Algorithmen für Fortgeschrittene

Dozent: Prof. Dr. Helmut Alt

Sommersemester 2004

Inhaltsverzeichnis

Scheinkriterien:

- abwechselndes Mitschreiben des Vorlesungsskripts
- Übungszettel (näheres dazu beim nächsten Termin)

Literatur:

Cormen, Leiserson, Rivest, Stein:
„Introduction to Algorithms“

Tutorien:

Di, 14 – 16 Uhr
Mi, 8 – 10 Uhr
beide bei Claudia

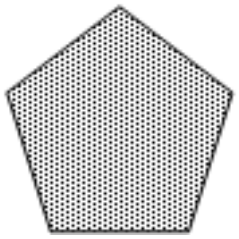
Inhalt der Vorlesung (Überblick):

- Netzwerkfluss: Matching in Graphen
- Stringmatching
- Approximationsalgorithmen für „schwere“ Probleme
  - Beispiel: TSP
    (max. bis 3000 Knoten berechenbar, danach Approximation)
- Algorithmen für algebraische/zahlentheoretische Probleme
  - FFT schnelle Fourier-Transformation
  - Primzahltest (naiv) $n = logN\mathcal{O}(\sqrt{N}) = \mathcal{O}(2^{1/2n}) \approx 1,4^n$
    braucht exponentielle Zeit, Algorithmus für polynomielle Zeit schon gefunden
- Faktorisierung RSA-Kryptosystem
- Lineare Optimierung

  Beispiel:

  gegeben: Funktion $f(x_1, \ldots, x_n) = a_1x_1 + \ldots + a_nx_n$
  Bedingungen $b_1x_1 + b_2x_2 + \ldots + b_nx_n \leq c$

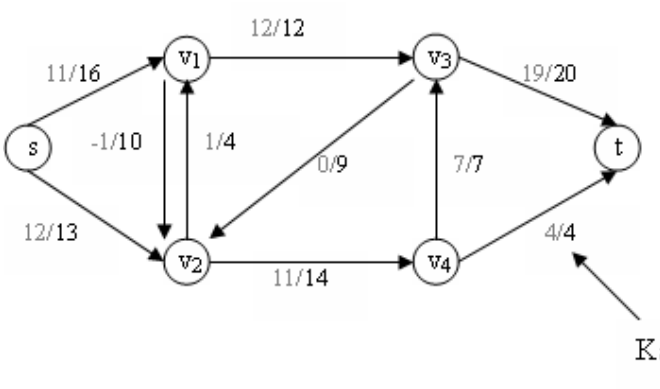
  Spannt Polynom auf:



  - Lösungsalgorithmus: Simplex

1 Flussprobleme und Matching in Graphen

Beispiel: Flussproblem



Anwendungsbeispiele:

- Röhrensystem für Flüssigkeiten
- Verkehrsnetz
- Information in Kommunikationsnetzen

formale Definition:

Netz besteht aus:

- gerichteter Graph $G = (V, E)$
- Kapazitätsfunktion $c : E \to \mathbb{R}_{\geq 0}$
- Quelle $s \in V$, Senke $t \in V$

Problem: Finde maximalen Fluss von s nach t

Definition:

Fluss ist eine Funktion $f : VxV \to \mathbb{R}$ mit

Vereinbarung:
$c(u, v) := 0$ falls $(u, v) \notin E$

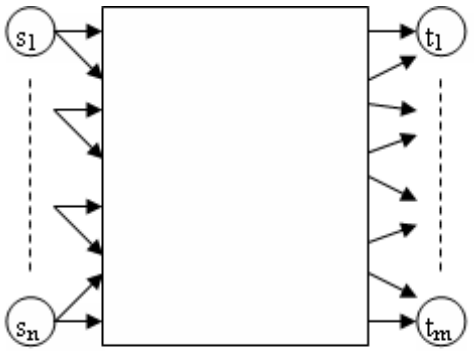
1. $f(u, v) \leq c(u, v)$ für alle $u, v \in V$
2. $f(u, v) = -f(v, u)$
3. $\sum_{v \in V} f(u, v) = 0$ für alle $u \in V \backslash \{s, t\}$

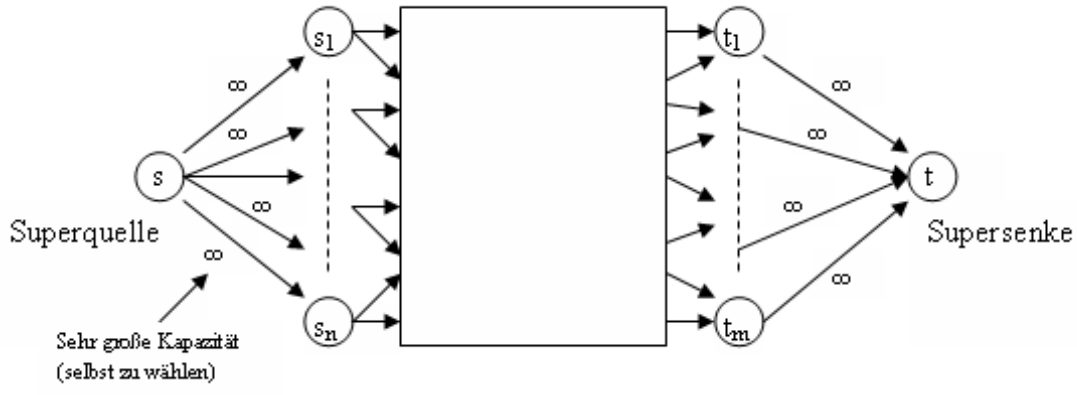
Wert des Flusses: $|f| = \sum_{v \in V} f(s, v)$ (Fluss aus der Quelle)

Maximaler Fluss: für ein gegebenes Netz (G, c):
ein Fluss mit maximalem Wert
Zwischen Knoten u, v kann Fluss nur in einer Richtung positiv sein. Nur dieser wird angezeigt.

Mehrere Quellen und Senken

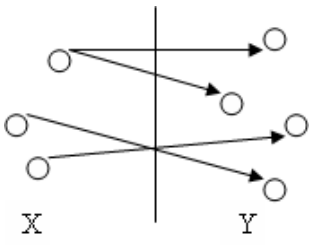
Problem ist auf eine Quelle und eine Senke zurückführbar





Wert des Flusses: $|f| = \sum_{i=1}^{n} \sum_{v \in V} f(s_i, v)$ (im neuen Netzwerk)

Notation: X, Y Menge von Knoten $f(X,Y) := \sum_{x \in X} \sum_{y \in Y} f(x,y)$



Es gilt:

Lemma 1:

a) $f(X,X) = 0 \quad \forall\, X \subset V$

b) $f(X,Y) = -f(Y,X) \quad \forall\, X,Y \subset V$

c) $f(X \cup Y, Z) = f(X,Z) + f(Y,Z) \quad \forall\, X,Y,Z \subset V$

d) $f(Z, X \cup Y) = f(Z,X) + f(Z,Y) \quad mit\ X \cap Y = \emptyset$

Beweis:

leicht (nach obigen Definitionen)

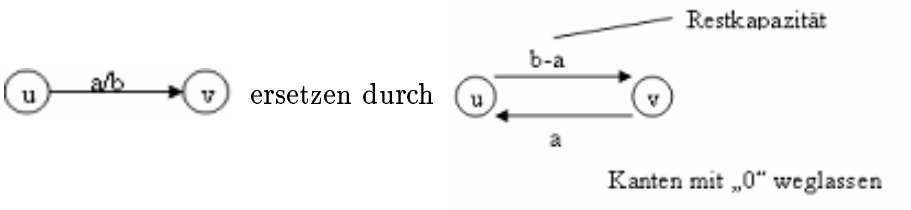
1.1 Ford-Fulkerson-Methode (Zur Bestimmung des maximalen Flusses:)

„Augmentierender Weg“ (augmenting Path) ist für gegebenen Fluss f ein Weg p in G, so dass für jede Kante e auf p gilt $f(e) < c(e)$, der von s nach t führt.

Idee des Algorithmus:

- fange an mit Fluss $f = 0$
- solange es einen augmentierenden Weg p gibt: erhöhe Fluss entlang p.

Restnetz: bei gegebenen Fluss f

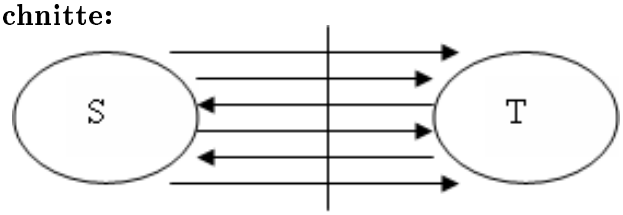


Ergibt Netz G_f mit Kapazitäten c_f

Definition:

Ein *augmentierender Weg* für ein Netz G, c mit Fluss f ist ein einfacher Weg von s nach t in G_f.
Kapazität des augmentierenden Weges: $c_f(p) = min\{c_f(e) | e \text{ Kante auf } p\}$
Entlang p kann der Fluss um $c_f(p)$ erhöht werden.

Schnitte:



Definition:

Schnitt ist Partition $V = S \cup T$ mit $s \in S, t \in T$
$c(S,T)$ Kapazität des Schnitts
$f(S,T)$ Fluss über Schnitt bei geg. Fluss f

Es gilt:

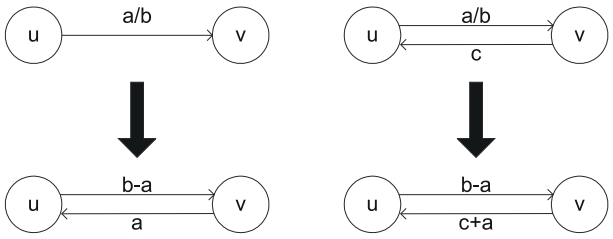
Lemma 2:

$f(S,T) = |f|$ für jeden Fluss f und jeden Schnitt S,T

Beweis:

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) \quad \textit{(Lemma 1)} \\ &= f(\{s\},V) + f(S \setminus \{s\},V) \\ &= f(\{s\},V) \\ &= |f| \end{aligned}$$

Restnetzbildung:



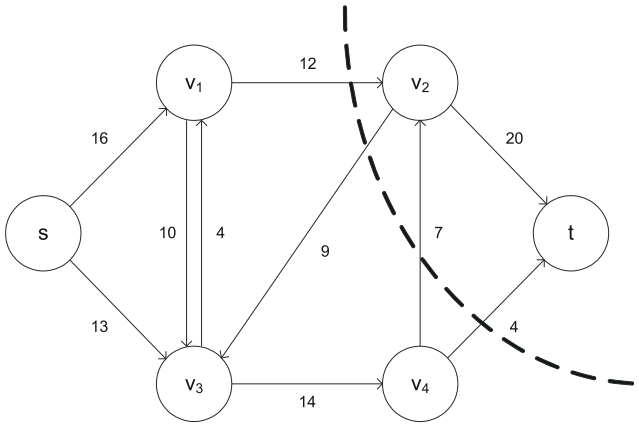
Kanten mit der Kapazität 0 können weggelassen werden.

Korollar:

Der Wert jedes Flusses im Netz G ist durch die minimale Kapazität aller denkbarer Schnitte nach oben beschränkt, denn

$$f(S,T) \leq c(s,t).$$

Im Beispiel kann der Fluss nicht größer sein als 23 (Schnittverlauf laut Skizze):



Satz 1:

Sei f Fluss im Netz G, c, dann sind folgende Aussagen äquivalent:

1. f ist maximal.
2. Es gibt keine augmentierenden Wege bzgl. G, c, f.
3. Es gibt einen Schnitt S, T mit $|f| = c(S,T)$.
   (Bemerkung: Dieser Schnitt hat minimale Kapazität.)

Beweis:

- $1 \Rightarrow 2$:

  trivial

- $2 \Rightarrow 3$:

  Es gibt in G keinen augmentierenden Weg, d.h. in G_f gibt es keinen Weg von s nach t.

  Sei $S = \{v \in V \mid \exists \text{ Weg von } s \text{ nach } v \text{ in } G_f\}$ und $T = V \backslash S$.

  Betrachte Schnitt S, T, bei dem $f(u,v) = c(u,v)$ für alle $u \in S$ und $v \in T$.

  Daraus folgt:

$$\begin{aligned} |f| &= f(S,T) && \text{(nach Lemma 2)} \\ &= c(S,T) \end{aligned}$$

- $3 \Rightarrow 1$:

  Nach dem Korollar gilt $|f| \leq c(S,T)$ für alle Flüsse und Schnitte, also auch für diesen.

  Da $|f| = c(S,T)$ gilt, ist f maximal.

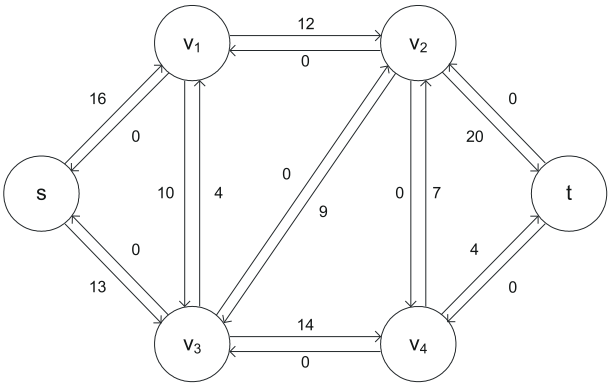
□

1.2 Ford-Fulkerson-Algorithmus (Schema)

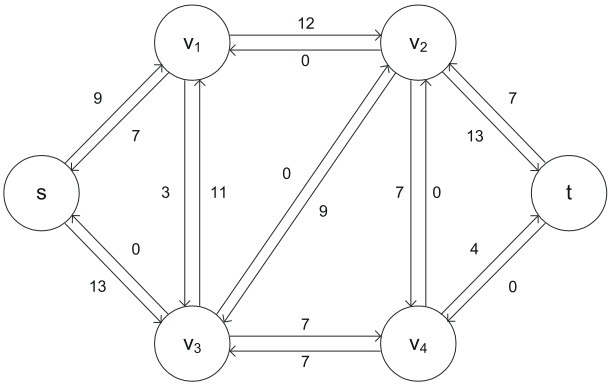
1: Initialisiere den Fluss f auf 0.
2: **while** $\exists$ augmentierender Weg p von s nach t im Restnetz G_f **do**
3: $\forall$ Kante $e \in p$ erhöhe den Fluss f um die Kapazität $c_f(p)$ dieses Weges, wobei $c_f(p) = \min c_f(e)$
4: **end while**

In unserem Beispiel:

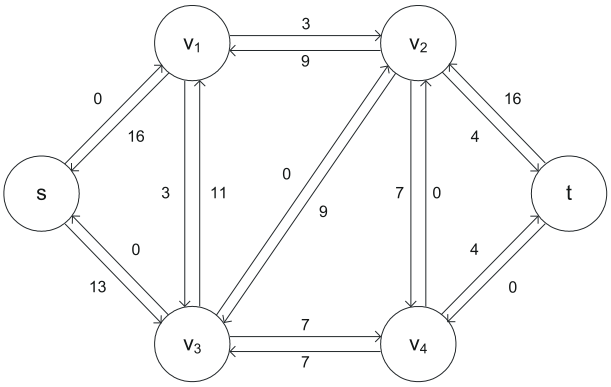
- Initialisierung:



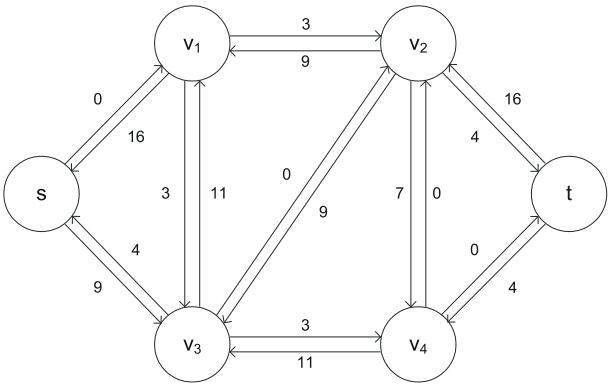
- Schritt 1 (Weg über s, v_1, v_3, v_4, v_2, t mit Minimum 7):



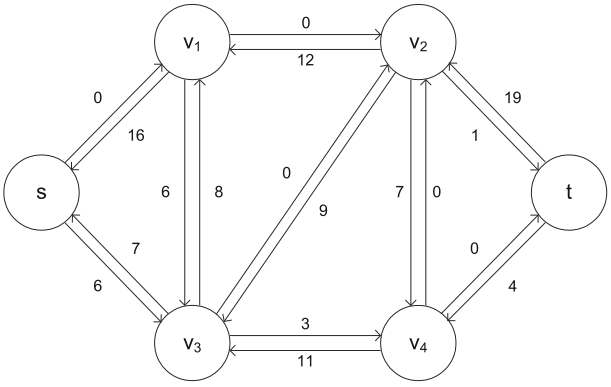
- Schritt 2 (Weg über s, v_1, v_2, t mit Minimum 9):



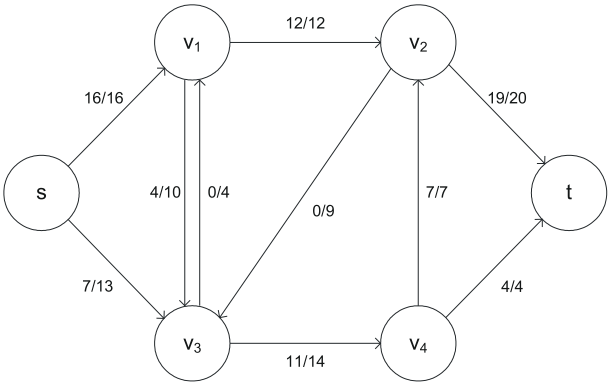
- Schritt 3 (Weg über s, v_3, v_4, t mit Minimum 4):



- Schritt 4 (Weg über s, v_3, v_1, v_2, t mit Minimum 3):



- Lösung:



Einzelheiten des Algorithmus:

- Schritt 1:

  Variable für Fluss f definieren und diese auf 0 setzen.

- Schritt 2:

  Finden eines Weges.

  Konstruiere dazu als Datenstruktur einen Graphen G':

  $$G' = (V, E') \quad \text{mit} \quad E' = E \cup \{(u, v) \mid (v, u) \in E\}$$

  Jedes Restnetz ist Teilgraph von G'.

  Kanten mit Rest 0 können ignoriert werden.

- Schritt 3:

  Konstruktion bzw. Aktualisierung des Restnetzes.

Laufzeitanalyse:

- Schritt 1:

  Kostet $\mathcal{O}(|E|)$.

- Schritt 2:

  Kostet $\mathcal{O}(|E|)$ mit Breiten- oder Tiefensuche pro Durchlauf.

- Schritt 3:

  Kostet $\mathcal{O}(|E|)$ pro Durchlauf.

  Jede Kante auf p und die Gegenkante muss aktualisiert werden.

Wie viele Durchläufe benötigt nun der Algorithmus insgesamt?

- Bei jedem Durchlauf wird der Fluss erhöht.
- Wenn wir annehmen, dass die Kapazitäten ganze Zahlen sind, erhöht sich der Fluss um mindestens 1 je Durchgang.

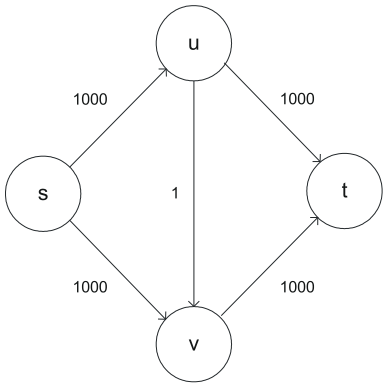
Wenn f^* der maximale Fluss ist, so gibt es höchstens $|f^*|$ Durchläufe. Die Laufzeit des Ford-Fulkerson-Algorithmus ist also $\mathcal{O}(|E| \cdot |f^*|)$.

Bemerkungen zur Laufzeitanalyse:

Diese Aussage zur Laufzeit ist unbefriedigend, weil

- wir angenommen haben, dass die Kapazitäten ganze Zahlen sind und
- der maximale Fluss $|f^*|$ exponentiell zur Eingabegröße sein kann.

Der maximale Fluss $|f^*|$ ist tatsächlich möglich!



Es werden 2000 Durchläufe erreicht, wenn abwechselnd die Pfade s, u, v, t und s, v, u, t gewählt werden.

1.3 Edmond-Karp-Algorithmus

Dieser findet immer den kürzesten Weg durch Breitensuche in Schritt 2.

Lemma 3:

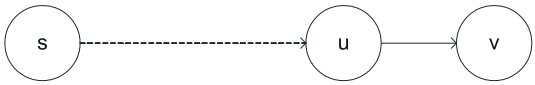
Sei $\delta_f(u, v)$ der Abstand $u, v \in V$ im Restnetz G_f.

Beim Edmond-Karp-Algorithmus gilt für alle Knoten $v \in V \backslash \{s, t\}$, dass in jedem augmentierendem Schritt der Abstand $\delta_f(u, v)$ monoton wächst.

Beweis:

Angenommen das Lemma gilt nicht, d.h. es existieren ein augmentierender Schritt und ein Knoten v, so dass $\delta_{f'}(s, v) < \delta_f(s, v)$ gilt mit f Fluss vor und f' Fluss nach dem augmentierendem Schritt.

O.b.d.A. sei v der Knoten mit der Eigenschaft, dass der Abstand $\delta_f(s, v)$ minimal ist. Der kürzeste Weg von s nach v in $G_{f'}$ sei p'. Der Knoten u sei der vorletzte Knoten auf diesem Weg p'.

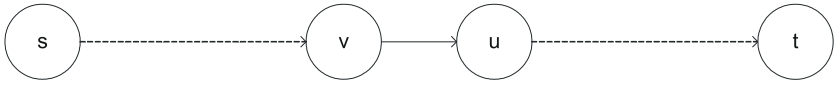


Wegen der Minimalität von v gilt $\delta_{f'}(s, v) \geq \delta_f(s, u)$. Betrachte den Fluss $f(u, v)$ vor dem augmentierenden Schritt:

- Fall 1: $f(u,v) < c(u,v)$, d.h. (u,v) ist eine Kante in G_f.

$$\begin{aligned} \delta_f(s,v) &\leq \delta_f(s,u) + 1 \\ &\leq \delta_{f'}(s,u) + 1 \\ &= \delta_{f'}(s,v) \end{aligned} \qquad \text{(Widerspruch!)}$$

- Fall 2: $f(u,v) = c(u,v)$, d.h. (u,v) ist keine Kante in G_f.
  Damit muss der augmentierende Weg p die Kante (u,v) enthalten haben.



$$\begin{aligned} \delta_f(s,v) &= \delta_f(s,u) - 1 \\ &\leq \delta_{f'}(s,u) - 1 \\ &= \delta_{f'}(s,v) - 2 \\ &< \delta_{f'}(s,v) \end{aligned} \qquad \text{(Widerspruch!)}$$

□

Lemma 4:

Der Algorithmus von Edmond-Karp führt auf einem Netz mit Kantenmenge E, Knotenmenge V höchstens $\mathcal{O}(|E| \cdot |V|)$ Augmentierungen durch.

Beweis:

Kante $u,v)$ heiße kritisch auf einem augmentierenden Weg p genau dann, wenn $c_f(p) = c_f(u,v)$.

Nach Augmentierung verschwindet eine kritische Kante aus dem Restnetz. Sie kann später nur wieder auftreten, wenn die Kante inzwischen irgendwo eine Restkapazität > 0 wieder erhalten hat, d.h. (v,u) auf dem augmentierenden Weg lag.
Sei f der Fluss bei dem (u,v) kritisch auf dem nach Edmond-Karp gefundenen Weg ist.
$\Rightarrow \delta_f(s,v) = \delta_f(s,u) + 1$.

Sei f' der Fluss, bei dem das nächste Mal (v,u) auf dem augmentierenden Weg ist.

$$\begin{aligned} \delta_{f'}(s,u) &= \delta_{f'}(s,v) + 1 \\ &\geq \delta_f(s,v) + 1 \\ &= \delta_f(s,u) + 2 \end{aligned} \qquad \text{(nach Lemma 3)}$$

Zwischen zwei Malen wo (u,v) kritisch ist, erhöht sich $\delta_f(s,u)$ um mindestens zwei:

(max. Abstand zw. s und u)

$\Rightarrow (u,v)$ wird höchstens $\frac{\overbrace{|V|-2}}{2}$ mal kritisch.

Also das ist $\mathcal{O}(|V|)$ mal.
Also kann es höchstens $\mathcal{O}(|E| \cdot |V|)$ Durchläufe geben.
Ein Durchlauf (Breitensuche) kostet $\mathcal{O}(|E|)$ Zeit, damit

Satz 2:

Der Algorithmus von Edmond-Karp hat damit eine Gesamtlaufzeit von $\mathcal{O}(|E|^2 \cdot |V|)$ $(\Rightarrow \mathcal{O}(|V|^5$ im schlimmsten Fall)

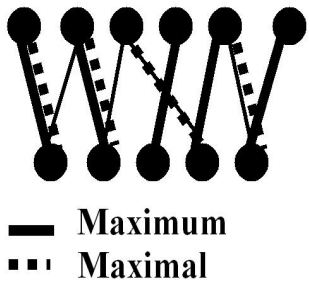
Verbesserung auf $\mathcal{O}(|V|^2 \cdot |E|)$ von Goldberg '87

am Besten bekannte Algorithmen:

- $\mathcal{O}(|V| \cdot |E| \cdot log(\frac{|V|^2}{|E|}))$ von Goldberg, Tarjan '86
- $\mathcal{O}(|V| \cdot |E| + |V|^2 \cdot log|V|)$ von Cheriyan, Mehlhorn, Hagerup '97

1.4 Bipartites Matching

Beispiel: Heiratsinstitut



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

Ein *Matching* ist eine unabhängige (d.h. kein Knoten ist inzident zu mehr als einer Kante von M) Kantenmenge $M \subset E$.
Ein Matching M heißt *maximal*, wenn $M \cup \{e\}$ kein Matching mehr ist für $e \in E \backslash M$.
Ein Matching M heißt *Maximum-Matching* wenn gilt $|M| \geq |M'|$ für alle Matchings M'.

