T}\left(\frac{n}{2}\right) + cn \\
&= cn + 2\mathrm{T}\left(\frac{n}{2}\right) \\
&= cn + 2\left(c\frac{n}{2} + 2\mathrm{T}\left(\frac{n}{4}\right)\right) \\
&= cn + cn + 4\mathrm{T}\left(\frac{n}{4}\right) \\
&\approx cn\log_2 n \text{ (nur genau für Zweierpotenzen, sonst } \pm 1)
\end{aligned}$$

Allgemein: $\mathrm{C}(n) = \mathrm{F}_1(\mathrm{C}(n^{(1)}), \mathrm{C}(n^{(2)}), \ldots, \mathrm{C}(n^{(r)})) + \mathrm{F}_2(n)$

0.3.5.1 Multiplikatoren

$\mathrm{f}_n = 2\mathrm{f}_{n-1} + n,\ n \geq 2;\ \mathrm{f}_1 = 1$

$$\left.\begin{array}{ll}
\mathrm{f}_n = 2\mathrm{f}_{n-1} + n & \mid \cdot 1 \\
\mathrm{f}_{n-1} = 2\mathrm{f}_{n-2} + n - 1 & \mid \cdot 2 \\
\vdots & \vdots \\
\mathrm{f}_2 = 2\mathrm{f}_1 + 2 & \mid \cdot 2^{n-2} \\
\mathrm{f}_1 = 1 & \mid \cdot 2^{n-1}
\end{array}\right\} \mathrm{f}_n = 2^{n-1}\mathrm{f}_1 + \sum_{i=0}^{n-2} 2^i(n-i) = 2^{n+1} - n - 2$$

Addieren aller Gleichungen:

$$\mathrm{f}_n = 2^{n-1}\mathrm{f}_1 + \sum_{i=0}^{n-2} 2^i(n-i) = \underbrace{2^{n-1}}_{(1)} + \underbrace{n\sum_{i=0}^{n-2} 2^i}_{(2)} - \underbrace{\sum_{i=0}^{n-2} i2^i}_{(3)}$$

(2) (Geometrische Reihe): $n\sum_{i=0}^{n-2} 2^i = n(2^{n-1} - 1)$

(3) (Substitution: $n - 2 = k$ und $2 = x$):

$$\begin{aligned}
\sum_{i=0}^{k} ix^i &= \sum_{i=1}^{k} ix^i \text{(erster Summand ist 0)} \\
&= x\sum_{i=1}^{k} ix^{i-1} \\
&= x\sum_{i=1}^{k} \frac{dx^i}{dx} \\
&= x\frac{d}{dx}\sum_{i=1}^{k} x^i \\
&= x\frac{d}{dx}\frac{x^{k+1} - 1}{x - 1} \\
&= \frac{kx^{k+2} - x^{k+1}(k+1) + x}{(x-1)^2}
\end{aligned}$$

Einsetzen von $k = n-2$ und $x = 2$:

$$\rightsquigarrow \sum_{i=0}^{n-2} i2^i = 2^n(n-2) - 2^{n-1}(n-1) + 2 = 2^n n - 2^{n+1} - 2^{n-1}n + 2^{n-1} + 2$$

(2) − (3):

$$\begin{aligned} n\sum_{i=0}^{n-2} 2^i - \sum_{i=0}^{n-2} i2^i &= n \cdot (2^{n-1} - 1) - 2^n n + 2^{n+1} + 2^{n-1}n - 2^{n-1} - 2 \\ &= 2^{n-1}(2n-1) - 2^n(n-2) - n - 2 \end{aligned}$$

(1) + (2) − (3):

$$\begin{aligned} 2^{n-1} + n\sum_{i=0}^{n-2} 2^i + \sum_{i=0}^{n-2} i2^i &= 2^{n-1} + 2^{n-1}(2n-1) - 2^n(n-2) - n - 2 \\ &= 2^{n+1} - n - 2 \end{aligned}$$

0.3.5.2 Charakteristisches Polynom

Sei

$$(*)\ \mathrm{f}_n = \mathrm{f}_{n-1} + \mathrm{f}_{n-2} \text{ für } n \geq 2,\ \mathrm{f}_1 = 1,\ \mathrm{f}_0 = 0$$

(lineare homogene Rekursionsgleichung zweiter Ordnung)
Setze $\mathrm{f}_n = a^n$ für ein unbekanntes a.
$a^n - a^{n-1} - a^{n-2} = 0$; da hier $a \neq 0$: $a^2 - a - 1 = 0$; $a_{1/2} = \frac{1 \pm \sqrt{5}}{2}$
a_1 und a_2 Lösungen von $(*)$, dann auch $\mathrm{f}_n = c_1 a_1^n + c_2 a_2^n$ eine Lösung für $(*)$.
$\mathrm{f}_1 = 1$ und $\mathrm{f}_0 = 0$ liefern zwei Gleichungen für c_1 und c_2.
Lösen: $\mathrm{f}_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$

Satz: Sei $\mathrm{p}(x)$ das charakteristische Polynom der Rekursionsgleichung

$$(**)\ p_0\mathrm{f}_n + p_1\mathrm{f}_{n-1} + \ldots + p_k\mathrm{f}_{n-k} = 0$$

mit den konstanten Koeffizienten p_i. Seien $r_i,\ i = 1, \ldots, m$ die (i. a. komplexen) Wurzeln von $\mathrm{p}(x)$, jeweils mit Vielfachheit m_i. Dann ist die allgemeine Lösung von $(**)$ gegeben durch

$$\mathrm{f}_n = \sum_{i=1}^{m} \left(r_i^n \sum_{j=0}^{m_i - 1} c_{ij} n^j \right),$$

mit beliebigen Konstanten c_{ij}.

0.3.5.3 Erzeugendenfunktionen

$$(\mathrm{f}_n)_{n \geq 0} \mapsto \mathrm{F}(z) = \sum_{n=0}^{\infty} \mathrm{f}_n z^n; \quad z \in \mathbb{C}$$

Exponentielle Erzeugendenfunktion:

$$(\mathrm{f}_n)_{n \geq 0} \mapsto \sum_{n \geq 0} \frac{\mathrm{f}_n}{n!} z^n$$

auch Notation:

$$\mathrm{F}(z) = \sum_{n>0} \mathrm{f}_n z^n \Rightarrow \mathrm{f}_n = [z^n]\mathrm{F}(z)$$

Sei $\mathrm{F}(z) = \sum_{n>0} \mathrm{f}_n z^n$ und $\mathrm{G}(z) = \sum_{n\geq 0} \mathrm{g}_n z^n$

Erzeugendenfunktion	n-tes Folgenglied	Anmerkungen:
$c\mathrm{F}$	$c\mathrm{f}_n$	
$\mathrm{F}+\mathrm{G}$	$\mathrm{f}_n + \mathrm{g}_n$	
$\mathrm{F}\cdot\mathrm{G}$	$\mathrm{h}_n := \sum_{i=0}^{n} \mathrm{f}_i \mathrm{g}_{n-i}$ (Konvolution)	$\left(\sum_{n\geq 0} \mathrm{f}_n z^n\right)\left(\sum_{n\geq 0} \mathrm{g}_n z^n\right) = \sum_{n\geq 0} \mathrm{h}_n z^n$ (mit $\mathrm{h}_n = \sum_{i=0}^{n} \mathrm{f}_i \mathrm{g}_{n-i}$)
$z^k\mathrm{F}$	if $n<k$ then 0 else f_{n-k} fi	
$\frac{\mathrm{F}(z)}{1-z}$	$\sum_{i=0}^{n} \mathrm{f}_i$	$\frac{1}{1-z} = \sum_{n\geq 0} z^n$
$z\frac{d\mathrm{F}(z)}{dz}$	$n\mathrm{f}_n$	
$\int_0^x \mathrm{F}(t)dt$	if $n=0$ then 0 else $\frac{\mathrm{f}_{n-1}}{n}$ fi	$\mathrm{f}_n z^n$ („geht über auf“) $\mathrm{f}_n \frac{z^{n+1}}{n+1}$
$\mathrm{F}(cz)$	$c^n\mathrm{f}_n$	$\mathrm{F}(cz) = \sum_{n\geq 0} \mathrm{f}_n (cz)^n = \sum_{n\geq 0} \mathrm{f}_n c^n z^n$

Beispiel:

$$\begin{aligned}
\mathrm{F}(z) :=& \sum_{n\geq 0} 2^n z^n = \frac{1}{1-2z} \\
\mathrm{G}(z) :=& \sum_{n\geq 0} n z^n = \frac{z}{(1-z)^2} \\
\Rightarrow \mathrm{F}(z)\mathrm{G}(z) =& \frac{z}{(1-z)^2(1-2z)} = \sum_{n\geq 0}\sum_{i=0}^{n} (n-i)2^i z^n
\end{aligned}$$

Partialbruchzerlegung:

$$\begin{aligned}
\frac{z}{(1-z)^2(1-2z)} &= \overbrace{\frac{-2}{(1-z)}}^{(1)} - \frac{-z}{(1-z)^2} + \overbrace{\frac{2}{1-2z}}^{(2)} \\
&= \underbrace{\sum_{n\geq 0} 2^{n+1} z^n}_{(2)} - \sum_{n\geq 0} n z^n - \underbrace{2\sum_{n\geq 0} z^n}_{(1)};
\end{aligned}$$

Also:

$$\begin{aligned}
\sum_{i=0}^{n} 2^i (n-i) &= [z^n](\mathrm{FG})(z) \\
&= [z^n]\left(\sum_{n\geq 0} (2^{n+1} - n - 2) z^n\right) \\
&= 2^{n+1} - n - 2;
\end{aligned}$$

0.3.5.4 Transformation des Definitions- bzw. Wertebereichs

Beispiel 1:

$f_n = f_{n-1} \cdot f_{n-2}$ für $n \geq 2$;
$f_1 = 2;\ f_0 = 1$

Setze: $g_n = \log f_n$
$\Rightarrow g_n = g_{n-1} + g_{n-2}$ für $n \geq 2$
$g_1 = \log 2 = 1,\ g_0 = 0$ für $\log = \log_2$
$g_n = F_n$ n-te Fibonacci-Zahl
$\Rightarrow f_n = 2^{F_n}$

Beispiel 2:

$f_n = 3f_{\frac{n}{2}} + n$ für $n = 2^k$;
$f_1 = 1$

Setze: $g_k = f_{2^k}$
Dann: $g_k = 3g_{k-1} + 2^k,\ k \geq 1$
$g_0 = 1$
Es ergibt sich: $g_k = 3^{k+1} - 2^{k+1}$, also
$f_n = 3 \cdot 3^k - 2 \cdot 2^k$
$= 3(2^{\log 3})^k - 2 \cdot 2^k$
$= 3(2^k)^{\log 3} - 2 \cdot 2^k$
$= 3n^{\log 3} - 2n$

Kapitel 1

Höhere Datenstrukturen

1.1 Grundlegende Operationen

Menge S mit Ordnung $\leq$. Oft auch S als Schlüsselmenge. Ordnung auf den Schlüsseln $S \ni s \mapsto k(s)$.

1. Element(a, S): Ist $a \in S$?
2. Insert(a, S): $S := S \cup \{a\}$
3. Delete(a, S): $S := S - \{a\}$
   (a) Falls Ordung auf S (bzw. Schlüsseln für S):
   Delete_Min(S): Delete(a, S), wobei a minimal in S (bzw. $k(a)$ minimal)
4. Replace(a, a', S): $S := S - \{a\} \cup \{a'\}$
   (a) Decrease_Key(a, Δ, S): ersetze $a = \underbrace{(k, v)}_{\text{Schlüssel k}(v)}$ durch $a' = (k - \Delta, v)$
5. Union(S_1, S_2): $S_1 := S_1 \cup S_2$; falls $S_1 \cap S_2 = \emptyset$, dann Merge
6. Sei $S_i;\ i = 1, \ldots, r$ eine Ansammlung von disjunkten Mengen.
   Find(a): liefert $i \in \{1, \ldots, r\}$, falls $a \in S_i$, ansonsten undefiniert
7. Find_Min(S): minimales Element in S (falls S geordnet), bzw. Element in S mit minimalem Schüssel; undefiniert, falls $S = \emptyset$
8. Find_Max(S): analog
9. Split(S_i, a, S_j); if $a \in S_i$ then $S_j := \{b \in S_i;\ b > a\},\ S_i := \{b \in S_i;\ b \leq a\}$ (bzw. $k(b) > k(a),\ k(b) \leq k(a)$) else undefiniert fi
10. Concatenate(S_i, S_j): if Find_Max$(S_i) <$ Find_Min(S_j) then $S_i := S_i \cup S_j$ else undefiniert fi

Datenstrukturklasse	mindestens angebotene Funktionen	realisiert in
Wörterbuch (Dictionary)	Element(), Insert(), Delete()	Hashtable, balancierte Bäume
Vorrangwarteschlange (Priority Queue)	Find_Min(), Insert(), Delete(), [Element()]	balancierte, Leftist-, Binomialbäume
Mergeable heaps	Find_Min(), Insert(), Delete(), Merge()	2-3-Bäume, Binomialwälder, Leftist-Bäume
Concatenable queues	Find_Min(), Insert(), Delete(), Concatenate()	2-3-Bäume

1.2 Suchbäume, balancierte Bäume

1.2.1 Höhenbalancierte Bäume

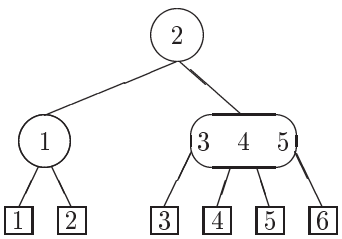
Suchbäume:

- extern: Elemente nur in den Blättern gespeichert
- intern: Elemente in allen Knoten

1.2.1.1 (a,b)-Bäume

Definition: Ein (a,b)-Baum ist ein externer Suchbaum, alle Blätter haben die gleiche Tiefe, die Anzahl der Kinder eines jeden internen Knoten ist $\leq$ b und $\geq$ a (Für die Wurzel ≥ 2), und es gilt $b \geq 2a-1$. Spezialfall: $b = 2a-1$ (B-Bäume (Bayer))

Beispiel:
Wir betrachten im folgenden (2,4)-Bäume.



Interne Knoten: -∞, k_1, ..., k_{d-1}, +∞

Lemma: Sei T ein (a,b)-Baum mit n Blättern und der Höhe h. Dann

$$(1) \quad 2a^{h-1} \leq n \leq b^h$$

$$(2) \quad \log_b n = \frac{\log n}{\log b} \leq h \leq 1 + \log_a \frac{n}{2}$$

Beweis:

(1) $n \leq b^h$: Maximal für vollständigen Baum-Verzweigungsgrad b. Minimal für Baum mit Grad der Wurzel = 2, alle anderen internen Knoten = a.

(2) aus (1).

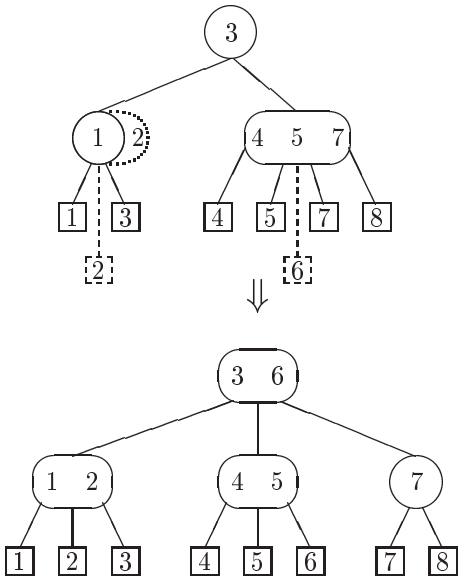
□

Operationen:

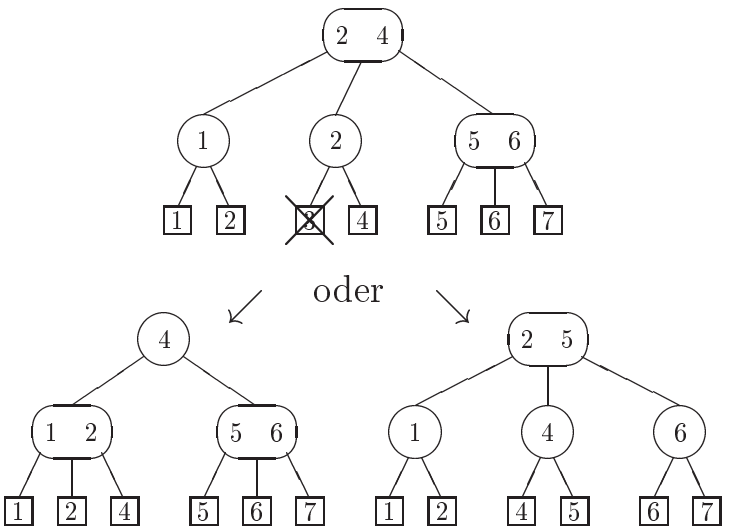
1. Element(x, S)
   $v :=$ Wurzel von T;
   <u>while</u> v ist nicht Blatt <u>do</u>
   $i := \min\{j;\ 1 \le j \le d(v),\ x \le k_j\}$;
   $v := i$-tes Kind von v;
   <u>od</u>;
   <u>if</u> $x = v$ <u>then</u> return (ja) <u>else</u> return (nein) <u>fi</u>;

   Zeitbedarf: $\mathcal{O}(\log n)$

2. Insert(x, S)
   Element, dann neues Blatt, rekursiv nach oben. Splitting von Knoten mit $> b$ Kindern.



3. Delete(x, S)



   Zeitbedarf: $\mathcal{O}(\log n)$

Bemerkungen:

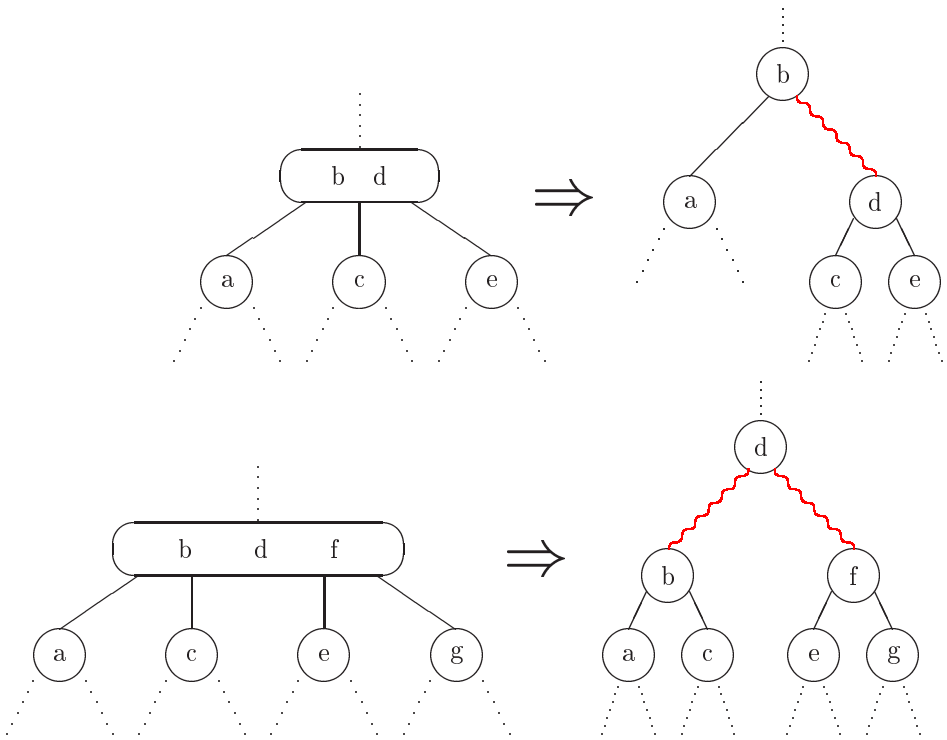
1. Zur Wahl von a:
   - Daten in RAM $\Rightarrow$ a klein, z.B. $a = 2$ oder $a = 3$.
   - Daten auf Platte $\Rightarrow$ a groß, z.B. $a = 100$.
2. Zur Wahl von b:
   $b \geq 2a$ liefert wesentlich bessere amortisierte Komplexität (ohne Beweis, siehe Mehlhorn).
3. Top-Down-Rebalancierung: $b \geq 2a$.
   Bei der Restrukturierung nach der Top-Down-Strategie, folgen wir wie gewohnt dem Pfad von der Wurzel zum gesuchten Blatt. Beim Einfügen stellen wir jetzt jedoch für jeden besuchten Knoten sicher, daß der Knoten weniger als b Kinder hat. Wenn der gerade betrachtete Knoten ein b-Knoten ist, dann spalten wir ihn sofort auf. Weil der Vater kein b-Knoten ist (das haben wir ja bereits sichergestellt) pflanzt sich der Aufspaltungsprozeß nicht nach oben hin fort. Insbesondere kann das neue Element ohne Probleme eingefügt werden, wenn die Suche das Blattniveau erreicht hat. Damit diese Strategie möglich ist, muß b mindestens $2a$ sein (und nicht $2a - 1$), da sonst nach der Spaltung nicht genügend Elemente für die beiden Teile vorhanden sind.
   Beim Löschen verfahren wir analog. Für jeden besuchten Knoten außer der Wurzel des (a, b)-Baumes stellen wir sicher, daß er mindestens $a + 1$ Kinder hat. Wenn der gerade betrachtete Knoten nur a Kinder hat, so versuchen wir zuerst ein Element des rechten oder linken Nachbarknoten zu stehlen. Sind beide Nachbarknoten vom Grad a, so verschmelzen wir unseren Knoten mit einem der beiden Nachbarn. Ist der Vater nicht die Wurzel, so hatte er aber vorher mindestens $a + 1$ Kinder, und dieser Verschmelzungsprozeß kann sich nicht nach oben fortsetzen. Andernfalls erniedrigt sich der Grad der Wurzel um 1, wenn die Wurzel Grad größer 2 hat, oder die alte Wurzel wird gelöscht und der soeben verschmolzene Knoten wird zur neuen Wurzel.
   Restrukturierung nach der Top-Down-Strategie sorgt nicht für eine bessere Laufzeit, sondern erleichtert die Synchronisation, wenn mehrere Prozesse gleichzeitig auf einen (a, b)-Baum zugreifen wollen. Bei herkömmlichen (a, b)-Bäumen schreiten die Such- und die Restrukturierungsoperationen in entgegengesetzten Richtungen fort, was dazu führt, daß sich oft Prozesse gegenseitig behindern (der eine will im Baum absteigen, der andere muß auf dem gleichen Pfad aufwärts restrukturieren). Bei der Top-Down-Strategie gibt es nur Operationsfolgen die den Baum hinabsteigen, mehrere Prozesse können so in einer Art Pipeline gleichzeitig einen Pfad mit kurzem Abstand begehen.

1.2.1.2 Rot-Schwarz-Bäume

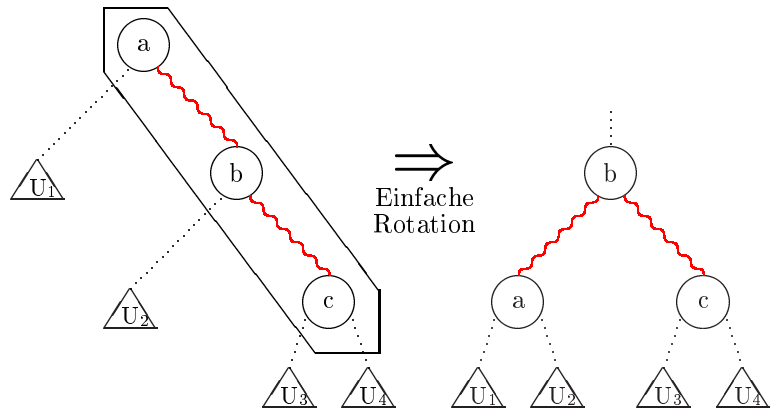
Definition: Rot-Schwarz-Bäume sind Binärbäume (jeder Knoten hat 0 oder 2 Kinder) mit roten und schwarzen Kanten, so daß gilt:

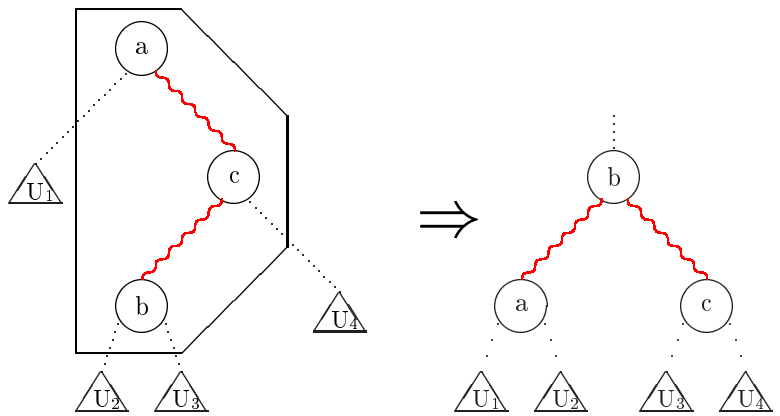
1. alle Blätter hängen an schwarzen Kanten (durchgezogene Linien)
2. alle Blätter haben die gleiche „Schwarztiefe“.
3. kein Pfad von der Wurzel zu einem Blatt enthält aufeinanderfolgende rote Kanten (gewellte Linien).

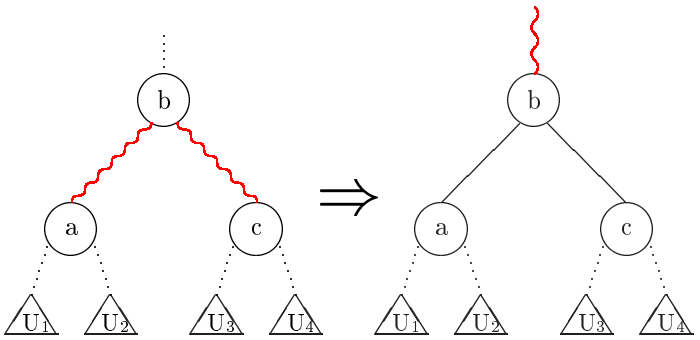
Rot-Schwarz-Bäume können zur Implementierung von (2,4)-Bäumen mittels Binärbäumen dienen.
Vorteil: Interne Knoten haben Verzweigungsgrad = 2



Rebalancierung:







Satz: Die Tiefe eines Rot-Schwarz-Baumes mit n Blättern ist $\mathcal{O}(\log n)$

Beweis: klar □

Korollar: Die Wörterbuch-Operationen benötigen auf Rot-Schwarz-Bäumen $\mathcal{O}(\log n)$ Zeit.

1.2.1.3 AVL-Bäume

Benannt nach Adelson, Velskii und Landis

Siehe

- Aho, Hopcroft, Ullmann
- Sedgewick/Flajolet
- Info IV Skript SS94, Prof. Mayr

In der Theorie: Sehr schöne Eigenschaften.
In der Praxis: Sehr aufwendig zu implementieren.

