Algorithmen & Komplexität I

Wintersemester 1994/1995

Inhaltsverzeichnis

1 Graphenalgorithmen **1**
1.1 Grundlagen . . . 1
1.2 Billigste Wege . . . 3
1.2.1 Billigste Wege von einem Knoten aus . . . 3
1.2.2 Warteschlangen . . . 5
1.3 Amortisierte Analyse . . . 8
1.3.1 Amortisierte Analyse für Fibonacci-Heaps . . . 9
1.4 Depth First Search (Tiefensuche) . . . 11
1.4.1 Wichtige Eigenschaften . . . 11
1.4.2 Starke Zusammenhangskomponenten . . . 13
1.5 Matchings . . . 16

2 Wörterbücher **23**
2.1 Hashing mit Verkettung . . . 23
2.2 Hashing mit offener Adressierung . . . 25
2.3 Universelles Hashing . . . 27
2.4 Perfektes Hashing . . . 29

3 Höhenbalancierte Bäume, Union Find **35**
3.1 (2,4)-Bäume . . . 35
3.2 Union Find . . . 38

4 Lineare Programmierung **43**
4.1 Simplexalgorithmus . . . 45
4.2 Dualität . . . 49
4.3 Alternative Darstellung . . . 49

5 NP-Vollständigkeit **53**
5.1 Weitere Komplexitätsklassen . . . 60

1 Graphenalgorithmen

1.1 Grundlagen

Definition 1.1:

Ein Paar (V, E) heißt *gerichteter Graph* $:\Leftrightarrow$ V ist endliche Menge von Knoten.
$E \subseteq V \times V$ ist eine Menge von Kanten.
Ein Element $e = (u, v) \in E$ heißt *Kante* von u nach v $(u \to v)$.
u ist Startknoten von e, v ist Zielknoten. v ist Nachbarknoten von u (v ist *adjacent* zu u).

Ein *Pfad* im Graph G ist eine Folge $(v_0, \ldots, v_k)$ von Knoten mit $k \geq 0$ und $(v_i, v_{i+1}) \in E$ für $0 \leq i \leq k-1$. Das ist ein Pfad von v_0 nach v_k. Die Länge des Pfades ist k. Falls $v_0 = v_k$ und $k \geq 1$, so heißt der Pfad *Kreis*. Falls $v_i \neq v_j$ für $i \neq j$, so heißt der Pfad einfach. Für einen Pfad von u nach v in G schreibe $u \overset{*}{\to}_G v$.
Ein gerichteter Graph ist zyklisch, falls es einen Kreis gibt, sonst azyklisch.

Der *Eingangsgrad* $indeg(v)$ eines Knoten v $:= \{(w, v) | (w, v) \in E\}$
Der *Ausgangsgrad* $outdeg(v)$ eines Knoten v $:= \{(v, w) | (v, w) \in E\}$

Ein Graph G heißt *Baum*, falls:

a) V enthält genau einen Knoten v_0 mit $indeg(v_0) = 0$

b) $\forall v \in V \backslash \{v_0\} : indeg(v) = 1$

c) G ist azyklisch

Beispiel 1.1: Topologisches Sortieren

Definition 1.2:

Sei $G = (V, E)$ ein gerichteter Graph.
Eine Abbildung $num : V \to \{1, \ldots, n\}$ mit $n = |V|$ heißt *topologische Sortierung*, falls $num(u) < num(v)$ $\forall (u, v) \in E$.

Lemma 1.1:

G besitzt genau dann eine topologische Sortierung, wenn G azyklisch ist.

Beweis:

„$\Rightarrow$“:
Sei G zyklisch. Sei $(v_0, \ldots, v_k)$, $k \geq 1$ ein Kreis. Für topologisches sortieren gilt: $num(v_0) < num(v_1) < \ldots < num(v_k) = num(v_0)$. Widerspruch.

„$\Leftarrow$“:
Behauptung: G enthält Knoten mit $indeg(v) = 0$
$|V| = 1$: klar, sonst $v \to v$ Kreis

$|V| > 1$: Entferne einen beliebigen Knoten v aus G.
Man erhält $G' = (V', E')$ mit $V' = V \backslash \{v\}$, $E' = E \cap (V' \times V')$.
G' ist azyklisch. Mit Induktionsannahme gilt:
G' enthält v' mit $indeg(v') = 0$. Entferne v' aus G und erhalte G''.
G'' enthält mit Induktionsannahme Knoten v'' mit $indeg(v'') = 0$.

Lies Beweis nochmals mit $v = v''$. Dann gilt:
In G ist entweder $indeg(v') = 0$ oder $indeg(v'') = 0$, weil $(v', v'') \notin E$ oder $(v'', v') \in E$.

Topologische Sortierung:

$|V| = 1$: klar

$|V| > 1$: Sei v ein Knoten mit $indeg(v) = 0$.
Entferne v und sortiere G' topologisch induktiv.
Sei $num' : V' \to \{1, \ldots, |V| - 1\}$ topologische Sortierung für G'.
Dann ergibt sich die topologische Sortierung für G mit

$$num(w) = \begin{cases} num'(w) + 1 & \text{falls} \quad w \neq v \\ 1 & \text{falls} \quad w = v \end{cases}$$

Grundalgorithmus

1. count$\leftarrow 0$;
2. `while` $\exists v \in V$ mit $indeg(v) = 0$ `do`
3. count++;
4. num(v)$\leftarrow$count;
5. streiche v und alle ausgehenden Kanten aus G;
6. `od`
7. `if` count$< |V|$ `then` ausgabe("G zyklisch") `fi`;

Problem: Test auf „Existenz eines Knotens mit $indeg() = 0$".
Lösung: Merke alle Knoten mit $indeg() = 0$ in Menge ZERO. Führe Buch über die Anzahl jeweils einmündender Kanten.

Algorithmus TopSort

01. ZERO$\leftarrow \emptyset$
02. `for all` $v \in V$ `do` INDEG$[v]\leftarrow 0$ `od`;
03. `for all` $e \in E$ `do` INDEG[target(e)]++ `od`;
04. `for all` $v \in V$ `do`;
05. `if` INDEG$[v]$=0 `then` ZERO$\leftarrow$ZERO$\cup\{v\}$ `fi`;
06. `od`;
07. count$\leftarrow$0;
08. `while` ZERO$\neq \emptyset$ `do`
09. wähle und streiche v aus ZERO;
10. count++;
11. num$[v] \leftarrow$count;
12. `for all` $(v, w) \in E$ `do`
13. INDEG$[w]$−−;
14. `if` INDEG$[w]$=0 `then` ZERO$\leftarrow$ZERO$\cup\{w\}$ `fi`;
15. `od`;
16. `od`;
17. `if` count$< |V|$ `then` ausgabe("G zyklisch") `fi`;

noch zu klären: wie implementiert man ZERO? Antwort: mit einem Stack.
Laufzeit:

$$O(|V| + |E| + \underbrace{\sum_{v} \sum_{w} (v, w) \in E)}_{|E|}$$

Korrektheit: klar, Verfeinerung von Lemma 1.

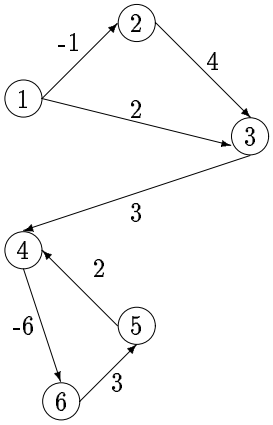
1.2 Billigste Wege

Definition 1.3:
Sei $G = (V, E)$ ein gerichteter Graph, $c : E \to \mathbb{R}$ eine Kostenfunktion. Sei $P = (v_0, \ldots, v_k)$ ein Pfad von $v_0 = v$ nack $v_k = w$, dann sind

$$c(P) = \sum_{i=0}^{k-1} c(v_i, v_{i+1})$$

die Kosten des Pfades von v nach w.
$dist(v, w) = inf\{c(P) |$ P ist Pfad von v nach $w\}$ heißt Entfernung.

Beispiel 1.2:



$\text{dist}(1, 2) = -1$
$\text{dist}(3, 1) = \infty$
$\text{dist}(1, 4) = -\infty$, da Pfad $(1, 3, (4, 6, 5)^i)$ Kosten $5 - i$ hat $\Rightarrow$ negativer Zyklus.

1.2.1 Billigste Wege von einem Knoten aus

Eingabe: Netzwerk $G = (V, E, c)$, $c : E \to \mathbb{R}$ und Knoten $s \in V$.
Ausgabe: $dist(s, v)$ $\forall v \in V$.

wichtig: die $dist()$-Funktion erfüllt Dreiecksungleichung:
$dist(s, v) \leq dist(s, u) + dist(u, v)$

Idee: Überschätze die $dist()$-Funktion und erniedrige sie dann schrittweise. Anfangs:

$$dist(s, v) = \begin{cases} \infty & \text{falls} \quad v \neq s \\ 0 & \text{falls} \quad v = s \end{cases}$$

Später: Falls $\exists (u, v)$ mit $dist(s, v) > dist(s, u) + c(u, v)$, dann erniedrige $dist(s, v)$ auf $dist(s, u) + c(u, v)$.

