Effiziente Algorithmen

Dr.habil. Sergei Bezrukov, (sb)
Büro: F2.411
Tel.: 60-6728
Sprechstunden: immer wenn ich im Büro bin.

Vorlesungen: (4 SWS)
Mo. 14:00 - 15:30
Mi. 11:07 - 12:37
Raum: D1

Übungen: (2 SWS)
Mi: 14 - 16, 16 - 18 (J2.130)
Friedhelm Neugebauer, (freddy, 60-6690, F2.313)
Ulf-Peter Schroeder, (ups, 60-6726, F2.409)

Do: 9 - 11, 11 - 13 (N3.206)
Petra Berenbrink, (pebe, 60-6690, F2.313)
Friedhelm Neugebauer, (freddy, 60-6690, F2.313)

Folien + Übungszettel:
http://www.uni-paderborn.de/fachbereich/AG/monien/LEHRE/SS98/Lehre_ss98.html

Inhalt:

1. Grundlagen
   (O-Kalkül, Master-Satz)
2. Einige Methoden
   (Dynamische Programmierung, Greedy Methoden)
3. Graphen-Algorithmen
   (Zusammenhangskomponenten, Kürzeste Wege, Färbungen)
4. Flußalgorithmen
   (Anwendungen: Zusammenhang von Graphen, Matchings)
5. Algebraische Algorithmen
   (Schnelle Multiplikation und Inverse der Matrizen, FFT)
6. Geometrische Algorithmen
   (Konvexe Hüllen, Range-Search, Proximity)
7. NP-Vollständigkeit
   (verschiedene Reduktionen)
8. Approximations-Algorithmen
   (Knotenüberdeckung, TSP, Schedulings, ...)
9. Einfühgung in randomisierte Algorithmen
   (Monte-Carlo-Methode, Las-Vegas-Algorithmen)

Literatur

1. Cormen T.H., Leiserson C.E., Rivest R.L.:
   "Introduction to Algorithms", 1990.

2. Mehlhorn K.:
   "Data Structures and Efficient Algorithms", 1984.

3. Aho A.V., Hopcroft J.E., Ullman J.D.:
   "Data Structures and Algorithms", 1974

4. Garey M.R., Johnson D.S.:
   "Computers and Intractability: A Guide to the Theory of NP-completeness", 1979.

5. Hochbaum D.S. (ed.)
   "Approximation Algorithms for NP-Hard Problems", 1997.

6. Rajeev M., Prabhakar R.:
   "Randomised Algorithms", 1995.

Scheinerwerb:

- 2-mal Vorrechnen in der Übungen
- Klausur bestehen

1 Lektion 1: Grundlagen

Definition 1.1 *Seien $f(n)$ und $g(n)$ zwei Funktionen und sei n eine ganze Zahl. Wir schreiben (nach E.Landau):*

$$f(n) = o(g(n)) \qquad \Longleftrightarrow \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Beispiel:

$$n^\alpha = o(n^\beta) \quad \Longleftrightarrow \quad \alpha < \beta.$$

Die Relation $o(\cdot)$ ist transitiv, d.h.

$$f(n) = o(g(n)) \quad \wedge \quad g(n) = o(h(n)) \quad \Rightarrow \quad f(n) = o(h(n)),$$

und klassifiziert die Funktionen, z.B.:

$$1 = o(\log\log n), \qquad \log\log n = o(\log n), \qquad \log n = o(n^\epsilon)$$
$$n^\epsilon = o(n^c), \qquad n^c = o(n^{\log n}), \qquad n^{\log n} = o(c^n)$$
$$c^n = o(n^n), \qquad n^n = o(c^{c^n}),$$

wobei $0 < \epsilon < 1 < c$ gilt.

Diese Klassifizierung ist nur dann sinnvoll, wenn n groß ist.

Beispiel:
Betrachten wir $\log n = o(n^{0.0001})$ und sei $n = 10^{100}$.
Es gilt: $\log n = 100$ aber $n^{0.0001} \approx 1.023$.

Andererseits sei $n = 10^{10^{100}}$. Dann gilt:
$\log n = 10^{100}$ im Vergleich mit $n^{0.0001} = 10^{10^{96}}$.

Definition 1.2 *Seien $f(n)$ und $g(n)$ zwei Funktionen und sei n eine ganze Zahl. Wir schreiben (nach P.Bachmann, 1894):*

$$\begin{aligned} f(n) &= O(g(n)) \quad \textit{für } n \to \infty \qquad \Longleftrightarrow \\ \exists n_0 : |f(n)| &\leq c|g(n)| \qquad \forall n \geq n_0 \end{aligned}$$

für eine Konstante c.

Beispiel:

$$4n^3 + 5n^2 - 6n + 2 = O(n^3),$$

aber auch

$$4n^3 + 5n^2 - 6n + 2 = O(n^{10}).$$

Definition 1.3 *Wir schreiben:*

$$f(n) = \Omega(g(n)) \quad \Longleftrightarrow \quad |f(n)| \geq c|g(n)|$$

für eine Konstante $c > 0$ $\forall n \geq n_0$.

Offensichtlich gilt:

$$f(n) = \Omega(g(n)) \quad \Longleftrightarrow \quad g(n) = O(f(n)).$$

Definition 1.4 *Wir schreiben für $n \to \infty$:*

$$\begin{aligned} f(n) &= \Theta(g(n)) \quad \Longleftrightarrow \\ f(n) &= O(g(n)) \quad \wedge \quad f(n) = \Omega(g(n)). \end{aligned}$$

Definition 1.5 *Wir schreiben für $n \to \infty$:*

$$\begin{aligned} f(n) \sim g(n) \quad &\Longleftrightarrow \quad f(n) = g(n) + o(g(n)) \\ &\Longleftrightarrow \quad \lim_{n \to \infty} \frac{f(n)}{g(n)} = 1. \end{aligned}$$

Es gilt:

$$\begin{aligned}
n^m &= O(n^{m'}), \qquad \text{falls } m \le m' \\
O(f(n)) + O(g(n)) &= O(|f(n)| + |g(n)|), \\
f(n) &= O(f(n)), \\
c \cdot O(f(n)) &= O(f(n)), \qquad \text{falls } c = \text{const} \\
O(O(f(n))) &= O(f(n)), \\
O(f(n)) \cdot O(g(n)) &= O(f(n) \cdot g(n)) = f(n) \cdot O(g(n)), \\
\log(1 + O(f(n))) &= O(f(n)), \qquad \text{falls } f(n) = o(1) \\
e^{O(f(n))} &= 1 + O(f(n)), \qquad \text{falls } f(n) = o(1).
\end{aligned}$$

Beispiele: $(|x| < 1)$

$$\begin{aligned}
e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + O(x^4), \\
\ln(1+x) &= x - \frac{x^2}{2} + O(x^3), \\
\frac{1}{1-x} &= 1 + x + x^2 + O(x^3),
\end{aligned}$$

Stirling-Formel:

$$\begin{aligned}
n! &= \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right), \\
\binom{n}{k} &\sim \frac{n^k}{k!}, \qquad \text{falls } k = o(\sqrt{n}), \\
\binom{n}{\lfloor \frac{n}{2} \rfloor} &\sim \sqrt{\frac{2}{\pi n}}\, 2^n.
\end{aligned}$$

Grundlegende Summen.

- Arithmetische Folgen:

$$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1) = \binom{n+1}{2} = \Theta(n^2).$$

- Geometrische Folgen:

$$\begin{aligned} \sum_{k=0}^{n} x^k &= \frac{x^{n+1}-1}{x-1} \qquad \text{für } x \neq 1, \\ \sum_{k=0}^{\infty} x^k &= \frac{1}{1-x} \qquad \text{für } |x| < 1. \end{aligned}$$

- Harmonische Folge:

$$H_n = \sum_{k=1}^{n} \frac{1}{k} = \ln n + \gamma + o(1),$$

wobei $\gamma \approx 0.577$ (Euler's Konstante) ist.

- Teleskopische Folgen:

$$\begin{aligned} \sum_{k=1}^{n} (a_k - a_{k-1}) &= a_n - a_0 \\ \sum_{k=0}^{n-1} (a_k - a_{k+1}) &= a_0 - a_n. \end{aligned}$$

Beispiel:

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} = \sum_{k=1}^{n-1} \left(\frac{1}{k} - \frac{1}{k+1}\right) = 1 - \frac{1}{n}.$$

Theorem 1.1 [1] (Mastersatz)
Seien $a \geq 1$ und $b > 1$ Konstanten, und sei $T(n)$ eine Funktion mit ganzzahligen Variablen die durch

$$T(n) = a\,T(n/b) + f(n)$$

definiert ist (hier ist n/b entweder $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$). Dann gilt:

a. *Falls $f(n) = O(n^{\log_b a - \epsilon})$ für eine Konstante $\epsilon > 0$ gilt, dann ist:*
$$T(n) = \Theta(n^{\log_b a}).$$

b. *Falls $f(n) = \Theta(n^{\log_b a})$ gilt, dann ist:*
$$T(n) = \Theta(n^{\log_b a} \log n).$$

c. *Falls für alle $n \geq n_0$ $f(n) = \Omega(n^{\log_b a + \epsilon})$ für eine Konstante $\epsilon > 0$ und $af(n/b) \leq cf(n)$ für eine Konstante $c < 1$ gilt, dann ist:*
$$T(n) = \Theta(f(n)).$$

Der Mastersatz ist aber <u>**nicht für alle**</u> Funktionen $f(n)$ anwendbar !
z.B. **nicht** für $f(n) = n^{\log_b a} \log n$.

Anwendungen des Mastersatzes.

Sei

$$T(n) = 9T(n/3) + n.$$

Mit $a = 9$ und $b = 3$, haben wir

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

und

$$f(n) = n = O(n^{\log_3 9 - \epsilon})$$

mit $\epsilon = 1$. Damit ist der Fall a. erfüllt und wir erhalten:

$$T(n) = \Theta(n^2).$$

Sei

$$T(n) = T(2n/3) + 1.$$

Da $a = 1$ und $b = 3/2$, haben wir

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

und

$$f(n) = 1 = \Theta(n^{\log_b a}) = \Theta(1).$$

Der Fall b. ergibt:

$$T(n) = \Theta(\log n).$$

Sei

$$T(n) = 3T(n/4) + n\log n.$$

Mit $a = 3$ und $b = 4$ haben wir:

$$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$$

und

$$f(n) = n\log n = \Omega(n\mathrm{log}_4 3 + \epsilon)$$

mit $\epsilon \approx 0.2$. Der Fall c. ergibt:

$$T(n) = \Theta(n\log n).$$

Wenn aber

$$T(n) = 2T(n/2) + n\log n$$

ist, dann gilt

$$n^{\log_b a} = n^{\log_2 2} = n,$$

und

$$\frac{f(n)}{n^{\log_b a}} = \frac{n\log n}{n} = \log n = o(n^{\epsilon})$$

für jedes $\epsilon > 0$.

Damit können wir den Mastersatz in diesem Fall nicht anwenden.

2 Lektion 2: Allgemeine Methoden zum Algorithmendesign

1. Dynamische Programmierung
   - Multiplikation von Matrizen
   - Längste allgemeine Teilfolge
   - Optimale Triangulierung eines Polygons

2. Greedy Methoden
   - Das Schedulproblem
   - Das Ruksackpakungsproblem
   - Matroide
   - Minimale und Maximale Spannbäume
   - Huffman's Kodierung

Sei A eine $p \times q$ Matrix und sei B eine $q \times r$ Matrix. Dann ist $C = A \cdot B$ eine $p \times r$ Matrix.

Man braucht pqr Multiplikationen um die Matrix C zu berechnen:

<u>**Algorithmus** 2.1</u> $\mathrm{MM}(A, B)$;

if # Spalten$(A) \neq$ # Zeilen(B)
  then ERROR
  else **for** $i = 1$ **to** # Zeilen(A)
    do **for** $j = 1$ **to** # Spalten(B)
      do $C[i, j] := 0$
        for $k = 1$ **to** # Spalten(A)
          do $C[i, j] := C[i, j] + A[i, k] \cdot B[k, j]$
return C

Die Multiplikation $A_1 A_2 A_3$ ist assoziative und die Anzahl der Multiplikationen ist von der Multiplikationsordnung abhängig.

Beispiel:
Seien A_1 10×100, A_2 100×5 und A_3 5×50 i Matrizen.

Das Schema $((A_1 A_2) A_3)$ benötigt $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$ Multiplikationen.
Das Schema $(A_1 (A_2 A_3))$ benötigt $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75000$ Multiplikationen.

Das Problem:
Finde eine Optimale Multiplikationsordnung für $A_1 A_2 ... A_n$.

Für $n = 4$ gibt es nur 5 Multiplikationsordnungen:

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

Im Allgemeinen beträgt Anzahl der Multiplikationsordnungen

$$P(n) = \begin{cases} 1 & \text{falls } n = 1, \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & \text{falls } n \geq 2. \end{cases}$$

Das ergibt:

$$P(n) = \frac{1}{n}\binom{2n-2}{n-1} = \Omega\left(\frac{4^n}{n^{3/2}}\right).$$

Deswegen ist "the complete choice" exponentiel groß.

a. Optimale Multiplikationsordnung:

Sei

$$A_{i..j} = A_i \cdot ... \cdot A_j.$$

Es gibt ein k $(i \leq k < j)$, so daß die optimale Multiplikationsordnung für $A_i \cdot ... \cdot A_j$ die Form:

$$A_i \cdot ... \cdot A_j = A_{i..k} \cdot A_{k+1..j},$$

besitzt und die Multiplikationsordnungen für $A_{i..k}$ und $A_{k+1..j}$ optimal sind.

b. Rekursive Lösung:

Sei $m[i,j]$ die minimale Anzahl der Multiplikationen in der optimalen Ordnung für $A_{i..j}$ und A_i eine $p_{i-1} \times p_i$ Matrix. Dann gilt:

$$m[i,j] = \begin{cases} 0 & \text{falls } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{falls } i < j. \end{cases}$$

Sei $p = p_0, ..., p_n$.

<u>**Algorithmus** 2.2</u> $\mathrm{RMC}(p, i, j)$;

if $i = j$
  then **return** 0
$m[i,j] := \infty$
for $k = i$ **to** $j - 1$
  do $q := \mathrm{RMC}(p, i, k) + \mathrm{RMC}(p, k+1, j) + p_{i-1}p_kp_j$
    if $q < m[i,j]$
      then $m[i,j] := q$
return $m[i,j]$

Die Laufzeit:

$$T(n) \ge 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \qquad \text{für } n > 1,$$

oder

$$T(n) \ge 2 \cdot \sum_{k=1}^{n-1} T(k) + n,$$

was ergibt

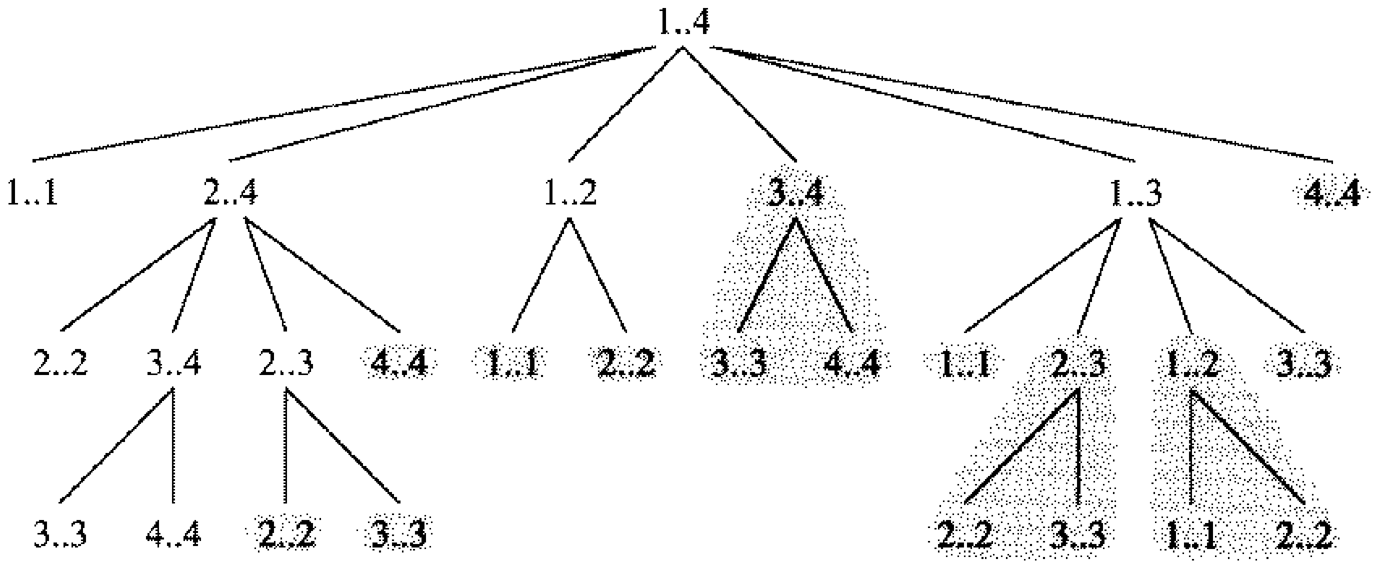
$$T(n) \ge 2^{n-1}.$$

Wir benutzen eine Induktion über n.

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

Warum ist die Laufzeit so groß?

Viele Berechnungen werden mehrfach wiederholt:



c. Besserer Algorithmus:

Die Anzahl der verschiedenen Teilprobleme ist $O(n^2)$.

Das "bottom-up" Prinzip.

<u>**Algorithmus** 2.3</u> $\mathrm{MCO}(p)$;

$\mathbf{for}\ i = 1\ \ \mathbf{to}\ n$
$\quad \mathbf{do}\ m[i,i] := 0$

$\mathbf{for}\ l = 2\ \ \mathbf{to}\ n$
$\quad \mathbf{do}\ \ \mathbf{for}\ i = 1\ \ \mathbf{to}\ n - l + 1$
$\quad\quad \mathbf{do}\ (j := i + l - 1; \quad m[i,j] := \infty)$

$\quad\quad\quad \mathbf{for}\ k = i\ \ \mathbf{to}\ j - 1$
$\quad\quad\quad\quad \mathbf{do}\ q := m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
$\quad\quad\quad\quad\quad \mathbf{if}\ q < m[i,j]$
$\quad\quad\quad\quad\quad\quad \mathbf{then}\ (m[i,j] := q; \quad s[i,j] := k)$

$\mathbf{return}\ m$ und s

Die Laufzeit von MCO ist $O(n^3)$, darüberhinaus braucht MCO $O(n^2)$ Speicherplatz um die Werte von m und s zu speichern.

Beispiel:

Sei $p = (30, 35, 15, 5, 10, 20, 25)$.

$j \setminus i$	1	2	3	4	5	6
6	15.125	10.500	5.375	3.500	5.000	0
5	11.875	7.125	2.500	1.000	0	
4	9.375	4.375	750	0		
3	7.875	2.625	0			
2	15.750	0				
1	0					

Tabelle der Werte von $m[i, j]$.

$j \setminus i$	1	2	3	4	5
6	3	3	3	5	5
5	3	3	3	4	
4	3	3	3		
3	1	2			
2	1				

Tabelle der Werte von $s[i, j]$.

$$\begin{aligned} m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 & (= 13000) \\ m[2,3] + m[4,5] + p_1 p_3 p_5 & (= 7125) \\ m[2,4] + m[5,5] + p_1 p_4 p_5 & (= 11375) \end{cases} \\ &= 7125 \end{aligned}$$

d. Konstruierung der optimalen Lösung:

Sei $A = \{A_1, A_2, ..., A_n\}$.

$\mathrm{MCM}(A, s, 1, n)$

<u>**Algorithmus** 2.4</u> $\mathrm{MCM}(A, s, i, j)$;

if $i < j$
  then $X := \mathrm{MCM}(A, s, i, s[i, j])$
    $Y := \mathrm{MCM}(A, s, s[i, j] + 1, j)$
    return $\mathrm{MM}(X, Y)$
  else **return** A_i

In unserem Beispiel ist die optimale Multiplikationsordnung:

$$((A_1(A_2A_3))((A_4A_5)A_6)).$$

2. Elemente der dynamischen Programmierung:

- Struktur der optimalen Lösung
- Sich überlappende Teilprobleme
- Memorization

Memorizierte Version von RMC

Algorithmus 2.5 $\mathrm{MMC}(p)$;

for $i = 0$ **to** n
  do **for** $j = i$ **to** n
    do $m[i,j] := \infty$
return $\mathrm{LC}(p, 1, n)$

Algorithmus 2.6 $\mathrm{LC}(p, i, j)$;

if $m[i,j] < \infty$
  then **return** $m[i,j]$

if $i = j$
  then $m[i,j] := 0$
  else **for** $k = i$ **to** $j-1$
    do $q := \mathrm{LC}(p,i,k) + \mathrm{LC}(p,k+1,j) + p_{i-1}p_kp_j$
      if $q < m[i,j]$
        then $m[i,j] := q$

return $m[i,j]$

Dieser "top-down" Algorithmus berechnet die Lösung in $O(n^3)$ Zeit.

3. Längste allgemeine Teilfolge

Definition 2.1 *Seien* $X = (x_1, ..., x_m)$ *und* $Z = (z_1, ..., z_k)$ *Folgen. Die Folge* Z *heißt* Teilfolge *von* X *falls* $\exists\ i_1, ..., i_k$ $(i_1 < \cdots < i_k)$*, so daß* $x_{i_j} = z_j$ *gilt für* $j = 1, ..., k$.

Definition 2.2 *Seien* X *und* Y *Folgen. Eine Folge* Z *heißt* allgemeine Teilfolge *von* X *und* Y *falls* Z *Teilfolge von* X *und* Y *ist.*
Z *heißt* längste *allgemeine Teilfolge falls ihre Länge maximal ist.*
Schreibweise: $Z = CS(X, Y)$*, bzw.* $Z = LCS(X, Y)$.

Beispiel:

Seien

$$X = (A, B, C, B, D, A, B) \qquad Y = (B, D, C, A, B, A).$$

Dann gilt:

$$\begin{aligned} Z &= (B, C, A) = CS(X, Y) \\ Z' &= (B, C, B, A) = LCS(X, Y). \end{aligned}$$

Das Problem:

Gegeben Folgen X und Y. Finde $LCS(X, Y)$.

a. Die Struktur von $LCS(X, Y)$

Sei $X = (x_1, ..., x_n)$ eine Folge. Wir bezeichnen $X_i = (x_1, ..., x_i)$, $i = 1, ..., n$.

Behauptung 2.1 *Seien $X = (x_1, ..., x_m)$ und $Y = (y_1, ..., y_n)$ Folgen. Weiterhin sei $Z = (z_1, ..., z_k) = LCS(X, Y)$. Dann gilt:*

1. $x_m = y_n \Longrightarrow z_k = x_m$ *und* $Z_{k-1} = LCS(X_{m-1}, Y_{n-1})$.
2. $x_m \neq y_n$, $z_k \neq x_m \Longrightarrow Z = LCS(X_{m-1}, Y)$.
3. $x_m \neq y_n$, $z_k \neq y_n \Longrightarrow Z = LCS(X, Y_{n-1})$.

b. Rekursive Lösung

Sei $c[i, j]$ die Länge von $LCS(X_i, Y_j)$. Dann gilt:

$$c[i,j] = \begin{cases} 0 & \text{falls } i \cdot j = 0 \\ c[i-1, j-1] + 1 & \text{falls } i, j > 0 \text{ und } x_i = y_j \\ \max\{c[i, j-1],\ c[i-1, j]\} & \text{falls } i, j > 0 \text{ und } x_i \neq y_j \end{cases}$$

Die Anzahl der Teilprobleme ist $\Omega(mn)$.

c. Der Algorithmus

Algorithmus 2.7 $\mathrm{LCS}(X, Y)$;

for $i = 1$ **to** m
    do $c[i, 0] := 0$
for $j = 0$ to n
    do $c[0, j] := 0$

for $i = 1$ **to** m
    do **for** $j = 1$ **to** n
        do **if** $x_i = y_j$
            then $c[i, j] := c[i-1, j-1] + 1$
                $b[i, j] :=$ "$\nwarrow$"
            else **if** $c[i-1, j] \geq c[i, j-1]$
                then $c[i, j] := c[i-1, j]$
                    $b[i, j] :=$ "$\uparrow$"
                else $c[i, j] := c[i, j-1]$
                    $b[i, j] :=$ "$\leftarrow$"

return c und b

Die Laufzeit von LCS ist $O(mn)$.

d. **Beispiel:**

Sei

$$X = (A, B, C, B, D, A, B) \quad \text{und} \quad Y = (B, D, C, A, B, A)$$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

$$LCS(X, Y) = (B, C, B, A)$$

Konstruierung der LCS

$\textsc{Print-LCS}(b, X, |X|, |Y|)$

<u>**Algorithmus** 2.8</u> $\textsc{Print-LCS}(b, X, i, j)$;

if $i = 0$ or $j = 0$
  then **return**

if $b[i, j] =$ "$\nwarrow$"
  then $\textsc{Print-LCS}(b, X, i - 1, j - 1)$
  else
    if $b[i, j] =$ "$\uparrow$"
      then $\textsc{Print-LCS}(b, X, i - 1, j)$
      else $\textsc{Print-LCS}(b, X, i, j - 1)$

Die Laufzeit von $\textsc{Print-LCS}$ ist $O(m + n)$.

4. Triangulierung des Polygons

Sei P ein einfaches convexes Polygon mit der Knotenmenge

$$\{v_0, v_1, ..., v_n\}.$$

Definition 2.3 *Die Segmente der Form* $v_i v_{i+1 \textit{ mod } n+1}$ *heissen* Seiten *und die Segmente der Form* $v_i v_j$ *die nicht Seiten sind heißen* Chorden *des Polygons* P.

Definition 2.4 *Eine* Triangulierung T *des Polygons* P *ist eine Menge Chorden, die* P *in "disjunkte" Dreiecke zerlegt.*

Gegeben sei eine Gewichtfunktion $w : \{v_i v_j\} \mapsto R$ mit $0 \leq i, j \leq n$, $i \neq j$ und $w(v_i v_j) = w(v_j v_i)$. Weiterhin sei $v_i v_j v_k$ ein Dreieck der Triangulierung. Setzen wir:

$$w(v_i v_j v_k) = w(v_i v_j) + w(v_i v_k) + w(v_j v_k).$$

Definieren wir das Gewicht von einer Triangulierung T als die Summe der Gewichte von allen Dreiecken.

Das Problem:

Gegeben Polygon P. Finde eine Triangulierung T von P mit minimalem Gewicht.

Wir bezeichnen eine solche Triangulierung als optimal.

a. Die Struktur der optimalen Lösung

Sei T eine optimale Triangulierung von P die das Dreieck $v_0 v_k v_n$ enthält $1 \leq k < n$. Dann besteht T aus eine optimalen Triangulierung der Polygons $v_0, ..., v_k$ und $v_k, v_{k+1}, ..., v_n$.

b. Zerlegung in Teilprobleme

Sei $w[i,j]$ das Gewicht der optimalen Triangulierung des Polygons $v_{i-1}, ..., v_j$.

Die Anzahl der Teilprobleme ist $O(n^2)$.

c. Rekursive Lösung

Es gilt:

$$w[i,j] = \begin{cases} 0 & \text{falls } i = j \\ \min\limits_{i \leq k < j} \{w[i,k] + w[k+1,j] + w(v_{i-1} v_k v_j)\} & \text{falls } i < j. \end{cases}$$

Deswegen, kann die Optimale Triangulierung von P mit (fast) demgleichen Algorithmus wie für die Matrizenmultiplikation konstruirt werden.

5. Greedy Methoden

Sei S die Menge der n Vorlesungen die in demselben Saal stattfinden müssen. Die Vorlesung V_i charakterisiert sich durch die Anfangszeit s_i und mit der Endezeit f_i, $i = 1, ..., n$.

Wir representieren V_i als Interval $[s_i, f_i)$. Die Intervale (bzw. die Vorlesungen) $[s_i, f_i)$ und $[s_j, f_j)$ heißen *kompatibel* dann und nur dann wenn

$$f_i \leq s_j \qquad \text{oder} \qquad f_j \leq s_i$$

gilt.

Das Schedulproblem:
Finde eine maximale Menge der kompatiblen Vorlesungen.

Wir nehmen an, daß die Vorlesungen so sortiert sind daß

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

gilt.

Sei $s = (s_1, ..., s_n)$ und $f = (f_1, ..., f_n)$.

Algorithmus 2.9 $\textsc{Schedule}(s, f)$;

$A := \{1\}$
$j := 1$
for $i = 2$ **to** n
    do **if** $s_i \geq f_j$
        then $A := A \cup \{i\}$
            $j := i$
return A

Die Laufzeit von dem $\textsc{Schedule}$-Algorithmus ist $\Theta(n)$ falls die Endezeiten $\{f_i\}$ sortiert sind.

Theorem 2.1 [1] *Der* $\textsc{Schedule}$-*Algorithmus ergibt eine Lösung des Scheduleproblems.*

Die Greedy-Methode erlaubt auch z.B. die minimale Anzahl der Säle zu finden, die gebraucht werden, um ALLE Vorlesungen zu schedulen.

NICHT JEDE Greedy-Methode ermöglicht aber das optimale Schedul zu finden !

Elemente der Greedy-Methode:

- Optimale Strategie für jeden Schritt
- Optimale Teilstruktur der Teilprobleme

6. Das Rucksackpakungsproblem

Sei $S = (s_1, ..., s_n)$ die Menge der n Objekte und jedes Objekt s_i enthält das Gewicht w_i und der Preis p_i, $i = 1, ..., n$. Weiterhin, sei $W > 0$ das maximale Gewicht das der Rucksack erlaubt.

0-1 VERSION:
Finde die Pakungsmenge $P = \{s_{i_1}, ..., s_{i_k}\} \subseteq S$, so daß

$$\sum_{j=1}^{k} w_{i_j} \leq W \qquad \text{und} \qquad \sum_{j=1}^{k} p_{i_j} \text{ ist maximal.}$$

FRAKTIONALE VERSION:
Finde die Proportionalkoeffizienten c_i mit $0 \leq c_i \leq 1$ für $i = 1, ..., n$, so daß

$$\sum_{j=1}^{n} c_i w_i \leq W \qquad \text{und} \qquad \sum_{j=1}^{n} c_i p_i \text{ ist maximal.}$$

Theorem 2.2 [1] *Es gibt eine Greedy-Methode zur Lösung der fraktionalen Version des Rucksackpakungsproblems.*

7. Matroide

Wann funktioniert die Greedy-Methode eingentlich ?

Definition 2.5 *Ein Matroid ist das Paar* $M = (S, \mathcal{I})$*, wobei* $S \neq \emptyset$ *eine endliche Menge der Elemente ist und* $\mathcal{I} \subseteq 2^S$ *ist eine nichtleere Menge der Teilmengen von* S (unabhängige Teilmengen)*, so daß:*
a. $[B \in \mathcal{I}] \wedge [A \subseteq B] \Longrightarrow A \in \mathcal{I}$.
b. $[A, B \in \mathcal{I}] \wedge [|A| < |B|] \Longrightarrow \exists x \in B \backslash A$ *mit* $A \cup \{x\} \in \mathcal{I}$.
Das Element x *heißt* Extention *von* A.

Beispiel:
Das Paar $(S, \mathcal{I}_k)$ ist ein Martoid, wobei $\mathcal{I}_k$ die Menge alle Teilmengen von S ist die höhstens k Elemente enthalten.

Beispiel:
Sei S die Menge der Spalten einer $n \times n$ Matrix über $\mathbf{R}$ und $A \subseteq S$. Setzen wir $A \in \mathcal{I} \Longleftrightarrow$ die Spalten von A sind linear unabhängig. Dann ist $(S, \mathcal{I})$ ein Matroid.

Definition 2.6 *Sei* $(S, \mathcal{I})$ *ein Matroid und für* $A \in \mathcal{I}$ *gibt es keine Extention. Dann heißt die Teilmenge* A maximal.

Behauptung 2.2 [1] *Seien* A *und* B *maximale unabhängige Teilmengen des Matroids. Dann gilt* $|A| = |B|$.