Bemerkung:

$$M \text{ Maximum-Matching } \Rightarrow M \text{ maximal}$$
$$\Leftarrow \text{ gilt nicht!}$$

deutsche Begriffe: maximale Paarung oder größte Paarung

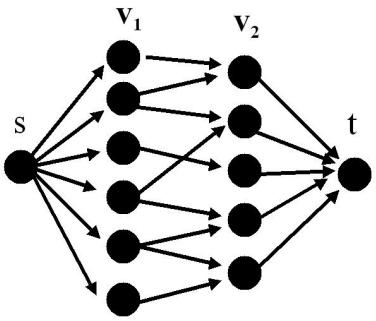
Definition:

Ein Graph $G = (V, E)$ heißt *bipartit* $\Leftrightarrow$ es gibt eine Zerlegung $V = V_1 \cup V_2$, so dass alle Kanten zwischen einem Knoten in V_1 und einem in V_2 verlaufen.

Ein maximales Matching kann einfach durch eine Greedy-Strategie gefunden werden. $\mathcal{O}(|E| \cdot |V|)$
Wir wollen das Maximum-Matching des bipartiten Graphen zurückführen auf den Netzwerkfluss.

gegeben: $G = (V, E)$ bipartiter Graph

konstruiere Netz wie folgt:



1. Richte alle Kanten in E von V_1 nach V_2
2. Füge neue Knoten s und t und alle Kanten (s, v_1) und (v_2, t) für alle $v_1 \in V_1, v_2 \in V_2$
3. Alle Kapazitäten sind 1

Lemma 5:

Für $k \in \mathbb{N}$ gilt:
Es gibt einen ganzzahligen Fluss [d.h. $f(u, v) \in \mathbb{Z}$ für alle Kanten (u, v)]
f mit $|f| = k$ in $G' \Leftrightarrow$ es gibt ein Matching M in G mit $|M| = k$.

Beweis:

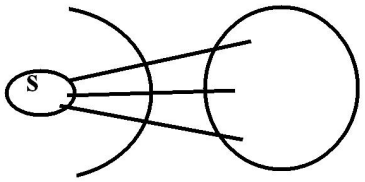
$\Leftarrow$ gegeben Matching M
schicke Fluss

- von s nach allen Knoten in V, die inzident zu einer Matching-Kante
- entlang aller M-Kanten
- von jedem Knoten in V_2, der inzident zu einer Matching-Kante ist

$\Rightarrow$ Sei f ganzzahliger Fluss.
Definiere $M = \{\{u, v\} \mid u \in V_1, v \in V_2, \underbrace{f(u,v) > 0}_{\text{also } =1}\}$

Für jedes $u \in V$, das in M vorkommt, muss $f(s, u) = 1$ sein, also gibt es nur eine von u ausgehende Kante mit Fluss > 0. Analog funktioniert es für nach $v \in V_2$ eingehende Kanten.
Für die restlichen Knoten $w \in V_1$ (oder $b \in V_2$) muss gelten $f(s, w) = 0$ (bzw. $f(b, t) = 0$).
Also ist M ein Matching.

Betrachte Schnitt $(\{s\}, V'\backslash\{s\})$: $f(\{s\}, V'\backslash\{s\}) = |M|$

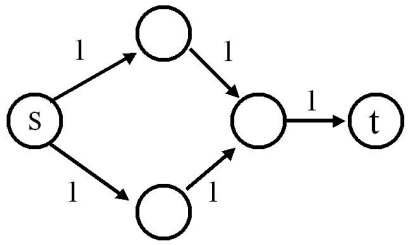


Nach dem Satz 1 (max Fluss - min Schnitt) ist $|f| = f(\{s\}, V'\backslash\{s\})$.

Also kann das Maximum-Matching in G gefunden werden durch den maximalen Fluss im Netz G'.
Es bleibt zu überlegen, damit *Lemma 5* anwendbar wird, ob der maximaler Fluss ganzzahlig ist.

Satz 3:

In einem Netz mit ganzzahligen Kapazitäten ist der max. Fluss ganzzahlig. Der Fluss kann so gewählt werden, dass $f(u, v)$ ganzzahlig für alle Kanten (u, v) ist.



Beweis:

Übung

Satz 4:

Mit dem Edmond-Karp-Flussalgorithmus kann ein Maximum-Matching in einem bipartiten Graphen $G = (V, E)$ mit der asymptotischen Laufzeit $\mathcal{O}(|V| \cdot |E|^2)$ gefunden werden.

Bester bekannter Algorithmus ist: $\mathcal{O}(\sqrt{|V|} \cdot |E|)$ von Hapcraft/Karp '73

Varianten:

- Maximum-Matching in allgemeinen Graphen $\mathcal{O}(\sqrt{|V|} \cdot |E|)$
- Maximum-Matching mit minimalem Gewicht $\mathcal{O}(|E| \cdot |V| \cdot log|V|)$

2 String Matching

endliches Alphabet $\sum$

- $\sum = \{0, 1\}$
- $\sum =$ ASCII-Zeichen

gegeben:

- „Text" $T \in \sum^*$ mit $|T| = n$
- „Muster" $P \in \sum^*$ mit $|P| = m$

Problemstellung: finde alle Vorkommen von P in T als Felder $T[1..n]$, $P[1..m]$

also gesucht: alle Stellen s mit $T[s + i] = P[i]$ für $i = 1, .., m$

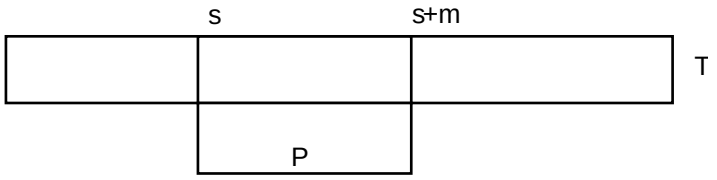
Anwendungsbeispiele:

- Textverarbeitung
- Internet-Suchmaschine
- DNS-Folgen

Präfix	$w \sqsubset x$	w ist Präfix von x
Suffix	$w \sqsupset x$	w ist Suffix von x

Abkürzung $P_k := P[1..k]$

also gesucht alle s mit $P \sqsupset T_{s+m}$



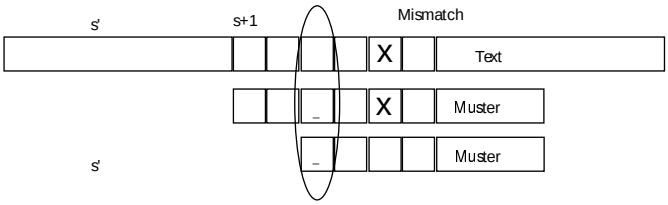
Algorithm 1 Naiver Algorithmus

```
1:  for i ← 1, n − m + 1 do
2:    miss ← false
3:    for j ← 1, m do
4:      if P[j] ≠ T[i + j] then
5:        miss ← true
6:        break
7:      end if
8:    end for
9:    if miss=false then
9:      P ⊐ T_{i+m}
10:     gib i-m aus
11:   end if
12: end for
```

Laufzeit: Die Laufzeit beträgt $\mathcal{O}(n \cdot m)$

Im schlechtesten Fall, wenn z.B. $T = a^n$ und $P = a^{m-1}b$, kann die Laufzeit auch $\Theta(n \cdot m)$ betragen.

2.1 Algorithmus von Knuth-Morris-Pratt



Wie weit kann man ein Muster schieben.
Beim nächsten Versuch s' nur sinnvoll, wenn hier(_) Übereinstimmung
also:

$P[1..q]$ stimmt mit $T[s+1...s+q]$ überein $\Leftrightarrow$ $k < q$ maximal mit $P_k \sqsupset P_q$
$P[q+1] \neq T[s+q+1]$

Finde kleinstes s' mit: $P[1..k] = T[s'+1...s'+k]$
wobei: $s'+k = s+q$, also $k = s+q-s'$

Definiere Präfix Funktion von P

$\pi := \{1,..,m\} \rightarrow \{0,..,m-1\}$
$\pi(q) := max\{k \mid k < q \text{ und } P_k \sqsupset P_q\}$

Algorithm 2 Algorithmus von Knuth-Morris-Pratt

```
1:  berechne für P die Präfixfunktion von π
2:  q ← 0
3:  for i ← 0, n do
4:     while (q > 0) & (P[q+1] ≠ T[i]) do
4:        q ← π(q)
5:     end while
6:     if P[q+1] == T[i] then
7:        q ← q+1
8:     end if
9:     if q == m then
10:       Ausgabe:(i − m);
11:       q ← π(q)
12:    end if
13: end for
```

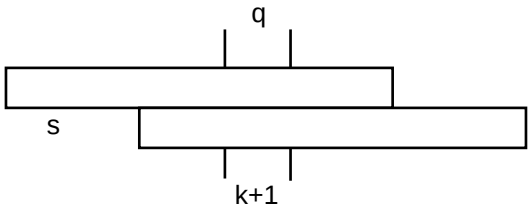
Algorithm 3 Berechnen der Präfix-Funktion PREF(P)

```
1:  π[1] ← 0
2:  k ← 0
3:  for q ← 2, m do
4:     while (k > 0) & (P[k + 1] ≠ P[q]) do
5:        k ← π(k)
6:     end while
7:     if P[k+1] == P[q] then
8:        k ← k + 1
9:     end if
10:    π[q] ← k
11: end for
```

Laufzeit von Pref(P)

Zeilen(3,7,10) m-Iterationen mit einer Komplexität von jeweils 1.
Das Ergibt $\mathcal{O}(m)$.

Zeilen(4,5) Der Abstand $s = q-k+1$ wird in jeder Iteration der while-Schleife vergrößert. Er kann maximal m sein.
$\Rightarrow$ Insgesamt sind das $\leq m$ Iterationen.
Damit ergibt sich als Laufzeit $\mathcal{O}(m)$.



insgesamt $\mathcal{O}(m)$

Laufzeit von KMP: analoge Analyse

Abstand s vergrößert sich in jeder Iteration der while Schleife und kann nicht größer als n werden.

Laufzeit insgesamt für KMP: $\mathcal{O}(m+n)$

Korrektheit Sei $\pi^*(q) = \{q, \pi(q), \underbrace{\pi^2(q)}_{\pi(\pi(q))}, \ldots, \underbrace{\pi^t(q)}_{=0}\}$

Lemma 1

$\pi^*(q) = \{k \mid P_k \sqsupset P_q\}$

Beweis

$\subset$: Induktion über l: Behauptung: $i \in \pi^*(q) \Rightarrow \exists l : \; i = \pi^l(q)$

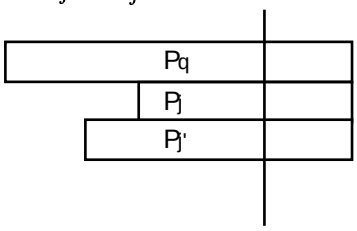
I.A.: $l = 0$: $\quad \pi^0(q) = q$ und $P_q \sqsupset P_q$

I.S.: $l \to l+1$: $\quad P_{\pi^{l+1}(q)} \underbrace{\sqsupset}_{Def.von\pi} P_{\pi^l(q)} \underbrace{\sqsupset}_{I.V.} P_q$

$\supset$: Beweis durch Widerspruch
Sei j maximal mit dieser Eigenschaft, d.h.

$P_j \sqsupset P_q$ aber $j \notin \pi^*(q) \Rightarrow j < q$
Sei $j' \in \pi^*(q)$ minimal mit $j' > j$

also $j < j' \leq q$
$\Rightarrow P_j \sqsupset P_{j'}$



denn $P_j \sqsupset P_q$ und es gibt kein $l, j < l < j'$ mit $P_l \sqsupset P_{j'}$ wegen Maximalität von j und Minimalität von j'.

also $j = \pi(j')$ mit $j' \in \pi^*(q)$
$\Rightarrow j \in \pi^*(q)$ (Widerspruch!)

Lemma 2

Falls $\pi(q) > 0$, dann ist $(\pi(q) - 1) \in \pi^*(q-1)$

Beweis

$$\begin{aligned} P_{\pi(q)} \sqsupset P_q \quad &\Rightarrow \quad P_{\pi(q-1)} \sqsupset P_{q-1} \\ &\Rightarrow \quad (\pi(q) - 1) \in \pi^*(q-1) \end{aligned}$$

Definition:

$E_{q-1} \subset \pi^*(q-1)$ durch
$E_{q-1} = \{k \in \pi^*(q-1) \mid P[k+1] = P[q]\}$
also $k \in E_{q-1} \Rightarrow P_k \sqsupset P_{q-1}$ und $P_{k+1} \sqsupset P_q$.

Korollar:

$$\pi(q) = \begin{cases} 0 & \text{, falls } E_{q-1} = \emptyset \\ 1 + \max E_{q-1} & \text{, sonst} \end{cases}$$

Beweis:

Sei $r = \pi(q)$, dann $P_r \sqsupset P_q$ und es gibt kein $r' > r$ mit $P_{r'} \sqsupset P_q$.

$$\begin{aligned}
\text{Da } P_r \sqsupset P_q &\Leftrightarrow P[r] = P[q] \\
&\Leftrightarrow r = 1 + \max\{ \underbrace{k \in \pi^*(q-1)}_{\begin{pmatrix} P_k \sqsupset P_{q-1} \\ P_{k+1} \sqsupset P_q \end{pmatrix}} \mid P[k+1] = P[q]\} \\
&\Rightarrow r = 1 + \max E_{q-1})
\end{aligned}$$

$$\begin{aligned}
\text{Falls } r = 0 \Leftrightarrow & \text{ es gibt kein } j < q \text{ mit } P_j \sqsupset P_q \\
\Leftrightarrow & \text{ Bedingung von } E_{q-1} \text{ wird nie wahr} \\
\Leftrightarrow & E_{q-1} = \emptyset
\end{aligned}$$

Korrektheit von $PREF(P)$

Am Anfang jeder Iteration der for-Schleife ist $k = \pi(q-1)$ wegen Zeile 1,2,10. While-Schleife durchsucht $\pi^*(q-1)$, beginnt bei $k = \pi(q-1)$ und durchläuft $\pi^*(q-1)$ [mit der Anweisung $k := \pi(k)$] bis ein k gefunden wird
mit $P[k+1] = P[q]$

$\Rightarrow k = max\,(E_{q-1})$ in Zeile 8 wird k um 1 erhöht
$\Rightarrow \pi(q) = 1 + max\,(E_{q-1})$, das ist korrekt nach dem obigen Korollar

Falls keine Stelle j mit $P[j] = P[q]$ gefunden wird, bleibt $k = 0$, das ist ebenfalls korrekt nach dem obigen Korollar

2.2 Suffixbäume

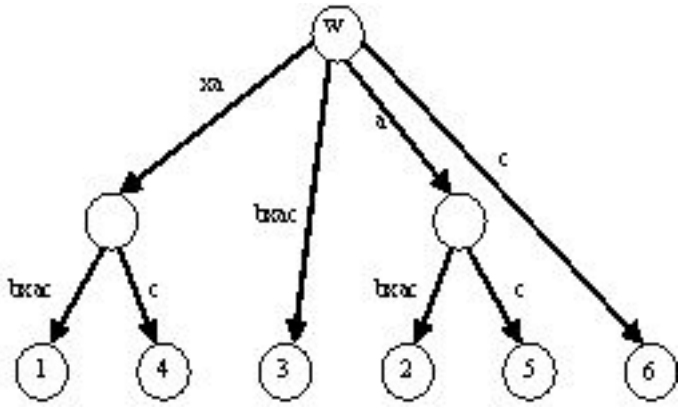
Datenstrukturen für String-Matching Probleme

Definition

Ein *Suffixbaum* T für ein Wort S der Länge n ist ein gewichteter Baum mit genau n Blatttern und den folgenden Eigenschaften:

1. jeder innere Knoten, der nicht die Wurzel ist, hat ≥ 2 Kinder
2. jede Kante ist mit einem nichtleeren Teilwort von S beschriftet
3. keine zwei Kanten aus einem Knoten haben Markierungen, die mit dem gleichen Zeichen anfangen
4. für Blatt mit Nummer i gilt:
   Die Konkatenation der Markierungen der Kanten auf dem Weg von der Wurzel zu Blatt mit Nummer i ist der Suffix $S[i, \ldots, n]$

Beispiel: $S = xabxac$



Weg $\rightarrow$ 1: xabxac $= S[1 \ldots 6]$
Weg $\rightarrow$ 2: xac $= S[4 \ldots 6]$

Problem: Konstruktion ist nicht möglich, wenn ein Suffix α Präfix von einem Suffix β ist.

Abhilfe: Sonderzeichen \$ ans Ende von S, wobei \$ nicht in S vorkommt.

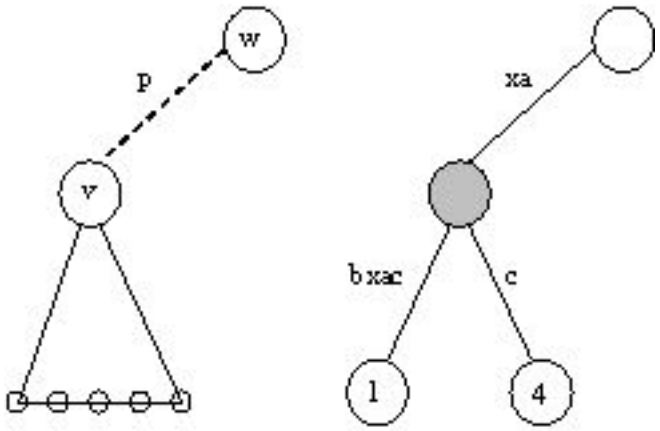
Definition

Die *Markierung* eines Weges in einem Suffixbaum T ist die Konkatenation der Markierung der Kanten auf diesem Weg.
Die Markierung eines Knoten v: ist die Markierung des Weges von der Wurzel zu v.

2.3 String-Matching mit Hilfe von Suffixbäumen

gegeben: Wort S, Muster P
gesucht: Alle Stellen von S an denen P vorkommt

Annahme: Der Suffixbaum T für S ist vorhanden



Suche $xa \rightarrow$ Blätter geben Indizes an!

1. Beginnend bei der Wurzel durchsuche T nach einem Knoten v, dessen Markierung $= P$ ist (Sonderfall: Markierung beginnt mit P)
2. besuche alle Blätter des Teilbaumes mit Wurzel v
   $\rightarrow$ Gib diese Indizes aus.

Schritt 1 genauer:

- Beginne bei der Wurzel und nimm die Kante e, deren Markierung mit $P[1]$ beginnt.

  $\rightarrow$ falls es so eine Kante nicht gibt

  $\Rightarrow$ Muster nicht gefunden

- Überprüfe zeichenweise das Muster P mit der Markierung der Kante

  $\rightarrow$ falls P ein Präfix der Markierung der Kante $e = (u, v)$ ist

  $\Rightarrow$ Muster nicht gefunden
  gib alle Indizes in den Blättern des Teilbaumes von v aus

  $\rightarrow$ falls Markierung von e ein Präfix von P ist

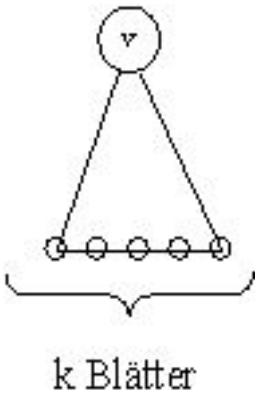
  $\Rightarrow$ $P' = P$ ohne die Markierung von e
  falls $P' \neq \epsilon$ suche rekursiv weiter mit P'

  $\rightarrow$ falls Markierung von e kein Präfi von P ist

  $\Rightarrow$ Muster nicht gefunden

Laufzeit:

- Schritt 1:
  $\mathcal{O}(m)$ Zeit, wobei $|P| = m$, weil höchstens das ganze Muster einmal durchlaufen wird.
- Schritt 2:



Sei k die Anzahl der Indizes, an denen P in S vorkommt. Baum an v hat genau k Blätter und $\leq k-1$ innere Knoten, weil Außengrad jedes inneren Knoten ≥ 2.
$\rightarrow \mathcal{O}(k)$

- Zusammen:
  Falls P k-mal in S vorkommt:
  Laufzeit: $\mathcal{O}(m+k)$

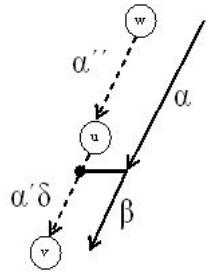
2.4 Konstruktion von Suffixbäumen

naiv: Konstruiere Folge von Suffixbäumen $T_1, \ldots, T_n$, wobei T_i enthält alle Suffizes $S[1 \ldots n], \ldots, S[i \ldots n]$

$T_1 :=$ ist der Baum, der nur eine Kante enthält und diese ist mit S beschriftet.

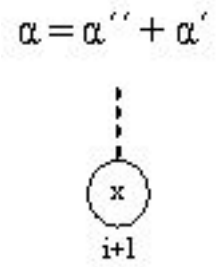
Erzeuge T_{i+1} wie folgt aus T_i:

1. Durchlaufe T_i mit dem Suffix $S[i+1, \ldots, n]$ als Eingabe (wie beim String-Matching-Algorithmus) bis es nicht mehr weiter geht.

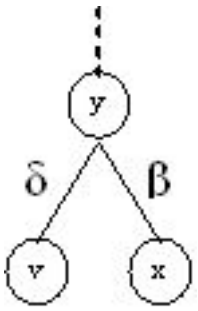


Sei $S[i+1, \ldots, n] = \alpha\beta$, wobei α ein maximaler Präfix ist, der als Markierung eines Knoten v plus Präfix α' der Markierung $\alpha'\delta$ der Kante $e = (u, v)$ auftritt.

2. Schaffe neues Blatt x mit Index $i+1$



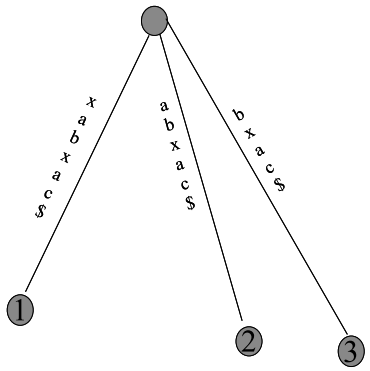
3. falls $\alpha' = \epsilon$ also $\alpha'' = \alpha$, schaffe neue Kante $e' = (u, x)$ mit Markierung β.
   sonst: Schaffe neuen Knoten y, ersetze Kante e durch $e_1 = (u, y)$ und $e_2 = (y, v)$. Schaffe neue Kante $e' = (y, x)$.
   beschrifte: e_1 mit α', e' mit β und e_2 mit δ.

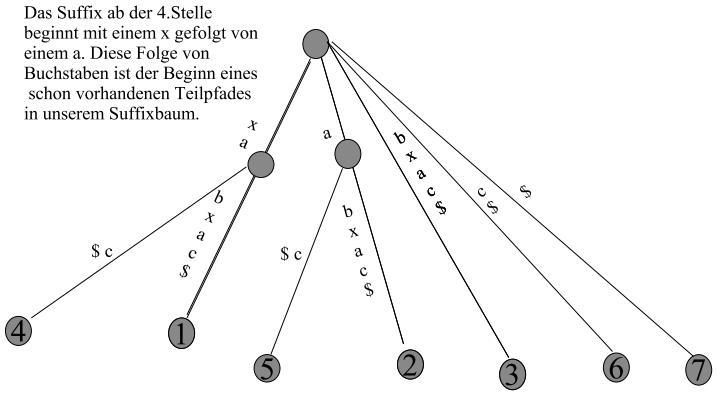


Wiederholung von Suffixbäumen

Zum Beispiel für die Konstruktion eines Suffixbaums für das Wort $xabxac\$$

der naive Konstruktionsalgorithmus





Erläuterung des naiven Algorithmus:
Bei der Konstruktion des Suffixbaumes werden nacheinander die Teilpfade der Suffixe des Wortes beginnend mit dem 1.Index konstruiert.
Ist dabei ein Suffix s ein Präfix eines schon im Baum enthaltenen Teilpfades, so wird nach der Stelle, an der es keine Übereinstimmung zwischen s und dem Teilpfad mehr gibt, ein zusätzlicher Knoten eingefügt, von dem nun eine weitere Kante abgeht. Diese Kante wird dann mit dem Rest des Suffixes s beschriftet und bekommt ein Blatt mit dem Index des Suffix s.

Laufzeit:

$$\begin{aligned}\mathcal{O}(\sum_{i=1}^{n}|S_i|) &= \mathcal{O}(\sum_{i=1}^{n}(n-i)) \\ &= \mathcal{O}(n^2)\end{aligned}$$

Der naive Algorithmus konstruiert also den Suffixbaum für ein gegebenes Wort der Länge n ist $\mathcal{O}(n^2)$ Zeit.
Es geht jedoch besser: Der Algorithmus von Ukkonen konstruiert solch einen Suffixbaum in $\mathcal{O}(n)$ Zeit.

2.5 Anwendungen von Suffixbäumen

1. **String-Matching** (bereits besprochen)

   Der Algorithmus von Ukkonen liefert einen alternativen Algorithmus:

   gegeben: Muster P, Text T mit $|T| = n, |P| = m$

   - konstruiere Suffixbaum von $T \Rightarrow$ in $\mathcal{O}(n)$ möglich
   - Suche von P im Suffixbaum $\Rightarrow$ in $\mathcal{O}(m)$ Zeit
   - Beschriftungen der Blätter im entsprechenden Unterbaum liefern Positionen, an denen P auftritt.

     $\Rightarrow$ in $\mathcal{O}(k)$ Zeit, wobei k die Anzahl der Vorkommen in P darstellt.

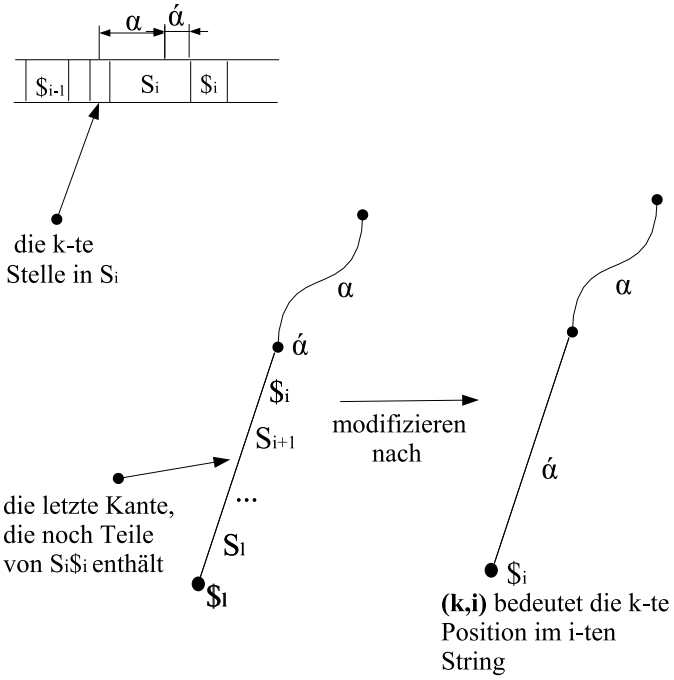
     Also kann man das Entscheidungsproblem (Ist P in T enthalten?) in $\mathcal{O}(n+m)$ lösen.