Grundalgorithmus

1. DIST[s]$\leftarrow 0$;
2. `for all` $v \in V \backslash \{s\}$ `do` DIST[v] $\leftarrow \infty$ `od`;
3. `while` $\exists e = (u, v) \in E$ mit DIST[v] >DIST[u] + $c(u, v)$ `do`
4. wähle eine solche Kante e;
5. DIST[v] $\leftarrow$DIST[u] + $c(e)$;
6. `od`;

Problem:

- Test in 3.
- welche Kante e in 4 ist am günstigsten

Idee: Merke in Menge U die Knoten, aus denen Kanten ausgehen, die die Dreiecksungleichung verletzen.

Invariante:

- Initialisierung: $U = \{s\}$
- v wird in U aufgenommen, wenn DIST[v] kleiner wird (5.)

Test auf Verletzung der Δ-Ungleichung wird zum Test: $U \neq \emptyset$
Wahl der Kante: Wähle beliebigen Knoten aus U, prüfe alle ausgehenden Kanten auf Δ-Ungleichung

Verfeinerung

```
01. DIST[s] ← 0;
02. for all v ∈ V\{s} do DIST[v] ← ∞ od;
03. U ← {s};
04. while U ≠ ∅ do
05.         wähle und streiche beliebiges u ∈ U
06.         for all (u, v) ∈ E do
07.             if DIST[v] >DIST[u] + c(u, v)
08.                then DIST[v] ←DIST[u] + c(u, v);
09.                     U ← U ∪ {v};
10.             fi;
11.         od;
12. od;
```

Lemma 1.2:

a) Falls $v \notin U$, so gilt für alle Kanten $(v, w) \in E$: DIST[w] $\leq$DIST[v] + $c(v, w)$

b) Sei $s = v_0, v_1, \ldots, v_k = v$ ein billigster Pfad von s nach v. Falls DIST[v] > $dist(s, v)$, dann $\exists v_i \in U$, $0 \leq i \leq k-1$ mit DIST[v_i] = $dist(s, v_i)$

c) Es existiert stets ein $v \in U$ mit DIST[v] = $dist(s, v)$

d) Wird in (5.) immer ein Knoten $u \in U$ mit DIST[u] = $dist(s, u)$ ausgewählt, so wird die while-Schleife n-mal ausgeführt

Beweis:

a) Induktion über Anzahl i der Iterationen

$i = 0$: Vor erstem Durchlauf gilt: $\forall v \notin U$: DIST[v] = ∞
$\Rightarrow$ Behauptung $\forall (v, w) \in E$.

$i > 0$: Induktionsannahme sagt: Behauptung gilt vor i-ter Iteration.
Betrachte $v \notin U$ danach.

1. Vorher war $v \notin U$, d.h. die Behauptung galt für $(v, w) \in E$ vor dem i-ten Durchlauf der Iteration.
   DIST[v] wird nicht verändert
   DIST[w] wird höchstens erniedrigt $\Rightarrow$ Behauptung a) gilt für $(v, w) \in E$ danach.
2. $v \in U$ zuvor. Wir testen alle $(v, w) \in E$ und verringern DIST[w] evtl., so daß die Δ-Ungleichung danach erfüllt ist.

b) Sei $v_0, \ldots, v_k$ ein billigster Pfad von s nach v.

$$\left.\begin{array}{l} DIST[s] = dist(s,s) = 0 \\ DIST[v] \geq dist(s,v) \end{array}\right\} \text{anfangs}$$

Sei i maximal mit DIST$[i] = dist(s, v_i)$ für $i \geq 0$.
Annahme:
$v_i \notin U \stackrel{a)}{\Rightarrow}$DIST$[v_i] + c(v_i, v_{i+1}) \geq$DIST$[v_{i+1}]$ und

$$\begin{array}{rcl} dist(s, v_{i+1}) & = & dist(s, v_i) + c(v_i, v_{i+1}) \\ & = & DIST[v_i] + c(v_i, v_{i+1}) \\ & \geq & DIST[v_{i+1}] \end{array}$$

$\Rightarrow dist(s, v_{i+1})$=DIST$[v_{i+1}]$ Widerspruch zu Wahl von i
$\Rightarrow v_i \in U$

c) folgt aus b)

d) Wird in (5.) Knoten $u \in U$ mit $dist(s,u) = DIST[u]$ gewählt, so wird er aus U gestrichen und nicht mehr aufgenommen $\Rightarrow$ nach $|V| = n$ Iterationen sind wir fertig. ❏

Problem: Finde $u \in U$ effizient mit $dist(s,u)$ =DIST$[u]$.

Lösung:

- allgemein schwierig
- für Spezialfälle einfacher:
  - azyklisch (gehe nach topologischer Sortierung vor)
  - nicht-negative Kantengewichte:
    ersetze (5.) durch (5'.) wähle/streiche $u \in U$ mit DIST$[u]$ minimal

Problem: Wie realisieren wir U?

1.2.2 Warteschlangen

Ein Objekt PQ vom Typ *priorityqueue* ist eine Menge von Elementen *pq.item*. Das Element $< k, i >$ bezeichnet *pq.item* mit Schlüssel k (vom Typ K) und Information i (vom Typ I). Eine Warteschlange besitzt lineare Ordnung auf den Schlüsseln.

Operationen

1. priorityqueue PQ.create: erzeugt ein Objekt vom Typ *priorityqueue*. PQ ist leere Warteschlange
2. bool PQ.empty: liefert true, falls PQ leer ist
3. int PQ.size: liefert Größe von PQ
4. void PQ.clear: entleert PQ
5. K PQ.key(*pq_item it*): liefert Wert *key* von Element *it*
6. I PQ.inf(*pq_item it*): liefert Wert *inf* von Element *it*
7. pq_item PQ.insert(K k, I i): fügt neues Element $< k, i >$ ein, liefert *it*
8. pq_item PQ.find_min: liefert Element mit kleinstem Schlüssel

9. pq_item PQ.delete_min: liefert und löscht Element mit kleinstem Schlüssel
10. pq_item PQ.decrease_key(it, j): ersetze Schlüssel k von it durch j

Dijkstra-Algorithmus

```
PQ : Warteschlange;
it : pq_item;
K : array[1..n] of pq_item;

01. PQ.create;
02. K[s]←PQ.insert(0, s);
03. DIST[s]← 0;
04. for all v ∈ V\{s} do
05.     DIST[v]← ∞; K[v]←PQ.insert(∞, v);
06. od;
07. while ¬ PQ.empty do
08.       u ←PQ.inf(PQ.delete_min);
09.       K[u]←nil;
10.       for all (u, v) ∈ E do
11.           if DIST[u]+c(u, v) <DIST[v] then
12.               DIST[v]←DIST[u]+c(u, v);
13.               PQ.decrease_key(K[v], DIST[v]);
14.           fi;
15.       od;
16. od;
```

Laufzeit:

$n = |V|$, $m = |E|$
$O(T_{create} + n(T_{insert} + T_{empty} + T_{delete_min}) + m \cdot T_{decrease_key})$

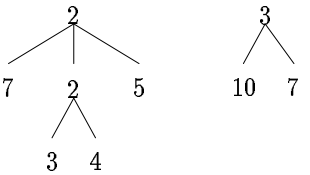
1. Implementierung durch doppelt verkettete Listen

delete_min teuer ($O(n)$), alles andere billig ($O(1)$).
Laufzeit: $O(1 + n(1 + 1 + n) + m \cdot 1) = O(n^2 + m)$
Falls $m = \Omega(n^2)$, so ist Implementierung optimal.
Falls $m = O(n)$, so ist Laufzeit quadratisch in Eingabegröße.

2. Fibonacci-Heaps

$O(n \cdot \log n + m)$
heap_ordered_tree: Knoten des Baums mit Schlüsseln beschriftet. $\forall v \neq$ Wurzel: key(v) $\geq$key(Elter(v))

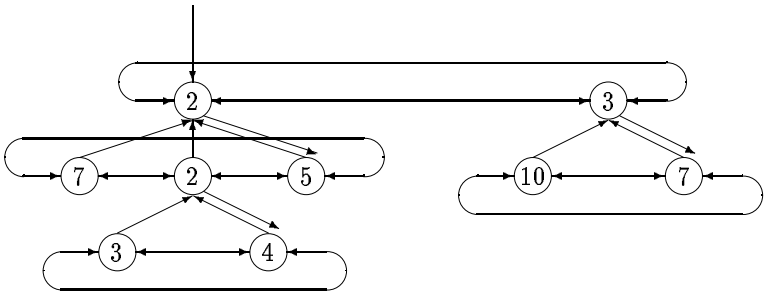
Beispiel 1.3:



Knoten wird realisiert durch

```
type node record key : K;
                 inf : I;
                 parent, child: ↑node;
                 leftsib, rightsib: ↑node;
                 mark: bool;rang: integer;
          end;
```

Geschwister bzw. Wurzeln bilden zirkuläre doppelt verkettete Liste.
rang(v) = # Kinder von v
$mark \in \{0, 1\}$
Fibonacci-Heap: Menge von heap ordered trees + Zeiger auf „kleinste“ Wurzel.



Operationen:
create: Min $\leftarrow$ nil: $O(1)$
find_min: Zugriff über Min-Zeiger: $O(1)$
insert(k, i):

1. erzeuge neuen Baum mit Knoten $v =< k, i >$
2. füge ihn in Wurzelliste ein
3. verändere evtl. Min-Zeiger: $O(1)$

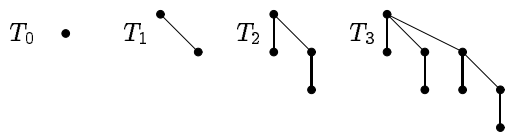
delete_min:

1. lies *key* von v, auf den Min-Zeiger zeigt
2. streiche v aus Wurzelliste
   füge alle Kinder von v in Wurzelliste ein: O(max. Rang)
3. `while` $\exists$2 Wurzeln mit gleichem Rang
   `do` „verschmelze“ die beiden `od`;
4. Finde neues MIN durch Durchlaufen der Wurzelliste: O(max. Rang). Laufzeit so, da alle Knoten in Wurzelliste unterschiedlichen Rang haben.