Definition 2.7 *Ein Matroid $M = (S, \mathcal{I})$ heißt* gewichtet *falls es eine Abbildung $w : S \mapsto \mathbf{R}^+$ gibt.*

Für $A \subseteq S$ definieren wir das Gewicht von A:

$$w(A) = \sum_{x \in A} w(x).$$

Definition 2.8 *Die unabhängige Menge $A \in \mathcal{I}$ von einem gewichteten Matroid heißt* optimal *falls ihre Gewicht maximal ist.*

Das Matroidsproblem:

Gegeben ein gewichtetes Matroid M. Finde seine optimale unabhängige Menge.

Wir nehmen an, daß die Elemente von $S = \{x_1, ..., x_n\}$ so sortiert sind, daß $w(x_1) \geq w(x_2) \geq \cdots \geq w(x_n)$ gilt.

Algorithmus 2.10 $\textsc{Greedy}(M, w)$;

$A := \emptyset$
for $i = 1$ **to** n
  do **if** $A \cup \{x_i\} \in \mathcal{I}$
    then $A := A \cup \{x_i\}$
return A

Theorem 2.3 [1] *Der $\textsc{Greedy}$-Algorithmus ergibt eine optimale unabhängige Menge.*

Der Beweis begründed sich auf folgende 3 Lemmata:

Lemma 2.1 [1] *Sei $M = (S, \mathcal{I})$ ein gewichtetes Matroid und $x \in S$ ein Element mit maximalem Gewicht, so daß $\{x\} \in \mathcal{I}$. Wenn x existiert, dann gibt es eine optimale unabhängige Menge $A \subseteq \mathcal{I}$ mit $x \in A$.*

Lemma 2.2 [1] *Sei $M = (S, \mathcal{I})$ ein Matroid und sei $\{x\} \in S$ nicht die Extension von $\{\emptyset\}$. Dann ist x nicht die Extention von irgendeiner unabhängigen Menge $A \in \mathcal{I}$.*

Lemma 2.3 [1] *Sei x das erste Element das $Greedy$ auswählt. Dann reduziert sich das Matroidsproblem für M auf das Matroidsproblem für Matroid $M' = (S', \mathcal{I}')$ mit der auf S' reduzierten Gewichtsfunktion, wobei*

$$\begin{aligned} S' &= \{y \in S \mid \{x, y\} \in \mathcal{I}\}, \\ \mathcal{I}' &= \{B \subseteq S - \{x\} \mid B \cup \{x\} \in \mathcal{I}\}. \end{aligned}$$

8. Graphen und ihre Spannbäume

Definition 2.9 *Ein Graph G ist ein Paar (V, E), wobei V die* Knotenmenge *und*

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

die Kantenmenge *ist.*

Definition 2.10 *Eine* Kantenfolge *in einem Graph G ist eine endliche Folge von Kanten der Form*

$$(v_0, v_1),\ (v_1, v_2),\ \cdots\ (v_{l-1}, v_l),$$

wobei $l \geq 1$ seine Länge *ist.*

Die Kantenfolge heißt

- Weg, *falls alle Kanten verschieden sind.*
- einfacher Weg, *falls alle Knoten $v_0, ..., v_l$ verschieden sind.*
- Kreis, *falls sie ein Weg ist mit $v_0 = v_l$ und $l \geq 2$.*
- einfacher Kreis, *falls sie ein Weg ist mit $v_0 = v_l$ und $l \geq 2$, wobei die Knoten $v_0, ..., v_{l-1}$ verschiedene sind.*

Definition 2.11 *Ein Graph G heißt* zusammenhängend, *falls für jedes $u, v \in V$ ein Weg von u nach v existiert.*

Definition 2.12 *Ein Graph heißt* Wald, *falls er kein Kreis enthält. Ein zusammenhängender Wald heißt* Baum.

Definition 2.13 *Ein Graph $H = (V(H), E(H))$ ist ein* Teilgraph *von $G = (V(G), E(G))$, falls $V(H) \subseteq V(G)$ und $E(H) \subseteq E(G)$ gilt.*

Definition 2.14 *Wir nennen einen Teilgraph H von G* Spanngraph *von G, falls $V(H) = V(G)$ gilt. Falls H ein Baum ist, nennen wir ihn* Spannbaum.

Definition 2.15 *Ein Graph $G = (V, E)$ heißt* gewichtet, *falls es eine Abbildung $w : E \mapsto \mathbf{R}^+$ gibt. Für ein Teilgraph H von G definieren wir sein Gewicht als*

$$w(H) = \sum_{e \in E(H)} w(E).$$

Das Spannbaumproblem:

Gegeben ein gewichteter zusammenhängender Graph G. Konstruiere einen Spannbaum mit maximalem Gewicht.

Sei $A \subseteq E(G)$. Definieren wir die Menge $\mathcal{I}(G)$ durch:
$A \in \mathcal{I}(G) \iff A$ enthält kein Kreis.

Theorem 2.4 *Sei $G = (V(G), E(G))$ ein Graph. Dann ist $(E(G), \mathcal{I}(G))$ ein Matroid* (graphischer Matroid).

Korollar 2.1 *Das Spannbaumproblem ist mit der Hilfe der Greedy-Methode lösbar.*

Die Greedy-Methode erlaubt auch ein Spannbaum mit minimalen Gewicht zu konstruiren (Kruskal-Algorithmus).

9. Kruskal-Algorithmus

Sei M eine Menge der Elemente und $A, B \subseteq M$, $A \cap B = \emptyset$.

Wir gebrauchen die folgenden Prozeduren:

$\textsc{Make-Set}(v)$: Konstruiert die neue Menge $\{v\}$ für $v \in M$.

$\textsc{Find-Set}(v)$: Findet die Menge $A \subseteq M$ mit $v \in A$.

$\textsc{Union}(u, v)$: Vereinigt die Mengen $A = \textsc{Find-Set}(u)$ und $B = \textsc{Find-Set}(v)$ falls $A \cap B = \emptyset$ gilt.

Algorithmus 2.11 $\textsc{MST-Kruskal}(G, w)$;

$B := \emptyset$
for all $v \in V(G)$
  do $\textsc{Make-Set}(v)$
Sortiere die Kanten E nach nichtfallenden Gewicht w
for all $(u, v) \in E$
  do **if** $\textsc{Find-Set}(u) \neq \textsc{Find-Set}(v)$
    then $B := B \cup (u, v)$
      $\textsc{Union}(u, v)$
return B

10. Prim-Algorithmus

Sei $r \in V(G)$. Wir benutzen die Heap-Prozedure $\textsc{Extract-Min}(Q)$.

Algorithmus 2.12 $\text{MST-}\textsc{Prim}(G, w, r)$;

$Q := V(G)$
 for jedes $u \in Q$
  do $key[u] := \infty$
$key[r] := 0$
$\pi[r] :=$ nil

 while $Q \neq \emptyset$
  do $u := \textsc{Extract-Min}(Q)$
   for jedes $v \in Adj[u]$
    do **if** $v \in Q$ und $w(u, v) < key[v]$
     then $\pi[v] := u$
      $key[v] := w(u, v)$

Im jeden Schritt konstruiert der $\text{MST-}\textsc{Prim}$ ein Baum, der ein Teilbaum eines minimalen Spannbaumis ist.

11. Huffman Kode

Sei C ist eine Menge von Symbolen, die aus f_i Kopien des Symbols a_i bestehe mit $i = 1, ..., n$. Wir wollen die Symbole $\{a_i\}$ mit binärische Folgen $\{w_i\}$ kodieren, so daß jede Folge w_i kein Prefix von einer anderen Folge w_i ist. Wir definieren:

$$L(C) = \sum_{i=1}^{n} f_i \cdot |w_i|.$$

Das Kodierungsproblem:
Gegeben $\{f_i\}$. Finde eine Prefixkodierung C mit $L(C)$ minimal (optimale Kodierung).

Für $n = 2$ ist das Problem trivial. Wir nehmen an, daß

$$f_1 \geq f_2 \geq \cdots \geq f_n \qquad \text{und} \qquad n \geq 3$$

gilt.

Behauptung 2.3 [1] *Es gibt eine optimale Prefixkodierung, so daß $|w_{n-1}| = |w_n|$ gilt und die Folgen w_{n-1} und w_i sind nur in einem Bit verschieden.*

Theorem 2.5 [1] (Optimales Teilstruktur)
Sei C eine optimale Prefixkodierung für die Folge $f_1, ..., f_n$ und C' eine optimale Prefixkodierung für die Folge $f_1, ..., f_{n-2}, f_{n-1} + f_n$. Dann gilt:

$$L(C) = L(C') + f_{n-1} + f_n.$$

Dieses Theorem ergibt eine Greedy-Methode für die Konstruierung der optimalen Prefixkodierung.

Sei $f'_i = f_i/|C|$, $i = 1, ..., n$. Dann gilt:

$$f'_1 \geq f'_2 \geq \cdots \geq f'_n \qquad \text{und} \qquad \sum_{i=1}^{n} f'_i = 1.$$

Die Huffman-Methode besteht aus folgenden Schritten:

1. **while** $|f'| > 2$ **do**
   a. Nehme zwei kleinste Elemente f'_{n-1} und f'_n der Folge f' und konstruiere die neue Folge $(f'_1, ..., f'_{n-2}, f'_{n-1} + f'_n)$.
   b. Sortiere die neue Folge in nicht steigender Ordnung.
2. Kodiere die Folge (g'_1, g'_2) mit 0 ind 1.
3. Befolge die Schritte aus 1a. in umgekerter Richtung:
   while $|g'| < n$ **do**
   Sei die Folge g' in der Form: $(g'_1, ..., f'_{m-1} + f'_m, ..., g'_{m-2})$ und die korrespondierte Kodierung ist $a_1, ..., a_i, ..., a_m$, wobei $m = |g'| \leq n - 1$.
   Konstruiere die Folge $(g'_1, ..., g'_{i-1}, g'_{i+1}, ..., g'_m, f'_{m-1}, f'_m)$ und kodiere sie mit $a_1, ..., a_{i-1}, a_{i+1}, ..., a_m, a_i0, a_i1$.

Beispiel:

Sei $f' = (0.20, 0.20, 0.19, 0.12, 0.11, 0.09, 0.09)$.

0.20	0.20	0.23	0.37	0.40	0.60	0	1	00	01	10	10
0.20	0.20	0.20	0.23	0.37	0.40	1	00	01	10	11	11
0.19	0.19	0.20	0.20	0.23			01	10	11	000	000
0.12	0.18	0.19	0.20					11	000	001	010
0.11	0.12	0.18							001	010	011
0.09	0.11									011	0010
0.09											0011

3 Lektion 3: Einige Graphenalgorithmen

1. DFS und Starke Zusammenhangskomponenten
2. Topologische Sortierung
3. Kürzeste Wege in Graphen
4. Färbung von Graphen

1. Repräsentierung von Graphen

Sei $G = (V, E)$ ein Graph und seien $u, v \in V$.

- Adjazensliste.

  $Adj[u]$ ist eine Liste, das aus allen zu u adjazenten Knoten besteht.
  Speicherplatz: $O(|V| + |E|)$.
  Nachteil: es gibt keinen schnellen Weg zu prüfen ob $(u, v) \in E$ gilt.

- Adjazensmatrix $A(G) = a_{ij}$.

  $$a_{ij} = \begin{cases} 1, & \text{falls } (v_i, v_j) \in E \\ 0, & \text{ansonsten} \end{cases}$$
  Speicherplatz: $O(|V|^2)$.
  Vorteil: man kann Speicherplatz sparen indem jeder Eintrag a_{ij} nur durch ein Bit kodiert wird.

In der gleichen Art und Weise kann man die beiden Repräsentierungen auch für gerichtete und gewichtete Graphen benutzen.

2. Die Tiefensuche

Diese Methode besucht alle Knoten und alle Kanten von G und färbt die Knotenmenge in weiß, grau und schwarz.

Alle Knoten von G sind initial weiß gefärbt. Sobald wir ein Knoten v zum ersten mal besuchen, farben wir ihn grau. Falls alle zu v adjazente Knoten schon untersucht sind, bekommt v die schwarze Farbe.

Wir weisen jeden $v \in V$ drei Labels zu: $d[v]$, $f[v]$ und $\pi[v]$:

$d[v]$: der Zeitpunkt wann v die Farbe grau bekam

$f[v]$: der Zeitpunkt wann v die Farbe schwarz bekam

$\pi[v]$: der Vorkommer von v in DFS.

Es gilt,

$$\begin{aligned} d[v], f[v] &\in \{1, \ldots, 2|V|\}, \\ d[v] &< f[v] \end{aligned}$$

für jedes $v \in V$.

Die Methode konstruiert einen Spannwald W von G, wobei

$$E(W) = \{(\pi[v], v) \mid v \in V, \text{ und } \pi[v] \neq \text{nil}\}.$$

Die Kanten aus $E(W)$ heißen Baumkanten.

Algorithmus 3.1 $\mathrm{DFS}(G, s)$;

for all $u \in V$
    do $color[u] :=$ white
        $\pi[u] :=$ nil
$zeit := 0$
for all $u \in V$ /* sei s der erste Knoten */
    do **if** $color[u] =$ white
        then $\mathrm{DFS\text{-}V{\scriptsize ISIT}}(u)$

$\mathrm{DFS\text{-}V{\scriptsize ISIT}}(u)$;
$color[u] :=$ grau
$d[u] := zeit := zeit + 1$
for all $v \in Adj[u]$
    do **if** $color[v] =$ white
        then $\pi[v] := u$
            $\mathrm{DFS\text{-}V{\scriptsize ISIT}}(v)$
$color[u] :=$ schwarz
$f[u] := zeit := zeit + 1$

Die Laufzeit von DFS ist $\Theta(|V| + |E|)$.

3. Eigenschaften der Tiefensuche

Theorem 3.1 *Sei $G = (V, E)$ ein (gerichteter oder ungerichteter) Graph und $u, v \in V$, $u \neq v$. Dann ist genau eine der folgenden 3 Situationen möglich:*

- *Die Intervale $[d[u], f[u]]$ und $[d[v], f[v]]$ schneiden sich nicht.*
- *Das Interval $[d[u], f[u]]$ ist ein Teilinterval des $[d[v], f[v]]$ und u ist ein Nachfolger von v im Tiefensuchebaum.*
- *Das Interval $[d[v], f[v]]$ ist ein Teilinterval des $[d[u], f[u]]$ und v ist ein Nachfolger von u im Tiefensuchebaum.*

Korollar 3.1 *Der Knoten v ist ein Nachfolger von u genau dann, wenn*

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

gilt.

Theorem 3.2 *Seien $G = (V, E)$ ein (gerichteter oder ungerichteter) Graph, W sein Tiefensuchwald und $u, v \in V$. Dann ist v ein Nachfolger von u genau dann, wenn zum Zeitpunkt $d[u]$ ein Weg von u nach v existiert, der nur aus weißen Knoten besteht.*

4. Starke Zusammenhangskomponenten

Definition 3.1 *Sei $G = (V, E)$ ein gerichteter Graph. Eine starke Zusammenhangskomponente ist eine maximale Knotenmenge $K \subseteq V$, so daß für alle $u, v \in K$ ein gerichteten Weg von u nach v und von v nach u existiert.*

Das Problem:
Gegeben $G = (V, E)$. Konstruiere die Partitionierung von V in starken Zusammenhangskomponenten.

Wir bezeichnen $G^T = (V, E^T)$, wobei

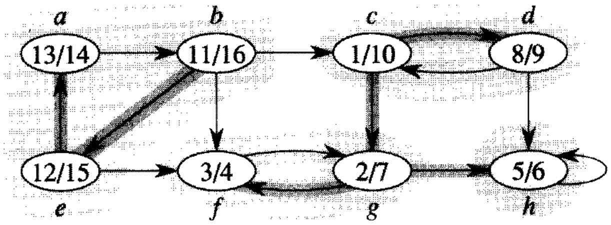
$$E^T = \{(u \to v) \mid (v \to u) \in E\}.$$

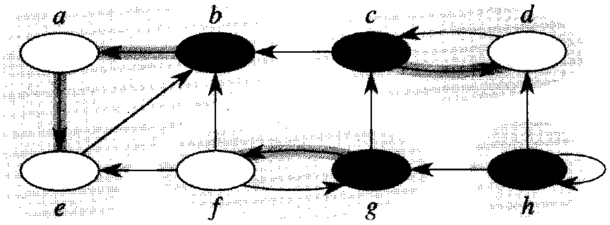
Algorithmus 3.2 $\mathrm{ZHK}(G)$;

1. Rufe $\mathrm{DFS}(G)$ um die Zahlen $\{f[u]\}$ zu berechnen
2. Konstruiere G^T
3. Rufe $\mathrm{DFS}(G^T)$, wobei die Knoten in der Ordnung der fallenden Werte von $f[u]$ ausgewählt sind
4. Output die Knotenmengen der Bäume des Tiefensuchwaldes von G^T als die starke Zusammenhangskomponenten

Die Laufzeit von ZHK ist $\Theta(|V| + |E|)$.

Theorem 3.3 [1]. *Der $\mathrm{ZHK(G)}$ ergibt die Partitionierung von V in starken Zusammenhangskomponenten.*





Lemma 3.1 *Sei K eine starke Zusammenhangskomponente von G und $u, v \in K$. Dann liegt jeder Knoten auf einem gerichteten Weg von u nach v und von v nach u in K.*

Beweisskizze:
$u, v \in K \Rightarrow \exists$ Weg P von u nach v.
Sei $w \in P$. Dann existiert ein Weg $w \to v \to u$.
$\Rightarrow w, u \in K$. $\square$

Lemma 3.2 *Sei K eine starke Zusammenhangskomponente von G und W ein Tiefensuchwald. Dann gibt es einen Baum $T = (V', E')$ von W, so daß $K \subseteq V'$ gilt.*

Beweisskizze:
Sei $r \in K$ der erste Knoten der in der DFS besucht wurde.
$\forall\, w \in K\ \exists$ ein Weg $r \to w$ der (z. Zeitpunkt $d[r]$) aus weißen Knoten besteht.

Nach T. 3.2 ist w ein Nachfolger von r in T. $\square$

Definition 3.2 *Wir bezeichnen mit $\phi(u)$ einen Knoten $w \in V$ für den gilt: $\exists$ ein gerichteter Weg von u nach w (in G) und $f[w]$ ist maximal.*

Bemerkungen: $f[u] \leq f[\phi(u)]$
$u \to v \Rightarrow f(\phi(u)) \geq f(\phi(v))$
$\{w : u \to w\} \supseteq \{w : v \to w\}$.

Lemma 3.3 *$\phi(u)$ ist ein Vorgänger von u (in DFS).*

Beweisskizze:
O.B.d.A. sei $\phi(u) \neq u$. Wir betrachten den Zeitpunkt $d[u]$.
$[color(\phi(u)) = schwarz] \Rightarrow f[\phi(u)] < f[u]$. Widerspruch.
$[color(\phi(u)) = grau] \Rightarrow \phi(u)$ ist ein Vorgänger von u.

Angenommen, $color(\phi(u)) = weiß$. Sei $W = u \to \phi(u)$.

a. Sei jeder Knoten vom W ist weiß $\Rightarrow$ (T. 3.2) $\phi(u)$ ist ein Nachfolger von $u \Rightarrow f[\phi(u)] < f[u]$. Widerspruch.

b. $\exists$ $w \in W$ mit $color(w) \neq weiß$ und jeder andere Knoten von dem Teilweg $w \to \phi(u)$ ist weiß.
$\Rightarrow color(w) = grau$
$\Rightarrow$ (T. 3.2) $\phi(u)$ ist ein Nachfolger von w
$\Rightarrow f[\phi(u)] < f[w]$. Widerspruch zur Definition von $\phi(u)$.

Korollar 3.2 *Die Knoten u und $\phi(u)$ sind im derselben starken Zusammenhangskomponente.*

Beweisskizze: Definition $\phi(u) \Rightarrow \exists$ ein Weg $u \to \phi(u)$.
$\phi(u)$ ist ein Vorgänger von $u \Rightarrow \exists$ ein Weg $\phi(u) \to u$. $\square$

Lemma 3.4 *Die Knoten $u, v \in V$ sind in derselben starken Zusammenhangskomponente genau dann wenn $\phi(u) = \phi(v)$.*

Beweisskizze:

"$\Rightarrow$" Sei $u, v \in K$. $\Rightarrow \forall\ w \in K\ \exists$ ein Weg $u \to w$ und ein Weg $v \to w \Rightarrow \phi(u) = \phi(v)$.

"$\Leftarrow$" Angenommen, $\phi(u) = \phi(v)$.
$\Rightarrow u$ und $\phi(u)$ sind in derselben ZHK (K. 3.2).
Analog, sind v und $\phi(v)$ in derselben ZHK
$\Rightarrow u$ und v sind in derselben ZHK. $\square$

<u>Beweisskizze für Theorem 3.3</u>. Induktion über die Anzahl der Tiefensuchbäume in DFS auf G^T.

Die Induktionsvoraussetzung ist richtig im Zeitpunkt 0. Sei T ein Tiefensuchbaum und sei r seine Wurzel. Weiterhin, sei

$$C(r) = \{v \in V \mid \phi(v) = r\}.$$

Wir zeigen: $u \in T \iff u \in C(r)$.

"$\Leftarrow$" Nach L. 3.2, ist jeder $u \in C(r)$ in demselben Baum. Da $r \in C(r)$ und $r \in T \Rightarrow u \in T$ gilt $\forall\ u \in C(r)$.

"$\Rightarrow$" Sei $w \in V$ mit $f[\phi(w)] \neq f[r]$. Wir zeigen $w \notin T$.
Falls $f[\phi(w)] > f[r] \Rightarrow$ (Induktion) $\Rightarrow w \notin T$.
Sei $f[\phi(w)] < f[r]$. Angenommen, $w \in T$. Dann

$$\exists \text{ Weg } w \overset{G}{\to} r \quad \Rightarrow \quad f[\phi(w)] \geq f[\phi(r)] = f[r].$$

Widerspruch. $\square$

5. Klassifizierung der Kanten

Sei $G = (V, E)$ ein Graph. Die Tiefensuche partitioniert die Kantenmenge E durch:

Baumkanten : die Kanten des Spannwaldes W;

Rückwärtskanten : die Kanten (u, v), wo v ein Vorgänge von u im einen Tiefensuchbaum ist. Alle Schleifen sind Rückwärtskanten;

Vorwärtskanten : die Kanten (u, v), die nicht Baumkanten sind und wo v ein Nachfolger von u im einen Tiefensuchebaum ist;

Kreuzkanten : alle anderen Kanten von G.

Sei $color(u) = grau$ und die Kante $e = (u, v) \in E$ wird zum ersten mal untersucht. Dann gilt:

- $[color(v) = wei\ss] \Longrightarrow e$ ist eine Baumkante.
- $[color(v) = grau] \Longrightarrow e$ ist eine Rückwärtskante.
- $[color(v) = schwarz] \Longrightarrow e$ ist eine Vorwärtskante oder eine Kreuzkante.

Theorem 3.4 *Sei G ein ungerichteter Graph. Dann ist jede Kante von G in jeder Tiefensuche entweder eine Baumkante oder eine Rückwärtskante.*

6. Topologische Sortierung

Sei $G = (V, E)$ ein DAG (Directed Acyclic Graph) (d.h. G ist ein gerichteter Graph der keinen gerichteten Kreis enthält) ohne Schleifen.

Das Problem:
Finde eine Numerierung $\psi : V \mapsto \{1, \ldots, |V|\}$, so daß:

$$(u \to v) \in E \qquad \Rightarrow \qquad \psi(u) < \psi(v)$$

gilt.

Algorithmus 3.3 $\textsc{Top-Sort}(G)$;

1. Rufe $\mathrm{DFS}(G)$ um die Zahlen $\{f[u]\}$ zu berechnen.
2. Plaziere u vorne in der Liste so bald $f[u]$ bekannt ist;
 return die Liste

Theorem 3.5 [1]. *Sei G ein DAG. Dann ergibt der $\textsc{Top-Sort}(G)$ Algorithmus eine richtige topologische Sortierung von G.*

Der Beweis ist durch die folgende Lemma begründet:

Lemma 3.5 [1]. *Ein gerichteter Graph G ist ein DAG genau dann, wenn es keine Rückwärtskante im Tiefensuchewald von G gibt.*

Beweisskizze für Lemma 3.5:

"$\Rightarrow$" Angenommen, $\exists$ Rückwärtskante (u, v)
$\Rightarrow v$ ist ein Vorgänger von u
$\Rightarrow \exists$ Weg $v \to u$ in G und G enthält ein Kreis. Widerspruch.

"$\Leftarrow$" Angenommen, G enthält einen Kreis C.
Sei v der erste Knoten von C in DFS und $(u, v) \in E(C)$.
$\Rightarrow \exists$ "weißer" Weg $v \to u$ im Zeitpunkt $d[v]$. (T. 3.2)
$\Rightarrow u$ ist ein Nachfolger von v und (u, v) ist eine Rückwärtskante. Widerspruch. $\square$

Beweisskizze für Theorem 3.5:

Wir zeigen, daß $\forall\, u, v \in V\ (u \neq v)$
$(u, v) \in E \Longrightarrow f[u] > f[v]$.

Sei die Kante (u, v) zum ersten mal untersucht worden.
$\Rightarrow color(u) = grau$ und $color(v) \neq grau$.

($color(v) = grau \Rightarrow v$ ist ein Vorgänger von $u \Rightarrow$
(u, v) ist eine Rückwärtskante. Widerspruch.)

$\Rightarrow color(v) \in \{weiß, schwarz\}$.

a. [$color(v) = weiß$] $\Rightarrow v$ ist ein Nachfolger von u
$\Rightarrow f[v] < f[u]$.

b. [$color(v) = schwarz$] $\Rightarrow f[v] < f[u]$. $\square$

7. Kürzeste Wege in Graphen

- Grundlagen
- Teil I. Kürzester Weg
  - Der Dijkstra-Algorithmus
  - Der Bellman-Ford-Algorithmus
- Teil II. Alle kürzesten Wege
  - Die Matrizenmultiplikation
  - Der Floyd-Warshall-Algorithmus
  - Der transitive Abschluß

7a. Grundlagen

Sei $G = (V, E)$ ein gerichteter Graph und $w : E \mapsto \mathbf{R}$ eine Gewichtsfunktion. Weiterhin sei $P = (v_0, \ldots, v_k)$ ein (gerichteter) Weg in G. Wir definieren

$$w(P) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Für $u, v \in V$ setzen wir

$$\delta(u,v) = \begin{cases} \min\limits_{P=(u \to v)} w(P), & \text{falls ein Weg } (u \to v) \text{ existiert} \\ \infty, & \text{sonst.} \end{cases}$$

Definition 3.3 *Ein Weg $P = (u \to v)$ heißt* kürzester Weg, *falls $w(P) = \delta(u, v)$ gilt.*

Problemstellungen:

Gegeben $G = (V, E)$ und eine Gewichtsfunktion w.

- Sei $s \in V$. Finde die kürzesten Wege von s nach allen anderen Knoten von G.
- Finde die kürzesten Wege zwischen allen Knotenpaaren $(u, v) \in V \times V$.

Wir repräsentieren einen Weg P durch seine Vorgängermenge $\{\pi[v]\}$ für $v \in P$, und definieren den Vorgängergraphen $G_\pi = (V_\pi, E_\pi)$, wobei

$$\begin{aligned} V_\pi &= \{v \in V \mid \pi[v] \neq \text{NIL}\} \cup \{s\} \\ E_\pi &= \{(\pi[v], v) \in E \mid v \in V_\pi - \{s\}\}. \end{aligned}$$

Teil I. Kürzester Weg

Angenommen G enthält keinen Kreis mit negativem Gewicht. Wir konstruieren einen Kürzesten-Wege-Baum $G' = (V', E')$:

- $V' = \{v \in V \mid \exists$ Weg $s \to v\}$.
- G' ist ein Baum und s ist seine Wurzel.
- $\forall v \in V'$ ist der einfache Weg $s \to v$ in G' auch ein kürzester Weg von s nach v in G.

Lemma 3.6 *Sei $(v_1, \ldots, v_k)$ ein kürzester Weg von v_1 nach v_k. Dann ist $(v_i, \ldots, v_j)$ ein kürzester Weg von v_i nach v_j, für alle $1 \leq i \leq j \leq k$.*

Lemma 3.7
a. Sei $(v_1, \ldots, v_k, u)$ ein Weg von v_1 nach v_k und $(v_k, u) \in E$. Dann gilt:

$$\delta(v_1, u) \leq \delta(v_1, v_k) + w(v_k, u).$$

b. Sei $(v_1, \ldots, v_k, u)$ ein kürzester Weg von v_1 nach u und $(v_k, u) \in E$. Dann gilt:

$$\delta(v_1, u) = \delta(v_1, v_k) + w(v_k, u).$$

Algorithmus 3.4 $\textsc{Initialize-SS}(G, w)$;

for each $v \in V$
  do $d[v] := \infty$
    $\pi[v] := \text{NIL}$
$d[s] := 0$

7b. Die Relaxation

Sei $(u, v) \in E$.

Algorithmus 3.5 $\textsc{Relax}(u, v, w)$;

if $d[v] > d[u] + w(u, v)$
  then $d[v] := d[u] + w(u, v)$
  $\pi[v] := u$

Angenommen, wir haben einen Graph $G = (V, E)$ auf dem die $\textsc{Initialize-SS}$ Prozedur angewandt werde. Nehmen wir weiterhin an, daß wir auf G eine Folge der $\textsc{Relax}$-Prozedur anwenden. Dann gelten die folgenden Behauptungen:

Lemma 3.8 *Sei $(u, v) \in E$. Dann gilt direkt nach der Anwendung der $\textsc{Relax}(u, v, w)$-Prozedur:*

$$d[v] \leq d[u] + w(u, v).$$

Lemma 3.9
a. Es gilt: $d[v] \geq \delta(s, v)$ für alle $v \in V$.
b. Falls $d[v] = \delta(s, v)$ gilt, dann kann keine weitere Relaxation den Wert von $d[v]$ verminden.

Beweisskizze:
a. Die Ungleichung gilt direkt nach der Initializierung, da $d[s] = 0 > \delta(s, s)$.

Sei v der erste Knoten für den der Aufruf der $\textsc{Relax}$-Prozedur $d[v] < \delta(s,v)$ liefert. Nach dem Aufruf hat man:

$$\begin{aligned} d[u] + w(u,v) &= d[v] \\ &< \delta(s,v) \\ &\leq \delta(s,u) + w(u,v) \quad \text{(nach L. 3.7a)} \end{aligned}$$

Widerspruch.

b. Da $d[v] \geq \delta(s,v)$ und die $\textsc{Relax}$-Prozedur erhöht die d-Werte nicht, folgt die Behauptung. $\square$

Lemma 3.10 *Sei $s \to u \to v$ ein kürzester Weg von s nach v in G und $(u,v) \in E$. Falls vor dem Aufruf der $\textsc{Relax}(u,v,w)$-Prozedur $d[u] = \delta(s,u)$ gilt, dann gilt danach $d[v] = \delta(s,v)$.*

Beweisskizze:
$d[u] = \delta(s,u)$ vor dem Aufruf der $\textsc{Relax}$-Prozedur
$\Rightarrow d[u] = \delta(s,u)$ nach dem Aufruf (L. 3.9b).

Es gilt:

$$\begin{aligned} d[v] &\leq d[u] + w(u,v) \quad \text{(L. 3.8)} \\ &= \delta(s,u) + w(u,v) \\ &= \delta(s,v) \qquad\qquad \text{(L. 3.7b).} \end{aligned}$$

Andererseits, nach L. 3.9a gilt $d[v] \geq \delta(s,v)$. $\square$

7c. Der Dijkstra-Algorithmus

Sei $w(u, v) \geq 0$ für alle $(u, v) \in E$.

Algorithmus 3.6 $\textsc{Dijkstra}(G, w, s)$;

$\textsc{Initialize-SS}(G, s)$
$S := \emptyset$
$Q := V(G)$
while $Q \neq \emptyset$
  do $u := \textsc{Extract-Min}(Q)$
    $S := S \cup \{u\}$
    for each $v \in Adj[u]$
      do $\textsc{Relax}(u, v, w)$

Die Laufzeit des $\textsc{Dijkstra}$-Algorithmus ist $O(|V|^2)$.