   Insbesondere sind Suffixbäume eine Datenstruktur für einen Text T, die für die Beantwortung von Matchinganfragen geeignet ist;
   beispielsweise für das Problem:

1a. Finde eine Menge von String s in einem Text.
   Aho/Corasick entwickelten einen Algorithmus dafür, der vergleichbare Laufzeit hat, jedoch nicht von Syntaxbäumen Gebrauch macht.

2. Angenommen, man hat eine Datenbank von Texten $S_1, \ldots, S_l$.
   Gegeben sei ein Muster P.
   Finde nun alle Vorkommen von P in den Texten $S_1, \ldots, S_l$.
   $\rightarrow$ Konstruktion eines verallgemeinerten Suffixbaum für $S_1, \ldots, S_l$.
   Sei $n = \sum_{i=1}^{l} |S_i|$.

   Idee: Bilde den Gesamttext $S_1\$_1 S_2\$_2, \ldots, S_l\$_l$, wobei $\$_1, \ldots, \$_l$ alle verschieden sind und sonst nicht vorkommen.
   Für diesen Gesamttext wird der Suffixbaum konstruiert, was mit dem Algorithmus von Ukkonen in $\mathcal{O}(n)$ Zeit geht. Der Baum wird noch modifiziert (denn wir wollen das Muster in den einzelnen S_i mit $i = 1, \ldots, l$ suchen).



   Dann folgt die Suche nach Mustern wie vorher auch.
   Die Datenbank von Texten der Gesamtlänge n kann damit in Zeit $\mathcal{O}(n)$ vorverarbeitet werden, so dass die Suche nach allen Vorkommen eines Musters der Länge m in $\mathcal{O}(m + k)$ Zeit möglich ist, mit k als der Anzahl der Vorkommen des Musters.

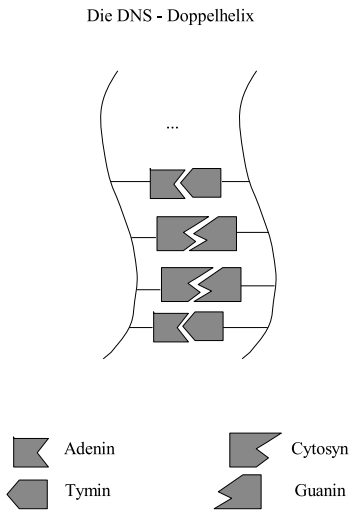
3. **Finden des längsten gemeinsamen Teilwortes**
   Gegeben: sind zwei Zeichenketten S_1 und S_2 .
   Gesucht: ist das längste Wort w, das Teilwort von S_1 und S_2 ist.

   Dies geht mit folgendem Algorithmus:

   - Konstruiere den verallgemeinerten Suffixbaum.
     $\rightarrow$ Dank Ukkonen braucht das $\mathcal{O}(n)$ Schritte.
   - Markiere jeden inneren Knoten v mit 1 (bzw. 2), wenn es im Unterbaum von v mindestens ein Blatt gibt, welches zu einem Suffix in S_1 (bzw. S_2) gehört. Dies kann schon während des Aufbaus des Suffixbaumes im vorigen Schritt geschehen.
   - Mache eine Breitensuche im Suffixbaum, um den tiefsten Knoten t zu finden der mit 1 und 2 beschriftet ist. Das Wort welches auf dem Pfad von der Wurzel zu p ablesbar ist, ist damit das längste gemeinsame Teilwort.

   Da sich die Dauer der Breitensuche auch linear mit der Größe des Suffixbaumes erhöht, kann das längste gemeinsame Teilwort in $\mathcal{O}(n)$ gefunden werden. (mit $n = |S_1| + |S_2|$)

4. **Anwendungen in der Bioinformatik**
   Die DNS (Desoxyribonukleinsäure) ist eine Ansammlung von Molekülen, die die Erbinformation einen Organismus enthält.
   Sie besteht aus einer Doppelkette und 4 Bausteinen (ebenfalls Moleküle): Adenin, Thymin, Cytosin und Guanin (sog. Nukleotide).
   Sie werden abgekürzt mit $\{A, T, C, G\}$.
   Es können sich jeweils immer nur A und T sowie C und G verbinden.



   Mehrere DNS - Moleküle bilden die Chromosomen, deren Gesamtheit man Genom nennt.

Proteine sind Ketten von 20 verschiedenen Aminosäuren, können somit als Zeichenketten über einem 20-elementigen Alphabet aufgefasst werden. Typische Längen dieser Ketten liegen im Bereich von einigen 100 Bausteinen. Bei Bakterien gibt es 500 bis 1500 verschiedene Proteine beim Menschen etwa 100000.

Je ein Tripel der Nukleotide (A, T, C, G) kodiert eine Aminosäure. Dies ist keine injektive Zuordnung, da $4^3 = 64$ verschiedene Nuleotid-Tripel nur 20 verschiedene Aminosäuren kodieren. Die Tripel heissen auch Codon.

Bsp: TTT kodiert Phenylalanin (F) unf GTT kodiert Valin (V)

Hat man ein DNS-Fragment, so muss man an der richtigen Stelle anfangen, es in solche Tripel zu zerlegen.
Da man den Anfang nicht kennt muss man alle 3 Möglichkeiten ausprobieren.

Ein DNS-Fragment (Tripel), welches ein Protein erzeugt, nennt man Gen. Die Aufgabe, die sich hier stellt ist die sog. Sequenzierung. Das ist die Bestimmung der Folge von Nukleotiden eines DNS-Moleküls aus einer Menge vorhandener Fragmente.

Zwei Projekte (Celera, Human Genome Project) befassten sich mit der Sequenzierung des menschlichen Genoms, welches aus 10^9 bis 10^{10} Nukleotiden besteht. (Etwa 98% davon sind bei jedem Menschen gleich, nur die restlichen 2% machen die Unterschiede zwischen verschiedenen Menschen aus.)
Die vorhandenen Fragmente müssen bei der Sequenzierung zusammengesetzt werden. Es müssen dazu Überlappungen der Fragmente untersucht werden: das Präfix des einen Fragments überlappt mit dem Suffix eines anderen Fragments.

Dieses ganze Problem läßt sich formulieren als:
„Finde das kürzeste Oberwort (Genom) für eine Menge an Worten (Fragmente).“
Wie in der Übung gezeigt wird, ist dieses Problem NP-schwer.

Übersetzt auf das Problem *1a* kann man die Sequenzierung so deuten:
Der Text aus *1a* ist ein bereits sequenziertes Stück des Genoms.
Muster sind neu hinzugekommene Stücke die in dem vorhandenen Teilwort des Genoms gesucht werden.

3 Approximationsalgorithmen

Für viele Probleme in der Praxis sind keine effizienten Algorithmen bekannt (NP-schwer).

Diese (z.B. TSP) werden mit Approximationsalgorithmen berechnet, die effizient berechenbar sind, aber nicht unbedingt die optimale Lösung liefern.
Die „Qualität“ der Lösung ist höchstens um einen konstanten Faktor schlechter ist als die des Optimums.

Definition:

P ist *Optimierungsproblem* mit Kostenfunktion c : Lösungen $\mapsto \mathbb{R}_+$.
Sei A ein Algorithmus der bei Eingabe I die Lösung $f_A(I)$ liefert.
Sei $f_{opt}(I)$ die optimale Lösung (d.h. die c maximiert oder minimiert).

A heißt ϵ-*Approximationsalgorithmus*, falls $\frac{|c(f_A(I)-c(f_{opt}(I)|}{c(f_{opt}(I)} \leq \epsilon$.

Für ein Minimierungsproblem heißt das: $c(f_A(I) \leq (1+\epsilon) \cdot c(f_{opt}(I))$.

3.1 Approximationsalgorithmus für TSP:

TSP für n Knoten sei durch Abstandsmatrix $D = (d_{ij})\ ,\ 1 \leq i,j \leq n$

gegeben: vollst. Graph mit n Knoten, d_{ij} = Kosten der Kante (i,j).

gesucht: Rundreise mit minimalen Kosten. Dies ist NP-schwer!

Wir sagen: D erfüllt die Dreiecksungleichung
$\Leftrightarrow d_{ij} + d_{jk} \geq d_{ik}$ für $\forall i,j,k \in \{1,...,n\}$

Dies ist insbesondere dann erfüllt, wenn D die Abstände bezüglich einer Metrik darstellt oder D Abschluss einer beliebigen Abstandsmatrix C ist, d.h. d_{ij} = Länge des kürzesten Weges (bzgl. C) von i nach j.

Definition:

TSP mit Dreiecksungleichung heiße ΔTSP.

Satz:

ΔTSP ist NP-schwer.

Beweis:

Reduktion Hamilton-Kreis $\leq_p \Delta TSP$

Sei $G = (V,E)$ Problemstellung für Hamilton-Kreis.
Definiere Matrix $(d_{ij})\quad 1 \leq i,j \leq n, n = |V|$

$$d_{ij} = \begin{cases} 1 & ,(i,j) \in E \\ 2 & ,sonst \end{cases}$$

Dann existiert ein Hamilton-Kreis in G
$\Leftrightarrow$ die optimale Lösung von ΔTSP hat Kosten n.

Approximationsalgorithmus

Wir betrachten den Multigraphen $G = (V, E)$, d.h. zwischen 2 Knoten sind mehrere Kanten erlaubt.

G heißt Eulersch
$\Leftrightarrow \exists$ ein geschlossener Weg, der jeden Knoten mindestens einmal und jede Kante genau einmal durchläuft.

Satz:

$G = (V, E)$ ist Eulersch $\Leftrightarrow$

a) G ist zusammenhängend

b) alle Knoten haben geraden Grad

gegeben: Abstandmatrix $D = (d_{ij})$, $1 \leq i, j \leq n$
Eulerscher aufspannender Graph ist ein Multigraph Eulersch und hat $\{1...n\}$ als Knotenmenge und Kosten $c(G) := \sum_{i,j \in E} d_{ij}$

Lemma:

Ist $G = (V, E)$ ein EAG für D, dann lässt sich eine Rundreise τ von V mit $c(\tau) \leq c(G)$ in $\mathcal{O}(|E|)$ Zeit finden.

Beweis:

Konstruiere Eulerschen Weg (in $\mathcal{O}(|E|)$)

$$\begin{aligned} W &= i_1 \dots i_2 \dots i_3 \dots i_n \text{ mit } \{i_1, \dots i_n\} \\ &= \{1 \dots n\}. \end{aligned}$$

allgemein: i_j ist der erste Knoten der $\notin \{i_1 ... i_{j-1}\}$, $\alpha_j =$ Weg von i_j nach i_{j+1}.

Betrachte Tour $\tau = (i_1, i_2, ..., i_n)$, Kosten $c(\tau) = d_{i_1 i_2} + d_{i_2 i_3} + \ldots + d_{i_{n-1} i_n} + d_{i_n i_1}$
wegen Dreiecksungleichung: $d_{i_j, i_{j+1}} \leq c(\alpha_j)$

$$\begin{aligned} \text{Damit } c(\tau) &\leq c(\alpha_1 + \alpha_2 + \ldots + \alpha_{n-1} + \alpha_n \\ &= c(w) \\ &= c(G) \end{aligned}$$

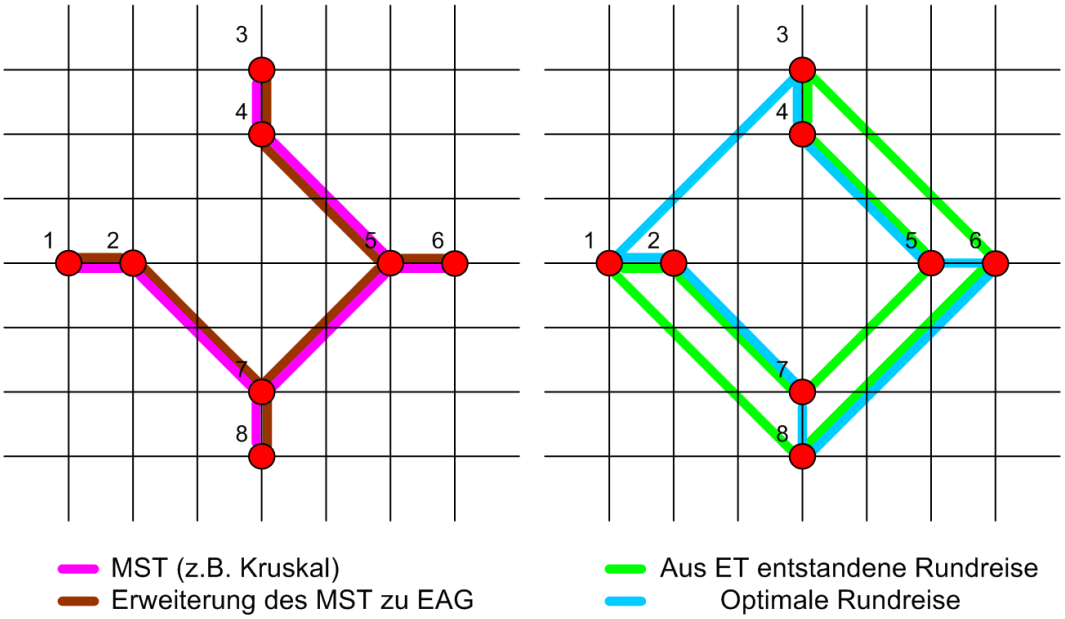
Der Algorithmus erzeugt einen minimal spannenden Baum (MST) aus der Matrix D. Ein MST kann in polynomieller Zeit $\mathcal{O}(n^2 log(n))$ bestimmt werden, z.B. mit dem Algorithmus von Kruskal.

3.2 Baum-Heuristik für ΔTSP:

Eingabe $D = (d_{ij}) \quad 1 \leq i, j \leq n$, wobei D die Dreiecksungleichung erfüllt.

1. Berechne $MSTT$ für D
2. Verdopple jede Kante von T: EAG von D
3. Bestimme in G Eulerweg und daraus (siehe Lemma) eine Rundreise τ

L_∞ Abstand $a, b \in R^2$
$d(a,b) = max(|a_x - b_x|, |a_y - b_y|)$ erfüllt die Dreiecksungleichung



Satz:

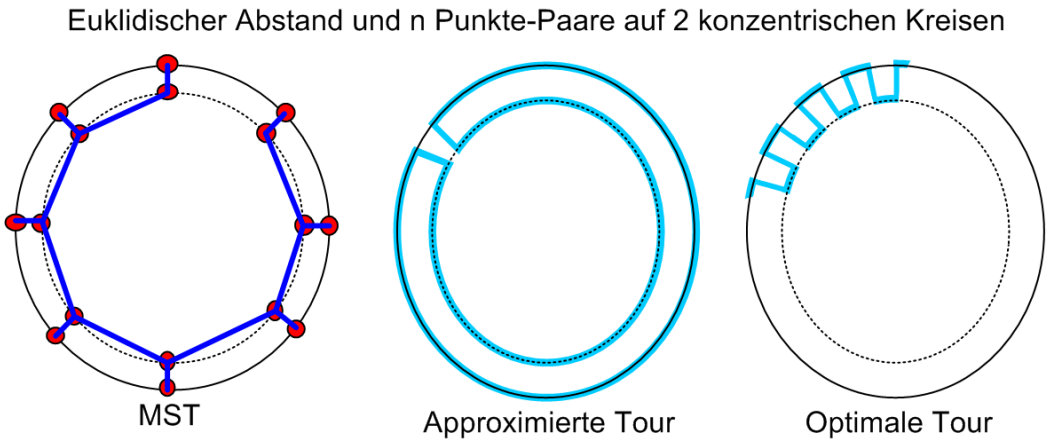
Die Baum-Heuristik ist ein 1-approximativer Algorithmus für ΔTSP.

Beweis:

Betrachte eine optimale Tour τ_{opt}.
Nimmt man eine Kante heraus, erhält man einen spannenden Baum T.

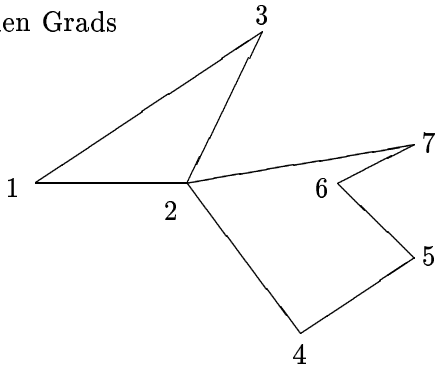
$$\begin{aligned} \text{Es gilt: } c(\tau_{opt}) &\geq c(T) \\ &\geq c(MST) \\ &= 1/2 c(G) \\ &\geq 1/2 c(\tau) \end{aligned}$$

Es gibt Beispiele, wo die Baum-Heuristik dem Faktor 2 beliebig nahe kommt.



3.3 Christofides' Heuristik - Annäherung an die perfekte Lösung für TSP

Idee: Mit Hilfe eines Perfekten Matchings auf Knoten ungeraden Grads aus einem MST einen Eulergraphen machen.



Wir suchen: Perfektes Matching minimalen Gewichtes.
[vgl. Problem 35 - 4 in Introduction to Algorithms]

Perfektes Matching ist ein Matching auf Graphen mit n Knoten, das aus $\lfloor \frac{n}{2} \rfloor$ Kanten besteht und kein Knoten zu mehr als einer Kante inzident ist.

Gewichtetes Matching: alle Kanten haben Gewichte, und die Summe aller ist das Matching-Gewicht

Vorgehen: Füge Kanten zwischen Knoten ungeraden Grads hinzu um die ungeraden Grade zu eliminieren und einen Eulerschen Aufspannenden Graphen (EAG) zu erhalten.

Dies ist in polynomieller Zeit findbar. $O(n^3)$ mit Algorithmus von Lawler.

3.4 Christofides' Algorithmus für $\triangle$ TSP

Eingabe: Abstandsmatrix $\mathfrak{D}$

1. Berechne MST $\mathfrak{T}$ bezüglich $\mathfrak{D}$

2. für Knoten in $\mathfrak{T}$ mit ungeradem Grad:

   finde perfektes Matching minimalen Gewichtes $\mathfrak{M}$

   $\mathfrak{G}$:= Multigraph aus Kanten von $\mathfrak{T}$ und $\mathfrak{M}$

3. Bestimme Eulerschen Weg in $\mathfrak{G}$

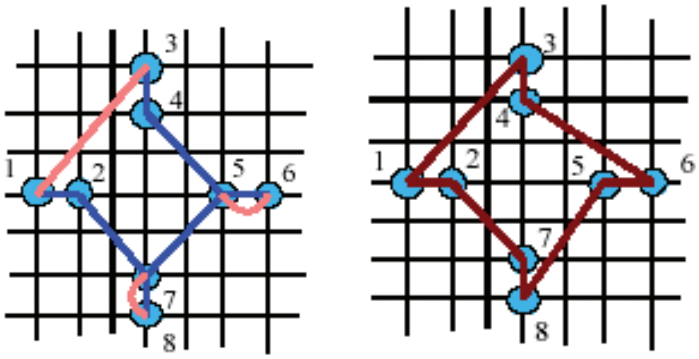
4. aus Eulerschen Weg konstruiere Rundreise τ wie vorher

Das Verfahren von Christofides erreicht im schlechtesten Fall eine Abweichung um Faktor 1.5 von der optimalen Rundreise.

Beispiel:

Christofides:

1. Erzeuge einen MST aus n Knoten. L_∞-Abstand: max (x,y)
2. Berechne perfektes Matching mit min. Gewicht für die Knoten mit ungeradem Grad und füge sie in den MST ein, um einen Eulergraphen zu erhalten. Baum-Heuristik: Tour der Länge 17; Optimale Lösung: Tour der Länge 14
3. Konstruiere einen Eulerweg. Eulerweg: $\longrightarrow$ 1 2 7 8 7 5 6 5 4 3 1
4. Leite den Hammiltonkreis (die Tour) ab. Tour: $\longrightarrow$ 1 2 7 8 5 6 4 3
   Kosten (Summe der L_∞-Abstände)$\longrightarrow$ 1+2+1+3+1+3+1+3 = 15



Satz: Die Christofides-Heuristik hat Laufzeit $O(n^3)$ und ist ein $\frac{1}{2}$ - approximativer Algorithmus für $\triangle$ TSP.

Beweis: $O(n^3)$ fürs Matching (Zeile 2), alle anderen Operationen sind billiger. Der in Schritt 2 konstruierte Graph $\mathfrak{G}$ ist Eulersch, denn alle Knoten erhalten geraden Grad. Begründung: jeder Konten mit ungeradem Grad wird Endpunkt einer Matchingkante, weil das Matching perfekt ist und es geradzahlig viele Knoten mit ungeraden Grad in $\mathfrak{T}$ gibt. Dies gilt übrigens für jeden Graphen.
Für die Rundreise (Tour) τ gilt:

$c(\tau) \leq c(\mathfrak{G}) = c(\mathfrak{T}) + c(\mathfrak{M})$ und $c(\mathfrak{T}) \leq c(\tau_{opt})$

|
Lemma

Argument: $\mathfrak{T}$ ist Baum, τ_{opt} ist ein Kreis
Kante aus τ_{opt} raus $\longrightarrow \mathfrak{T}$

Sei $i_1, i_2, ..., i_{2m}$ eine Folge der Knoten von $\mathfrak{T}$ mit ungeradem Grad in der Reihenfolge wie in τ_{opt}.
Also $\tau_{opt} = \alpha_0\, i_1\, \alpha_1\, i_2\, ...\, \alpha_{2m-1}\, i_m\, \alpha_{2m}$

|
Stücke dazwischen

Betrachte Matchings:
$M_1 = \{\{i_1, i_2\}, \{i_3, i_4\}\ ...\ \{i_{2m-1}, i_{2m}\}\}$
$M_1 = \{\{i_2, i_3\}, \{i_4, i_3\}\ ...\ \{i_{2m}, i_1\}\}$

Wegen $\triangle$-Ungleichung gilt: $c(\mathfrak{T}_{opt}) \geq c(M_1) + c(M_2)$

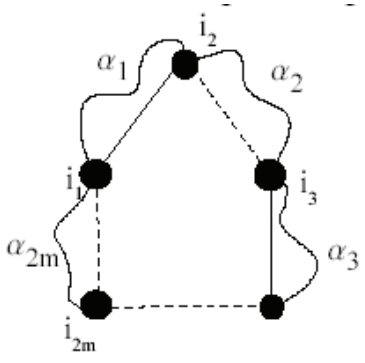
$c(\tau_{opt}) = c(\alpha_1) + c(\alpha_2) + ... + c(\alpha_m)$

$\geq c(i_1, i_2) + c(i_2, i_3) + ... + c(i_{2m}, i_1)$

$= c(M_1) + c(M_2) \geq 2c(\mathfrak{M})$

Also ist $c(\mathfrak{M}) \geq \frac{1}{2}\, c(\tau_{opt})$ und damit

$c(\tau) \leq c(\mathfrak{T}) + c(\mathfrak{M}) \leq c(\tau_{opt}) + \frac{1}{2} c(\tau_{opt}) = 1{,}5\ c(\tau_{opt})$



Negative Ergebnisse:

Satz: Für kein $\varepsilon > 0$ gibt es einen ε-approximativen Polynomzeit-Algorithmus für TSP, es sei denn P = NP.

Beweis: Angenommen doch, d.h. $\exists \varepsilon > 0$ und einen Polynomzeit-Algorithmus $\mathfrak{A}_\varepsilon$ der TSP ε-approximativ löst;
daraus könnte man konstruieren:

$\mathfrak{A}_{HK} \longleftarrow$ polynomialzeit Algorithmus für HAM-Kreis Problem

$\mathfrak{A}_{HK}$: gegeben G = (V,E)
konstruiere Problemstellung für TSP
$d_{ij} = \begin{cases} 1 \text{ für (i,j)} \in E \\ 2 + \varepsilon n \text{ sonst} \end{cases}$
wende $\mathfrak{A}_\varepsilon$ darauf an

Behauptung: $\mathfrak{A}_\varepsilon$ liefert Rundreise der Länge n $\longleftrightarrow$ $\mathfrak{G}$ hat einen Hammilton Kreis

Beweis: $\Rightarrow$: Rundreise der Länge n benutzt nur Kanten mit Gewicht 1 also Kanten in E
$\rightarrow$ Sie ist Hammilton Kreis von $\mathfrak{G}$
$\Leftarrow$: Falls $\mathfrak{G}$ einen Hammilton Kreis hat, dann gibt es eine Rundreise der Länge n
$\rightarrow \mathfrak{A}_\varepsilon$ liefert eine Rundreise τ der Länge $\leq (1+\varepsilon)n$.
Angenommen in τ würde eine Kante des Gewichtes $2 + \varepsilon n$ benutzen
dann wäre $c(\tau) \geq 2 + \varepsilon n + n - 1 = (1 + \varepsilon)n + 1$
Widerspruch!
Also Annahme falsch, d.h. $\mathfrak{A}_\varepsilon$ benutzt nur Kanten vom Gewicht 1, d.h. Kanten $\in$ E
$\Rightarrow \tau$ ist Hammilton Kreis in $\mathfrak{G}$

3.5 Das Knapsack-Problem

KNAPSACK ist NP-vollständig !