Lemma 1.3: :
Schritt 3 kann in Zeit O(1+max. Rang + #Verschmelzungen) realisiert werden.

Beweis: Übung

Wie groß ist der maximale Rang?
Insert + Verschmelze erzeugen Bäume spezieller Struktur, sog. binomische Bäume



Eigenschaften:

- Wurzel von T_i hat Rang i
- T_i hat 2^i Knoten
- Gesamtzahl der Knoten ist n
  $\rightarrow$ Wurzel des Gesamtbaums hat Rang $\log n$
- Nachfolgerknoten haben kleineren Rang

Lemma 1.4: :
In binomischen Bäumen ist max. Rang $\leq \log n$

1.3 Amortisierte Analyse

Beobachtung: Manche Operationen sind teuer, manche billig.

Wir interessieren uns dafür, wieviel n Operationen (im schlechtesten Fall) kosten. Sei $T(n)$ die maximale Zeit für n Operationen auf einer Datenstruktur $\Rightarrow$ amortisierte Zeit für eine Operation beträgt $\frac{T(n)}{n}$.

Beispiel:
Binärer Zähler A mit Operationen

- Setze Null
- Erhöhe um Eins

Zähler als String $< \ldots a_2 a_1 a_0 >$, mit $a_i \in \{0,1\}$. Länge: $\log A$ und $A = \sum_{i \geq 0} a_i \cdot 2^i$

Setze Null: $a_0 \leftarrow 0; A \leftarrow 0$; Kosten $O(1)$

Erhöhe um Eins: $a_0 \leftarrow a_0 + 1$;
$i \leftarrow 0$;
`while` $a_i = 2$ `do`
$a_i \leftarrow 0; a_{i+1} \leftarrow a_{i+1} + 1; i \leftarrow i + 1$;
`od`;

Kosten $O(1 + \#Iterationen)$

Bis n zu zählen kostet also:
Setze Null + n·(Erhöhe um Eins) bei $\log n$ Bits $\rightarrow O(n \log n)$
Aber: Zähler läuft nur selten $O(\log n)$ Stellen $\rightarrow$ Amortisierte Analyse

Jedem Zustand $D^{(k)}$ der Datenstruktur wird mit Potentialfunktion Φ eine reelle Zahl $\Phi(D^{(k)})$ zugeordnet.
$T_{amort}(\mathrm{Op}_i) := T_{tat}(\mathrm{Op}_i) + \Phi(D^{(i)}) - \Phi(D^{(i-1)})$
Dann ist:

$$\begin{aligned} \sum_{i=1}^{n} T_{amort}(\mathrm{Op}_i) &= \sum_{i=1}^{n} (T_{tat}(\mathrm{Op}_i) + \Phi(D^{(i)}) - \Phi(D^{(i-1)})) \\ &= \sum_{i=1}^{n} T_{tat}(\mathrm{Op}_i) + \Phi(D^{(n)}) - \Phi(D^{(0)}) \end{aligned}$$

$$\Rightarrow \quad \underbrace{\sum_{i=1}^{n} T_{tat}(\mathrm{Op}_i)}_{\text{das interessiert uns}} \quad = \quad \underbrace{\sum_{i=1}^{n} T_{amort}(\mathrm{Op}_i)}_{\text{das ist oft leicht abzuschätzen}} \quad -\Phi(D^{(n)}) + \Phi(D^{(0)})$$

(mit $\Phi(D^{(0)}) = 0$ und $\Phi(D^{(k)}) \geq 0 \Rightarrow \sum_{i=1}^{n} T_{tat}(\mathrm{Op}_i) \leq \sum_{i=1}^{n} T_{amort}(\mathrm{Op}_i)$)

Jetzt anwenden auf „Binären Zähler“

Sei $D^{(k)}$ Liste am Anfang der k-ten Operationen $< a_{k-1}, \ldots, a_0 >$
$T_{tat} = c_k = 1 +$ #Iterationen während k-ter Operation
$T_{amort} = \hat{c}$, unabhängig von k
Wir setzen $\Phi(D^{(k)}) = \sum_{i=0}^{k-1} a_i$ Anzahl 1'en zuvor
Es gilt: $\Phi(D^{(0)}) = 0$ und $\Phi(D^{(i)}) \geq 0 \; \forall i \geq 0 \Rightarrow \hat{c} = c_k + \Phi(D^{(k+1)}) - \Phi(D^{(k)})$, mit

$$\left.\begin{array}{ll} D^{(k)} & :< \ldots a_{i+2} a_{i+1} 0 \overbrace{111\ldots11}^{\text{i Stück}} \\ D^{(k+1)} & :< \ldots a_{i+2} a_{i+1} 1000\ldots00 > \end{array}\right\} \Rightarrow c_k = 1 + i$$

$\Rightarrow \Phi(D^{(k+1)}) - \Phi(D^{(k)}) = l + 1 - (l + i) = 1 - i$
$\Rightarrow \hat{c} = (1 + i) + (1 - i) = 2$
$\Rightarrow$ Gesamtlaufzeit:

$$\begin{aligned} \sum_{k=0}^{n-1} c_k &= \sum_{k=0}^{n-1} \hat{c} + \Phi(D^{(0)}) - \Phi(D^{(n)}) \\ &\leq \sum_{k=0}^{n-1} \hat{c} \\ &= 2 \cdot n = O(n) \end{aligned}$$

1.3.1 Amortisierte Analyse für Fibonacci-Heaps

Sei Φ die Anzahl der Wurzeln in jedem Schritt
$T_{amort}(\mathrm{Op}) = T_{tat}(\mathrm{Op})$ + Potentialänderung
Operationen *create, insert, find_min* einfach: O(1)

delete_min: Potentialerhöhung: Rang(min. Elements)-1-#Verschmelzungen
$T_{amort}(delte_min) = O($max Rang+#Verschmelzungen
+Rang(min)-1-#Verscmelzungen$) = O($max Rang$) = O(\log n)$

decrease_key:

a) Verringere Schlüssel auf v

b) Hänge v mit Unterbaum in Wurzelliste. Ändere evtl. Min-Zeiger
Aber: Bäume nicht mehr binomisch $\Rightarrow$ Schranke für max. Rang ungültig

c) Idee: Hänge auch Vorgängerknoten (+Unterbäume) oben in Wurzelliste (falls möglich)

```
if  Elter(v) existiert then errege (Elter(v));
mit errege(w):
while (w markiert) and (w nicht Wurzel) do
        streiche Kante(w, Elter(w));
        trage w in Wurzelliste ein;
        w ← Elter(w);
od;
markiere w;
```

Markieren geht mit „mark“, welches mit 'FALSE' initialisiert ist. Jeder Knoten verliert nur ein Kind. Beim Verlust des zweiten ist er markiert und wird selbst hochgehängt. Eine *decrease_key* - Operation kann also sehr teuer sein, aber amortisiert wird es billiger bleiben.

Lemma 1.5: : Der maximale Rang $\leq 1.4404 \log n$

Beweis:

Sei v beliebiger Knoten vom Rang i. Ordne Kinder nach Zeit, zu der sie nach v kamen. Sei w_j das j-te Kind.

Behauptung: Der Rang $(w_j) \geq j-2$

Beweis:

Als w_j mit v verschmolzen wurde, war Rang(w_j)=Rang(v) $= j-1$. w_j hat höchstens ein Kind verloren $\Rightarrow$ Rang(w_j) $\geq j-2$

Sei v ein Knoten vom Rang i. S_i minimale #Knoten im Unterbaum (incl. v).

$$\begin{aligned} S_0 &= 1 \\ S_1 &= 2 \\ S_i &= 2 + S_0 + S_1 + \cdots + S_{i-2}; \\ &\quad \text{das j-te Kind hat mindestens } S_{j-2} \text{ Knoten im Unterbaum} \\ &= 2 + \sum_{j=0}^{i-2} S_j \end{aligned}$$

Fibonacci Zahlen:

$F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}$. Es gilt: $F_{i+2} = 2 + \sum_{j=2}^{i} F_j$

Behauptung: $S_i \geq F_{i+2}$ für $i \geq 0$

Beweis: durch Induktion

Mit $F_{i+2} \geq \left(\frac{1+\sqrt{5}}{2}\right)^i = 1.618^i$ ist $S_i \geq 1.618^i$. Für den max. Rang r gilt:

$1.618^r \leq n$ und $r \leq \frac{\log n}{\log(1.618)} = 1.4404 \log n$. ❏

Amortisierte Analyse:

Potential zu #Wurzeln + 2*#markierte Knoten:
Analyse von *insert*, *find_min*, *delete_min* bleibt gleich.
decrease_key:
Tatsächliche Kosten: $O(k)$ mit k # markierte Knoten, die zu Wurzeln werden
Potentialänderung:

# Wurzeln erhöht um $k+1$:	$k+1$
# markierte Knoten um $k-1$ verringert:	$-2(k-1)$
	$-k+3$

$\Rightarrow T_{amort} = O(k - k + 3) = O(1)$

Fibonacci-Heaps erlauben:
insert, *find_min*, *decrease_key* in Zeit $O(1)$ und *delete_min* in Zeit $O(\log n)$.