Theorem 3.6 *Sei $G = (V, E)$ ein Graph mit einer nicht negativen Gewichtsfunktion w. Dann gilt nach der Anwendung des $\textsc{Dijkstra}$-Algorithmus: $d[u] = \delta(s, u)$ für alle Knoten $u \in V$.*

Beweisskizze:
Wir zeigen, daß zum Zeitpunkt, wenn u in S eingefügt ist, $d[u] = \delta(s, u)$ gilt.

Nehmen wir an, daß ein u existiert, so daß $d[u] > \delta(s,u)$ gilt und sei u der erste dieser Art Knoten. Dann ist u von s erreichbar und $u \neq s$.

Sei P ein kürzester Weg von s nach u in G und $(x,y) \in E$ die erste Kante von P mit $x \in S$, $y \notin S$. Dann gilt $d[x] = \delta(s,x)$ und $d[y] = \delta(s,y)$ nach L. 3.10. Darüberhinaus gilt:

$$d[y] = \delta(s,y) \leq \delta(s,u) \leq d[u].$$

Nach der Auswahl von u gilt: $d[u] \leq d[y]$.
$\Rightarrow d[u] = \delta(s,u)$. Widerspruch. $\square$

Korollar 3.3 *Der Graph G_π, den der* DIJKSTRA-*Algorithmus konstruiert, ist ein Kürzester-Wege-Baum.*

Beweisskizze:
Wir folgen der Definition von der Kürzesten-Wege-Baum.

- $\delta(s,v)$ ist nur für die Knoten v die aus s erreichbar sind endlich. $d[v]$ ist endlich $\Leftrightarrow \pi[v] \neq \text{NIL}$.
- Angenommen es gibt 2 verschiedene Wege von s nach v:

  $$\begin{aligned} P_1 &= s \to u \to x \to z \to v \\ P_2 &= s \to u \to y \to z \to v, \end{aligned}$$

  wobei $(x,z),(y,z) \in E'$. Dann gilt: $x = \pi[z]$ und $y = \pi[z]$ $\Rightarrow x = y$. Widerspruch.

- Der Weg $P = (s \to v)$ ist ein kürzester Weg.
  Sei $P = v_0, \ldots, v_k$, wobei $s = v_0$ und $v = v_k$.

  Für $i = 1, \ldots, k$ gilt:

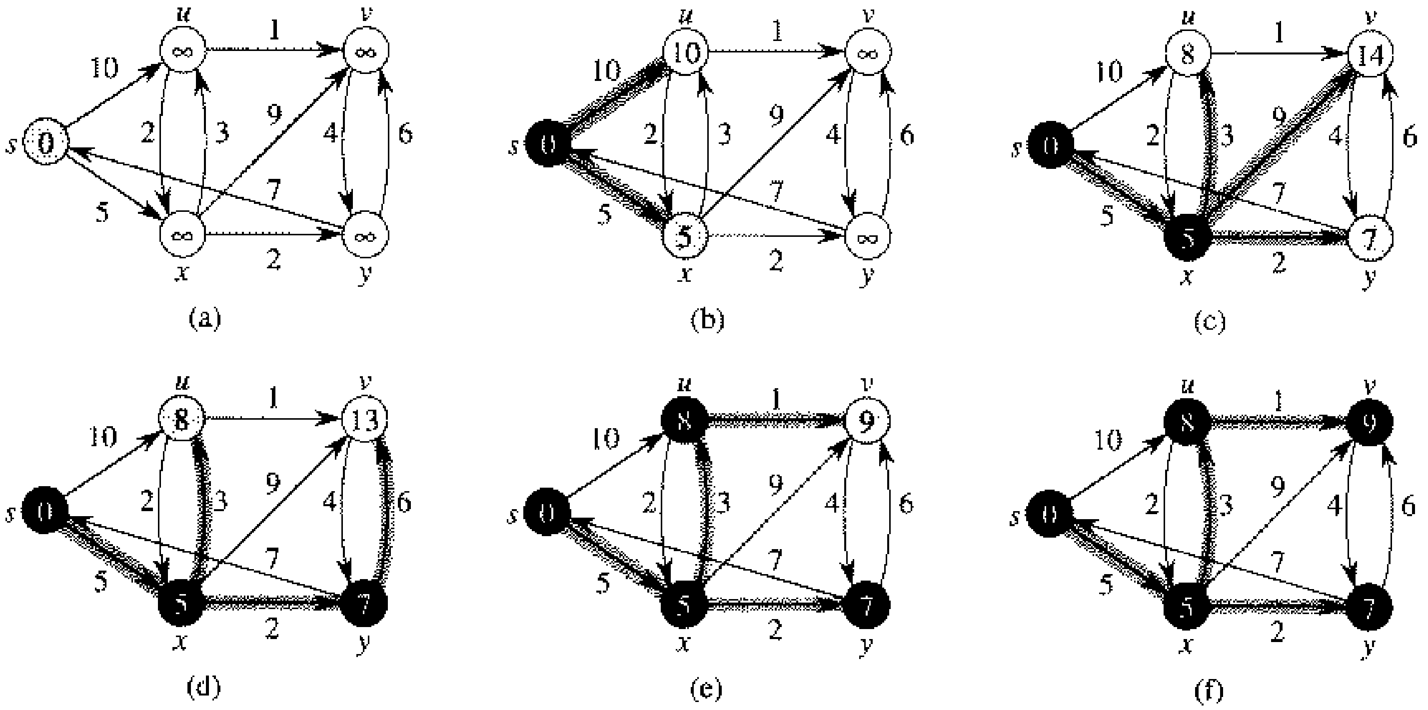
$$\begin{aligned} d[v_i] &= \delta(s, v_i) \\ d[v_i] &\geq d[v_{i-1}] + w(v_{i-1}, v_i). \end{aligned}$$

  $\Rightarrow w(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$
  und

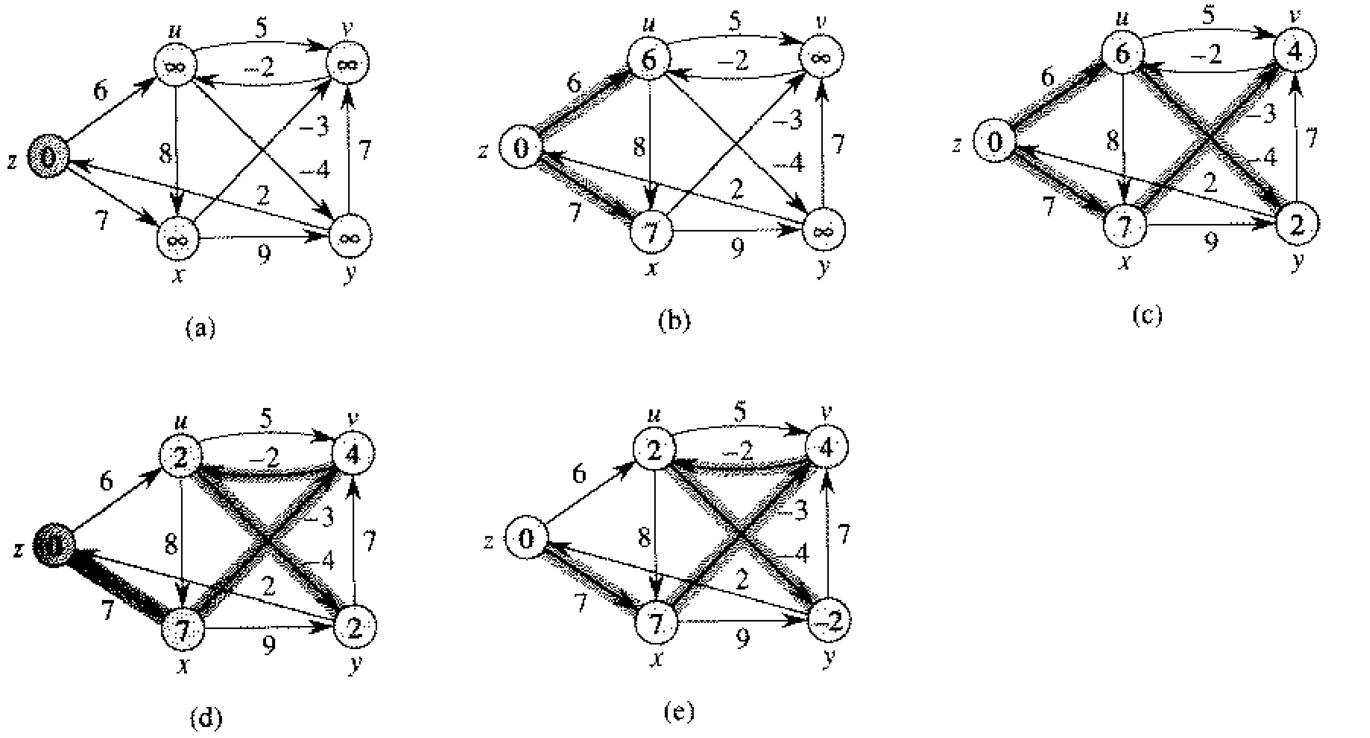
$$\begin{aligned} w(P) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \\ &\leq \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \\ &= \delta(s, v_k) - \delta(s, v_0) \\ &= \delta(s, v_k). \end{aligned}$$

  Andererseits gilt: $w(P) \geq \delta(s, v_k)$.
  Somit gilt: $w(P) = \delta(s, v_k)$ und P ist ein kürzester Weg.

Beispiel: der Dijkstra-Algorithmus



Beispiel: der Bellman-Ford-Algorithmus



7d. Der Bellman-Ford-Algorithmus

Der Algorithmus ergibt TRUE genau dann wenn G keinen Kreis mit negativem Gewicht enthält, der von s aus erreichbar ist.

Algorithmus 3.7 $\textsc{Bellman-Ford}(G, w, s)$;

$\textsc{Initialize-SS}(G, w, s)$
 for $i := 1$ **to** $|V| - 1$
  do **for** every $(u, v) \in E$
   do $\textsc{Relax}(u, v, w)$
 for every $(u, v) \in E$
  do **if** $d[v] > d[u] + w(u, v)$
   then **return** FALSE
 return TRUE

Die Laufzeit des $\textsc{Bellman-Ford}$-Algorithmus ist $O(|V| \cdot |E|)$.

Lemma 3.11 *Sei G ein Graph, der keinen negativen Kreis enthält, der von s aus erreichbar ist. Dann gilt nach der Anwendung des Algorithmus: $d[v] = \delta(u, v)$ für alle Knoten, die von s aus erreichbar sind.*

Theorem 3.7 *Falls G keinen von s aus erreichbaren negativen Kreis enthält, ergibt der Algorithmus* TRUE *und der Vorgänger-Graph ist ein Baum mit der Wurzel s. Falls G einen solchen Kreis enthält, ergibt der Algorithmus* FALSE.

Beweisskizze zu Lemma 3.11.

Sei v ein Knoten der von s aus erreichbar ist und $P = (v_0, \ldots, v_k)$ ein kürzester Weg von $v_0 = s$ nach $v_k = v$. Dann gilt: $k \leq |V| - 1$.

Wir zeigen per Induktion über i, daß $d[v_i] = \delta(s, v_i)$ nach den erste i Durchläufen der erste **for** -Schleife des Algorithmus gilt.

Induktionsbasis: $d[v_0] = \delta(s, v_0) = 0$.
Induktionsschritt: Wir nehmen an, daß $d[v_{i-1}] = \delta(s, v_{i-1})$ gilt. Da ist die Kante (v_{i-1}, v_i) in dem i-ten Durchlauf der erste **for** -Schleife des Algorithmus relaxiert wird, geht die Induktion mit Hilfe des Lemma 3.10 durch. $\square$

Bemerkung 3.1 *Ganz analog kann man beweisen, daß nach der Anwendung des* BELLMAN-FORD-*Algorithmus gilt: Der Knoten v ist von s aus erreichbar $\iff d[v] < \infty$.*

Beweisskizze zu Theorem 3.7.

Sei G ein Graph, der keinen von s aus erreichbaren negativen Kreis enthält. Dann folgt $d[v] = \delta(s, v)$ aus Lemma 3.11 und Bemerkung 3.1.

Der Vorgänger-Graph G_π ist ein Baum und s ist seine Wurzel (der Beweis ist analog zu Korollar 3.3).

Wir zeigen, daß der Algorithmus TRUE ergibt. Es gilt:

$$\begin{aligned} d[v] &= \delta(s, v) \leq \delta(s, u) + w(u, v) \quad \text{Lemma 3.9a} \\ &= d[u] + w(u, v). \end{aligned}$$

Somit ist die **if** -Bedingung nicht erfüllt und der Algorithmus liefert TRUE.

Angenommen, daß G einen Kreis $C = (v_0, \ldots, v_k)$ $(v_0 = v_k)$ mit negativem Gewicht enthält, der von s aus erreichbar ist. Dann gilt

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0.$$

Nehmen wir an, daß der Algorithmus TRUE ergibt. Dann gilt:

$$d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i), \qquad \text{für } i = 1, \ldots, k.$$

Somit erhalten wir:

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Da jedes $d[v_i]$ endlich ist (Bemerkung 3.1) und wegen $v_0 = v_k$

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}] \qquad \text{gilt,}$$

erhalten wir schließlich:

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Widerspruch. $\square$

Teil II. Alle kürzesten Wege

Da ist die Laufzeit des $\textsc{Dijkstra}$-Algorithmus $O(|V|^2)$, kann man alle kürzesten Wege in $O(|V|^3)$ Zeit konstruieren, falls die Gewichtsfunktion nicht negativ ist.

Falls ein Graph G keinen Kreis mit negativem Gewicht enthält, ergibt der $\textsc{Bellman-Ford}$-Algorithmus eine Lösung in $O(|V|^4)$ Zeit.

Wir konstruieren nun bessere Algorithmen.

Wir repräsentieren einen Graphen $G = (V, E)$ mit der Knotenmenge $V = \{v_1, \ldots, v_n\}$ durch die Adjazensmatrix w_{ij}, wobei

$$w_{ij} = \begin{cases} 0, & \text{falls } i = j \\ w(v_i, v_j), & \text{falls } i \neq j \text{ und } (v_i, v_j) \in E \\ \infty, & \text{falls } i \neq j \text{ und } (v_i, v_j) \notin E \end{cases}$$

Die kürzesten Wege sind durch die Vorgänger-Matrix $\Pi = \{\pi_{ij}\}$ definiert.

Wir nehmen an, daß der Graph G keinen Kreis mit negativem Gewicht enthält.

7e. Matrizenmultiplikation

Sei d_{ij}^m das minimale Gewicht eines Weges von v_i nach v_j, der höchstens aus m Kanten besteht. Dann gilt:

$$d_{ij}^0 = \begin{cases} 0, & \text{falls } i = j \\ \infty, & \text{falls } i \neq j. \end{cases}$$

Nach Lemmata 3.6 und 3.7 für $m \geq 1$ gilt:

$$\begin{aligned} d_{ij}^m &= \min\left\{d_{ij}^{m-1}, \min_{1 \leq k \leq n}\left(d_{ik}^{m-1} + w_{kj}\right)\right\} \\ &= \min_{1 \leq k \leq n}\left(d_{ik}^{m-1} + w_{kj}\right). \end{aligned}$$

Da jeder kürzeste Weg $v_i \to v_j$ höchstens aus $n-1$ Kanten besteht, gilt:

$$\delta(i,j) = d_{ij}^{n-1} = d_{ij}^n = d_{ij}^{n+1} = \ldots$$

Wir berechnen die Matrizen D^1, D^2,...,D^{n-1}, mit $D^1 = W := \{w_{ij}\}$, so daß $D^m = \{d_{ij}^m\}$ gilt.

Wir benutzen die folgende Prozedur, die die Matrix D^m aus den Matrizen D^{m-1} und W konstruiert.

Algorithmus 3.8 EXTEND-SHORTEST-PATH(D, W);

$n := \#Zeilen(D)$
/* Sei $D' = \{d'_{ij}\}$ eine $n \times n$ Matrix */
for $i = 1$ **to** n
  do **for** $j = 1$ **to** n
    do $d'_{ij} := \infty$
      for $k = 1$ **to** n
        do $d'_{ij} := \min\{d'_{ij},\, d_{ik} + w_{kj}\}$
return D'

Dieser Algorithmus ist analog zu den Algorithmus zu Matrizenmultiplikation:

Sei $C = A \cdot B$ mit $c_{ij} = \Sigma_{i=1}^{k} a_{ik} \cdot b_{kj}$.
Ersetzen wir:

$$
\begin{aligned}
d &\rightarrow a \\
w &\rightarrow b \\
d' &\rightarrow c \\
\min &\rightarrow + \\
+ &\rightarrow \cdot \\
\infty &\rightarrow 0,
\end{aligned}
$$

So erhält man aus den EXTEND-SHORTEST-PATH-Algorithmus den MM-Algorithmus:

Algorithmus 3.9 $\mathrm{MM}(A, B)$;

$n := \#Zeilen(A)$
/* Sei C eine $n \times n$ Matrix */
for $i = 1$ **to** n
  do **for** $j = 1$ **to** n
    do $c_{ij} := 0$
      for $k = 1$ **to** n
        do $c_{ij} := c_{ij} + a_{ik} \cdot b_{kj}$
return C

Die Laufzeit des MM-Algorithmus ist $O(n^3)$. Somit bekommen wir einen $O(n^4)$-Algorithmus für die Berechnung aller kürzesten Wege:

Algorithmus 3.10 $\mathrm{AKW}\text{-}1(W)$;

$D^1 := W$
for $m := 2$ **to** $n - 1$
  do $D^m :=$ Extend-Shortest-Path(D^{m-1}, W)
return D^{n-1}

Die Laufzeit des $\mathrm{AKW}\text{-}1$-Algorithmus ist $O(n^4)$.

Wir brauchen also eigentlich nur die Matrix D^{n-1} und die können wir schneller berechnen:

$$\begin{aligned}
D^1 &= W \\
D^2 &= W^2 = W \cdot W \\
D^4 &= W^4 = W^2 \cdot W^2 \\
D^8 &= W^8 = W^4 \cdot W^4 \\
&\vdots \\
D^{2^{\lceil \log(n-1) \rceil}} &= W^{2^{\lceil \log(n-1) \rceil}} = W^{2^{\lceil \log(n-1) \rceil - 1}} \cdot W^{2^{\lceil \log(n-1) \rceil - 1}}.
\end{aligned}$$

Da $2^{\lceil \log(n-1) \rceil} \geq n-1$, gilt somit $D^{2^{\lceil \log(n-1) \rceil}} = D^{n-1}$.

Algorithmus 3.11 $\text{AKW-2}(W)$;

$D^1 := W$
$m := 1$
while $n - 1 > m$
  do $D^{2m} := \textsc{Extend-Shortest-Path}(D^m, D^m)$
    $m := 2m$
return D^m

Die Laufzeit des AKW-2-Algorithmus ist $O(n^3 \log n)$.

7f. Der Floyd-Warshall-Algorithmus

Sei $P = (v_1, \ldots, v_l)$ ein kürzester Weg zwischen v_1 und v_l. Wir nennen die Knoten $v_2, \ldots, v_{l-1}$ (falls solche Knoten existieren) die inneren Knoten des Weges P.

Wir bezeichnen die Knotenmenge von G mit $\{1, 2, \ldots, n\}$. Für $i, j \in V$ betrachten wir die kürzesten Wege zwischen den Knoten i und j, wobei die inneren Knoten nur aus der Menge $\{1, \ldots, k\}$ sind. Sei P ein Weg von solcher Art.

- Sei $k \notin P$. Dann ist ein kürzester Weg von $i \to j$ bei dem alle inneren Knoten aus der Menge $\{1, \ldots, k-1\}$ stammen, auch ein kürzester Weg bei dem alle inneren Knoten aus der Menge $\{1, \ldots, k\}$ stammen.
- Sei $k \in P$. Wir zerteilen P in zwei Wege: $P_1 = (i \to k)$ und $P_2 = (k \to j)$. Dann ist P_1 ein kürzester Weg zwischen i und k bei dem alle inneren Knoten aus $\{1, \ldots, k-1\}$ stammen und gleiches gilt auch für P_2.

Wir bezeichnen durch d_{ij}^k das Gewicht des kürzesten Weges $i \to j$ bei dem alle inneren Knoten aus $\{1, \ldots, k\}$ stammen. Dann gilt:

$$d_{ij}^k = \begin{cases} w_{ij}, & \text{falls } k = 0 \\ \min\left\{d_{ij}^{k-1},\ d_{ik}^{k-1} + d_{kj}^{k-1}\right\}, & \text{falls } k \geq 1. \end{cases}$$

Wir repräsentieren diese Zeilen mittels eine Matrix $D^k = \{d_{ij}^k\}$, wobei $d_{ij}^n = \delta(i, j)$ für $1 \leq i \leq j \leq n$ gilt.

Algorithmus 3.12 $\textsc{Floyd-Warshall}(W)$;

$n := \#Zeilen(W)$
$D^0 := W$
for $k := 1$ **to** n
  do **for** $i := 1$ **to** n
    do **for** $j := 1$ **to** n
      do $d_{ij}^k := \min\{d_{ij}^{k-1},\, d_{ik}^{k-1} + d_{kj}^{k-1}\}$
return D^n

Die Laufzeit des $\textsc{Floyd-Warshall}$-Algorithmus ist $O(n^3)$.

Konstruktion der kürzesten Wege.

Wir konstruieren die Folge der Matrizen: $\Pi^0,\ldots,\Pi^n$ mit $\Pi^k = \{\pi_{ij}^k\}$, wobei π_{ij}^k ein Vorgänger des Knotens j auf einem kürzestem Weg von i nach j ist, bei dem alle inneren Knoten nur aus der Menge $\{1,\ldots,k\}$ stammen.

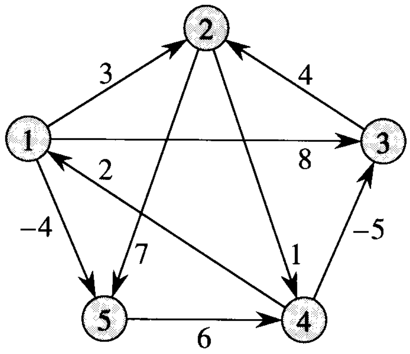
$$\pi_{ij}^0 = \begin{cases} \text{NIL}, & \text{falls } i = j \text{ oder } w_{ij} = \infty \\ i, & \text{falls } i \neq j \text{ und } w_{ij} < \infty. \end{cases}$$

Für $k \geq 1$ definieren wir:

$$\pi_{ij}^k = \begin{cases} \pi_{ij}^{k-1}, & \text{falls } d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1} \\ \pi_{kj}^{k-1}, & \text{falls } d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1}. \end{cases}$$

Die Elemente der Matrix Π^n ergeben für jeden Knoten j seinen Vorgänger π_{ij}^n auf einem kürzesten Weg von i nach j.

Beispiel:



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

7g. Der transitive Abschluß

Für einen gerichteten Graphen $G = (V, E)$ mit $V = \{1, \ldots, n\}$ definieren wir seinen Abschluß $G^* = (V, E^*)$, wobei

$$E^* = \{(i,j) \mid \exists \text{ Weg } i \to j \text{ in } G\}.$$

Triviale Lösung: setze die Gewichten aller Kanten auf 1 und wende den Floyd-Warshall-Algorithmus an. Dann gilt:

$$(i,j) \in E^* \quad \Leftrightarrow \quad d_{ij} < n.$$

Wir konstruieren einen etwas schnellere Algorithmus (der aber auch $\Theta(n^3)$ Schritten benötigt), allerdings weniger Speicher benutzt.

Wir definieren:

$$t_{ij}^0 = \begin{cases} 0 & \text{falls } i \neq j \text{ und } (i,j) \notin E \\ 1 & \text{falls } i = j \text{ oder } (i,j) \in E \end{cases}$$

$$t_{ij}^k = t_{ij}^{k-1} \vee \left(t_{ik}^{k-1} \wedge t_{kj}^{k-1}\right), \qquad \text{für } k \geq 1.$$

Somit gilt:

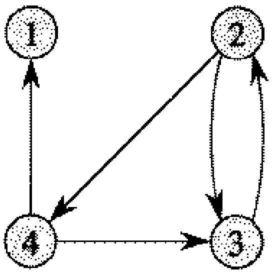
$$(i,j) \in E^* \quad \Leftrightarrow \quad t_{ij}^k = 1.$$

Der folgende Algorithmus berechnet die Matrizen $T^k = (t_{ij}^k)$ für $k = 1, \ldots, n$.

Algorithmus 3.13 Transitiver Abschluss(G);

for $i := 1$ **to** n
    do **for** $j := 1$ **to** n
        do **if** $i = j$ or $(i, j) \in E$
            then $t^0_{ij} := 1$
            else $t^0_{ij} := 0$
for $k := 1$ **to** n
    do **for** $i := 1$ **to** n
        do **for** $j := 1$ **to** n
            do $t^k_{ij} := t^{k-1}_{ij} \vee \left(t^{k-1}_{ik} \wedge t^{k-1}_{kj}\right)$
return T^n

Beispiel:



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

8. Färbung von Graphen

Definition 3.4 *Eine* Färbung *eines Graphen ist eine Belegung der Knoten mit Farben, so daß je zwei adjazente Knoten verschiedene Farben erhalten.*

Eine k-Färbung ist eine Färbung, die k verschiedene Farben $\{1, 2, \ldots, k\}$ verwendet.

Die chromatische Zahl $\chi(G)$ eines Graphen G ist die kleinste Zahl k, für welche der Graph eine k-Färbung besitzt.

Eine Färbung, die genau $\chi(G)$ Farben verwendet, nennt man eine minimale Färbung.

Es gilt:

$$\chi(K_n) = n, \qquad \chi(C_{2n}) = 2, \qquad \chi(C_{2n+1}) = 3.$$

Theorem 3.8 *Ein Graph G hat genau dann eine 2-Färbung, wenn er keinen Kreis mit ungerader Länge enthält.*

Beweisskizze:
"$\Longrightarrow$" Offensichtlich.

"$\Longleftarrow$" Der folgende Algorithmus konstruiert eine 2-Färbung eines Graphen G und ergibt FALSE falls G nicht 2-Färbbar ist.

Sei Q (eine FIFO)-Queue.

Algorithmus 3.14 BIPARTITE(G);

Wähle einen Knoten $v \in V$ aus und färbe ihn mit 1
$Q := \{v\}$
repeat **while** $Q \neq \emptyset$
  $u := head(Q)$
  $S := Adj[u]$
  for all $w \in S$
    do if $color[w] = color[u]$
      then Graph ist nicht bipartit. FALSE
  Färbe alle ungefärbten Knoten w in S mit $3 - color[w]$
    und füge sie in Q ein.
  $Q := Q - \{u\}$
return 2-Färbung

Sei $\omega(G)$ die Größe der maximalen Clique in G.

Theorem 3.9 *Es gilt*

$$\omega(G) \leq \chi(G) \leq \Delta(G) + 1.$$

Die untere Schranke ist offensichtlich. Der folgende Algorithmus produziert eine Färbung, die die obere Schranke erfüllt.

Algorithmus 3.15 FÄRBUNG(G);

Wähle $v \in V$ aus und farbe v mit 1
$V' := V - \{v\}$
repeat **while** $V' \neq \emptyset$
Wähle $u \in V'$ aus
$S := Adj[u]$
Färbe u mit minimaler unbenutzten Farbe in S
$V' := V' - \{u\}$
return maximale Farbe

Bemerkung 3.2 *Für jede zwei Zahlen Δ, k mit $2 \leq k \leq \Delta$ gibt es einen Graphen G mit maximalen Knotengrad Δ und $\chi(G) = k$.*

Allgemeine Methode zur Bestimmung von $\chi(G)$:

Definition 3.5 *Sei $G = (V, E)$ ein Graph und $a, b \in V$, $(a, b) \notin E$. Wir definieren:*
$G : ab = (V', E')$, wobei

$$\begin{aligned} V' &= (V - \{a, b\}) \cup \{z\}, \quad (z \notin V) \\ E' &= (E - \{(x, y) \mid x \in V,\, y \in \{a, b\}\}) \cup \{(x, z) \mid x \notin \{a, b\}, \\ &\qquad \text{und entweder } (x, a) \in E \text{ oder } (x, b) \in E\}. \end{aligned}$$

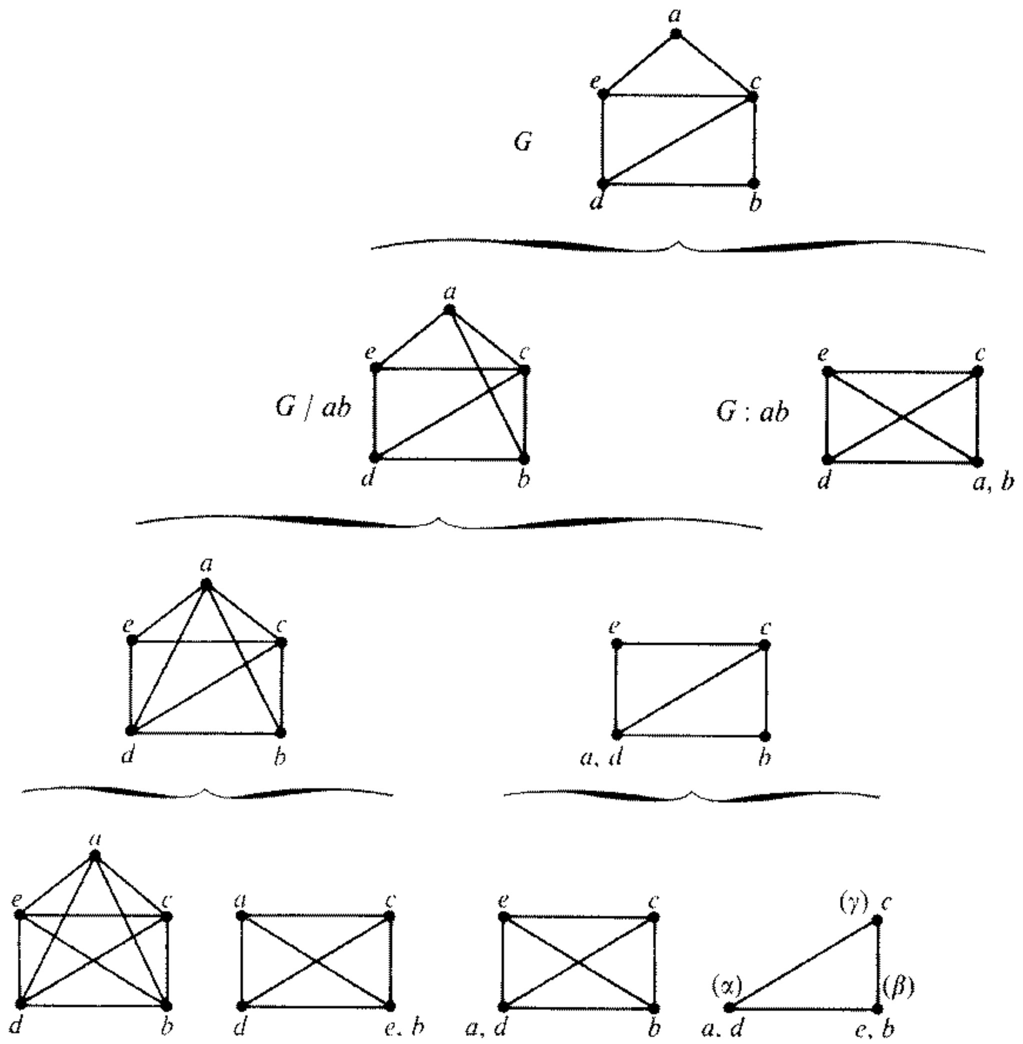
$G/ab = (V, E'')$, wobei $E'' = E \cup (a, b)$.

Eine Färbung von G wobei $color(a) = color(b)$ gilt, ergibt auch eine Färbung von $G : ab$, und eine Färbung von G, wobei $color(a) \neq color(b)$ gilt ergibt auch eine Färbung von G/ab.

Wiederhole diese zwei Operationen bis alle resultierenden Graphen Cliquen sind. Wenn die kleinste Clique aus k Knoten besteht, dann gilt $\chi(G) = k$.

Diese Methode besteht im allgemeinen aus exponentiell vielen Schritten.