- Gegeben: $\{c_1, .., c_n\}$, K
- Frage: $\exists L \subseteq \{1, .., n\}$, so daß $\sum_{i \in I} c_i = K$

Das Optimierungsproblem dazu lautet: Maximiere $\sum_{i \in L} c_i$, wobei $\sum_{i \in I} c_i \leq k$

Algorithm 4 $EXKNAPSACK(c_1, ..., c_n, K)$

1: $L_0 \leftarrow \{0\}$
2: **for** $i \leftarrow 1, n$ **do**
3: $\quad L_i \leftarrow MISCHE(L_{i-1}, L_{i-1} + c_i)$
4: $\quad L_i \leftarrow entferne\ alle\ Elemente\ > K\ aus\ L_i$
5: **end for**
6: $Gib\ Groesstes\ Element\ von\ L_n\ aus$

Exakte Lösung des Problems

Man kann leicht durch Induktion zeigen: L_i enthält alle Summen der Form $\sum_{i \in I} c_j$ wobei $I \subseteq \{1, .., i\}$, so daß deren Summe $\leq K$ sind. Daraus folgt die Korrektheit des Algorithmus. Der Algorithmus EXKNAP hat exponentielle Laufzeit.

Approximation des Problems

Herleitung des Algorithmus: Betrachte zusätzlich zu dem gegebenen Problem ein $\epsilon > 0$ als gegeben:

- Gesucht: ϵ-approximativer Algorithmus mit polynomieller Laufzeit
- Gegeben: Parameter δ
- Idee: 'Trimme' jede Liste L_i nach der Konstruktion, d.h. entferne unnötige Elemente.

L trimmen: So viele Elemente wie möglich entfernen, so daß in der neuen Liste L' für jedes entfernte $y \in L$ ein z existiert mit $(1 - \delta) \cdot y \leq z \leq y$

Algorithm 5 $TRIM(L, \delta)$

1: $L' \leftarrow (y_1)$
2: **for** $i \leftarrow 2, m$ **do**
3: $\quad$ **if** $letztes\ Element\ von\ L' < (1 - \delta) \cdot y_i$ **then**
4: $\quad\quad HAENGE\ y_i \in L\ AN\ L'\ AN$
5: $\quad$ **end if**
6: **end for**
7: $Gib\ L'\ aus$

wobei $L = (y_1, .., y_n), \quad y_1 \leq y_2 \cdots \leq y_n$

Diese Prozedur wird in den alten Algorithmus $EXKNAP$ eingefügt, woraus folgender Algorithmus resultiert:

Laufzeit und Qualität der Approximation: Sei

$$P_i = \{\sum_{j \in I} c_j | I \subseteq \{1, .., i\} \wedge \sum_{j \in I} c_j \leq K\}$$

Algorithm 6 $APKNAP(c_1, ..., c_n, K)$

1: $L_0 \leftarrow \{0\}$
2: **for** $i \leftarrow 1, n$ **do**
3: $\quad L_i \leftarrow MISCHE(L_{i-1}, L_{i-1} + c_i)$
4: $\quad L_i \leftarrow TRIM(L_i, \frac{\epsilon}{n})$
5: $\quad L_i \leftarrow entferne\ alle\ Elemente\ > K\ aus\ L_i$
6: **end for**
7: $Gib\ Groesstes\ Element\ von\ L_n\ aus$

Für jedes i gilt weiterhin, daß alle seine Elemente in P_i liegen, also wird in Zeile 7 eine Teilmengen-Summe $z \in P_n$ ausgegeben

Behauptung: $z > (1 - \epsilon) \cdot S_{max}$ wobei $S_{max} \leq K$ maximale Teilmengen-Summe.

Beweis: Zeile 4 des Algorithmus bewirkt: Zu jedem Element s der ursprünglichen Liste L_i existiert ein Element s' im neuen L_i mit

$$s \cdot (1 - \frac{\epsilon}{n}) \leq s' \leq S_{max}$$

Mittels Induktion über i lässt sich beweisen, daß

$$\forall s \in P_i \exists s' \in L_i : (1 - \frac{\epsilon}{n})^i \cdot s \leq s' \leq S_{max} \tag{1}$$

Insbesondere gilt bei

$$i = n : \underbrace{S_{max}}_{maxP_n} \geq \underbrace{z}_{maxL_n} \geq (1 - \frac{\epsilon}{n})^n \cdot S_{max}$$

Es gilt:

$$\forall n \geq 1 : (1 - \frac{\epsilon}{n})^n \geq (1 - \epsilon)$$

denn sei

$$f(x) = (1 - \frac{\epsilon}{n})^x = e^{x \cdot \ln(1 - \frac{\epsilon}{n})}$$

ist

$$f'(x) = e^{x \cdot \ln(1 - \frac{\epsilon}{x})} \cdot \left[\ln(1 - \frac{\epsilon}{x}) + x \cdot \frac{1}{1 - \frac{\epsilon}{x}} \cdot \frac{\epsilon}{x^2} \right] > 0, \forall x \geq 1$$

Also ist $f(x)$ monoton wachsend und somit ist $\forall x \geq 1$:

$$f(x) \geq f(1) = 1 - \epsilon$$

, damit ist z begrenzt durch:

$$S_{max} \geq z \geq (1 - \epsilon) \cdot S_{max} \tag{2}$$

Und damit liefert der Algorithmus die gewünschte ϵ-Approximation.

Laufzeit: Da alle Schritte in der Schleife $\mathcal{O}(|L_i|)$ sind, stellt sich die Frage, wie lange L_i werden kann? Für je zwei aufeinander folgende Elemente $s, t \in L_i$ gilt nach Zeile 4:

$$\frac{s}{t} \geq \frac{1}{1-\frac{\epsilon}{n}}$$

Siehe hierzu Formel (1). Somit lassen sich die Elemente in L_i wie folgt abschätzen:

$$L_i = \{0, \underbrace{s_1}_{\geq 1}, \underbrace{s_2}_{\geq \frac{1}{1-\frac{\epsilon}{n}}}, \underbrace{s_3}_{\geq \frac{1}{(1-\frac{\epsilon}{n})^2}}, \ldots, \underbrace{s_m}_{\geq \frac{1}{(1-\frac{\epsilon}{n})^{m-1}}} \leq K\}, |L_i| = m+1$$

Damit ergibt sich durch das größte Element eine obere Grenze für die Anzahl der Elemente m mit:

$$\begin{aligned} \Rightarrow \quad -(m-1)\ln(1-\tfrac{\epsilon}{n}) &\leq \ln K \\ m-1 &\leq \frac{-\ln K}{\ln(1-\frac{\epsilon}{n})} \\ m &\leq \frac{-\ln K}{\ln(1-\frac{\epsilon}{n})} + 1 \end{aligned}$$

Für $x > 0$ gilt: $\ln x \leq x - 1$. Damit ist die maximale Größe m der Listen durch $m \leq \frac{\ln K}{\frac{\epsilon}{n}} + 1 = n \cdot \frac{\ln K}{\epsilon} + 1$ gegeben.

Somit ergibt sich die Gesamtlaufzeit $\mathcal{O}\left(n^2 \cdot \frac{logK}{\epsilon}\right)$. Man sieht anhand der Formel, daß die Kodierung des Problems mit in die Größe des Problems eingeht. $\log K$ lässt sich durch die Kodierung der Summen, z.B. in Binärdarstellung, deuten.

Satz: Der Algorithmus APKNAPP ϵ-approximiert für jedes gegebene $\epsilon > 0$ das KNAPSACK-Optimierungsproblem für Eingaben $c_1, .., c_n, K$ in Zeit $\mathcal{O}\left(n^2 \frac{\log K}{\epsilon}\right)$

Nun kennen wir zwei Klassen von Approximierungsproblemen. Einerseits Probleme die sich wie KNAPSACK beliebig genau approximieren lassen, und andererseits Probleme wie das ΔTSP, von dem wir wissen, daß es nicht möglich ist es beliebig genau zu approximieren.

Definition 1: Ein Algorithmus heisst PTAS („Polynomial Time Approximation Scheme") für ein Problem P, falls er bei einer Eingabe A, wobei A Problemstellung für P ist, für jedes $\epsilon > 0$ eine ϵ-Approximation der Lösung liefert und seine Laufzeit für Konstante ϵ polynomiell in der Größe von A ist.

Beispiel: Der Algorithmus APKNAP für KNAPSACK $\mathcal{O}\left(n^2 \cdot \frac{\log K}{\epsilon}\right)$ ist polynomiell in der Eingabegröße, wenn ϵ fest gewählt ist. Aber auch $p(n) \cdot 2^{\frac{1}{\epsilon^n}}$ ist ein PTAS, wobei n die Eingabegröße ist. Die Abhängigkeit von ϵ wird weiter nicht berücksichtigt. Um dies weiter zu differenzieren:

Definition 2: Algorithmus heisst FPTAS „Fully PTAS", wenn er ein PTAS ist und seine Laufzeit auch nur polynomiell von $\frac{1}{\epsilon}$ abhängt

Bemerkung: Für das Optimierungsproblem von KNAPSACK existiert ein FPTAS, nämlich APKNAP

3.6 Geometrische Packungs- und Überdeckungsprobleme

Gegeben: Menge S von Punkten in der Ebene (z.B. kartesische Koordinaten), fester Radius r

Problemstellung: Überdecke S mit möglichst wenig Kreisscheiben vom Durchmesser D (oder durch Quadrate vorgegebener Größe)

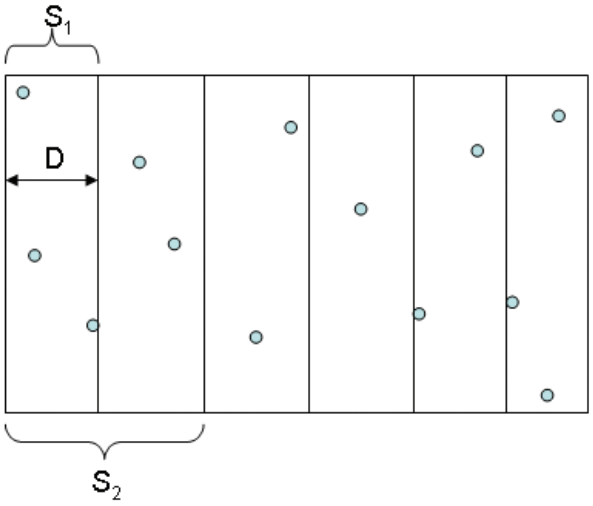
Lösung: Rechteck I enthält alle Punkte, zerschneide I in Streifen der Breite $D = 2 \cdot r$

Sei $\mathfrak{A}$ ein Algorithmus, der das Problem auf Streifen der Breite $\leq l \cdot D$ löst, $\mathfrak{A}_{S_i}$ der Algorithmus, der $\mathfrak{A}$ auf die Partition S_i anwendet und die Vereinigung bildet und sei $\mathfrak{S}_{\mathfrak{A}}$ der Algorithmus, der $\mathfrak{A}_{S_i}$ $i = 1, \ldots, l$ anwendet und dann das Minimum nimmt.
Für einen Algorithmus $\mathfrak{B}$ sei $r_{\mathfrak{B}}$ das Verhältnis:

$$\frac{|L_{\mathfrak{B}}|}{|L_{opt}|},$$

wobei $L_{\mathfrak{B}}$ Lösung des Algorithmus $\mathfrak{B}$ ist, und L_{opt} die optimale.
Dann gilt das Lemma:

$$r_{\mathfrak{S}_{\mathfrak{A}}} \leq r_{\mathfrak{A}} \cdot (1 + \frac{1}{l}) \tag{3}$$



Schema für den Algorithmus (Schiebe-Methode): Sei A irgendein Approximationsalgorithmus für die Streifen der Breite $\leq$ lD.
Wir definieren einen Algorithmus A_{γ_i}, der A auf die Streifen der Partition S_i

anwendet und deren Vereinigung bildet.
γ_A wendet alle A_{S_i} (i=1..l) an und bildet daraus das Minimum.
Der eigentliche Algorithmus folgt später.

Lemma : $r_{\gamma_A} \leq r_A(1 + \frac{1}{l})$, wobei $r_L = \frac{|Lösungenvon L|}{|optimale Lösungen|}$ (Damit wird das Problem reduziert auf die Lösung des einzelnen Streifens.)

Beweis: Zunächst betrachten wir die Anzahl der Lösungen für den Algorithmus A_{S_i}. Wegen der Arbeitsweise von A_{S_i} gilt:

$$|L_{A_{S_i}}| \leq r_A \sum_{j \in S_i} |OPT_j| \tag{4}$$

für i=1...l. Nun schauen wir uns die optimale Lösung an.
Sei OPT die otimale Lösung für γ überhaupt. Wir betrachten den j-ten Streifen bei Partition S_i und die Kreise von OPT, die den j-ten Streifen schneiden. Die Anzahl dieser Kreise ist größer als die optimale Lösung vom j-ten Streifen: Anzahl $\geq |OPT_j|$. Die Anzahl muß $\geq$ sein, weil sie nur irgendeine Lösung ist und somit auf jeden Fall $\geq$ der optimalen Lösung ist.
Wir wollen $\sum_{j \in S_i} |OPT_j|$ nach oben abschätzen:

$$|OPT| + |OPT^{(i)}| \geq \sum_{j \in S_i} |OPT_j| \tag{5}$$

$|OPT^{(i)}|$ ist dabei die Anzahl der Kreise der optimalen Lösung OPT, die bei der Partition S_i eine der vertikalen Geraden schneiden, die die Streifen begrenzen (und nur die, auch die Begrenzung tangierenden Kreise zählen nicht zu $|OPT^{(i)}|$).
Eine weitere Überlegung ist, dass alle $OPT^{(i)}$ disjunkt sind, d.h. wenn wir Streifen der Breite lD haben und eine Scheibe liegt in $OPT^{(i)}$, dann kann sie nicht auch noch in einem anderem $OPT^{(i)}$ liegen.
Damit bilden wir:

$$\sum_{1 \leq i \leq l} |OPT^{(i)}| \leq |OPT| \tag{6}$$

Was ist das Minimum der Partitionen? Es muss kleiner als der Durchschnitt der optimalen Lösungen sein.

$$
\begin{aligned}
min_{(i=1..l)} \sum_{j \in S_i} |OPT_j| &\leq \frac{1}{l} \sum_{1 \leq i \leq l} \sum_{j \in S} |OPT_j| \\
&\leq |OPT| + |OPT^{(i)}| (nachFormel2) \\
&\leq \frac{1}{l} \sum_{1 \leq i \leq l} (|OPT| + |OPT^{(i)}) \\
&\leq |OPT| + \frac{1}{l} |OPT| (nachFormel3) \\
&= |OPT|(1 + \frac{1}{l})
\end{aligned} \tag{7}
$$

$$
\begin{aligned}
|L_{\gamma_A}| &= min_{i=1..l} \\
&\leq min_{i=1..l} r_A \sum_{j \in S_i} |OPT_j| (mitFormel1) \\
&\leq r_A |OPT|(1 + \frac{1}{l}) (nachFormel4)
\end{aligned} \tag{8}
$$

Nun müssen noch beide Seiten durch $|OPT|$ dividiert werden und das Lemma ist bewiesen.

Wie lautet nun die Laufeit der Schiebe-Methode in Abhängikeit von dem angewendeten Algorithmus A?

$$
T(n) = \sum_{1 \leq i \leq l} T_{A_{S_i}}(n) \tag{9}
$$

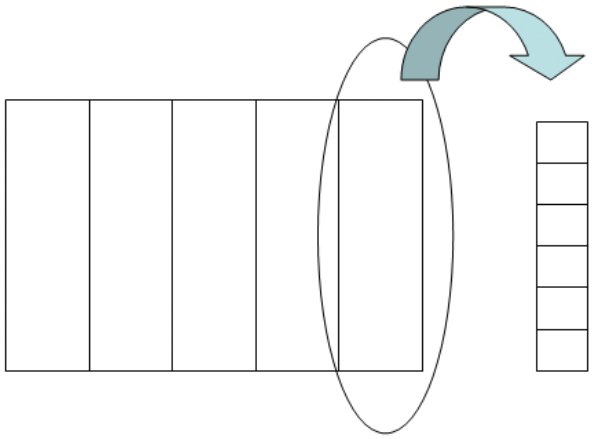
, wobei $T_{A_{S_i}}(n) = T_A(n_1) + T_A(n_2) + ...$, wobei die n_j die Anzahl der Punkte im j-ten Streifen sind.
Das ergibt für die Schiebe-Methode eine Laufzeit von $O(T_A(n))$, falls T_A folgende Eigenschaft besitzt: $T_A(n) + T_A(m) \leq T_A(n + m)$ (was durchaus eine vernünftige Annahme ist.)
Damit lautet die Gesamtlaufzeit:

$$
T_{\gamma_A}(n) = O(l T_A(n)) \tag{10}
$$

Anwendung: ϵ-approximative Überdeckung einer Punktmenge mit möglichst wenig Kreisscheiben.
Gegeben seien eine Menge S $\subset \mathbb{R}^2$ von n Punkten, ein Durchmesser D und ein $\epsilon > 0$. Finde dafür die ϵ-approximative Überdeckung von S mit möglichst wenig Kreisscheiben vom Durchmesser D.

Algorithmus ApÜK

1. Bestimme geeignetes $l \in \mathbb{N}$ und ein Rechteck I mit $S \subset I$.
2. Berechne eine approximative Lösung mit der Schiebe-Methode, wobei der zugrunde liegende Algorithmus A die
3. Schiebe-Methode in vertikaler Richtung ist, deren zugrunde liegender Algorithmus folgender ist:
4. Bestimme die optimale Überdeckung der Punkte in einem $lDxlD$-Quadrat.

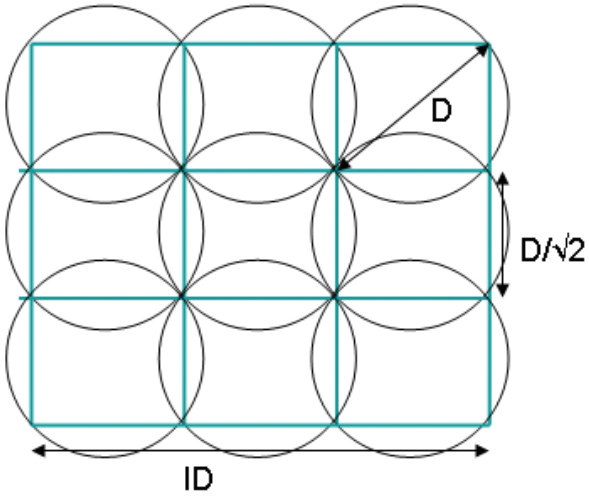


Einzelheiten:
für (4): Ein optimaler Algorithmus für die Quadrate c sieht so aus: Jedes Quadrat wird mit einer Kreisscheibe vom Durchmesser D so überdeckt, dass die Kreisscheibe durch alle 4 Eckpunkte geht.

Für die Anzahl der Kreisscheiben ergibt sich $k \leq (\frac{ld}{\frac{D}{\sqrt{2}}})^2 = O(l^2)$

Folgerungen:

1. Die optimale Überdeckung hat damit $\leq k = O(l^2)$ Kreisscheiben.
2. Es gibt eine Überdeckung, bei der jeder Kreis durch mindestens zwei Punkte geht. Denn angenommen, es gilt nicht. Dann kann man einen Kreis nach links oder rechts verschieben, so dass er auf einem Punkt liegt. Dann wird er auf diesem Punkt festgehalten und um ihn rotiert, bis der Kreis auf einem zweiten Punkt liegt. Allerdings gilt dies nur für Punkte, die einen Abstand $\leq D$ haben, ansonsten ist es unmöglich, dass zwei Punkte auf dem Kreis liegen.



Damit gilt folgendes:
Falls $n' = |S \cap c|$ (Anzahl der Punkte im Quadrat), gibt es $2\binom{n'}{2}$ Kandidatenkreise, die in einer optimalen Überdeckung vorkommen können, denn es gibt $\binom{n'}{2}$ Punktpaare und ≤ 2 Kreise vom Durchmesser D durch jedes Punktepaar (1 Kreis für den Abstand $= D$, 2 Kreise für den Abstand $< D$ und keinen Kreis für den Abstand $> D$)
Draus wählen wir $\leq k$ auf alle möglichen Arten aus und prüfen, ob die Auswahl eine Überdeckung darstellt.

$$\begin{aligned} \sum_{1 \leq i \leq k} \binom{2\binom{n'}{2}}{j} &\leq \sum_{1 \leq i \leq (n'^2)^j} \\ &= \frac{(n'^2)^{k+1} - 1}{n'^2 - 1} - 1 (geometrische Reihe) \\ &= O(n'^{2k}) \subset O(n'^{O(l^2)}) \end{aligned} \tag{11}$$

Für jede solche Auswahl muß man noch feststellen, ob sie alle Punkte überdeckt (jeden Punkt mit jedem Kreis vergleichen).
$O(n'k)Zeit = O(n'l^2)Zeit$ je Auswahl. Und insgesamt: $O(n'l^2 n'^{O(l^2)}) = O(l^2 n'^2 O(l^2))$

Nochmal der Algorithmus:

1. Rechteck I, $S \in I$
2. berechne approximative Lösung mit Schiebemethode, wobei
3. zugrunde liegender Algorithmus $\mathfrak{A}$ die Schiebemethode in vertikaler Richtung ist. Daraus resultieren Quadrate der Kantenlänge lD
4. bestimme optimale Überdeckung auf den Quadraten (Zellen). Jeder Kreis überdeckt 2 Punkte. Punkte mit Abstand größer als D zu jedem anderen Punkt werden entfernt und mit einem einzelnen Kreis überdeckt.

Frage: Wie stark hängt die Laufzeit von ϵ ab?

Schritt 4: Sei $\mathfrak{A}_4$ der Aufruf des Algorithmus auf den Quadraten (Zellen), so hat dieser eine Laufzeit von $T_{\mathfrak{A}_4}(\bar{n}) = l^2 \cdot \bar{n}^{O(l^2)}$. (Siehe weiter oben)

Schritt 3: Sei $\mathfrak{A}_3$ der Algorithmus auf einem Streifen, also $\mathfrak{A}_3 = \gamma_{\mathfrak{A}_4}$.

Laufzeit: $T_{\mathfrak{A}_3}(n') = O(l \cdot T_{\mathfrak{A}_4}(n')) = O(l^3 \cdot n'^{O(l^2)})$

Approximationsgüte: $r_{\mathfrak{A}_3} \leq r_{\mathfrak{A}_4} \cdot (1+\frac{1}{l})$ nach Lemma, wobei $r_{\mathfrak{A}_4} = 1$ ist, da $\mathfrak{A}_4$ optimale Überdeckung in den Zellen findet. $\Rightarrow r_{\mathfrak{A}_3} \leq 1 + \frac{1}{l}$

Schritt 2: Sei $\mathfrak{A}_2$ der Algorithmus für diesen Schritt, $\mathfrak{A}_2 = \gamma_{\mathfrak{A}_3}$

Laufzeit: $T_{\mathfrak{A}_2}(n) = O(l \cdot T_{\mathfrak{A}_3}(n)) = O(l^4 \cdot n^{O(l^2)})$

Approximationsgüte: $r_{\mathfrak{A}_2} \leq r_{\mathfrak{A}_3} \cdot (1 + \frac{1}{l}) \leq (1 + \frac{1}{l})^2$

Für die ϵ-Approximation gilt: Wähle l so, dass $(1 + \frac{1}{l})^2 \leq 1 + \epsilon$ ist.
$\Rightarrow 1 + \frac{1}{\epsilon} \leq \sqrt{1+\epsilon}$
$\Rightarrow l \geq \frac{1}{\sqrt{1+\epsilon}-1}$

Für $\epsilon \in [0,1]$ gilt: $1 + 0.4\epsilon \leq \sqrt{1+\epsilon} \leq 1 + \frac{1}{2}\epsilon$
$\Rightarrow$ Ein hinreichendes $l \geq \frac{1}{1+0.4\epsilon-1} = \frac{2.5}{\epsilon}$
Die resultierende Laufzeit ist damit $O(\frac{1}{\epsilon^4} \cdot n^{O(\frac{1}{\epsilon^2})})$

Bsp: für $\epsilon = \frac{1}{10}$ und einer Konstanten von 5 wäre die Laufzeit $O(n^{500})$. Also polynomiell :-).

Zsfg: Der Algorithmus ApÜK hat für konstante $\epsilon > 0$ polynomielle Laufzeit. Diese ist aber nicht polynomiell in $\frac{1}{\epsilon}$. Darum:

Satz: Algorithmus ApÜK ist ein PTAS (aber kein FPTAS) für minimale Überdeckung einer Punktemenge durch Kreisscheiben.

Anmerkung:

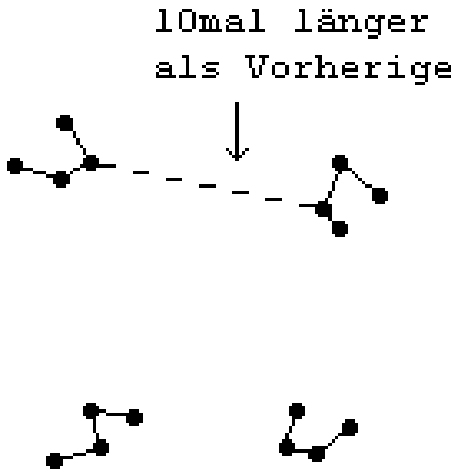
- Überdeckung durch gleichgroße Quadrate, Rechtecke und beliebige andere Körper mit gleicher Methode möglich.
- für höhere Dimensionen $n > 2$ ist Verallgemeinerung möglich. Beispielsweise statt Kreise Kugeln, etc.

3.7 Clustering Probleme

intuitive Lösungsidee:

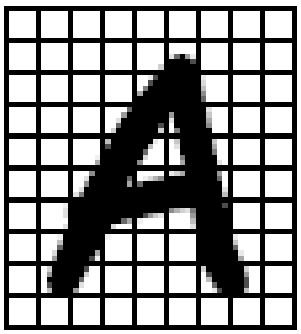
- Kruskal-MST-Algorithmus anwenden
- wenn man nun plötzlich eine Kante erhält, die 10-mal länger ist als die zuvorige, dann Abbrechen
- nun hat man einen spannenden Wald $\Rightarrow$ Cluster.

- im schlechtesten Fall jedoch sind die Punkte derart gleichverteilt, dass ein zusammenhängender Graph entsteht.



Anwendung:

- Mustererkennung (OCR). Aus einer Grafik wird ein ca. 20-dimensionaler Vektor erstellt. Ein bestimmter Vektor ergibt dann einen Punkt im Raum als Prototyp. Ähnliche Zeichen bzw. Vektoren bilden dann einen Cluster.