Beachte: $delete(v) = decrease_key(v, -\infty) + delete_min$ ($\Rightarrow O(\log n)$)

Satz 1.1:

Fibonacci-Heaps erlauben es, kürzeste Wege mit nichtnegativen Kantenkosten in Zeit O(n log n + m) zu berechnen.

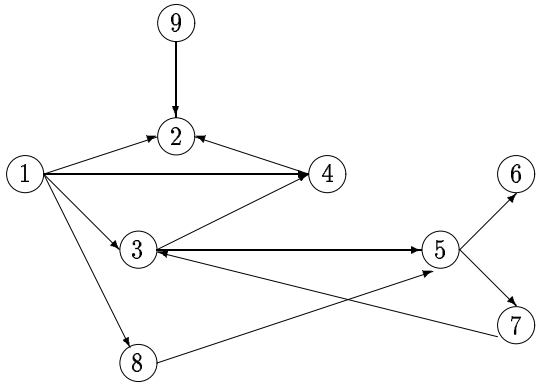
Andere Versionen:

- der allgemeine Fall (siehe Übung)
- all pairs shortest pathes
- one pair shortest path

1.4 Depth First Search (Tiefensuche)

1.4.1 Wichtige Eigenschaften

Dient zur Durchmusterung eines Graphen von Knoten aus s aus.



rekursiv:

```
procedure DFS(s);
          besucht[s] ← true; drucke(s);
          for all (s, v) ∈ E do;
              if not besucht[v] then DFS(v); fi;
          od;
```

Wir teilen die Kanten in vier Klassen, je nach Art des Besuchs. Sei (v, w) die gerade betrachtete Kante:

1. (v, w) gehört zu Baumkanten T, falls w noch nicht besucht
2. (v, w) gehört zu Vorwärtskanten F, falls w schon besucht und $v \xrightarrow{*}_T w$.
3. (v, w) gehört zu Rückwärtskanten B, falls w schon besucht und $w \xrightarrow{*}_T v$.
4. (v, w) gehört zu Querkanten C, falls w besucht, aber weder $v \xrightarrow{*}_T w$ noch $w \xrightarrow{*}_T v$

Jeder Knoten v enthält Nummern dfsnum[v] und compnum[v], wobei:

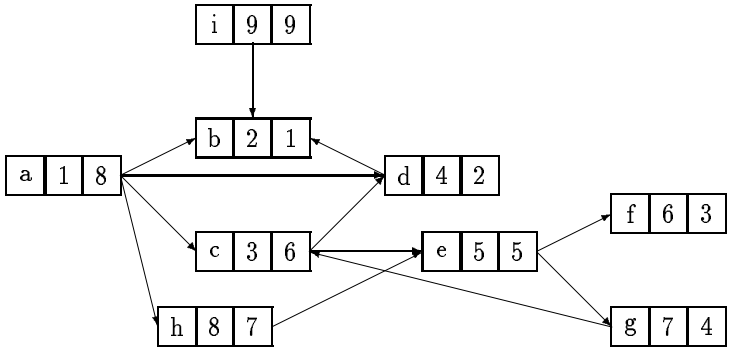
- dfsnum: Reihenfolge der rekursiven DFS-Aufrufe
- compnum: Abschlußreihenfolge der DFS-Aufrufe

Wir brauchen zwei Zähler $z1, z2$:

```
01. procedure DFS(v);
02.       besucht[v] ← true; z1 ← z1+1;
03.       dfsnum[v] ← z1;
04.       for all (v, w) ∈ E do
05.           if not besucht[w] then
06.               DFS(w);
07.           fi;
08.       od;
09.       z2 ← z2+1;
10.       compnum[v] ← z2;
```

Hauptprogramm

```
for all v ∈ V do besucht[v] ← false od;
    z1 ← z2 ← 0;
for all v ∈ V do
    if not besucht[v] then DFS(v) fi;
od;
```



Laufzeit:
$O(n+m)$, da

a) Einzellaufzeit eines Aufrufs (ohne Rekursion) ist $O(1+$ outdeg $(v))$

b) DFS wird für jeden Knoten einmal aufgerufen

$\Rightarrow O(\sum_{v \in V}(1+ \text{outdeg}(v)) = O(n+m)$

Lemma 1.6: (Eigenschaften)

1. Kantenklassen T, B, F, C bilden eine Partition von E
2. T entspricht dem Wald der rekursiven Aufrufe
3. $v \xrightarrow{*}_T w \Leftrightarrow$ dfsnum[v]<dfsnum[w] und compnum[w]< compnum[v]
4. Seien $v, w, z \in V$ mit $v \rightarrow^*_T w$, $(w, z) \in E$ und $\neg(v \rightarrow^*_T z)$, dann
   (a) dfsnum[z]<dfsnum[v]
   (b) $(w, z) \in B \cup C$
   (c) compnum[z]>compnum[v]$\Leftrightarrow (w, z) \in B$
   (d) compnum[z]<compnum[v]$\Leftrightarrow (w, z) \in C$
5. Sei $(v, w) \in E$ ($(v, w) \in T \cup F \Leftrightarrow$ dfsnum[v]<dfsnum[w].

6. Sei $(v,w) \in E$ $(v,w) \in B \Leftrightarrow$ dfsnum[v]>dfsnum[w] und compnum[v]<compnum[w]

7. Sei $(v,w) \in E$ $(v,w) \in C \Leftrightarrow$ dfsnum[v]>dfsnum[w] und compnum[v]>compnum[w]

Beweis:

a), b) klar durch Algorithmus

c)
$v \rightarrow_T^* \Leftrightarrow$ Aufruf DFS(w) geschachtelt in DFS(v)
$\Leftrightarrow$ dfsnum[v] <dfsnum[w] und compnum[v] >compnum[w]

d)
$v \rightarrow_T^* w \Rightarrow$ dfsnum[v] < dfsnum[w]
$(w,z) \in E \Rightarrow$ DFS(z) wird aufgerufen, bevor DFS(w) endet, und auch bevor DFS(v) endet.

$\neg(v \rightarrow_T^* z) \Leftrightarrow$ Aufruf DFS(z) *nicht* in DFS(v) geschachtelt
$\Rightarrow$ DFS(z) startet *vor* Aufruf DFS(v)
$\Rightarrow$ dfsnum[z] < dfsnum[v] (i)
$\Rightarrow (w,z) \in B \cup C$, da $\neg(w \rightarrow_T^* z)$ (ii)

$(w,z) \in B \Leftrightarrow z \rightarrow_T^* w$ nach Definition von B
$\Leftrightarrow z_T^* v$, weil T Baum bildet
$\Leftrightarrow$ compnum[z] > compnum[v] (iii)
(iv) folgat aus $(ii), (iii)$

e)
„$\Rightarrow$“: $(v,w) \in T \cup F \Rightarrow v \rightarrow_w^* \Rightarrow$ dfsnum[v] < dfsnum[w]
„$\Leftarrow$“: Sei dfsnum[v] < dfsnum[w].
Da (v,w) existiert, wird DFS(w) vor Abschluß von DFS(v) aufgerufen,
d.h. echte Schachtelung$\Rightarrow v_T^* w \Rightarrow (v,w) \in T \cup F$

f), g) folgen aus d) und c)

Sei G azyklisch. Es gibt keine Rückwärtskanten.
$\forall (v,w) \in E$: compnum[v]>compnum[w], d.h. $n+1-$compnum[v] für alle $v \in V$ ist topologische Sortierung.

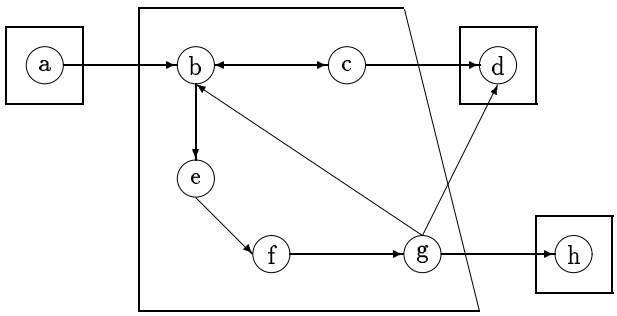
1.4.2 Starke Zusammenhangskomponenten

Definition 1.4:

a) Ein gerichteter Graph $G = (V,E)$ ist stark zusammenhängend $\Leftrightarrow \forall v, w \in V$: $v \rightarrow_E^* w$

b) Maximale stark zusammenhängende Teilgraphen von G heißen *starke Zusammenhangskomponenten* von G (SZKs).

Idee:
Erweitere DFS. Sei $G' = (V', E')$ der bisher erforschte Teilgraph von G. Berechne jeweils die SZK von G'. Starte mit $V' = \{a\}, E' = \emptyset$, SZKs= $\{\{a\}\}$. Sei (v,w) die nächste Kante in DFS. Dann ist $v \in V'$. Ist $w \notin V'$, so ist $(v,w) \in T$. Erweitere SZKs um $\{w\}$. Ist $w \in V'$, vereinige mehrere SZK zu einer (falls $(v,w) \in B \cup C$). Aber wie?