1.2.2 Gewichtsbalancierte Bäume

Siehe Mehlhorn, Bd. 1, S. 189 ff.

1.2.3 Hashing

Siehe Info IV Skript SS94, Prof. Mayr

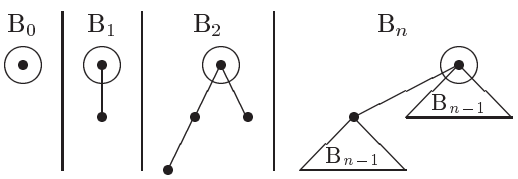
1.3 Priority Queues

Auch: Vorrangwarteschlangen, Heaps.

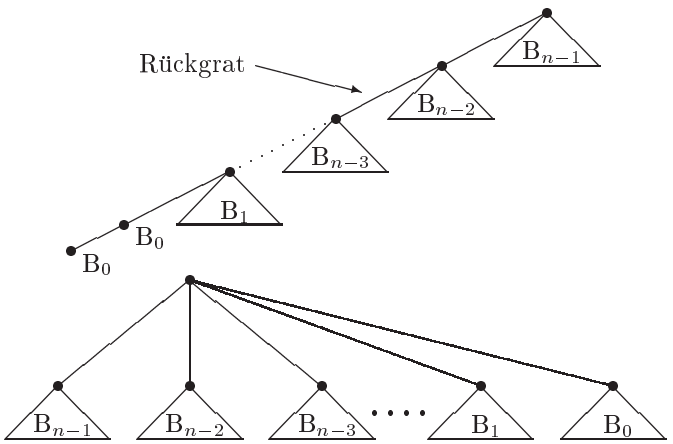
1.3.1 Wiederholung

Binomial-Warteschlangen.

1.3.1.1 Binomialbäume



Zwei (interessante) Dekompositionen:



Eigenschaften:

1. $|B_n| = 2^n \quad [= 2^{n-1} + 2^{n-2} + \ldots + 2 + 1 + 1]$
2. Tiefe$(B_n) = n$
3. Anzahl der Knoten in B_n mit Tiefe $k = \binom{n}{k}$

1.3.1.2 Binomial Queues (binomial heaps)

- Daten(schlüssel) an allen Knoten, aus total geordnetem Universum.
- Heap-Bedingung: $\forall$ Knoten x mit Vater p(x) gilt: Das in p(x) gespeicherte Element ist $\leq$ dem in x gespeicherten (Min. heap).
- Größe n beliebig: Wald von $\leq \lceil \log(n+1) \rceil$ Binomialbäumen [ein Baum für jede 1 in der Binärdarstellung von n]. Doppelt verkettete Liste der Wurzeln, mit Zeiger auf Wurzel mit minimalem Element.
- Implementierung: Pointer für am weitesten links befindliches Kind, rechtes Geschwister und Vater

Operationen:

1. Verschmelzen (Mergen) zweier Binomial Heaps
   $B_i \cup B_i \Rightarrow B_{i+1}$, kleinere Wurzel wird Wurzel des B_{i+1}
   $B_i \cup B_j, i \neq j \Rightarrow$ Wald aus B_i und B_j.

   B_i, B'_j Binomialbäume der Ordnung i

   $F_1 = \{B_{i_1}, B_{i_2}, \ldots, B_{i_r}\} \cup \{B'_{j_1}, B'_{j_2}, \ldots, B'_{j_s}\} = F_2$
   $i_1 < i_2 < \ldots < i_r$
   $j_1 < j_2 < \ldots < j_s$

   Von kleinen zu großen Indizes, entsprechend der Schulmethode für die Addition von Binärzahlen (mit Übertrag), unter Benutzung der obigen Basisschritte.

   Zeitaufwand proportional zur Länge der beiden Listen und der Anzahl der neu geschaffenen Kanten. Die Anzahl der Kanten in einem Binomialwald mit n Knoten ist $n - \#_1(n)$ ($\#_1(n) \equiv$ Anzahl der Einsen in der Binärdarstellung von n).

Die Anzahl der neu geschaffenen Kanten ist daher

$$n_1 + n_2 - \#_1(n_1 + n_2) - (n_1 - \#_1(n_1)) - (n_2 - \#_1(n_2))$$

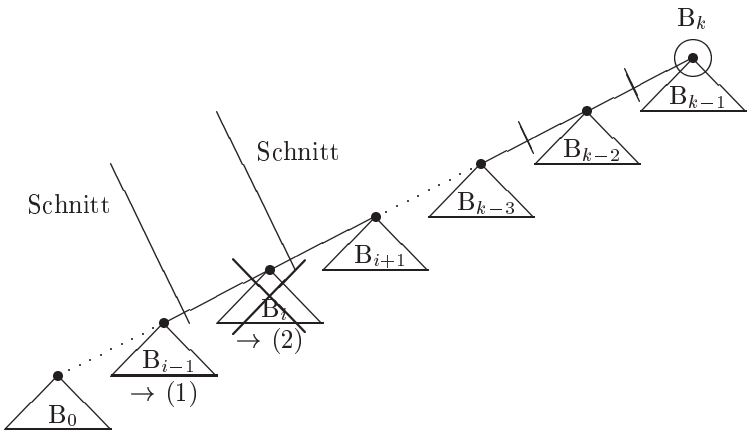
Zeitbedarf: $\mathcal{O}(\log(n_1 + n_2))$, wobei n_1 (bzw. n_2) die Anzahl der Knoten in F_1 (bzw. F_2) ist.

2. Insert(x): $\widehat{=}$ Verschmelzen mit B_0.
   Einfügen von n einzelnen Elementen der Reihe nach: Kosten $\mathcal{O}(n)$

3. Löschen:

   (a) Finde richtigen Baum

   (b)



   $\Rightarrow B_{k-1}, B_{k-2}, \ldots, B_{i+1}$
   Zu (1): $\Rightarrow B_i$
   Zu (2): $\Rightarrow$ Behandle dieses B_i rekursiv, bis das zu löschende Element auf dem Rückgrat liegt. In diesem Fall: lösche die Wurzel von $B_i (= x)$, Liste der Kinder $(B_{i-1}, B_{i-2}, \ldots, B_0)$. Merge all diese Binomialbäume (höchstens einer pro Index) mit dem Rest des Binomialwaldes.

1.3.2 Fibonacci-Heaps

$\equiv$ Priority Queue, (Fredman/Tarjan 1984)

Effiziente Datenstruktur für viele Probleme, z.B. Dijkstra-Algorithmus, minimale Spannbäume.

Operationen:	worst case	amortisiert
Insert	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Merge	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Find_Min	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Decrease_Key	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
Delete	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Delete_Min	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$

Potentialmethode (Potential $\equiv$ Bankguthaben): Folgen von Operationen. Jede Operation hat

1. tatsächliche Kosten (Zeitbedarf)

2. amortisierte Kosten := tatsächliche Kosten $+\Delta$(Bankguthaben)

$\Rightarrow$ Ansparen für „schlechte“ Zeiten.

Datenstruktur:

- Wald von Bäumen, mit Heap-Bedingung, rang(v) eines Knoten v ist Anzahl seiner (direkten) Kinder.
- Wurzeln der Bäume doppelt zyklisch verkettet
- Zeiger auf das kleinste Element (Min_Pointer)
- Kinder eines jeden Knoten doppelt zyklisch verkettet
- jeder Knoten (außer den Wurzeln) hat ein Markierungsbit, das angibt, ob eines seiner Kinder bereits gelöscht wurde.

1.3.2.1 Implementierung

Prinzip ist Binomial Queue mit „Lazy Delete“

1. Insert: Neues Element in Liste der Wurzeln einfügen, dann Update Min_Pointer.
2. Merge: Konkateniere die beiden Wurzellisten, Update Min_Pointer.
3. Find_Min: ...
4. Delete:

   Falls x minimales Element dann Delete_Min, ansonsten

   $v := x$
   while true do
   if $v = x$ then
   lösche Knoten v
   hänge Liste der Kinder von v in die Wurzelliste
   des Fibonacci-Heaps ein
   else
   hänge den Unterbaum mit Wurzel v in die Wurzelliste ein;
   mache v unmarkiert
   fi
   if p(v) existiert then
   $v :=$ p(v)
   else
   return
   fi
   if v nicht markiert and v nicht Wurzel then
   markiere v
   return
   fi
   od

   reelle Kosten für Delete (falls nicht Delete_Min): $\mathcal{O}$(1+ Anzahl der kaskadierenden Schritte)

5. Decrease_Key:

   if an Wurzel then
   verringere Schlüssel an dieser Wurzel;
   update Min_Pointer (falls nötig)
   else
   führe Delete und Insert (mit dem neuen Schlüssel) aus;
   fi

   reelle Kosten: wie Delete

6. Delete_Min:

   lösche minimale Wurzel.
   füge Liste ihrer Kinder in die Wurzelliste ein.
   solange möglich, verbinde je zwei Bäume in der Wurzelliste, deren
   Wurzeln gleichen Rang haben, zu einem Baum (beachte Heap-Bedingung)
   update Min_Pointer.

   reelle Kosten: $\mathcal{O}$(max. Rang + Anzahl der Verbindungs-Operationen)

1.3.2.2 Amortisierte Kostenanalyse für Fibonacci-Heaps

Amortisierte Kosten für Operation := reelle Kosten für Operation + Δ Potential;
Potential := Anzahl der Bäume in der Wurzelliste + $2 \cdot$ Anzahl der markierten Knoten.
Damit ergeben sich folgende amortisierte Kosten:

1. Insert: $\mathcal{O}(1)$
2. Merge: $\mathcal{O}(1)$
3. Find_Min: $\mathcal{O}(1)$
4. Delete: $\mathcal{O}$(max. Rang)
5. Decrease_Key: $\mathcal{O}(1)$, da ein neuer Baum in der Wurzelliste + kaskadierende Schnitte.
6. Delete_Min: $\mathcal{O}$(max. Rang)

Lemma: Sei x ein Knoten im Fibonacci-Heap mit Rang$(x) = k$. Seien $y_1, y_2, \ldots, y_k$ die Kinder von x, in der Reihenfolge, in der sie hinzugefügt wurden. Für $i \geq 2$ gilt: Rang$(y_i) \geq i - 2$.

Beweis: Als y_i hinzugefügt wurde, war Rang$(x) \geq i-1$, also auch y_i. Danach kann es höchstens ein Kind verloren haben. (y_i ist nicht Wurzel). □

Satz: Sei x Wurzel eines Baums im Fibonacci-Heap, Rang$(x) = k$. Dann enthält der Baum mit Wurzel x mind. F_{k+2} Elemente.
($F_i = i$-te Fibonacci-Zahl mit $F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2}$ für $n \geq 2$)

Beweis: Sei f_k die minimale Anzahl der Elemente im Baum, dessen Wurzel Rang k hat.
$\Rightarrow f_k \geq f_{k-2} + f_{k-3} + \ldots + f_0 + 1 + 1$
Setze: $g_k := g_{k-2} + g_{k-3} + \ldots + g_0 + 2$, d. h. $f_k \geq g_k$; $g_0 := 1$

(1)	g_k	$=$	g_{k-2}	$+$	g_{k-3}	$+$	$\cdots$			
(2)	g_{k-1}	$=$			g_{k-3}	$+$	g_{k-4}	$+$	$\cdots$	
(1) - (2)	$g_k - g_{k-1}$	$=$	g_{k-2}							

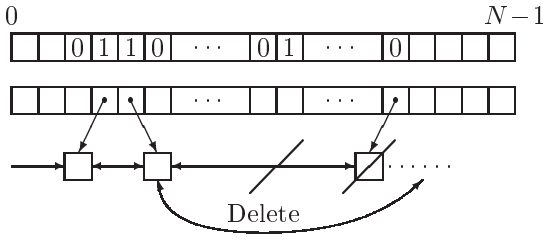
$\Rightarrow g_k = g_{k-1} + g_{k-2} \Rightarrow f_k \geq g_k = F_{k+2}$

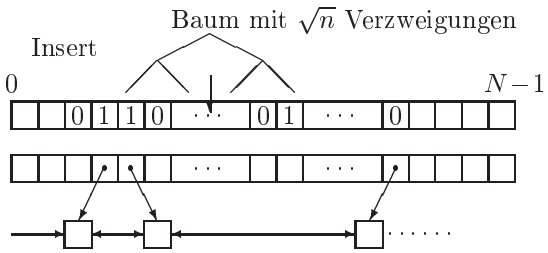
□

Variante: Lazy Delete

1.3.2.3 Eine $\mathcal{O}(\log\log N)$ Priority-Queue

(Peter van Emde Boas)
Universum U, $|U| = N$, $U \subseteq \mathbb{N}$
Hier: $U = \{0, 1, \ldots, N-1\}$
Bessere Priority-Queue für den Fall $n \geq \log N$



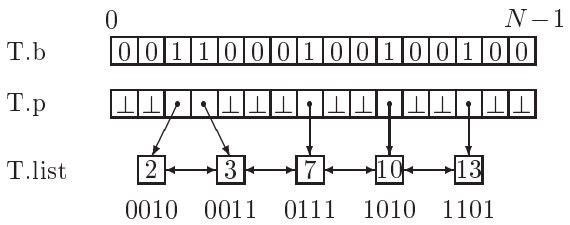


Sei $k \in \mathbb{N}$, $k \geq 2$; $k' = \lceil \frac{k}{2} \rceil$, $k'' = \lfloor \frac{k}{2} \rfloor$
$x \in [0, \ldots, 2^k - 1] \leq k$ Bits
$x' = \left\lfloor \frac{x}{2^{k''}} \right\rfloor$, $x'' := x \mod 2^{k''}$ (x' vordere Hälfte, x'' hintere Hälfte der Bits)
Sei $S = \{x_1, \ldots, x_m\} \subseteq [0, \ldots, 2^k - 1]$

Definition: Eine k-Struktur T für S besteht aus:

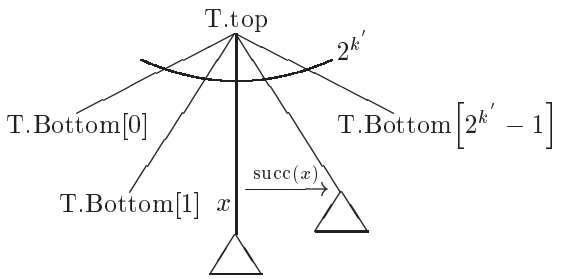
1. der Zahl T.size $= |\{x_1, \ldots, x_m\}| = |S| = m$
2. einer doppelt verketteten Liste T.list, die die Elemente von S in aufsteigender Reihenfolge enthält
3. einem Bitvektor T.b$[0 \ldots 2^k - 1]$ mit T.b$[i] = \begin{smallmatrix} 0 \\ 1 \end{smallmatrix}$ falls $\begin{smallmatrix} i \notin S \\ i \in S \end{smallmatrix}$ einem Zeiger (Pointer) Vektor T.p$[0 \ldots 2^k - 1]$. Falls T.b$[i] = 1$, dann zeigt T.p$[i]$ auf i in der Liste aus 2.
4. einer k'-Struktur T.top und einem Feld T.bottom$[0 \ldots 2^{k'} - 1]$ von k''-Strukturen. Falls $m = 1$, dann T.top, T.bottom und die zugehörigen k''-Strukturen leer, T.size $= 1$. T.list $= \{x\}$, der Bitvektor wird nicht benötigt. Falls $m > 1$, dann ist T.top eine k'-Struktur für die durch $\{x'_1, x'_2, \ldots, x'_m\}$ gegebene Menge, und für jedes $x \in [0 \ldots 2^{k'} - 1]$ ist T.bottom$[x]$ eine k''-Struktur für die Menge $\{x''_i;\ 1 \leq i \leq m;\ x = x'_i\}$

Beispiel:
$k = 4,\ S = \{2, 3, 7, 10, 13\},\ \text{T.size} = 5$



T.top ist 2-Struktur $\{0, 1, 2, 3\}$
T.bottom[0] ist eine 2-Struktur für $\{2, 3\}$
T.bottom[1] ist eine 2-Struktur für $\{3\}$
T.bottom[2] ist eine 2-Struktur für $\{2\}$
T.bottom[3] ist eine 2-Struktur für $\{1\}$
Operation succ(x) findet $\min\{y \in \text{S};\ y > x\}$ in der k-Struktur T.

if $k = 1$ or T.Size ≤ 2 then
    naive Suche
else
    if T.size $= 0$ or $x \geq$ max in T.list then
        return(succ(x) gibt's nicht)
    else
        $x' := \left\lfloor \frac{x}{2^{k''}} \right\rfloor$
        $x'' := x \mod 2^{k''}$
        if T.top.b$[x'] = 1$ and $x'' < \max\{\text{T.bottom}[x']\}$ then
            return$(x' \cdot 2^{k''} + \text{succ}(x'', \text{T.bottom}[x']))$
        else
            $z' := \text{succ}(x', \text{T.top})$, return$(z' \cdot 2^{k''} + \min\{\text{T.bottom}[z']\})$
        fi
    fi
fi



Kosten: $\text{T}(k) \leq c + \text{T}(\lceil \frac{k}{2} \rceil) = \mathcal{O}(\log k)$

Lemma: Die Succ-Operation hat Zeitbedarf $\mathcal{O}(\log k)$

Beweis: $\surd$ □

Insert Operation:

- falls x bereits in Priority-Queue, dann fertig

- bestimme succ(x, T), füge x davor ein
- bestimme x' und x''
- behandle die entsprechende Unterstrukturen rekursiv:
  Insert$(x', \mathrm{T.top})$, Insert$(x'', \mathrm{T.bottom}[x'])$ (nur ein nicht trivialer rekursiver Aufruf)

Zeitbedarf: naiv $\mathcal{O}(\log^2 k)$, Optimierung: das oberste succ tut alles $\mathcal{O}(\log k)$

Delete Operation:

Komplexität von Delete in k-Struktur: $\mathcal{O}(\log k)$
Kosten der Initialisierung: $\sim$ Größe der Datenstruktur
Platzbedarf für k-Struktur: Sei $\mathrm{S}(k)$ der Platzbedarf für eine k-Struktur.
Dann gilt:
$\mathrm{S}(1) = c$
$\mathrm{S}(k) = c2^k + \mathrm{S}(k') + 2^{k'}\mathrm{S}(k')$ für $k \geq 2$
Wir ersetzen zur Vereinfachung: $\mathrm{S}(k) = c2^k + \mathrm{S}(\frac{k}{2}) + 2^{\frac{k}{2}}\mathrm{S}(\frac{k}{2})$

Lemma: $\mathrm{S}(k) = \mathcal{O}(2^k \cdot \log k)$

Beweis:
Setze: $\mathrm{S}(k) := c' \cdot 2^k \log k$ (besser: Zeige $\mathrm{S}(k) := c'2^k \log k$ funktioniert)
Für $k = 1$ klar
Rekursionsschritt:

$$\begin{aligned}
\text{Platz für } k\text{-Struktur} &\leq c2^k + c'2^{\frac{k}{2}}(\log k - 1) + 2^{\frac{k}{2}}c'2^{\frac{k}{2}}(\log k - 1) \\
&= c2^k + c'2^{\frac{k}{2}}(1 + 2^{\frac{k}{2}})(\log k - 1) \\
&= c2^k + c'2^{\frac{k}{2}}(2^{\frac{k}{2}}\log k + \log k - 2^{\frac{k}{2}} - 1) \\
&\leq c2^k + c'2^{\frac{k}{2}}(2^{\frac{k}{2}}\log k + \log k - 2^{\frac{k}{2}}) \\
&\leq c'2^k \log k \text{ falls} \\
&\quad c2^k + c'2^k \log k + c'2^{\frac{k}{2}}\log k - c'2^k \leq c'2^k \log k \\
&\quad \Leftrightarrow c'2^{\frac{k}{2}}\log k \leq (c' - c)2^k \\
&\quad \Leftrightarrow \frac{c' - c}{c'} \geq \frac{\log k}{2^k} \text{ (Gilt für } c' \text{ groß genug.)}
\end{aligned}$$

□

Satz: Sei $N = 2^k$, Universum $U = \{0, \ldots, N-1\}$. Wird eine Teilmenge $S \subseteq U$ durch eine k-Struktur dargestellt, dann benötigen die Operationen Insert, Delete und Find_Min jeweils Zeit $\mathcal{O}(\log\log N)$, die Initialisierung Zeit $\mathcal{O}(N \log\log N)$. Der Platzbedarf ist ebenfalls $\mathcal{O}(N \log\log N)$.

Beweis: s.o. □

Literatur zu van Emde Boas-Priority Queue:

1. Mehlhorn Bd. 1, S.290-296
2. van Emde Boas, P., R. Kaas, E. Zijlstra: Design and Implementation of an efficient priority queue, Math. Systems Theory 10 (1977), S. 99-127

1.3.3 Sich selbst organisierende Datenstrukturen

1.3.3.1 Sich selbst organisierende lineare Listen (als Wörterbücher)

Heuristiken, um häufig besuchte Elemente nahe des Listenanfangs zu halten.

Beispiele:

1. Move_to_front
2. Transpose: vertausche das besuchte Element mit seinem Vorgänger in der Liste.

Wir wollen zeigen, daß Move_to_front (MF) bis auf einen konstanten Faktor (im amortisierten Sinn) optimal ist. Für die Algorithmen nehmen wir an:

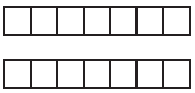
1. Kosten für access(x_i) und Delete(x_i) sind gleich der Position von x_i in der Liste.
2. Kosten für Insert(x) (x nicht in Liste) sind die Länge der Liste nach der Insert-Operation.
3. Ein Algorithmus kann die Liste jederzeit umorganisieren.
   (a) kostenlose Vertauschung: nach einer Access(x)- oder Insert(x)-Operation kann x beliebig näher an den Anfang der Liste verschoben werden.
   (b) bezahlte Vertauschung: jede andere Vertauschung benachbarter Listenelemente kostet eine Einheit.

Bemerkung: MF (und auch Transpose) benutzen nur kostenlose Vertauschungen. 3a ist pessimistisch für MF.

Satz: Sei s eine beliebige Folge von Access, Insert und Delete Operationen, beginnend mit einer leeren Liste. Sei A ein beliebiger (optimaler) Wörterbuch-Algorithmus mit obigen Einschränkungen. Sei $C_A(s)$ der Zeitaufwand von A auf s, ohne bezahlte Vertauschungen. Sei $X_A(s)$ die Anzahl der bezahlten Vertauschungen. Sei $C_{MF}(s)$ der Gesamtaufwand der Move_to_front-Heuristik auf s. Sei $F_A(s)$ die Anzahl der kostenlosen Vertauschungen, die A auf s ausführt. Sei $m := |s|$ die Länge von s. Dann

$$C_{MF}(s) \leq 2 \cdot C_A(s) + X_A(s) - F_A(s) - m$$

Beweis: Induktion über $|s|$. Potentialmethode für amortisierte Kosten von MF. □



Potential := $|\{\{x, y\};\ x \neq y;\ x, y$ kommen in beiden Listen vor, jedoch in verschiedener Reihenfolge$\}|$

Betrachte Insert(x) als access auf die (letzte + 1) Position in der Liste. Eine Access(x) oder Delete(x) Operation kostet also MF reell Zeit t, wenn t die Position von x in MF's Liste ist. i sei die Position von x in A's Liste.