3.8 MinMax-Radius-Clustering (euclidian k-center problem, MRC)

Geg: $S = s_1, s_2, \ldots, s_n \in R^2, k \in N$

Finde: Partition $S = S_1 \cup S_2 \ldots \cup S_k$, die $max_{i=1,\ldots,k} rad(S_i)$ minimiert. Wobei $rad(S_i)$ der minimale Radius einer Kreischeibe ist, welche S_i überdeckt. Das Problem ist NP-schwer.

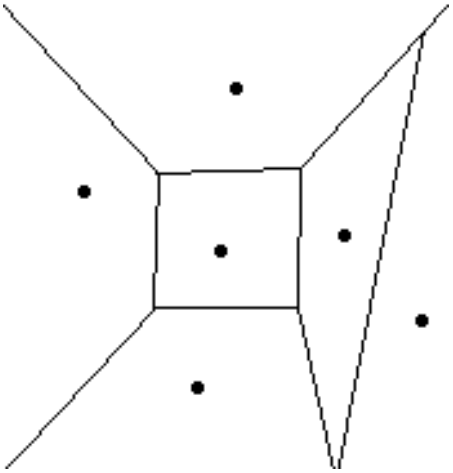
Approximationsalgorithmus von Gonzales: proc WPC(s,k) (weitester Punkt Clusterung)

(1) s_1 :=beliebiger Punkt aus S
(2) $C := s_1$
(3) for $i = 1$ to k do
(4) s_i = am weitesten zu C entfernter Pkt., wobei $d(s, C) = min_{x \in S_i} d(x, s_i)$
(5) $C = C \cup s_i$
(6) for each s in S do
ordne s dem nächstgelegenem Pkt. in C zu

Laufzeit: $T_proc = sum_{i=2}^{k}(i \cdot n) + O(n \cdot k) = O(n \cdot k^2 + n \cdot k) = O(n \cdot k)$
Da k n annehmen kann, erhalten wir $O(n^3)$

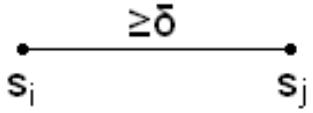
es geht besser mit Methoden der algorithmischen Geometrie:

- Voronoi-Diagramm in $O(n \log n)$ berechenbar.
- damit Clustering-Algorithmus auch in $O(n \log n)$ berechenbar.



Satz: WPC ist ein 1-approximativer Algorithmus für MR-Clustering, dh. es liefert eine Partition $S_1, \ldots, S_k$, für die $max_{i=1,\ldots,k} rad(S_i)$ um höchstens den Faktor zwei schlechter ist, als das Minimum.

Beweis: angenommen, der Algorithmus würde noch einen weiteren Punkt s_{k+1} auswählen. s_{k+1} maximiert $\delta = min_{i=1,\ldots,k} d(s_{k+1}, s_i)$ (d = euklidischer Abstand).
Es gilt: $d(s_i, s_j) \geq \delta$, für $1 \leq i, j \leq k$ und $i <> j$.
In jedem k-Clustering müssen nun mindestens zwei der Punkte $s_1, \ldots, s_k$ im gleichen Cluster S_l sein (Schubfachprinzip).



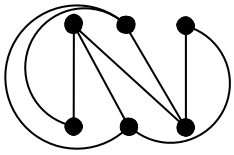
Damit ist der Durchmesser von $S_l \geq \delta$ und daraus folgt: $rad(S_l) \geq \frac{\delta}{2}$ und WPC findet ein Clustering mit Radius $\leq \delta$.

Satz: ϵ-Approximation für MR-Clustering ist NP-schwer für $\epsilon \leq 0.8027$.

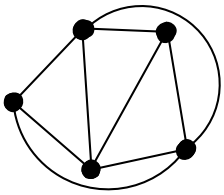
3.9 Beweis (ohne einige Einzelheiten):

Vorgehensweise: Wir machen zunächst eine Reduktion von Ük („überdeckende Knotenmenge") auf MiniMax-Radius-Clustering bei 3-regulären planaren Graphen. Damit zeigen wir, dass MiniMax-Clustering NP-schwer ist. Im zweiten Schritt zeigen wir dann, dass Approximationslösungen mit $\epsilon < 0.8027$ gleich der Optimallösung sind, und damit ebenfalls NP-schwer.

Kurzer Exkurs Graphentheorie: In einem 3-regulären Graphen hat jeder Knoten genau den Grad 3. Ein planarer Graph kann in die Ebene eingebettet werden, ohne dass sich 2 Kanten kreuzen. Beispiele für nicht-planare Graphen: $K_{3,3}$ (von Kneipenabenden bekannt als Wasserwerk-Elektrizitätswerk-Gaswerk-Problem) und K_5 (siehe Abbildungen):



$K_{3,3}$ - vollständiger, nicht planarer Graph

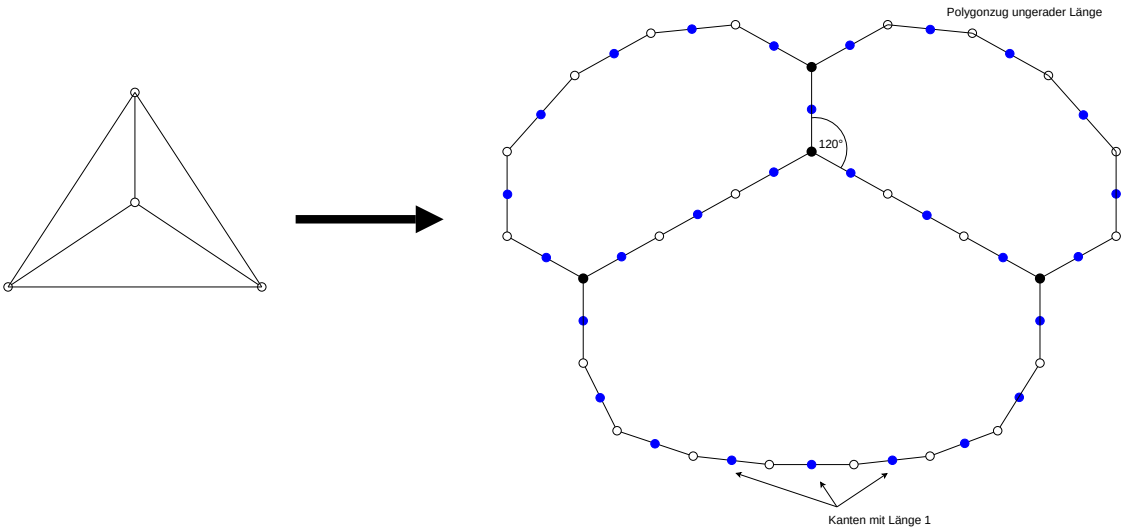


K_5 - vollständiger, nicht planarer Graph

Konstruktion: Wir stellen ohne Beweis fest dass Ük, eingeschränkt auf 3-reguläre planare Graphen, bereits NP-vollständig ist. Gegeben sei also ein 3-regulärer planarer Graph $G = (V, E)$. Wir betten ihn in die Ebene ein, so dass folgendes gilt:

- Jede Kante $e \in E$ wird durch einen Polygonzug P_e ungerader Länge (d.h. bestehend aus ungerade-vielen Strecken) dargestellt.
- Die Länge der einzelnen Strecken des Polygonzugs ist 1.
- Die Wege (Polygonzüge) treffen sich bei Knoten in 120^o-Winkeln
- Kanten, die keine Knoten gemeinsam haben, sind „weit auseinander"

Beispiel: K_4 ist 3-regulär und planar.



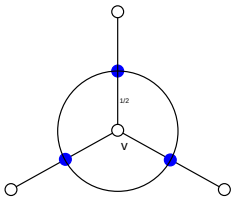
Reduktion: Sei S die Menge der Mittelpunkte der Kanten aller Polygonzüge. Dann hat S ein k'-Clustering mit

$$k' = k + \frac{1}{2} \sum_{e \in E} (|Pe| - 1)$$

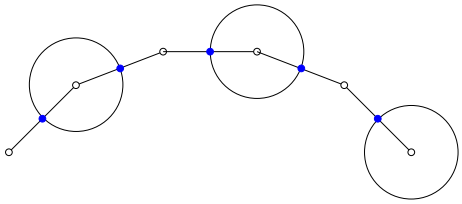
genau dann, wenn G eine Überdeckung der Größe k hat.

Beweis:

- ($\Leftarrow$) Für jeden Knoten aus der überdeckenden Knotenmenge wähle ein 3-Cluster wie folgt:

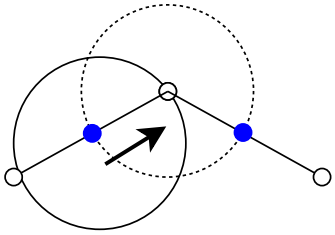


Der Kreis um V mit Radius $\frac{1}{2}$ enthält 3 Knoten aus S. Auf diese Art wird von jedem Kantenzug P_e mindestens ein Punkt aus S überdeckt. Die restlichen $\leq |P_e| - 1$ werden mit $\frac{1}{2}(|P_e| - 1)$ Kreisscheiben paarweise überdeckt.

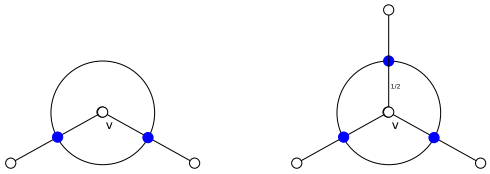


Also existiert insgesamt ein Clustering mit $k + \frac{1}{2}\sum_{e \in E}(|Pe| - 1)$ Kreisen.

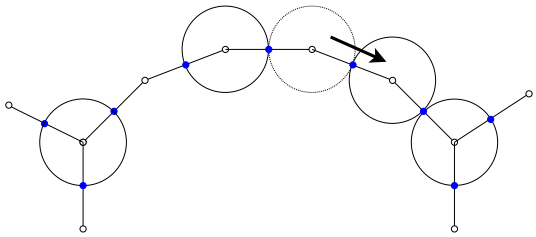
- ($\Rightarrow$) Angenommen, S hat ein solches Clustering (also eine Überdeckung mit Kreisscheiben). Dann folgt:
  - Jedes Cluster hat höchstens 3 Punkte (denn nicht-inzidente Kanten sind ,,weit auseinander").
  - Es gibt ein gleich großes Clustering, wobei jeder Kreis mindestens 2 Punkte überdeckt. (Denn: Würde ein Kreis nur einen Punkt überdecken, so könnte man ihn verschieben und rotieren, so dass er ebenfalls einen Nachbarpunkt überdeckt. Wenn ein Kreis keine Punkte überdeckt, kann man diesen weglassen)



  - Es gibt ein gleich großes Clustering, wo jeder Kreis entweder ein 2-Kreis oder ein 3-Kreis ist (siehe Bild).



  - Es gibt ein nicht größeres Clustering, wo sich keine 2 Kreise berühren oder schneiden.



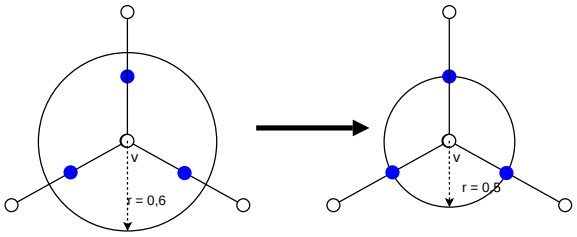
Also kann man annehmen: Auf jedem P_e sind $\frac{1}{2}(|P_e| - 1)$ disjunkte 2-Kreise und einer oder zwei 3-Kreise. Es ist nicht möglich, dass wir keine 3-Kreise haben, denn die Anzahl der Strecken ist ja ungerade.

Da die Anzahl der Kreise insgesamt $k + \frac{1}{2}\sum_{e \in E}(|Pe| - 1)$ beträgt, gibt es $\frac{1}{2}\sum_{e \in E}(|Pe| - 1)$-viele 2-Kreise und k-viele 3-Kreise. Deren Mittelpunkte bilden damit eine Überdeckung.

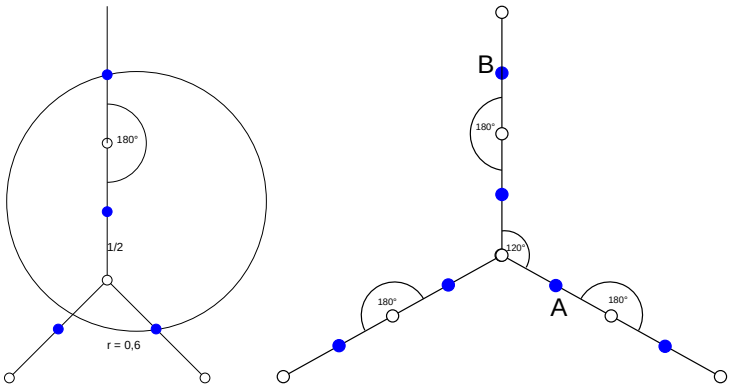
□

Zwischenergebnis: Damit gilt: G hat eine Überdeckung der Größe k genau dann, wenn S ein $\frac{1}{2}$-Clustering der Größe k' hat. Es folgt: MiniMax-Radius-Clustering ist NP-schwer. Wir müssen nun noch zeigen, dass die Approximation für $\epsilon < 0.8027$ ebenfalls NP-schwer ist.

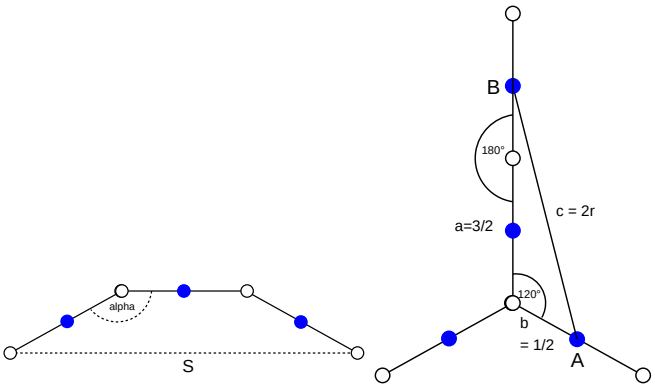
Beweis: Betrachte bei unserer Konstruktion den Cluster-Radius $> \frac{1}{2}$. Ein Clustering mit Radius z. B. 0.6 lässt sich auf Radius $\frac{1}{2}$ schrumpfen, ohne dass sich etwas ändert, denn:



Die Situation ändert sich erst, wenn der Radius so groß ist, dass andere Punktkonfigurationen als mit Radius $\frac{1}{2}$ überdeckt werden können.



Das ist erst dann der Fall, wenn der Radius r so groß wird, dass Punkte in der Konfiguration A, B überdeckt werden können. Dazu braucht man in der Konstruktion die zusätzliche Forderung, dass bei jedem der Sterne ein 180^o-Winkel zwischen den ersten beiden Kanten vorliegt (siehe Bild). Die Konstruktion lässt sich trotzdem noch genau so machen.



Sei nun α so groß, dass s größer wird als der Abstand zwischen A und B. Jeder Winkel $< 180^o$ ermöglicht die Konstruktion. Damit diese Situation eintritt, muss für den Kreisradius r gelten:

$$2r \geq c$$

Nun gilt nach Pythagoras:

$$c^2 = a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos\alpha = \left(\frac{3}{2}\right)^2 + \left(\frac{1}{2}\right)^2 - 2 \cdot \frac{3}{4}\cos(120^o) = \frac{13}{4}$$

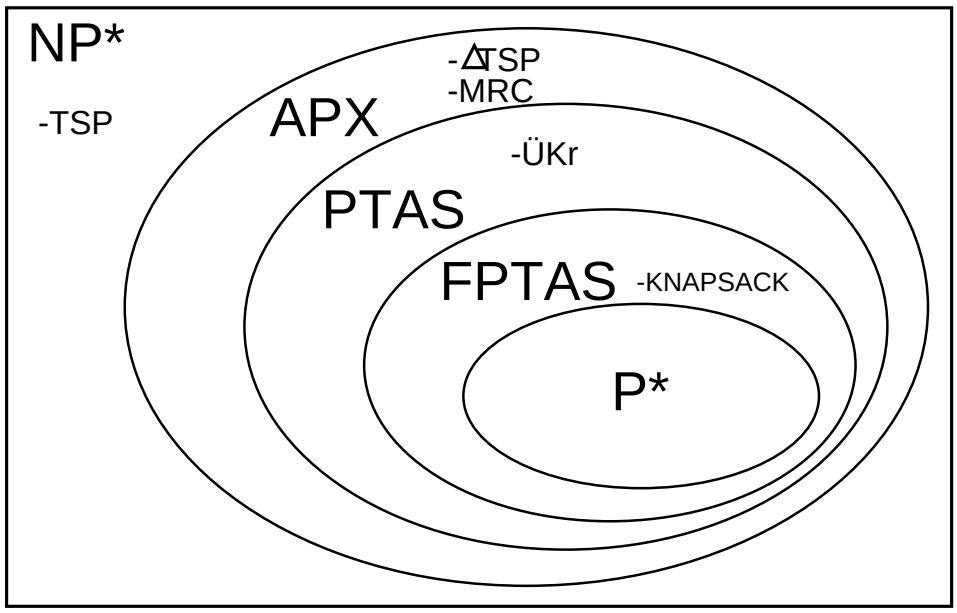
Eingesetzt ergibt das

$$2r \geq \sqrt{\frac{13}{4}} = 1.8027...$$

Wenn eine Approximation mit dem Faktor $\leq 1.8027...$ möglich wäre, dann wäre die Lösung auch schon eine, die mit dem Radius $\frac{1}{2}$ ebenfalls möglich ist, und darauf lässt sich Ük reduzieren.

□

Überblick über Approximationsmöglichkeiten



NP* - Optimierungsprobleme, deren Entscheidungsproblem in NP liegt.

APX - mit konstantem Faktor approximierbar

PTAS - Probleme für die ein PTAS existiert

FPTAS - Probleme für die ein FPTAS existiert

P* - Optimierungsprobleme, deren Entscheidungsproblem in polynomieller Zeit lösbar ist

4 Algebraische und arithmetische Algorithmen

4.1 Matrizenoperationen

4.1.1 Matrizenmultiplikation

4.1.2 Matrizeninversion

Zurückführung der Matrizeninversion auf Matrizenmultiplikation (Wiederholung)

$$
\begin{aligned}
B &= \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\
\Rightarrow B^{-1} &= \begin{pmatrix} I_{\frac{n}{2}} & -B_{11}^{-1} \cdot B_{12} \\ 0_{\frac{n}{2}} & I_{\frac{n}{2}} \end{pmatrix} \cdot \begin{pmatrix} B_{11}^{-1} & 0_{\frac{n}{2}} \\ 0_{\frac{n}{2}} & D^{-1} \end{pmatrix} \cdot \begin{pmatrix} I_{\frac{n}{2}} & 0_{\frac{n}{2}} \\ -B_{21} \cdot B_{11}^{-1} & I_{\frac{n}{2}} \end{pmatrix}
\end{aligned}
$$

$$\text{mit } D = B_{22} - B_{21} \cdot B_{11}^{-1} \cdot B_{12}$$

Laufzeitanalyse Anzahl der Operationen $\mathcal{I}(n)$:

$$\mathcal{I}(1) = 1$$

Für unseren Algorithmus mit zwei rekursiven Aufrufen (B_{11}^{-1}, D^{-1}) ergibt sich folgende Rekursionsgleichung:

$$I(n) = 2\, I(\frac{n}{2}) + \gamma \quad .$$

γ repräsentiert hierbei die Kosten für alle stattfindenden Matrizenmultiplikationen auf den $n \times n$ und $\frac{n}{2} \times \frac{n}{2}$ Matrizen.

- 4 Multiplikationen auf $n \times n$ Matrizen
  - $B = A^T \cdot A$
  - $B^{-1} \cdot A^T = A^{-1}$
  - $B^{-1} = \begin{pmatrix} \quad \end{pmatrix} \cdot \begin{pmatrix} \quad \end{pmatrix} \cdot \begin{pmatrix} \quad \end{pmatrix}$
- 3 Multiplikationen auf $\frac{n}{2} \times \frac{n}{2}$ Matrizen
  - $-B_{11}^{-1} \cdot B_{12}$
  - $-B_{21} \cdot B_{11}^{-1} = \delta$
  - $D = B_{22} + \delta \cdot B_{12}$
- 1 Subtraktion

Insgesamt inklusive der Subtraktionen:

$$\mathcal{I}(n) = 2\, \mathcal{I}(\frac{n}{2}) + 4\, \mathcal{M}(n) + 3\, \mathcal{M}(\frac{n}{2}) + \frac{n^2}{4} \quad .$$

Hierbei steht $\mathcal{M}(n)$ für die Komplexität (Anzahl der Operationen) der eingesetzten Matrizenmultiplikationen.

Wir nehmen an:

$$\mathcal{M}(\frac{n}{2}) \leq \frac{1}{4}\,\mathcal{M}(n) \quad .$$

Diese Annahme ist sinnvoll, denn sie gilt z.B. für $\mathcal{M}(n) = c\,n^{\alpha}$, mit $\alpha \geq 2$.

$$\begin{aligned}
\mathcal{I}(n) &\leq 2\,\mathcal{I}(\frac{n}{2}) + c \cdot \mathcal{M}(n) \quad \text{, für eine Konstante } c \\
&\leq 4\,\mathcal{I}(\frac{n}{4}) + 2\,c\,\mathcal{M}(\frac{n}{2}) + c \cdot \mathcal{M}(n) \\
&\vdots \quad (k-2) \text{ Schritte weiterentwickelt} \\
&\leq 2^k\,\mathcal{I}(\frac{n}{2^k}) + 2^{k-1}\,c \cdot \mathcal{M}(\frac{n}{2^{k-1}}) + \cdots + c\,\mathcal{M}(n) \\
&\quad \text{mit } \mathcal{M}(\frac{n}{2}) \leq \frac{1}{4}\,\mathcal{M}(n) \text{ folgt} \\
&\leq 2^k\,\mathcal{I}(\frac{n}{2^k}) + 2^{k-1} \cdot \frac{1}{4^{k-1}}\,c \cdot \mathcal{M}(n) + \cdots + c\,\mathcal{M}(n) \\
&\quad \text{ausklammern von } c \cdot \mathcal{M}(n) \\
&\leq 2^k\,I(\frac{n}{2^k}) + c \cdot \mathcal{M}(n) \cdot \underbrace{\sum_{j=0}^{\infty} 2^{-j}}_{\text{strebt gegen } 2} \\
&\quad \text{Unter der Annahme, dass } n \text{ eine Zweierpotenz ist,} \\
&\quad \text{gilt: } k = \log n \text{ oder } n = 2^k. \\
&\leq n \cdot \mathcal{I}(1) + 2\,c\,\mathcal{M}(n) \\
&= O(\,\mathcal{M}(n)\,)
\end{aligned}$$

Die Voraussetzung dafür, dass n eine Zweierpotenz ist, kann durch das Auffüllen der Matrix mit Nullen erreicht werden.

Die Determinante Bestimmung der Determinanten durch

$$det\ B = det\ B_{11} \cdot det\ D \quad .$$

Der Algorithmus hat zwei rekursive Aufrufe. Bei der Berechnung der Matrix D werden eine Subtraktion und zwei Multiplikationen auf $\frac{n}{2} \times \frac{n}{2}$ Matrizen durchgeführt. Die Multiplikation der beiden Determinanten kostet konstant viel.

$$\begin{aligned}
\mathcal{D}(1) &= 0 \\
\mathcal{D}(n) &= 2\,\mathcal{D}(\frac{n}{2}) + 2\,\mathcal{M}(\frac{n}{2}) + \frac{n^2}{4} + 1 \\
&\Downarrow \quad \text{(analog zum Invertieren)} \\
\text{Lsg} : \mathcal{D}(n) &= \mathcal{O}(\,\mathcal{M}(n)\,)
\end{aligned}$$

$$\begin{aligned} \text{bestimmt } det\ B &= det\ A \cdot det\ A^T \\ &= (\ det\ A\ )^2 \\ &\quad \downarrow \\ &\quad \text{Daraus berechnet sich } det\ A \text{ bis auf das Vorzeichen.} \end{aligned}$$

Dies funktioniert alles nicht für nicht invertierbare Matrizen, also wenn die Determinante 0 ist.

$$\left.\begin{array}{l} det\ A = 0 \\ A^{-1} \text{ existiert nicht} \end{array}\right\} \Leftrightarrow \text{der Alg. an einer Stelle eine Division durch 0 versucht.}$$

Dieses wird nur erkannt bei einer eindeutigen Zahldarstellung, jedoch nicht, wenn Gleitkommazahlen verwendet werden.

Satz: Sind $n \times n$ Matrizen mit $\mathcal{M}(n)$ Operationen multiplizierbar, so kann man auch Inverse und Determinanten in $\mathcal{O}(\ \mathcal{M}(n)\ )$ Zeit berechnen, falls $\mathcal{M}(\frac{n}{2}) \leq \frac{1}{4}\ \mathcal{M}(n)$.

Die Laufzeit des naiven Algorithmus (rekursive Entwicklung nach der ersten Spalte) zur Berechnung der Determinanten ergibt sich mit:

$$\mathcal{D}(n) = n \cdot \mathcal{D}(n-1) + \mathcal{O}(n) \qquad \rightsquigarrow \qquad \Omega(n!)$$

und liefert damit eine exponentielle Laufzeit. Daher ist Standartmethode für die Determinanten-Berechung die Gauss Elimination.

$$det \begin{pmatrix} \alpha_{11} & * & \dots & * \\ 0 & \alpha_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & * \\ 0 & \dots & 0 & \alpha_{nn} \end{pmatrix} = \alpha_{11} \cdot \alpha_{22} \cdot \ldots \cdot \alpha_{nn}$$

Boolsche Matrizenmultiplikation

- Matrizen über $0, 1$
- Operationen: $\vee, \wedge$

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

Da die Laufzeit für den straight-forward Ansatz $\mathcal{O}(n^3)$ ist, würde man gerne die schnellen Algorithmen zur Matrizenmultiplikation verwenden. Dies geht aber nicht direkt, da $(0, 1, \vee, \wedge)$ kein Ring ist und die Algorithmen dies aber voraussetzen. Um sie dennoch verwenden zu können führt man den Algorithmus zurück auf die Arithmetik über $\mathbb{Z}$ und berechnet

$$\hat{c}_{ij} = \sum_{i=1}^{n} a_{ik} \cdot b_{kj} \qquad .$$

Da $\hat{c} \in \mathbb{Z}$ gilt, benutzen wir für die Umsetzung:

$$c_{ij} = \begin{cases} 0 & \text{für } \hat{c}_{ij} = 0 \\ 1 & \text{für } \hat{c}_{ij} \neq 0 \end{cases}$$

Da $(\mathbb{Z}, \cdot, +)$ ein Ring ist, kann die boolsche Matrizenmultiplikation mit $\mathcal{M}(n)$ Operationen berechnet werden, wobei $\mathcal{M}(n)$ die Komplexität der Multiplikation zweier $n \times n$ Matrizen auf dem Ring ist.