Bezeichnungen:

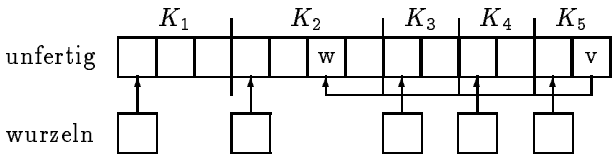
- SZK K heißt abgeschlossen, wenn alle DFS-Aufrufe für Knoten in K abgeschlossen sind
- Wurzel von K ist der Knoten mit kleinster dfsnum in K
- „unfertig“ ist eine Folge von Knoten, deren DFS aufgerufen, aber deren SZK noch nicht abgeschlossen ist (sortiert nach dfsnum)
- „wurzeln“ ist eine Folge der Wurzeln der noch nicht abgeschlossenen SZKs (sortiert nach dfsnum)

Beobachtung:

1. $\forall v \in$ unfertig: $v \overset{*}{\rightarrow}_E g$
2. $\nexists (v,w) \in E$ mit v in abgeschlossener SZK und w in nichtabgeschlossener SZK

Kanten aus g
$(g,d) \in C$: tue nichts, d liegt in abgeschlossener Komponente
$(g,b) \in B$: vereinige SZKs: $\{b,c\},\{e\},\{f\},\{g\}$
$(g,h) \in T$: erzeuge neue SZK $\{h\}$
Beim Abschluß von DFS(v) teste, ob v Wurzel. Wenn ja, gib v als Wurzel der SZK + Knoten der SZK aus und streiche sie aus unfertig und wurzeln.
Operationen auf unfertig/wurzeln erlauben Implementierung als Keller (Stack).

Allgemein: Vereinigung vom Komponenten



Vereinigung K_2, K_3, K_4, K_5 durch pop($wurzeln$) bis top($wurzeln$) = Wurzel der SZK von w

Korrektheit:

Invarianten:

I1: $\nexists (v,w) \in E$ mit v in einer abgeschlossener und w in einer nichtabgeschlossener SZK

I2: Nichtabgeschlossene SZKs liegen auf einem Pfad, insbesondere liegen ihre Wurzeln auf Baumpfad

I3: Die Knoten jeder nicht abgeschlossenen SZK K_i bilden ein Intervall in „unfertig“.
Erstes Element des Intervalls ist Wurzel von K_i.

z.z.: Invarianten bleiben beim Einfügen neuer Kanten erhalten

1. $(v, w) \in T$: push$(w, unfertig)$, push$(w, wurzeln)$;
   I1: keine SZK abgeschlossen
   I2: sei r Wurzel von SZK von v. Dann gilt $r \stackrel{*}{\rightarrow}_T v$ und $(v, w) \in T$, damit $r \stackrel{*}{\rightarrow}_T w$.
   I3: trivial

2. $(v, w) \notin T$:

   a) w in abgeschlossener SZK: tue nichts

   b) w in nichtabgeschlossener SZK. Vereinige oberste SZKs bis zu der, in der w liegt.

   I1: keine SZK abgeschlossen
   I2: in a) passiert nichts, in b) wird Pfad verkürzt
   I3: Löschen der obersten Wurzel vereinigt Intervalle zu einem neuen ⇒ Dieses eine Intervall bildet neue SZK

```
01. procedure DFS(v)
02.     count ← count+1; besucht[v] ← true; dfsnum[v] ← count;
03.     push(v, unfertig); push(v, wurzeln);
04.     for all (v, w) ∈ E do
05.         if not besucht[w] then DFS(w)
06.            else if w ∈ unfertig
07.                  then while dfsnum[top(wurzeln)] > dfsnum[w]
08.                       do pop(wurzeln) od;
09.            fi;
10.         fi;
11.     od;
12.     if v=top(wurzeln) then
13.        repeat w ← pop(unfertig);
14.               print(w);
15.        until v = w;
16.        pop(wurzeln)
17.     fi;
```

Bemerkungen:

- Implementierungen von „w ∈ unfertig“: bool'sches Feld ∈ unfertig
- jeder Knoten wird höchstens 1 mal nach unfertig/wurzeln aufgenommen

Satz 1.2:

SZKs eines gerichteten Graphen können in Zeit $O(n + m)$ berechnet werden.

1.5 Matchings

(insbesondere *maximales Matching* in bipartiten Graphen)
Anwendung: Scheduling

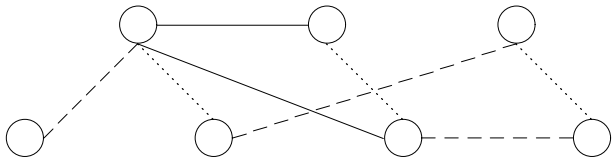
Definition 1.5: Sei $G = (V, E)$ ein ungerichteter Graph

1. $M \subseteq E$ heißt *Matching*, falls keine zwei Kanten aus M einen gemeinsamen Endpunkt haben.

2. Eine Kante $e \in E$ heißt *paarend*, falls $e \in M$. Ein Knoten heißt *gepaart*, falls er Endpunkt einer Kante aus M ist, sonst *frei*.

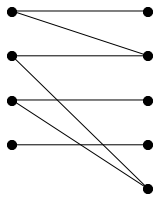
3. Ein Matching ist *maximal*, falls $|M| \geq |M'|$ für alle Matchings M'.

   Beispiel 1.4:

   $|M| = 3$, gepunktete und getrichelte Linien = maximales Matching

4. G heißt *bipartit*, falls $V = A \cup B$ und $A \cap B = \emptyset$ und $\forall e = (u, v) \in E : u \in A$ und $v \in B$ oder $u \in B$ und $v \in A$.

5. Ein einfacher Pfad $(v_1, v_2, \ldots, v_{2k})$ heißt *erweiternd* für ein Matching M, falls

   - v_1 und v_{2k} frei
   - Kanten liegen abwechselnd in M und $E \backslash M$

Idee: Vergrößere M, indem auf erweiterndem Pfad paarende und nichtpaarende Kanten ausgetauscht werden. Wir erhalten ein Matching M' mit $|M'| = |M| + 1$.

Formale Operation *symmetrische Differenz* $\oplus$:
$M \oplus P = (M \cup P) \backslash (M \cap P) = (M \backslash P) \cup (P \backslash M)$

Algorithmus: Finde jeweils einen erweiterten Pfad P für M, solange es ihn gibt:
$M \leftarrow M \oplus P$

Beobachtung: Knotendisjunkte Pfade $P_1, \ldots, P_k$ können gleichzeitig verarbeitet werden:

$$M \leftarrow M \oplus P_1 \oplus P_2 \oplus \cdots \oplus P_k$$

Matchinggröße wächst um k, man braucht nicht so oft suchen.

Lemma 1.7:
Seien M, M' Matchings eines Graphen $G = (V, E)$ mit $|M| = r$, $|M'| = s$ und $s > r$.
Dann gilt:
$M \oplus M'$ enthält mindestens $s - r$ knotendisjunkte erweiternde Pfade für M in G.

Beweis: Betrachte $\bar{G} = (V, M \oplus M')$
Jeder Knoten in $\bar{G}$ ist ein Endpunkt von höchstens einer Kante in $M \backslash M'$ und höchstens einer aus $M' \backslash M \Rightarrow \text{outdeg}_{\bar{G}}(v) \leq 2 \; \forall v$.
$\bar{G}$ zerfällt in folgende Komponenten:

1. isolierte Knoten
2. Pfade
3. Kreise

Pfade/Kreise bestehen abwechselnd aus Kanten aus $M \backslash M'$ und $M' \backslash M$.
Seien $C_1, \ldots, C_k$ die Komponenten von $\bar{G}$ mit $C_i = (V_i, E_i)$ und $\delta(C_i) = |E_i \cap M'| - |E_i \cap M|$. Dann ist $\delta(C_i) = \{0, -1, 1\}$.
Es gilt: $\delta(C_i) = 1 \Leftrightarrow C_i$ ist erweitender Pfad für M.
$\sum_{i=1}^{k} \delta(C_i) \leq \#$ erweiternder knotendisjunkter Pfade für M.

$$\begin{aligned}
\sum_{i=1}^{k} \delta(C_i) &= \sum_{i=1}^{k} |E_i \cap M'| - \sum_{i=1}^{k} |E_i \cap M| = |M' \backslash M| - |M \backslash M'| \\
&= |M' \backslash M| + |M' \cap M| - |M' \cap M| - |M \backslash M'| \\
&= |M'| - |M| \\
&= s - r
\end{aligned}$$

Korollar 1.1: M maximal $\Leftrightarrow \nexists$ erweiternder Pfad für M in G.

Korollar 1.2:
Sei M nichtmaximales Matching für G mit $|M| = r$ und M' maximal mit $|M'| = s$.
Dann gibt es einen erweiternden Pfad der Länge $\leq 2 \cdot \left\lfloor \frac{r}{s-r} \right\rfloor + 1$.

Beweis:
$M \oplus M'$ enthalten $\geq s - r$ Pfade, die für M erweiternd sind. Diese Pfade enthalten $\leq r$ Kanten aus M.
$\Rightarrow$ einer davon hat $\leq \left\lfloor \frac{r}{s-r} \right\rfloor$ Kanten aus M.
$\Rightarrow$ seine Länge ist $\leq 2 \cdot \left\lfloor \frac{r}{s-r} \right\rfloor + 1$.

Korollar des Korollars: Einsetzen ergibt für $|M| < \frac{|M'|}{2}$, daß es in M einen erweiternden Pfad der Länge 1 gibt.