Behauptung:

1. Die amortisierten Kosten von Access- (bzw. Insert-) Operationen sind $\leq 2i-1$ (i Index des Elements in der Liste).

2. Die amortisierten Kosten für eine kostenlose Vertauschung durch Algorithmus A sind ≤ -1.

3. Die amortisierten Kosten für eine bezahlte Vertauschung durch Algorithmus A sind $\leq +1$.

Beweis:

1. Liste L_A des Algorithmus A: $x_1\ x_2 \ldots x_i \ldots x_h \ldots x_n$
   Liste L_{MF} des Algorithmus MF: $\ldots x_h \ldots\ldots x_i \ldots$ $(\mathrm{Pos}(x_i) = t)$

   Sei k die Anzahl der Elemente (wie x_h), die in L_{MF} vor und in L_A nach x_i vorkommen.
   $\Rightarrow t \leq i + k \Rightarrow k \geq t - i$
   $\Delta_{\text{balance}} \leq -k + (i-1)$, also: $\Delta_{\text{balance}} \leq i - t + i - 1 \leq 2i - 1 - t$
   amortisierte Kosten = tatsächliche Kosten + $\Delta_{\text{balance}} \leq t + 2i - 1 - t = 2i - 1$
   Bei Delete Operation: $\Delta_{\text{balance}} = -k \leq i - t \Rightarrow$ amortisierte Kosten $\leq i$

2. L_A : $x_1 x_2 \ldots x_{i-1} x_i x_{i+1} \ldots x_n$ $\Delta_{\text{balance}} \leq -1$
   L_{MF} : $x_i \ldots\ldots$

3. L_A : $x_1 x_2 \ldots x_{i-1} x_i x_{i+1} \ldots x_n$ $\Delta_{\text{balance}} \leq 1$
   L_{MF} : $\ldots\ldots x_i \ldots\ldots$

   tatsächliche Kosten C_{MF} = amortisierte Kosten $- \underbrace{\Delta_{\text{balance}}}_{\geq 0}$
   $\leq$ amortisierte Kosten
   $\leq 2C_A(s) + X_A(s) - F_A(s) - m$

□

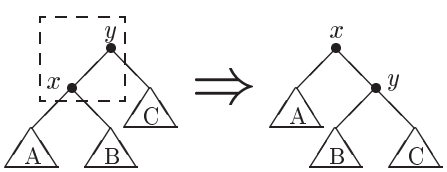
Anmerkung: Wenn wir nicht mit einer leeren Liste (oder zwei identischen Listen, d.h. $L_A = L_{MF}$) starten, ergibt sich:
$C_{MF} \leq 2C_A(s) + X_A(s) - F_A(s) - m + n^2$

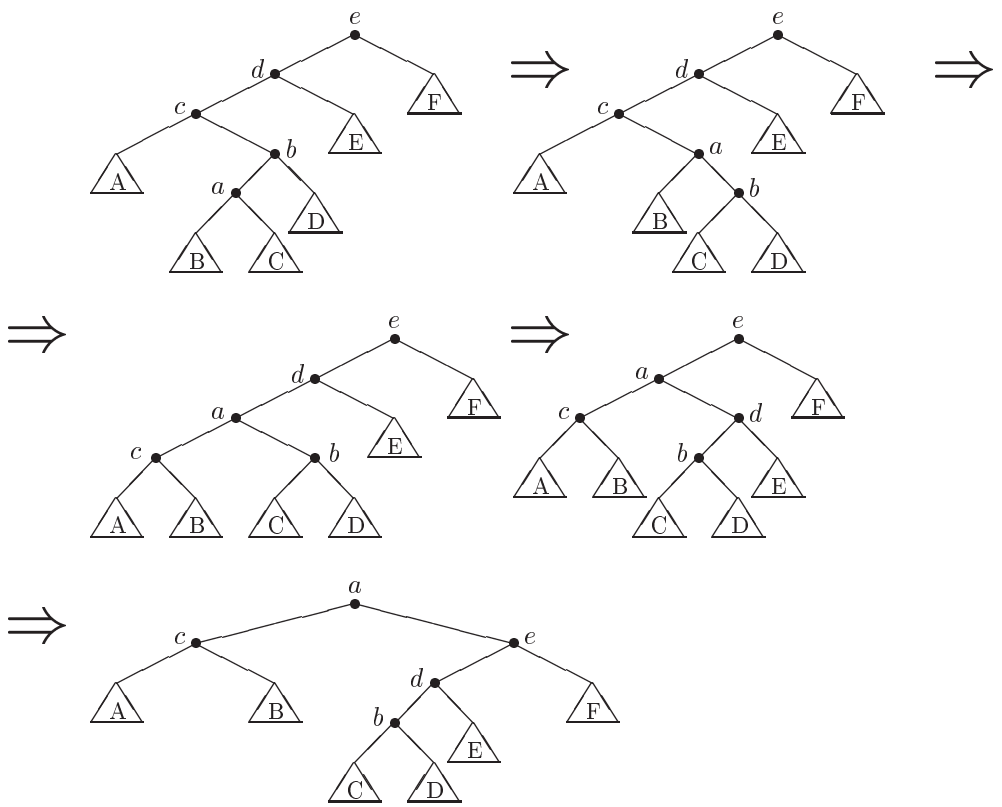
Literatur: Sleator, D.R., Tarjan: Amortized efficiency of list update and paging rules. Comm. ACM 28(2), p. 202-208 (1985)

1.3.3.2 Splay-Trees als Suchbäume

1.3.3.2.1 Einleitung

Rotationen, um Knoten an die Wurzel zu bewegen.
Beispiele:

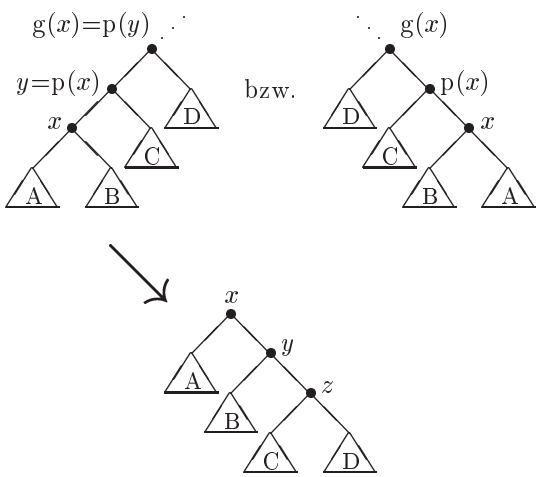




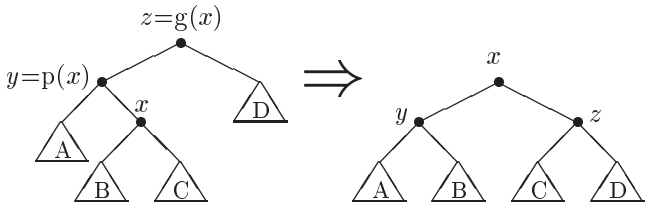
1.3.3.2.2 Die Splaying-Operation

Splay(x, T) bewegt mittels (einfacher + doppelter) Rotationen x zur Wurzel.
Fälle:

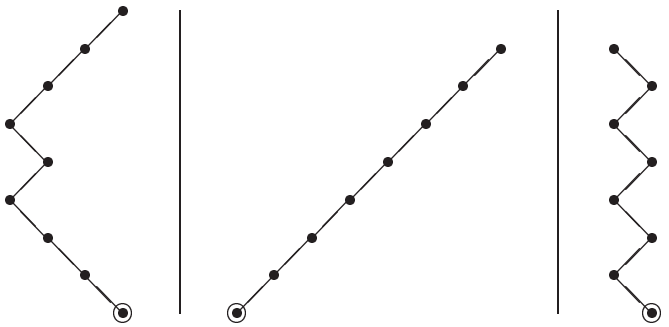
1. (zig): x ist Kind der Wurzel von T: einfache Rotation
2. (zig-zig): x hat Großvater g(x) (und Vater p(x)) und x und p(x) sind jeweils linke (bzw. rechte) Kinder ihres Vaters.



3. (zig-zag): x hat Großvater g(x) und Vater p(x), x ist linkes (rechtes) Kind von p(x), p(x) ist rechtes (linkes) Kind von g(x).



Übung: (Splaying-Operation mit dem eingekreisten Element durchführen)



1.3.3.2.3 Amortisierte Kostenanalyse der Splay-Operation

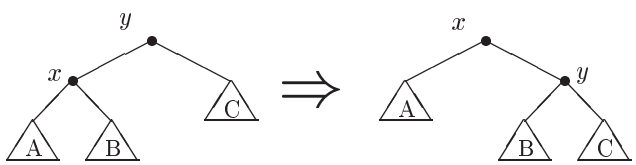
Jeder Knoten habe Gewicht $\mathrm{w}(x) > 0$. Gewicht $\mathrm{tw}(x)$ des Unterbaums mit Wurzel x ist $\sum_{\substack{y \text{ im} \\ \text{Unterbaum}}} \mathrm{w}(y)$.

Rang $\mathrm{r}(x) = \log(\mathrm{tw}(x))$
Potential eines Baumes T: $\sum_{x \in T} \mathrm{r}(x)$

Lemma: Sei T ein Splay-Tree mit Wurzel u, x ein Knoten in T. Die amortisierten Kosten für Splay(x, T) sind $\leq 1 + 3(\mathrm{r}(u) - \mathrm{r}(x)) = \mathcal{O}\left(\log \frac{\mathrm{tw}(u)}{\mathrm{tw}(x)}\right)$.

Beweis:
Berechne r und r′, tw und tw′, die Rang- bzw. Gewichtsfunktion vor und nach dem Rotationsschritt. Wir zeigen, daß die amortisierten Kosten im Fall 1 (zig) $\leq 1+3(\mathrm{r}'(x)-\mathrm{r}(x))$ und in den Fällen 2 und 3 (zig-zig bzw. zig-zag) $\leq 3(\mathrm{r}'(x)-\mathrm{r}(x))$ sind. y sei der Vater von x, z der Großvater (falls er existiert).

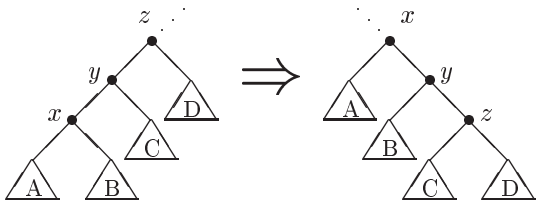
1. Fall:



amortisierte Kosten:

$$\leq 1 + \mathrm{r}'(x) + \mathrm{r}'(y) - \mathrm{r}(x) - \mathrm{r}(y) \leq 1 + \mathrm{r}'(x) - \mathrm{r}(x), \quad \text{da } \mathrm{r}'(y) \leq \mathrm{r}(y)$$
$$\leq 1 + 3(\mathrm{r}'(x) - \mathrm{r}(x)), \quad \text{da } \mathrm{r}'(x) \geq \mathrm{r}(x)$$

2. Fall:



amortisierte Kosten:

$$\begin{aligned}
&\leq 2 + \mathrm{r}'(x) + \mathrm{r}'(y) + \mathrm{r}'(z) - \mathrm{r}(x) - \mathrm{r}(y) - \mathrm{r}(z) \\
&= 2 + \mathrm{r}'(y) + \mathrm{r}'(z) - \mathrm{r}(x) - \mathrm{r}(y), \quad \text{da } \mathrm{r}'(x) = \mathrm{r}(z) \\
&\leq 2 + \mathrm{r}'(x) + \mathrm{r}'(z) - 2\mathrm{r}(x), \quad \text{da } \mathrm{r}'(x) \geq \mathrm{r}'(y) \text{ und } \mathrm{r}(y) \geq \mathrm{r}(x)
\end{aligned}$$

Es gilt, daß

$$2 + \mathrm{r}'(x) + \mathrm{r}'(z) - 2\mathrm{r}(x) \leq 3(\mathrm{r}'(x) - \mathrm{r}(x))$$

d.h.

$$2\mathrm{r}'(x) - \mathrm{r}(x) - \mathrm{r}'(z) \geq 2$$

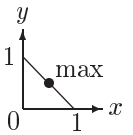
Betrachte dazu die Funktion

$$\mathrm{f}(x, y) = \log(x) + \log(y)$$

auf dem Bereich

$$x, y > 0, \ x + y \leq 1$$

d.h. $\mathrm{f}(x, y)$ nimmt ihr eindeutiges Maximum im Punkt $\left(\frac{1}{2}, \frac{1}{2}\right)$ an.



Beweis: Da die Funktion $\mathrm{z} \to \log(z)$ streng monoton wachsend ist, kann sich das Maximum der Funktion $\mathrm{f}(x, y) = \log(x) + \log(y)$ nur auf dem Geradensegment $x + y = 1$, $x, y > 0$ befinden. Dadurch erhalten wir ein neues Maximierungsproblem für die Funktion $\mathrm{g}(x) = \log(x) + \log(1 - x)$ auf diesem Geradensegment. Da $\mathrm{g}(x)$ an den Rändern gegen $-\infty$ strebt, muß es sich um ein lokales Maximum handeln. Die einzige Nullstelle der Ableitung

$$\mathrm{g}'(x) = \frac{1}{\ln a}\left(\frac{1}{x} - \frac{1}{1-x}\right),$$

wenn $\log = \log_a$, ist $x = 0.5$ (unabhängig von a). Weiter ist

$$\mathrm{g}''(x) = -\frac{1}{\ln a}\left(\frac{1}{x^2} + \frac{1}{(1-x)^2}\right).$$

Da $\mathrm{g}''(0.5) < 0$ ist nimmt $\mathrm{g}(x)$ in $x = 0.5$ sein globales Maximum an. Insgesamt folgt, daß die Funktion $\mathrm{f}(x, y) = \log(x) + \log(y)$ ihr globales Maximum im Bereich $x, y > 0$, $x + y \leq 1$ an der Stelle $(0.5, 0.5)$ annimmt. □

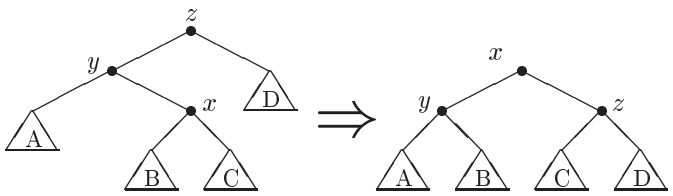
Damit gilt:

$$\mathrm{r}(x) + \mathrm{r}'(z) - 2\mathrm{r}'(x) = \log\left(\frac{\mathrm{tw}(x)}{\mathrm{tw}'(x)}\right) + \log\left(\frac{\mathrm{tw}'(z)}{\mathrm{tw}'(x)}\right) \leq -2$$

da

$$\mathrm{tw}(x) + \mathrm{tw}'(z) \leq \mathrm{tw}'(x)$$

3. Fall:



amortisierte Kosten:

$$\begin{aligned}
&\leq 2 + \mathrm{r}'(x) + \mathrm{r}'(y) + \mathrm{r}'(z) - \mathrm{r}(x) - \mathrm{r}(y) - \mathrm{r}(z) \\
&\leq 2 + \mathrm{r}'(y) + \mathrm{r}'(z) - 2\mathrm{r}(x), \quad \text{da } \mathrm{r}'(x) = \mathrm{r}(z) \text{ und } \mathrm{r}(x) \leq \mathrm{r}(y) \\
&\leq 2(\mathrm{r}'(x) - \mathrm{r}(x)), \quad \text{da } 2\mathrm{r}'(x) - \mathrm{r}'(y) - \mathrm{r}'(z) \geq 2
\end{aligned}$$

Letzteres folgt aus der Behauptung über $\mathrm{f}(x, y)$ wie im Fall 2.

Die Gesamtbehauptung des Lemmas folgt dann durch Aufaddieren der amortisierten Kosten für die einzelnen Schritte (Beachte: Teleskop-Summe). □

Betrachte Folge von m Zugriffen in einem Splay-Tree mit n Knoten. Die Verringerung des Potentials ist, falls sich die Gewichte nicht ändern, beschränkt durch

$$\sum_{i=1}^{n} \log \frac{W}{w_i}; \qquad W := \sum_{i=1}^{n} w_i.$$

Satz: die gesamten Kosten für die m Zugriffe sind $\mathcal{O}((m+n)\log n + m)$

Beweis:
amortisierte Kosten = reelle + Δ_{balance}; reelle Kosten = amortisierte - Δ_{balance}.
Wähle $w_i = \frac{1}{n}$ für alle Knoten. $\Rightarrow$ amortisierte Kosten für einen Zugriff $\leq 1 + 3\log n$, da $W = \sum\limits_{i=1}^{n} w_i = 1$. Die Verringerung im Potential ist $\leq \sum\limits_{i=1}^{n} \log \frac{W}{w_i} = \sum\limits_{i=1}^{n} \log n = n \log n$. $\Rightarrow$ reelle Kosten $\leq m(1 + 3\log n) + n \log n$. □

Satz: Sei $\mathrm{q}(i)$ die Anzahl der Zugriffe auf das Element i. Falls auf jedes Element zugegriffen wird (also $\mathrm{q}(i) > 0$ für alle i), dann sind die Gesamtkosten für die Zugriffe

$$\mathcal{O}\left(m + \sum_{i=1}^{n} \mathrm{q}(i) \cdot \log\left(\frac{m}{\mathrm{q}(i)}\right)\right)$$

Beweis: Setze Gewicht des i-ten Knotens gleich $\frac{\mathrm{q}(i)}{m}$.

$$\Rightarrow \mathrm{W} = \sum_{i=1}^{n} \frac{\mathrm{q}(i)}{m} = 1$$

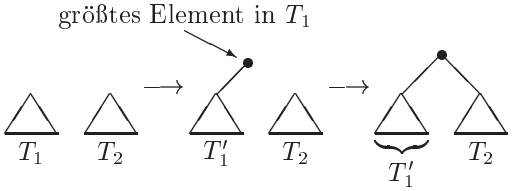
□

1.3.3.2.4 Wörterbuchoperationen in Splay-Trees

Betrachte (Alle diese Operationen werden mit Hilfe von Splay implementiert):

- Access$(i,\ T)$: $\surd$ (Haben wir bereits)
- Join$(T_1,\ T_2)$:

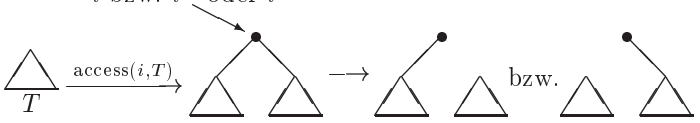
größtes Element in T_1



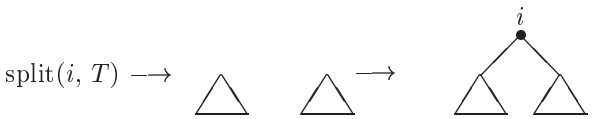
beachte $x \in T_1,\ y \in T_2$, dann gilt $x < y$

- Split$(i,\ T)$:

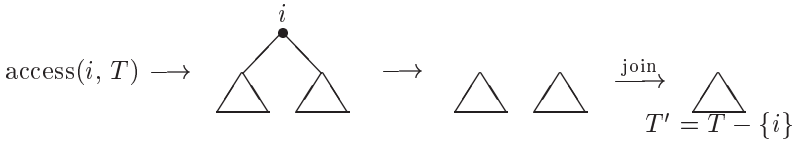
i bzw. i^- oder i^+



- Insert$(i,\ T)$:



- Delete$(i,\ T)$:



Definition: Sei $i \in U$; T ein Splay-Tree. Dann bezeichnen wir mit $i-$ (bzw. $i+$) den Vorgänger (bzw. den Nachfolger) von i in der gegebenen Ordnung in T (falls diese existieren). Falls $i-$ bzw. $i+$ undefiniert sind, so setzen wir $\mathrm{w}(i-) = \infty$ bzw. $\mathrm{w}(i+) = \infty$. Weiterhin sei W das Gesamtgewicht aller an einer Wörterbuch-Operation beteiligten Knoten.

Satz: Für die amortisierten Kosten der Wörterbuch-Operationen in Splay-Trees gelten die folgenden oberen Schranken: $(T,\ T_1,\ T_2 \neq 0)$

$$\text{access}(i,T): \begin{cases} 3\log\left(\frac{W}{\mathrm{w}(i)}\right)+1, \text{ falls } i \in T \\ 3\log\left(\frac{W}{\min\{\mathrm{w}(i-),\ \mathrm{w}(i+)\}}\right)+1, \text{ falls } i \notin T \end{cases}$$

$$\text{split}(i,T): \begin{cases} 3\log\left(\frac{W}{\mathrm{w}(i)}\right)+\mathcal{O}(1), \text{ falls } i \in T \\ 3\log\left(\frac{W}{\min\{\mathrm{w}(i-),\ \mathrm{w}(i+)\}}\right)+\mathcal{O}(1), \text{ falls } i \notin T \end{cases}$$

$$\text{Join}(T_1,T_2): 3\log\left(\frac{W}{\mathrm{w}(i)}\right)+\mathcal{O}(1),\ i \text{ maximal in } T_1$$

$$\underset{i\notin T}{\text{Insert}}(i,T): 3\log\left(\frac{W-\mathrm{w}(i)}{\min\{\mathrm{w}(i-),\mathrm{w}(i+)\}}\right)+\log\left(\frac{W}{\mathrm{w}(i)}\right)+\mathcal{O}(1)$$

$$\underset{i\in T}{\text{Delete}}(i,T): 3\log\left(\frac{W}{\mathrm{w}(i)}\right)+3\log\left(\frac{W-\mathrm{w}(i)}{\mathrm{w}(i-)}\right)+\mathcal{O}(1),$$

falls i nicht minimal in T

1.3.4 Weitere Arten von Datenstrukturen

- Dynamische Datenstrukturen:
  Referenz: Bent, S. W.: Dynamic weighted data structures, TR STAN-CS-82-916, Department of Computer Science, Stanford University, 1982

- Persistente Datenstrukturen:
  Referenz: Driscoll, J. R., Sarnak, N., Sleater, D. D., Tarjan, R. E., Making data structures persistent, Proceedings of the 18^{th} Annual ACM Symposium on Theory of Computing (STOC), 1986. p.109–121.

Kapitel 2

Selektion

2.1 Einleitung

Gegeben: Menge S von n Elementen aus einem total geordneten Universum U, $i \in \mathbb{N}$, $1 \leq i \leq n$.
Gesucht: i-kleinstes Element in S
Bekannt: Fälle $i = 1,\ n$ (Minimum, Maximum bestimmen)
Standardalgorithmus: $n-1$ Vergleiche
Zeigen: Diese Anzahl ist optimal:
Beweis:
Interpretiere jeden Algorithmus als Turnier. Ein Spiel wird vom kleineren Element gewonnen. Beobachtung: Jedes Element außer dem Gesamtsieger muß mindestens ein Spiel verloren haben $\Rightarrow n-1$ Vergleiche notwendig.
Nun: Bestimmung des Vize-Meisters (bzw. zweitkleinstes Element) □

Satz: Das zweitkleinste Element kann mit $n + \lceil \log_2 n \rceil - 2$ Vergleichen bestimmt werden.

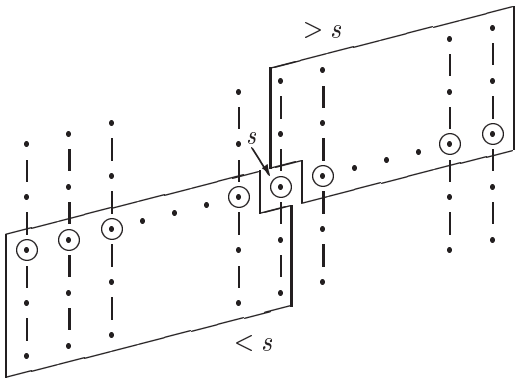
Beweis:
Bestimme das Minimum in einem K.O. Turnier: $(n-1)$ Vergleiche. Beobachtung: Das zweitkleinste Element ist unter den „Verlierern" gegen das Minimum zu suchen. Deren Anzahl ist $\lceil \log_2 n \rceil$. Bestimme unter diesen Elementen wiederum das Minimum $\rightsquigarrow$ zweitkleinstes Element $(\lceil \log_2 n \rceil - 1)$ □

2.2 Der Blum-Floyd-Pratt-Rivest-Tarjan Selektions-Algorithmus

Sei m eine kleine ungerade Zahl (etwa $5 \leq m \leq 15$). Wir betrachten folgenden Algorithmus BFPRT:

1. Teile S in $\lceil \frac{n}{m} \rceil$ Blöcke auf, $\lfloor \frac{n}{m} \rfloor$ davon mit m Elementen
2. Sortiere diese $\lfloor \frac{n}{m} \rfloor$ Blöcke; falls $\lfloor \frac{n}{m} \rfloor$ gerade, sortiere auch die restlichen Elemente
3. Sei S' die Menge der Mediane dieser sortierten Mengen. Bestimme rekursiv den Median dieser Mediane (also das $\left\lceil \frac{|S'|}{2} \right\rceil$-kleinste Element). Nenne diesen Median s.

4. Partitioniere $S - \{s\}$ in $S_1 := \{x \in S : x < s\}$, $S_2 := \{x \in S : x > s\}$.
   Bemerkung.: $|S_1|,\ |S_2| \geq \frac{n}{4}$, falls $n \geq 3m - 1$



5. Falls $i \leq |S_1|$, bestimme rekursiv das i-kleinste Element in S_1
   Falls $|S_2| > n - i$, bestimme rekursiv das $(i - |S_1| - 1)$-kleinste Element in S_2
   Ansonsten: return(s)

Komplexitätsbetrachtung (# Vergleiche im worst-case):

Sei $\mathrm{T}(n)$ die worst-case Anzahl von Vergleichen für $|S| = n$ des Algorithmus BFPRT. Es gilt:

$$\mathrm{T}(n) \leq \underbrace{\mathrm{T}\left(\left\lceil \frac{n}{m} \right\rceil\right)}_{3.} + \underbrace{\mathrm{T}\left(\left\lfloor \frac{3}{4}n \right\rfloor\right)}_{5.} + \underbrace{\left\lceil \frac{n}{m} \right\rceil}_{2.} C_m + \underbrace{\left\lfloor \frac{n}{2} \right\rfloor}_{4.}$$

(C_m ist die Anzahl von Vergleichen, um m Elemente zu sortieren, z.B. $C_5 = 7$, $C_{11} = 26$)

Satz: Der Selektions-Algorithmus BFPRT bestimmt das i-te Element von n Elementen mit $\mathcal{O}(n)$ Vergleichen (und Zeit).

Behauptung:

$$\mathrm{T}(n) \leq c_m n;$$

Behauptung ist ok, falls

$$\mathrm{T}\left(\left\lceil \frac{n}{m} \right\rceil\right) + \mathrm{T}\left(\left\lfloor \frac{3}{4}n \right\rfloor\right) + \left\lceil \frac{n}{m} \right\rceil C_m + \left\lfloor \frac{n}{2} \right\rfloor \leq c_m n, \text{ dies gilt, falls}$$

$$\left\lceil \frac{n}{m} \right\rceil c_m + \left\lfloor \frac{3}{4}n \right\rfloor c_m + \left\lceil \frac{n}{m} \right\rceil C_m + \left\lfloor \frac{n}{2} \right\rfloor \leq c_m n \qquad | \cdot \frac{1}{n}$$

$$\Leftrightarrow \text{(bis auf } \lceil\ \rceil, \lfloor\ \rfloor\text{)} \ \frac{c_m}{m} + \frac{3}{4}c_m + \frac{C_m}{m} + \frac{1}{2} \leq c_m$$

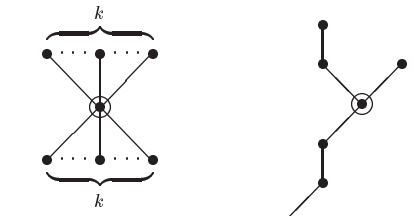
$$\Leftrightarrow -\frac{c_m}{m} - \frac{3}{4}c_m + c_m \geq \frac{C_m}{m} + \frac{1}{2}$$

$$\Leftrightarrow c_m \geq \frac{\frac{C_m}{m} + \frac{1}{2}}{1 - \frac{3}{4} - \frac{1}{m}}$$

Schlußbemerkung: $m = 11 \rightsquigarrow c_m \approx 20$

2.3 Der Schönhage-Paterson-Pippenger-Median-Algorithmus

Definition: Sei $k \in \mathbb{N}\backslash\{0\}$. P_k ist die folgende partielle Ordnung:

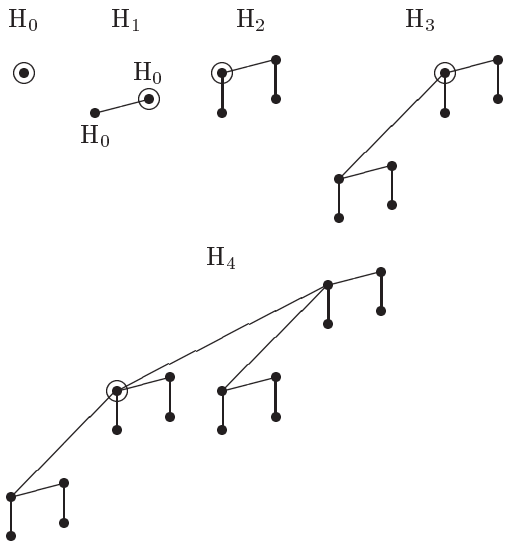


$2k+1$ Elemente $\supseteq P_3$

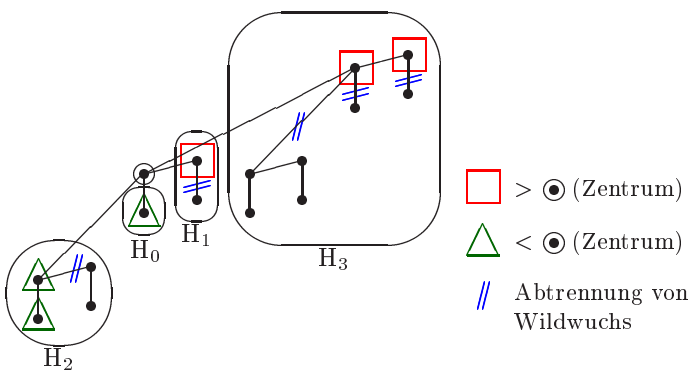
Spezielle Binomialbäume mit „Zentren“.