Beispiel:



Definition 3.6 *Ein Graph $G = (V, E)$ heißt Intervallgraph, falls er als Schnittgraph einer Menge von Intervallen auf einer Geraden dargestellt werden kann.*

Theorem 3.10 *Sei G ein Intervallgraph. Dann gilt: $\chi(G) = \omega(G)$ und der Algorithmus 3.15 liefert eine Färbung, die aus $\omega(G)$ Farben besteht.*

Definition 3.7 *Ein Graph $G = (V, E)$ heißt planar, falls er so in die Ebene gezeichnet (eingebettet) werden kann, daß keine zwei Kanten sich kreuzen.*

Theorem 3.11 (Euler)
Sei $G = (V, E)$ ein planarer zusammenhängender Graph mit $|V| = n$, $|E| = e$ und f die Anzahl der Fenster. Dann gilt:

$$n - e + f = 2.$$

Korollar 3.4

1. *Sei $G = (V, E)$ ein planarer Graph mit $|V| = n$, $|E| = e$. Dann gilt:*
$$e \leq 3 \cdot n - 6.$$
2. *Sei $G = (V, E)$ ein planarer Graph mit $|V| \geq 4$. Dann gibt es einen Knoten mit Knotengrad ≤ 5.*

Theorem 3.12 *Jeder planare Graph ist 5-färbbar.*

4 Lektion 4: Flüsse in Netzwerken

1. Anleitung
2. Maximaler Fluß
   a. Die Ford-Fulkerson-Methode
   b. Das MAXFLOW-MINCUT-Theorem
   c. Der Edmonds-Karp-Algorithmus
   d. Der Dinic-Algorithmus
3. Anwendungen
   a. Zusammenhang von Graphen
   b. Matchings in bipartiten Graphen

1. Anleitung

Definition 4.1 *Ein* Netzwerk $N = (V, E, s, t)$ *ist ein gerichteter Graph* (V, E) *mit eine Funktion* $c : E \mapsto \mathbf{R}^{\geq 0}$ *und mit zwei ausgezeichneten Knoten* $s, t \in V$ (Quelle und Senke).

Falls $(u, v) \notin E$, erweitern wir $c(u, v)$ durch $c(u, v) = 0$.

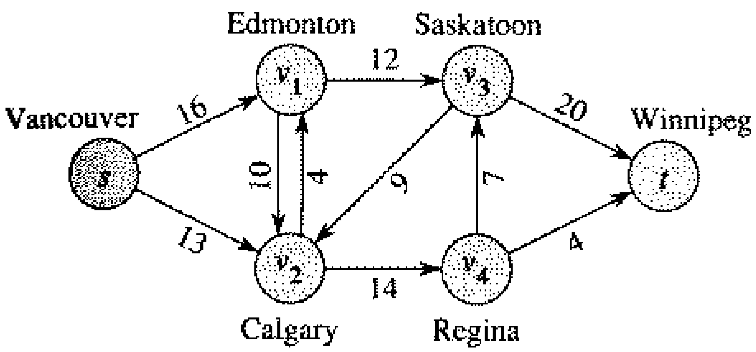
Definition 4.2 *Sei* $N = (V, E, s, t)$ *ein Netzwerk. Ein* Fluß *in* N *ist eine Funktion* $f : V \times V \mapsto \mathbf{R}$, *so daß:*

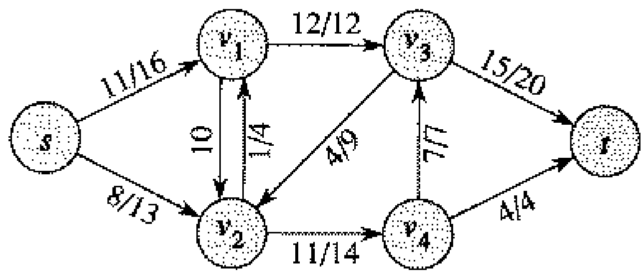
- $f(u, v) \leq c(u, v)$ *für alle* $u, v \in V$. *(Kapazitätsbedingung)*
- $f(u, v) = -f(v, u)$ *für alle* $u, v \in V$. *(Symmetrie)*
- $\sum_{v \in V} f(u, v) = 0$ *für alle* $u \in V - \{s, t\}$. *(Flußerhaltung)*

Der Wert $|f| = \sum_{v \in V} f(s, v)$ *heißt* totaler Fluß *von* f.

Die letzte Definition impliziert $f(u, u) = 0$ und falls $(u, v) \notin E$ und $(v, u) \notin E \Rightarrow f(u, v) = f(v, u) = 0$.

Beispiel:





Das Problem:

Gegeben sei ein Netzwerk N. Konstruiere eine Flußfunktion f für N mit maximalen Wert $|f|$ (maximaler Fluß).

Die volgenden Lösungsalgorithmen basieren auf 3 Ideen:

- Das residuale Netzwerk
- Der Erweiterungsweg
- Der Kanten-Schnitt des Netzwerks

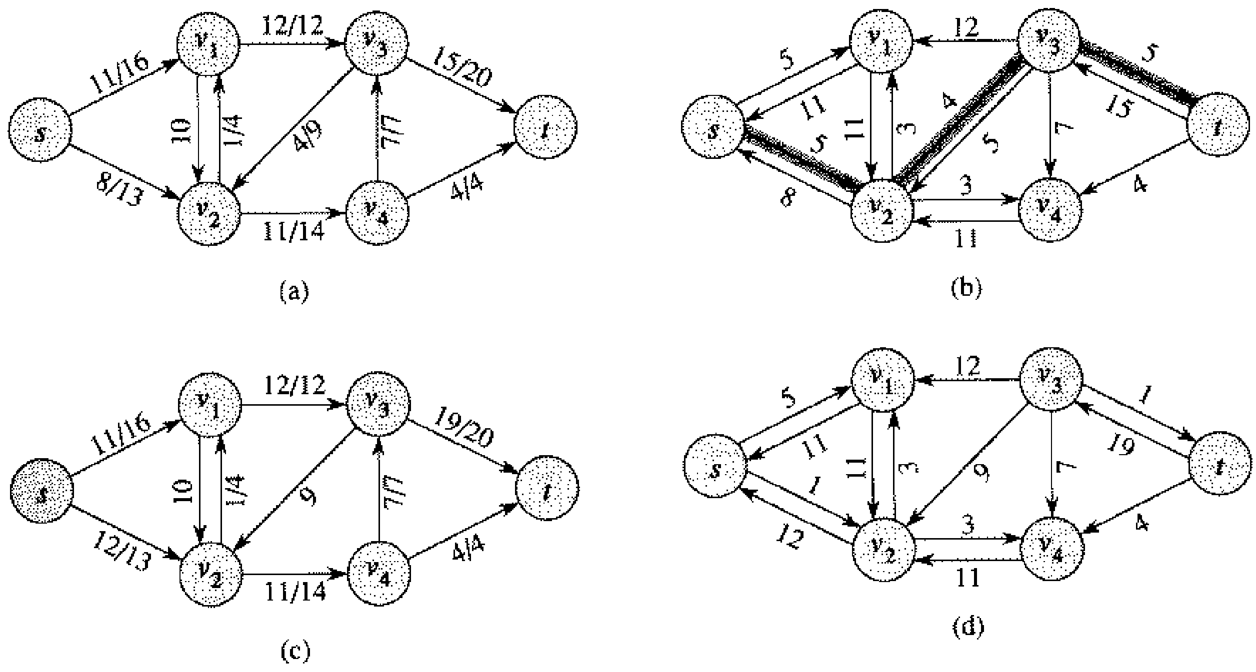
Definition 4.3 *Sei $N = (V, E, s, t)$ ein Netzwerk mit einer Flußfunktion f. Für jede $u, v \in V$ setzen wir*

$$c_f(u, v) = c(u, v) - f(u, v)$$

und sei $N_f = (V, E_f, s, t)$ das residuale Netzwerk, wobei

$$E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$$

und somit $|E_f| \leq 2|E|$.



Lemma 4.1 *Sei $N = (V, E, s, t)$ ein Netzwerk mit einer Flußfunktion f und sei f' eine Flußfunktion für N_f. Dann ist $f + f'$ eine Flußfunktion für N und $|f + f'| = |f| + |f'|$.*

Beweis:
Offensichtlich, gilt: $(f + f')(u, v) = -(f + f')(v, u)$:

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

Da $f(u, v) \leq c(u, v)$ und $f'(u, v) \leq c_f(u, v)$ haben wir:

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v) \\ &\leq f(u, v) + (c(u, v) - f(u, v)) = c(u, v).\end{aligned}$$

Darüberhinaus gilt: $(u \in V - \{s, t\})$

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

Schließlich gilt:

$$\begin{aligned}|f + f'| &= \sum_{v \in V} (f(s, u) + f'(s, u)) \\ &= \sum_{v \in V} f(s, u) + \sum_{v \in V} f'(s, u) = |f| + |f'|. \quad \square\end{aligned}$$

2a. Die Ford-Fulkerson-Methode

Definition 4.4 *Sei* $N = (V, E, s, t)$ *ein Netzwerk mit einer Flußfunktion* f*. Ein* Erweiterungsweg p *ist ein gerichteter Weg von* s *nach* t *in* N_f*.*
Wir seitzen: $c_f(p) := \min\{c_f(u, v) \mid (u, v) \in E(p)\}$.

Lemma 4.2 *Sei* $N = (V, E, s, t)$ *ein Netzwerk mit einer Flußfunktion* f *und sei* p *ein Erweiterungsweg in* N_f*. Wir setzen:*

$$f_p(u, v) = \begin{cases} c_f(p), & \textit{falls } (u, v) \in E(p), \\ -c_f(p), & \textit{falls } (v, u) \in E(p), \\ 0, & \textit{sonst} \end{cases}$$

Dann ist $c_f(p)$ *eine Flußfunktion für* N_f*, und* $|f_p| = c_f(p) > 0$.

Korollar 4.1 *Sei* $N = (V, E, s, t)$ *ein Netzwerk mit einer Flußfunktion* f *und sei* p *ein Erweiterungsweg in* N_f*. Weiterhin sei* $f' = f + f_p$*. Dann ist* f' *eine Flußfunktion mit* $|f'| > |f|$.

Die allgemeine Methode:

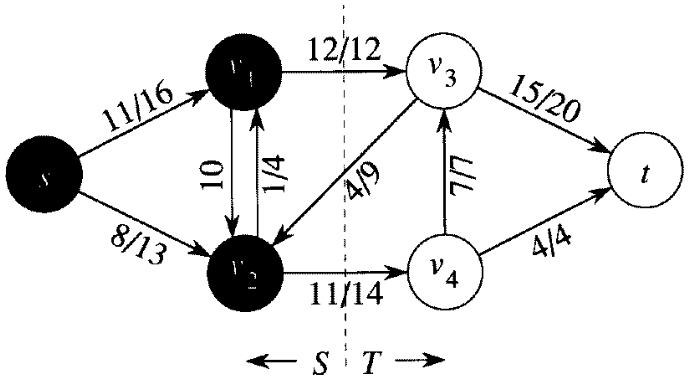
1. Setze $f := 0$.
2. **while** ein Erweiterungsweg p in N_f existiert
   setze $f := f + f_p$.
3. **return** f

2b. Das MAXFLOW-MINCUT-Theorem

Definition 4.5 *Ein* Schnitt (S,T) *in einem Netzwerk* $N = (V, E, s, t)$ *ist eine Zerteilung der Knotenmenge* $V' = S \cup T$*, mit* $s \in S$*,* $t \in T$ *und* $S \cap T = \emptyset$*.*

Wir definieren:

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



Lemma 4.3 *Sei* $N = (V, E, s, t)$ *ein Netzwerk mit einer Flußfunktion* f *und sei* (S,T) *ein Schnitt in* N*. Dann gilt:* $f(S,T) = |f|$*.*

Beweis:

$$\begin{aligned} f(S,T) &= \sum_{u \in S} \sum_{v \in T} f(u,v) = \sum_{u \in S} \sum_{v \in V} f(u,v) - \sum_{u \in S} \sum_{v \in S} f(u,v) \\ &= \sum_{u \in S} \sum_{v \in V} f(u,v) \\ &= \sum_{v \in V} f(s,v) + \sum_{u \in S-s} \sum_{v \in V} f(u,v) \\ &= \sum_{v \in V} f(s,v) = |f|. \quad \square \end{aligned}$$

Theorem 4.1 (MAXFLOW-MINCUT-THEOREM)
Sei $N = (V, E, s, t)$ ein Netzwerk mit einer Flußfunktion f. Dann sind die folgenden Aussagen äquivalent:

1. *f ist ein Fluß mit maximalem Wert $|f|$.*
2. *Das Netzwerk N_f enthält keinen Erweiterungsweg.*
3. *$|f| = c(S, T)$ für einen Schnitt (S, T) in N.*

Beweisskizze.

(1)⇒(2):
Angenommen, daß N_f einen Erweiterungsweg p enthält.
(Korollar 4.1) $\Rightarrow |f + f_p| > |f|$. Widerspruch.

(2)⇒(3):
Wir setzen:

$$S = \{v \in V \mid \exists \text{Weg } s \to v \text{ in } N_f\}, \quad T = V - S.$$

Dann ist (S, T) ein Schnitt und es gilt $f(u, v) = c(u, v)$ für $u \in S$ und $v \in T$.
(Lemma 4.3) $\Rightarrow |f| = f(S, T) = c(S, T)$.

(3)⇒(1): Sei (S, T) ein Schnitt.

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

Da $[|f| = c(S, T)] \Rightarrow |f|$ ist maximal. $\square$

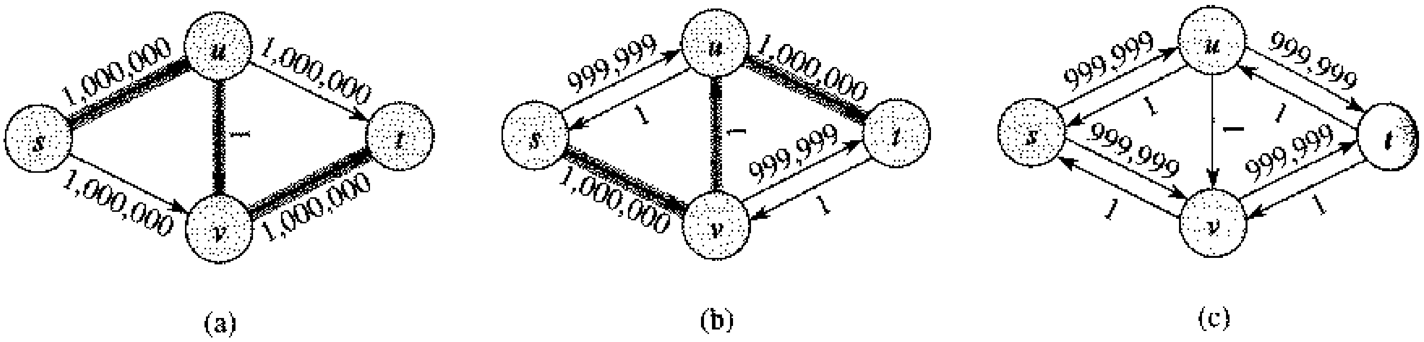
2c. Der Ford-Fulkerson-Algorithmus

Algorithmus 4.1 FORD-FULKERSON(N, s, t);

for all $(u, v) \in E$
  do $f[u, v] := 0$
    $f[v, u] := 0$

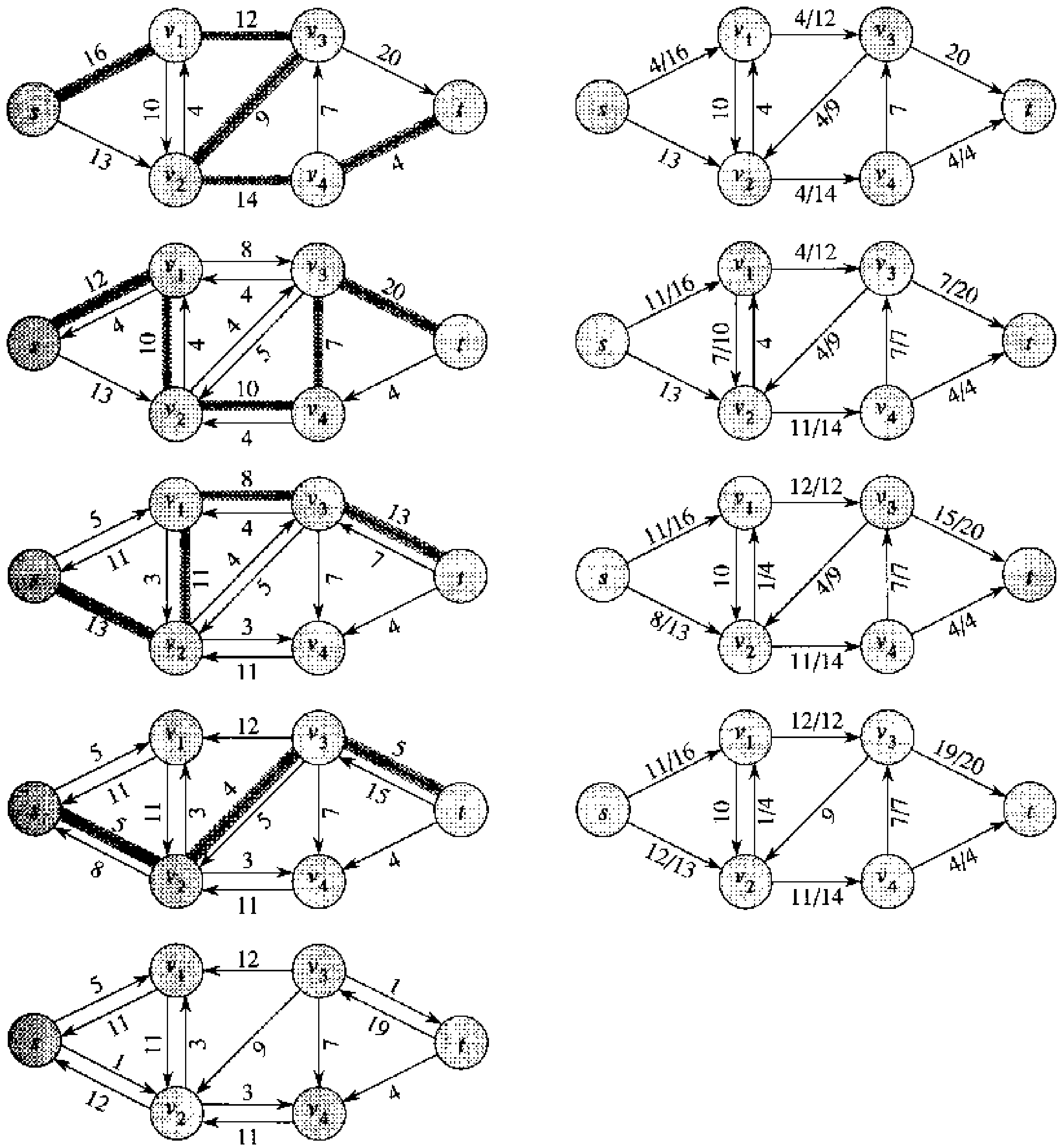
while $\exists$ Erweiterungsweg p in N_f
  do $c_f(p) := \min\{c_f(u, v) \mid (u, v) \in p\}$
    for all $(u, v) \in p$
      do $f[u, v] := f[u, v] + c_f(p)$
        $f[v, u] := -f[u, v]$

Falls alle $c(u, v)$ rational sind, ist die Laufzeit des FORD-FULKERSON-Algorithmus $O(|E| \cdot |f^*|)$, wobei $|f^*|$ der maximale Fluß ist, den der Algorithmus konstruiert.



Falls $c(u, v)$ beliebige rationale Zahlen sind, terminiert der FORD-FULKERSON-Algorithmus nicht zwingend.

Beispiel:


s
t
v_1
v_2
v_3
v_4
4/12
4/16
4/9
4/14
4/4
11/16
7/10
7/20
7/7
11/14
12/12
15/20
8/13
1/4
12/13
19/20

2c. Der Edmonds-Karp-Algorithmus

Seien nun alle $c(u, v)$ rational. Weiterhin sei jeder Erweiterungsweg p im FORD-FULKERSON-Algorithmus ein kürzester Weg zwischen s und t in N_f, der mit Hilfe der Breitensuche konstruiert wird. Dann heißt der so implementierte Algorithmus EDMONDS-KARP-Algorithmus.

Sei $\delta_f(u, v)$ die Länge des kürzesten Weges in G_f zwischen u und v, wobei jede Kante die Länge 1 hat.

Behauptung 4.1 *Nach jedem Durchlauf der* **while** *-Schleife des* EDMONDS-KARP-*Algorithmus vergrößert sich die Distanz* $\delta_f(s, v)$ *monoton für alle* $v \in V - \{s, t\}$.

Beweis: siehe [1], Lemma 27.8 auf Seite 597.

Behauptung 4.2 *Die Anzahl der* **while** *-Schleifen des* EDMONDS-KARP-*Algorithmus ist* $O(|V| \cdot |E|)$.

Beweis: siehe [1], Lemma 27.9 auf Seite 598.

Somit ist die Laufzeit des EDMONDS-KARP-Algorithmus $O(|V| \cdot |E|^2)$.

2d. Der Dinic-Algorithmus

Definition 4.6 *Ein Netzwerk* $N = (V, E, s, t)$ *heißt* Schichtennetzwerk, *wenn es eine Partitionierung* $V = V_1, ..., V_r$ *gibt mit*

- $V_1 = \{s\}$, $V_r = \{t\}$.
- $\forall (u, v) \in E$ *gilt:* $\exists\ i \in \{1, ..., r-1\}$, *so daß* $u \in V_i$, $v \in V_{i+1}$.

Definition 4.7 *Sei* $N = (V, E, s, t)$ *ein Netzwerk mit einer Flußfunktion* f. *Die Kante* $(u, v) \in E$ *heißt* benutzbar, *falls* $f(u, v) < c(u, v)$ *gilt.*

<u>**Algorithmus** 4.2</u> SCHICHTENNETZWERK(N);

$V_1 := s$, $i := 1$
for $k := 2$ **to** $|V|$
  do $V_k := \emptyset$
repeat
  $T := \{v \notin \bigcup_{j \le i} V_j \mid \exists$ benutzbare Kante (u, v) mit $u \in V_i\}$
  if $t \in V$ **then** $V_{i+1} := t$, $i := i + 1$
    else $V_{i+1} := T$, $i := i + 1$
until $T = \emptyset$ oder $t \in V_i$
for $k := 1$ **to** $i - 1$
  do $E_k := \{(u, v) \in E \mid u \in V_k,\ v \in V_{k+1},\ (u, v)$ benutzbar$\}$
$V' = \cup_k V_k$, $E' = \cup_k E_k$
return $N' = (V', E', s, t)$

Sei $N' = (V', E', s, t)$ ein Schichtennetzwerk von N. Wir setzen $c'(u,v) = c(u,v) - f(u,v)$ für alle $(u,v) \in E'$.

Definition 4.8 *Sei $N' = (V', E', s, t)$ ein Schichtennetzwerk mit einer Flußfunktion f'. Dann heißt f'* ausfüllend, *falls es auf jedem Weg von s nach t eine Kante (u,v) gibt mit $f'(u,v) = c'(u,v)$.*

<u>Algorithmus 4.3</u> AUSFÜLLENDER-FLUSS(N');

for all $(u,v) \in E'$
  do $f'(u,v) := 0$
repeat
  Bestimme einen Weg $p = (s = v_0 \to v_1 \to \cdots \to v_k = t)$
  /* Weg existiert nicht $\Rightarrow$ f ist ausfüllend */
  $c_{f'}(p) := \min\limits_{1 \le i \le k} \{c'(v_{i-1}, v_i) - f'(v_{i-1}, v_i)\}$
  for all $(u,v) \in E'$
    do $f'(u,v) := \begin{cases} f'(u,v), & \text{falls } (u,v) \notin p \\ f'(u,v) + c_{f'}(p), & \text{falls } (u,v) \in p \end{cases}$
  Streiche alle Kanten $(u,v) \in E'$ mit $f'(u,v) = c'(u,v)$
until Weg p existiert nicht
return f'

Algorithmus 4.4 $\textsc{Dinic}(N)$;

for all $(u, v) \in E$
  do $f(u, v) := 0$
repeat
  Konstruiere Schichtennetzwerk N' für N und f
  Bestimme ausfüllenden Fluß f' für N'
  for all $(u, v) \in E$

$$\textbf{do } f(u,v) := \begin{cases} f(u,v), & \text{falls } (u,v) \notin E' \\ f(u,v) + f'(u,v), & \text{falls } (u,v) \in E \\ f(u,v) - f'(u,v), & \text{falls } (v,u) \in E \end{cases}$$

until kein Schichtennetzwerk gefunden
return f

Theorem 4.2 *Wenn für N, f kein Schichtennetzwerk mehr gefunden wird, dann ist f maximal.*

Beweisskizze:
Es werde kein Schichtennetzwerk gefunden
$\Rightarrow \exists i \in \{1, ..., |V|\}$ mit: $\nexists$ benutzbare Kante $(u, v) \in E$ mit $u \in V_i$, $v \notin \bigcup\limits_{j \le i} V_j$.
Sei $S = \bigcup\limits_{j \le i} V_j$ und $T = V - S$. Dann gilt: $s \in S$, $t \in T$.

$$\Rightarrow \qquad f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) = \sum_{u \in S} \sum_{v \in T} c(u,v) = c(S,T),$$

somit ist f maximal (Theorem 4.1). $\square$

Die Laufzeit des DINIC-Algorithmus:

- Schichtennetzwerk aufbauen: $O(|E|)$ (Breitensuche).
- Ausfüllender Fluß: $O(E) \cdot$ (DFS/BFS) $= O(|E|^2)$
  Sogar $O(|E| \cdot |V|)$ (s. Übung).
  Schnellste: $O(|V|^2)$ (3 Indians Algorithmus)

Frage: Wieviele Schichtennetzwerke ?

Lemma 4.4 *Sei N_k das Schichtennetzwerk in k-ten Durchlauf des DINIC-Algorithmus. Sei l_k die Anzahl der Schichten im N_k minus 1. Dann gilt: wenn der k-te Durchlauf nicht der letzte ist, so ist $l_{k+1} > l_k$.*

Beweisskizze:
Sei $p = (s = v_0 \to v_1 \to \cdots \to v_{l_{k+1}} = t$ ein Weg in N_{k+1}.
Seien $V_0, ..., V_{l_k}$ die Schichten in N_k.

Wir zeigen per Induktion über i: $v_i \in V_j \Rightarrow i \geq j$.
(I.A.) $i = 0 \Rightarrow q = v_0 \in V_0$.
(I.S.) Annahme: $v_{j+1} \in V_l \Rightarrow l > j + 1$.
$\Rightarrow$ Kante (v_i, v_{i+1}) wurde in N_k nicht benutzt, war also nicht benutzbar.
$\Rightarrow$ Auch zum Aufbau von N_{k+1} war (v_i, v_{i+1}) nicht benutzbar. Widerspruch.

Für $i = l_{k+1}$ gilt: $t = v_{l_{k+1}} \in V_{l_k} \Rightarrow l_{k+1} \geq l_k$.
Annahme: $l_{k+1} = l_k$ gilt.
$\Rightarrow v_i \in V_i\ \forall i$ und p wäre auch ein Weg in N_k gewesen. Widerspruch (da der Fluß für N_k ausfüllend sein sollte). $\square$

$\Rightarrow$ Gesamtlaufzeit des DINIC-Algorithmus: $O(|V|^2 \cdot |E|)$.

3a. Zusammenhang von Graphen

Definition 4.9 *Ein ungerichteter Graph $G = (V, E)$ heißt κ-fach zusammenhängend, falls es zwischen jedem Knotenpaar $u, v \in V$ (mit $u \neq v$) κ knotendisjunkte Wege gibt.*

Definition 4.10 *Sei $G = (V, E)$ ein ungerichteter Graph und $a, b \in V$. Eine Menge $S \subseteq V$ heißt (a, b)-Knoten-Separator, falls $\{a, b\} \subset V - S$ und jeder Weg von a nach b in G wenigstens ein Knoten aus S enthält.*

$N(a, b) :=$ minimale Kardinalität eines (a, b)-Knoten-Separators.
$p(a, b) :=$ maximale Anzahl der knotendisjunkten Wege von a nach b.

Theorem 4.3 (Menger-Satz)
Falls $(a, b) \notin E$, dann gilt $N(a, b) = p(a, b)$.

Beweisskizze: (über Flüsse)
Sei $G = (V, E)$ gegeben.
Konstruiere das folgende Netzwerk $N_G = (V', E', s, t)$:

- $\forall v \in V$: Füge zwei Knoten v', v'' ein in V' und verbinde diese durch die Kante (v', v'') (interne Kanten). Setze: $s = a''$ und $t = b'$.
- $\forall (u, v) \in E$: Füge (u'', v') und (v'', u') in N_G ein. (externe Kanten)

Wir setzen:

$$c(u, v) = \begin{cases} 1, & \text{für alle interne Kanten} \\ \infty, & \text{für alle externe Kanten} \end{cases}.$$

Wir zeigen: Für den maximalen Fluß f von s nach t gilt:

$$|f| = p(a,b).$$

Es gibt $p(a,b)$ knotendisjunkte Wege in G von a nach b:

$$a \to v_1 \to \cdots \to v_l = b.$$

$\Rightarrow$ in N_G gibt es $p(a,b)$ k.-d. Wege von s nach t:

$$s = a'' \to v_1' \to v_1'' \to \cdots \to v_l' \to v_l'' = t$$

$\Rightarrow |f| \geq p(a,b)$.

Angenommen, f ist maximale Flußfunktion für N_G mit $f(e) \in \{0,1\}$ $\forall e \in E'$.
$\Rightarrow$ auf jedem Weg erreicht f den Fluß 1
$\Rightarrow$ es gibt wenigstens $|f|$ Wege, d.h. $p(a,b) \geq |f|$. $\square$

Menger-Theorem führt zu folgende Methode zur Berechnung von $N(a,b)$:

1. Konstruiere das Netzwerk N_G wie in Theorem 4.3.
2. Finde maximalen Fluß f für N_G von s nach t.
3. Bilde minimalen Kanten-Schnitt C in N_G.
4. Bilde (a,b)-Knoten-Separator in Abhängigkeit von C, indem wir alle entsprechenden Knoten in G wählen

<u>κ-fach Zusammenhang:</u>
Für alle Knotenpaare (a,b) mit $(a,b) \notin E$ berechne den minimalen Knoten-Separator und bestimme κ.

Kanten-Zusammenhang von Graphen

Definition 4.11 *Ein ungerichteter Graph $G = (V, E)$ heißt κ-fach kanten-zusammenhängend, falls es zwischen jedem Knotenpaar $u, v \in V$ κ kantendisjunkte Wege gibt.*

Definition 4.12 *Sei $G = (V, E)$ ein ungerichteter Graph und $a, b \in V$. Eine Menge $S \subseteq E$ heißt (a, b)-Kanten-Separator, falls jeder Weg von a nach b in G wenigstens eine Kante aus S enthält.*

$K(a, b) :=$ minimale Kardinalität eines (a, b)-Kanten-Separators.
$q(a, b) :=$ maximale Anzahl der kanten-disjunkten Wege von a nach b.

Theorem 4.4 *Es gilt: $K(a, b) = q(a, b)$.*

Beweisskizze:
Sei $G = (V, E)$ gegeben. Konstruiere das Netzwerk $N_G = (V, E', a, b)$, wobei es zu jeder Kante $(u, v) \in E$ die gerichteten Kanten (a, b) und (b, a) geben. Setzen wir die Kapazität aller Kanten auf 1.

Sei f maximaler Fluß in N_G. Dann gilt: $|f| = q(a, b)$. $\square$

Theorem 4.4 führt zur Berechnung von $K(a, b)$ durch die Berechnung von $q(a, b)$. Wir wenden diese Methode für alle $a, b \in V$ an und berechnen somit den κ-fach Kantenzusammenhang.