Wieviele Boolsche Operationen braucht man, für Add./ Mul./ Sub. auf $\mathbb{Z}$?
Alle $\hat{c}_{ij}$ sind ganze Zahlen $\leq n$, d.h. statt über $\mathbb{Z}$ kann auch über $\mathbb{Z}_{n+1} = 0, \ldots, n$ gerechnet werden. Die c_{ij} bleiben gleich. Alle Zahlen in $\mathbb{Z}_{n+1}$ können binär mit $\lceil \log n \rceil$ Bits codiert werden. Die Operationen $+, -, \cdot$ werden durch boolsche Operationen realisiert.

Algorithmus insgesamt: Wir berechnen die $\hat{c}_{ij}$ im Ring $\mathbb{Z}_{n+1}$ und die einzelnen Operationen führen wir binär durch.

Insgesamt benötigte Operationen:

$$i\mathcal{M}(n) \cdot m(\lceil \log n \rceil)$$

Wobei $m(k)$ die Anzahl der boolschen Operationen ist die benötigt werden zur Addition/ Subtraktion/ Multiplikation von k Bit Zahlen ($k = \lceil \log n \rceil$). Die Schulmethode zur binären Multiplikation braucht

$m(k) = \mathcal{O}(k^2)$ (es geht auch schneller, das jedoch später).

Also funktionieren Boolsche Matrizenmultiplikationen in

$$\begin{array}{ll} & \mathcal{O}(\ \mathcal{M}(n)\ ) \cdot m(\lceil \log n \rceil) \\ \text{z.B.} & \mathcal{O}(n^{2,376} \cdot (\log n)^2) \end{array}$$

4.2 Polynommultiplikation, diskrete Fourier-Transformation

Wir betrachten einen Körper (oder Ring) K und Polynome mit einer Variablen $X : K[X]$, d.h. Ausdrücke der Form:

$$P(X) = a_0 + a_1 \cdot X^1 + \ldots + a_n \cdot X^n$$

$$a_0, a_1, \ldots, a_n \in K$$

Die Addition erfolgt komponentenweise (einfach).
Multiplikation:

$$\begin{aligned} Q(X) &= b_0 + b_1 \cdot X^1 + \ldots + b_n \cdot X^n \\ P(X) \cdot Q(X) &= c_0 + c_1 \cdot X^1 + \ldots + c_{2n} \cdot X^{2n} \\ &\text{mit } c_i = \begin{cases} \sum_{j=0}^{i} a_j \cdot b_{i-j} & \text{für } i \leq n \\ \sum_{j=i-n}^{n} a_j \cdot b_{i-j} & \text{für } i > n \end{cases} \end{aligned}$$

Man kann die Polynome auch als Vektoren auffassen.

$$(a_0, a_1, \ldots, a_n) \otimes (b_0, b_1, \ldots, b_n) = (c_0, c_1, \ldots, c_{2n})$$

Die Operation $\otimes$ heisst Konvolution oder Faltung.

Wie schnell (Anzahl arithm. Operationen) kann man diese berechnen? Direkt nach der Formel:

$$\begin{array}{rcl} 2 \cdot i + 1 & & \text{Add/Mul für } i \leq n \\ 2 \cdot (i - n) + 1 & & \text{Add/Mul für } i > n \\ \hline 2 \cdot (n+1)^2 + 1 & = & \mathcal{O}(\ n^2\ ) \end{array}$$

Diese Laufzeit wollen wir verbessern. Dafür wählen wir eine andere Möglichkeit als den Vektor der Koeffiezienten (z.B. $(0,0,1) = 0 + 0 \cdot x^1 + 1 \cdot x^2 = x^2$), um ein Polynom zu repräsentatieren. Seien:

$$x_0, x_1, \ldots, x_n \in K \text{ feste Werte mit } x_i \neq x_j \text{ für } i \neq j$$

dann ist ein Repräsentation von P gegeben mit:

$$P : (y_0, \ldots, y_n) \text{ mit } y_i = P(x_i) \qquad ,$$

also Werte von P an den Stützstellen $x_0, \ldots, x_n$. So kann z.B. eine Parabel (siehe Abbildung 7) durch drei Punkte festgelegt werden und jede Gerade durch zwei Punkte.

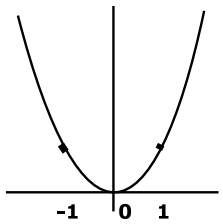


Abbildung 1: Die Parabel ist durch 3 Punkte eindeutig bestimmt

4.2.1 Umrechnung von Koeffizienten- in Wertedarstellung (Auswertung, Evalutation)

$$\begin{array}{rcl} (a_0, \ldots, a_n) & \longmapsto & (y_0, \ldots, y_n) \\ \text{Koeffiezientendarstellung} & & \text{Wertedarstellung} \\ \text{also: } A \cdot \vec{a} & = & \vec{y} \end{array}$$

Diese lineare Abbildung A heisst „van-der-Monde“-Matrix.

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ \vdots \\ \vdots \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ \vdots \\ \vdots \\ y_n \end{pmatrix}$$

$$detA = \prod_{i>j} (x_i - x_j) \neq 0$$

falls alle x_i verschieden sind. Sobald dieses gilt, ist A invertierbar.

4.2.2 Umrechung von Werte- in Koeffizientendarstellung (Interpolation)

$$\begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} = A^{-1} \cdot \begin{pmatrix} y_0 \\ \vdots \\ y_n \end{pmatrix}$$

4.2.3 Algorithmus zur Polynommultiplikation

Angenommen: es gibt Stützstellen $x_0, \cdots, x_n$ für die Auswertung und Interpolation effizient machbar sind

Algorithm 7 Polynommultiplikation

Require: P(x),Q(x) durch Koeffizienten $(a_0, \ldots, a_n), (b_0, \ldots, b_n)$
Ensure: P(x) · Q(x) durch Koeffizienten $(c_0, \ldots, c_{2n})$
werte P,Q (mit 0 aufgefüllt bis Dimension 2n+1) für $x_0, \ldots, x_{2n}$ aus $\rightarrow$ $(y_0, \ldots, y_{2n}), (z_0, \ldots, z_{2n})$
berechne $w_i = y_i \cdot z_1$
interpolieren $(w_0, \ldots, w_{2n}) \rightarrow (c_0, \ldots, c_{2n})$

Zeit: 2·Zeit für Auswertung (2n)+ Zeit für Interpolation(2n)+O(n)

4.2.4 Die k-ten Einheitswurzeln

ω heißt k-te Einheitswurzeln im Körper K gdw. $w^k = 1$. ω heißt primitive k-te EW gdw. $\omega^j \neq 1$ für $1 \leqslant j < k$

Beispiele

1. in $\mathbb{R}$: $1, -1$ sind 2-te EW und -1 ist prim. 2-te EW
2. Komplexe Zahlen: $K = \mathbb{C}$

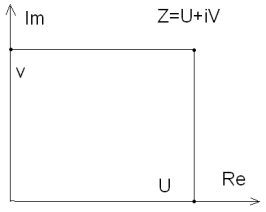


Abbildung 2: Nomale Darstellung

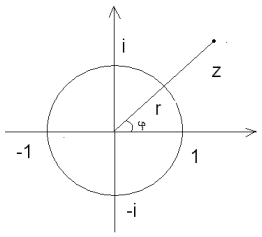


Abbildung 3: Darstellung in Polar-Koordinaten

Andere Darstellung (Polar-Koordination): $Z = r \cdot e^{i\varphi}$

Einheitswurzeln liegen auf Einheitskreis um 0, z.B.
k = 2 $\{1,-1\}$
k = 4 $\{1,-1,i,-i\}$ und i,-i sind Prim.
k = 3 $\{1, \frac{-1}{2} + i \cdot \frac{\sqrt{3}}{2}, \frac{-1}{2} + i \cdot \frac{-\sqrt{3}}{2}\}$

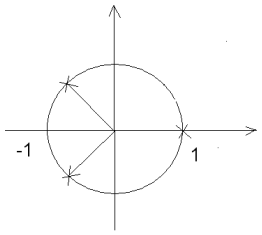


Abbildung 4: k=3

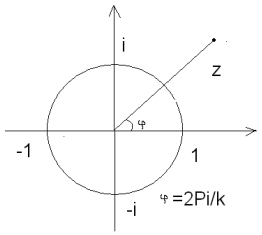


Abbildung 5: allgemeines k

allgemeines k: k-ten EW. sind $\{1, e^{i\cdot 2\pi/k}, e^{i\cdot 2\cdot 2\pi/k}, \cdots, e^{i\cdot(h-1)\cdot 2\pi/k}\}$
wobei $e^{i\pi} = -1 = \{1, \omega, \omega^2, \cdots, \omega^{k-1}\}$
wobei $\omega = e^{i\cdot 2\pi/k}$ primitive k-te EW

Lemma Ist $\omega \in k$ primitive k-te EW $\Rightarrow \{1, \omega, \omega^2, \cdots, \omega^{k-1}\}$ alle k-ten EW

3. $\mathbb{N}_p$ ist ein Körper, falls p eine Primzahl ist
   z.B. $\mathbb{N}_5 = \{0, 1, 2, 3, 4\}$,
   2 und 3 sind 4. primitive EW, 4 ist 2. EW

Allgemein gilt: Falls $M = C \cdot 2^r + 1$ eine Primzahl ist, so gibt es eine primitive $2^r - te$ EW in $\mathbb{N}_M$

So gibt es in $\mathbb{N}_{41}$ r=3 acht primitive EW:
z.B. $3^2 = 9, 9^2 = 81 = 40 = -1, -1^2 = 1$

Diskrete Fourier-Transformation (DFT)

Seien $P, Q \in K[x]$ Polynome vom Grad n

Falls K eine primitive n+1-ten EW ω besitzt, so sind n+1-te EW $\{1, \omega, \omega^2, \cdots, \omega^n\}$ alle verschieden.

Wähle diese als Stützstellen $x_0, \cdots, x_n \to$ Auswertung

$$\begin{pmatrix} y_0 \\ \vdots \\ y_n \end{pmatrix} = V \cdot \begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} \text{ mit } V = \begin{Bmatrix} 1 & 1 & \cdots & \cdots & 1 \\ 1 & \omega & \cdots & \cdots & \omega^n \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^n & \omega^{2n} & \cdots & \omega^{n^2} \end{Bmatrix}$$

Diese Abbildung $\begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} \mapsto \begin{pmatrix} y_0 \\ \vdots \\ y_n \end{pmatrix}$ heißt diskrete Fourier-Transformation(DFT).

Ein effizienter Algorithmus dafür ist die schnelle Fourier-Transformation (FFT, engl.: fast Fourier-transformation).

Auswertung: Annahme: $n + 1$ ist Zweierpotenz

$$\begin{aligned} P(x) &= a_0 + \cdots + a_n x^n \\ &= \underbrace{a_0 + a_2 x^2 + \cdots + a_{n-1} x^{n-1}}_{P_1(x^2)} + x \cdot (\underbrace{a_1 + a_3 x^3 + \cdots + a_n x^n}_{P_2(x^2)}) \\ &= P_1(x^2) + x \cdot P_2(x^2) \end{aligned}$$

Die Auswertung von P für $(n+1)$-EW: $(1, \omega, \cdots, \omega^n)$ ist also zurückführbar auf Auswertung von P_1 und P_2 an den Stellen $1, \omega^2, \omega^4, \cdots, \omega^{n-1}$. P_1 und P_2 haben Grad den $\frac{n-1}{2}$ und dies sind die $\frac{n-1}{2}$-ten EW, denn Potenz von ω^2, ω^2 ist primitive $\frac{n+1}{2}$-te EW.

Daraus ergibt sich ein rekursiver Algorithmus.

Verankerung Grad 1: Direkt

damit ist die Laufzeit:

$$\begin{aligned} T(1) &= C \\ T(m) &= 2T(\frac{m}{2}) + m + 1 \\ &\quad \text{wobei m=n+1 und m+1 bedeutet m Multplikation und 1 Addition} \\ &= 2T(\frac{m}{2}) + \mathcal{O}(n) \\ &= 4T(\frac{m}{4}) + 2 \cdot d \cdot \frac{m}{2} + dm \\ &= \mathcal{O}(m \log m) \end{aligned}$$

Interpolation $(n+1)$-ten EW $(1, \omega, \omega^2, \cdots, \omega^n)$
Interpolation

$$\begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} = V^{-1} \begin{pmatrix} y_0 \\ \vdots \\ y_n \end{pmatrix}$$

Behauptung: $V^{-1} = (\omega_{ij})_{0 \leqslant i,j \leqslant n}$ mit $\omega_{ij} = \frac{\omega^{-ij}}{n+1}$
Beweis:

$$\begin{aligned} \textstyle\sum_{k=0}^{n} v_{ik}\omega_{kj} &= \tfrac{1}{n+1} \textstyle\sum_{k=0}^{n} \omega^{k(i-j)} \\ &= \begin{cases} \frac{1}{n+1} \sum_0^n 1 = 1, & \text{für } i = j \\ 0, & \text{für } i \neq j \end{cases} \end{aligned}$$

Bemerkung: $i \neq j$: $\frac{w^{(i-j)(n+1)}-1}{w^{i-j}-1} = \sum_{k=0}^{n} \omega^{k(i-j)} = 0$, da $\omega^{n+1} = 0$

$$\begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} = \frac{1}{n+1} \underbrace{\begin{Bmatrix} 1 & 1 & \cdots & \cdots & 1 \\ 1 & \omega & \cdots & \cdots & \omega^n \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^n & \omega^{2n} & \cdots & \omega^{n^2} \end{Bmatrix}}_{\text{Wieder DFT}} \begin{pmatrix} y_0 \\ \vdots \\ y_n \end{pmatrix}$$

wobei $\omega' = \omega^{-1}$ ist auch eine primitive (n+1)-te EW. Also Interpolation auch mit FFT berechenbar (dann noch $\frac{1}{n+1}$ multiplizieren). Damit durch Algorithmus 1 mit FFT.

Satz

1. DFT und Inverse DFT für Dimension n ist in $\mathcal{O}(n \log n)$ Zeit berechenbar
2. Produkte von Polynomen von Graden in $\mathcal{O}(n \log n)$ Zeit berechenbar.

4.2.5 Schnelle Fourier-Transformation (FFT)

Der FFT Algorithmus Die Eingabe von FFT:

- Vektor $a = (a_0, \ldots, a_n)$ - die Koeffizienten eines Polynoms P von Grad n
- ω primitive (n+1)-te Einheitswurzel; o.B.d.A. (n+1) ist 2-er Potenz (man kann ja gegebenenfalls den Vektor mit 0-en Auffllen)

Bedeutung der Zeilen 7 und 8:
Im rekursiven Aufruf von FFT werden in Zeile 6 $P_1(1), P_1(\omega^2), P_1(\omega^4), \ldots$ bzw. in Zeile 7 $P_2(1), P_2(\omega^2), P_2(\omega^4), \ldots$ berechnet.
Erklärung der Zeilen 8 - 10:

Algorithm 8 Der Algorithmus für FFT in Pseudocode

```
1:  proc FFT(a, n, ω)
2:  if n ← 0 then
3:     return (a)
4:  else
5:     ag ← (a_0, a_2, a_4, ..., a_{n-1})
6:     au ← (a_1, a_3, a_5, ..., a_n)
7:     u ← FFT(ag, (n-1)/2, ω^2)
8:     v ← FFT(au, (n-1)/2, ω^2)
9:  end if
10: for i ← 0 to (n-1)/2 do
11:    y_i ← u_i + ω^i v_i
12:    y_{(n+1)/2+i} ← u_i - ω^i v_i
13: end for
14: return (y_0, ..., y_n)
```

P ist ein Polynom mit Grad n. Dann ist

$$\begin{aligned} P(x) &= a_0 + a_1 x + a_2 x^2 + \ldots + a_a x^n \\ &= a_0 + a_2 x^2 + a_4 x^4 + \ldots + a_{n-1} x^{n-1} + \\ &\quad x(a_1 + a_3 x^2 + a_5 x^4 + \ldots + a_n x^{n-1}) \\ &= P_1(x^2) + x P_2(x^2) \end{aligned}$$

Also gilt für $0 \leq i \leq \frac{n-1}{2}$:

$$P(\omega^i) = P_1(\omega^{2i}) + \omega^i P_2(\omega^{2i})$$

und für $\frac{n+1}{2} \leq j \leq n$, mit $i = j - \frac{n+1}{2} \rightarrow j = \frac{n+1}{2} + i$:

$$\begin{aligned} y_j &= P(\omega^j) \\ &= P_1(\omega^{2j}) + \omega^j P_2(\omega^{2j}) \\ &= P_1(\omega^{n+1+2i}) + \omega^{\frac{n+1}{2+i}} P_2(\omega^{n+1+2i}) \\ &= P_1(\omega^{2i}) - \omega^i P_2(\omega^{2i}) \end{aligned}$$

da $\omega^{n+1} = 1$ und $\omega^{\frac{n+1}{2}} = -1$.

Die Laufzeit (Anzahl der arithmetischen Operationen) $\mathcal{T}$:

$$\begin{aligned} \mathcal{T}(0) &= 0 \\ \mathcal{T}(n) &= 2\mathcal{T}\left(\frac{n-1}{2}\right) + \mathcal{O}(n) \\ \Rightarrow \mathcal{T}(n) &= \mathcal{O}(n \log n) \end{aligned}$$

Somit kann man die Polynom-Multiplikation mit Hilfe der DFT in $\mathcal{O}(n \log n)$ berechnen:

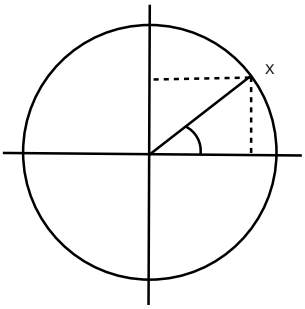
Seien $P = \sum_{i=0}^{n} a_i x^i$ und $Q = \sum_{i=0}^{n} b_i x^i$ Polynome. Dann ist:

$$DFT^{-1}(DFT(a) \cdot DFT(b)) = a \otimes b$$

4.2.6 Anwendungen der DFT

Die diskreten Fourier-Transformation ist nicht nur für die Polynom-Multiplikation von Bedeutung, sondern ist in der Signalverarbeitung eine wichtige Technik. Dazu noch einmal eine Wiederholung der Einheitswurzeln (in $\mathbb{C}$)

ω ist eine primitive k-te Einheitswurzel.



Dabei ist der Punkt x auf dem Einheitskreis $x = e^{i\varphi} = \cos\varphi + i\sin\varphi$

$$\begin{aligned} \omega &= e^{i\cdot\frac{2\pi}{k}} = \cos\left(\frac{2\pi}{k}\right) + i\sin\left(\frac{2\pi}{k}\right) \\ \omega^j &= \cos\left(\frac{2\pi\ j}{k}\right) + i\sin\left(\frac{2\pi\ j}{k}\right) \end{aligned}$$

Signale, die aus Frequenzüberlappungen bestehen, können dann mit Hilfe der Sinus- und Kosinusfunktionen beschrieben werden; sie sind ja Summen von hohen und niedrigen Frequenzen. In der Signalverarbeitung wird dann die diskrete Fourier-Transformation für die Bestimmung der einzelnen Frequenzen eingesetzt. Dabei sei noch angemerkt, dass die Fourier-Transformation im Allgemeinen über ein Integral angewandt wird. So hilft die DFT nun bei folgenden beispielhaften Anwendungen:

- Akustik: Zerlegen von Tönen in die einzelnen (Teil-)Frequenzen
- Bildverarbeitung: Spektren erstellen

Dazu benutzt man *Filterung*:

Bei der Filterung wird ein Signal „ Fourier“-transformiert und anschliessend bestimmte (Teil-)Frequenzen ausgeblendet. Dies beispielsweise indem man in einem bestimmten Intervall die Koeffizienten auf 0 setzt. Auf diese Weise erstellt man sogenannte Hoch- bzw. Tiefpassfilter.

- Hochpassfilter: niedrige Frequenzen werden ausgeblendet; hohe nicht
- Tiefpassfilter: niedrige Frequenzen werden nicht ausgeblendet; hohe dafür schon

Beispiel Bildbearbeitung bei Graustufenbildern:

Hohe Frequenzen bedeuten extreme hell-dunkel-Unterschiede und niedrige Frequenzen geringe hell-dunkel-Unterschiede zwischen benachbarten Bildpunkten. So werden durch das Anwenden eines Tiefpassfilters die hohen Frequenzen

herausgefiltert, die hell-dunkel-Unterschiede werden „weichgezeichnet“. Bei der Anwendung eines Hochpassfilters werden die niedrigen Frequenzen herausgefiltert und man erhält nur noch hohe Frequenzen, also die Umrisse der Objekte auf dem Bild.

4.2.7 Korrelation und Mustererkennung

Wir kennen die Konvolution:

$$c_k = \sum_i a_i b_{k-i}$$

Analog dazu ist die Korrelation:

$$c_k = \sum_i a_i b_{k+i}$$

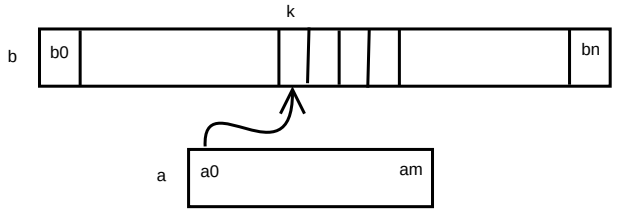
Die Korrelation kann auf die Konvolution zurückgeführt werden, indem man den Vektor $b = (b_0, \ldots, b_n)$ durch den Vektor $\hat{b} = (b_n, b_{n-1}, \ldots, b_0)$ als Eingabe für die Konvolution ersetzt.

$$\text{Korrelation von a, b} \leftrightarrow \text{Konvolution von a, } \hat{\text{b}}$$

Die Anwedung der Korrelation bei der Mustererkennung:
Gegeben sind:

- Ein Text $b = b_0, \ldots, b_n$
- Ein Muster $a = a_0, \ldots, a_m$
- meist gilt dabei $m \ll n$

Man nimmt an, dass das Alphabet {0,1} ist. Dann wandelt man alle 0-en in -1 in a und b um.



Schliesslich lässt man folgendes berechnen:

$$\sum_{i=0}^{m} a_i b_{k+i} \leftarrow k - \text{ter Koeffizient der Korrelation}$$

Dieser Koeffiezient ist groß, falls viele a_i mit b_{k+i} übereinstimmen und klein (negativ) wenn nicht. Somit stellt er ein Maß für die Güte der Übereinstimmung

von a an $b_k \dots b_{k+m}$ dar. Die Korrelation von a und b liefert diesen Wert für alle Stellen k.

Nimmt man den FFT-Algorithmus ist diese Berechnung in $\mathcal{O}(n \log n)$ Aufwand zu berechnen. Dabei füllt man vorher a mit 0-en auf, so dass $m = n$.

Im Vergleich dazu berechnet das bisherigen String-Matching nur „perfekte" Ergebnisse, ist also kein Gütemaß. Die Mustererkennung in Bildern ist dahingehend schwieriger, da auch Rotationen der Muster in dem Bild möglich sind.

4.3 Arithmetik

Man hat Zahlen in Binärdarstellung (oder einer beliebigen anderen Zahlendarstellung) und sucht nach Algorithmen für die arithmetischen Operationen $+, -, *, /, \sqrt{\ }, \dots$. Da effiziente Algorithmen erst durch die Zifferndarstellung von Zahlen möglich sind, wurden diese Operationen auch erst mit Einzug der arabischen Zahlen in Europa gesucht. Adam Ries (1492 - 1559) hat die arithmetischen Operationen auf die dezimalen Zahlen beschrieben.

Das Bitmodell Die Ein- und Ausgabe der Operationen sind binäre Darstellungen (ganzer) Zahlen. Als Kostenfunktion gelte die Anzahl der binären Operationen auf einzelne Bits; o.B.d.A seien dies $\wedge, \vee$ und $\neg$.

4.3.1 Addition von n-bit Zahlen

$$\begin{array}{r} 1101 \\ \underline{1001} \\ 10110 \end{array}$$

So wie man es kennt. XOR die einzelnen Ziffern von rechts nach links und dabei Überträge beachten. $\mathcal{O}(n)$ Kosten.

4.3.2 Subtraktion

ebenfalls in $\mathcal{O}(n)$, da auf Addition reduzierbar.

4.3.3 Multiplikation von n-bit Zahlen

- Nach der Schulmethode:

$$\begin{array}{r} \underline{1011 \cdot 101} \\ 1011 \\ \underline{1011\ \ } \\ 110111 \end{array}$$

  Die Schulmethode multipliziert zwei binäre Zahlen in dem die eine Zahl Stelle für Stelle mit der anderen Zahl multipliziert (AND) wird. Anschliessend werden alle berechneten Zwischenergebnisse stellenabhängig zusammen addiert.

Die Schulmethode ist zurückzuführen auf n Additionen von $\mathcal{O}(n)$-Bit-Zahlen $\rightarrow \mathcal{O}(n^2)$ binäre Operationen bei zwei n-Bit-Zahlen. Geht es besser?

- Multiplikation nach Karatsuba und Offman

  Gegeben sind zwei n-Bit-Zahlen $a = a_1 2^{\frac{n}{2}} + a_2$ und $b = b_1 2^{(\frac{n}{2})} + b_2$ (ohne wesentliche Einschränkungen (owE): n Zweierpotenz).