Definition:

1. Der Baum H_0 besteht aus einem Knoten und dieser ist auch das Zentrum.
2. $H_{2h}\,(h > 0)$ besteht aus zwei H_{2h-1}, deren Zentren durch eine neue Kante verbunden sind. Das Zentrum des H_{2h} ist das kleinere der beiden Zentren der H_{2h-1}.
3. $H_{2h+1}\,(h \geq 0)$ besteht aus zwei H_{2h}, deren Zentren durch eine neue Kante verbunden sind, sein Zentrum ist das größere dieser beiden Zentren.



Das Zerlegungslemma:



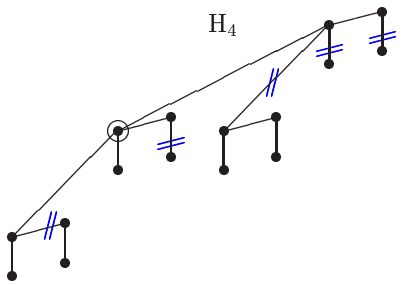
a) H_h hat 2^h Knoten, es werden $2^h - 1$ Vergleiche benötigt, um H_h zu konstruieren.

b) H_{2h} kann zerlegt werden in
   - sein Zentrum
   - eine Menge $\{H_1, H_3, \ldots, H_{2h-1}\}$ von disjunkten Teilbäumen, deren Zentren alle größer sind als das Zentrum von H_{2h}.
   - eine Menge $\{H_0, H_2, H_4, \ldots, H_{2h-2}\}$ von disjunkten Teilbäumen mit Zentren kleiner als das von H_{2h}.

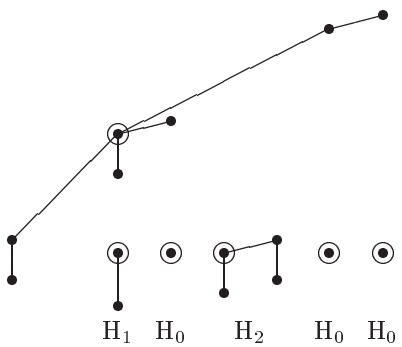
c) H_{2k+1} kann so zerlegt werden, daß die Zusammenhangskomponente des Zentrums genau 2^k Knoten $\geq$ dem Zentrum enthält, indem höchstens $2^{k+1} - 1$ Kanten entfernt werden.
   H_{2k} kann so zerlegt werden, daß die Zusammenhangskomponente des Zentrums genau 2^k Knoten enthält, die alle $\leq$ dem Zentrum sind, indem höchstens $2^k - 1$ Kanten entfernt werden.

d) Falls $k \leq 2^h - 1$, dann kann H_{2h} so zerlegt werden, daß die Zusammenhangskomponente des Zentrums genau $2k+1$ Elemente enthält, von denen k größer und k kleiner als das Zentrum sind ($\Rightarrow P_k$).
   Dazu genügt es, höchstens $3k + 2h$ Kanten zu entfernen. Die restlichen Zusammenhangskomponenten sind wieder H_i.

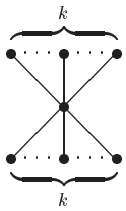
Bemerkung: Bei jedem Konstruktionsschritt wird ein Vergleich durchgeführt, um zu bestimmen, welcher der beiden Teilbäume das kleinere Zentrum hat. Im Algorithmus von Schönhage, Paterson und Pippenger werden aus Teilstücken H_r größere Bäume H_{r+1} zusammengebaut, wodurch schrittweise eine partielle Ordnung auf den Eingabewerten bestimmt wird. Wurde ein Baum H_{2h} hinreichender Größe hergestellt, so wird er durch Zerlegung in einen Baum umgewandelt, der nur noch sein altes Zentrum, sowie k darüberliegende und k darunterliegende Elemente enthält, wobei $k \leq 2^h - 1$. Im folgenden Beispiel wollen wir H_4 zerlegen und wählen $k = 3$:



Um einen H_4 derart zu zerlegen, müssen wir 5 Kanten aufbrechen. Dabei werden drei H_0, ein H_1 sowie ein H_2 abgespalten.



Übrig bleibt die gewünschte Struktur mit k Knoten über dem Zentrum und k unter dem Zentrum, wodurch eine partielle Ordnung auf $2k+1$ Eingabewerten bestimmt wurde:



$2k+1$ Elemente

Die bei der Zerlegung angefallenen Reststücke werden beim Aufbau weiterer Bäume benutzt. So geht das bereits angesammelte Wissen über die Ordnung der Elemente nicht verloren.

Beweis der Teile a) bis d) des Zerlegungslemmas:

a) **Lemma 1:** H_r hat 2^r Knoten, es werden $2^r - 1$ Vergleiche benötigt, um H_r aufzubauen.

Beweis: In jedem der r Konstruktionsschritte wird die Anzahl der Knoten verdoppelt. Da wir mit einem Knoten beginnen, hat H_r folglich 2^r Knoten. Die Anzahl der notwendigen Vergleiche C_r unterliegt folgender Rekursionsgleichung ($r \geq 1$): $C_r = 1 + 2C_{r-1}$ und $C_0 = 0$. Damit folgt sofort $C_r = 2^r - 1$.

□

b) **Lemma 2:** H_r kann in folgende disjunkte Bereiche unterteilt werden:

- sein Zentrum,
- eine Reihe H_1, H_3, ..., H_{r-1} (r gerade) bzw. ..., H_{r-2} (r ungerade) von Unterbäumen, deren Zentren über dem von H_r liegen,
- eine Reihe H_0, H_2, ..., H_{r-2} (r gerade) bzw. ..., H_{r-1} (r ungerade) von Unterbäumen, deren Zentren unter dem von H_r liegen.

Beweis: durch Induktion über r. Induktionsanfang: für H_0 gilt die Behauptung. Induktionsannahme: die Behauptung gelte für H_{r-1}.

1. Sei $r = 2h, h > 0$. H_{2h} besteht aus zwei H_{2h-1}, wobei das kleinere der beiden alten Zentren das neue Zentrum z bildet. Wende auf den H_{2h-1}, der z enthält, die Induktionsannahme an. Wir können diesen Unterbaum also in z, sowie H_1, H_3, ..., H_{2h-3} (Zentren über z) und H_0, H_2, ..., H_{2h-2} (Zentren unter z) partitionieren. Zusammen mit dem H_{2h-1}, dessen Zentrum über z liegt, ergibt sich die Induktionsbehauptung für H_{2h}.
2. Sei $r = 2h+1, h \geq 0$. H_{2h+1} besteht aus zwei H_{2h}, wobei das größere der beiden alten Zentren das neue Zentrum z bildet. Wende auf den H_{2h}, der z enthält, die Induktionsannahme an. Wir können diesen Unterbaum also in z, sowie H_1, H_3, ..., H_{2h-1} (Zentren über z) und H_0, H_2, ..., H_{2h-2} (Zentren unter z) partitionieren. Zusammen mit dem H_{2h}, dessen Zentrum unter z liegt, ergibt sich die Induktionsbehauptung fr H_{2h+1}.

□

Wir bezeichnen im folgenden mit H^-_{2h} den Baum, der entsteht, wenn wir H_{2h} so zerlegen, da alle Elemente oberhalb des Zentrums wegfallen. Mit H^+_{2h+1} bezeichnen wir den Baum, der entsteht, wenn wir H_{2h+1} so zerlegen, da alle Elemente unterhalb des Zentrums wegfallen.

c) **Lemma 3:** H^-_{2h} und H^+_{2h+1} haben jeweils 2^h Knoten. Bei der Herstellung aus H_{2h} bzw. H_{2h+1} werden $2^h - 1$ bzw. $2^{h+1} - 1$ Kanten aufgebrochen. Die wegfallenden Teile haben die Form $H_s, s < 2h$ bzw. $s < 2h + 1$.

Beweis: durch Induktion über r. Induktionsanfang: für H_0 und H_1 gilt die Behauptung. Induktionsannahme: die Behauptung gilt für alle $H_p, p < r$.

1. Sei $r = 2h, h > 0$. Wir betrachten die Partitionierung von H_{2h} mit Zentrum z wie in Lemma 2. Die Unterbäume H_1, H_3, ..., H_{2h-1} haben ihre Zentren oberhalb von z. Wir trennen sie von H_{2h}, indem wir h Kanten aufbrechen. Die abgetrennten Teile haben offensichtlich die Form $H_s, s < 2h$. Bei den Unterbäumen H_0, H_2, ..., H_{2h-2}, mit Zentren unterhalb von z, wenden wir jeweils die Induktionsannahme an, d.h. wir erzeugen H^-_0, H^-_2, ..., H^-_{2h-2}. Als Ergebnis erhalten wir H^-_{2h}. Damit gilt für die Zahl der aufzubrechenden Kanten $K^-(2h)$ zur Herstellung von H^-_{2h}:

$$K^-(2h) = h + \sum_{i=0}^{h-1} K^-(2i) \stackrel{I.A.}{=} h + \sum_{i=0}^{h-1}(2^i - 1) = \sum_{i=0}^{h-1} 2^i = 2^h - 1.$$

Für die Zahl $E^-(2h)$ der Elemente in H^-_{2h} gilt:

$$E^-(2h) = 1 + \sum_{i=0}^{h-1} E^-(2i) \stackrel{I.A.}{=} 1 + \sum_{i=0}^{h-1} 2^i = 1 + \underbrace{\sum_{i=1}^{h} 2^{i-1}}_{2^h-1} = 2^h.$$

2. Sei $r = 2h + 1, h > 0$. Wir betrachten die Partitionierung von H_{2h+1} mit Zentrum z wie in Lemma 2. Die Unterbäume H_0, H_2, ..., H_{2h} haben ihre Zentren unterhalb von z. Wir trennen sie von H_{2h+1}, indem wir $h + 1$ Kanten aufbrechen. Die abgetrennten Teile haben offensichtlich die Form $H_s, s < 2h + 1$. Bei den Unterbäumen H_1, H_3, ..., H_{2h-1}, mit Zentren oberhalb von z, wenden wir jeweils die Induktionsannahme an, d.h. wir erzeugen H^+_1, H^+_3, ..., H^+_{2h-1}. Als Ergebnis erhalten wir H^+_{2h+1}. Damit gilt für die Zahl der aufzubrechenden Kanten $K^+(2h+1)$ zur Herstellung von H^+_{2h+1}:

$$\begin{aligned} K^+(2h+1) &= h + 1 + \sum_{i=1}^{h} K^+(2(i-1)+1) \\ &\stackrel{I.A.}{=} h + 1 + \sum_{i=1}^{h}(2^i - 1) = 1 + \sum_{i=1}^{h} 2^i \\ &= 1 + \underbrace{\sum_{i=1}^{h+1} 2^{i-1}}_{2^{h+1}-1} - 1 = 2^{h+1} - 1. \end{aligned}$$

Für die Zahl $E^+(2h+1)$ der Elemente in H^+_{2h+1} gilt:

$$E^+(2h+1) = 1 + \sum_{i=1}^{h} E^+(2(i-1)+1) \stackrel{I.A.}{=} 1 + \underbrace{\sum_{i=1}^{h} 2^{i-1}}_{2^h-1} = 2^h.$$

□

d) **Lemma 4:** Falls $k \leq 2^h - 1$, dann kann H_{2h} so zerlegt werden, daß die Komponente des Zentrums genau $2k+1$ Elemente enthält, k davon über und k unter dem Zentrum. Dazu müssen $\leq 3k + 2h$ Kanten entfernt werden. Die entfernten Teile sind von der Form $H_s, s < 2h$.

Beweis: Betrachte die Binärdarstellung von $k = k_0 2^0 + k_1 2^1 + ... + k_{h-1} 2^{h-1}$ und die Partitionierung von H_{2h} mit Zentrum z wie in Lemma 2.

- Für jedes i mit $k_i = 1$, betrachte H_{2i+1} aus der Sequenz H_1, H_3, ..., H_{2h-1} von Unterbäumen deren Zentren oberhalb von z liegen und schneide alle Elemente aus H_{2i+1}, die kleiner als sein Zentrum sind (bilde also H^+_{2i+1}). Dazu müssen höchstens $2k$ Kanten aufgebrochen werden, denn jedes $k_i = 1$ steht für 2^i in k, kostet uns aber nach Lemma 2 $K^+(2i+1) = 2^{i+1} - 1$ Kanten, also:

$$\sum_{i=0}^{h-1} k_i K^+(2i+1) \leq 2k.$$

  Für jedes i mit $k_i = 0$, schneide H_{2i+1} ganz weg. Dabei werden höchstens h Kanten aufgebrochen. Genau k Elemente oberhalb von z bleiben zurück, da jedes $k_i = 1$ für 2^i in k steht, und ein H^+_{2i+1} genau $E^+(2i+1) = 2^i$ Elemente enthält, also:

$$\sum_{i=0}^{h-1} k_i E^+(2i+1) = k.$$

- Für jedes i mit $k_i = 1$, betrachte H_{2i} aus der Sequenz H_0, H_2, ..., H_{2h-2} von Unterbäumen, deren Zentren unterhalb von z liegen und schneide alle Elemente aus H_{2i}, die größer als sein Zentrum sind (bilde also H^-_{2i}). Dazu müssen höchstens $k-1$ Kanten aufgebrochen werden, denn jedes $k_i = 1$ steht für 2^i in k und kostet uns nach Lemma 3 $K^-(2i) = 2^i - 1$ Kanten, also:

$$\sum_{i=0}^{h-1} k_i (2^i - 1) \leq k - 1.$$

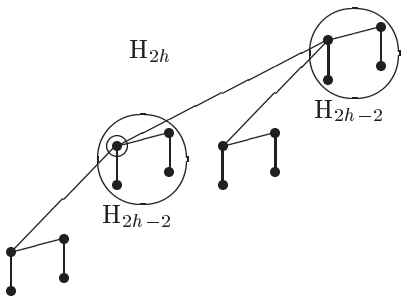
  Für jedes i mit $k_i = 0$, schneide H_{2i} ganz weg. Dabei werden höchstens h Kanten aufgebrochen. Genau k Elemente unterhalb von z bleiben zurück, da jedes $k_i = 1$ für 2^i in k steht, und ein H^-_{2i} genau $E^-(2i) = 2^i$ Elemente enthält, also:

$$\sum_{i=0}^{h-1} k_i E^-(2i) = k.$$

Damit ergibt sich fr die Gesamtanzahl aufzubrechender Kanten eine obere Schranke von $3k + 2h$. Lemma 3 liefert uns darüberhinaus die gewünschte Aussage ber die Form der abgetrennten Teile. □

Betrachte H_{2h}

- „größer“: $H_{2h-1}, H_{2h-3}, \ldots, H_1$
- „kleiner“: $H_{2h-2}, H_{2h-4}, \ldots, H_0$

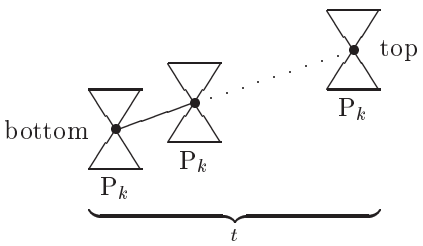


$U(h) :=$ Anzahl der Elemente in $H_{2h} \geq$ Zentrum: $U(h) = 2U(h-1) = 2^h U(0) = 2^h$
$D(h) :=$ Anzahl der Elemente in $H_{2h} \leq$ Zentrum: $D(h) = 2D(h-1) = 2^h D(0) = 2^h$

Anzahl der Kanten, die entfernt werden müssen:

$$\left.\begin{aligned} C_u(h) &\leq 2 + 2C_u(h-1) \\ &= 2 + 4 + 2^3 + \ldots + 2^h \\ &= 2^{h+1} - 2 \\ C_d(h) &\leq 1 + 2C_d(h-1) \\ &= 2^h - 1 \end{aligned}\right\} \quad C_H \leq 2^{h+1} - 2 + 2^h - 1 \approx 3 \cdot 2^h$$

Kette von P_k's:



Gesamtzahl der Elemente:

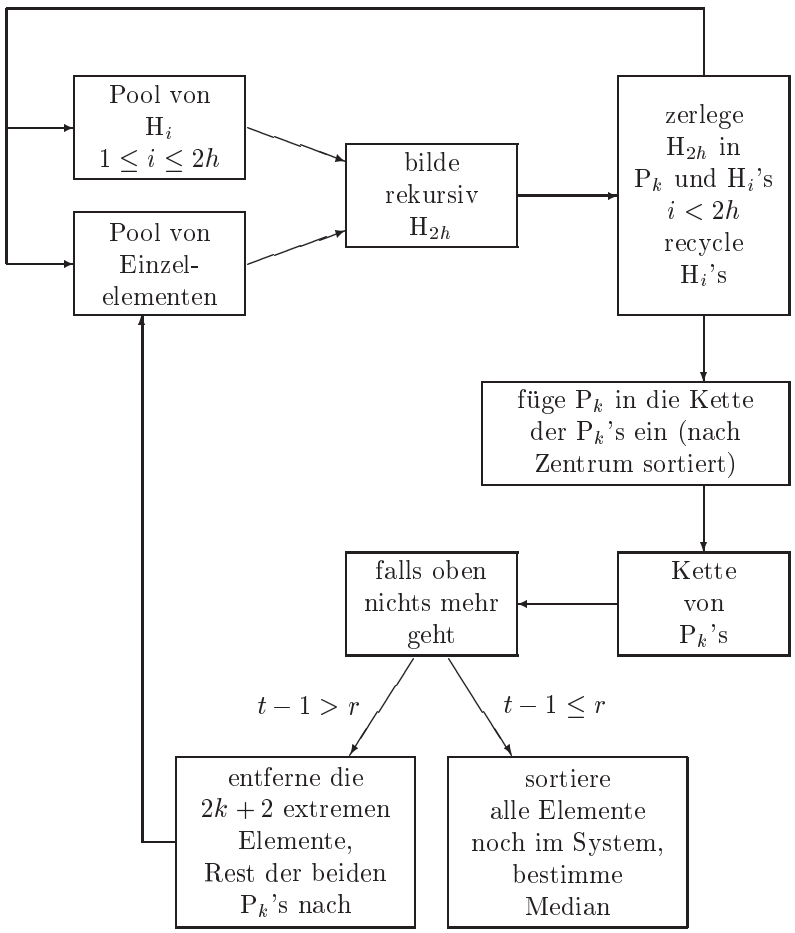
n
$t(2k+1)$ in den P_k's
$r = n - t(2k+1)$ Rest

Wenn $r < t - 1$, dann wissen wir, daß top größer ist als

$$k + (t-1)(k+1) > k + (k+1)\left(\frac{n+1}{2k+2} - 1\right) = \frac{n-1}{2}$$

$\Rightarrow$ top $>$ Median

$k := \left\lfloor n^{\frac{1}{4}} \right\rfloor$
h sdg. $2^{h-1} \leq k < 2^h$



Definiere: $r :=$ Anzahl der noch im H_{2h} Produktionsprozeß steckenden Elemente. (für jedes $i < 2h$ höchstens ein $H_i \Rightarrow r \leq \sum_{i=0}^{2h-1} = 2^{2h} - 1$)

$R :=$ Anzahl der im letzten Schritt zu sortierenden Elemente. Es gilt: $t \leq r+1$, und damit

$$R = t(2k+1) + r \leq 2^{2h}(2k+1) + 2^{2h} - 1$$

$m :=$ Gesamtzahl der im Algorithmus produzierten P_k's.

$$m = t + 2\frac{n-R}{2(k+1)} = t + \frac{n-R}{k+1}$$

Gesamtzahl der vom Algorithmus durchgeführten Vergleiche =

1. Anzahl der Kanten in allen P_k's
2. + Anzahl der Kanten, die gelöscht werden, um die P_k's zu formen
3. + Anzahl der Kanten, die zum Schluß in übriggebliebenen H_i's ($i < 2h$) stecken
4. + Anzahl der Vergleiche, um jedes Zentrum der P_k's in die (sortierte) Kette einzufügen
5. + Anzahl der Vergleiche, um die zum Schluß übriggebliebenenen R Elemente zu sortieren

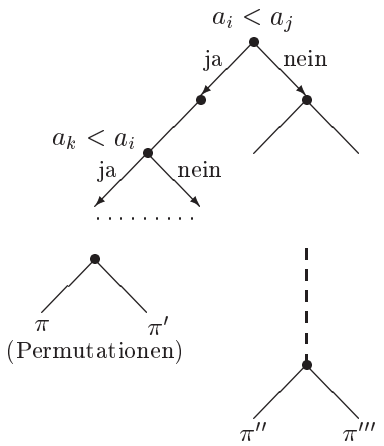
$$\leq \left(\frac{n-R}{k+1}+t\right)\left[\underbrace{2k}_{1}+\underbrace{3k+2h}_{2}+\underbrace{\log\frac{n}{2k+1}}_{4}\right]+\underbrace{R\log R}_{5}+\underbrace{r}_{3}$$

$k := \left\lfloor n^{\frac{1}{4}} \right\rfloor$, h so daß $2^{h-1} \leq k < 2^h$

$$\begin{aligned} \text{Damit } r &= \mathcal{O}(k^2) \\ t &= \mathcal{O}(k^2) \text{ zum Schluß} \\ R &= \mathcal{O}(k^3), \text{ und damit die} \\ \text{Anzahl der Vergleiche } &= \mathrm{T}(n) \leq 5n + o(n) \end{aligned}$$

Verbesserte Version: $T(n) = 3n + o(n)$
Dor/Zwick: $2,95n + o(n)$

Einschub: Untere Schranke für das vergleichsbasierte Sortieren
Entscheidungsbaum:



Stirling: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$
Binärbaum, alle Permutationen $\in S_n$ kommen als Blattmarkierungen vor.

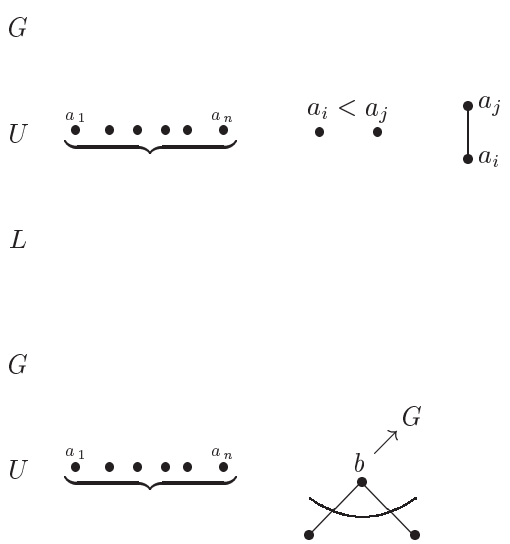
$$\begin{aligned} \text{worstcase Laufzeit} &\geq \text{Tiefe des Baums} \\ &\geq \log_2(\text{Anzahl der Blätter}) \\ &= \log_2(n!) \\ &= n\log n + \mathcal{O}(n) \end{aligned}$$

2.4 Eine untere Schranke für die Medianbestimmung

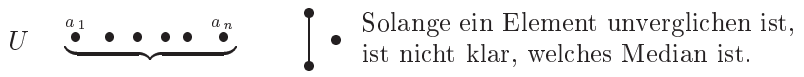
Satz: Jeder (vergleichsbasierte) Medianalgorithmus benötigt mindestens $\left\lceil \frac{3n}{2} \right\rceil - 2$ Vergleiche

Beweis: Gegenspielerargument
n Elemente, o.B.d.A. n ungerade, alle Elemente paarweise verschieden. Die Menge aller Elemente wird in drei Teilmengen partitioniert: U enthält die Kandidaten für den Median, G enthält Elemente, die sicher größer als der Median sind und L enthält Elemente, die sicher kleiner als der Median sind. Anfangs sind alle Elemente in U. Der Algorithmus stellt nun Fragen der Form $x < y$. Der Gegenspieler gibt

konsistente Antworten, die den Algorithmus jedoch dazu zwingen, möglichst viele Fragen zu stellen (d.h. U soll möglichst ungeordnet bleiben).



G $\quad \leq \frac{n+1}{2} - 1$

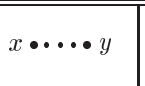
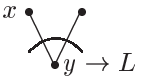
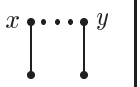
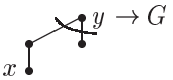
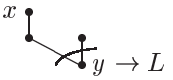
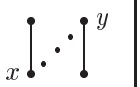
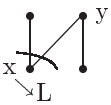


L $\quad \leq \frac{n+1}{2} - 1$

Solange $|L|$, $|G| < \frac{n+1}{2}$, kann der Algorithmus annehmen, daß der Median in U ist. Solange U mindestens zwei unverglichene Elemente und keine Zusammenhangskomponente mit > 2 Elementen enthält, kann der Algorithmus den Median nicht bestimmen.
Strategie des Gegenspielers: Zwei Phasen
Erste Phase: Fragen des Algorithmus: $\bullet \ldots \bullet$

$C :=$ Anzahl der Vergleiche
$P :=$ Anzahl der Paare in U
Dann gilt folgende Invariante: $C - P + 2|U| \geq 2n$

Beweis: vollständige Induktion □

	Status	Antwort des Gegenspielers	Anzahl der Paare in U	$\lvert U\rvert$	$\lvert L\rvert$	$\lvert G\rvert$
1.			+1	—	—	—
2.			-1	-1	0	+1
3.			-1	-1	1	0
4.			-1	-1	0	+1
5.			-1	-1	+1	0
6.			-1	-1	+1	0

Die erste Phase endet, wenn $|L| = \frac{n-1}{2}$ oder $|G| = \frac{n-1}{2}$ oder wenn U keine unverglichenen Elemente mehr enthält, d.h. $|U| \leq 2$. Wenn eine dieser Bedingungen eintritt, beginnt der Gegenspieler mit Phase 2:

- 1. Fall:

  $$|U| \leq 2$$

  Dann folgt aus der Invariante für Phase 1:

  $$C - P + 2 \cdot 2 \geq 2n, \text{ also } C \geq 2n - 3$$

- 2. Fall: $|L|$ wird $\frac{n-1}{2}$ oder $|G|$ wird $\frac{n-1}{2}$
  o.B.d.A.:

  $$|L| = \frac{n-1}{2}$$

  Der Gegenspieler zwingt nun den Algorithmus, das minimale Element in U zu bestimmen (da dieses unter den Vorgaben der Median ist).
  Von den dazu nötigen $|U|-1$ Vergleichen hat der Algorithmus bereits höchstens P ausgeführt (P = Anzahl der Paare), bleiben also $|U| - 1 - P$.
  Zusammen (Phase 1 + Phase 2) ergeben sich damit $\geq C + |U| - 1 - P$ Vergleiche.
  Da $C \geq 2n + P - 2|U|$ ist die Anzahl der Vergleiche:

  $$\geq 2n + P - 2|U| + |U| - 1 - P = 2n - |U| - 1 \geq \left\lceil \frac{3n}{2} - 2 \right\rceil$$

  da $|U| \leq \frac{n-1}{2}$.