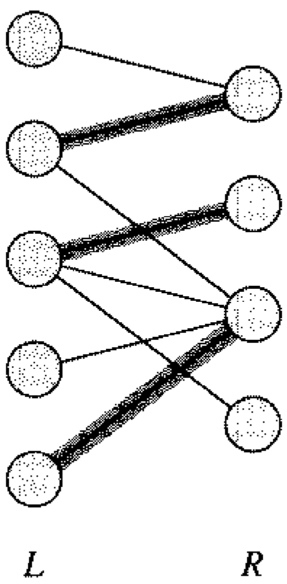
3b. Matching in bipartiten Graphen

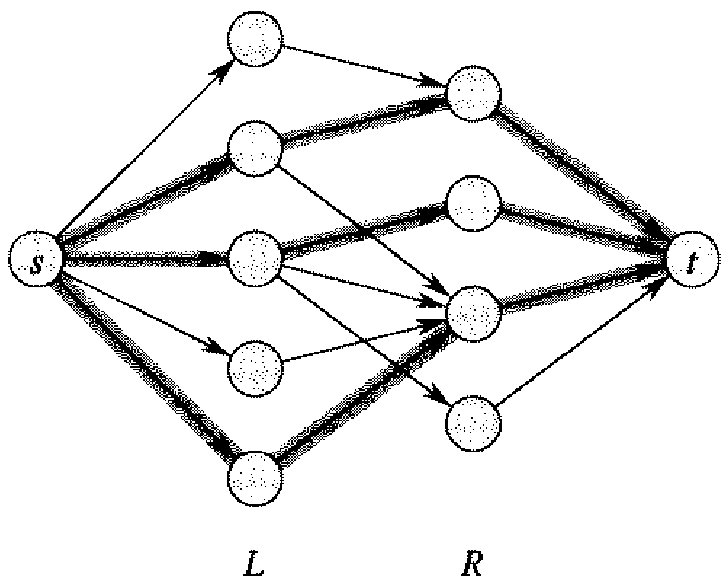
Definition 4.13 *Sei $G = (V, E)$ ein ungerichteter Graph. Eine Teilmenge $M \subseteq E$ heißt* Matching, *falls die Kanten in M keine gemeinsamen Knoten haben. Ein Matching heißt maximal, falls es kein Matching mit mehr Kanten gibt.*

Das Problem:
Gegeben sei ein bipartiter Graph $G = (V_1 \cup V_2, E)$. Konstruiere ein maximales Matching.

Sei $N_G = (V', E', s, t)$ ein Netzwerk mit:

$$
\begin{aligned}
V' &= V_1 \cup V_2 \cup \{s, t\} \\
E' &= \{(s, u) \mid u \in V_1\} \\
&\quad \cup \{(u, v) \mid u \in V_1,\ v \in V_2,\ (u, v) \in E\} \\
&\quad \cup \{(v, t) \mid v \in V_2\}.
\end{aligned}
$$





Theorem 4.5 *Sei $G = (V_1 \cup V_2, E)$ ein bipartiter Graph. Wenn M ein Matching ist, dann gibt es ein Fluß f in N_G mit $|f| = |M|$. Wenn f ein ganzzahliger Fluß für N_G ist, dann gibt es eine Matching M in G mit $|M| = |f|$.*

Beweisskizze:
Sei M ein Matching in G. Wir definieren einen Fluß:

$$\begin{aligned} f(s,u) &= f(u,v) = f(v,t) = 1 \\ f(u,s) &= f(v,u) = f(t,v) = -1 \end{aligned}$$

für $(u,v) \in M$ und $f(u,v) = 0$ sonst.
Für jedes $(u,v) \in M$ es gibt einen Weg $s \to u \to v \to t$.
Sei $S = \{s\} \cup V_1$ und $T = \{t\} \cup V_2$.
Dann ist (S,T) ein Schnitt in N_G und (Lemma 4.3) $|f| = |M|$.

Sei f ein ganzzahliger Fluß in N_G. Wir betrachten

$$M = \{(u,v) \mid u \in V_1,\ v \in V_2,\ f(u,v) > 0\}.$$

Dann ist M ein Matching, für das $|M| = |f|$ gilt. $\square$

Korollar 4.2 *Die Kardinalität des maximalen Matchings in G ist gleich dem Wert des maximalen Flusses in N_G.*

Beweisskizze:
Sei M ein maximales Matching. Wir konstruieren den Fluß f wie im Beweis des Theorems 4.5.
Angenommen f ist nicht maximal. $\Rightarrow \exists$ ein ganzzahliger Fluß f' für N_G mit $|f'| > |f|$.
(Theorem 4.5) $\Rightarrow$ es gibt ein Matching M' in G mit

$$|M'| = |f'| > |f| = |M|.$$

Dann ist M kein maximales Matching. Widerspruch. $\square$

Sei $G = (V, E)$ ein Graph und $A \subseteq V$. Wir definieren:

$$N(A) = \{v \in V - A \mid (v, w) \in E,\ w \in A\}.$$

Theorem 4.6 (P. Hall)
Sei $G = (V_1 \cup V_2, E)$ ein bipartiter Graph. G hat ein Matching M mit $|M| = |V_1| \iff$ für jede Teilmenge $A \subseteq V_1$ gilt:

$$|N(A)| \geq |A|.$$

Beweisskizze:
"$\Longrightarrow$" Offensichtlich.

"$\Longleftarrow$" Sei $|N(A)| \geq |A|$ für jede Teilmenge $A \subseteq V_1$. Sei f ein maximaler ganzzahliger Fluß in N_G und $S \subseteq V'$ die Knoten, die im residualen Netzwerk von s aus erreichbar sind. Dann gilt: $|f| = c(S, T)$, wobei $T = V(N_G) - S$.

Sei $v \in S \cap V_1$ und $(v, w) \in E(N_G)$. Wir zeigen $w \in S$.
Angenommen $w \notin S \Rightarrow f(v, w) = 1$ (sonst wäre $w \in S$).
$f(s, v) = 1 \Rightarrow \sum_{u \in V(N_G)} f(v, u) \neq 0$. Widerspruch.

Somit gilt: $N(S \cap V_1) \subseteq S$. Deswegen gilt:
$(v, w) \in E(V_G)$, $v \in S$, $w \in T \Rightarrow v = s$ oder $w = t$.
Da $S \cap V_2 = N(S \cap V_1)$, bekommen wir:

$$|f| = |V_1 - S| + |N(S \cap V_1)| \geq |V_1 - S| + |S \cap V_1| = |V_1| \geq |f|.$$

(Theorem 4.5) $\Rightarrow G$ enthält ein Matching mit $|V_1|$ Kanten. $\square$

Korollar 4.3 *Ein regulärer bipartiter Graph enthält ein vollständiges Matching.*

5 Lektion 5: Algebraische Algorithmen

1. Schnelle Matrizenmultiplikation
2. Inverse der Matrizen
3. Polynome und FFT

1. Matrizenmultiplikation

Seien A, B zwei $(n \times n)$-Matrizen. Dann ist

$$C = A \cdot B \qquad \text{mit} \qquad c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{k_j}.$$

Schulmethode:

iterativ: Berechne c_{ij} nach obiger Formel.
$\Longrightarrow$ Aufwand: n Multiplikationen, $n-1$ Additionen d.h. $2n-1$ arithmetische Operationen für jedes Paar i, j.
$\Longrightarrow n^2(2n-1) = O(n^3)$ Operationen.

rekursiv: (Divide and Conquer)
Zerlege A, B in 4 $(\frac{n}{2} \times \frac{n}{2})$-Matrizen. (O.B.d.A. sei $n = 2^k$).

$$A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = C$$

wobei:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22}. \end{aligned}$$

$\Longrightarrow$ Aufwand:

$$\begin{aligned} T(n) &= 8 \cdot T(\frac{n}{2}) + 4\left(\frac{n}{2}\right)^2 \\ &= O\left(n^{\log_2 8}\right) = O(n^3). \end{aligned}$$

Strassen-Algorithmus.

Idee: Benutze Divide and Conquer Schema zur Berechnung der C_{ij}, $i, j \in \{1, 2\}$, allerdings mit weniger ($m \leq 7$) Multiplikationen und mehr ($a \geq 4$) Additionen:

$$T(n) = m \cdot T\left(\frac{n}{2}\right) + a\left(\frac{n}{2}\right)^2 = O\left(n^{\log_2 m}\right),$$

Theorem 5.1 *Zwei $(n \times n)$-Matrizen lassen sich mit 7 Multiplikationen und 18 Additionen (Matrixoperationen) multiplizieren.*

Korollar 5.1 *Die Laufzeit des Strassen-Algorithmus ist $O\left(n^{\log_2 7}\right) \approx O(n^{2.81})$.*

Bemerkung 5.1 *Es existiert bessere Algorithmen mit der Laufzeit $O(n^{2.38})$. Die untere Schranke ist $\Omega(n^2)$, da man n^2 Elemente berechnen muss.*

Beweisskizze zum Theorem 5.1:

Sei $n = 2^k$. Berechne:

$$\begin{aligned}
M_1 &= (A_{12} - A_{22})(B_{21} + B_{22})\\
M_2 &= (A_{11} + A_{22})(B_{11} + B_{22})\\
M_3 &= (A_{11} - A_{21})(B_{11} + B_{12})\\
M_4 &= (A_{11} + A_{12})B_{22}\\
M_5 &= A_{11}(B_{12} - B_{22})\\
M_6 &= A_{22}(B_{21} - B_{11})\\
M_7 &= (A_{21} + A_{22})B_{11}.
\end{aligned}$$

Somit gilt:

$$\begin{aligned}
C_{11} &= M_1 + M_2 - M_4 + M_6\\
C_{12} &= M_4 + M_5\\
C_{21} &= M_6 + M_7\\
C_{22} &= M_2 - M_3 + M_5 - M_7.
\end{aligned}$$

Falls $2^{k-1} < n < 2^k$ gilt, fülle die Matrizen A, B zu $(2^k \times 2^k)$-Matrizen auf. Damit ergibt sich der Aufwand $d \cdot n^{\log_2 7}$, für konstantes $d > 0$. $\square$

Bemerkung 5.2 *Die Konstante d ist ziemlich groß. Die Praxis zeigt: bis $n <\approx 1000$ ist die Schulmethode besser.*

Bemerkung 5.3 *Die exakte Laufzeit, die für die Matrizenmultiplikation nötig ist, ist unbekannt.*

Boolesche Matrizenmultiplikation

Definition 5.1 *Das System* $(S, \oplus, \odot, 0, 1)$, *mit einer endliche Menge* S, $\oplus, \odot$ *zwei binäre Operationen und* $0, 1 \in S$ *zwei bezeichnete Elementen, heißt* Quasiring, *falls:*

1. $(S, \oplus, 0)$ *ist ein Monoid, d.h.:*
   - $a \oplus b \in S$, *für alle* $a, b \in S$.
   - $a \oplus (b \oplus c) = (a \oplus b) \oplus c$, *für alle* $a, b, c \in S$.
   - $a \oplus 0 = 0 \oplus a = a$, *für alle* $a \in S$.
2. $a \odot 0 = 0 \odot a = 0$, *für alle* $a \in S$.
3. $a \oplus b = b \oplus a$, *für alle* $a, b \in S$.
4. $a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c)$ *und* $(b \oplus c) \odot a = (b \odot a) \oplus (c \odot a)$, *für alle* $a, b, c \in S$.

Falls zusätzlich gilt:

5. $\forall a \in S \quad \exists b \in S$, *so daß* $a \oplus b = b \oplus a = 0$,

heißt das System $(S, \oplus, \odot, 0, 1)$ *ein Ring.*

Beispiel: (Boolesche Quasiring:)
Wir definieren das System $(\{0, 1\}, \oplus, \odot)$:

0	1	$\oplus$	$\odot$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Bemerkung 5.4 *Strassen-Algorithmus funktioniert korrekt auf jedem Ring von Matrixelementen.*

Beweisskizze: Die Korrektheit des Strassen-Algorithmus ist nur von seiner Korrektheit auf (2×2)-Matrizen abhängig. Da die Matrixelemente einen Ring formen, ist die Menge der (2×2)-Matrizen auch ein Ring. Per Induktion läßt sich zeigen, daß der Strassen-Algorithmus korrekt funktioniert. $\square$

Wir betrachten jetzt die boolsche Multiplikation von Matrizen. Wir definieren $C = AB$ mit $c_{ij} = \oplus_{k=1}^{n}(a_{ik} \odot b_{kj})$. Seien $B(n)$ $(M(n))$ die Anzahl der Operationen zur Berechnung der boolschen (üblichen) Matrizenmultiplikation für $(n \times n)$-Matrizen.

Theorem 5.2 *Es gilt:* $B(n) = O(M(n))$.

Beweisskizze:
Wir berechnen $c'_{ij} = \Sigma_{k=1}^{n} a_{ik} b_{kj}$.

Dann gilt:

$$\begin{aligned} a_{ik}b_{kj} = 0 \quad &\Longleftrightarrow \quad a_{ik} \odot b_{kj} = 0 \\ a_{ik}b_{kj} = 1 \quad &\Longleftrightarrow \quad a_{ik} \odot b_{kj} = 1. \end{aligned}$$

Somit gilt:

$$c'_{ij} = 0 \quad \Longleftrightarrow \quad c_{ij} = 0,$$

und die Matrix C läßt sich mit $\Theta(n^2)$ Operationen aus der Matrix C' konstruieren.

Die Anzahl der arithmetischen Operationen ist:

$$O(M(n)) + \Theta(n^2) = O(M(n)). \qquad \square$$

2. Berechnung der Inversen der Matrizen

Sei A eine $(n \times n)$-Matrix. Unser Ziel ist, die Matrix A^{-1} zu berechnen, so daß $A \cdot A^{-1} = A^{-1} \cdot A = I_n$ gilt.

Falls die Matrix A^{-1} existiert, nennen wir die Matrix A nicht singulär. In diesen Fall ist A^{-1} eindeutig definiert.

Sei $M(n)$ die Zeit für die Matrizenmultiplikation und sei $I(n)$ die Zeit für Berechnung der Inversion.

Theorem 5.3 *Falls $I(3n) = O(I(n))$ ist, dann gilt:*

$$M(n) = O(I(n)).$$

Beweisskizze: Seien A, B die Matrizen, die wir multiplizieren müssen. Wir konstruieren:

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}.$$

Dann gilt:

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix}.$$

Da die Matrix D in $\Theta(n^2)$ Zeit konstruierbar ist, $I(n) = \Omega(n^2)$ und die Matrix D^{-1} in Zeit $O(I(3n)) = O(I(n))$ berechenbar ist, gilt das Theorem. $\square$

Bemerkung 5.5 *Falls $I(n) = \Theta\left(n^c \log^d n\right)$ mit Konstanten $c > 0$ und $d \geq 0$ ist, dann gilt $I(3n) = O(I(n))$.*

Theorem 5.4 *Falls $M(n) = O(M(n+k))$ für $0 \leq k \leq n$, dann gilt:*

$$I(n) = O(M(n)).$$

Beweisskizze: O.B.d.A. nehmen wir an, daß $n = 2^p$ gilt. Falls $2^{p-1} < n < 2^p$ gilt, haben wir für $k = 2^p - n$:

$$\begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I_k \end{pmatrix}.$$

Somit vergrößert sich die Laufzeit höhstens um einen konstanten Faktor.

Eine $(n \times n)$-Matrix A heißt positive-definite, falls für alle n-dimensionalen Vektoren $x \neq 0$ gilt: $x^T A x > 0$.

Sei A symmetrisch und positiv-definite. Zerlege A in 4 $(n/2 \times n/2)$-Martizen:

$$A = \begin{pmatrix} B & C^T \\ C & B \end{pmatrix}$$

und setze

$$S = D - CB^{-1}C^T.$$

Die Matrizen B und S sind nicht singulär (siehe [1], Lemmas 31.13 – 31.15 auf die Seiten 766–767).

Somit haben wir:

$$A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}C^T S^{-1} C B^{-1} & -B^{-1} C^T S^{-1} \\ -S^{-1} C B^{-1} & S^{-1} \end{pmatrix}.$$

Um A^{-1} zu berechnen, muß diese Methode die Matrix $E = C \cdot B^{-1}$ berechnen und auch die folgende 3 Matrizenmultiplikationen durchführen:

$$E \cdot C^T \qquad S^{-1} \cdot E \qquad E^T \cdot (S^{-1} E).$$

Damit erhalten wir:

$$\begin{aligned} I(n) &\leq 2I(n/2) + 4M(n) + O(n^2) \\ &= 2I(n/2) + O(M(n)) = O(M(n)). \end{aligned}$$

Falls A nicht symmetrisch ist, betrachten wir die Matrix $A^T A$. Für jede nicht singuläre Matrix A ist die Matrix $A^T A$ symmetrisch und positive-definite (siehe [1]. Theorem 31.6).

Da $((A^T A)^{-1} A^T) A = (A^T A)^{-1} \cdot (A^T A) = I_n$ ist, erhalten wir:

$$A^{-1} = (A^T A)^{-1} \cdot A^T.$$

Um A^{-1} zu berechnen, konstruieren wir erst die Matrix $A^T A$ und berechnen $(A^T A)^{-1}$ mit der obige Methode.

Damit ist A^{-1} in Zeit $O(M(n))$ berechenbar. $\square$

Bemerkung 5.6 *Das letzte Theorem zeigt, daß das System der linearen Gleichungen $Ax = b$ mit einer nicht singulären Martix A sich in Zeit $O(M(n))$ lösen läßt. Man konstruiert die Matrix A^{-1}, und danach: $A^{-1} A x = x = A^{-1} b$.*

3. Polynome und FFT

Sei $A(x) = \sum\limits_{j=0}^{n-1} a_j x^j$ ein Polynom, wobei $a_j \in C$ (komplexe Zahlen), $j = 0, \ldots, n-1$.

Für $B(x) = \sum\limits_{j=0}^{n-1} b_j x^j$ definieren wir

$$\begin{aligned} C(x) &= A(x) + B(x) = \sum_{j=0}^{n-1} c_j x^j \\ D(x) &= A(x) \cdot B(x) = \sum_{j=0}^{2n-1} d_j x^j \end{aligned}$$

wobei $c_j = a_j + b_j$ und $d_j = \sum\limits_{k=0}^{j} a_k b_{j-k}$, $j = 0, \ldots, n-1$.

Das Polynom $C(x)$ läßt sich in Zeit $\Theta(n)$ berechnen, allerdings brauchen wir mit der obigen Formel $\Sigma_{j=0}^{n-1} j = O(n^2)$ Schritte zur Bestimmung von $D(x)$.

<u>Point-value Darstellung von $A(x)$</u>:

$$\{(x_0, y_0),\ (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}, \quad y_k = A(x_k)$$

(wir nehmen an, daß alle x_k verschieden sind).

Da

$$A(x_k) = a_0 + x_k(a_1 + x_k(a_2 + \cdots + x_k(a_{n-2} + x_k a_{n-1})) \cdots),$$

können wir die point-value Darstellung in Zeit $O(n^2)$ bestimmen.

Behauptung 5.1 *Für jede Menge*

$$\{(x_0, y_0),\ (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

gibt es nur ein Polynom $A(x)$ mit Grad $\leq n$, so daß $y_k = A(x_k)$, $k = 0, \ldots, n$.

Beweisskizze:
Wir zeigen daß das System von linearen Gleichungen

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

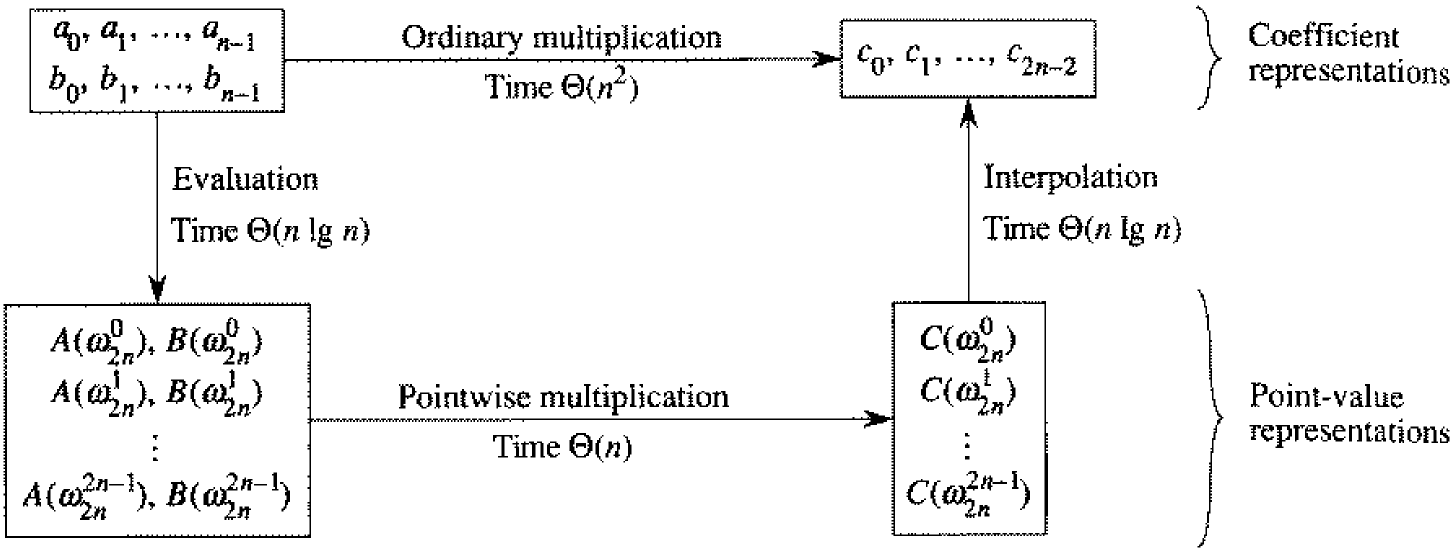
die einzige Lösung $(a_0, a_1, \ldots, a_n)$ hat.

Die Determinante des Systems ist die Vandermonde Determinante und ist gleich $\prod_{j<k}(x_k - x_j) \neq 0$ für verschiedene x_k, x_j.
$\square$

Die point-value Darstellung erlaubt die Summe und die Multiplikation schneller zu berechnen. Da $C(x) = A(x) + B(x)$ und $D(x) = A(x) \cdot B(x)$ für jedes x, können wir $C(x)$ und $D(x)$ in $O(n)$ Schritten bestimmen.

Allerdings müssen wir zur Berechnung von $D(x)$ die Werte von $A(x)$ und $B(x)$ in $2n$ Punkten wissen (also nicht nur in n Punkten).

Idee: Benutze die point-value Darstellung zur Bestimmung von $D(x)$, falls $A(x)$ und $B(x)$ in Koeffizienten-Darstellung gegeben sind.

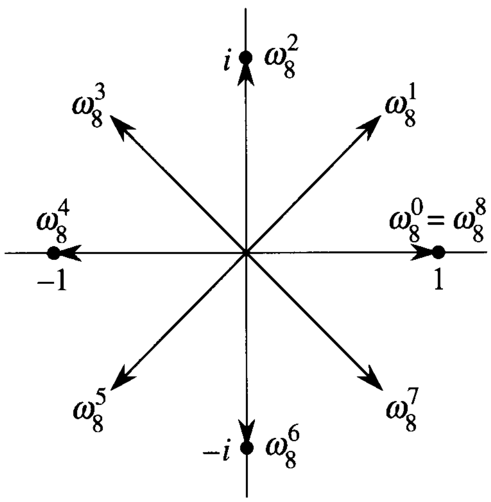


Sei $i = \sqrt{-1}$ und $e^{iu} = \cos(u) + i\sin(u)$.

Wir berechnen die Polynome $A(x)$ und $B(x)$ für (komplexe) Zahlen ω_{2n}^k, wobei $\omega_{2n} = e^{2\pi i/2n}$.

Die Zahlen $\{\omega_{2n}^k \mid k = 0, \ldots, 2n-1\}$ erfüllen die Gleichung $\omega^{2n} = 1$ und bilden eine multiplikative Gruppe:

$\omega_{2n}^{2n} = \omega_{2n}^0 = 1 \Rightarrow \omega_{2n}^j \omega_{2n}^k = \omega_{2n}^{(j+k) \text{ mod } 2n}$ und $\omega_{2n}^{-1} = \omega_{2n}^{2n-1}$.



Lemma 5.1 *Für alle $m \geq 0$, $k \geq 0$ und $d > 0$ gilt:*
$\omega_{dm}^{dk} = \omega_m^k$.

Beweis:
$\omega_{dm}^{dk} = \left(e^{2\pi i/dm}\right)^{dk} = \left(e^{2\pi i/m}\right)^k = \omega_n^k. \quad \square$

Insbesondere gilt: $\omega_{2n}^n = \omega_2 = -1$.

Lemma 5.2 *Es gilt:*

$$\{\omega^2 \mid \omega^{2n} = 1\} = \{\omega \mid \omega^n = 1\}.$$

Beweis:
Da $(\omega_{2n}^k)^2 = \omega_n^k$ nach Lemma 5.1, haben wir:

$$(\omega_{2n}^{k+n})^2 = \omega_{2n}^{2k+2n} = \omega_{2n}^{2k}\omega_{2n}^{2n} = \omega_{2n}^{2k} = (\omega_{2n}^k)^2. \qquad \square$$

Seien $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{2n-1}x^{2n-1}$ und

$$y_k = A(\omega_{2n}^k) = \sum_{j=0}^{2n-1} a_j \cdot \omega_{2n}^{kj}, \qquad k = 0, \ldots, 2n-1.$$

Der Vektor $\vec{y} = (y_0, y_1, \ldots, y_{2n-1})$ heißt Diskrete Fourier Transformation (DFT) vom Vektor $\vec{a} = (a_0, a_1, \ldots, a_{2n-1})$.

Wir berechnen den Vektor $\vec{y}$ mit der Hilfe von Fast Fourier Transformation (FFT). Dafür bezeichnen wir:

$$\begin{aligned} A_0(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{2n-2}x^{n-1} \\ A_1(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{2n-1}x^{n-1}. \end{aligned}$$

Somit haben wir:

$$A(x) = A_0(x^2) + x \cdot A_1(x^2).$$

Wir nehmen oBdA an, daß n eine Potenz von 2 ist.

Algorithmus 5.1 FFT($\vec{a}$);

1. $m :=$length(a) $//m$ ist eine Potenz von 2 $//$
2. **if** $m = 1$ **then** **return** a
3. $\omega_m := e^{2\pi i/m}$
4. $\omega := 1$
5. $a^0 := (a_0, a_2, \ldots, a_{m-2})$
6. $a^1 := (a_1, a_3, \ldots, a_{m-1})$
7. $\vec{y}^0 := FFT(\vec{a}^0)$
8. $\vec{y}^1 := FFT(\vec{a}^1)$
9. **for** $k = 0$ **to** $m/2 - 1$
10. **do** $y_k := y_k^0 + \omega \cdot y_k^1$
11. $y_{k+m/2} := y_k^0 - \omega \cdot y_k^1$
12. $\omega := \omega \cdot \omega_m$
13. **return** y

Zeile 2: Basis der Rekusrion. $y_0 = a_0\omega_1^0 = a_0$.

Zeilen 7-8: nach Lemma 5.2 gilt

$$\begin{aligned} y_k^0 &= A_0(\omega_{m/2}^k) = A^0(\omega_m^{2k}) \\ y_k^1 &= A_1(\omega_{m/2}^k) = A^1(\omega_m^{2k}) \end{aligned}$$

Zeile 10: für $k = 0, \ldots, m/2 - 1$ gilt

$$\begin{aligned} y_k &= y_k^0 + \omega_m^k \cdot y_k^1 \\ &= A_0(\omega_m^{2k}) + \omega_m^k \cdot A_1(\omega_m^{2k}) \\ &= A(\omega_m^k). \end{aligned}$$

Zeile 11: für $k = 0, \ldots, m/2 - 1$ gilt

$$\begin{aligned}
y_{k+m/2} &= y_k^0 - \omega_m^k \cdot y_k^1 \\
&= y_k^0 + \omega_m^{k+m/2} \cdot y_k^1 \qquad (\omega_m^{m/2} = \omega_{2m}^m = \omega_2 = -1) \\
&= A_0(\omega_m^{2k}) + \omega_m^{k+m/2} \cdot A_1(\omega_m^{2k}) \\
&= A_0(\omega_m^{2k+m}) + \omega_m^{k+m/2} \cdot A_1(\omega_m^{2k+m}) \\
&= A(\omega_m^{k+m}).
\end{aligned}$$

Für die Laufzeit des FFT-Algorithmus gilt:

$$T(m) = 2 \cdot T(m/2) + \Theta(m) = \Theta(m \log_2 m).$$

Das letzte Problem ist die point-value Darstellung der Polynome in Koeffizienten-Darstellung schnell zu transformieren. Für die Punkte ω_{2n}^k, $k = 0, \ldots, 2n-1$ haben wir die Gleichungen:

$$\begin{pmatrix}
1 & 1 & 1 & 1 & \cdots & 1 \\
1 & \omega_{2n} & \omega_{2n}^2 & \omega_{2n}^3 & \cdots & \omega_{2n}^{2n-1} \\
1 & \omega_{2n}^2 & \omega_{2n}^4 & \omega_{2n}^6 & \cdots & \omega_{2n}^{2(2n-1)} \\
1 & \omega_{2n}^3 & \omega_{2n}^6 & \omega_{2n}^9 & \cdots & \omega_{2n}^{3(2n-1)} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega_{2n}^{2n-1} & \omega_{2n}^{2(2n-1)} & \omega_{2n}^{3(2n-1)} & \cdots & \omega_{2n}^{(2n-1)(2n-1)}
\end{pmatrix}
\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{2n-1} \end{pmatrix}$$

$$= (y_0, y_1, y_2, y_3, \ldots, y_{2n-1})^T = \vec{y}^T.$$

In anderen Worten, $\vec{y} = V_{2n}\vec{a}$.

Wir bestimmen jetzt die Matrix V_{2n}^{-1}.

Theorem 5.5 *Für $V_{2n}^{-1} = \{v_{jk}\}$ gilt: $v_{jk} = \omega_{2n}^{-kj}/2n$.*

Beweis:
Wir zeigen, daß für die Matrix $W = \{v_{jk}\}$ gilt: $W \cdot V_{2n} = I_{2n}$.
Dafür betrachten wir das Element (j, j') von $W \cdot V_{2n}$:

$$\begin{aligned}
[W \cdot V_{2n}]_{jj'} &= \sum_{k=0}^{2n-1} (\omega_{2n}^{-kj}/n)(\omega_{2n}^{kj'}) \\
&= \sum_{k=0}^{2n-1} \omega_{2n}^{k(j-j')}/n.
\end{aligned}$$

Falls $j = j'$, dann gilt offensichtlich: $[W \cdot V_{2n}]_{jj'} = 1$.

Für $s = j - j' \neq 0$ haben wir:

$$\begin{aligned}
n \cdot [W \cdot V_{2n}]_{jj'} = \sum_{k=0}^{2n-1} \omega_{2n}^{ks} &= \frac{(\omega_{2n}^{s})^{2n} - 1}{\omega_{2n}^{s} - 1} \\
&= \frac{(\omega_{2n}^{2n})^{s} - 1}{\omega_{2n}^{s} - 1} \\
&= \frac{(1)^{s} - 1}{\omega_{2n}^{s} - 1} = 0
\end{aligned}$$

da $-(2n-1) < s < 2n-1$ und $s \neq 0 \Rightarrow \omega_{2n}^{s} \neq 1$. $\square$

Somit für $\vec{a} = (a_0, \ldots, a_{2n-1})$ haben wir:

$$a_j = \sum_{k=0}^{2n-1} \left(\frac{1}{2n} \cdot y_k\right) \cdot \omega_{2n}^{-kj}, \qquad j = 0, \ldots, 2n-1.$$

Um $\vec{a}$ zu berechnen, wenden wir die DFT und den FFT-Algorithmus an (mit $(1/2n) \cdot \vec{y}$ statt $\vec{a}$ und mit ω_{2n}^{-1} statt ω_{2n}).

Somit ist die gesamte Laufzeit $O(n \log n)$.

6 Lektion 6: Geometrische Algorithmen

1. Definitionen
2. Line-Segment Schnitte
3. Innere Punkte des Polygons
4. Einfache Polygone
5. Konvexe Hülle
6. Bestimmung der nähsten Nachbarn
7. Durchmesser einer Punktmenge

1. Objekte

- 2-dimensionale Fläche.
- Koordinatessystem $X - 0 - Y$.
- Punkte $\{p_i\}$, wobei $p_i = (x_i, y_i)$.
- Segmente $p_i - p_j$.
- Linien $-p_i - p_j-$.

Falls $p_1 = (0,0)$ gilt, betrachten wir das Segment p_1, p_2 als den Vektor p_2,

Wir sagen, daß zwei Segmenten $p_1 - p_2$ und $q_1 - q_2$ sich schneiden, falls sie mindestens einen gemeinsamen Punkt haben.