Wir berechnen kürzeste erweiternde Pfade und erhalten:
eine Folge $M_0, M_1, \ldots, M_i, \ldots$ und eine Folge von Pfaden $P_0, P_1, \ldots$ mit $M_0 = \emptyset, M_{i+1} = M_i \oplus P_i$. Wir werden zeigen:

- Folge $P_0, P_1, \ldots, P_{s-1}$ enthält $\leq 2 \cdot \lfloor \sqrt{s} \rfloor + 2$ verschiedene Längen
- Pfade gleicher Länge sind knotendisjunkt
- Längen sind monoton wachsend

Lemma 1.8:
Sei M Matching, P kürzester erweitender Pfad für M. Sei P' erweitender Pfad für $M \oplus P$. Dann gilt: $|P'| \geq |P| + |P \cap P'|$.

Beweis: Sei $M' = M \oplus P \oplus P'$.

Dann gilt: $|M'| = |M| + 2$. Es gilt nach Lemma: $M \oplus M'$ enthält ≥ 2 knotendisjunkte erw. Pfade für M. Seien dies die Pfade P_1, P_2.
$M \oplus M' = M \oplus (M \oplus P \oplus P') = \underbrace{M \oplus M}_{\emptyset} \oplus (P \oplus P') = P \oplus P'$.

$\Rightarrow |P \oplus P'| = |M \oplus M'| \geq |P_1| + |P_2|$
Außerdem gilt: $|P_1| \geq |P| und |P_2| \geq |P| \Rightarrow |P_1| + |P_2| \geq 2|P|$
$\Rightarrow |P| + |P'| \geq 2|P|$
$\Rightarrow |P| + |P'| - |P \cap P'| \geq 2|P|$
$\Rightarrow |P'| \geq |P| - |P \cap P'|$ ❏

Korollar 1.3:
$|P_i| \leq |P_{i+1}|$, da $|P_{i+1}| \geq |P_i| + |P_i \cap P_{i+1}| \geq |P_i|$

Lemma 1.9: $\forall i, j$ mit $i \neq j$ und $|P_i| = |P_j| : P_i, P_j$ knotendisjunkt.

Beweis:

Annahme, es gibt i, j mit $i < j$ und $|P_i| = |P_j|$ und P_i, P_j nicht knotendisjunkt. Wähle i, j so, daß $|j - i|$ minimal. Es gilt $\forall k$, $i < k < j$:

- P_k knotendisjunkt zu P_i und P_j
- $|P_k| = |P_i| = |P_j|$

Da nach Lemma gilt: $|P_j| \geq |P_i| + |P_i \cap P_j| \rightarrow$ Schnitt muß leer sein
$\Rightarrow P_i$ und P_j sind kantendisjunkt
$\Rightarrow P_i, P_j$ haben gemeinsamen Knoten v.
Da P_i, P_k knotendisjunkt für $i < k < j$, gilt $(u, v) \in M_{i+1}, \in M_{i+2}, \ldots, \in M_j$
$\Rightarrow (u, v) \in P_j$, da $v \in P_j$
$\Rightarrow P_i, P_j$ nicht kantendisjunkt. ❏

Korollar 1.4:
Sei $i < j$ und $|P_i| = |P_j|$. Dann ist auch P_j kürzester erweitender Pfad für M_i.

Lemma 1.10:
Sei M_s ein maximales Matching, $|M_s| = s$. In der Folge $P_0, P_1, \ldots, P_{s-1}$ gibt es höchstens $2 \cdot \lfloor \sqrt{s} \rfloor + 2$ verschieden lange Pfade.

Beweis:
Sei $r = \lfloor s - \sqrt{s} \rfloor$ und M_r das dazugehörende Matching $|M_r| = r$. Dann gilt (Korollar):

$$\begin{aligned} |P_r| &\leq 2 \cdot \lfloor \frac{r}{s-r} \rfloor + 1 \\ &\leq 2 \cdot \frac{s - \sqrt{s}}{s - \lfloor s - \sqrt{s} \rfloor} + 1 \\ &\leq 2 \cdot \frac{s - \sqrt{s}}{\sqrt{s}} + 1 \\ &\leq 2 \cdot (\sqrt{s} - 1) + 1 \\ &\leq 2 \cdot \lfloor \sqrt{s} \rfloor + 1 \end{aligned}$$

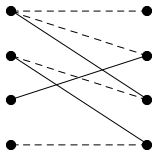
Für alle $i \leq r : |P_i| \leq 2 \cdot \lfloor \sqrt{s} \rfloor + 1$ und $|P_i|$ ungerade. Es gibt nur $\lfloor \sqrt{s} \rfloor + 1$ ungerade Werte für diese $|P_i|$. Es gibt nur noch $s - r = \lceil \sqrt{s} \rceil$ Pfade mit höherem Index als r, also höchstens $\lfloor \sqrt{s} \rfloor + 1$. Damit insgesamt: $\lfloor \sqrt{s} \rfloor + 1 + \lfloor \sqrt{s} \rfloor + 1$ verschiedene Werte $= 2 \cdot \lfloor \sqrt{s} \rfloor + 2$. ❏

Idee: Behandle alle kürzesten erw. Pfade (gleiche länge, knotendisjunkt) gleichzeitig

$M \leftarrow \emptyset$
`repeat`
    l(M) $\leftarrow$ Länge des kürzesten erw. Pfades in M
    finde maximale Menge S von Pfaden
    $S = \{Q_1, \ldots, Q_t\}$ mit
    a) Q_i ist erw.
    b) $|Q_i|$ =l(M)
    c) alle Q_i knotendisjunkt
    $M \leftarrow M \oplus Q_1 \oplus Q_2 \oplus \cdots \oplus Q_t$
`until` $S = \emptyset$

Problem: Finde max. Menge der knotendisjunkten kürzesten erw. Pfade. Kompliziert im allgemeinen, einfach für *bipartite Graphen*

Beispiel 1.5:



Richte Kanten so, daß Kanten in M von rechts nach links und die anderen von links nach rechts verlaufen. Ein erweiterter Pfad sieht so aus, daß erster freier Knoten in L ist, letzter freier Knoten ist in R.

Kanten werden gerichtet durch: $e \in M \Leftrightarrow$ source(e) in R

Finden kürzester Pfade

Idee:
Starte in freien Knoten aus L. Teile G in Schichten ein $(L_1, L_2, \ldots)$, so daß

1. $L_1 = \{v | v$ freier Knoten in $L\}$

2. $L_{i+1} = \{v | v \notin L_1 \cup L_2 \cup \cdots \cup L_i \wedge \exists (u,v) \in E \text{ mit } u \in L_i\}$
3. $k = \min\{i | L_i \text{ enthält freie Knoten aus } R\}$

Alle Kanten (u,v) mit $u \in L_i, v \in L_{i+1}$ nennen wir rot, falls $1 \leq i \leq k$. Sei $\bar{G}$ der von roten Kanten aufgespannte Teilgraph. Dann gilt:

1. Knoten aus ungeraden Schichten $(L_1, L_3, \ldots, L_{k-1})$ sind aus L.
2. die aus geraden Schichten $(L_2, L_4, \ldots, L_{2k})$ aus R.
3. $\bar{G}$ ist azyklisch
4. jeder einfache gerichtete Pfad in $\bar{G}$, der in freien Knoten aus L startet und in freiem Knoten aus R endet, entspricht genau einem kürzesten erw. Pfad für M in G (beachte: in $\bar{G}$ sind einige Knoten weggefallen).

Wir machen aus $\bar{G}$ einen *s-t-Graphen*:
erweitere $\bar{V}$ um $\{s,t\}$, verbinde s mit freien Knoten aus L und t mit freien Knoten aus R.

Kürzester erw. Pfad entspricht kürzestem Weg von s nach t. Menge S entspricht maximaler Menge von ,bis auf s und t, knotendisjunkten Pfaden von s nach t.

Algorithmus

Init:
richte alle Kanten von L nach R. (*$M \leftarrow \emptyset$*)
füge (s,v) ein $\forall v \in L$ (* Anfangs sind alle Konten frei *)
füge (w,t) ein $\forall w \in R$ (* Anfangs sind alle Knoten frei *)

Iterative Erweiterung von M:

Bestimme $\bar{G}$;
`while` t von s aus erreichbar `do`
    bestimme max. Menge S von
    knotendisjunkten Wegen von s nach t in $\bar{G}$
    `for all` Pfade $P \in S$ `do`
        streiche erste/letzte Kante; (* Knoten sind nicht mehr frei *)
        drehe Richtung aller übrigen Kanten auf P um (* Erweiterung von M*)
    `od`;
    Bestimme $\bar{G}$ neu;
`od`;

$\bar{G}$ wird konstruiert mit BFS von s aus durch Berechnung der kürzesten Wege.

$$v \in L_i \Leftrightarrow \text{dist}(v) = i \text{ und } i < \text{dist}(t)$$

$$(v,w) \text{ rot } \Leftrightarrow \text{dist}(v) = \text{dist}(w) - 1 \text{ und } (\text{dist}(w) < \text{dist}(t) \text{ oder } w = t)$$

Wir suchen max. Menge knotendisj. Pfade von s nach t in $\bar{G}$ mit DFS. Pfade finden wir nacheinander und zwar immer solche, die disjunkt sind zur Menge der bisher gefundenen Pfade. Speichern Pfade implizit ab durch Feld pred, das für jeden Knoten seinen Vorgänger angibt. Pfad ist angegeben durch letzte Kante von (u,t) oder durch *nil*.

```
function next_path(G, v, t, pred, dist);
    result← 0;
    while result=0 ∧ ∃ unbenutzte Kante aus v do
          markiere (v, w) als benutzt;
          if dist(v) = dist(w) − 1 und dist(w) ≤dist(t) then
             if (w = t) then result← (v, w);
                else if pred[w] = 0 then
                          pred[w]← v;
                          result←next_path(G, w, t, pred, dist)
                fi;
             fi;
          fi;
    od;
    return(result);
```

Zur Implementierung:

Wir halten G in einer Adjazenzlistendarstellung, d.h. zu jedem Knoten v gibt es eine Liste seiner Nachbarn w. Wir durchlaufen die Liste und merken uns die aktuelle Kante mit Zeiger $p(v)$.