□

2.5 Eine bessere untere Schranke

Literatur: Samuel W. Bent, John W. John: "Finding the median requires $2n$ comparisons." Proceedings of the 17^{th} Annual ACM Symposium on Theory of Computing 1985, p.213–216.

Satz: Sei T ein Entscheidungsbaum für die Bestimmung des i-kleinsten von n verschiedenen Elements, mit $i^2 \geq \log\left[\binom{n}{i}\frac{1}{n-i+1}\right] + 3$. Dann gilt, mit

$$p := 2\sqrt{\log\left[\binom{n}{i}\frac{1}{n-i+1}\right] + 3} - 2,$$

$$\text{Höhe}(T) \geq \log\left[\binom{n}{i}\frac{2^{n-p}}{n-i+1}\right]$$

($\rightarrow 2n - o(n)$ für $i = \frac{n}{2}$ (untere Schranke für Median))

Beweis: Gegenspielerargument + Abzählargument
„T hat viele Blätter", A Teilmenge der Schlüssel, $|A| = i$
T_A Teilbaum von T, alle Blätter von T_A sind auch Blätter von T.
Zu zeigen: T_A hat viele Blätter. ($\binom{n}{i}$ Möglichkeiten, T_A's zu konstruieren, Anzahl Blätter in T_A: 2^{n-p}, jedes Blatt von T kommt in höchstens $n - i + 1$ T_A's vor)
Beobachtungen: Jedes Blatt w von T liefert folgende Informationen:

answer(w)	$\xrightarrow{\text{liefert}}$	i-kleinstes Element x
little(w)	$\xrightarrow{\text{liefert}}$	$i-1$ Elemente $< x$
big(w)	$\xrightarrow{\text{liefert}}$	$n-i$ Elemente $> x$

Wähle A als beliebige Teilmenge der n gegebenen Schlüssel, mit $|A| = i$. Wir geben für den Gegenspieler eine Strategie an, welche dazu führt, daß wir durch Zurechtschneiden aus T einen Baum T_A konstruieren können, so daß gilt:

- T_A ist Binärbaum
- jedes Blatt von T_A ist auch Blatt von T (keine zus. Blätter durch Abschneiden)
- für jedes Blatt w von T_A gilt: little(w) $\subseteq A$

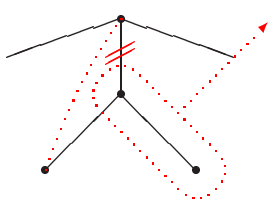
Setze $\bar{A}$ das Komplement von A,

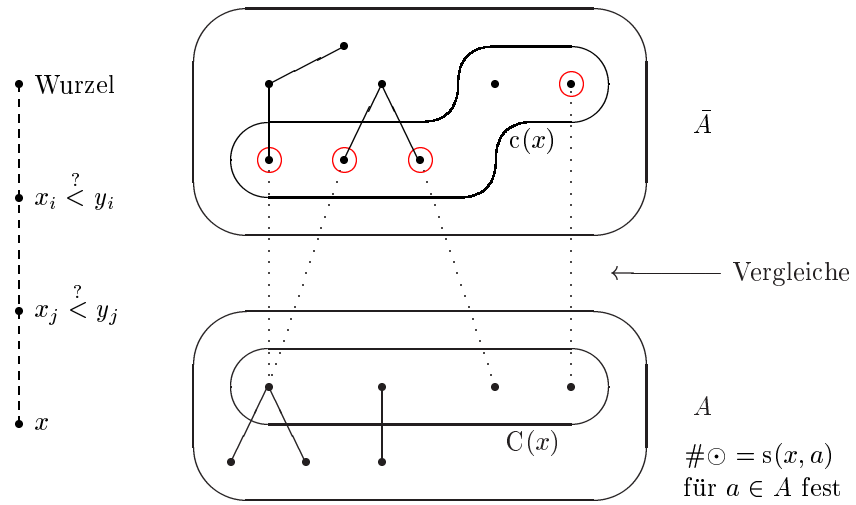
$$r := \sqrt{\log\left[\binom{n}{i}\frac{1}{n-i+1}\right] + 3}$$

$$s := r - 1$$

Es gilt: $p = r + s - 1$

Das Zurechtstutzen von T erfolgt in zwei Phasen:





Erste Phase von der Wurzel nach unten (Breitensuche)
Betrachte Knoten x. Definiere:

- C(x) sind die Elemente $a \in A$ für die es kein $b \in A$ gibt und einen Knoten auf dem Pfad von der Wurzel von T zu x, an dem sich $a < b$ ergab.
- c(x) sind entsprechend die in x bekannten Minima in $\bar{A}$.
- s(x, a) ($a \in A$) ist die Anzahl der Elemente $\in \bar{A}$ (genauer: $\in$ c(x)) mit denen a auf dem Pfad oberhalb von x verglichen wurde.

Regeln für Phase 1:
Es werden in Knoten x die Elemente a und b verglichen:

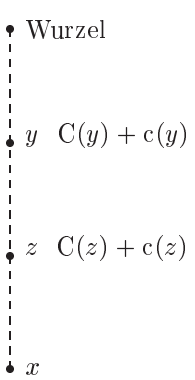
Regel 1.1: Falls $a \in A$ und $b \in A$ behalte x in T_A bei.

Regel 1.2: Falls $a \in \bar{A}$ und $b \in \bar{A}$ behalte x in T_A bei.

Regel 1.3: Sei o.B.d.A. $b \in \bar{A}$, $a \in A$. Ersetze den Unterbaum in T mit Wurzel x mit dem Unterbaum, dessen Wurzel das Kind von x ist, das dem Ergebnis entspricht (d.h. lasse den Vergleich $a < b$ aus, nach Voraussetzung $A < \bar{A}$).

Phase 1 läuft solange C(x) $> r$. Ein Knoten auf einem Pfad in T von der Wurzel, bei dem C(x) erstmals $= r$ wird, heißt *kritisch*.
Jeder Pfad in T von der Wurzel zu einem Blatt enthält genau einen kritischen Knoten.

Betrachte in T_A einen Pfad von der Wurzel zu einem kritischen Knoten x. Sei y ein Knoten auf diesem Pfad, z sein Kind.



Dann gilt:

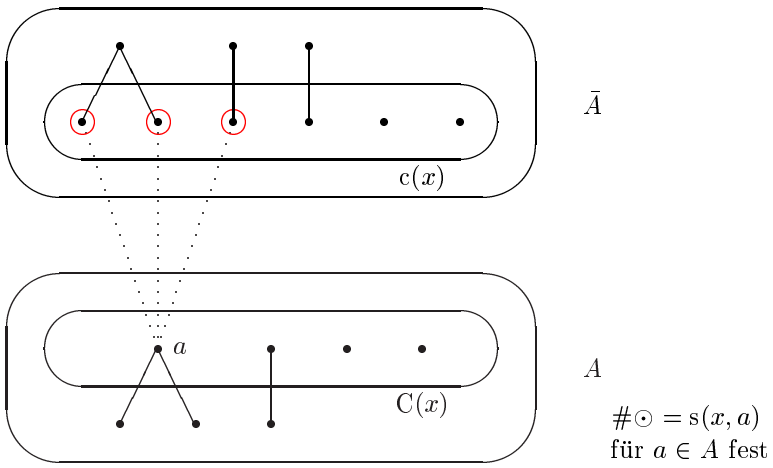
$$|\mathrm{C}(z)| + |\mathrm{c}(z)| \geq |\mathrm{C}(y)| + |\mathrm{c}(y)| - 1$$

Da $|\mathrm{C}(\text{Wurzel})| = |A| = i$ und $|\mathrm{c}(\text{Wurzel})| = |\bar{A}| = n - i$, müssen überhalb eines jeden kritischen Knoten x mindestens

$$i - |\mathrm{C}(x)| + n - i - |\mathrm{c}(x)| = n - r - |\mathrm{c}(x)|$$

Vergleiche erfolgen. Von jedem kritischen Knoten abwärts arbeitet der Gegenspieler nach einer Strategie für Phase zwei. Sei x ein solcher kritischer Knoten. Dann ist $|\mathrm{C}(x)| = r$.

Sei $a \in \mathrm{C}(x)$ ein Element mit minimalem $\mathrm{s}(x, a)$.



Fall 1: $\mathrm{s}(x, a) \geq s$.

Betrachte irgendeinen Pfad von der Wurzel durch x zu einem Blatt w. Jeder solche Pfad muß mindestens $n - 1$ Vergleiche enthalten, um answer(w) zu verifizieren: $\geq n - i$, für die answer(w) sich (direkt oder indirekt) als das kleinere Element ergibt, und $\geq i - 1$, wo es sich als das größere ergibt.
Damit sind $\geq (r-1)s$ Vergleiche redundant (nämlich alle die, die zwischen Elementen aus $\mathrm{C}(x) \setminus \{\text{answer}(w)\}$ und Elementen in $\bar{A}$ erfolgt sind).

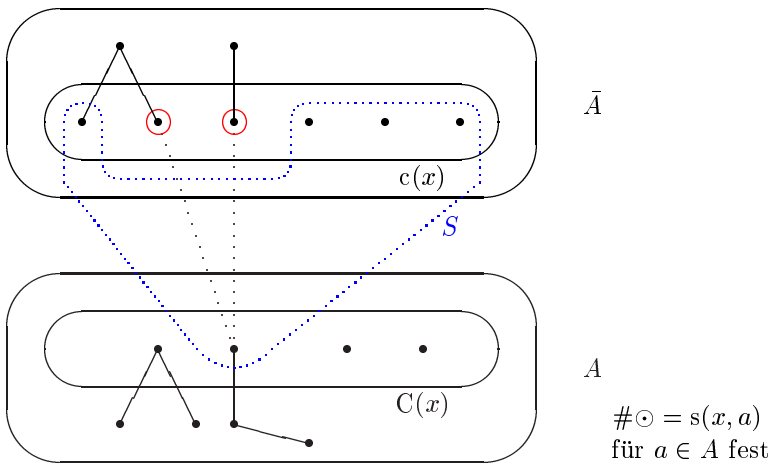
Also:

$$\begin{aligned}
\text{Tiefe}(T) &\geq n - 1 + s(r-1) \\
&= n - r - s - 1 + (s+1)r \\
&= n - r - s + 2 + \log \left[\frac{\binom{n}{i}}{n - i + 1} \right] \\
&> \log \left[\binom{n}{i} \frac{2^{n-p}}{n - i + 1} \right]
\end{aligned}$$

In diesem Fall folgt also die Behauptung direkt!

Fall 2: $s(x, a) < s$

Fall 2.1:



$|\mathrm{c}(x)| \geq p$. Sei $S := \mathrm{c}(x) \cup \{a\} \backslash$Elemente, die in $\mathrm{s}(x, a)$ gezählt werden. Der Gegenspieler antwortet nun so, daß das Ergebnis das kleinste Element in S wird.
Der Entscheidungsbaum T wird also gemäß folgender Regeln gestutzt (sei y der aktuelle Knoten und seien a und b die beiden Elemente, die in y verglichen werden):

Regel 2.1: falls $a, b \in S$, dann bleibt y erhalten.

Regel 2.2: andernfalls, sei oBdA. $a \in A \setminus S$ oder $b \in \bar{A} \setminus S$; ersetze y mit seinem Unterbaum durch das Kind von y und dessen Unterbaum, das der Antwort von $a < b$ entspricht.

Also, da überhalb des kritischen Knoten x keine zwei Elemente in S verglichen wurden, muß der Unterbaum von T_A unterhalb von x Tiefe $\geq |S| - 1$ haben.
Zusammen mit Phase 1 ergibt sich eine Tiefe von T_A:

$$\begin{aligned} &\geq n - r - |\mathrm{c}(x)| + |S| - 1 \\ &\geq n - r - |\mathrm{c}(x)| + |\mathrm{c}(x)| + 1 - (s-1) - 1 \\ &= n - r - s + 1 \\ &= n - p \end{aligned}$$

Fall 2.2: $|\mathrm{c}(x)| < p$. $S := \mathrm{C}(x)$
Regeln so, daß der Algorithmus als Antwort das Maximum von S bestimmt. Tiefe von T_A:

$$\begin{aligned} &\geq n - r - |\mathrm{c}(x)| + |S| - 1 \\ &\geq n - r - (p-1) + r - 1 \\ &= n - p \end{aligned}$$

Also enthält jedes T_A mindestens 2^{n-p} Blätter (von T). Alle T_A's zusammen enthalten $\geq \binom{n}{i} 2^{n-p}$ Blätter von T, wobei jedes Blatt von T höchstens $n - i + 1$ mal vorkommt: Sei w Blatt von T, dann ist little(w) eindeutig bestimmt und muß, falls T_A w auch enthält, $\subseteq A$ sein. Es gibt $\leq n - i + 1$ solche A's. □

Kapitel 3

Minimale Spannbäume

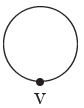
3.1 Grundlagen

Graphen: ungerichtet, einfach, zusammenhängend, gewichtet.

$$G = (V, E)$$

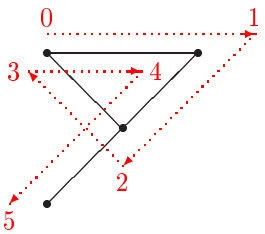
Knoten: $V = \{x, y, z, \dots\}$ oder $\{v_1, \dots, v_n\}$

Kanten: $\{v_i, v_j\}; i \neq j$

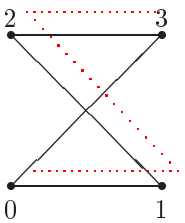


keine Schleifen erlaubt

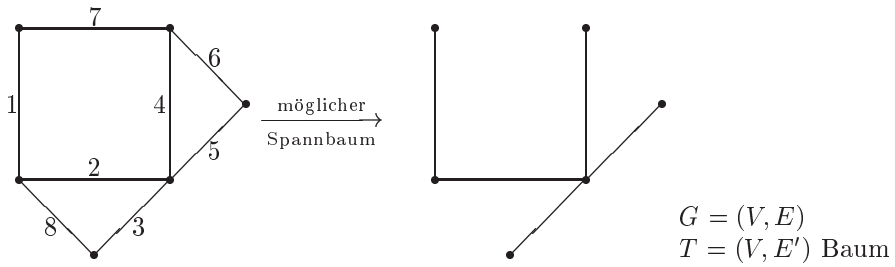
keine doppelten Kanten erlaubt



Weg 0-5 nicht knoteneinfach und nicht kanteneinfach



Weg 0-3 knoteneinfach und kanteneinfach



Anwendungen:

- Telekom: Telefonvermittlungen
- Leiterplatinen

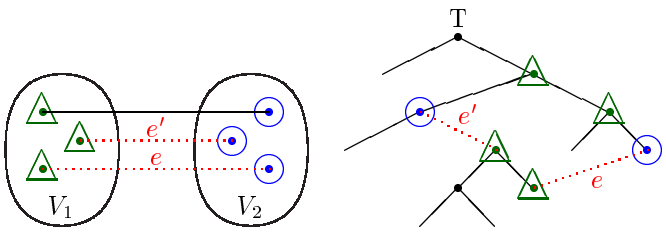
Zwei Prinzipien für die Konstruktion von minimalen Spannbäumen: „blaue“ Regel und „rote“ Regel (Tarjan).

Satz: Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph, w: $E \to \mathbb{R}$ eine Gewichtsfunktion, $C = (V_1, V_2)$ ein Schnitt (d.h. $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, $V_1 \neq \emptyset \neq V_2$). Sei $E_C = E \cap (V_1 \times V_2)$ die Menge der Kanten „über den Schnitt hinweg“. Dann gilt: (**„blaue“ Regel**)

1. Ist $e \in E_C$ die einzige Kante minimalen Gewichts (über alle Kanten in E_C), dann ist e in jedem minimalen Spannbaum für (G,w) enthalten.
2. Hat $e \in E_C$ minimales Gewicht (über alle Kanten in E_C), dann gibt es einen minimalen Spannbaum von (G,w), der e enthält.

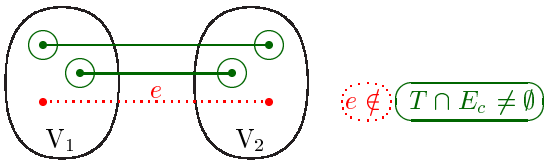
Beweis: durch Widerspruch

1. Sei T ein minimaler Spannbaum von (G,w), sei $e \in E_C$ die minimale Kante. Annahme: $e \notin T$. Da T Spannbaum $\Rightarrow T \cap E_C \neq \emptyset$.
   Sei $T \cap E_C = \{e_1, e_2, \ldots, e_k\}, k \geq 1$. Dann enthält $T \cup \{e\}$ einen eindeutig bestimmten Kreis (den sogenannten durch e bzgl. T bestimmten Fundamentalkreis). Dieser Kreis muß mindestens eine Kante $\in E_c \cap T$ enthalten, da die beiden Endpunkte von e auf verschiedenen Seiten des Schnitts C liegen.



   Sei $e' \in E_c \cap T$. Dann gilt nach Vorraussetzung w(e') >w(e). Also ist $T' := T - \{e'\} \cup \{e\}$ ein Spannbaum von G, der echt kleineres Gewicht als T hat, Widerspruch zu T ist minimaler Spannbaum.

2. Sei $e \in E_c$ minimal. Annahme: e kommt in <u>keinem</u> minimalen Spannbaum vor. Sei T ein beliebiger minimaler Spannbaum von (G,w).



$e \notin T \cap E_c \neq \emptyset$. Sei $e' \in E_c \cap T$ eine Kante auf dem durch e bezüglich T erzeugten Fundamentalkreis. Dann ist $T' = T - \{e'\} \cup \{e\}$ wieder ein Spannbaum von G, und es ist $\mathrm{w}(T') \leq \mathrm{w}(T)$. Also T' minimaler Spannbaum und $e \in T'$.

□

Satz: Sei $G = (V, E)$, $w : E \to \mathbb{R}$, ein gewichteter, zusammenhängender Graph. Dann gilt: (**„rote" Regel**)

1. Gibt es zu $e \in E$ einen Kreis C in G, der e enthält und $w(e) > w(e') \quad \forall e' \in C \setminus \{e\}$ erfüllt, dann kommt e in keinem minimalen Spannbaum vor.
2. Ist $c_1 = e_1, \ldots, e_k$ ein Kreis in G und $w(e_i) = \max\{w(e_j) | 1 \leq j \leq k\}$, dann gibt es einen minimalen Spannbaum, der e_i nicht enthält.

Beweis:

1. Nehmen wir an, da es einen minimalen Spannbaum T gibt, der $e = \{v_1, v_2\}$ enthält. Wenn wir e aus T entfernen, so zerfällt T in zwei nicht zusammenhängende Teilbäume T_1 und T_2 mit $v_i \in T_i$, $i = 1, 2$. Da aber e auf einem Kreis in G liegt, muß es einen Weg von v_1 nach v_2 geben, der e nicht benützt. Mithin gibt es eine Kante $\hat{e} \neq e$ auf diesem Weg, die einen Knoten in T_1 mit T_2 verbindet. Verbinden wir T_1 und T_2 entlang $\hat{e}$, so erhalten wir einen von T verschiedenen Spannbaum $\hat{T}$. Wegen $w(\hat{e}) < w(e)$ folgt $w(\hat{T}) < w(T)$. Ein Widerspruch.

2. Wir nehmen an, e_i liege in jedem minimalen Spannbaum (MSB) von G, und zeigen die Behauptung durch Widerspruch.
   Sei T ein beliebiger MSB von G. Entfernen wir e_i aus T, so zerfällt T in zwei nicht zusammenhängende Teilbäume T_1 und T_2. Da e_i auf einem Kreis $c_1 = e_1, \ldots, e_k$ in G liegt, können wir wie unter 1. e_i durch eine Kante e_j des Kreises c_1 ersetzen, die T_1 und T_2 verbindet. Dadurch erhalten wir einen von T verschiedenen Spannbaum $\tilde{T}$, der e_i nicht enthält. Da nach Voraussetzung $w(e_j) \leq w(e_i)$ gilt, folgt $w(\tilde{T}) \leq w(T)$ (und sogar $w(\tilde{T}) = w(T)$, da T nach Annahme ein MSB ist). Also ist $\tilde{T}$ ein MSB von G, der e_i nicht enthält, im Widerspruch zur Annahme, e_i liege in jedem MSB von G.

□

Literatur: R.E. Tarjan: "Data Structures and Network Algorithms", SIAM CBMS-NSF Reihe Bd. 44

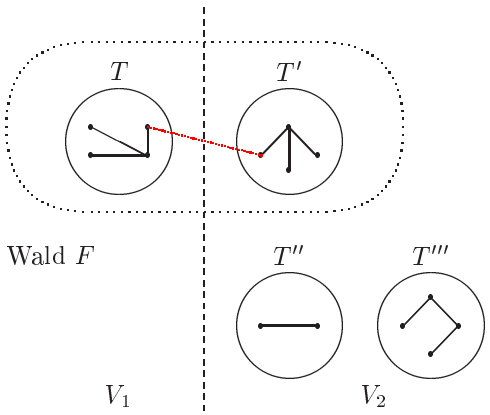
Generischer minimaler Spannbaum-Algorithmus:

Initialisiere Wald F von Bäumen, jedes $v \in V$ bildet einen Baum.

while F enthält mehr als einen Baum do

- wähle einen Baum $T \in F$ aus

- bestimme eine leichteste Kante $e = \{v, w\}$ aus T heraus
- sei $v \in T, w \in T'$
- vereinige T und T', füge e zum minimalen Spannbaum hinzu.

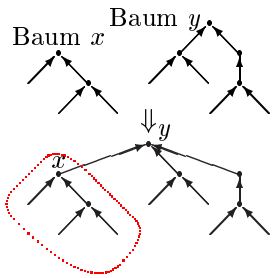


3.2 Kruskals Algorithmus

algorithm Kruskal (G,w):=

sortiere die Kanten nach aufsteigendem Gewicht in eine Liste L;
initialisiere $F = \{T_i, i = 1, \ldots, n\}$ Wald mit $T_i = \{v_i\}$;
    co $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$ oc
co implementiere T_i's als in-Trees mit den jeweiligen Wurzeln als
    Repräsentanten; allgemein: UNION-FIND-Problem oc
MSB:=$\emptyset$;
for $i := 1$ to Länge(L) do
    $\{v, w\} := L_i$;
    $x :=$ Baum $\in F$, der v enthält; co $x :=$ FIND(v) oc
    $y :=$ Baum $\in F$, der w enthält; co $y :=$ FIND(w) oc
    if $x \neq y$ then
        MSB:=MSB $\cup$ $\{\{v, w\}\}$;
        if $|x| \leq |y|$ then
            co Baum x enthält $\leq$ Knoten als Baum y oc

            vereinige x und y, indem die Wurzel von x zu einem neuen Kind der Wurzel von y wird;



            co $y :=$ UNION(x, y) oc
        else
            vereinige x und y, mit y neues Kind der Wurzel von x;
        fi
    fi
od

Korrektheit: $\surd$

Zeitkomplexität: (mit $n = |V|, m = |E|$)
$\mathcal{O}(m \log m + m \cdot \text{Kosten für } \underbrace{\text{FIND}}_{\mathcal{O}(\log n)} \text{ und } \underbrace{\text{UNION}}_{\mathcal{O}(1)}) = \mathcal{O}(m \log n)$
(wegen $m \geq n - 1$)

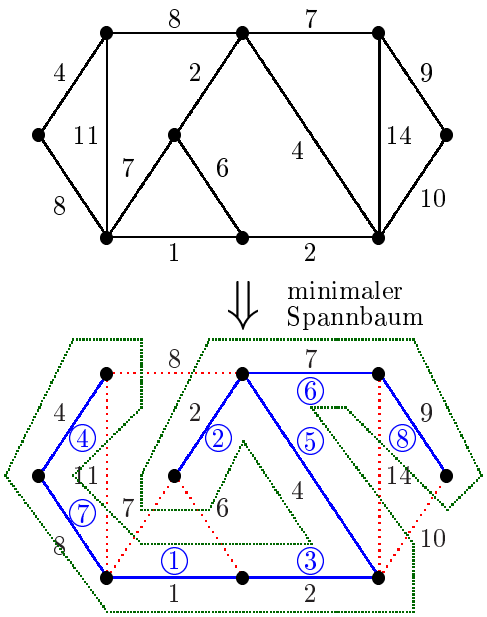
Tiefe des in-Trees $\leq \log n$:
Sei $T'' = \text{UNION}(T, T')$, $|T| \leq |T'|$.