Wir definieren eine Kette, als ein Menge von Punkte $\{p_1, ..., p_n\}$ und die Segmente $p_1 - p_2, ..., p_{n-1} - p_n$.

Ein Polygon ist eine Kette, wobei $n \geq 3$ und $p_1 = p_n$ gilt. Ein Polygon heißt einfach, falls sich 2 Segmenten nicht schneiden (Ausnahme: die benachbarten Segmente).

Ein einfaches Polygon zerteilt die Fläche in innere Punkten und aussere Punkten. Ein Polygon P heißt convex, falls für jede zwei innere Punkte p und q jeder Punkt des Segmentes $p - q$ ein innerer Punkt von P ist.

Sonderfälle von Anwendungen:

- Sehr große Anzahl der Punkte (über 10^6).
- Große Anzahl von besonderen Fälle.
- Klassifizierung der arithmetischen Operationen:

  Billige Operationen: Addition, Substarktion, Vergleichungen, Boolische Operationen.

  Teurer Operation: Multiplikation

  Teuerste Operationen: Division, arithmetische Wurzel.

2. Bestimmung des Schnittes von zwei Segmente

Das Problem:
Eingabe: Segmente $p_1 - p_2$ und $p_3 - p_4$ von positiver Länge.
Frage: Haben die Segmenten einen gemeinsamen Punkt ?

Sei $q_1 = (x_1, y_1)$ und $q_2 = (x_2, y_2)$. Wir definieren

$$q_1 \times q_2 = x_1 y_2 - x_2 y_1.$$

Zwei-Schritt Methode:

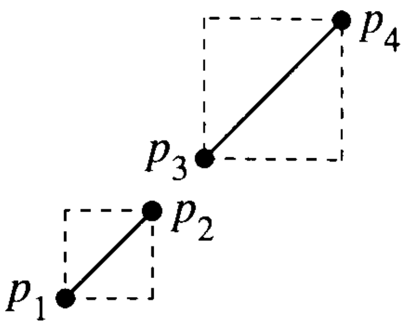
1. Quick rejection.
   Seien $p_1 = (x_1, y_1)$ und $p_2 = (x_2, y_2)$. Wir bezeichnen:

   $$\begin{array}{ll} \hat{x}_1 = \min\{x_1, x_2\}, & \hat{x}_2 = \max\{x_1, x_2\} \\ \hat{y}_1 = \min\{y_1, y_2\}, & \hat{y}_2 = \max\{y_1, y_2\}. \end{array}$$

   Wir betrachten das Rechteck $P = (\hat{p}_1, \hat{p}_2)$, wobei $\hat{p}_1 = (\hat{x}_1, \hat{y}_1)$ und $\hat{p}_2 = (\hat{x}_2, \hat{y}_2)$ gilt.
   Analog konstruiere das Rechteck $Q = (\hat{p}_3, \hat{p}_4)$.

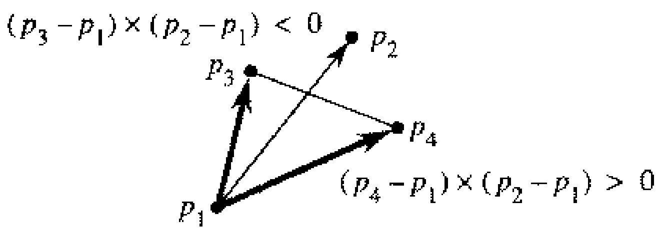
   Wir haben: $P \cap Q = \emptyset \Rightarrow (p_1 - p_2) \cap (p_3 - p_4) = \emptyset$.
   Darübe hinaus gilt: $P \cap Q \neq \emptyset \iff$

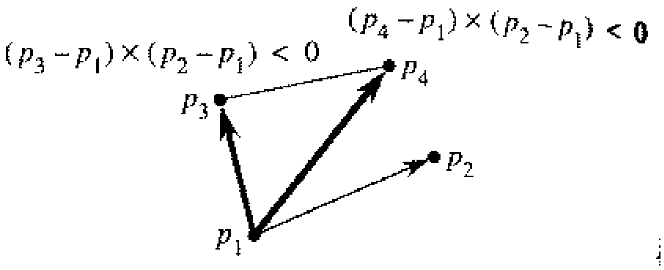
   $$(\hat{x_1} \geq \hat{x_3}) \wedge (\hat{x_4} \geq \hat{x_1}) \wedge (\hat{y_1} \geq \hat{y_3}) \wedge (\hat{y_4} \geq \hat{y_1}) = 1.$$



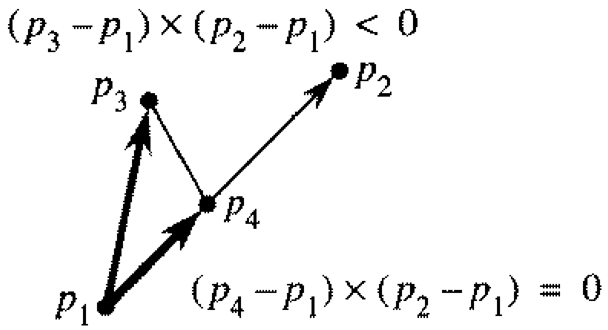
2. Es gilt:

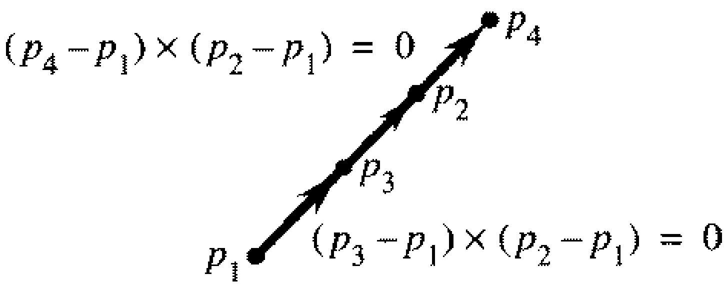
- sign$((p_3-p_1)\times(p_2-p_1)) \neq$ sign$((p_4-p_1)\times(p_2-p_1))$ $\Rightarrow p_1-p_2$ und p_3-p_4 schneiden sich.
- sign$((p_3-p_1)\times(p_2-p_1)) =$ sign$((p_4-p_1)\times(p_2-p_1))$ $\Rightarrow p_1-p_2$ und p_3-p_4 schneiden sich nicht.





- z.B. $(p_3-p_1)\times(p_2-p_1) \neq 0$ und $(p_4-p_1)\times(p_2-p_1) = 0$ $\Rightarrow p_4 \in p_1-p_2$.
- $(p_3-p_1)\times(p_2-p_1) = 0$ und $(p_4-p_1)\times(p_2-p_1) = 0$ $\Rightarrow p_3 \in p_1-p_2$ und $p_2 \in p_3-p_4$.





3. Innere Punkte des Polygons

Das Problem:
Eingabe: Ein Polygon P und ein Punkt $q = (x, y)$.
Frage: Bestimme ob q ein interner Punkt von P ist.

Idee: wir betrachten einen externen Punkt s von P und berechnen wieviel mal das Segment $q - s$ die Grenze von P schneidet. Falls diese Zahl gerade ist, dann ist q ein externer Punkt, sonst ist q ein interner Punkt.

Bestehe P aus den Punkte $\{p_0, ..., p_{n-1}\}$ und aus den Segmenten $\{e_0, ..., e_{n-1}\}$, wobei $e_i = p_i - p_{(i+1) \bmod n}$ gilt. Wir wählen eine vertikale Linie $-q - s-$ aus.

Algorithmus 6.1 PUNKT-IM-POLYGON(P, q);

Sei s ein externer Punkt von P und sei $L = q - s$
$c := 0$
for $i := 0$ **to** $n - 1$
  do if $e_i \cap L$ besteht aus 1 Punkt /* Vorsicht !!! */
    then $c := c + 1$
if c ist ungerade
  then Inside := TRUE
  else Inside := FALSE
return c

Mit diesen Algorithmus ergibt sich ein Problem, falls $e_i \cap L \in \{p_i, p_{(i+1) \bmod n}\}$ ist, aber es ist leicht das zu korregieren.

Die Laufzeit dieses Algorithmus ist allerdings $O(n)$.

4. Einfache Polygone

Eingabe: Eine Punktmenge $\{p_1, ..., p_n\}$.
Problem: Konstruiere ein einfaches Polygon mit Knoten in der Punkte $p_1, ..., p_n$.

Sei $n \geq 3$ und die Punkte p_1, p_2, p_3 sind nicht colinear. Wir setzen das 0 des Koordinatssystem in einen internen Punkte des Dreiecks p_1, p_2, p_3.

Weiterhin, definieren wir die Ordnung $\mathcal{L}$ der Punkte:
$p_i \leq_{\mathcal{L}} p_j \Longleftrightarrow$

(i) $\angle(p_1, 0, p_i) < \angle(p_1, 0, p_j)$ (d.h. die Ecke), oder

(ii) Falls $\angle(p_1, 0, p_i) = \angle(p_1, 0, p_j)$ gilt, dann ist $dist(0.p_i) < dist(0, p_j)$.

Algorithmus 6.2 POLYGON$(p_1, ..., p_n)$;

Sortiere die Punkte $p_1, ..., p_n$ bezüglich der Ordnung $\mathcal{L}$.
Verbinde den Punkt mit der Nummer i in Ordnung $\mathcal{L}$
mit dem Punkt, der Number $i + 1$ mod n hat
return Polygon

Die Laufzeit des POLYGON-Algorithmus ist durch die Sortierung definiert und ist allerdings $O(n \log n)$.

Das Convex Polygon Inclusion Problem: (CPI)
Gegeben seien ein konvexes Polygon P und ein Punkt z.
Bestimme ob z im Inneren von P liegt.

Behauptung 6.1 *Das CPI-Problem ist in $O(\log n)$ Zeit lösbar mit Preprozessing-Zeit $O(n)$.*

Beweisskizze:
Sei q ein Innererpunkt des Polygons $\{p_1, ..., p_n\}$. Die halbunendliche Linien der Form q, p_i zerteilen der Fläche in n Sektoren.

Wir stellen das polar Koordinatesystem (r, φ) mit dem Koordinatzentrum in q vor und berechnen die Winkeln $\angle(p_i)$, $i = 1, ..., n$.

Algorithmus 6.3

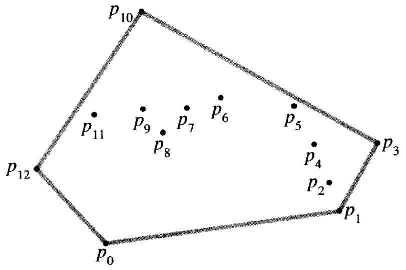
1. Bestimme i mit $\angle(p_i) \leq \angle(z) < \angle(p_{i+1})$.
2. Wenn die Segmente $\overline{q, z}$ und $\overline{p_i, p_{i+1}}$ sich schneiden,
   dann ist z ein äuser Punkt des Polygons.
   Sonst ist z ein inner Punkt.

Ein Polygon $P = \{p_1, ..., p_n\}$ heißt Star-Polygon, falls es einen inner Punkt q von P gibt, so daß jeder Punkt des Segmentes $\overline{q, p_i}$ ein inner Punkt von P ist.

Korollar 6.1 *Das PI-Problem für Star-Polygons ist in Zeit $O(\log n)$ lösbar mit $T_P = T_S = O(n)$.*

5. Konvexe Hülle

Definition 6.1 *Die Konvexe Hülle für eine Punktmenge Q ist das minimale konvexe Polygon, das alle Knoten von Q enthält. Schreibweise: $CH(Q)$.*



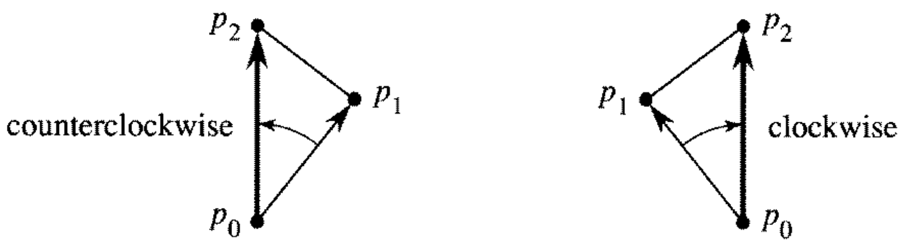
Triviale Lösung: $O(n^2)$ Operationen. Wir betrachten einen Algorithmus, der in der Zeit $O(n \log n)$ die Lösung konstruiert. Das beste Ergebnis: $O(n \log(|CH(Q)|))$ Operationen.

Bemerkung 6.1 *Seien p_0, p_1, p_2 Punkte. Dann gilt:*

$$p_1 \times p_2 = \begin{cases} > 0 \;\Leftrightarrow p_1 \textit{ ist clockwise von } p_2 \\ = 0 \;\Leftrightarrow p_1, p_2 \textit{ sind colinear} \\ < 0 \;\Leftrightarrow p_1 \textit{ ist counterclockwise von } p_2 \end{cases}$$

$$\Downarrow$$

$(p_1 - p_0) \times (p_2 - p_0) > 0 \Leftrightarrow$ *der Vektor p_0p_1 ist clockwise bezüglich des Vektors p_0p_2 und umgekehrt.*



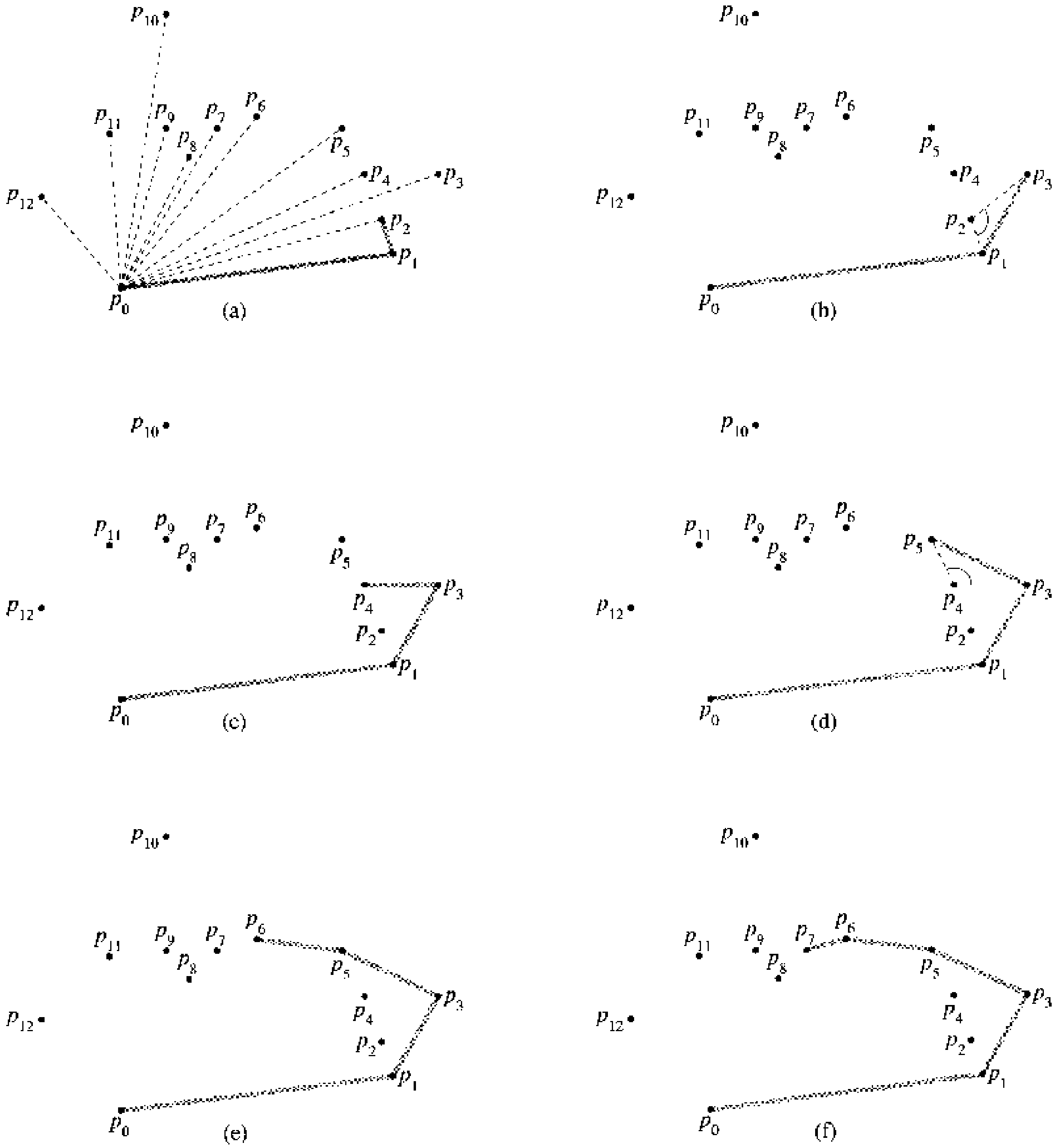
Sei $Q = \{p_0, ..., p_n\}$ $(n \geq 2)$ und sei p_0 der Punkt mit minimaler y-Koordinate. Falls mindestens 2 solche Punkte existieren, wählen wir den Punkt mit minimaler x-Koordinate aus.

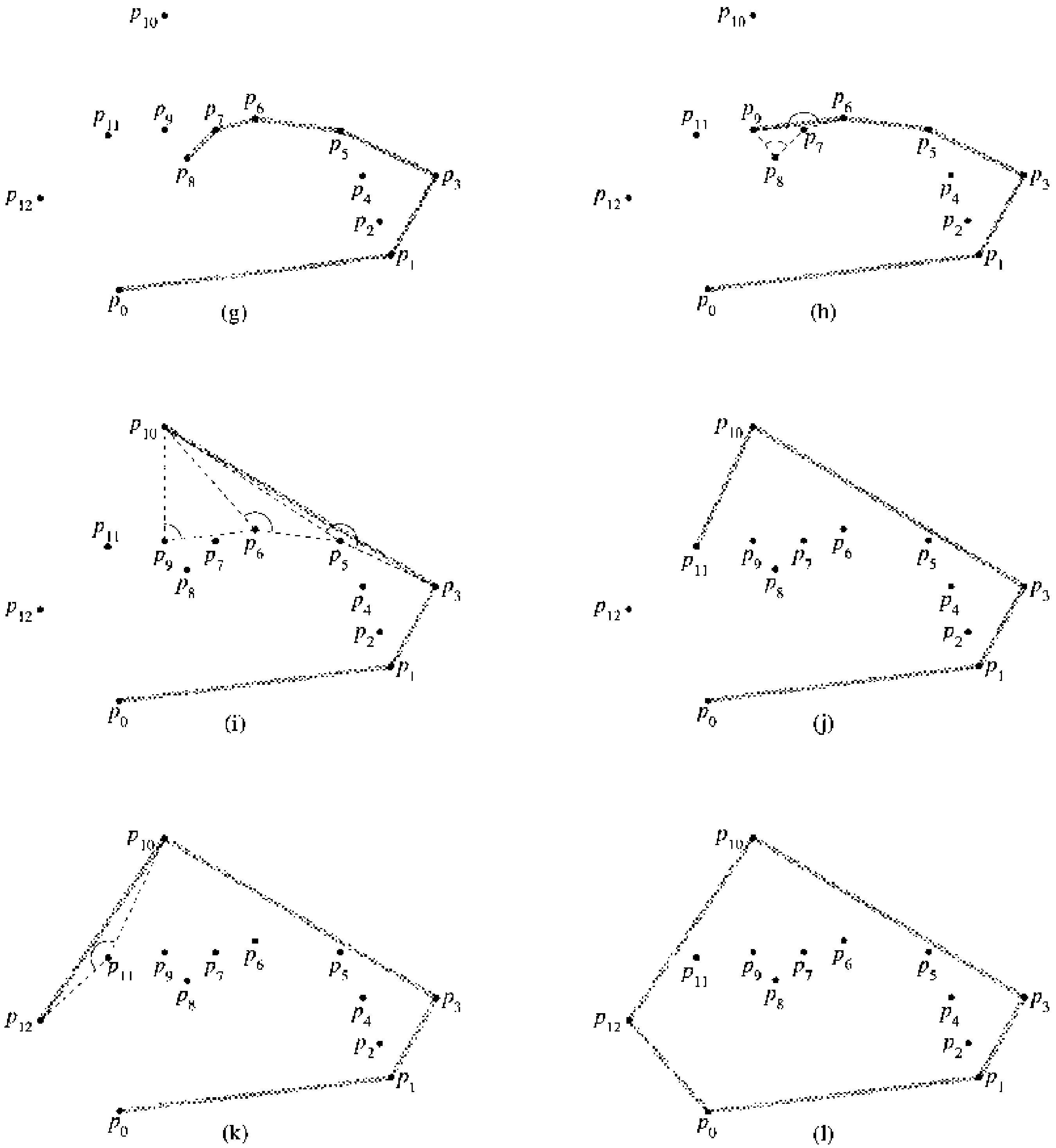
Wir nehmen an, daß die Menge der Punkte $\{p_1, ..., p_m\}$ bezüglich der Ecken der Vektoren p_0, p_i sortiert sind (in counterclockwise Ordnung). Falls mehr als 1 Punkt dieselbe Ecke mit p_0 hat, lassen wir nur den Punkt mit maximalem Abstand von p_0 stehen.

Wir benutzen einen Stack S. Die Prozedur $\textsc{Top}(S)$ liefert das obigste Element von S und ändert den Stack S nicht. Analog ergibt die Prozedur $\textsc{Next-to-Top}(S)$ das zweitoberste Element von S.

Algorithmus 6.4 $\textsc{Graham-Scan}(Q)$;

```
PUSH(p_0, S)
PUSH(p_1, S)
PUSH(p_2, S)
 for i := 3  to m
    do    while p_i nicht links von
                    NEXT-TO-TOP(S),TOP(S) liegt
               do POP(S)
          PUSH(p_i, S)
 return S
```

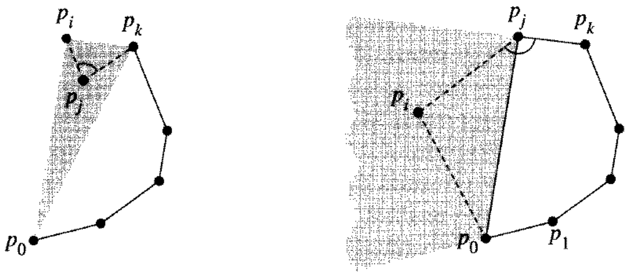

(a)
(b)
(c)
(d)
(e)
(f)
p_0 p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9 p_{10} p_{11} p_{12}


p_{10}
p_{11}
p_9
p_7
p_6
p_5
p_8
p_3
p_{12}
p_4
p_2
p_1
p_0
(g)
(h)
(i)
(j)
(k)
(l)

Theorem 6.1 *Ein Punkt p ist im Stack S des* Graham-Scan(Q)*-Algorithmus genau dann, wenn $p \in CH(Q)$ gilt.*

Beweisskizze: Falls $p_i \in p_0 - p_j$, dann gilt $p_i \notin S$.

Wir zeigen: $p_j \notin S \Rightarrow p_j \notin CH(Q)$.



$p_j \notin S \Rightarrow \exists i : p_j \in \Delta(p_0, p_i, p_k)$, wobei $p_i = \textsc{Top}(S)$ und $p_k = \textsc{Next-to-Top}(S)$ zum Zeitpunkt i.
$\Rightarrow p_j \notin CH(Q)$.

$$\Rightarrow \qquad CH(Q) \subseteq S$$

Wir zeigen jetzt die umgekehrte Richtung. Per Induktion:
Zu jedem Zeitpunkt der Abarbeitung von Graham-Scan, besteht S aus dem Punkte, die ein konvexes Polygon bilden.

$$\Rightarrow \qquad S \subseteq CH(Q)$$

<u>Die Laufzeit:</u>
Die Sortierung braucht $O(n \log n)$ Zeit.
Die Pop- und Push-Prozeduren brauchen $O(1)$ Zeit.
Die ganze **while** -Schleife läuft $O(n)$-mal, weil jeder Punkt nur einmal auf dem Stack gepusht wird.

$\Rightarrow$ Gesamtlaufzeit ist $O(n \log n)$.

6. Bestimmung der nächsten Nachbarn

Eingabe: Eine Menge von Punkte $P = \{p_i\}$, $|P| = n$.
Problem: Finde ein Paar von Punkte mit minimalem Abstand.

Triviale Lösung: teste alle Paare von Punkte. Das führt zu einem $\Theta(n^2)$-Algorithmus.

Wir benutzen die Divide and Conquer Methode um das Problem in Zeit $O(n \log n)$ zu lösen.

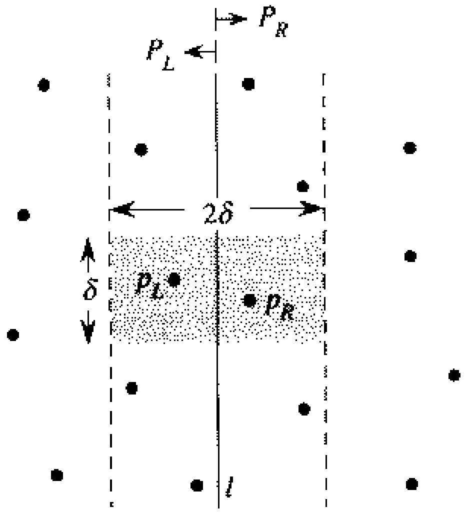
Unser Algorithmus besteht aus den folgenden Schritten:

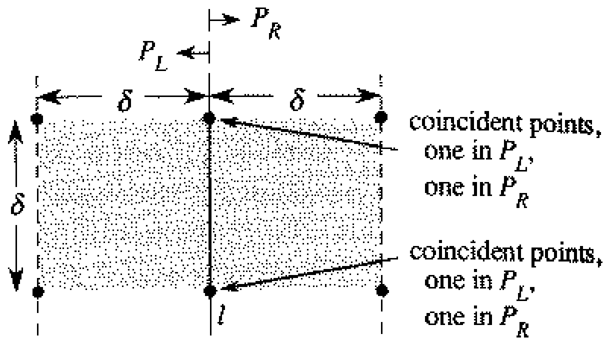
Divide: Wir zerteilen P dürch eine vertikale Linie ℓ in zwei gleichgroße Mengen P_L und P_R ($|P_L| = \lfloor n/2 \rfloor$, $|P_R| = \lceil n/2 \rceil$). Falls einige Punkte auf die Linie ℓ liegen, ordnen wir die belibig P_L oder P_R zu.

Conquer: Wir finden die nähsten Nachbarn in P_L und P_R (sei d_L und d_R die Abstände und sei $d = \min\{d_L, d_R\}$). Wir sortieren die Punkte in P_L und P_R bezüglich x- und y-Koordinaten und bekommen die sortierten Arrays X_L, X_R, Y_L und Y_R.

Kombinieren: Die nächsten Nachbarn x, y haben entweder den Abstand d, oder es gilt: $x \in P_L$ und $y \in P_R$. Wir betrachten nur die Punkte, höchstens den Abstand d von den Linie ℓ haben (die Z-Zone).

Falls die Punkte der Z-Zone bezüglich der Y-Koordinate sortiert sind, dann braucht man für jeden Punkt $x \in P_L \cap Z$ nur bis zu 7 Punkte aus $P_R \cap Z$ untersuchen.





Um die Punkte in der Z-Zone zu sortieren, benutzen wir die Induktionsvoraussetzung und die MERGE-Prozedur. Das erlaubt uns die Punkte von Z in $O(n)$ zu sortieren.

Deswegen gilt für die Laufzeit $T(n)$:

$$T(n) = 2T(n/2) + O(n),$$

was $T(n) = O(n \log n)$ ergibt.

7. Durchmesser einer Punktmenge

Wir definieren zwei Probleme:

DIAMETER
Gegeben seien n Punkte in E^2. Finde zwei maximal von einander entfernte Punkte.

DISJOINTNESS
Gegeben seien die Mengen A, B von positive Zahlen. Bestimme, ob $A \cap B \neq \emptyset$ gilt.

Theorem 6.2 (Ben-Or, 1983)
Um das DISJOINTNESS-*Problem zu lösen, sind* $O(n \log n)$ *Vergleich notwendig.*

Wir transformieren DISJOINTNESS zu DIAMETER:

Sei C der Kreis mit Radius 1 und mit Zentrum $(0, 0)$.
Für $a_i \in A$ sei a_i' der gemeinsame Punkt von C und der Linie $y = a_i x$ (mit $x > 0$).
Für $b_j \in B$ sei b_j' der gemeinsame Punkt von C und der Linie $y = b_j x$ (mit $x < 0$).
Offensichtlich, gilt: Diam$(\{a_i\} \cup \{b_j\}) = 2 \Leftrightarrow A \cap B \neq \emptyset$.

Somit erhalten wir:

Theorem 6.3 *Zur Bestimmung des Durchmessers sind* $\Omega(n \log n)$ *Operationen notwendig.*

Theorem 6.4 (Hocking-Young, 1961)
Der Durchmesser einer PunktMenge ist gleich dem Durchmesser ihrer konvexe Hülle.

Theorem 6.5 (Yaglom-Boltyanskii, 1961)
Der Durchmesser einer konvexen Region ist dergrößte Abstand zwischen tangentialen Linien.

Wir nennen zwei Punkte einer konvexen Polygons antipodal, falls es zwei tangentiale parallele Linien gibt, die durch diese Punkte gehen.

Unser Ziel ist es, all Paare von antipodalen Punkten zu bestimmen.

Sei P ein enfaches Polygon und seien seine Punkte in counterclockwise Ordnung numeriert. Wir starten in Punkt $p_i \in P$, gehen durch die Punkte von P und konstruieren den ersten Punkt $q_R \in P$, der maximal von der Linie (p_{i-1}, p_i) entfernt ist.

Danach gehen wir weiter und finden einen Punkt q_L, der maximal von der Linie (p_i, p_{i+1}) entfernt ist. Dann ist die Menge der Punkte zwischen q_R und q_L (inklusive die Punkte q_R, q_L) gleich aller Punkte, die antipodale Paare mit p_i formen.

Wir nehmen an, daß die Punkte $p_0, ..., p_{n-1}$ von P in zyklischer Ordnung dargestellt sind.

<u>**Algorithmus** 6.5</u> ANTIPAAR;

1. $i := n - 1$
2. $q := \textsc{Next}(p_i)$
3. **while** $\mathsf{Dist}(p_i, p_{i+1}; \textsc{Next}(q)) > \mathsf{Dist}(p_i, p_{i+1}; q)$ **do**
   $q := \textsc{Next}(q)$
4. $q_0 := q$

5. **while** $q \neq p_0$ **do**
6. $i := i + 1$ (mod n)
7. Output(p, q)
8. **while** $\mathsf{Dist}(p_i, p_{i+1}; \textsc{Next}(q)) > \mathsf{Dist}(p_i, p_{i+1}; q)$ **do**
9. $q := \textsc{Next}(q)$
10. **if** $(p, q) \neq (p_0, q_0)$ **then** Output(p, q)
11. **if** $\mathsf{Dist}(p_i, p_{i+1}; \textsc{Next}(q)) = \mathsf{Dist}(p_i, p_{i+1}; q)$ **then**
12. **if** $(p, q) \neq (q_0, p_{n-1})$ **then** Output$(p, \textsc{Next}(q))$

Da die Anzahl der Paare der parallelen Segmenten des Polygons höchstens $\lfloor n/2 \rfloor$ ist, ist die Anzahl der antipodalen Paare höchstens $3n/2$. Somit erhalten wir:

Theorem 6.6 *Der Durchmesser eines konvexes Polygones ist in $O(n)$ Zeit berechenbar.*

Korollar 6.2 *Der Durchmesser einer Menge von n Punkte ist in $O(n \log n)$ Zeit berechenbar.*

Korollar 6.3 *Der Durchmesser eines einfachen Polygons ist in linearer Zeit berechenbar.*

p	q	Print	Step
p_9			1
	p_0		2
	p_1		3
	p_2		3
	p_3		3
	p_4		3
	p_5		3
p_0			6
		$(p_0\ p_5)$	7
p_1			6
		$(p_1\ p_5)$	7
	p_6		9
		$(p_1\ p_6)$	10
		$(p_1\ p_7)$	12
p_2			6
		$(p_2\ p_6)$	7
	p_7		9
		$(p_2\ p_7)$	10
	p_8		9
		$(p_2\ p_8)$	10
	p_9		9
		$(p_2\ p_9)$	10
p_3			6
		$(p_3\ p_9)$	7
p_4			6
		$(p_4\ p_9)$	7
p_5			6
		$(p_5\ p_9)$	7
	p_0		9

While Loop

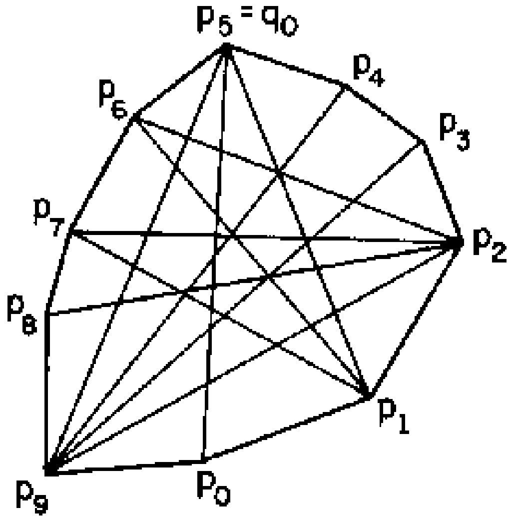


Figure 4.22 Illustration of ANTIPODAL PAIRS. Note that $\overline{p_1p_2}$ and $\overline{p_6p_7}$ are parallel edges.