Gesamtprogramm

```
01. for all v ∈ V do layer[v] ← 0; pred[v] ← 0; od;
02. BFS(G, s, layer) (* dist-Werte = layer *)
03. while layer(t) > 0 do
04.       while (e ←next_path(G, s, t, pred, layer)) ≠ nil do
05.             hänge e an Liste EL an; (* Liste letzter Kanten *)
06.       od;
07.       while EL ≠ ∅ do
08.             lösche oberstes Element e aus EL
09.             x ←source(e);
10.             while pred[x] ≠ s do
11.                   drehe(pred(x), x) um; x ←pred(x) (* erweitere Matching *)
12.             od;
13.             lösche e und (s, x) aus Ḡ (* beide jetzt unfrei *)
14.       od;
15.       (* neue Init *)
16.       for all v ∈ V do
17.           layer[v] ← 0;
18.           pred[v] ← 0;
19.       od;
20.       BFS(Ḡ, s, layer);
21. od;
```

Laufzeit:

- Äußere while-Schleife wird $O(\sqrt{n})$ mal ausgeführt (für jede Dist. einmal)
- BFS geht in $O(n+m)$
- Bestimmen disjunkter Pfade: $O(n+m)$ durch DFS
- Kanten in Listen umdrehen: $O(n+m)$

Satz 1.3:

Maximales Matching in bipartiten Graphen kann in $O(\sqrt{n}(n+m))$ berechnet werden.

Schwierigkeiten im allg. Fall: wie findet man erweiternden Pfad?
Insbesondere bei Zykel ungerader Länge. Darf man Knoten nur $1\times$ ansehen, findet man $\{a, c, d, f, e, g\}$ nicht, da e vorher schon betrachtet. Darf man mehrmals, so findet man eventuell a, c, d, e, f, d, c, b bei dem Erweiterung keinen Sinn macht.

Satz 1.4:

Maximales Matching in nichtbipartiten Graphen kann in Zeit $O(\sqrt{n} \cdot m)$ berechnet werden, bzw. $O(\min\{n^{2.5}, \sqrt{n} \cdot m \cdot \log n\})$.

2 Wörterbücher

Informationen werden mittels Schlüssel abgelegt.

Anwendung:

- Bankkonten
- Katalogisierung in Bibliotheken
- Symboltabellen für Compiler

Operationen:

- $Access(x, S)$
- $Insert(x, S)$
- $Delete(x, S)$

S = die zu verwaltende Menge
U = Universum = $[0 \ldots N-1]$, $S \subseteq U$, $|S| = n$

1. Realisierung:
Feld A der Länge N, $A[i] = 1 \Leftrightarrow i \in S$
Laufzeit: $O(1)$, Platzbedarf ist riesig: $O(N)$
Ziel:
Platzbedarf: $\approx O(|S|)$, Laufzeit $O(1)$

Hashing
brauchen Hashtafel T[$0..m-1$] und Hashfunktion $h : U \leftarrow [0..m-1]$
$x \in S : x \rightarrow T[h(x)]$

Beispiel 2.1: $N = 50$, $m = 3$, $S = \{2, 21\}, h(x) = x \bmod 3$

0	21
1	
2	2

Problem: Kollision, wenn $h(x) = h(y)$ mit $x \neq y \in S$

Ausweg:

1. Hashing mit Verkettung
   T[i] repräsentiert verkettete Liste von Elementen x mit $h(x) = i$
2. Hashing mit offener Adressierung
   Statt einer Hashfunktion gibt es eine Folge von Hashfunktionen, die durchforstet werden, bis ein freier Platz gefunden wurde
3. Universelles Hashing
4. Perfektes Hashing: wähle Hashfunktion geeignet

2.1 Hashing mit Verkettung

i-te Liste enthält alle $x \in S$ mit $h(x) = i$.

Operationen:

1. berechne h(x)
2. Suche (Einfüge/Streiche) x in der Liste T[h(x)]

im schlechtesten Fall: Laufzeit $O(n)$ (miserabel), im Mittel viel besser.

Annahmen:

1. $h(x)$ kann in $O(1)$ ausgewertet werden.
2. $|h^{-1}(i)| = \frac{|U|}{m}\ \forall 0 \leq i \leq m-1$,
   d.h. die Elemente aus U werden gleichmäßig auf T verteilt
3. Für Folge von n Operationen gilt:
   Die Wahrscheinlichkeit, daß j-tes Element in der Folge ein festes $x \in U$ ist, ist $\frac{1}{N}$
   $\Rightarrow$ Operationen sind unabhängig und gleichverteilt.

$\Rightarrow$ Sei x_k, $1 \leq k \leq n$ Argument der k-ten Operation. Dann prob$(h(x_k) = i) = \frac{1}{m}\ \forall 0 \leq i \leq m-1$. D.h. mit Folgenelementen sind auch Funktionswerte gleichverteilt.

Definition 2.1:

$$\delta_h(x,y) = \begin{cases} 1 & \text{falls} \quad x \neq y \wedge h(x) = h(y) \\ 0 & \text{falls} \quad \text{sonst} \end{cases}$$

$\delta_h(x,S) = \sum_{y \in S} \delta_h(x,y) =$ Anzahl der Elemente in Liste T[h(x)] verschieden von x

Kosten einer Operation: XYZ$(x) = 1 + \delta_h(x,S)$.

Satz 2.1:

Mittlere Kosten von Operationen XYZ(x) sind

$$1 + \beta = 1 + \frac{n}{m}$$

(β = Belegungsfaktor)

Beweis:

Sei $h(x) = i$ und sei $p_{i,k}$ die Wahrscheinlichkeit, daß Liste i genau k Elemente enthält. Dann ist

$$p_{i,k} = \binom{n}{k} \left(\frac{1}{m}\right)^k \left(1 - \frac{1}{m}\right)^{n-k}$$

Mittlere Kosten von XYZ(x) sind

$$\begin{aligned}
\sum_{k \geq 0} p_{i,k}(1+k) &= \underbrace{\sum_{k \geq 0} p_{i,k}}_{1} + \sum_{k \geq 0} k \cdot \binom{n}{k} \left(\frac{1}{m}\right)^k \left(1 - \frac{1}{m}\right)^{n-k} \\
&= 1 + \sum_{k \geq 0} n \cdot \binom{n-1}{k-1} \left(\frac{1}{m}\right)^k \left(1 - \frac{1}{m}\right)^{n-k} \\
&= 1 + \frac{n}{m} \sum_{k \geq 1} \binom{n-1}{k-1} \left(\frac{1}{m}\right)^{k-1} \left(1 - \frac{1}{m}\right)^{n-1-(k-1)}, \\
&\quad \text{da} (a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i\, b^{n-i} \\
&= 1 + \frac{n}{m} = 1 + \beta
\end{aligned}$$

Problem:
Wenn β groß $\rightarrow$ lange Zugriffszeit; bei β klein $\rightarrow$ Platzverschwendung.

Beispiel:
$\beta \leq 1 \Rightarrow$ Laufzeit im Mittel $O(1)$
$\beta \geq \frac{1}{4} \Rightarrow$ Platz für Hashtafel + Listen $O(n+m) = O(n+\frac{n}{\beta}) = O(n)$

Idee:
Mit Einfügen/Streichen kann β schnell zu groß oder zu klein werden $\Rightarrow$ Rehashing: Folge von Hashtafeln $T_0, T_1, T_2, \ldots$ der Größe $m, 2m, 4m, \ldots$ In T_i der Größe $2^i \cdot m$ sei $\beta = 1$ erreicht
$\Rightarrow$ Umspeichern nach T_{i+1} für alle $2^i m$ Elemente; bei Belegungsfaktor $\frac{1}{4}$ Umspeichern auf T_{i-1}.

Amortisierte Analyse zeigt: Mittlere Laufzeit $O(1)$, aktuelle Größe der Hashtafel $O(|S|)$.

2.2 Hashing mit offener Adressierung

Jedes $x \in U$ definiert eine Folge $h(x,0), h(x,1), \ldots$ von Tafelpositionen. Gehe jeweils Folge durch, bis x oder freie Position gefunden.

Beispiel 2.2:

$h(x,i) = [h_1(x) + i \cdot h_2(x)] \bmod m$
$m = 7$, $h_1(x) = x \bmod 7$, $h_2 = 1 + (x \bmod 4)$
$S = \{3, 17, 6, 9, 15, 13, 10\}$ Einfügen in dieser Reihenfolge:

0	13
1	15
2	9
3	3
4	10
5	17
6	6

$h(3,i) = 3 + i \cdot 4 = 3$ (mit $i = 0$)
$h(17,i) = 3 + i \cdot 2 = 5$ (mit $i = 1$)
$h(6,i) = 6 + i \cdot 3 = 6$ (mit $i = 0$)
$h(9,i) = 2 + i \cdot 2 = 2$ (mit $i = 0$)
$h(15,i) = 1 + i \cdot 4 = 1$ (mit $i = 0$)
$h(13,i) = 6 + i \cdot 2 = 0$ (mit $i = 4$)
$h(10,i) = 3 + i \cdot 3 = 4$ (mit $i = 5$)

Satz 2.2:

Sei S, $|S| = n$ abgespeichert in Tafel der Größe $m, m > n$. Die mittlere Anzahl der Proben $\bar{p}_{n,m}$ beim Einfügen von x ist

$$\frac{m+1}{m+1-n} \approx \frac{1}{1-\beta}$$

$\beta = \dfrac{n}{m}$ Belegungsfaktor

Beweis:

Es gelten die drei Annahmen aus 2.1
Sei $q_j(n,m)$=Wahrscheinlichkeit, daß $T[h(x,0)], T[h(x,1)], \ldots, T[h(x,j)]$ besetzt.