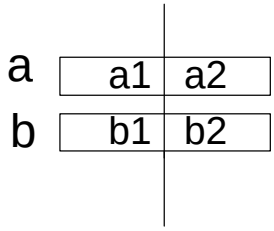


Abbildung 6:

Dann kann die Multiplikation geschrieben werden als:

$$a \cdot b = \underbrace{a_1 b_1}_{m_2} 2^n + (\underbrace{a_1 b_2 + a_2 b_1}_{m_1 - m_2 - m_3}) 2^{\frac{n}{2}} + \underbrace{a_2 b_2}_{m_3}$$

Die hier gekennzeichneten Terme m_1, m_2 und m_3 werden wie folgt berechnet:

$$\begin{aligned} m_1 &= (a_1 + a_2) \cdot (b_1 + b_2) \\ m_2 &= a_1 b_1 \\ m_3 &= a_2 b_2 \end{aligned}$$

Die Koeffizienten können jetzt rekursiv berechnet werden. Es ergibt sich eine Reduzierung der benötigten Operationen. Also wendet man diese Idee bzw. diesen Algorithmus rekursiv an, um die Koeffizienten a und b zu berechnen.

$$\begin{aligned} \mathcal{T}(n) &= 3\mathcal{T}(\frac{n}{2}) + \underbrace{\mathcal{O}(n)}_{Add/Sub} \\ &= \mathcal{O}(3^{\log_2 n}) = \mathcal{O}(n^{\log_2 3}) = \mathcal{O}(n^{1.585\ldots}) \end{aligned}$$

- Geht es noch schneller?

  Eine schnelle Methode der Multiplikation von Polynomen liefert die „Fast Fourier-Transformation“ (FFT), die nur $\mathcal{O}(n \log n)$ Operationen benötigt.

  Für die Multiplikation der Polynome a und b gilt:

$$a = \sum_{i=0}^{n} \alpha_i \cdot 2^i, \qquad b = \sum_{i=0}^{n} \beta_i \cdot 2^i \qquad , \alpha_i, \beta_i \in \{0, 1\},$$

$$\text{dann} \qquad a \cdot b = (P \cdot Q)(2)$$

$$\text{wobei} \qquad \begin{cases} P(x) = \sum_{i=0}^{n} \alpha_i \cdot x^i \Longrightarrow a = P(2) \\ Q(x) = \sum_{i=0}^{n} \beta_i \cdot x^i \Longrightarrow b = Q(2) \end{cases}$$

Für diesen Fall gibt es einen effizienten Multiplikationsalgorithmus auf der Basis der FFT, welche auf der diskreten Fourier Transformation aufbaut.

Zuerst wird in eine Koeffizientenmatrix-Darstellung transformiert, die Berechnungen ausgeführt, und schliesslich mittels der inversen Fourier Transformation zurück-transformiert.

(Nachzulesen bei: Schönhage / Strassen, 1971)

Anzahl der Bitoperationen $\mathcal{O}(n \log n \cdot \log\log n)$

$$\begin{aligned} F_n &= 2^{2^n} + 1 \\ F_{16} &= 2^{2^{16}} + 1 = 2^{65536} + 1 = \ldots \end{aligned}$$

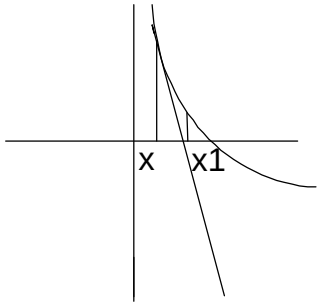
4.3.4 Division von n-bit-Zahlen

gegeben: $a, b \in \mathbb{N}$, n-Bit-Zahlen
gesucht: $\lfloor a/b \rfloor \longleftarrow$ ganzzahlige abgerundete Division von zwei Zahlen a und b

Die Verallgemeinerung auf Festkommazahlen a, b und Festkommaergebnis mit festgesetzter Genauigkeit verbleibt als Übungsausgabe.

Schulmethode braucht $\Theta(n^2)$ Binäroperationen. Geht es schneller?
dazu: zunächst Inversion $\frac{1}{b}$ mit Fehler $\leq 2^{-2n}$ für gegebenes $b \in \mathbb{N}$, n-stellig
danach Ergebnis mit a multiplizieren.

Newton-Iterationsverfahren $\frac{1}{b}$ ist Nullstelle der Funktion $f(x) = \frac{1}{x} - b$



$$\begin{aligned} x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} \qquad \text{konvergiert gegen Nullstelle bei guten Startwert } x_0 \\ &= x_i - \frac{\frac{1}{x_i} - b}{\frac{-1}{x_i 2}} \\ &= x_i + x_i - b x_i 2 \\ &= x_i(2 - b x_i) \qquad \text{konvergiert gegen} \frac{1}{b}, \text{falls Startwert } x_0 \in (0, \frac{1}{b}] \end{aligned}$$

Laufzeit

- pro Iterationsschritt 2 Multiplikationen, 2 Subtraktionen
  d.h. $\mathcal{O}(\mathcal{M}(n))$ Operationen, wobei $\mathcal{M}(n)$ Komplexität des verwendeten Multiplikations-Algoritmus

- Wie viele Iterations-Schritte?
  Newton-Iterationsverfahren konvergiert quadratisch, d.h. Fehler quadriert sich in jedem Schritt.

  $$\epsilon_{i+1} = \mathcal{O}(\epsilon_i^2)$$

  um Fehler $\leq 2^{-2n}$ zu erreichen braucht man $\mathcal{O}(\log n)$ Schritte, also $\mathcal{O}(\mathcal{M}(n) \cdot \log n)$ Schritte insgesamt.

  Der $\log n$-Faktor kann eliminiert werden in dem man nur mit den Bits rechnet, die ohnehin richtig sind. Das sind im i-ten Schritt: $\mathcal{O}(2^i)$ Bits.

  Also sind die Gesamtkosten der Division:

$$\begin{aligned}
\mathcal{D}(n) &= \mathcal{M}(c) + \mathcal{M}(2c) + \cdots + \mathcal{M}(2n) \\
&= \sum_{i=0}^{\mathcal{O}(\log n)} \mathcal{M}(c2^i) = \sum_{i=0}^{\mathcal{O}(\log n)} \mathcal{M}(c\frac{2n}{2^i}) \\
&= d \cdot \mathcal{M}(n) \sum_{i=0}^{\mathcal{O}(\log n)} \alpha^i \\
&= \mathcal{O}(\mathcal{M}(n))
\end{aligned}$$

  Annahme:

  M ist gutartig, d.h. $\exists$ Konstante d mit $M(c \cdot \frac{n}{2}) \leq d\alpha M(n)$, konst $\alpha < 1$

  Satz: Die ganzzahlige Division von n-Bit-Zahlen ist möglich mit $\mathcal{D}(n) = \mathcal{O}(\mathcal{M}(n))$ binären Operationen.

  Wobei $\mathcal{M}(n)$ Kosten der Multiplikation, gutartige Fkt.
  also $\mathcal{D}(n) = \mathcal{O}(n \log n \log\log n)$ ist möglich.

4.3.5 Quadratwurzel

gegeben: n-Bit-Zahl $a \in N$
gesucht: $\lfloor \sqrt{a} \rfloor$

Die Verallgemeinerung auf Festkommazahlen verbleibt als Übung.

Schulmethode: in $\Theta(n^2)$

Wobei das Zeichen * in der Zeile von $r := r*$ für die Konkatenation von Ziffernfolgen steht.

Algorithm 9 Schulmethode zur Berechnung der Quadratwurzel $w = \sqrt{a}$

1: Teile a, von rechts beginnend in Blöcke der Grösse 2
2: $w \leftarrow 0$, $r \leftarrow 0$,
3: **repeat**
4: $r \leftarrow r*$ (nächste Block von a von links)
5: $z \leftarrow$ die maximale Ziffer mit $[(2w) * z] \cdot z \leq r$
6: $r \leftarrow r - [(2w) * z] \cdot z$
7: $w \leftarrow w * z$
8: **until** gewünschten Genauigkeit erreicht

Beispiele

- $\sqrt{a} = \sqrt{34'23}$
  der erste Block von links ist damit: $r = 34$
  $5 * 5 = 25 \leq [34] < 36 = 6 * 6$
  daraus folgt: $\sqrt{a} = \sqrt{(34)'23} = 5, ...$
  nun der nächste Schritt: $0\underline{5} \cdot \underline{5}$

  $\sqrt{(34)'23} = 58,5...$
  $\underline{25}$
  923
  $\underline{864}$
  5900
  $\underline{5825}$

  $0\underline{5} \cdot \underline{5}$
  $10\underline{8} \cdot \underline{8}$
  $116\underline{5} \cdot \underline{5}$

- mit binären Zahlen:
  $\sqrt{(10)} = 1,01$
  $\underline{1}$
  $1'00'00$
  $\underline{1001}$
  0011100
  $\underline{10101}$

mit Newton-Iteration: $\sqrt{a}$ ist Nullstelle der Funktion $f(x) = x^2 - a$
Newton-Iteration:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^2 - a}{2x_i} = \frac{1}{2}(x_i + \frac{a}{x_i})$$

Dieser Sachverhalt war schon Heron von Alexandrien um 150 v. Chr. bekannt.

Idee: für jedes x liegt $\sqrt{a}$ zwischen x und $\frac{a}{x}$
$x < \sqrt{a} \Leftrightarrow \frac{a}{x} > \sqrt{x}$ und $x > \sqrt{a} \Leftrightarrow \frac{a}{x} < \sqrt{x}$

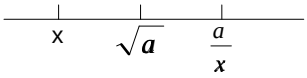


Abbildung 7:

Iterationsvorschrift reduziert $\sqrt{\ }$ auf Division und Addition. analog wie bei Reduktion von Division auf Multiplikation:

$$\underbrace{\mathcal{R}(n)}_{\sqrt{\ }-Kosten} = \underbrace{\mathcal{O}(\mathcal{D}(n))}_{Divisionskosten} = \underbrace{\mathcal{O}(\mathcal{M}(n))}_{Multiplikationskosten}$$

In der Praxis ist folgendes Verfahren besser:
$1/\sqrt{a}$ ist Nullstelle von $f(x) = a - \frac{1}{x2}$
Newton-Iteration (divisionsfreie Iteration):

$$\begin{aligned} x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{(a - \frac{1}{x_i^2})x_i^3}{\frac{2}{x_i^3}} \\ &= x_i - \frac{1}{2}ax_i^3 + \frac{1}{2}x_i = \frac{x_i}{2}(3 - ax_i^2) \end{aligned}$$

am Schluss noch mit a multiplizieren $\longrightarrow \sqrt{a}$
umgekehrt:
Multiplikation ist reduzierbar auf Quadrieren:

$$a \cdot b = \frac{1}{2}[(a+b)^2 - a^2 - b^2]$$

Quadrieren reduzierbar auf Inversion:

$$x^2 = \frac{1}{\frac{1}{x} - \frac{1}{x-1}} - x$$

Somit ist die Multiplikation reduzierbar auf die Inversion.

4.4 Zahlentheoretische Probleme

Zahlen sind binär dargestellt(oder bzgl. einer anderen Basis), sei n die Anzahl der Bits

4.4.1 Größter gemeinsamer Teiler

gegeben: zwei n-Bit Zahlen a, b mit $a \geq b$

Laufzeit: $\mathcal{O}(n * \mathcal{M}(n))$ wobei $\mathcal{M}(n)$ die Bit-Kosten der Multiplikation sind

4.4.2 Faktorisierung:

gegeben Zahl $a \in \mathbb{N}$,finde Zerlegung von a in Primfaktoren.

Algorithm 10 Der Euklidische Algorithmus (300 v. Chr.)

```
1: proc Euklid(a, b)
2: if b == 0 then
3:    return a
4: else
5:    return Euklid(b, a mod b)
6: end if
```

Beispiel: $996 = 2 * 2 * 3 * 83$

Für die Faktorisierung ist kein Polynomialzeitalgorithmus bekannt! Für die Faktorisierung von Polynomen ist jedoch ein effizienter Algorithmus bekannt. $P(x) = P_1(x) * ... * P_k(x)$ wenn alle $P_i, 1 \leq i \leq k$ irreduzible Faktoren sind. (Lenstra,Lenstra, Lovasz 1982)

4.4.3 Einfacher Primzahltest:

gegeben $a \in \mathbb{N}$, ist a eine Primzahl?

einfacher Primzahltest hat eine Laufzeit von $\Omega(\sqrt{a}) = \Omega(2^{\frac{1}{2}n}) \approx \Omega(1,41^n)$

Sieb des Eratosthemes(200 v.Chr.) bestimmt alle Primzahlen $\leq a$.

gegeben Liste l = [2..a],nimm das erste Element e von l, gib es aus, und entferne alle Vielfachen von e aus der Liste.

Wiederhole bis l leer ist.

etwas Zahlentheorie ... $\mathbb{Z}_n, +, *$ RestklassenRing, $\mathbb{Z}'_n, *$ multiplikative Gruppe, diese enthält alle zu n teilerfremden Elemente aus Z_n, alle diese haben ein multiplikatives Inverses.

Anzahl: $|\mathbb{Z}'_n| = \varphi(n)$ Eulersche φ-Funktion. Diese zählt alle Zahlen $< n$ auf, die teilerfremd zu n sind.

Beispiel: $\mathbb{Z}_6 = \{0, .., 5\}$ $\mathbb{Z}'_6 = \{1, 5\}$
$\mathbb{Z}_9 = \{0, .., 8\}$ $\mathbb{Z}'_9 = \{1, 2, 4, 5, 7, 8\}$

Falls n eine Primzahl ist, gilt:

- $\mathbb{Z}_n$ ist ein Körper
- $\mathbb{Z}'_n = Z_n \setminus \{0\}$ also $\varphi(n) = n - 1$
- $a^{n-1} \equiv 1 \bmod n \forall a \in \mathbb{Z}'_n$ (Satz von Fermat)
- $a^{\frac{n-1}{2}} \equiv \pm 1 \bmod n \forall a \in \mathbb{Z}'_n$

Anmerkungen: Satz von Fermat ist nicht äquivalent zu Primalität, es gibt zusammengesetzte Zahlen mit dieser Eigenschaft, die *Carmichael-Zahlen*. Diese sind allerdings sehr selten, und es ist erst seit kurzem bekannt, dass unendlich viele existieren.

Die kleinste Carmichael-Zahl ist 561, die zweitkleinste ist 1729.
daraus ergibt sich ein Pseudo-Primzahltest (siehe Listing)

Algorithm 11 Pseudo-Primzahltest

```
1: gegeben n ∈ ℕ
2: for x = 0; x < t; x = x + 1 do
3:    wähle zufällig (gleichverteilt) a ∈ ℤ_n \ {0}
4:    if ggT(a, n) ≠ 1 then
5:       sage 'nein'
6:    else
7:       falls a^{n-1} mod n ≠ 1 sage 'nein'
8:    end if
9: end for
10: falls alle Tests bestanden gib 'ja' aus.
```

Die Antwort *nein* ist immer richtig bei einer zusammengesetzten Zahl, die keine Carmichael-Zahl ist, ist die Wahrscheinlichkeit einer falschen Antwort $\leq 2^{-t}$.

Es existiert kein anerkannter probabilistischer Algorithmus.

4.4.4 Probabilistische Algorithmen

verwenden Zufallsgenerator und treffen darauf basierend zufällige Entscheidungen.

Zwei Typen:

- Monte-Carlo-Algorithmen: effiziente Laufzeit, aber Antwort kann (mit geringer Wahrscheinlichkeit für jede einzelne Eingabe) falsch sein
- Las-Vegas-Algorithmen: Antwort ist sicher richtig; mit hoher Wahrscheinlichkeit effiziente Laufzeit.

Der Monte-Carlo-Algorithmus für Primzahltest von Miller-Rabin

Korrektheit: Beobachtung: Ist n eine Primzahl, so gilt für jedes $b \in \mathbb{Z}_n \setminus \{1, -1\}$ $b^2 \not\equiv 1 \bmod n$

Beweis: in $\mathbb{Z}_n$

falls $b^2 = 1 \Leftrightarrow b^2 - 1 = (b-1) * (b+1) = 0$

Weil $\mathbb{Z}_n$ ein Körper ist, muß einer dieser Faktoren null sein.

Damit gilt **Lemma 1:**

Falls n Primzahl ist, so akzeptiert der Miller-Rabin-Algorithmus.

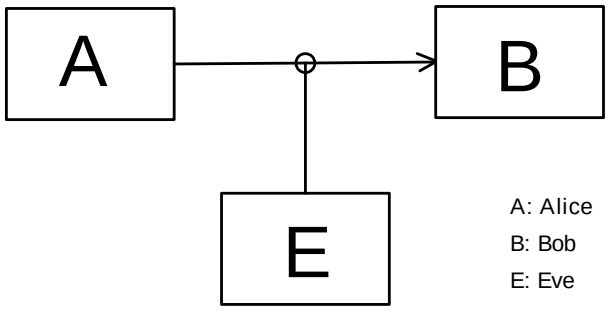
Die Tests in Zeile 16 werden alle bestanden, da das $a \neq \pm 1$ und s ungerade. Alle anderen Elemente der Folge sind Quadratzahlen, alle $\neq 1$, das letzte $a^{s2^h} = a^{n-1}$.

Algorithm 12 Miller-Rabin

gegeben $n \in \mathbb{N}$
if n gerade **then**
  if $n = 2$ **then**
    gib ja aus
  else
    gib $nein$ aus.
  end if
else
  führe k-mal aus
  for $x = 0; x < k; x = x + 1$ **do**
    wähle zufällig gleichverteilt ein $a \in Z_n \ \{1, -1\}$
    if $a^{n-1} \not\equiv 1 \bmod n$ **then**
      gib $nein$ aus
    else
      bestimme $s * t$ mit $n - 1 = s * t$, s ungerade $t = 2^h$ Zweiterpotenz
      berechne $a^{s2^0}, a^{s2^1}, \ldots a^{s2^h}$ letzte, das ungleich eins ist, und sage $nein$ falls dieses $\neq -1$ ist.
    end if
  end for
end if
falls alle Tests bestanden gib 'ja' aus.

22. Vorlesung fehlt. (nur pdf)

4.5 Public-Key-Kryptosysteme:



- A schickt eine verschlüsselte Nachricht P_B(M) an B. Wobei M die eigentliche Nachricht ist. Die Schlüsselfunktion P_B ist *öffentlich* bekannt, d.h. jeder kann Nachrichten für B verschlüsseln.

- aber: Die Umkehrfunktion S_B ist nur B selbst bekannt und für alle anderen geheim. Umgekehrt hat B eine Nachricht M und verschlüsselt diese mittels S_B:S_B(M) , so kann sie *jeder* entschlüsseln mit:
  P_B:$P_B(S_B(\mathrm{m}))$= M

- Kommt eine sinnvolle Nachricht M heraus, so kann *nur* Bob sie geschickt

haben, denn nur er kennt den Schlüssel S_B.
Dies ist eine Art : „digitale Unterschrift, Authentisierung“

Wie findet man Funktionen P_B, S_B?
Theorie: Einwegfunktionen (engl. one-way-Functions)

Definition: Sei Σ ein Alphabet. f: $\Sigma^* \to \Sigma^*$ heisst *Einwegfunktion* (EWF) gdw.

1. f ist in polynomieller Zeit berechenbar
2. f ist injektiv
3. Die Umkehrfunktion f^{-1} ist nicht in polynomieller Zeit berechenbar.

(es gibt schärfere Def. : probabilistische Alg.)

Anmerkungen:

- falls f eine Einwegfunktion ist, so ist f^{-1} nichtdeterministisch in polynomieller Zeit berechenbar.
  (Klasse FNP : $g \in$FNP gdw. g(y)=x? für geg. x,y in polynomieller Zeit verifizierbar)
- Es ist von keiner Funktion bekannt, dass sie eine Einwegfunktion ist.

In der Praxis, werden *Kandidaten* benutzt, von denen man glaubt, dass sie Einwegfunktionen sind. z.B.: f(u,v) $\to$ u·v , mit Primzahlen u,v

Algorithm 13 RSA-Kryptosystem (Rivest, Shamir, Adleman)

1: Wähle zwei beliebige, "grosse" Primzahlen p,q
2: Berechne n=p·q
3: Wähle "kleine" ungerade Zahl e, die teilerfremd zu φ(n) ist.
4: Berechne d, das multiplikative Inverse zu e im Ring $\mathbb{Z}_{\varphi(n)}$
5: Veröffentliche: P=(e,n) als öffentlichen Schlüssel
6: Behalte: S=(d,n) als geheimen Schlüssel

Jede zu verschlüsselnde Nachricht wird als Element von $\mathbb{Z}_n$ betrachtet. (also haben Nachrichten maximal log(n) Bits)
M$\in\mathbb{Z}_n$ wird verschlüsselt zu: P(M) = M^e mod n, Entschlüsselung: S(C)=C^d mod n

Zu 1: Was ist eine "grosse" Primzahl?
In der Praxis sollten p,q mehrere 1000 Dezimalstellen haben. Man fängt mit irgendeiner Zahl a dieser Grösse an und testet ob a eine Primzahl ist. (z.B. mit Miller/Rabin) - Falls nein, probiere a+2,a+4,... bis eine Primzahl gefunden wird.

Wie lange muss man suchen? $\Rightarrow$ Primzahldichte π(n) = Anzahl der Primzahlen $\leq$n
Dann gilt: $\pi(n) = \frac{n}{ln(n)} + O(1)$ (Gauß)
Zum Beispiel: n=10000 $\sim e^8$ - jede 8. Zahl ist Primzahl, aber selbst bei 10^{3000} etwa jede 5000. Man muss also etwa log(n) Schritte suchen.

Zu 3.:

$$\begin{aligned}
\varphi(n) &= \text{Anzahl der teilerfremden Zahlen } < n \\
&= \underbrace{(n-1)}_{\text{alle Zahlen} < n} - (\overbrace{\frac{n}{p}-1}^{\text{Vielfache von } p}) - \underbrace{(\frac{n}{p}-1)}_{\text{Vielfache von } q} \\
&= (p-1)(q-1) \qquad (\text{da } n = p \cdot q)
\end{aligned}$$

Um e zu finden muß man eine erschöpfende Suche durchführen, also 3,5, ... durchprobieren. Jedes Mal berechnet man den ggT mit φ(n). Wenn dieser = 1 ist, hat man ein e gefunden.
Anzahl der Tests $\mathcal{O}(\log \varphi(n))$

Zu 4.: Allgemein gilt: $\forall$r,s$\in \mathbb{Z}$, $\exists$a,b$\in \mathbb{Z}$ mit a$\cdot$r + b$\cdot$s = ggT(r,s)
Und a,b lassen sich mit dem Euklidische Algorithmus berechnen.

Beispiel $r = 51, s = 18$

$$\begin{aligned}
51 &= 2 \cdot 18 + 15 \\
18 &= 1 \cdot 15 + 3 \quad \Rightarrow 3 = 18 - 15; 15 = 51 - 2 \cdot 18 \\
15 &= 5 \cdot 3 + 0 \quad \Rightarrow 3 = -51 + 3 \cdot 18
\end{aligned}$$

Berechne a, b mit $a \cdot \varphi(n) + b \cdot e = 1$ (mod $\varphi(n)$ nehmen)
$\Leftrightarrow [b \bmod \varphi(n)] \cdot e \bmod \varphi(n) \equiv 1$ (mod $\varphi(n)$)
$\Rightarrow d = b \bmod \varphi(n)$

Korrektheit von Ver- und Entschlüsseln: zu Zeigen: S(P(M)) = M bzw. P(S(C))=C

$$\begin{aligned}
\text{in } \mathbb{Z}_n : S(P(M)) &= (M^e)^d = M^{ed} \\
\text{in } \mathbb{Z} : M^{ed} &= M^{1+k\cdot\varphi(n)} \text{für ein } k \in \mathbb{Z} \\
&= M^{1+k\cdot(p-1)\cdot(q-1)}
\end{aligned}$$

Falls M $\neq 0$ mod p, gilt:

$$\begin{aligned}
M^{ed} &= M(M^{p-1})^{k\cdot(q-1)} \equiv M \bmod p \text{ denn} \\
M^{p-1} &\equiv 1 \bmod p \text{ (nach dem kleinen Satz von Fermat)}
\end{aligned}$$

Analog dazu:

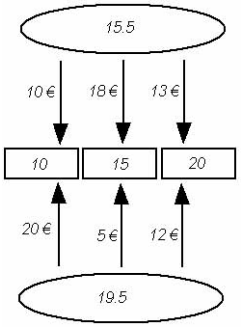
$$\begin{aligned}
M^{ed} &\equiv M \bmod q \\
\Rightarrow M^{ed} &\equiv M \bmod (p \cdot q) \\
&\text{(Chinesischer Restsatz: } a \bmod p, a \bmod q \Rightarrow a \bmod (p \cdot q))
\end{aligned}$$

RSA verlässt sich darauf, dass schwer zu berechnen ist:

- p,q aus n (n öffentlich)
- M aus r=M^e in $\mathbb{Z}_n$ „M=$\sqrt[e]{r}$“ (e-te Wurzel ziehen ist in $\mathbb{R}$ leicht aber in $\mathbb{Z}_n$ schwer)

5 Lineare Programmierung

Beispiel 1: Wir betrachten zwei Milchsammelstellen (Elipsen) und drei Weiterverarbeitungsstellen (Rechtecke). An den Kanten stehen die Transportkosten und in den Elipsen bzw. Rechtecken die Kapazitäten. Gesucht sind die minimalen Kosten für die Weiterverabeitung, bei grösstmöglicher Weiterverarbeitung.



minimiere $13x_1 + 18x_2 + 10x_3 + 12x_4 + 5x_5 + 20x_6$
Nebenbedingungen:

$$\begin{aligned} x_1 + x_2 + x_3 &= 15.5 \\ x_4 + x_5 + x_6 &= 19.5 \\ x_1 + x_4 &\leq 20 \\ x_2 + x_5 &\leq 15 \\ x_3 + x_6 &\leq 10 \\ x_1 + \ldots + x_6 &\geq 0 \end{aligned}$$

(Berechnung in Mapple mit minimize + Term + Nebenbedingungen)

allgemein: lineares Programm (im Sinne von Beschreibung):
Variablen $x_1, \ldots, x_n$
Ziel: maximiere oder minimiere die Funktion f mit
$f(x_1, \ldots, x_n) = c_1x_1 + \ldots + c_nx_n$, $c_i \in \mathbb{R}$
unter den Nebendbedingungen

$$\begin{aligned} a_{11}x_1 + \ldots + a_{1n}x_n &\leq b_1 \\ a_{k1}x_1 + \ldots + a_{kn}x_n &\leq b_k \end{aligned}$$

Ersetze $l(x) = b$ durch $l(x) \leqq b$ und $l(x) \geqq b$. Außerdem ersetze $l(x) \geq b$ durch $-l(x) \leq -b$ Eine Maximierung von f, ist gleichbedeutend mit der Minimierung von $-f$.