7 Lektion 7: NP-Vollständigkeit

1. Die Komplexitätklasse P
   a. Kodierung der Eingabe
   b. Formale Sprachen
2. Die Komplexitätklasse NP
   a. NP-Vollständigkeit und Reduzierbarkeit
   b. Beweismethode zur NP-Vollständigkeit
3. Einige NP-vollständige Probleme
   a. SAT
   b. 3-SAT
   c. Clique
   d. Partition
   e. k-Färbung
   f. 3D-Matching

1a. Die Komplexitätklasse P

Ein abstraktes Problem ist eine Relation auf die Problemeingaben und Problemlösungen.

Beispiel: Das KÜRZESTE-WEGE-Problem:
Eingabe: Ein einfacher Graph G, zwei Knoten u, v.
Lösung: Ein kürzester (u, v)-Weg zwischen u und v in G (falls so eine Weg existiert).

Entscheidungsprobleme:
Die Lösung ist von der Form "Y" oder "N".

Beispiel:
Eingabe: Ein einfacher Graph G, zwei ausgezeichnete Knoten u, v und $k > 0$.
Frage: existiert ein (u, v)-Weg in G der Länge $\leq k$?

Optimierungsprobleme:
Eine Funktion soll minimiert oder maximiert werden.

Jedes "diskretes Optimizationsproblem" kann auch als ein Entscheidungsproblem formuliert werden.

Bemerkung 7.1
Optimierungsproblem ist "schnell lösbar" $\Rightarrow$ korrespondierendes Entscheidungsproblem ist auch "schnell lösbar".

Entscheidungsproblem ist "schwer lösbar" $\Rightarrow$ korrespondierendes Optimierungsproblem ist auch "schwer lösbar".

Eine Kodierung ist eine Abbildung von der Menge abstrakter Objekten in dei Menge der binären Folgen.

Jeder Algorithmus der ein abstraktes Entscheidungsproblem "löst", arbeitet auf der Kodierung dieses Problems als Eingabe. Wir nennen ein Problem mit kodierter Eingabe konkretes Problem.

Definition 7.1 *Wir sagen, daß ein Algorithmus ein Problem in Zeit $O(T(n))$ löst, falls für jede Eingabekodierung der Länge n der Algorithmus eine Lösung in Zeit $O(T(n))$ bestimmt.*

Definition 7.2 *Ein konkretes Problem mit Eingabekodierungslänge n ist in Polynomzeit lösbar, falls ein Algorithmus existiert, der dieses Problem in Zeit $O(n^k)$ für eine Konstante k löst.*

Definition 7.3 *Die Komplexitätklasse P ist die Menge aller konkreten Entscheidungsprobleme, die in Polynomzeit lösbar sind.*

Sei f eine Abbildung $f : \{0,1\}^* \mapsto \{0,1\}^*$. f heißt in Polynomzeit berechenbar, falls ein Algorithmus existiert, der für jedes $x \in \{0,1\}^*$ die Folge $f(x)$ in Polynomzeit konstruiert.

Sei I eine Menge von Problemeingaben. Wir nennen zwei Kodierungen e_1 und e_2 polynomiell-ähnlich, falls zwei in Polynomzeit berechenbare Funktionen f_{12} und f_{21} existieren, so daß für jedes $i \in I$ gilt: $f_{12}(e_1(i)) = e_2(i)$ und $f_{21}(e_2(i)) = e_1(i)$.

Lemma 7.1 *Sei Q ein abstraktes Entscheidungsproblem und e_1 und e_2 zwei polynomiell-ähnliche Kodierungen von Eingabemenge $I = \{i\}$. Dann gilt $Q(e_1(i)) \in P \iff Q(e_2(i)) \in P$.*

1b. Elemente der Theorie der Formalen Sprachen

Sei $\Sigma = \{0, 1\}$. Eine Sprache L ist eine Teilmenge von Σ^*. Falls Q ein Entscheidungsproblem ist, kann man Q auch als die folgende Sprache repräsentieren:

$$L = \{x \in \Sigma^* \mid Q(x) = 1\}.$$

Definition 7.4 *Ein Algorithmus A akzeptiert die Folge $x \in \Sigma^*$, falls der Output $A(x)$ gleich 1 ist. Der Algorithmus A lehnt eine Folge $x \in \Sigma^*$ ab, falls $A(x) = 0$ gilt.*

Die von dem Algorithmus A akzeptierte Sprache ist die Menge $L = \{x \in \Sigma^* \mid A(x) = 1\}$.

Eine Sprache L wird von dem Algorithmus A entscheiden, falls für jedes $x \in \Sigma^*$ entweder A akzeptiert x oder A lehnt x ab gilt.

Definition 7.5 *Die Sprache L ist von dem Algorithmus A in Polynomzeit akzeptierbar, falls jedes $x \in L$ mit $|x| = n$ in Zeit $O(n^k)$ von A akzeptiert wird.*

Die Sprache L ist von dem Algorithmus A in Polynomzeit entscheidbar, falls jedes $x \in \Sigma^*$ mit $|x| = n$ in Zeit $O(n^k)$ von A entscheidbar wird.

Weitere Definitionen der Komplexitätsklasse P:

$$P = \{L \subseteq \Sigma^* \mid \exists A \text{ der } L \text{ in polynom. Zeit entscheidet}\}.$$

Theorem 7.1 (siehe [1] Theorem 36.2 auf Seite 923)

$$P = \{L \subseteq \Sigma^* \mid L \textit{ wird von einem polynom.-Zeit Alg. akzeptiert}\}.$$

2a. Die Komplexitätsklasse NP

Seien x und y binäre Folgen.

Definition 7.6 *Ein Prüfungsalgorithmus ist ein Algorithmus mit zwei argumenten. Wir sagen: Algorithmus A prüft eine Folge x, falls eine Folge y existiert, so daß $A(x, y) = 1$ gilt.*

Der Algorithmus A prüft eine Sprache L, falls gilt:

$$L = \{x \in \Sigma^* \mid \exists y \in \Sigma^* \text{ mit } A(x, y) = 1\}.$$

Definition 7.7 *Die Komplexitätsklasse NP ist die Menge aller Sprachen L, für die ein polynomieller Prüfungsalgorithmus A und eine Konstante c existieren, so daß:*

$$L = \{x \in \Sigma^* \mid \exists y \text{ mit } |y| = O(|x|^c),\ A(x, y) = 1\}.$$

Offensichtlich gilt: P $\subseteq$ NP. Die Frage ist aber, ob P $\neq$ NP gilt?

Definition 7.8 *Eine Sprache L_1 heißt in Polynomzeit auf Sprache L_2* reduzierbar, *falls eine in Polynomzeit berechenbare Funktion $f : \Sigma^* \mapsto \Sigma^*$ existiert, so daß für alle $x \in \Sigma^*$ gilt:*

$$x \in L_1 \qquad \Longleftrightarrow \qquad f(x) \in L_2.$$

<u>Schreibweise:</u> $L_1 \leq L_2$.

Wir schreiben $L_1 \equiv L_2$, falls $L_1 \leq L_2$ und $L_2 \leq L_1$ gilt.

Lemma 7.2 *Seien $L_1, L_2 \subseteq \Sigma^*$ zwei Sprachen und $L_1 \leq L_2$. $L_2 \in P$ impliziert $L_1 \in P$.*

Definition 7.9 *Sei $L \subseteq \Sigma^*$ eine Sprache.*

1. *Die Sprache L heißt NP-schwierig, falls $L' \leq L$ für alle Sprachen $L' \in$ NP gilt.*
2. *Die Sprache L heißt NP-vollständig, falls L NP-schwierig ist und $L \in$ NP. Schreibweise: $L \in$* NPC.

Theorem 7.2

1. *Wenn ein NP-vollständiges Problem in Polynomzeit lösbar ist, dann gilt P=NP.*
2. *Wenn ein Problem aus NP nicht in Polynomzeit lösbar ist, dann ist kein NP-vollständiges Problem in Polynomzeit lösbar.*

Beweisskizze:

1. Sei $L \in$ NPC und $L \in$ P.
   $\Rightarrow L' \leq L$ für jedes Problem $L' \in$ NP (Definition 7.9).
   $\Rightarrow L' \in$P (Lemma 7.2).
2. Angenommen, $\exists L \in$ NP mit $L \notin$ P.
   Sei $L' \in$ NPC. $\Rightarrow L \leq L'$ (Definition 7.9).
   Annahme: $L' \in$ P $\Rightarrow L \in$ P (Lemma 7.2). Widerspruch.

2b. NP-Vollständigkeits-Beweise

Lemma 7.3 *Sei L eine Sprache, so daß $L' \leq L$ für ein $L' \in$ NPC. Dann ist L NP-schwierig. Falls zusatzlich $L \in$ NP, dann gilt: $L \in$ NPC.*

Beweisskizze:
$L' \in$ NPC $\Rightarrow L'' \leq L'$ für alle $L'' \in$ NP.
Damit und $L' \leq L \Rightarrow L'' \leq L$
$\Rightarrow L$ ist NP-schwierig.
$\Rightarrow L \in$ NPC, falls $L \in$ NP gilt.

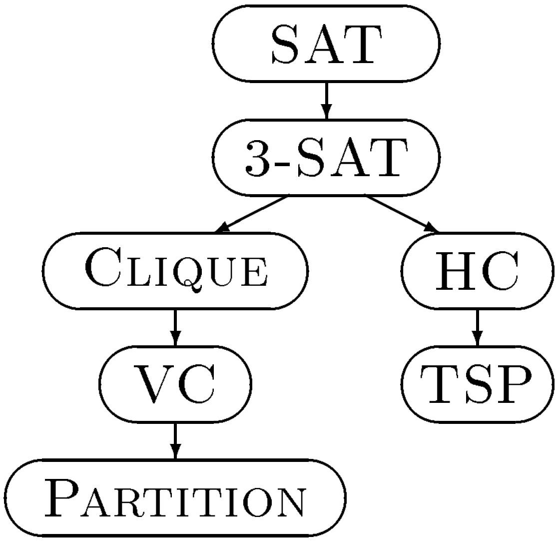
Eine Methode für NP-Vollständigkeits-Beweise.

1. Zeige: $L \in$ NP.
2. Wähle eine Sprache L' aus, vor der bekannt ist, daß sie NP-vollständig ist.
3. Finde einen Algorithmus, der eine Funktion f berechnet, die jede Eingabe für L' in eine Eingabe für L kodiert.
4. Beweise, daß $x \in L' \Leftrightarrow f(x) \in L$ für alle $x \in \{0,1\}^*$.
5. Beweise, daß die Funktion f in Polynomzeit berechenbar ist.

Theorem 7.3 (Cook).
Es gilt:

$$\mathrm{SAT} \in \text{NPC}.$$

3. Einige NP-vollständige Probleme



SATISFABILITY (SAT):

Eingabe: Boolesche Formel F.

Frage: Ist F erfüllbar?

3-SAT:

Wie SAT, aber die Formel ist in KNF, wobei jede Klausel hat genau drei Literale.

CLIQUE:

Eingabe: Graph G und $k \in \mathbb{N}$.

Frage: Gibt es eine k-Clique in G?

HAM-CYCLE (HC):

Eingabe: Graph $G = (V, E)$.

Frage: Gibt es einen einfachen Kreis in G der Länge $|V|$?

VERTEX-COVER (VC):

Eingabe: Graph G und $k \in \mathbb{N}$.

Frage: Gibt es $C \subset V$ der Größe k, so daß jede Kante von G zu einem Knoten aus C inzident ist?

Theorem 7.4 $\quad$ 3-SAT $\in$ NPC.

Beweisskizze. Wir zeigen $\mathrm{SAT} \leq$ 3-SAT.
Gegeben sein eine Boolesche Formel f (Eingabe für SAT), wir konstruieren eine Formel f' (Eingabe für 3-SAT).

Schritt 1. Für jede "interne Teilformel" bilden wir eine neue Variable y_i.
Beispiel:

$$f = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

$$\begin{aligned} f' = y_1 \;&\wedge\; (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ &\wedge\; (y_2 \leftrightarrow (y_3 \vee y_4)) \\ &\wedge\; (y_3 \leftrightarrow (x_1 \to x_2)) \\ &\wedge\; (y_4 \leftrightarrow \neg y_5) \\ &\wedge\; (y_5 \leftrightarrow (y_6 \vee x_4)) \\ &\wedge\; (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)) \end{aligned}$$

Schritt 2. Wir transformieren jede Klausel C_i für f' in konjunktive Normalform und bekommen die Formel f''.

Schritt 3. Wir machen jede Klausel C_i für f'' von genau 3 Variablen abhängig und bekommen die Formel f'''.

- $C_i = l_1 \vee l_2 \vee l_3 \Rightarrow C_i' := C_i \in f'''$.
- $C_i = l_1 \vee l_2 \Rightarrow C_i' := (l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$.
- $C_i = l \Rightarrow C_i' := (l \vee p \vee q) \wedge (l \vee \neg p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee \neg q)$.

Die Formel f ist erfüllbar $\iff$ f''' ist erfüllbar.

Die Formel f''' ist in Polynomzeit konstruierbar.

Somit gilt: 3-SAT $\in$ NP. $\square$

$$\textsc{Clique} = \{(G, k) \mid G \text{ enthält eine Clique mit } k \text{ Knoten}\}.$$

Theorem 7.5 $\quad$ $\textsc{Clique} \in$ NPC.

Beweisskizze.
Offensichtlich gilt: $\textsc{Clique} \in$ NP.

Wir zeigen: 3-SAT $\leq$ $\textsc{Clique}$.
Sei $f = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ eine Eingabe für 3-SAT mit $C_i = l_1^i \vee l_2^i \vee l_3^i$.

Wir konstruieren einen Graphen $G = (V, E)$ mit

$$V = \{v_1^i, v_2^i, v_3^i \mid i = 1, ..., k\}$$

und $(v_r^i, v_s^j) \in E$ genau dann, wenn

- $i \neq j$ und
- $l_r^i \neq \neg l_s^j$.

Die Formel f ist erfüllbar $\iff$ G enthält eine Clique mit k Knoten.

Der Graph G ist in Polynomzeit konstruierbar. $\square$

Partition:

Eingabe: Eine Menge $S = \{s\}$ von ganzen Zahlen, $t \in I\!N$.
Frage: Gibt es eine Teilmenge $S' \subseteq S$ mit $\sum_{s \in S'} s = t$?

Theorem 7.6 Partition $\in$ NPC.

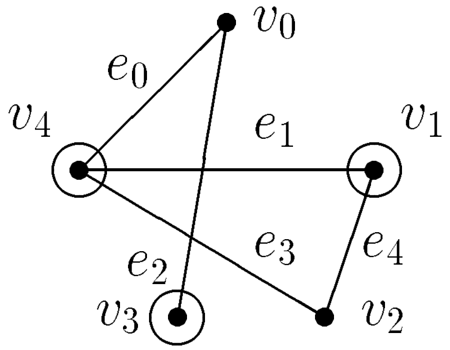
Beweisskizze. Offensichtlich gilt: Partition $\in$ NP.
Wir zeigen: $\mathrm{VC} \leq$ Partition.

Sei $G = (V, E)$ eine Eingabe für VC mit

$$V = \{v_0, ..., v_{n-1}\} \qquad \text{und} \qquad E = \{e_0, ..., e_{m-1}\}.$$

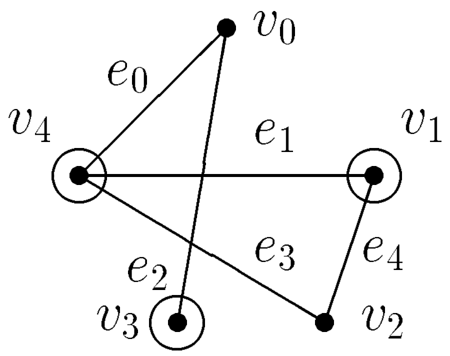
Wir repräsentieren G durch seine Inzidenzmatrix $B = \{b_{ij}\}$, wobei $b_{ij} \in \{0, 1\}$ und $b_{ij} = 1 \Longleftrightarrow e_i$ ist inzident zu v_j.

$$b_{ij} = \begin{cases} 1, & \text{falls } e_i \text{ ist inzident mit } v_j \\ 0, & \text{sonst} \end{cases}$$



$$\begin{array}{c|ccccc|}
B & e_4 & e_3 & e_2 & e_1 & e_0 \\
\hline
v_0 & 0 & 0 & 1 & 0 & 1 \\
v_1 & 1 & 0 & 0 & 1 & 0 \\
v_2 & 1 & 1 & 0 & 0 & 0 \\
v_3 & 0 & 0 & 1 & 0 & 0 \\
v_4 & 0 & 1 & 0 & 1 & 1 \\
\hline
\end{array}$$

Beispiel:



$$\begin{array}{c|ccccc|}
B & e_4 & e_3 & e_2 & e_1 & e_0 \\
\hline
v_0 & 0 & 0 & 1 & 0 & 1 \\
v_1 & 1 & 0 & 0 & 1 & 0 \\
v_2 & 1 & 1 & 0 & 0 & 0 \\
v_3 & 0 & 0 & 1 & 0 & 0 \\
v_4 & 0 & 1 & 0 & 1 & 1 \\
\hline
\end{array}$$

Wir setzen für $i = 0, ..., n-1$ und $j = 0, ..., m-1$:

$$x_i = 4^m + \sum_{j=0}^{m-1} b_{ij} 4^j, \qquad y_j = 4^j, \qquad t = k \cdot 4^m + \sum_{j=0}^{m-1} 2 \cdot 4^j$$

und erweitern die Matrix B:

$$\begin{array}{ccccccccc r}
x_0 & = & 1 & 0 & 0 & 1 & 0 & 1 & = & 1041 \\
x_1 & = & 1 & 1 & 0 & 0 & 1 & 0 & = & 1284 \\
x_2 & = & 1 & 1 & 1 & 0 & 0 & 0 & = & 1344 \\
x_3 & = & 1 & 0 & 0 & 1 & 0 & 0 & = & 1040 \\
x_4 & = & 1 & 0 & 1 & 0 & 1 & 1 & = & 1093 \\
y_0 & = & 0 & 0 & 0 & 0 & 0 & 1 & = & 1 \\
y_1 & = & 0 & 0 & 0 & 0 & 1 & 0 & = & 4 \\
y_2 & = & 0 & 0 & 0 & 1 & 0 & 0 & = & 16 \\
y_3 & = & 0 & 0 & 1 & 0 & 0 & 0 & = & 64 \\
y_4 & = & 0 & 1 & 0 & 0 & 0 & 0 & = & 256 \\
\hline
t & = & 3 & 2 & 2 & 2 & 2 & 2 & = & 3754
\end{array}$$

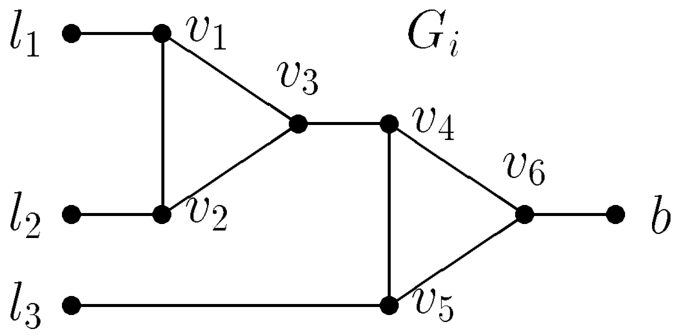
Wir zeigen: G hat Knotenüberdeckung der Größe $k \iff \exists S' \subseteq S$ mit $\sum_{s \in S'} s = t$. $\square$

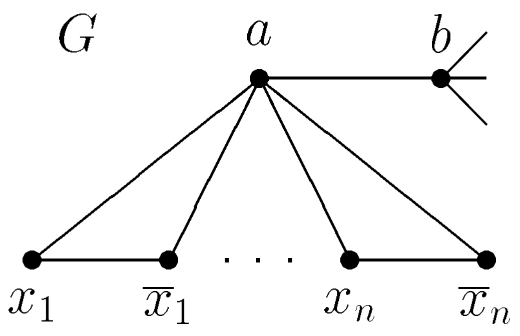
3-Färbung:
Eingabe: Ein Graph $G = (V, E)$.
Frage: Ist G 3-färbbar ?

Theorem 7.7 3-Färbung $\in$ NPC.

Beweisskizze. Offensichtlich gilt: 3-Färbung $\in$ NP.
Wir zeigen: 3-SAT $\leq$ 3-Färbung.

Sei $f = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ (hier $f = f(x_1, ..., x_n)$) eine Eingabe für 3-SAT. Wir konstruieren für jede Klausel $C_i = l_1 \vee l_2 \vee l_3$ einen Graphen $G_i = (V_i, E_i)$, $1 \leq i \leq m$:





Angenommen, die Knoten l_1, l_2, l_3 sind nur mit den Farben 0 oder 1 gefärbt. Dann kann der Knoten v_6 mit der Farbe 1 oder 2 gefärbt werden $\iff \exists\ l_i$, $1 \leq i \leq 3$, welches die Farbe 1 hat.

Wir konstruieren eine Eingabe $G = (V, E)$ für 3-Färbung:

$$V = \{a, b\} \bigcup_{i=1}^{m} V_i$$

$$E = \{(a, b)\} \cup \{(a, x_i),\ (a, \overline{x}_i),\ (x_i, \overline{x}_i) \mid 1 \leq i \leq n\} \bigcup_{i=1}^{m} E_i.$$

Wir zeigen: G ist 3-färbbar $\iff f$ ist erfüllbar. $\square$

3D Matching:

Eingabe: Eine Menge $M \subseteq X \times Y \times Z$, wobei X, Y, Z disjunkte Mengen mit $|X| = |Y| = |Z| = q$ sind.
Frage: Existiert eine Menge $M' = \{(x, y, z)\} \subseteq M$ mit $|M'| = q$, so daß die Tripel aus M' keine gleiche Koordinate haben ?

Theorem 7.8 3D Matching $\in$ NPC.

Beweisskizze. Offensichtlich gilt: 3D Matching $\in$ NP.
Wir zeigen: 3-SAT $\leq$ 3D Matching.

Sei $f = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ (hier $f = f(x_1, ..., x_n)$) eine Eingabe für 3-SAT. Wir fügen für jeden Variable x_i die Elemente x_i^j und $\bar{x}_i^j$ ein, $j = 1, ..., m$, und setzen

$$X = \{x_i^j, \bar{x}_i^j \mid j = 1, ..., m\}, \qquad q = 2mn.$$

Wir konstruieren die Menge $T_i = T_i^t \cup T_i^f$ von Tripeln:

$$\begin{aligned} T_i^t &= \{(\bar{x}_i^j, a_i^j, b_i^j) \mid 1 \leq j \leq m\}, \\ T_i^f &= \{(x_i^j, a_i^{j=1}, b_i^j) \mid 1 \leq j < m\} \cup \{(x_i^m, a_i^1, b_i^m)\} \end{aligned}$$

Die Elemente $\{a_i^j, b_i^j\}$ (interne Elemente) werden nur in Tripel T_i benutzt und die Elemente von X werden in anderen Tripel benutzt.

Bemerkung 7.2 *Jedes Matching M' enthält entweder alle Tripel von T_i^t oder alle Tripel von T_i^f, $i = 1, ..., m$.*

Weiterhin fügen wir für jede Klausel C_j die Elemente s_1^j und s_2^j ein und für $j = 1, ..., m$ konstruieren wir die Menge von Tripeln

$$C_j = \{(u_i^j, s_1^j, s_2^j) \mid u_i \in C_j\} \cup \{(\bar{u}_i^j, s_1^j, s_2^j) \mid \bar{u}_i \in C_j\}.$$

Bemerkung 7.3 *Jedes Matching M' enthält genau 1 Tripel von C_j, $j = 1, ..., m$.*

So weit haben wir:

$$\begin{aligned} Y &= \{a_i^j, s_1^j \mid 1 \leq i \leq n,\ 1 \leq j \leq m\}, \\ Z &= \{b_i^j, s_2^j \mid 1 \leq i \leq n,\ 1 \leq j \leq m\} \end{aligned}$$

mit $|Y| = |Z| = m(n+1)$.

Schließlich erweitern wir die Mengen Y, Z und definieren $m(n-1)$ neue Elemente g_1^k und g_2^k, $k = 1, ..., m(n-1)$. Wir konstruieren eine "garbage collection"

$$G = \{(x_i^j, g_1^k, g_2^k), (\bar{x}_i^j, g_1^k, g_2^k) \mid 1 \leq k \leq m(n-1),\ \forall i,\ \forall j\}.$$

Somit erhalten wir:

$$M = \left[\bigcup_{i=1}^{n} T_i\right] \cup \left[\bigcup_{j=1}^{m} C_j\right] \cup G$$

mit $|M| = 2mn + 3m + 2m^2n(n-1)$.

Bemerkungen 2.3 und 2.3 stellen sicher daß es kein Matching existiert, falls f nicht erfüllbar ist.

Falls f unter eine Abbildung $t : X \mapsto \{T, F\}$ erfüllbar ist, wählen wir das folgenden Matching aus:

$$M' = \left[\bigcup_{t(x_i)=T} T_i^t\right] \cup \left[\bigcup_{t(x_i)=F} T_i^f\right] \cup \left[\bigcup_{j=1}^{m} \{(z_i^j, s_1^j, s_2^j)\}\right] \cup G',$$

wobei $z_i^j \in c_j$ ein Literal mit $t(z_i^j) = T$ ist und $G' \subseteq G$ enthält alle g_1^k, g_2^k und die Reste von X, die noch nicht benutzt wurden.
$\square$

Beispiel: Sei $f = x_1 \vee x_2 \vee \bar{x}_3$.
Wir haben: $m = 1$, $n = 3$, $q = 6$, $|M| = 21$, $|M'| = 6$,

$$\begin{aligned}
X &= \{x_1^1, \bar{x}_1^1, x_2^1, \bar{x}_2^1, x_3^1, \bar{x}_3^1\} \\
Y &= \{a_1^1, a_2^1, a_3^1, s_1^1, g_1^1, g_2^1\} \\
Z &= \{b_1^1, b_2^1, b_3^1, s_2^1, g_1^1, g_2^1\},
\end{aligned}$$

und

$$\begin{aligned}
M = \;& \{(\bar{x}_1^1, a_1^1, b_1^1),\ (x_1^1, a_1^1, b_1^1),\ (\bar{x}_2^1, a_2^1, b_2^1), \\
& (x_2^1, a_2^1, b_2^1),\ (\bar{x}_3^1, a_3^1, b_3^1),\ (x_3^1, a_3^1, b_3^1)\} \\
\cup \;& \{(x_1^1, s_1^1, s_2^1),\ (x_2^1, s_1^1, s_2^1),\ (\bar{x}_3^1, s_1^1, s_2^1)\} \\
\cup \;& \{(x_1^1, g_1^1, g_2^1),\ (\bar{x}_1^1, g_1^1, g_2^1),\ (x_1^1, g_2^1, g_2^2), \\
& (\bar{x}_1^1, g_1^2, g_2^2),\ (x_2^1, g_1^1, g_2^1),\ (\bar{x}_2^1, g_1^1, g_2^1), \\
& (x_2^1, g_1^2, g_2^2),\ (\bar{x}_2^1, g_1^2, g_2^2),\ (x_3^1, g_1^1, g_2^1), \\
& (\bar{x}_3^1, g_1^1, g_2^1),\ (x_3^1, g_1^2, g_2^2),\ (\bar{x}_3^1, g_1^2, g_2^2)\}.
\end{aligned}$$

Für $x_1 = T$, $x_2 = T$, $x_3 = T$ erhalten wir:

$$\begin{aligned}
M' = \;& \{(x_1^1, s_1^1, s_2^1),\ (\bar{x}_1^1, a_1^1, b_1^1),\ (x_2, g_1^1, g_2^1), \\
& (\bar{x}_2^1, a_2^1, b_2^1),\ (x_3^1, a_3^1, b_3^1),\ (\bar{x}_3^1, g_1^2, g_2^2)\}.
\end{aligned}$$

8 Lektion 8: Approximations-Algorithmen für NP-vollständige Probleme

1. Approximationstypen
2. Einige polynomielle Approximations-Algorithmen
   a. Vertex-Cover
   b. TSP
   c. Scheduling
   d. Set-Cover
   e. Maximum Set-Cover
   f. Independent-Set
   g. Subset-Sum
   h. 3-Färbung

1. Definitionen

Definition 8.1 *Ein Approximations-Algorithmus hat die* Güte $p(n)$, *falls für jede Eingabe der Größe* n *gilt:*

$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \leq p(n),$$

wobei C *und* C^* *die Werte der approximative und exakte Lösungen sind.*

Definition 8.2 *Ein Approximations-Algorithmus macht einen* relativen Fehler $\epsilon(n)$, *falls für jede Eingabe der Größe* n *gilt:*

$$\frac{|C - C^*|}{C^*} \leq \epsilon(n).$$

Bemerkung 8.1 *Offensichtlich gilt:*
$\epsilon(n) \leq p(n) - 1$ *für ein Minimierungsproblem,*
$\epsilon(n) \leq (p(n) - 1)/p(n)$ *für ein Maximierungsproblem.*

Üblicherweise gibt es eine Korrelation zwischen der Laufzeit eines Algorithmus und seiner Approximations-Qualität.

Definition 8.3 *Ein Approximationsschema für ein Optimierungsproblem ist ein Approximations-Algorithmus, der für eine Eingabe des Problems und ein* $\epsilon(n)$ *eine Lösung mit relativem Fehler* $\epsilon(n)$ *konstruiert.*

Ein Approximationsschema heißt polynomielles Approximationsschema, falls für jedes feste $\epsilon > 0$ *diese Schema eine Lösung in Polynomzeit (bezuglich* n*) konstruiert, wobei* n *die Größe der Eingabe ist.*

2a. Das VERTEX-COVER-Problem

Eingabe: Ein ungerichteter Graph $G = (V, E)$.
Problem: Finde eine Knotenüberdeckung der minimalen Größe.

Algorithmus 8.1 APPROX-VERTEX-COVER(G);

$C := \emptyset$
$E' := E$
while $E' \neq \emptyset$
  do $C := C \cup \{u, v\}$ /* hier $(u, v) \in E'$ */
    $E' := E' - \{$Kanten von E' adjazent zu u oder $v\}$
return C

Theorem 8.1 *Der* APPROX-VERTEX-COVER-*Algorithmus hat die Güte 2.*

Beweisskizze: Die Knotenmenge C, die der Algorithmus konstruiert, ist eine Knotenüberdeckung.
Sei C^* eine minimale Knotenüberdeckung.

Sei A die Menge aller Kanten der Knotenüberdeckung nach Ausführung des Algorithmus. Dann gilt:

$$|C| = 2 \cdot |A|.$$

Da die Kanten von A unabhängig sind, haben wir:

$$|A| \leq |C^*|.$$

Somit gilt:

$$|C| \leq 2 \cdot |C^*|.$$

2b. Das TSP-Problem

Eingabe: Ein vollständiger Graph $G = (V, E)$ und eine Gewichtsfunktion $c : E \to \mathbf{R}^{\geq 0}$.
Problem: Finde eine Hamiltonische Tour mit minimalem Gewicht.