Beweis: durch Induktion
ok für $|T| = 1$

$$\begin{aligned}
\text{Tiefe}(T'') &\leq \max\{\text{Tiefe}(T'), \text{Tiefe}(T) + 1\} \\
&\leq \max\{\log |T'|, 1 + \log |T|\} \\
&\leq \log(|T| + |T'|) \\
&= \log |T''|.
\end{aligned}$$

□

Beispiel für Kruskals Algorithmus:

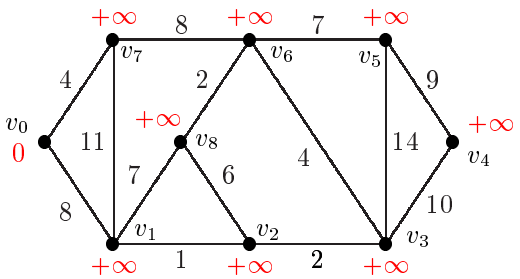


3.3 Prim's Algorithmus, erste Variante

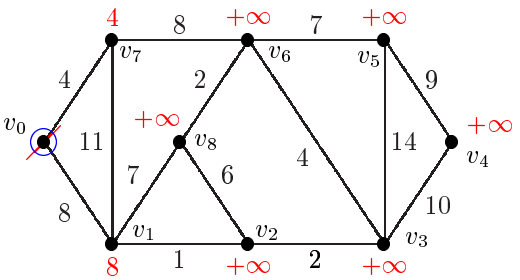
algorithm PRIM-MSB (G,w) :=

```
initialisiere Priority Queue PQ mit Knotenmenge V und Schlüssel +∞, ∀v ∈ V;
wähle Knoten r als Wurzel (beliebig);
Schlüssel[r] := 0;
Vorgänger[r] := nil;
while PQ ≠ ∅ do
    u := Find_Min(PQ);
    Delete_Min(PQ);
    for alle Knoten v, die in G zu u benachbart sind do
        if v ∈ PQ and w({u, v}) < Schlüssel[v] then
            Vorgänger[v] := u;
            Schlüssel[v] := w({u, v});
        fi
    od
od
```

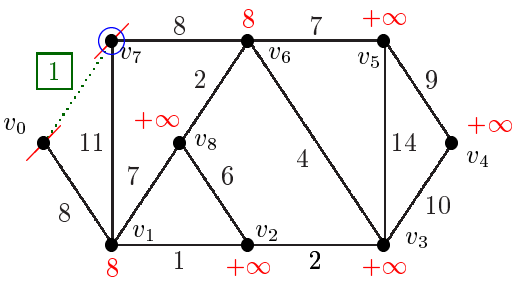
Beispiel für Prim's Algorithmus (erste Variante):



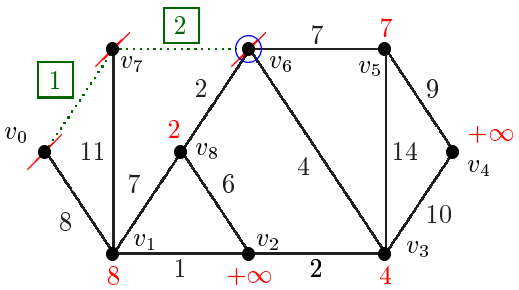
Ausgangszustand:
alle Schlüssel = $+\infty$
aktueller Knoten (= u): ◉
Startknoten: r (= v_0)



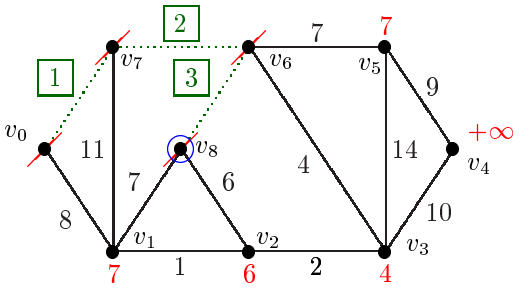
suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn
in PQ mit
w($\{u, v\}$) < Schlüssel[v]:
($v_1 = 8, v_7 = 4$)



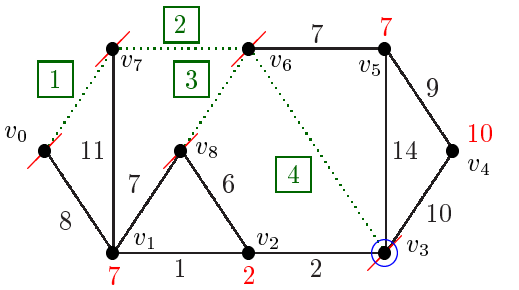
suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn
in PQ mit
w($\{u, v\}$) < Schlüssel[v]:
($v_6 = 8$)



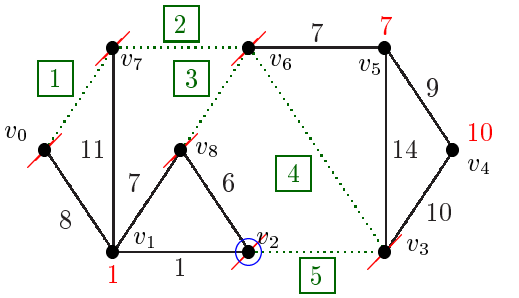
suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn
in PQ mit
w($\{u, v\}$) < Schlüssel[v]:
($v_3 = 4, v_5 = 7, v_8 = 2$)



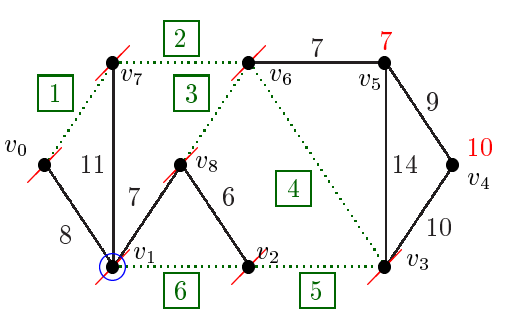
suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn
in PQ mit
w($\{u, v\}$) < Schlüssel[v]:
($v_1 = 7, v_2 = 6$)



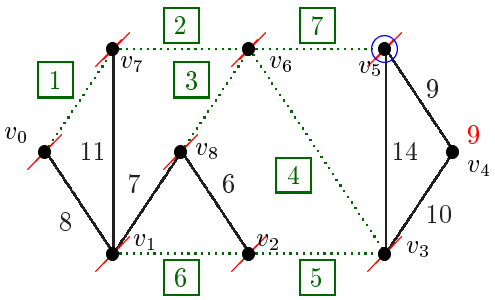
suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn
in PQ mit
w($\{u, v\}$) < Schlüssel[v]:
($v_2 = 2, v_4 = 10$)



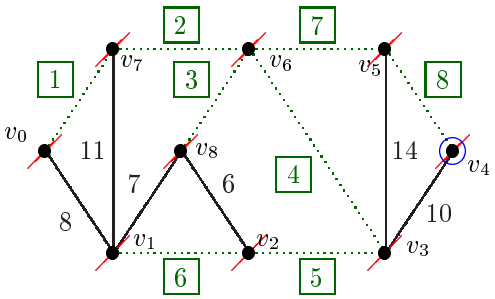
suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn
in PQ mit
w($\{u, v\}$) < Schlüssel[v]:
($v_1 = 1$)



suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn
in PQ mit
w($\{u, v\}$) < Schlüssel[v]:
solche Nachbarn existieren
nicht



suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn
in PQ mit
w($\{u, v\}$) < Schlüssel[v]:
($v_4 = 9$)



Endzustand:
suche u := Find_Min(PQ)
und entferne u aus PQ,
damit ist PQ leer und der
Algorithmus beendet

Implementierung der Priority Queue mittels Fibonacci-Heaps
Initialisierung: $\mathcal{O}(n)$
Find + Delete_Mins: $\mathcal{O}(n \log n)$ ($\leq n$ Stück)
Decrease_Keys: $\mathcal{O}(m)$ ($\leq m$ Stück)
$\rightsquigarrow$ Laufzeit: $\mathcal{O}(n) + \mathcal{O}(n \log(n)) + m \cdot \mathcal{O}(1) + m \cdot \underbrace{\mathcal{O}(1)}_{\text{sonstiger Overhead}} = \mathcal{O}(m + n \log\, n)$

Satz: Sei $G = (V, E)$ ein ungerichteter Graph (zusammenhängend, einfach) mit Kantengewichten w. Prim's Algorithmus berechnet, wenn mit Fibonacci-Heaps implementiert, einen minimalen Spannbaum von (G, w) in Zeit $\mathcal{O}(m+n \log n)$ (hier: $n = |V|,\ m = |E|$). Dies ist für $m = \Omega(n \log n)$ asymptotisch optimal.

3.4 Prim's Algorithmus, zweite Variante

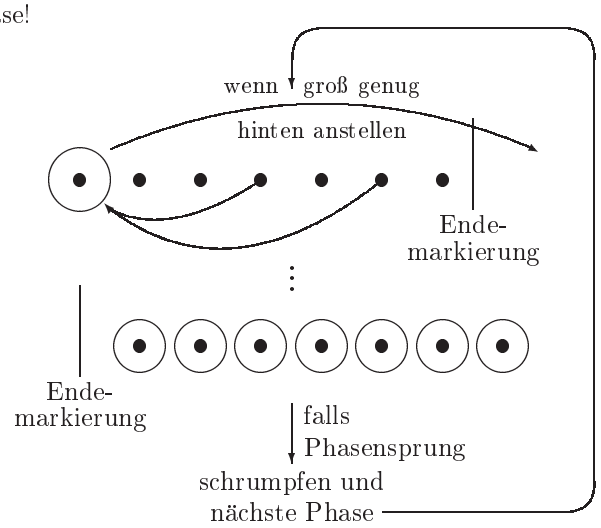
Idee: Lasse die Priority Queues nicht zu groß werden.
Sei $G = (V, E)$, $|V| = n$, $|E| = m$, w Gewichtsfunktion, k Parameter, der unten festgelegt wird.
Der Algorithmus wird in den folgenden Phasen abgearbeitet:

1. Initialisiere eine Schlange von Bäumen, jeder Baum anfangs (Super-) Knoten. Zu jedem Baum: Priority Queue (Fibonacci-Heap) mit den Nachbarn der Knoten im Baum, die selbst nicht im Baum sind. Schlüssel: Gewicht einer leichtesten Kante zu einem Knoten im Baum.

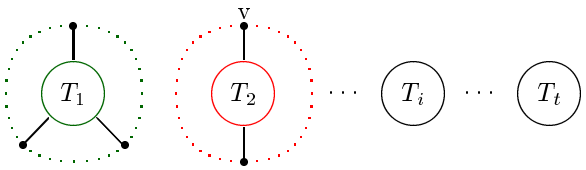
2. Markiere jeden Baum in der Schlange mit der Nummer der laufenden Phase

3. while vorderster Baum in der Schlange hat laufende Phasennummer do
   lasse ihn wachsen, solange seine Priority Queue weniger als
   k Elemente enthält (und noch was zu tun ist)
   if Priority Queue zu groß then
   füge Baum mit Phasennummer + 1 am Ende der Schlange ein
   fi
   od

4. Falls „Phasensprung“: Schrumpfe alle Bäume zu Superknoten, reduziere Kantenmenge (behalte zwischen zwei Knoten, jeweils nur die leichteste Kante)
5. nächste Phase!



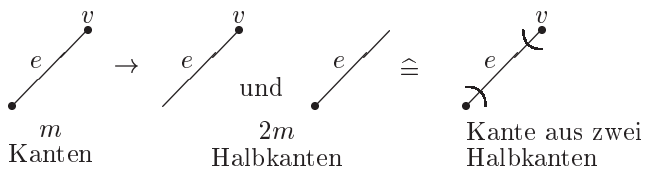
Betrachte Schlange von Bäumen:



(Arbeit pro Phase $\sim$ # Knoten, außer Vereinigung zweier Superknoten T_1 und T_2.)
Für jeden (Super-)Knoten v in T_2's Priority Queue:

1. $v \in T_1$: $\surd$ (nichts zu tun)
2. v in Priority Queue von T_1, Einfügen bzw. conditional Decrease_Key. Hilfsdatenstruktur: für alle Knoten in den Priority Queues ein Zeiger auf den Superknoten, in dessen Priority Queue der Knoten letztmals am Anfang der Schlange stand.
3. sonst: Einfügen

Betrachte Knoten mit „Halbkanten“: jede Halbkante kommt nur 1x in den Bäumen vor. Mit m Kanten ergeben sich $2m$ Halbkanten, d.h. Anzahl proportional zu m.



Zeitaufwand pro Phase: (+ Superknoten zu Beginn)

- Initialisierung: $\mathcal{O}(m)$
- Delete und Delete_Min: $\leq t$
- sonstige Priority Queue-Operationen, Overhead: $\mathcal{O}(m)$

Da die Priority Queues höchstens k Elemente enthalten, wenn darauf eine Priority Queue-Operation durchgeführt wird, sind die Kosten pro Phase

$$\mathcal{O}(t \log k + m)$$

Setze $k = 2^{\frac{2m}{t}}$ (damit $t \log k = m$). Damit sind die Kosten pro Phase $\mathcal{O}(m)$.

Wieviele Phasen? Sei t die Anzahl der Superknoten am Anfang einer Phase, t' diese Zahl zu Beginn der nächsten Phase. Sei a die durchschnittliche Anzahl ursprünglicher Knoten in den t Priority Queues zu Anfang der Phase, a' entsprechend zu Beginn der nächsten Phase.

Wir haben:

1. $a = \frac{2m}{t}$
2. $t' \leq \frac{2m}{k}$ (mit Ausnahme ev. der letzten Phase)

Also:

$$a' = \frac{2m}{t'} \geq k = 2^{\frac{2m}{t}} = 2^a$$

Für die erste Phase ist $a = \frac{2m}{n}$, a ist für jede Phase $\leq n - 1$. Also ist die Anzahl der Phasen

$$\leq 1 + \min\left\{ i; \log_2^{(i)}(n-1) \leq \frac{2m}{n} \right\}$$

Setzen wir $\beta(m,n) := \min\left\{ i; \log_2^{(i)} n \leq \frac{m}{n} \right\}$, dann gilt

$$\beta(m,n) \leq \log^* n \text{ für } n \leq m \leq \binom{n}{2}$$

(Bemerkung: $\log^* n = \min\{i; log_2^{(i)} \leq 1\}$, z.B. $\log^* 65536 = \log^* 2^{2^{2^2}} = 4$)

Satz: Für gewichtete, zusammenhängende Graphen mit n Knoten, m Kanten kann ein minimaler Spannbaum in Zeit $\mathcal{O}(\min\{m \cdot \beta(m,n), m + n \log n\})$ bestimmt werden.

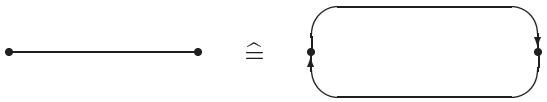
Euklidische minimale Spannbäume stellen ein Problem dar, für das es speziellere Algorithmen gibt. Literatur hierzu:
A.C.-C. Yao: Constructing Minimum Spanning Trees in k-dimensional spaces and related problems. SIAM Journal on Computing 11(4) p. 721–736 (1982)

Kapitel 4

Kürzeste Pfade

4.1 Grundlegende Begriffe

Betrachte Digraph $G = (V, A)$ oder Graph $G = (V, E)$.



Kante von ungerichtetem Graph

Entgegengesetzte (antiparallele) Kanten

Längenfunktion:

$$\mathrm{d} : A \longrightarrow \mathbb{R}^+ \text{ (bzw. } \longrightarrow \mathbb{R})$$

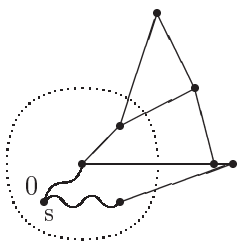
OBdA: $A = V^2$, $\mathrm{d}(x, y) = +\infty$ für Kanten, die eigentlich nicht vorhanden sind.
$\mathrm{dis}(v, w) :=$ Länge eines kürzesten Pfades von v nach $w \in \mathbb{R}^+ \cup \{+\infty\}$

Arten von kürzeste-Pfade-Problemen:

1. single-pair-shortest-path
2. single-source-shortest-path: gegeben G, d und $s \in V$, bestimme für alle $v \in V$ die Länge eines kürzesten Pfades von s nach v (bzw. einen kürzesten Pfad von s nach v)
3. all-pairs-shortest-path

4.2 Dijkstra's Algorithmus

Gegeben: $G = (V, A), (A = V^2)$, Distanzfunktion $d : A \to \mathbb{R}^+ \cup \{+\infty\}$. Startknoten s. G durch Adjazenzlisten dargestellt.

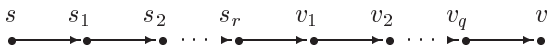


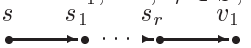
```
algorithm sssp:=
⌜
S := {s}; dis[s] := 0; initialisiere eine Priorityqueue, die alle Knoten v ∈ V \ {s}
enthält mit Schlüssel dis[v] := d(s, v);
for alle v ∈ V − {s} do
    from[v] := s
od
while S ≠ V do
    v := Find_Min(Q);
    Delete_Min(Q);
    S := S ∪ {v};
    for alle w ∈ V − S, d(v, w) < ∞ do
        if dis[v] + d(v, w) < dis[w] then
            Decrease_Key(Q, w, dis[v] + d(v, w));
                co Decrease_Key verändert dis[w] oc
            from[w] := v;
        fi
    od
od
⌟
```

Laufzeit (mit Fibonacci-Heaps):
Initialisierung: $\mathcal{O}(n)$
Sonstiger Aufwand: $\mathcal{O}(m)$ (z.B. Decrease_Key)
Zeitbedarf also $\mathcal{O}(m + n \log n)$ mit $n = |V|$ und $m :=$ Anzahl wirklicher Kanten

Korrektheit: Wir behaupten, daß in dem Moment, in dem ein $v \in V - \{s\}$ Ergebnis der Find_Min Operation ist, der Wert dis$[v]$ des Schlüssels von v gleich der Länge eines kürzesten Pfades von s nach v ist.

Beweis: durch Widerspruch: Sei $v \in V - \{s\}$ der erste Knoten, für den diese Behauptung nicht stimmt und sei



ein kürzester Pfad von s nach v, mit einer Länge $<$ dis$[v]$. Dabei sind $s_1, \ldots, s_r \in S$, $v \notin S$ [$r = 0$ und/oder $q = 0$ ist möglich]. Betrachte Pfad 

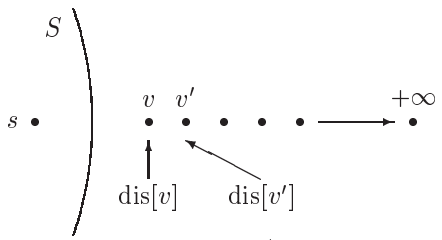
seine Länge ist $<$ dis$[v]$, also dis$[v_1]$ $<$ dis$[v]$. Widerspruch zur Wahl von v. □

Satz: Dijkstra's Algorithmus (mit Fibonacci-Heaps) löst das Single-Source-Shortest-Path-Problem in Zeit $\mathcal{O}(m + n \log n)$.

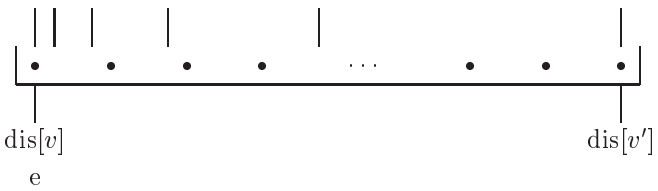
Bemerkung:

1. Verwendet man Dijkstra's Algorithmus mit d-Heaps, so erhält man Laufzeit $\mathcal{O}(m \log_{2+\frac{m}{n}} n)$
2. Mikkel Thorup (STOC '98): $\mathcal{O}(m)$
3. Dijkstra's Algorithmus mit Radix-Heaps (Akuja, Mehlhorn, Orlin, Tarjan; Faster Algorithms for the Shortest Path Problem. J ACM 37(1990), S. 213-223)

4.3 Dikstra's Algorithmus mit Radix-Heaps



d(v) $\in [0, ..., C]$ (alle Werte sind zwischen dis[v] und nC)



e

Einstufige Radix-Heaps:

Menge von $B = \lceil \log_2(C+1) \rceil + 2$ Buckets (jeder Bucket als doppelt verkettete Liste implementiert). Jeder Bucket hat Größe size, mit

$$\text{size}(i) = \begin{cases} 1 & \text{für } i = 1 \\ 2^{i-2} & \text{für } i = 2, \ldots, B-1 \\ nC+1 & \text{für } i = B \end{cases}$$

Beobachtung:

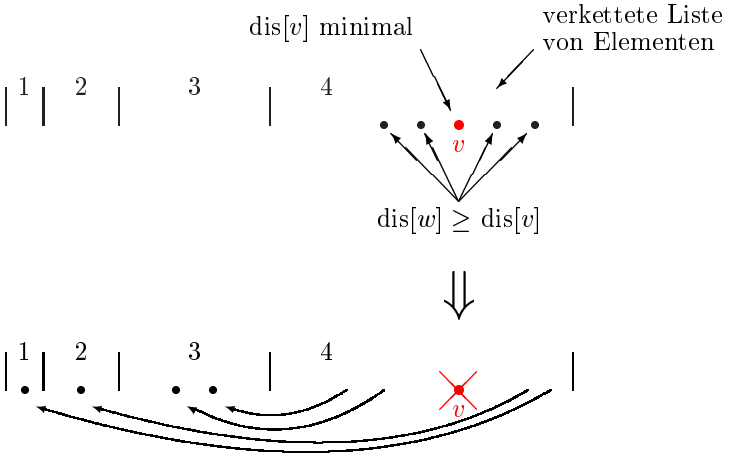
$$\sum_{i=1}^{j-1} \text{size}(i) \geq \min\{\text{size}(j), C+1\} \text{ für } j = 2, \ldots, B$$

Wir ordnen den Buckets Intervalle zu, indem wir für jeden Bucket i eine obere Intervallgrenze u(i) angeben. Das Intervall für Bucket i ist dann [u($i-1$) + 1,u(i)]. Setze zu Beginn u(0) := -1. Die u(i) ändern sich im Algorithmus, die size(i) bleiben konstant. Zu Beginn:

$$\left.\begin{array}{ll} \text{u}(i) & := 2^{i-1} - 1 \text{ für } i = 1, \ldots, B-1 \\ \text{u}(B) & := nC + 1 \end{array}\right\rangle \quad \begin{array}{l} \text{Größe des dem Bucket i} \\ \text{zugeordneten Intervalls:} \\ u(i) - u(i-1) \leq \text{size}(i) \end{array}$$

- Initialisierung: Einfügen aller Schlüssel in die Buckets: durchlaufe, für Schlüssel k, die Buckets B, $B-1$, ... bis wir Bucket j finden mit u(j) $< k$. Füge k in Bucket $j+1$ ein. Kosten: $n \cdot \log C$.
- Decrease_Key: wie bei Insert, beginne Suche mit momentanem Bucket des Schlüssels. Kosten für alle Decrease_Key Operationen: $\mathcal{O}(1)$ pro Decrease_Key $+\mathcal{O}(n \log C)$.
- Delete_Min: Suche von links den ersten nicht leeren Bucket j. Falls $j = 1$ ist alles ok (wegen size(1)=1). Falls $j \geq 2$: suche minimales Element v im Bucket j (v wird als Find_Min zurückgegeben und aus der Priority Queue gelöscht), verteile alle anderen Elemente im Bucket j auf die Buckets $1, \ldots, j-$

1, nachdem die u(i)'s wie folgt aktualisiert wurden:
u(0) := dis[v] − 1
u(1) := dis[v]
u(i) := min{u($i-1$) + size(i),u(j)} für $i = 2, \ldots, j-1$



Kosten für alle Delete_Mins: $\mathcal{O}(\log C)$ pro Delete_Min + $\mathcal{O}(n \log C)$.

Damit Gesamtkosten für Dijkstra's Algorithmus mit einstufigen Radix-Heaps:

$$\mathcal{O}(m + m + n\log C + n\log C + n\log C) = \mathcal{O}(m + n\log C)$$

Satz: Dijkstra's Algorithmus mit einfstufigen Radix-Heaps hat Zeitkomplexität $\mathcal{O}(m + n\log C)$.

Verbesserungen:

1. zweistufige Heaps: $\mathcal{O}(m + n \cdot \frac{\log C}{\log\log C})$.
2. mehrstufige Heaps + Fibonacci-Heaps: $\mathcal{O}(m + n\sqrt{\log C})$.

4.4 Floyd's Algorithmus für das All-pairs-shortest-path-Problem

(auch genannt „Kleene's Algorithmus")

Sei $G = (V, E)$ mit Distanzfunktion d ≥ 0 gegeben. Sei o.B.d.A. $V = \{v_1, \ldots, v_n\}$. Sei c_{ij}^k := Länge eines kürzesten Pfades $v_i = v^{(0)} - v^{(1)} - v^{(2)} - \ldots - v^{(r-1)} - v^{(r)} = v_j$ mit $v^{(l)} \in \{v_1, \ldots, v_k\}$ für $l = 1, \ldots, r-1$.

algorithm floyd:=
⌜
for alle (i,j) do $c_{ij}^{(0)} :=$ d$_{ij}$ od
for $k := 1$ to n do
  for alle (i,j), $1 \leq i,j \leq n$ do
    $c_{ij}^{(k)} := \min\left\{c_{ij}^{(k-1)},\ c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\right\}$
  od
od
⌟

Laufzeit: $\Theta(n^3)$.

Korrektheit:
Zu zeigen: $c_{ij}^{(k)}$ des Algorithmus $= c_{ij}^k$ (damit sind für $k = n$ die Kosten der kürzesten Pfade durch $c_{ij}^{(k)}$ gegeben).

Beweis: Richtig für $k = 0$. Induktionsschluß: Ein kürzester Pfad von v_i nach v_j mit inneren Knoten $\in \{v_1, \ldots, v_{k+1}\}$ enthält entweder v_{k+1} gar nicht als inneren Knoten, oder er enthält v_{k+1} genau einmal als inneren Knoten. Im ersten Fall wurde dieser Pfad also bereits für $c_{ij}^{(k)}$ betrachtet, hat also Länge $\geq c_{ij}^{(k)}$. Im zweiten Fall setzt er sich aus einem kürzesten Pfad P_1 von v_i nach v_{k+1} und einem kürzesten Pfad von P_2 von v_{k+1} nach v_j zusammen, wobei alle inneren Knoten von P_1 und $P_2 \in \{v_1, \ldots, v_k\}$ sind. Also ist die Länge des Pfades $= c_{i,k+1}^{(k)} + c_{k+1,j}^{(k)}$. □

Satz: Floyd's Algorithmus für das All-pair-shortest-path-Problem hat Zeitkomplexität $\mathcal{O}(n^3)$.

4.5 Min-Plus-Matrix-Produkt und Min-Plus-Transitive Hülle

Ring $\mathbb{R}(+, \bullet)$ — Gruppe, Halbgruppe

Semiring $\mathbb{N}(+, \bullet)$ — Halbgruppe, Halbgruppe

Semiring über $\mathbb{R} \cup \{\infty\}$ mit Operationen min und +, jeweils Halbgruppe.
Distributivgesetz $a + \min\{b, c\} = \min\{a + b, a + c\}$

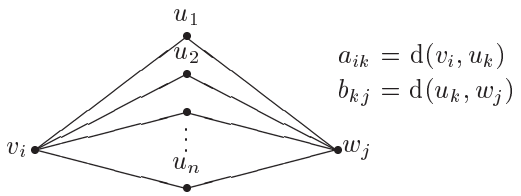
Normale Matrixmultiplikation:

$$A = (a_{ij})_{1 \leq i,j \leq n}, \quad B = (b_{ij})_{1 \leq i,j \leq n}, \quad I = (\delta_{ij})_{1 \leq i,j \leq n}$$

$$C = A \cdot B = (c_{ij})_{1 \leq i,j \leq n}, \quad c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Entsprechend für Min-Plus:

$$c_{ij} = \min\{a_{ik} + b_{kj}; 1 \leq k \leq n\}$$



Anwendung: $A = B$
kürzeste Wege von v_i nach w_j in einem Graph ($A = B$)

$$I_{\min,+} = \begin{pmatrix} 0 & & \infty \\ & \ddots & \\ \infty & & 0 \end{pmatrix}$$

Sei A Entfernungsmatrix, $A = (a_{ij})_{1 \leq i,j \leq n} = (\mathrm{d}(v_i, v_j))_{1 \leq i,j \leq n}$. Setze $a_{ii} = 0$ für $i = 1, \ldots, n$.

Betrachte A^2 (min-plus-Produkt):
$A^2 = (a_{ij}^{(2)})_{1 \le i,j \le n}$.
Dann ist $a_{ij}^{(2)}$ die Länge eines kürzesten Pfades von v_i nach v_j, der höchstens zwei Kanten enthält. Induktion: $a_{ij}^{(k)}$ ist die Länge des kürzesten Pfades von v_i nach v_j mit höchstens k Kanten.
Falls die $\mathrm{d}(v_i, v_j)$ alle ≥ 0, gibt es immer kürzeste Pfade, die $\leq n-1$ Kanten enthalten.

Alternative Lösung des All-Pairs-Shortest-Path-Problems: berechne A^{n-1} (min-plus). Es genügt auch $A^{2^{\lceil \log(n-1) \rceil}}$ (nicht $A^2, A^3, A^4, \ldots$ berechnen, sondern wiederholt quadrieren).