Vektorielle Schreibweise:

$$\vec{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \vec{c} = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} A = \begin{pmatrix} A_{11} & \dots & A_{1n} \\ \vdots & & \vdots \\ A_{n1} & \dots & A_{nn} \end{pmatrix} \vec{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

Minimiere Zielfunktion $f(x) = C^T * \vec{x}$ unter den Nebenbedingungen $A * \vec{x} \leq \vec{b}$

Beispiel 2: Wie Beispiel 1, außerdem kann die Milch nur in Kontainern von einem Kubikmeter Fassungsvermögen transportiert werden. In Beispiel 1 war ein Vektor $v \in \mathbb{R}^n$ gesucht, für Beispiel 2 wird nun ein Vektor $v \in \mathbb{Z}^n$ gesucht. Dies nennt sich ganzzahlige lineare Programmierung. Außerdem gibt es noch die 0-1 lineare Programmierung mit $\vec{x} \in \{0, 1\}$ und $|x| = n$.

Satz: 0-1 lineare Programmierung ist NP-schwer.

Beweis: Betrachte Optimierungsproblem MCLIQUE, welches die größte Clique in einem Graphen $G = (V, E)$ findet. Das Entscheidungsproblem CLIQUE ist NP-schwer, und somit auch das zugehörige Optimierungsproblem MCLIQUE. Reduktion von MCLIQUE auf $\{0, 1\}$-LP. Betrachte die Variablen $x_1, \dots, x_n$, $n = |V|$. $x_i = 1$ bedeutet: Knoten i in CLIQUE enthalten, $x_i = 0$ bedeutet Knoten i nicht in CLIQUE enthalten.

Zielfunktion: $x_1 + \dots + x_n$

Nebenbedingungen: $x_i + x_j \leq 1 \forall (i, j) \notin E$

Dies stellt sicher, das nur Knotenpaare in die Clique aufgenommen werden, zwischen denen auch eine Kante in G existiert. $\Rightarrow \mathcal{O}(n^2)$ für Nebenbedingungen.

Somit ist der optimale Wert der Zielfunktion gleich der Größe der maximalen CLIQUE in G und somit ist gezeigt das $\{0,1\}$-LP NP-schwer ist.

Korollar: Ganzzahlige LP ist NP-schwer, denn ein $\{0, 1\}$-lineares Programm kann auf ganzzahliges LP zurückgeführt werden. Dies erfolgt durch hinzufügen der Ungleichungen $x_i \leq 0$ sowie $x_i \geq 0 \forall i \in \{0..n\}$

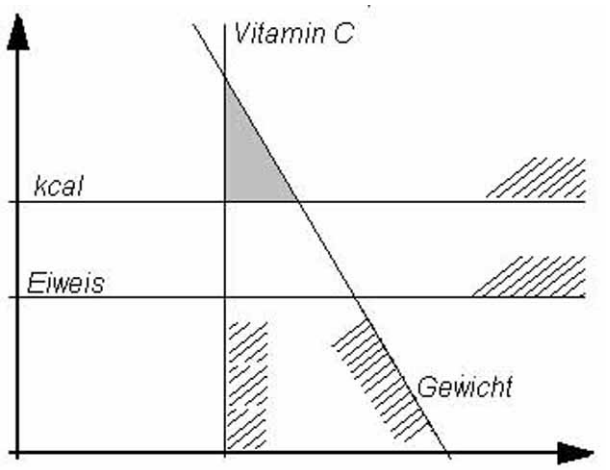
Beispiel 3: Ein Wildweststädtchen hat genau einen Laden. Dieser bietet 2 Nahrungsmittel an. Es gibt dort Bohnen mit Speck als Konserve zum Preis von 20cent/kg und frische Tomaten zum Preis von 10cent/kg. Ein Cowboy benötigt pro Tag 400kcal, 120g Eiweis und 80mg Vitamin C.

	kcal	Eiweiß	Vitamin C
Bohnen	2750	95	3
Tomaten	175	5	245

Es gibt eine Auflage, das pro Cowboy und Tag maximal 2,5kg eingekauft werden dürfen. Gesucht ist nach der billigsten Nahrungskombination, welche alle Bedingungen erfüllt.

Zielfunktion: x kg Tomaten und y kg Bohnen mit Speck
minimiere: 10x+20y mit den Nebenbedingungen:

$$\begin{aligned} 175x + 2750 &\geq 4000 \\ 5x + 95y &\geq 120 \\ 245x + 3y &\geq 80 \\ x + y &\leq 2,5 \\ x &\geq 0 \\ y &\geq 0 \end{aligned}$$

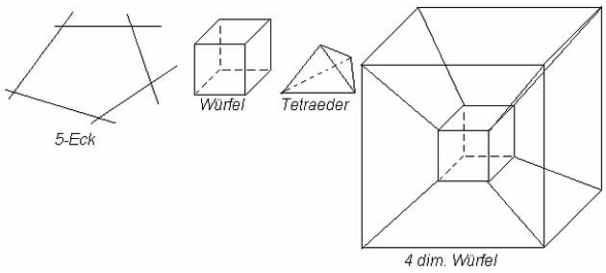


Es ergibt sich obiger Graph. Die Nebenbedingungen sind als Geraden eingezeichnet. Jede Gerade teilt die Ebene in zwei Halbebenen, in einer der beiden kann die Lösung liegen (schraffiert). Somit kann die Lösung nur im dem grauen Bereich liegen, da dieser die Schnittmenge aller schraffierten Halbebenen ist.

allgemein: eine Menge der Form

$$\{\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n | c_1 x_1 + \ldots + c_n x_n \leq b\}$$

heißt Halbraum. Ein Schnitt endlich vieler Halbräume heißt im allgemeinen konvexes Polyeder.



Der Schnitt kann auch leer sein bzw. niedrigere Dimensionen haben, kann aber auch unbeschränkt sein. Wenn der Schnitt beschränkt ist, heißt es konvexes Polytop. Ein Punkt v eines Polyeders P heißt Extrempunkt (Ecke), genau dann wenn er nicht als strikte Konvexkombination zweier Punkte $x, y \in P$ ist.

$$\nexists \lambda v = \lambda \cdot x + (1 - \lambda) \cdot y, 0 < \lambda < 1$$

also so, dass v auf der Strecke $\overline{xy}$ liegt, aber $v \neq x$ und $v \neq y$ ist, dies ist nicht möglich wenn v Ecke ist. Die Anzahl der Ecken steigt exponentiell mit den Nebenbedingungen.

25. Vorlesung fehlt. (nur pdf)

5.1 allgemeine Form eines linearen Programms

Standardform:

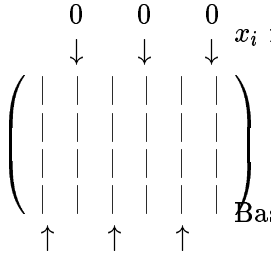
min. $c^T \cdot x$
Nb. $A \cdot x = b$
$x \geq 0$

wobei: $x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$,

A $m \times n$ Matrix, $m \geq n$, Zeilenvektoren sind linear unabhängig $\Rightarrow$ m linear unabhängige Vektoren

$b = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$

Basis: m linear unabhängige Vektoren



x_i für $i \notin J$ auf 0 setzen ($n - m$ Stück)

Die restlichen $x_i, i \in J$ sind dann eindeutig bestimmt.

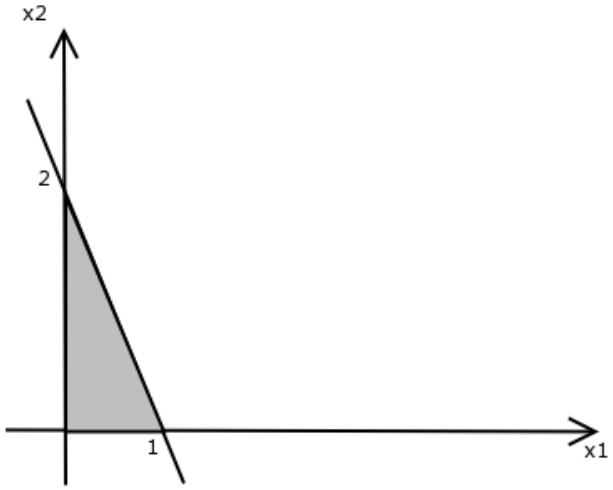
$$B \cdot \hat{x} = b$$

Eine Lösung heißt zulässige Basislösung, wenn sie die Vorzeichen-Bedingung ($x \geq 0$)erfüllt (bfs - basic feasible solution).

Beispiel: Lineare Programmierung:

min. $x_1 - x_2$, also $c = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$ Nb.:

$2x_1 + x_2 \leq 2$
$x_1 \geq 0$
$x_2 \geq 0$



Standardform:

$$2x_1 + x_2 + \underbrace{x_3}_{Schlupfvariable} = 2 \qquad A = \begin{pmatrix} 2 & 1 & 1 \end{pmatrix}$$

$x_1 \geq 0$

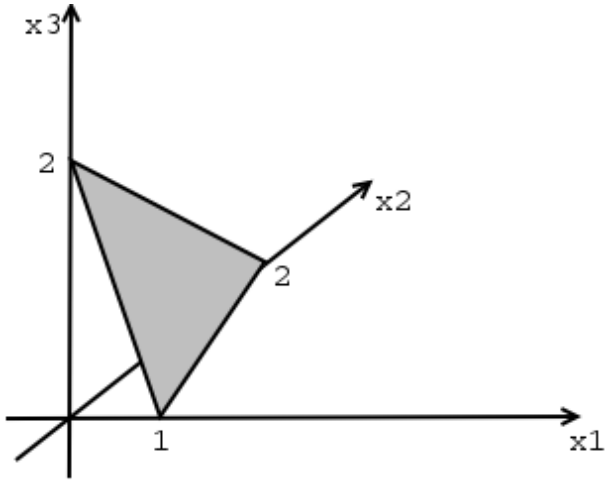
$x_2 \geq 0$

$x_3 \geq 0$

$$(2 \quad 1 \quad 1) \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = 2, n = 2, m = 1$$

$$A \cdot x = b$$

Basislösungen:



Spaltenauswahl, z.B. J={1}

$x_2 = 0$
$x_3 = 0$ $\} \rightarrow \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$-bfs $\qquad 2x_1 = 2 \rightarrow x_1 = 1$

Für J= $\{2\} \Rightarrow \begin{pmatrix} 0 \\ 2 \\ 0 \end{pmatrix}$-bfs

Für J= $\{3\} \Rightarrow \begin{pmatrix} 0 \\ 0 \\ 2 \end{pmatrix}$-bfs

Satz 1 Im Linearen Programm:

minimiere $c^T \cdot x$

$Ax = b$

$x \geq 0$

wobei:

$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$

A $m \times n$ Matrix

$b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$,

$c = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}$

ist $\omega \in \mathbb{R}^n$ eine bfs $\iff$ ω ist eine Ecke des Polyeders F der zulässigen Lösungen.

Beweis:

„$\Leftarrow$“ Sei ω eine Ecke von F. Sei $I = \{i | w_i > 0\}$

Fall 1: $E = \{ \underbrace{A_j}_{\text{j-ter Spaltenvektor von A}} | j \in I\}$ sind linear unabhängig.

I kann zu einer Basis $J \supset I$ erweitert werden. (Hinzunahme weiterer Spaltenvektoren)

Es ist $\omega_l = 0$ für $l \notin J \Rightarrow \omega_l$ bfs

$A \cdot \omega = b$, da $\omega \in F$

$\omega_l = 0, l \notin J$

Fall 2: E linear abhängig

Sei $\underbrace{z}_{\neq 0}$ Vektor mit $\sum_{j \in I} z_j \cdot A_j = 0$:

definiere $z_i = 0$ für $i \notin I$

$\Rightarrow A \cdot z = 0$

für hinreichend kleines $\Theta > 0$

$\left.\begin{array}{l}\omega_j + \Theta z_j \geq 0 \\ \omega_j - \Theta z_j \geq 0\end{array}\right\}$ für $j = 1, \ldots, n$

denn für $j \in I : \omega_j > 0$
für $j \notin I : z_j = 0$
$\omega \pm \Theta z_j = \underbrace{\omega_j}_{\forall \Theta : => 0}$

Sei $\omega^+ = \omega + \Theta z$
Sei $\omega^- = \omega - \Theta z$
dann $A\omega^+ = A\omega + \Theta \cdot \underbrace{A \cdot z}_{=0} = A\omega = b$ $A\omega^+ = A\omega - \Theta \cdot A \cdot z = A\omega = b$

damit $\omega^+, \omega^- \in F$
$\omega = 1/2\omega^+ + 1/2\omega w^-$
Kovenexkombination
$\Rightarrow \omega$ keine Ecke
WIDERSPRUCH!

„$\Longrightarrow$“ Sei ω bfs: angenommen ω keine Ecke, d.h. $\exists u, v \in F$ beide $\neq \omega$ und $\lambda, 0 < \lambda < 1$: $\omega = \lambda u + (1 - \lambda)v, J \subset 1, \ldots, n$ liefere Basis bezüglich ω

$\Rightarrow \omega_i = 0$ für $i \notin J$ und ω ist der *einzige* Vektor $\in F$ mit dieser Eigenschaft.
$\left.\begin{array}{l}\Rightarrow \exists i \notin J \text{ mit } u_i > 0 \\ \quad \text{sowieso } v_i \geq 0\end{array}\right\}$

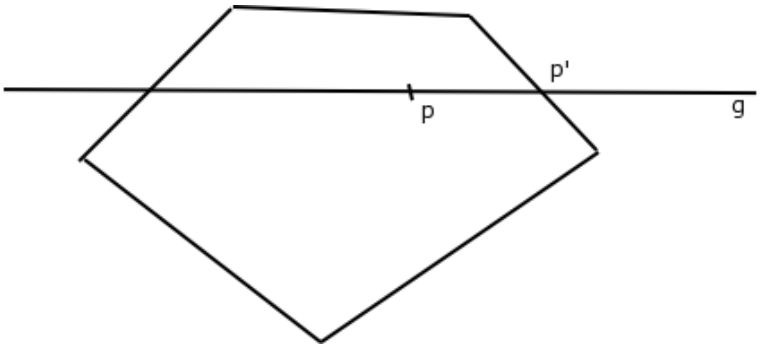
$\Rightarrow \lambda \underbrace{u_i}_{>0} + (1 - \lambda) \underbrace{v_i}_{\geq 0} > 0$

WIDERSPRUCH!, w_i bfs bezüglich J

Korollar: es gibt höchstens $\left(\begin{array}{c} n \\ m \end{array} \right)$ Ecken, denn jede Ecke entspricht einer Auswahl J von m der n Spaltenvektoren von A.

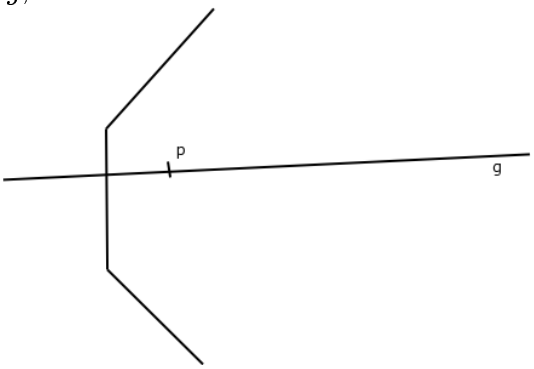
Satz 2 Voraussetzungen, wie bei Satz 1.
Sei $p \in F$. Dann ist entweder $c^T x$ auf F nach unten unbeschränkt oder es existiert eine Ecke v von F mit $c^T v = c^T p$,
d.h. die Zielfunktion nimmt ihr Optimum an der Ecke an oder garnicht.
ohne Beweis, daher:

Plausibilitätsbetrachtung: es gilt: falls Zielfunktion ihr Minimum annimmt, falls p keine Ecke ist und p liegt in einer k-dimensionalen Facette von $F, k > 0$, dann gibt es einen Punkt p' in einer niedrigerdimensionalen Facette mit $c^T p' \leq c^T p$,



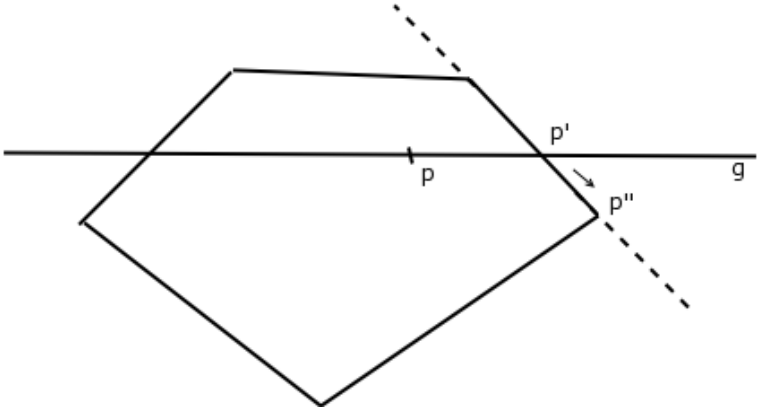
denn: lege Gerade g durch p,
Zielfunktion ist monoton auf g, laufe von p aus in Richtung in der sie fällt.

Fall a: der Rand von F wird nicht erreicht, d.h. $c^T \cdot x$ fällt immer wieder auf g, hat also in F kein Minimum.



Fall b: der Rand von F wird erreicht im Punkt p', deswegen $c^T p' \leq c^T p$ und p' auf niedriger dimensionalen Facette.

Der Satz folgt durch Induktion über die Dimension der Facette die p enthält.



aus den Sätzen 1 und 2:
Brute force - Algorithmus, um lineares Programmieren zu lösen:

- bestimme alle bfs (zulässige Basislösungen)

- nimm Minimum der Zielfunktion bei den bfs

dies ist kein effizienter Algorithmus:

Laufzeit: $\mathcal{O}\left(\begin{pmatrix} n \\ m \end{pmatrix} \cdot m^{\alpha}\right)$ wobei:
n Anzahl der Variablen
m Anzahl der Gleichungen
und $\begin{pmatrix} n \\ m \end{pmatrix}$ kann exponentiell in n sein.

5.2 Der Simplex-Algorithmus (Danzig)

Beispiel: maximiere $5x_1 + 4x_2 + 3x_3$

Nb.

$$\begin{pmatrix} 2 & 3 & 1 \\ 4 & 1 & 2 \\ 3 & 4 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \leq \begin{pmatrix} 5 \\ 11 \\ 8 \end{pmatrix}, x \geq 0$$

$\longrightarrow$ Standardform duch Schlupfvariablen x_4, x_5, x_6

umschreiben in folgende Form:

$x_4 = 5 - 2x_1 - 3x_2 - x_3$ $\qquad n = 6$
$x_5 = 11 - 4x_1 - x_2 - 2x_3$ $\qquad m = 3$
$x_6 = 8 - 3x_1 - 4x_2 - 2x_3$

minimiere $z = -5x_1 - 4x_2 - 3x_3$

erste bfs $\begin{pmatrix} 0 \\ 0 \\ 0 \\ 5 \\ 11 \\ 8 \end{pmatrix}$, $J = \{4, 5, 6\}$, $z = 0$

erhöhe $x_1 \rightarrow \frac{5}{2}$

$\Downarrow$

neue bfs $\begin{pmatrix} \frac{5}{2} \\ 0 \\ 0 \\ 0 \\ 1 \\ \frac{1}{2} \end{pmatrix}$, $J = \{1, 5, 6\}$, $z = \frac{-25}{2}$

$x_1 = \frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4$
$x_5 = 1 + 5x_2 + 2x_4$
$x_6 = \frac{1}{2} + \frac{1}{2}x_2 - \frac{1}{2}x_3 + \frac{3}{2}x_4$
$z = \frac{-25}{2} + \frac{7}{2}x_2 - \frac{1}{2}x_3 + \frac{5}{2}x_4$

1. Simplex-Tableau:

	x_1	x_2	x_3	x_4	x_5	x_6	b
	2	3	1	1	0	0	5
	4	1	2	0	1	0	11
	3	4	2	0	0	1	8
-z	5	4	3	0	0	0	0

Ein Pivot-Schritt d.h. Übergang von einer bfs zu einer anderen (besseren)

1. Finde das Maximum der untersten Zeile,
   falls ≤ 0: fertig, optimale Lösung gefunden
   sonst: weitermachen - die Spalte j in der das Maximum angenommen wird, heißt „Pivot-Spalte“

2. Für jede Zeile, deren Eintrag r in der Pivot-Spalte positiv ist:
   Bestimme die Zeile mit kleinstem $\frac{s}{r}$ („Pivot-Zeile“)
   s (Eintrag in der rechtesten Spalte)

3. Pivot-Zahl = Eintrag in Pivot-Zeile und Pivot-Spalte
   Teile die Einträge in der Pivot-Zeile durch die Pivot-Zahl.

4. Mache jeden anderen Eintrag in der Pivot-Spalte zu 0 durch Subtraktion eines geeigneten Vielfachen der Pivot-Zeile

2. Simplex-Tableau:

	x_1	x_2	x_3	x_4	x_5	x_6	b
	1	$\frac{3}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	0	$\frac{5}{2}$
	0	-5	0	-2	1	0	1
	0	$\frac{-1}{2}$	$\frac{1}{2}$	$\frac{-3}{2}$	0	1	$\frac{1}{2}$
-z	0	$\frac{-7}{2}$	$\frac{1}{2}$	$\frac{-5}{2}$	0	0	$\frac{-25}{2}$

3. Simplex-Tableau:

	x_1	x_2	x_3	x_4	x_5	x_6	b
	1	2	0	2	0	-1	2
	0	-5	0	-2	1	0	1
	0	-1	1	-3	0	2	1
-z	0	-3	0	-1	0	-1	-13

Optimum bei $z = -13$

angenommen bei $\begin{pmatrix} 2 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$, im ursprünglichen Problem bei $\begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix}$

Einzelprobleme:

1. Initialisierung: Wie findet man geeignete Start-bfs

2. Iteration: Findet man immer eine geeignete Variable (Eintritts-Variable) zum erhöhen und eine (Austritts-Variable), die statt ihrer aus der Basis herausfällt?

3. Termination: Könnte die Methode in eine unendliche Schleife laufen? Nach wie vielen Schritten hält sie?

Betrachtung:

1. Initialisierung: Wir wählten im Beispiel $x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij}x_j$ („dictionary“)
   Start-bfs $(\underbrace{0,\ldots,0}_{x_1\ldots x_n},\underbrace{b_1,\ldots,b_m}_{x_{n+1}\ldots x_m})$ Ursprung
   ist möglich,

   - falls $b_i \geq 0$, $i = 1,\ldots,m$
   - sonst: kan. minimiere $c^T * x$

   Nebenbedingung $\sum a_{ij}x_j \leq b_i$, $i = 1,\ldots,m$, $j = 1,\ldots,n$
   transformiere $\rightarrow$ Hilfsproblem, zusätzlichliche Variable x_0

   minimiere x_0
   Nebenbedingung $\sum_{j=1}^{n} a_{ij}x_j - x_0 \leq b_i$, $i = 1,\ldots,m$
   $A' = \begin{pmatrix} -1 & \\ \vdots & A \\ -1 & \end{pmatrix}$, $x \geq 0$
   zulässige Lösung des Hilfsproblemes: setze $x_0 = -b_j$ mit $-b_j$ maximal für $b_j < 0$
   dann gilt:
   ursprüngliches Problem hat überhaupt zulässige Lösung
   $\Leftrightarrow$ Hilfsproblem hat zulässige Lösung mit $x_0 = 0$
   $\Leftrightarrow$ Hilfsproblem hat optimalen Wert 0

   Lösung des Hilfproblemes:
   minimiere $w = x_0$
   $x_{n+i} = b_i - \sum_{j=1}^{m} a_{ij}x_j + x_0$ ursprüngliches dictionary, x_{n+1} Schlupfvariablen
   $x_0 \leftarrow \max\{-b_j | b_j < 0\}$ liefert zulässige Lösung
   dann Pivot-Schritte ausführen.
   Sobald x_0 als Austrittsvariable möglich ist: benutze es

   (a) falls x_0 nicht in Basis, also $w = 0$: fertig $\rightarrow$ 2. Phase
   (b) x_0 ist Basisvariable und $w \neq 0$: es gibt keine zulässige Lösung des ursprünglichen Problems

   Phase 2: Nach positiven Abschluss (Fall a) setzen wir im letzen dictionary überall $x_0 = 0$. Dies liefert Start-bfs (mit $b \geq 0$) und Start dictionary.

2. Iteration: Wahl der Eintrittsvariablen, z.B. irgendeines der größten (es gibt andere Strategien)
   Wahl der Austrittsvariablen - wähle irgendeine, aber das kann zu ∞-Schleife führen.

Eine andere Möglichkeit ist Blands Regel: Für die Eintritts- und Austrittsvariable wird immer die mit dem kleinsten Index bei mehreren Möglichkeiten gewählt. Es gilt, dass das Verfahren terminiert bei Verwendung von Blands Regel.

Laufzeit: 1 Pivot-Schritt ist polynomiell in n und $m \to \mathcal{O}(nm)$.
Wie viele Pivot-Schritte führt man durch? Wir laufen auf dem Polyeder über die Kanten und verkleinern die Zielfunktion.
Klie-Minty-Würfel hat exponentiell (in n) viele Ecken und es existiert ein Weg durch alle Ecken, wo die Zielfunktion immer verkleinert wird. Also im schlechtesten Fall exponentiell.
Empirische Beobachtungen an praktischen Problemen zeigen was anderes:

- etwa $\frac{3}{2}m$, selten mehr als $3m$ (Dantzig - 1947)
- Borgwardit, Smale - 1982: im Durchschnitt $\mathcal{O}(m)$
- Spielman, Teng - 2001: geglättete Analyse

Es gibt Polynozeit-Algorithmen:

- Khachian - 1979: Ellipsoid-Methode
- Karmarkar - 1984: innere-Punkt-Methode