Für $A \subseteq E$ definieren wir:

$$c(A) = \sum_{(u,v)\in A} c(u, v).$$

Wir nehmen an, daß die Dreiecksungleichung

$$c(u, v) \leq c(u, w) + c(w, v)$$

für alle $u, v, w \in V$ gilt.

Bemerkung 8.2 *Sogar mit der Dreiecksungleichung ist das* TSP*-Problem NP-vollständig.*

<u>**Algorithmus** 8.2</u> $\textsc{Approx}$-$\mathrm{TSP}(G, c)$;

1. Wähle einen Knoten $v \in V$ aus.
2. Konstruiere einen minimalen Spannbaum T für G
   mit der Wurzel v
   (benutze den MST-$\textsc{Prim}$-Algorithmus).
3. Konstruiere eine Hilfstour W um T herum.
4. Konstruiere eine Hamiltonische Tour
   (als eine Veränderung von W).

Theorem 8.2 *Der* $\textsc{Approx-TSP}$*-Algorithmus hat die Güte 2 für das* TSP*-Problem mit der Dreiecksungleichung.*

Beweisskizze: Sei H^* eine minimale Tour.
Wir konstruieren eine Tour H, für die $c(H) \leq 2 \cdot c(H^*)$ gilt.

Da T ein minimaler Spannbaum ist, haben wir:

$$c(T) \leq c(H^*).$$

Wir konstruieren eine Liste L der Knoten, die der Ordnung der Knoten wie sei im $\text{MST-}\textsc{Prim}$-Algorithmus besucht wurden entspricht, und bekommen einen Spaziergang W über aller Knoten von T.

Da W jede Kante genau 2-mal durchläuft, gilt:

$$c(W) = 2 \cdot c(T),$$

woraus folgt:

$$c(W) \leq 2 \cdot c(H^*).$$

Die Tour W ist aber nicht unbedingt Hamiltonisch.

Wir gehen die Liste L der Knoten durch und nehmen aus W die Knoten, die schon besucht wurden, heraus.

Somit erhalten wir einen Hamiltonischen Weg H. Die Dreiecksungleichung garantiert

$$c(H) \leq c(W).$$

Somit gilt:

$$c(H) \leq 2 \cdot c(H^*).$$

Das Euclidean-TSP (ETSP):

Eingabe: Eine Punktmenge S in der Ebene.
Problem: Finde eine Hamiltonische Tour mit minimaler Länge.

Theorem 8.3 (Garey, Graham, Johnson '76)
Das ETSP-Problem ist NP-hard.

Theorem 8.4 (Arora '96)
Für jedes $\epsilon > 0$ existiert ein Algorithmus, der das ETSP-Problem mit der Güte $(1+\epsilon)$ in $n^{O(1/\epsilon)}$ Zeit löst.

Das Minimale Euklidische Matching (MEM):

Eingabe: $2n$ Punkten in der Ebene.
Problem: Finde n Segmente, so daß jeder Punkt von S zu genau einem Segment gehört und die totale Länge der Segmente minimal ist.

Theorem 8.5 (Edmonds '65, Gabow '72)
Das MEM-Problem in beliebigen Graphen ist in $O(n^3)$ Zeit lösbar.

Theorem 8.6 (Christofides '76)
Es existiert ein Algorithmus, der das ETSP-Problem in $O(n^3)$ Zeit mit der Güte 1.5 löst.

Beweisskizze.
Der Algorithmus besteht aus den folgenden Schritten:

1. Konstruiere ein EMST T.
2. Finde ein Matching M auf den Knoten von T mit ungeraden Knotengrad.
3. Der Graph $G = T \cup M$ ist Eulersch. Konstruiere einen eulerschen Kreis Ω in G.
4. Konstruiere einen hamiltonischen Kreis Φ aus Ω.

Sei Φ^* ein optimaler Kreis. Dann gilt analog wie oben:

$$|T| \leq |\Phi^*|.$$

Aus jedem Kreis Φ auf der Knotenmenge S können wir 2 Matchings konstruieren. Die totale Länge einer der Matchings ist höchstens $|\Phi^*|/2$.

Somit haben wir:

$$\begin{aligned} |\Phi| &\leq |T| + |M| \\ &\leq |\Phi^*| + |\Phi^*|/2 \\ &= 1.5 \cdot |\Phi^*|. \end{aligned}$$

Das TSP-Problem für allgemeine Gewichtfunktion c ist allerdings schwieriger lösbar.

Theorem 8.7 *Sei $p \geq 1$. Falls $P \neq NP$ gilt, gibt es keinen Approximations-Algorithmus, der das TSP-Problem in Polynomzeit mit Güte p löst.*

Beweisskizze: O.B.d.A. sei $p \in I\!N$.
Angenommen für ein $p \geq 1$ gibt es einen polynomiellen Approximations-Algorithmus A.

Wir zeigen, wie man den Algorithmus A benutzen kann, um das HC-Problem in Polynomzeit zu lösen.

Sei $G = (V, E)$ eine Eingabe für das HC-Problem. Wir konstruieren einen vollstandigen Graphen $G' = (V, E')$. Sei:

$$c(u, v) = \begin{cases} 1, & \text{falls } (u, v) \in E \\ p|V| + 1, & \text{sonst} \end{cases} .$$

G ist Hamiltonisch $\Rightarrow$ G' enthält eine Tour mit Gewicht $|V|$.
G ist nicht Hamiltonisch $\Rightarrow$ G' enthält eine Tour mit Gewicht

$$\geq (p|V| + 1) + (|V| - 1) > p|V|.$$

Wir wenden A auf die Eingabe (G', c) an. Dann ergibt A eine Tour, die nicht p-mal länger als die optimale Tour ist.

Somit gilt:
G ist Hamiltonisch $\Rightarrow$ A konstruiert einen Ham. Kreis in G.
G ist nicht Hamiltonisch $\Rightarrow$ A konstruiert eine langere Tour in G'.

Da A alles in Polynomzeit durchläuft, gilt P=NP. $\square$

2c. Scheduling

Seien $J_1, ..., J_n$ Jobs und $M_1, ..., M_m$ identische Prozessoren.

Annahmen:

- jeder Job J_j benötigt die Zeit $p_j > 0$ und soll nicht unterbrochen werden
- jeder Prozessor M_i kann gleichzeitig nur einen Prozess bearbeiten.

Das Problem: konstruiere ein Schedule

$$\Sigma : \{J_j\}_{j=1}^n \mapsto \{M_i\}_{i=1}^m$$

das die schnellste Abarbeitung aller Jobs garantiert.

Theorem 8.8 (Graham '66). *Es existiert ein Algorithmus mit der Güte* $(2 - 1/m)$.

Beweisskizze:
Seien die Jobs in einer Liste in beliebiger Ordnung gespeichert.

Heuristik G: wir nehmen den nächsten Job aus der Liste, sobald als irgendein Prozessor frei ist.

Wir bezeichnen durch s_j und e_j die Start- und Endzeit des Jobs J_j bei der Heuristik G. Sei J_k der Job, der als letztes fertig wird. Dann ist vor dem Zeitpunkt s_k kein Prozessor frei.

Für die Laufzeit C^*_{max} des optimalen Schedules gilt:

$$\begin{aligned} C^*_{\mathsf{max}} &\geq p_k \\ C^*_{\mathsf{max}} &\geq \frac{1}{m} \cdot \sum_{j=1}^{n} p_j. \end{aligned}$$

Die Heuristik G liefert:

$$\begin{aligned} C^G_{\mathsf{max}} = e_k &= s_k + p_k \leq \frac{1}{m} \cdot \sum_{j \neq k} p_j + p_k \\ &= \frac{1}{m} \cdot \sum_{j=1}^{n} p_j + \left(1 - \frac{1}{m}\right) p_k \\ &\leq C^*_{\mathsf{max}} + \left(1 - \frac{1}{m}\right) C^*_{\mathsf{max}} = \left(2 - \frac{1}{m}\right) C^*_{\mathsf{max}}. \end{aligned}$$

Die LPT Regel: (Longest Processing Time)
Sortiere die Jobs bzgl. p_i in nichtsteigender Ordnung.

Theorem 8.9 *Es gilt:*

$$C_{max}^{LPT} \leq (4/3 - 1/3m)C_{max}^*.$$

Beweisskizze:
Sei J_k der "letzte" Job in dem LPT Schedule.

Angenommen, $p_k \leq C_{\max}^*/3$. Dann gilt:

$$\begin{aligned} C_{\max}^{\mathrm{LPT}} &\leq (1/m) \sum_{j=1}^{n} p_j + (1 - 1/m)p_k \\ &\leq C_{\max}^* + (1 - 1/m)C_{\max}^*/3 \\ &= (4/3 - 1/3m)C_{\max}^*. \end{aligned}$$

Sei $p_k > C_{\max}^*/3$. O.B.d.A. sei J_k der letzte Job in der Liste.

$\Rightarrow$ jeder Prozessor erhält höchstens 2 Jobs.

Somit haben wir: $n = 2m - h$ mit $h > 0$. In diesem Fall konstruiert die LPR Heuristik ein optimales Schedule. $\square$

2d. Das Überdeckungs-Problem

Eingabe: Eine endliche Menge X und eine Menge F von Teilmengen von X, so daß $\bigcup\limits_{S \in F} S = X$ gilt.
Problem: Finde die minimale Menge $C \subseteq F$, die X überdeckt.

Bemerkung 8.3 *Das* Überdeckungs*-Problem ist in NPC (Reduzierung aus* VC*-Problem).*

Algorithmus 8.3 $\textsc{Greedy-Set-Cover}(X, F)$;

$U := X$
$C := \emptyset$
while $U \neq \emptyset$
  do Wähle $S \in F$ mit $|S \cap U| \to \max$ aus
    $U := U - S$
    $C := C \cup \{S\}$
return C

Da die **while** -Schleife höchstens $\min\{|X|, |F|\}$-mal durchlaufen wird und jeweils höchstens $O(|X| \cdot |F|)$ Berechnungen benötigt, ist die Laufzeit des $\textsc{Greedy-Set-Cover}$- Algorithmus $O(|X| \cdot |F| \cdot \min\{|X|, |F|\})$.

Theorem 8.10 *Der* GREEDY-SET-COVER-*Algorithmus hat die Güte* $H(\max\{|S| \mid S \in F\})$,
wobei $H(d) = \sum_{i=1}^{d}(1/i)$ *ist.*

Beweisskizze: Sei C die Überdeckung, die der GREEDY-SET-COVER-Algorithmus konstruiert, und sei C^* eine minimale Überdeckung.

Sei S_i die Menge, die der Algorithmus in der i-ten **while** -Schleife auswählt. Weiterhin sei $x \in X$ zum erstenmal durch S_i überdeckt. Wir setzen das Gewicht c_x von x wie folgt:

$$c_x = \frac{1}{|S_i - (S_1 \cup \cdots \cup S_{i-1})|}.$$

Dann gilt:

$$|C| = \sum_{x \in X} c_x \leq \sum_{S \in C^*} \sum_{x \in S} c_x. \tag{1}$$

Wir zeigen später, daß

$$\sum_{x \in S} c_x \leq H(|S|) \tag{2}$$

gilt. Aus (1) und (2) folgt:

$$|C| \leq \sum_{S \in C^*} H(|S|) \leq |C^*| \cdot H(\max\{|S| \mid S \in F\}),$$

womit der Beweis beendet ist.

Um (2) zu zeigen, definieren wir für $S \subseteq F$ und $i \leq |C|$:

$$u_i = |S - (S_1 \cup \cdots \cup S_i)|,$$

d.h. die Anzahl der Elemente von S, die nicht durch $S_1, \ldots, S_i$ überdeckt sind.

Sei S fest, $u_0 = |S|$ und k der minimale Index, so daß $u_k = 0$ gilt. Dann gilt $u_{i-1} \geq u_i$ und $u_{i-1} - u_i$ Elemente von S sind zum erstenmal durch S_i überdeckt für $i = 1, ..., k$.

Somit gilt:

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup \cdots \cup S_{i-1})|}.$$

Da

$$|S_i - (S_1 \cup \cdots \cup S_{i-1})| \geq |S - (S_1 \cup \cdots \cup S_{i-1})|$$

wegen der Greedy-Auswahl von S_i gilt, haben wir:

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}}.$$

Da für ganze Zahlen a, b mit $a < b$ gilt:

$$H(b) - H(a) = \sum_{i=a+1}^{b} (1/i) \geq (b - a) \cdot (1/b),$$

erhalten wir eine teleskopische Summe:

$$\begin{aligned}
\sum_{x \in S} c_x &\leq \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) \\
&= H(u_0) - H(u_k) = H(u_0) - H(0) \\
&= H(u_0) = H(|S|).
\end{aligned}$$

und der Beweis ist beendet. □

Korollar 8.1 *Der* GREEDY-SET-COVER-*Algorithmus hat die Güte* $(\ln |X| + 1)$.

2e. Das MAXIMUM SET-COVER-Problem

Eingabe: Eine endliche Menge X, eine Gewichtsfunktion $w : X \mapsto \mathbf{R}$, eine Menge F von Teilmengen von X und $k \in I\!N$.
Problem: Finde eine Menge $C \subseteq F$ mit $|C| = k$, so daß $\sum_{x \in C} w(x)$ maximiert wird.

<u>**Algorithmus** 8.4</u> MAXIMUM-COVER(X, F, w);

$U := X$
$C := \emptyset$
for $i := 1$ **to** k **do**
  Wähle $S \in F$ mit $w(S \cap U) \to \max$
  $U := U - S$
  $C := C \cup S$
return C

Theorem 8.11 *Der* MAXIMUM-COVER-*Algorithmus hat die Güte* $(1 - 1/e)^{-1}$.

Beweisskizze:
Sei C^G die Menge, die die Heuristik liefert und sei C^* die optimale Lösung. Weiterhin sei S_i die Menge, die in Schritt i ausgewählt wird.

Die Greedy-Methode liefert:

$$w(\bigcup_{i=1}^{l} S_i) - w(\bigcup_{i=1}^{l-1} S_i) \geq \frac{w(C^*) - w(\cup_{i=1}^{l-1} S_i)}{k}, \quad l = 1, ..., k.$$

Wir zeigen per Induktion über l:

$$w(C^G) \geq \left(1 - (1 - 1/k)^l\right) \cdot w(C^*).$$

Für $l = 1$ haben wir: $w(S_1) \geq w(C^*)/k$.

Für $l \geq 1$ gilt:

$$\begin{aligned}
w(\bigcup_{i=1}^{l+1} S_i) &= w(\bigcup_{i=1}^{l} S_i) + w(\bigcup_{i=1}^{l+1} S_i) - w(\bigcup_{i=1}^{l} S_i) \\
&\geq w(\bigcup_{i=1}^{l} S_i) + \frac{w(C^*) - w(\cup_{i=1}^{l} S_i)}{k} \\
&= (1 - 1/k) \cdot w(\bigcup_{i=1}^{l} S_i) + w(C^*)/k \\
&\geq \left(1 - \frac{1}{k}\right)\left(1 - (1 - \frac{1}{k})^l\right) \cdot w(C^*) + w(C^*)/k \\
&= \left(1 - (1 - (1/k)^{l+1}\right) \cdot w(C^*).
\end{aligned}$$

Für $l = k$ erhalten wir:

$$w(C^G) \geq \left(1 - (1 - 1/k)^k\right) \cdot w(C^*) > (1 - 1/e) \cdot w(C^*).$$

Bemerkung 8.4 *Manchmal ist die Auswahl der Menge S ein schwieriges Problem. Falls allerdings die Auswahl nur im Faktor β ($\beta < 1$) schlechter als das maximale S, dann liefert der Algorithmus die Güte* $(1 - 1/e^\beta)^{-1}$.

2f. Das INDEPENDENT-SET Problem

Eingabe: Ein ungerichteter Graph $G = (V, E)$.
Problem: Finde eine unabhängige Menge maximaler Größe.

Für $v \in V$ und $n = |V|$ definieren wir $\delta = \frac{1}{n} \sum_{v \in V} \mathrm{deg}(v)$ und

$$N(v) = \{u \in V \mid \mathrm{dist}(u, v) = 1\}.$$

Algorithmus 8.5 INDEPENDENT-SET(G);
$S := \emptyset$
while $V(G) \neq \emptyset$ **do**
  Finde $v \in V$ mit $\mathrm{deg}(v) = \min_{u \in V} \mathrm{deg}(u)$
  $S := S \cup \{v\}$
  $G := G - (v \cup N(v))$
return S

Theorem 8.12 *Der* INDEPENDENT-SET-*Algorithmus liefert eine unabhängige Menge* S *mit* $q = |S| \geq n/(\delta + 1)$.

Beweisskizze: Wir betrachten v_i, die in Schritt i benutzt werden. Sei $d_i = \mathrm{deg}(v_i)$. Dann gilt: $\sum_{i=1}^{q} (d_i + 1) = n$,

$$\sum_{i=1}^{q} d_i(d_i + 1) \leq n\delta.$$

Somit erhalten wir:

$$(\delta + 1)n \geq \sum_{i=1}^{q} (d_i + 1)^2 \geq \frac{n^2}{q}$$

$\Rightarrow q \geq \frac{n}{\delta+1}$. $\square$

2g. Das SUBSET-SUM-Problem

Entscheidungsproblem:
Eingabe: Eine Menge $S = \{x_1, ..., x_n\}$ von ganzen Zahlen, $t \in I\!N$.
Frage: Gibt es eine Teilmenge $I \subseteq \{1, ..., n\}$ mit $\sum\limits_{i \in I} x_i = t$?

Optimierungproblem:
Eingabe: Eine Menge $S = \{x_1, ..., x_n\}$ von ganzen Zahlen, $t \in I\!N$.
Problem: Finde eine Teilmenge $I \subseteq \{1, ..., n\}$ mit $\sum\limits_{i \in I} x_i \leq t$ und $\sum\limits_{i \in I} x_i$ maximal.

Sei $A \subseteq S$. Wir definieren

$$A + s = \{a + s \mid a \in A\}.$$

Algorithmus 8.6 EXACT-SUBSET-SUM(S, t);

$n := |S|$
$L_0 := \langle 0 \rangle$
for $i = 1$ **to** n
  do $L_i :=$ MERGE-LISTS$(L_{i-1}, L_{i-1} + x_i)$
    $L_i := L_i - \{x \in L_i \mid x > t\}$
return Das maximal Element von L_n

Sei P_i die Menge aller partitieler Summen von $\{x_1, ..., x_i\}$.

$$P_i = P_{i-1} \cup (P_{i-1} + x_i).$$

Per Induktion über i ist L_i die sortierte Menge

$$\{x \in P_i \mid x \leq t\}.$$

Polynomielles Approximationsschema

Sei $L = \langle y_1, ..., y_m \rangle$ eine sortierte Liste und $0 < \delta < 1$. Wir konstruieren eine Liste $L' \subseteq L$, so daß:

$\forall y \in L \;\; \exists z \in L'$ mit $\dfrac{y-z}{y} \leq \delta$ (d.h. $(1-\delta)y \leq z \leq y$),

und $|L'|$ ist minimal.

Algorithmus 8.7 $\textsc{Trim}(L, \delta)$;

$m := |L|$
$L' := \langle y_1 \rangle$
$last := y_1$
for $i = 2$ **to** m
  do **if** $last < (1-\delta)y_i$
    then $\textsc{Append}(L', y_i)$
      $last := y_i$
return L'

Algorithmus 8.8 $\textsc{Approx-Subset-Sum}(S, t, \epsilon)$;

$n := |S|$
$L_0 := \langle 0 \rangle$
for $i = 1$ **to** n
  do $L_i := \textsc{Merge-Lists}(L_{i-1}, L_{i-1} + x_i)$
    $L_i := \textsc{Trim}(L_i, \epsilon/n)$
    $L_i := L_i - \{x \in L_i \mid x > t\}$
return Das maximal Element von L_n

Theorem 8.13 APPROX-SUBSET-SUM *ist ein polynomielles Approximationsschema für* SUBSET-SUM-*Problem.*

Beweisskizze:
Der Algorithmus ergibt einen Wert z, der die Summe von einer Teilmenge von S ist. Wir zeigen, daß z nicht $(1-\epsilon)$-mal kleiner als die optimale Lösung y^* ist.

Per Induktion über i:

$$\forall y \in P_i \ \text{ mit } \ y \le t \quad \exists z \in L_i \ \text{ mit } \quad (1-\epsilon/n)^i y \le z \le y.$$

Sei $y^* \in P_n$ die optimale Lösung. Dann $\exists z \in L_n$ mit

$$(1-\epsilon/n)^n\, y^* \le z \le y^*.$$

Das größtmögliche z ist die Ausgabe von dem Algorithmus. Da die Funktion $(1-\epsilon/n)^n$ monoton mit n steigt, gilt:

$$1-\epsilon \le (1-\epsilon/n)^n \qquad \Longrightarrow \qquad (1-\epsilon)\, y^* \le z.$$

Schließlich, zeigen wir, daß der APPROX-SUBSET-SUM-Algorithmus ein polynomieller Algorithmus ist. Dazu muß nur die Länge von L_i abschätzen werden.

Nach der Anwendung von TRIM-Prozedur für zwei auf einander folgende Elemente $z, z' \in L_i$ gilt:

$$\frac{z}{z'} \ge \frac{1}{1-\epsilon/n},$$

bekommen wir:

$$|L_i| \le \log_{1/(1-\epsilon/n)} t \le n \ln t/\epsilon.$$

Diese Schranke ist polynomiell bezüglich n und $1/\epsilon$. $\square$

2h. 3-Färbung

Theorem 8.14 *Sei G ein Graph mit $\chi(G) \leq 3$. Es existiert ein polynomieller Algorithmus, der G mit $O(\sqrt{n})$ Farben färbt.*

Beweisskizze: Wir benutzen 2 Hilfssätze:

- Falls $\chi(G) = 2$ (d.h. G ist bipartit), dann kann G in Polynomzeit mit 2 Farben gefärbt werden.
- Sei G ein Graphi mit maximalem Knotengrad Δ. Dann kann G in Polynomzeit mit $\Delta + 1$ Farben gefärbt werden. (Greedy-Methode)

O.B.d.A. nehmen wir an, daß $\chi(G) = 3$ und $\Delta(G) \geq \sqrt{n}$ gilt.

Sei für $v \in V(G)$: $\quad N(v) = \{u \in V \mid \mathrm{dist}_G(u, v) = 1\}$.
$\chi(G) = 3 \Rightarrow$ Der knoteninduzierte Teilgraph $G[N(v)]$ ist bipartit $\forall v \in V$ and in Polynomzeit 2-färbbar.
$\Rightarrow$ der knoteninduzierte Teilgraph $G[v \cup N(v)]$ ist in Polynomzeit 3-färbbar.

Algorithmus 8.9 3-Färbbung;
while $\Delta(G) \geq \sqrt{n}$ **do**
  Finde $v \in V(G)$ mit $\deg(v) \geq \sqrt{n}$
  Färbe $G[v \cup N(v)]$ mit 3 Farben
  $G := G - (v \cup N(v))$
Färbe G mit $\Delta(G) + 1$ Farben.

Offensichtlich ist die Laufzeit polynomiell und die Anzahl der Farben $O(\sqrt{n})$.

9 Lektion 9: Einführung in randomisierte Algorithmen

Beispiel:
Eingabe: Die reellen Zahlen $x_1, ..., x_n$.
Problem: Finde x_i mit $x_i \geq \overline{x}$, wobei $\overline{x}$ das $\lfloor n/2 \rfloor$-kleinste Element ist.

Ein deterministischer Algorithmus benötigt $\Theta(n)$ Schritte.

Seien x_i und x_j mit $i \neq j$ zufällig ausgewählt. Weiterhin sei $x_i \geq x_j$. Dann gilt:

$$\mathbf{P}[x_i \geq \overline{x}] \geq 1/2 \qquad \Longleftrightarrow \qquad \mathbf{P}[x_i \leq \overline{x}] \leq 1/2.$$

$$\Downarrow$$

$$\mathbf{P}[(x_i \leq \overline{x}) \text{ und } (x_j \leq \overline{x})] \leq 1/4.$$

Da $x_i \geq x_j$ haben wir:

$$\mathbf{P}[(x_i \leq \overline{x}) \text{ und } (x_j \leq \overline{x})] = \mathbf{P}[x_i \leq \overline{x}] \leq 1/4.$$

Analog, falls wir k Elemente $x_{i_1}, ..., x_{i_k}$ auswählen, dann gilt für $\tilde{x} = \max\{x_{i_1}, ..., x_{i_k}\}$:

$$\mathbf{P}[\tilde{x} \leq \overline{x}] \leq 2^{-k},$$

und somit ist die Wahrscheinlichkeit dafür, daß $x_i \geq \overline{x}$ groß.

Gilt $k = 10 \Rightarrow \mathbf{P}[x_i \geq \overline{x}] \geq 0.999$.
Gilt $k = 20 \Rightarrow \mathbf{P}[x_i \geq \overline{x}] \geq 0.999999$.

Die Laufzeit von unserer Methode ist aber von n unabhängig !

1. Das Max-Sat Problem:

Eingabe: Ein Formel $f(x_1, ..., x_n) = C_1 \wedge \cdots \wedge C_m$ in KNF.
Problem: Finde eine Abbildung $\{x_1, ..., x_n\} \mapsto \{T, F\}$, so daß die Anzahl der Klauseln, die erfüllt sind, maximiert wird.

Sei M^* die maximale Anzahl der erfüllten Klauseln. Wir konstruieren ein Approximationsalgorithmus mit der Güte 3/4.

Methode 1:
Setze $x_i = T$ für $i = 1, ..., n$ unabhängig von einander mit Wahrscheinlichkeit 1/2.
$\Rightarrow$ eine Klausel mit k Literalen ist nicht erfüllt mit W. $1/2^k$.

Sei n_1 die Anzahl der Klauseln, die mit dieser Methode erfüllt sind. Es gilt: $n_1 \geq (3/4)M^*$, falls jede Klausel mindestens 2 Literale enthält.

Methode 2:
Wir formulieren das Max-Sat Problem als ein IP:

- Für jede Klausel C_i setzen wir eine Indikator-Variable $z_i = 1$ falls C_i erfüllt ist und $z_i = 0$ sonst.
- Für jede Variable x_i setzen wir eine Indikator-Variable y_i mit $y_i = 1 \Leftrightarrow x_i = T$.

Sei S_i^+ (bzw. S_i^-) die Menge aller Variablen in C_i, die nicht negiert sind (bzw. negiert sind).

$$\begin{array}{rl} \text{Maximiere} & \sum\limits_{j=1}^{m} z_j \\ \text{mit} & \sum\limits_{S_j^+} y_i + \sum\limits_{S_j^-} (1-y_i) \geq z_j, \quad j = 1, ..., m \\ & y_i, z_j \in \{0,1\}, \qquad i = 1, ..., n \ j = 1, ..., m. \end{array}$$

Wir ersetzen die Bedingung $y_i, z_j \in \{0,1\}$ mit $y_i, z_j \in [0,1]$ und bekommen ein LP.

Sei $\hat{y}_i$ und $\hat{z}_i$ die Lösung des LPs. Offensichtlich gilt:

$$M^* \leq \sum_{j=1}^{m} \hat{z}_j.$$

Sei $\beta_k = 1 - \left(1 - \frac{1}{k}\right)^k$.

Lemma 9.1 *Sei C_j eine Klausel mit k Literalen. Dann gilt:*

$$\mathbf{Pr}[C_j \text{ ist erfüllt}] \geq \beta_k \hat{z}_j.$$

Beweisskizze: O.B.d.A. nehmen wir an, daß alle Variablen in C_j nicht negiert sind, d.h. $C_j = x_1 \vee \cdots \vee x_k$.

Da die LP-Beschränkungen erfüllt sind, gilt

$$\hat{y}_1 + \cdots + \hat{y}_k \geq \hat{z}_j.$$

Somit haben wir:

$$\begin{aligned} \mathbf{Pr}[C_j \text{ ist erfüllt}] &= 1 - \prod_{i=1}^{k} (1 - \hat{y}_i) \geq 1 - \prod_{i=1}^{k} (1 - \hat{z}_j/k) \\ &= 1 - (1 - \hat{z}_j/k)^k \geq \beta_k \hat{z}_j. \end{aligned}$$

Sei C^k die Menge aller Klauseln, die aus k Literale bestehen.

$$\begin{aligned} n_2 &= \mathbf{Ex}[|\{C_j \in C^k \mid C_j \text{ ist erfüllt}\}|] \\ &= \sum_{C_j \in C^k} \mathbf{Pr}[C_j \text{ ist erfüllt}] \geq \beta_k \sum_{C_j \in C^k} \hat{z}_j. \end{aligned}$$

Algorithmus 9.1 MAX-SAT;

1. Wende die Methode 1 an und bestimme n_1.
2. Wende die Methode 2 an und bestimme n_2.
3. Wähle die beste Lösung aus.

Theorem 9.1 *Es gilt:*

$$\max\{n_1, n_2\} \geq \frac{3}{4} \sum_{j=1}^{m} \hat{z}_j.$$

Beweisskizze:
Sei $\alpha_k = 1 - 1/2^k$. Dann gilt:

$$\begin{aligned} n_1 &= \sum_{k \geq 1} \sum_{C_j \in C^k} \alpha_k \geq \sum_{k \geq 1} \sum_{C_j \in C^k} \alpha_k \hat{z}_j, \\ n_2 &\geq \sum_{k \geq 1} \sum_{C_j \in C^k} \beta_k \hat{z}_j. \end{aligned}$$

Für $k \geq 1$ gilt:

$$\alpha_k + \beta_k = 1 - \frac{1}{2^k} + 1 - \left(1 - \frac{1}{k}\right)^k \geq 3/2.$$

Somit haben wir:

$$\max\{n_1, n_2\} \geq \frac{n_1 + n_2}{2} \geq \sum_{k \geq 1} \sum_{C_j \in C^k} \frac{\alpha_k + \beta_k}{2} \hat{z}_j \geq \frac{3}{4} \sum_{j=1}^{m} \hat{z}_j.$$

2. Las-Vegas-Algorithmen

Der Algorithmus, den wir soeben untersucht haben, ist ein Beispiel für einem Monte-Carlo-Algorithmus. Solche Algorithmen liefern mit ziemlich kleiner Wahrscheinlichkeit eine falsche Lösung, aber sie haben meistens kürzere Laufzeiten im Vergleich zu deterministischen Algorithmen.

Andererseits, gibt es Algorithmen, die ganz sicher eine richtige Lösung konstruieren, aber man kann ihre Laufzeit nur durchschnitlich abschätzen.

Set-Coloring-Problem:
Eingabe: Eine Menge S mit $|S| = n$ und Teilmengen $F = \{S_i\}$, $S_i \subseteq S$, $|S_i| = r$, $i = 1, ..., k$. Es gilt: $k \leq 2^{r-2}$.
Problem: Färbe jedes Element $x \in S$ entweder rot oder blau, so daß jede Teilmenge $S_i \in F$ Elemente von beiden Farben enthält.

Algorithmus 9.2 2-FÄRBUNG(S, F);

1. Färbe jedes Element von S zufällig und unabhängig voneinander rot oder blau mit einer Wahrscheinlichkeit von 1/2.
2. Wiederhole den Schritt 1 bis eine richtige Färbung konstruiert wird.

Wie hoch ist die Wahrscheinlichkeit, daß der Schritt 1 des 2-FÄRBUNG-Algorithmus eine falsche Färbung konstruiert ?

$$\mathbf{Pr}[\text{alle Elemente von } S_i \text{ sind rot}] = 2^{-r}.$$
$$\Downarrow$$
$$\mathbf{Pr}[\exists \text{ eine “rote” Teilmenge } S_i \in F] \le k\, 2^{-r} \le 1/4.$$
$$\Downarrow$$
$$\mathbf{Pr}[\text{die 2-Färbung ist falsch}] \le 1/2.$$

$\Rightarrow$ eine richtige 2-Färbung existiert, sonst ist die letzte Wahrscheinlichkeit 1.

Unserer Algorithmus ist ein Las-Vegas-Algorithmus, weil er solange eine Färbung von S konstruiert, bis diese Färbung eine richtige 2-Färbung ist.

$\Rightarrow$ die erwartete Anzahl vor Wiederholungen des Schrittes 1 ist 2.