$A^* := \min\limits_{i \geq 0}\{A^i\}$ heißt Min-Plus-Transitive Hülle.
Bemerkung: min wird Komponentenweise gebildet. Wenn $d \geq 0$, dann $A^* = A^{n-1}$.

4.6 Boolesche Matrixmuliplikation und Transitive Hülle

Ersetze in 4.5. Distanzmatrix durch (boolesche) Adjazenzmatrix und ersetze min-plus durch $\vee, \wedge$, d.h.:

$$C = A \cdot B; \quad c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj}$$

Setze $a_{ii} = 1, 1 \leq i \leq n$
Dann gilt für A^k (boolesches Produkt, $A^0 = I$)

$$a_{ij} = \begin{cases} 1 & \text{falls es im Graphen einen Pfad von } v_i \text{ nach } v_j \\ & \text{bestehend aus } \leq k \text{ Kanten gibt} \\ 0 & \text{falls es im Graphen keinen Pfad von } v_i \text{ nach } v_j \\ & \text{bestehend aus } \leq k \text{ Kanten gibt} \end{cases}$$

Transitive Hülle

$$A^* := \bigvee_{i \geq 0} A^i$$

ist damit die Adjazenzmatrix der transitiven Hülle des zugrundeliegenden Graphen $(= A^{n-1})$.
Sei $\mathrm{M}(n)$ die Zeitkomplexität für das boolesche Produkt zweier $n \times n$ - Matrizen, $\mathrm{T}(n)$ die Zeitkomplexität für die transitive Hülle einer $n \times n$ booleschen Matrix.

Satz: Falls $\underbrace{\mathrm{T}(3n) \leq c\mathrm{T}(n)}_{\substack{\text{sicher erfüllt, falls} \\ \mathrm{T} \text{ polynomiell}}}$ und $\underbrace{\mathrm{M}(2n) \geq 4\mathrm{M}(n)}_{\substack{\text{sicher erfüllt, falls} \\ \mathrm{M}(n) \geq n^2}}$, dann gilt $\mathrm{T}(n) = \Theta(\mathrm{M}(n))$.

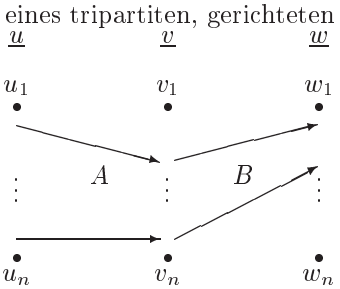
Beweis:

- Matrixmultiplikation $\prec$[1] transitive Hülle:
  Gegeben: A, B; Gesucht: $C = A \cdot B$ (boolesches Produkt); Setze:

$$L = \underbrace{\begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix}}_{3n} \Big\} 3n$$

  L ist Adjazenzmatrix eines tripartiten, gerichteten Graphen, denn:

	$\underline{u}$	$\underline{v}$	$\underline{w}$
$\underline{u}$	0	A	0
$\underline{v}$	0	0	B
$\underline{w}$	0	0	0



  Daher kann L^* leicht bestimmt werden:

$$L^* = \begin{pmatrix} I & A & AB \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}$$

  Also gilt: $\mathrm{M}(n) \leq \mathrm{T}(3n) = \mathcal{O}(\mathrm{T}(n))$.

- Transitive Hülle $\prec$ Matrixmultiplikation
  Gegeben: L $n \times n$ Boolesche Matrix; Gesucht: L^*; Annahme: n ist Zweierpotenz. Teile auf:

$$L = \underbrace{\begin{pmatrix} A & B \\ C & D \end{pmatrix}}_{\frac{n}{2}\ \frac{n}{2}} \begin{matrix} \} \frac{n}{2} \\ \} \frac{n}{2} \end{matrix} \quad ; \quad L^* = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

  Dann gilt:

$E =$	$(A \vee BD^*C)^*$	betrachte alle Pfade von der ersten Hälfte der Knoten zur ersten Hälfte der Knoten
$F =$	EBD^*	analog
$G =$	D^*CE	analog
$H =$	$D^* \vee GF$	analog

  Um L^* zu berechnen, benötigen wir zwei Transitive-Hülle-Berechnungen und sechs Matrixprodukte für Matrizen der Dimension $\frac{n}{2} \times \frac{n}{2}$, plus Aufwand für $\vee$, der $\leq c'n^2$ ist. Also:

$\mathrm{T}(n)$	$\leq 2\mathrm{T}(\frac{n}{2}) + 6\mathrm{M}(\frac{n}{2}) + c'n^2$	Behauptung:	$\mathrm{T}(n) \leq c\mathrm{M}(n)$
		Beweis:	durch Induktion: $n = 1 \surd$
	$\leq 2c\mathrm{M}(\frac{n}{2}) + 6\mathrm{M}(\frac{n}{2}) + c'n^2$	Vorraus.:	$\mathrm{M}(2n) \geq 4\mathrm{M}(n)$
			$\mathrm{M}(n) \geq n^2$
	$\leq \frac{1}{4}(2c + 6 + 4c')\mathrm{M}(n)$		
	$\leq c\mathrm{M}(n)$		

[1] $x \prec y$ bedeutet „reduziere x auf y“

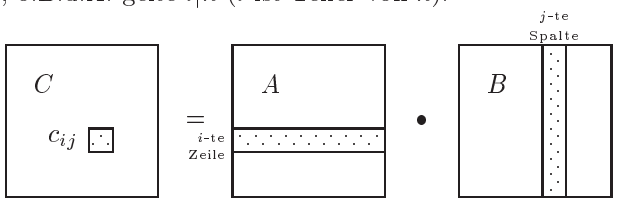
falls $c \geq \frac{1}{4}(2c + 6 + 4c')$, also falls $c \geq 3 + 2c'$.
Also $\mathrm{T}(n) = \mathcal{O}(\mathrm{M}(n))$.

□

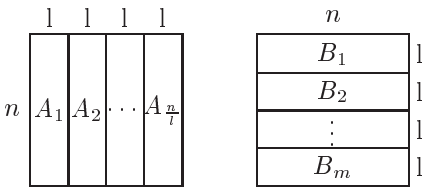
4.7 Der 4-Russen-Algorithmus für Boolesche Matrixmultiplikation

Autoren:
V.L. Avlazarov, E.A. Dinic, M.A. Kronod, I.A. Faradzev : "On economical construction of the transitive closure of a directed graph", Soviet Math. Doklady 11 (1970), p. 1209–1210.

Gegeben zwei Boolesche $n \times n$ Matrizen A, B; gesucht: $C = A \cdot B$.
Sei $l := \lfloor \log n \rfloor$, o.B.d.A. gelte $l | n$ (l ist Teiler von n).



Teile A auf (setze $m := \frac{n}{l}$):



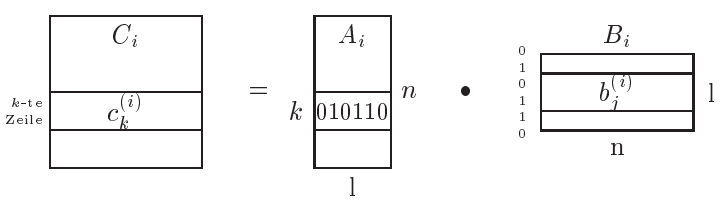
Sei $C_i := A_i \cdot B_i$ für $i = 1, \ldots, m$. Dann gilt

$$C = \bigvee_{i=1}^{m} C_i, \text{ da}$$

$$C = AB = \left(\bigvee_{i=1}^{m} A_i\right)\left(\bigvee_{i=1}^{m} B_i\right) = \bigvee_{\substack{1 \leq i \leq m \\ 1 \leq j \leq m}} A_i B_j = \bigvee_{i=1}^{m} A_i B_i,$$

da $A_i B_j = 0$ für $i \neq j$ (fasse A_i und B_j als $n \times n$ Matrizen auf mit 0 außerhalb des jeweiligen Streifens). Gegeben die C_i's, benötigen wir Zeit $\mathcal{O}(mn^2)$.

Betrachte eine Zeile von C_i:



$$c_k^{(i)} = \bigvee_{j=1}^{l} a_k^{(i)} \cdot b_j^{(i)}$$

Der Algorithmus berechnet einfach zunächst alle booleschen Linearkombinationen der Zeilen von B_i (Prozedur bcomb), und damit $c_k^{(i)}$ für jedes mögliche $a_k^{(i)}$.
Betrachte A, B und C als Matrizen von Zeilenvektoren:

$$A = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}, \quad B = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}, \quad C = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}$$

```
proc bcomb(int i) =
⌜
comb[0] := [0,...,0];
for j := 1 to 2^⌊log n⌋ − 1 do
    p := ⌊log j⌋;   co p Index der vordersten 1 von j oc
    comb[j] :=comb[j − 2^p] ∨ b_{(i−1)⌊log n⌋+p+1}
od
⌞
```

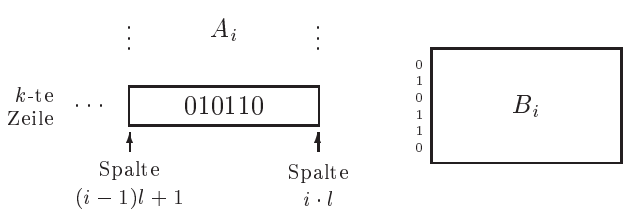
Zeitbedarf:

(a) sequentiell: $\mathcal{O}(n^2)$

(b) Vektoroperationen der Breite n: $\mathcal{O}(n)$

```
algorithm four-russians(array a, b, c) =
⌜
co die Matrizen a, b, c sind als Vektor von n Zeilenvektoren organisiert oc
const l = ⌊log n⌋ co wir nehmen l|n an oc;
array comb[0..2^{l−1}] of boolean-vector;

for i := 1 to n do c[i] := [0,...,0] od

for i := 1 to n/l do    co Berechne die C_i's oc
    bcomb(i);
    for j := 1 to n do
        co Bitmuster in der j-ten Zeile von A_i in Binärzahl wandeln oc
        nc := 0;
        for k := i · l downto (i − 1) · l + 1 do
            if a[j,k] then nc := nc + nc + 1 else nc := nc + nc fi
        od
        c[j] := c[j]∨comb[nc];
    od
od
⌞
```

Beispiel:



Zeitbedarf:

(a) sequentiell:
$\mathcal{O}\left(\frac{n}{l} \cdot \left(n^2 + n(l+n)\right)\right) = \mathcal{O}\left(\frac{n^3}{l}\right) = \mathcal{O}\left(\frac{n^3}{\log n}\right)$

(b) Vektorrechner der Breite n (Interpretation eines Bitintervalls als Zahl in $\mathcal{O}(1)$ Zeit):
$\mathcal{O}\left(\frac{n}{l} \cdot (n + n(1+1))\right) = \mathcal{O}\left(\frac{n^2}{\log n}\right)$ (Vektoroperationen)

Satz: Der 4-Russen Algorithmus berechnet das Produkt zweier Boolescher Matrizen sequentiell in Zeit $\mathcal{O}\left(\frac{n^3}{\log n}\right)$ bzw. mit $\mathcal{O}\left(\frac{n^2}{\log n}\right)$ Vektoroperationen.

Beweis: $\surd$ (s.o.) □

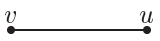
4.8 Transitive Hülle und DFS[2]

Tiefensuche(DFS):

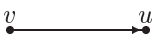
```
proc DFS-visit(node v) =
⌜
markiere v;
num[v] := ++count;
for alle Nachbarn u von v do
    if u unmarkiert then DFS-visit(u) fi
od
⌞

algorithm DFS =
⌜
for alle Knoten v do unmarkiere(v); num(v) := 0 od;
count := 0;
for alle Knoten v do
    if v unmarkiert then DFS-visit(v) fi
od
⌞
```

Bemerkung: Funktioniert für Graphen und Digraphen.

Nachbarn in Graphen und Digraphen:



u Nachbar von v
v Nachbar von u

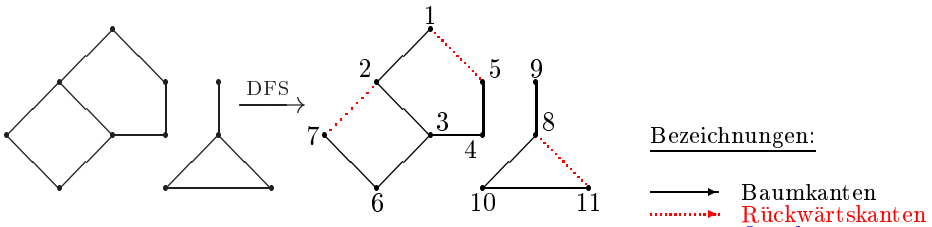


u Nachbar von v

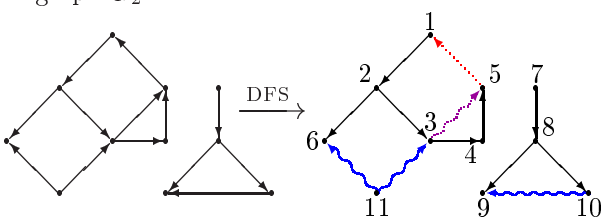
[2] DFS: Depth First Search

Beispiele für DFS:

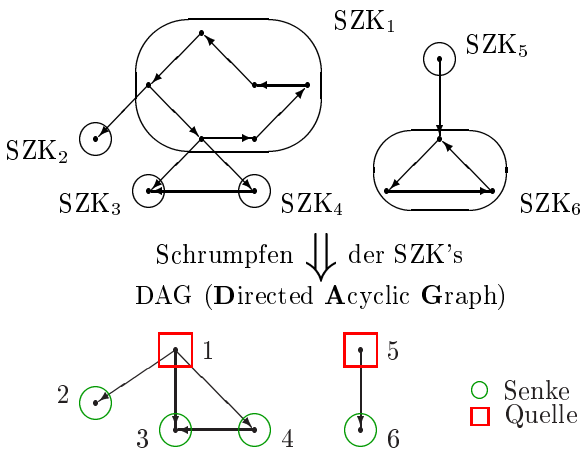
Graph G_1:



Digraph G_2:



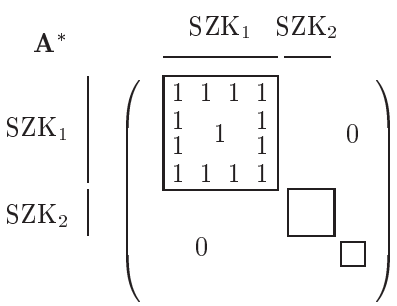
Bemerkung: Alle Baumkanten zusammen ergeben den DFS-Wald (Spannwald).



Bemerkung: SZK: starke Zusammenhangskomponente (in Digraphen)

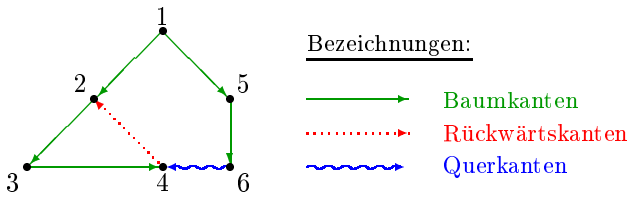
DFS in ungerichteten zyklischen Graphen, n Knoten, m Kanten, $n-1$ Baumkanten, $m-n+1$ Rückwärtskanten.

Zeitaufwand: $\mathcal{O}(n+m)$



A^* aus DFS mit Aufwand $\Theta(n^2)$ in ungerichteten Graphen (Anm.: SZK = ZK).

DFS in gerichteten Graphen (Digraphen), n Knoten, m Kanten, Baum-, Vorwärts-, Rückwärts- und Querkanten:



Zeitaufwand: $\mathcal{O}(n+m)$
Es gilt: u ist von v aus auf einem gerichteten Pfad erreichbar gdw. u Knoten in dem DFS-Baum (DFS-visit(v)) mit Wurzel v ist.
Also: Transitive Hülle in Zeit $\mathcal{O}(n \cdot (m+n))$ bzw. $\mathcal{O}(n \cdot m)$. Sehr gut für dünne Graphen (also solche mit relativ wenigen Kanten).

4.9 Ein besserer Algorithmus für das All-pairs-shortest-distance (bzw. paths)-Problem in ungewichteten Digraphen

$(d_{ij})_{0 \le i,j < n}$ Distanzmatrix
D^* Distanzmatrix für Transitive Hülle
} alle Kantenlängen $= 1$

$D^{(l)} = (d_{ij}^{(l)})_{o \le i,j < n}$ mit $d_{ij}^{(l)} = \begin{cases} d_{ij}^* \text{ falls } d_{ij}^* \le l \\ \infty \text{ sonst} \end{cases}$

algorithm 1 =
⌜

co Sei $A = (a_{ij})_{0 \le i,j < n}$ mit $a_{ij} = \begin{cases} 1 \text{ falls } d_{ij} \le 1 \\ 0 \text{ sonst} \end{cases}$ oc
co Sei B die Boolsche Einheitsmatrix oc
for $l := 2$ to $r+1$ do
  $B := A \cdot B$;
  for $0 \le i,j < n$ do
    if $b_{ij} = 1$ then $d_{ij}^{(l)} := l$ else $d_{ij}^{(l)} := \infty$ fi;
    if $d_{ij}^{(l-1)} \le l$ then $d_{ij}^{(l)} := d_{ij}^{(l-1)}$ fi
  od
od
⌞

Dieser Algorithmus berechnet $D^{(1)} = D, D^{(2)}, D^{(3)}, \ldots, D^{(r)}$.
Sei ω eine Zahl ≥ 2, so daß die Matrixmultiplikation in der Zeit $\mathcal{O}(n^\omega)$ durchgeführt werden kann (Winograd/Coppersmith: $\omega \le 2,376$).

Zeitaufwand für Algorithmus 1: $\mathcal{O}(rn^\omega)$

algorithm apsd =

⌜

Berechne, mit Algorithmus 1, $D^{(l)}$ für $l = 1, \ldots, r$;
$l := r$;
for $s := 1$ to $\left\lceil \log_{\frac{3}{2}} \frac{n}{r} \right\rceil$ do
    for $i := 0$ to $n-1$ do
        finde in der i-ten Zeile von $D^{(l)}$ den Wert d, $\lceil \frac{l}{2} \rceil \leq d \leq l$, so daß d in der i-ten Zeile am wenigsten oft vorkommt;
        $S_i :=$ Menge der Spaltenindizes, in denen dieses d in der i-ten Zeile von $D^{(l)}$ vorkommt;
    od
    $l_1 := \lceil \frac{3}{2} l \rceil$;
    for $0 \leq i, j < n$ do
        if $S_i \neq \emptyset$ then $m_{ij} := \min_{k \in S_i} \{d_{ik}^{(l)} + d_{kj}^{(l)}\}$ else $m_{ij} := \infty$ fi;
        if $d_{ij}^{(l)} \leq l$ then $d_{ij}^{(l_1)} := d_{ij}^{(l)}$
        elseif $m_{ij} \leq l_1$ then $d_{ij}^{(l_1)} = m_{ij}$
        else $d_{ij}^{(l_1)} := \infty$
        fi
    od
    $l := l_1$;
od

⌞

apsd berechnet:
Zunächst $D^{(1)}, \ldots, D^{(r)}$, dann $D^{(l)}$ für

$$l = r, \left\lceil \frac{3}{2} r \right\rceil, \left\lceil \frac{3}{2} \left\lceil \frac{3}{2} r \right\rceil \right\rceil, \ldots, \underbrace{n'}_{\geq n}$$

Da für d $\frac{l}{2}$ Werte zur Auswahl stehen, gilt:

$$|S_i| \leq \frac{2n}{l}$$

Damit ist die Laufzeit:

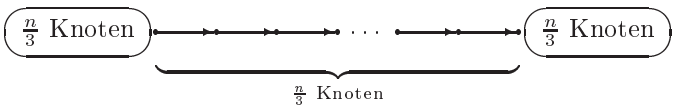
$$\mathcal{O}\left(rn^{\omega} + \sum_{s=1}^{\left\lceil \log_{\frac{3}{2}} \frac{n}{r} \right\rceil} \frac{n^3}{\left(\frac{3}{2}\right)^s \cdot r} \right) = \mathcal{O}\left(rn^{\omega} + \frac{n^3}{r} \right)$$

Setze r so, daß die beiden Summanden gleich sind, also $r = n^{\frac{3-\omega}{2}}$, dann ergibt sich damit für apsd die Laufzeit $\mathcal{O}\left(n^{\frac{3+\omega}{2}}\right)$.

Satz: Das All-Pairs-Shortest-Distance-Problem für Digraphen mit Kantenlänge = 1 läßt sich in Zeit $\mathcal{O}(n^{\frac{3+\omega}{2}})$ lösen (ω Exponent für Matrixmultiplikation).

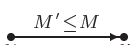
Beweis: $\surd$ □

Bemerkung: APSP: Problem: Ausgabegröße kann $\Omega(n^3)$ sein:



Kürzeste Pfade-Nachfolger-Matrix $N \in [0, \ldots, n-1]^{n \times n}$
n_{ij} = Nummer des zweiten Knoten auf „dem“ kürzesten Pfad von Knoten i zu Knoten j. So eine Matrixdarstellung für die kürzesten Pfade zwischen allen Knotenpaaren kann in Zeit $\mathcal{O}(n^{\frac{3+\omega}{2}} \cdot \log^c n)$ für ein $c > 0$ bestimmt werden.

Korollar: Für Digraphen mit Kantenlängen $\in \{1, 2, \ldots, M\}, M \in \mathbb{N}$, kann APSD in Zeit $\mathcal{O}((Mn)^{\frac{3+\omega}{2}})$ gelöst werden.

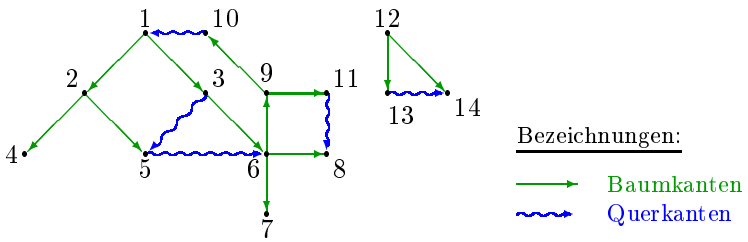
Idee: Ersetze 

durch

4.10 Ein Algorithmus für die transitive Hülle in Digraphen mit linearer erwarteter Laufzeit

Die Wahrscheinlichkeit eines Digraphen mit n Knoten und m Kanten ist (nur) eine Funktion von n und m. Daraus folgt (wir lassen der Einfachheit halber Schleifen (=Kanten $v \to v$) zu), daß jede Kante (u, v) mit Wahrscheinlichkeit $\frac{m}{n^2}$ vorhanden ist, falls wir Graphen mit n Knoten und m Kanten betrachten.
Einschub: Breitensuche (BFS): Schlange, queue, FIFO:

Durchlaufe Graphen, indem wir, solange möglich den vordersten Knoten v aus der queue nehmen, ihn behandeln und die Liste seiner noch nicht behandelten Nachbarn in die queue hinten einfügen.
Beispiel:



algorithm exp-lin-transitive-closure

(C.P. Schnorr, An Algorithm for Transitive Closure with Linear Expected Time, SIAM Journal on Computing 7 (1978), pages 127-133)
Phasen:

0. Konstruiere die linear geordneten Adjazenzlisten $L_i^r, i = 1, \ldots, n$, des Graphen G^r (entsteht aus G durch Umkehrung aller Knoten).
   Beispiel:

L_1	=	4	1	2	L_1^r	=	1	2	4
L_2	=	1	4	3	L_2^r	=	1	3	4
L_3	=	3	2		L_3^r	=	2	3	
L_4	=	1	4	2	L_4^r	=	1	2	4

Ersetze ebenfalls alle L_i durch L_i^{rr} ($\rightarrow$ sortierte Adjazenzlisten)

1. Berechne für jeden Knoten i in BFS-Art eine Liste S_i von von i aus erreichbaren Knoten, so daß (i) oder (ii) gilt:

   (i) $|S_i| < \lfloor \frac{n}{2} \rfloor + 1$ und S_i enthält alle von i aus erreichbaren Knoten

   (ii) $|S_i| = \lfloor \frac{n}{2} \rfloor + 1$

2. Entsprechende Listen P_i von Vorgängern von i

3. for $i := 1$ to n do
   $L_i^* := S_i \cup \{j; i \in P_j\} \cup \{j; |S_i| = |P_j| = \lfloor \frac{n}{2} \rfloor + 1\}$;
   od

   Bilde $(L_i^*)^{rr}$ für $i = 1, \ldots, n$

Korrektheit: $j \in L_i^*$ gdw. (i, j) in transitiver Hülle.

Satz: Sei die Wahrscheinlichkeit eines Graphen (lediglich) eine Funktion der Anzahl n seiner Knoten und der der Anzahl m seiner Kanten. Dann ist der Erwartungswert für die Laufzeit des Schnorr'schen Algorithmus $\mathcal{O}(n + m^*)$, wobei m^* der Erwartungswert für die Anzahl der Kanten in der transitiven Hülle ist.

Beweis: Das Durchlaufen des Graphen (von V_1 aus für die Konstruktion von S_1) in BFS-Manier liefert eine Folge $(a_t)_{t \geq 1}$ von Knoten. Sei $L_{\sigma(\nu)}$ die ν-te Adjazenzliste, die in der BFS erkundet wird,

$$L_{\sigma(\nu)} = (a_t; \mathrm{h}(\nu) < t \leq \mathrm{h}(\nu + 1))$$

Sei $\Delta L_{\sigma(\nu)}$ die Menge der a_t in $L_{\sigma(\nu)}$, und sei

$$S_i(t) := \{a_1, a_2, \ldots, a_t\}$$

Wie bereits gezeigt, ist

$$\mathrm{WS}(j \in L_i) = \frac{m}{n^2}, \quad \forall i, j$$

Alle n-Tupel von geordneten Listen $L_1, L_2, ..., L_n$ mit

$$m = \sum_{i=1}^{n} |L_i|$$

sind aufgrund der Voraussetzungen über die Wahrscheinlichkeitsverteilung gleich wahrscheinlich.

Also:
Beobachtung 1: Die Mengen $\Delta L_{\sigma(\nu)}, \nu = 1, 2, \ldots$ sind (paarweise) unabhängig.
Beobachtung 2: Die $\Delta L_{\sigma(\nu)}$ sind gleichverteilt, d.h. für alle $A \subseteq \{1, \ldots, n\}$ gilt:

$$\mathrm{WS}(\Delta L_{\sigma(\nu)} = A) = (\frac{m}{n^2})^{|A|}(1 - \frac{m}{n^2})^{n-|A|}$$

Lemma: Sei $A \subseteq \{1, \ldots, n\}$ mit $|A| = k$. Sei $(\bar{a}_t)_{t \geq 1}$ eine Folge von unabhängigen gleichverteilten Zufallsvariablen mit Wertemenge $\{1, \ldots, n\}$. Dann gilt:

$$\min\{t; \bar{a}_t \notin A\} \quad \text{hat Erwartungswert} \quad 1 + \frac{k}{n-k}.$$

Beweis: $\mathrm{WS}(\bar{a}_1, \ldots, \bar{a}_r \in A \text{ und } \bar{a}_{r+1} \notin A) = \left(\frac{k}{n}\right)^r \cdot \left(1 - \frac{k}{n}\right)$.
Also ist der Erwartungswert für $\min\{t; \bar{a}_t \notin A\}$:

$$\begin{aligned}
&1 + \sum_{r=0}^{\infty} r \left(\frac{k}{n}\right)^r \left(1 - \frac{k}{n}\right) \\
&= 1 + \frac{k}{n}\left(1 - \frac{k}{n}\right) \sum_{r \geq 1} r \left(\frac{k}{n}\right)^{r-1} \\
&= 1 + \frac{k}{n}\left(1 - \frac{k}{n}\right) \frac{1}{\left(1 - \frac{k}{n}\right)^2} \\
&= 1 + \frac{\frac{k}{n}}{1 - \frac{k}{n}} \\
&= 1 + \frac{k}{n-k}
\end{aligned}$$

□

Anmerkung:

$$\sum_{r=0}^{\infty} r z^{r-1} = \frac{d}{dz} \frac{1}{1-z} = \frac{1}{(1-z)^2}$$

Lemma: Sei $(\bar{a}_t)_{t \geq 1}$ wie oben, $k \leq \frac{n}{2}$. Dann hat $\min\{t; |\bar{a}_1, \bar{a}_2, \ldots, \bar{a}_t| > k\}$ Erwartungswert $\leq \frac{3}{2}(k+1)$.

Beweis: Wegen der Additivität des Erwartungswertes gilt: Der gesuchte Erwartungswert ist

$$\begin{aligned}
&\leq \sum_{\nu=0}^{k} \left(1 + \frac{\nu}{n-\nu}\right) \\
&\leq k + 1 + \sum_{\nu=1}^{k} \frac{\nu}{\frac{n}{2}} \\
&\leq k + 1 + \frac{k(k+1)}{n} \\
&\leq \frac{3}{2}(k+1)
\end{aligned}$$

□

Wenn wir jedes $L_{\sigma(\nu)}$ in sich zufällig permutieren, erhalten wir eine Folge $(\bar{a}_t)'_{t \geq 1}$, von Zufallsvariablen, so daß

$$|\{a_1, \ldots, a_t\}| = |\{a'_1, \ldots, a'_t\}| \text{ für } t = \mathrm{h}(\nu), \forall \nu$$

Da die a'_t innerhalb eines jeden Intervalls $\mathrm{h}(\nu) < t \leq \mathrm{h}(\nu+1)$ paarweise verschieden sind, gilt für sie die vorhergehende Abschätzung erst recht. Daher sind $(|S_1(t)|)_{t \geq 1}$ und $(|S_1(t)'|)_{t \geq 1}$ gleichverteilte Folgen von Zufallsvariablen, denn dies gilt zunächst für alle Zeitpunkte der Form $t = \mathrm{h}(y)$; da aber für diese Zeitpunkte $S_1(t) = S'_1(t)$

ist und h() zufällig verteilt ist, ist auch die Verteilung der beiden Folgen zwischen diesen Intervallgrenzen identisch.

Sei s der Erwartungswert und sei $|S_i|$ die tatsächliche Größe von S_i nach der Phase 1. Dann

$$s = \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor + 1} k \cdot \mathrm{WS}(|S_i| = k)$$

Die erwartete Anzahl der Schritte (und damit die Kosten) in der BFS sei q. Dann gilt:

$$q \leq \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor + 1} \mathrm{WS}(|S_i| = k)\frac{3}{2}k = \frac{3}{2}s.$$

Also ist der Aufwand des Algorithmus für die Phasen 1 und 2 im Erwartungswert $\leq 3sn$. Da $ns = m^*$, und da die Kosten der anderen Phasen offensichtlich durch $\mathcal{O}(n + m + m^*)$ beschränkt sind, folgt die Behauptung. □

4.11 Matrizenmultiplikation à la Strassen

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}; \quad c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j}$$

Bilde:

$$\begin{aligned} m_1 &:= (a_{12} - a_{22})(b_{21} + b_{22}) \\ m_2 &:= (a_{11} + a_{22})(b_{11} + b_{22}) \\ m_3 &:= (a_{11} - a_{21})(b_{11} + b_{12}) \\ m_4 &:= (a_{11} + a_{12})b_{22} \\ m_5 &:= a_{11}(b_{12} - b_{22}) \\ m_6 &:= a_{22}(b_{21} - b_{11}) \\ m_7 &:= (a_{21} + a_{22})b_{11} \end{aligned}$$

Dann gilt:

$$\begin{aligned} c_{11} &:= m_1 + m_2 - m_4 - m_6 \\ c_{12} &:= m_4 + m_5 \\ c_{21} &:= m_6 + m_7 \\ c_{22} &:= m_2 - m_3 + m_5 - m_7 \end{aligned}$$

Sei $n = 2^k$ und $\mathrm{M}(n)$ die Anzahl arithmetischer Operationen (Mult, Add, Sub), die Strassen's Algorithmus bei $n \times n$ Matrizen benötigt:

$$\begin{aligned} \mathrm{M}(1) &= 1 \\ \mathrm{M}(n) &= 7\mathrm{M}\left(\tfrac{n}{2}\right) + 18\left(\tfrac{n}{2}\right)^2 \end{aligned}$$

Expansion dieser Rekursionsgleichung →

$$\mathrm{M}(n) = 7^{k+1} - 6n^2 = 7 \cdot 2^{k \log_2 7} - 6n^2 = 7n^{\log_2 7} - 6n^2 < 7n^{2,81} - 6n^2$$

procedure Matmult(A, B, n) =
⌜
co A und B sind $n \times n$ Matrizen oc
if $n < 16$ then berechne $A \cdot B$ mit klassischer Methode
else if n gerade then
spalte A und B in je 4 $\frac{n}{2} \times \frac{n}{2}$ Matrizen auf und wende Strassen's Formeln an. Führe die Multiplikationen rekursiv mit Matmult aus.
else
spalte A und B in je eine $(n-1) \times (n-1)$ Matrix (A_{11}, B_{11}) und drei verbleibende Blöcke auf. Wende Matmult rekursiv auf A_{11}, B_{11} an und berechne die anderen Produkte mit klassischer Methode.
fi
fi
⌞

Satz: Der Matmult-Algorithmus hat folgende Eigenschaften:

1. Für $n < 16$ ist die Anzahl der arithmetischen Operationen genauso hoch wie bei der klassischen Methode.
2. Für $n \geq 16$ ist die Anzahl der arithmetischen Operationen echt kleiner als bei der klassischen Methode.
3. Matmult braucht nie mehr als $4.8n^{\log_2 7}$ arithmetische Operationen.

Beweis:

1. $\surd$
2. Sei n gerade und seien A, B zwei $n \times n$ Matrizen. Wir wenden Strassen's Formeln an, führen aber die 7 Multiplikationen der $\frac{n}{2} \times \frac{n}{2}$ Matrizen mit klassischer Methode aus.
   Gesamtzahl arithmetischer Operationen:
   $$7\left(2\left(\frac{n}{2}\right)^3 - \left(\frac{n}{2}\right)^2\right) + 18\left(\frac{n}{2}\right)^2 = \frac{7}{4}n^3 + \frac{11}{4}n^2$$
   Dies ist besser als die klassische Methode, falls:
   $$\frac{7}{4}n^3 + \frac{11}{4}n^2 < 2n^3 - n^2 \Leftrightarrow \frac{15}{4}n^2 < \frac{1}{4}n^3 \Leftrightarrow 15 < n \Leftrightarrow n \geq 16$$
   Sei n ungerade. Multiplizieren wir $A \cdot B$ mit klassischer Methode, so auch $A_{11} \cdot B_{11}$. Also brauchen wir durch Anwendung von Strassen's Formeln auf die $(n-1) \times (n-1)$ Matrizen ($n-1$ gerade) weniger Operationen, wenn $n - 1 \geq 16$ ist.
   Durch Induktion über die Anzahl der rekursiven Anwendungen von Strassen's Formeln folgt die Behauptung.
3. Sei $\mathrm{M}'(n)$ die Anzahl arithmetischer Operationen. Dann ist:
   $$\mathrm{M}'(n) = \begin{cases} 2n^3 - n^2 & \text{falls } n < 16 \\ 7\mathrm{M}'\left(\frac{n}{2}\right) + \frac{18}{4}n^2 & \text{falls } n \geq 16 \text{ gerade} \\ 7\mathrm{M}'\left(\frac{n-1}{2}\right) + \frac{42}{4}n^2 - 17n + \frac{15}{2} & \text{falls } n \geq 16 \text{ ungerade} \end{cases}$$

   Wir definieren für $x \in \mathbb{R}^+$:
   $$\overline{\mathrm{M}}(x) = \begin{cases} 2x^3 - x^2 & \text{falls } x < 32 \\ 7\overline{\mathrm{M}}\left(\frac{x}{2}\right) + \frac{42}{4}x^2 & \text{falls } x \geq 32 \end{cases}$$

Dann gilt:

$\overline{\mathrm{M}}(n) \geq \mathrm{M}'(n)$ für alle $n \in \mathbb{N}$

Weiterhin ist

$\overline{\mathrm{M}}'(x) = \sum_{i=0}^{k-1} \left[7^i \cdot \frac{42}{4} \left(\frac{x}{2^i}\right)^2\right] + 7^k \left(2 \cdot \left(\frac{x}{2^k}\right)^3 - \left(\frac{x}{2^k}\right)^2\right)$ für $x \geq 32$, wobei $k := \min\left\{l \,\middle|\, \frac{x}{2^l} < 32\right\}$

Mit $k = \lfloor \log_2 x \rfloor - 4 = \log_2 x - t$ für $t \in [4,5]$ erhalten wir:

$$\overline{\mathrm{M}}(x) \leq 7^{\log_2 x} \left(13 \left(\frac{4}{7}\right)^t + 2 \left(\frac{8}{7}\right)^t\right) \leq 4,8 \cdot 7^{\log_2 x} \text{ für } x \geq 32.$$

Für $x < 32$ direkt nachrechnen.

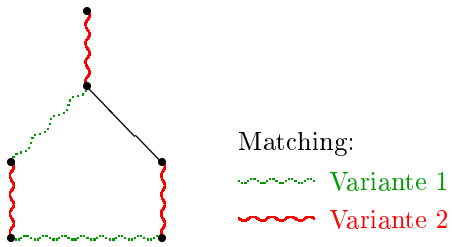
□

Kapitel 5

Matchings in Graphen

5.1 Grundlagen

Definition: Sei $G = (V, E)$ ein ungerichteter, schlichter Graph. Ein Matching M in G ist eine Teilmenge von E, so daß keine zwei Kanten aus M einen Endpunkt gemeinsam haben.



Definition:

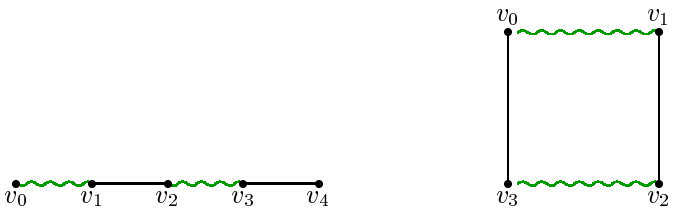
1. Ein Matching M in $G = (V, E)$ heißt perfekt (oder vollkommen), falls

$$|M| = \frac{|V|}{2}$$

2. Ein Matching M heißt „Matching maximaler Kardinalität“ (engl. maximum matching) in G, falls es in G kein Matching M' mit $|M'| > |M|$ gibt. ($\rightarrow$ durchgehend geringelt (rot) im Beispiel)
3. Ein Matching M heißt maximal in G, falls es bezüglich „$\subseteq$“ maximal ist. (engl. maximal matching, $\rightarrow$ gepunktet geringelt (grün) im Beispiel)

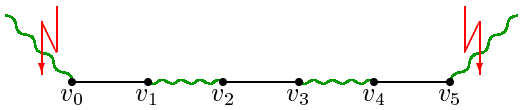
Definition:

1. Ein einfacher Pfad (Kreis) $v_0, v_1, \ldots, v_r$ heißt alternierend bzgl. eines Matchings M, falls die Kanten $\{v_i, v_{i+1}\}$, $0 \leq i < r$, abwechselnd in M und nicht in M liegen.



gerade Länge

2. Ein alternierender Pfad heißt augmentierend, falls er bzgl. „$\subseteq$“ maximal ist und an beiden Enden ungematchte Kanten hat.



ungerade Länge

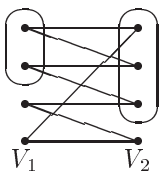
Bemerkung: Es kann keine augmentierenden Kreise geben.

Definition: Sei $G = (V, E)$ ein Graph. Wenn V in zwei nichtleere Teilmengen V_1 und V_2 partitioniert werden kann ($V_1 \cup V_2 = V, V_1 \cap V_2 = \emptyset$), so daß $E \subseteq V_1 \times V_2$ ist, so heißt G bipartit ($G = (V_1, V_2, E)$).

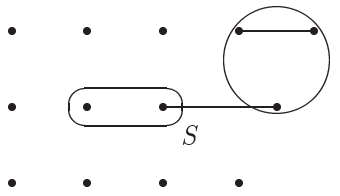
Heiratssatz: (Frobenius, Hall, Rado, König)
Sei $G = (V_1, V_2, E)$ ein bipartiter Graph. G enthält ein Matching der Kardinalität $|V_1|$ genau dann, wenn gilt:

$$\forall V \subseteq V_1 : |N(V)| \geq |V|,$$

wobei N(V) die Nachbarschaft von V (in V_2) bezeichnet.



Satz: (Tutte, 1947)
Ein Graph $G = (V, E)$ hat ein perfektes Matching genau dann, wenn $|V|$ gerade ist und es kein $S \subseteq V$ gibt, so daß der durch $V \setminus S$ induzierte Teilgraph mehr als $|S|$ Zusammenhangskomponenten ungerader Größe enthält.

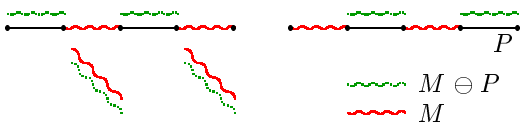


Satz: (Berge, 1957)
Ein Matching hat maximale Kardinalität genau dann, wenn es keinen augmentierenden Pfad dafür gibt.

Definition: Seien S, T zwei Mengen, dann bezeichne $S \ominus T$ die symmetrische Differenz von S und T, d.h. $S \ominus T = (S - T) \cup (T - S)$.

Lemma: Sei M ein Matching, P ein augmentierender Pfad bzgl. M. Dann ist auch $M \ominus P$ ein Matching und es gilt $|M \ominus P| = |M| + 1$.

Beweis:
„⇐“



„⇒“ Satz von Berge □

Lemma: Seien M, N Matchings in G, und sei $|N| > |M|$. Dann enthält $N \ominus M$ mindestens $|N| - |M|$ knotendisjunkte augmentierende Pfade bzgl. M.

Beweis: Der Grad eines Knotens in $(V,\ N \ominus M)$ ist ≤ 2.
Die Zusammenhangskomponenten von $(V,\ N \ominus M)$ sind also

1. isolierte Knoten
2. einfache Kreise (gerade Länge)
3. alternierende Pfade

Seien $C_1, \ldots, C_r$ die Zusammenhangskomponenten in $(V,\ N \ominus M)$ dann gilt:

$$M \ominus \underbrace{C_1 \ominus C_2 \ominus \ldots \ominus C_r}_{N \ominus M} = N$$

Nur die C_i's die augmentierende Pfade bzgl. M sind, vergrößern die Kardinalität des Matchings und zwar jeweils um genau 1. Also muß es mindestens $|N| - |M|$ C_i's geben, die augmentierende Pfade bzgl. M sind (ev. knotendisjunkt da ZHK's). □

Korollar: Satz von Berge

Lemma: Sei M ein Matching der Kardinalität r, und sei s die maximale Kardinalität eines Matchings in $G = (V, E)$. Dann gibt es einen augmentierenden Pfad bzgl. M der Länge $\leq 2\lfloor \frac{r}{s-r} \rfloor + 1$, wobei $s > r$ vorausgesetzt sei.

Beweis: Sei N ein Matching maximaler Kardinalität in G, $|N| = s$. $N \ominus M$ enthält $\geq s - r$ augmentierende Pfade bzgl. M, die alle knotendisjunkt und damit auch kantendisjunkt sind. Einer dieser augmentierenden Pfade enthält daher $\leq \lfloor \frac{r}{s-r} \rfloor$ Kanten aus M. □

Lemma: Sei P ein kürzester augmentierender Pfad bzgl. M und sei P' ein augmentierender Pfad bzgl. des neuen Matchings $M \ominus P$. Dann gilt:
$|P'| \geq |P| + 2|P \cap P'|$

Beweis: $N = M \ominus P \ominus P'$, also $N = |M| + 2$. Also enthält $M \ominus N$ mindestens 2 knotendisjunkte augmentierende Pfade bzgl. M, etwa $P_1,\ P_2$. Es gilt:

$N \ominus M = P \ominus P'$, also

$|P \ominus P'| \overset{\text{Def.}}{=} |(P - P') \cup (P' - P)| = |P| + |P'| - 2|P \cap P'| \geq |P_1| + |P_2| \geq 2|P|$,
also $|P| + |P'| - 2|P \cap P'| \geq 2|P|$, also
$|P'| \geq \underbrace{2|P| - |P|}_{|P|} + 2|P \cap P'|$

□

Schema für Matching Algorithmus:

1. Beginne mit Matching $M_0 := \emptyset$.
2. Berechne Folge $M_0,\ P_0,\ M_1,\ P_1, \ldots,\ M_i,\ P_i, \ldots$ Wobei jeweils P_i ein kürzester augmentierender Pfad bzgl. M_i ist und $M_{i+1} = M_i \ominus P_i$.

Es gilt: $|P_{i+1}| \geq |P_i|$

Lemma: Seien P_i und P_j in obiger Folge zwei augmentierende Pfade gleicher Länge. Dann sind P_i und P_j knotendisjunkt.

Beweis: Annahme: es gibt eine Folge $(P_k)_{k\geq 0}$ mit $|P_i| = |P_j|$, $j > i$, $P_i \cap P_j \neq 0$, $j - i$ minimal gewählt. Durch die Wahl von j sind die Pfade $P_{i+1}, \ldots, P_j$ knotendisjunkt. Also ist P_j ein augmentierender Pfad bzgl. M', dem Matching nach dem augmentieren mit $P_0,\ P_1, \ldots,\ P_i$. Mit vorigem Lemma: $|P_j| \geq |P_i| + 2|P_i \cap P_j|$ also $P_i,\ P_j$ kantendisjunkt. Da in M' jeder Knoten in P_i gematcht ist (und auch in $M' \ominus P_{i+1} \ominus P_{i+2} \ominus \ldots \ominus P_{j-1}$) können P_i und P_j auch keinen Knoten gemeinsam haben. □

Satz: Sei s die maximale Kardinalität eines Matchings in $G = (V, E)$. Dann enthält die Folge $|P_0|, |P_1|, \ldots$ höchstens $\lfloor 2\sqrt{s} + 1 \rfloor$ verschiedene Werte.

Beweis: Sei $r := \lfloor s - \sqrt{s} \rfloor$. Per Konstruktion ist $|M_i| = i$, also $|M_r| = r$. Mit dem Lemma über die maximale Länge eines kürzesten augmentierenden Pfades folgt

$$|P_r| \quad \leq \quad 2\left\lfloor \frac{\lfloor s - \sqrt{s} \rfloor}{s - \lfloor s - \sqrt{s} \rfloor} \right\rfloor + 1 \quad \leq \quad 2\left\lfloor \frac{s}{\sqrt{s}} \right\rfloor + 1 \quad = \quad 2\lfloor \sqrt{s} \rfloor + 1.$$

Für $i \leq r$ ist also $|P_i|$ eine der ungeraden Zahlen in $[1, 2\sqrt{s} + 1]$, also eine von $\lfloor \sqrt{s} \rfloor + 1$ ungeraden Zahlen. $P_{r+1}, \ldots, P_{s-1}$ tragen höchstens $s - r - 1 < \sqrt{s}$ zusätzliche Längen bei. □

Verfeinertes Schema:

$M := \emptyset$;
while $\exists$ augmentierender Pfad bzgl. M do
    $l :=$ Länge des kürzesten augmentierenden Pfades bzgl. M;
    bestimme eine bzgl. „$\subseteq$“ maximale Menge $\{Q_1, \ldots, Q_m\}$
    augmentierender Pfade bzgl. M, die alle Länge l haben und knotendisjunkt sind;
    $M := M \ominus Q_1 \ominus \ldots \ominus Q_m$;
od

Korollar: In obiger *while*-Schleife finden höchstens $\mathcal{O}\left(|V|^{\frac{1}{2}}\right)$ Durchläufe statt.

5.2 Matchings maximaler Kardinalität in bipartiten Graphen

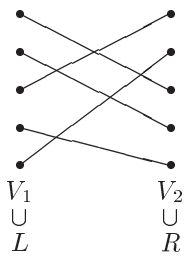
Gegeben: $G = (V_1, V_2, E)$ bipartiter Graph.

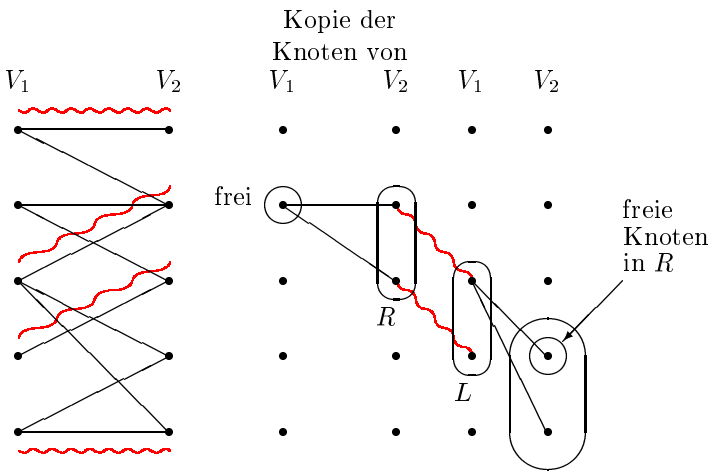
Simultane BFS zur Bestimmung der Länge eines kürzesten augmentierten Pfades.

⌜

<u>for</u> alle $v \in V_1 \cup V_2$ <u>do</u>
  $label[v] := 0$;
<u>od</u>
$R := \emptyset$; $l := 1$;
<u>for</u> alle freien Knoten $v \in V_1$ <u>do</u>
  <u>for</u> alle Kanten $\{v, w\} \in E$ <u>do</u>
    $label[w] := 1$; $R := R \cup \{w\}$;
  <u>od</u>
<u>od</u>
<u>while</u> $R \neq \emptyset$ <u>and</u> R enthält keinen freien Knoten <u>do</u>
  $L := \emptyset$; $l{+}{+}$;
  <u>for</u> alle $w \in R, \{v, w\} \in M$ <u>do</u>
    $L := L \cup \{v\}$;
    $label[v] := l$;
  <u>od</u>
  $R := \emptyset$; $l{+}{+}$;
  <u>for</u> alle $v \in L, \{v, w\} \in E \setminus M$ <u>do</u>
    <u>if</u> $label[w] = 0$ <u>then</u>
      $R := R \cup \{w\}$;
      $label[w] := l$;
    <u>fi</u>
  <u>od</u>
<u>od</u>
$R :=$ Menge der freien Knoten in R;

⌟





Literaturverzeichnis

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms.* Addison–Wesley Publishing Company, Reading MA, 1976.

[2] Akuja, Kurt Mehlhorn, Orlin, and Robert E. Tarjan. Faster Algorithms for the Shortest Path Problem. J ACM 37, 1990. p. 213–223.

[3] V.L. Avlazarov, E.A. Dinic, M.A. Kronod, and I.A. Faradzev. *On economical construction of the transitive closure of a directed graph.* Soviet Math. Doklady 11, 1970. p. 1209–1210.

[4] S. W. Bent. Dynamic weighted data structures. TR STAN–CS–82–916. Department of Computer Science, Stanford University, 1982.

[5] Samuel W. Bent and John W. John. *Finding the median requires* $2n$ *comparisons.*

[6] Thomas H. Cormen, Charles E. Leierson, and Ronald L. Rivest. *Introduction to algorithms.* The MIT Electrical Engineering and Computer Science Series. McGraw-Hill Book Company, New York – St. Louis – San Francisco – Montreal – Toronto, 1990.

[7] J. R. Driscoll, N. Sarnak, D. D. Sleator, and Robert E. Tarjan. Making data structures persistent. Proceedings of the 18^{th} Annual ACM Symposium on Theory of Computing (STOC), 1986. p. 109–121.

[8] Donald E. Knuth. *The art of computer programming*, volume 3: Sorting and searching. Addison–Wesley Publishing Company, Reading MA, 1973.

[9] Kurt Mehlhorn. *Data structures and algorithms*, volume 1: Sorting and searching of *EATCS Monographs on Theoretical Computer Science.* Springer Verlag, Berlin - Heidelberg – New York – London – Paris – Tokyo – Hong Kong, 1984.

[10] Kurt Mehlhorn. *Data structures and algorithms*, volume 2: Graph algorithms and $\mathcal{NP}$-Completeness of *EATCS Monographs on Theoretical Computer Science.* Springer Verlag, Berlin – Heidelberg – New York – London – Paris – Tokyo – Hong Kong, 1984.

[11] Ottmann and Widmayer. *Algorithmen und Datenstrukturen.* BI–Wissenschaftsverlag, 1990.

[12] Chirstos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: Algorithms and complexity.* Prentice–Hall, Englewood Cliffs NJ, 1982.

[13] C. P. Schnorr. *An Algorithm for Transitive Closure with Linear Expected Time.* SIAM Journal on Computing 7, 1978. p. 127–133.

[14] Sedgewick and Flajolet. *An intoduction to the analysis of algorithms.* Addison–Wesley Publishing Company, Reading MA, 1996.

[15] D. D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. Comm. ACM 28(2), 1985. p. 202–208.

[16] Bjarne Stroustroup. *The C++ programming language.* Addison–Wesley Publishing Company, Reading MA, 3rd edition, 1997.

[17] Robert E. Tarjan. *Data Structures and Network Algorithms.* CBMS-NSF Regional Conference Series in Applied Mathematics, Band 44. SIAM, Philadelphia PA, 1983.

[18] A. C.-C. Yao. Constructing Minimum Spanning Trees in k-dimensional spaces and related problems. SIAM Journal on Computing 11(4), 1982. p. 721–736, Weiterführend zum Thema: Euklidische, minimale Spannbäume.