Idealisierte Annahme: $h(x,0),\ldots,h(x,m-1)$ ist eine zufällige Permutation von $0,\ldots,m-1$.

$$q_j(n,m) = \frac{n}{m}\cdot\frac{n-1}{m-1}\cdot\ldots\cdot\frac{n-j}{m-j}$$

q_j bedeutet: mindestens $j+1$ Felder besetzt; q_j =prob{$\geq j+1$ Fehlversuche}.
$q_{-1}(n,m)$ =prob{≥ 0 Fehlversuche} $= 1$

$$\begin{aligned}
\bar{p}_{n,m} &= \sum_{j\geq -1}^{n}(q_j(n,m)-q_{j+1}(n,m))(j+2)\\
&= \sum_{j\geq -1} q_j(n,m) - (n+2)\cdot\underbrace{q_{n+1}(n,m)}_{=0}\\
&= q_{-1}(n,m) + \sum_{j\geq 0} q_j(n,m) \text{ mit } q_j(n,m) = \frac{n}{m}\cdot q_{j-1}(n-1,m-1) \quad \forall j\geq 0\\
&= 1+\frac{n}{m}\sum_{j\geq 0} q_{j-1}(n-1,m-1)\\
&= 1+\frac{n}{m}\sum_{j\geq -1} q_j(n-1,m-1) = 1+\frac{n}{m}\cdot\bar{p}_{n-1,m-1}
\end{aligned}$$

Weiter mit Induktion über n:
Behauptung: $\bar{p}_{n,m} = \dfrac{m+1}{m+1-n}$

$n=0$:
$\bar{p}_{0,m} = 1 = \dfrac{m+1}{m+1-0}$

$n>0$:

$$\begin{aligned}
\bar{p}_{n,m} &= 1+\frac{n}{m}\cdot\bar{p}_{n-1,m-1}\\
&= 1+\frac{n}{m}\cdot\frac{(m-1)+1}{(m-1)+1-(n-1)}\\
&= 1+\frac{n}{m}\cdot\frac{m}{m+1-n}\\
&= 1+\frac{n}{m+1-n}\\
&= \frac{m+1-n+n}{m+1-n} = \frac{m+1}{m+1-n} = \frac{1}{\frac{m+1-n}{m+1}}\\
&= \frac{1}{1-\frac{n}{m+1}} \approx \frac{1}{1-\beta}
\end{aligned}$$

β	0.5	0.8	0.9	0.95	0.99	0.995	
mittlere # beim Einf.	2	5	10	20	100	100	$\dfrac{1}{1-\beta}$
mittlere # beim Zugr.	1.38	2.01	2.55	3.15	4.65	5.32	$\dfrac{1}{\beta}\cdot ln\left(\dfrac{1}{1-\beta}\right)$

Tabelle 1: Ein paar konkrete Werte zur Veranschaulichung

Satz 2.3:
Sei $S, |S| = n$ in einer Tafel der Größe m abgespeichert. Die mittlere Anzahl der Proben bei Zugriff(x) ist

$$\frac{m+1}{n}(H_{m+1} - H_{m-n+1}) \approx \frac{1}{\beta} \ln\left(\frac{m+1}{m-n+1}\right) \approx \frac{1}{\beta} \ln\left(\frac{1}{1-\beta}\right)$$

Beweis:
Sei $S = \{x_1, \ldots, x_n\}$ in dieser Reihenfolge abgespeichert. Mittlere Anzahl der Proben:
$= \frac{1}{n} \sum_{i=1}^{n}$ mittlere Anzahl der Proben bei Zugriff(x_i)
$= \frac{1}{n} \sum_{i=1}^{n}$ mittlere Anzahl der Proben bei Einfüge(x_i)
(denn beim Zugriff(x_i) durchlaufen wir gleiche Folgen von Tafelpositionen wie bei Einfüge(x_i))

$$\begin{aligned}
&= \frac{1}{n} \sum_{i=1}^{n} \bar{p}_{i-1,m} \\
&= \frac{1}{n} \sum_{i=0}^{n-1} \frac{m+1}{m+1-i} \\
&= \frac{m+1}{n} \sum_{j=m-n+2}^{m+1} \frac{1}{j} \\
&= \frac{m+1}{n} \left(\sum_{j=1}^{m+1} \frac{1}{j} - \sum_{j=1}^{m-n+1} \frac{1}{j} \right) \\
&= \frac{m+1}{n}(H_{m+1} - H_{m-n+1})
\end{aligned}$$

Mit $H_m = \sum_{j=1}^{m} \frac{1}{j} \approx \ln m$ folgt

$$= \frac{1}{\beta} \ln\left(\frac{1}{1-\beta}\right)$$

2.3 Universelles Hashing

„Gleichverteilung der Operationen“ war kritisch, da benutzerabhängig.

- Wir arbeiten mit Klasse H von Hashfunktionen statt mit einer.
- Wir wählen jeweils h zufällig aus H.
- Wir mitteln nun über die Möglichkeiten zur Wahl von h.

Ziel: Für je zwei Punkte $x, y \in U, x \neq y$ soll nur ein Bruchteil der $h \in H$ zu Kollisionen führen.

Definition 2.2:
Sei $c \in \mathbb{R}$, $m, N \in \mathbb{N}$. Eine Klasse $H \subseteq \{h | h : [0..N-1] \to [0, \ldots, m-1]\}$ ist *c-universell*, wenn $\forall x, y \in [0..N-1]$, $x \neq y$:

$$|\{h | h \in H \text{ und } h(x) = h(y)\}| \leq \frac{c \cdot |H|}{m}$$

zur Definition: Für jedes Paar x, y führt nur ein Bruchteil $\frac{c}{m}$ der Funktionen in H zur Kollision. Hier ist eine fast 1-universelle Klasse H_1:

Satz 2.4:

Seien $m, N \in \mathbb{N}$, N sei eine Primzahl, H mit
$H = \{h_{a,b} | h_{a,b}(x) = [(ax + b) \bmod N] \bmod m,\ a, b \in [0..N-1]\}$
ist c-universell mit

$$c = \left(\frac{\left\lceil \frac{N}{m} \right\rceil}{\frac{N}{m}} \right)^2$$

Beweis:

$|H| = N^2$. Jedes a, b bestimmt eine Hashfunktion. Seien $x, y \in [0..N-1]$, $x \neq y$.
Zeige: $|\{(a,b) | h_{a,b}(x) = h_{a,b}(y)\}| \leq \frac{c \cdot N^2}{m}$
Wenn $h_{a,b}(x) = h_{a,b}(y)$ für $x \neq y$, dann $\exists q \in [0, \ldots, m-1]$:

$$\begin{aligned} (a \cdot x + b) \bmod N &= q + r \cdot m \\ (a \cdot y + b) \bmod N &= q + s \cdot m \end{aligned}$$

Da $Z|_N$ ein Körper (N Primzahl), gibt es in a, b genau eine Lösung für Gleichungen für jede Auswahl von q, r, s. Damit:

$$\begin{aligned} |\{(a,b) | h_{a,b}(x) = h_{a,b}(y)\}| &\leq m \left(\left\lceil \frac{N}{m} \right\rceil \right)^2 \\ &= \left(\left\lceil \frac{N}{m} \right\rceil \right)^2 \cdot m \cdot \frac{N^2}{m^2} \cdot \frac{m^2}{N^2} \\ &= \left(\frac{\left\lceil \frac{N}{m} \right\rceil}{\frac{N}{m}} \right)^2 \cdot \frac{N^2}{m} = c \cdot \frac{N^2}{m} \end{aligned}$$

Analyse des mittleren Verhaltens von universellem Hashing
Annahmen:

1. Hashfunktion h wird zufällig aus c-universeller Klasse H mit Wahrscheinlichkeit $\frac{1}{|H|}$ gezogen
2. Hashing mit verketteten Listen

Satz 2.5:

Sei $c \in \mathbb{R}$ und H eine c-universelle Klasse, $S \subseteq U$

a) Sei $|S| = n$ und $x \in [0..N-1]$. Es gilt:

$$\sum_{h \in H} \frac{1 + \delta_h(x, S)}{|H|} \leq \begin{cases} 1 + c \cdot \frac{n}{m} & \text{falls} \quad x \notin S \\ 1 + c \cdot \frac{n-1}{m} & \text{falls} \quad x \in S \end{cases}$$

b) Mittlere Kosten einer Operation sind $1 + c \cdot \beta$

c) Folge von n Operationen kosten im Mittel $O((1 + c \cdot \beta) \cdot n)$ Zeit

c ist ein Maß dafür, wie gleichmäßig eine beliebige Teilmenge S von fast allen $h \in H$ auf Tafel verteilt wird.

Beweis:

$$\begin{aligned}
\text{a)} \quad \sum_{h\in H} 1+\delta_h(x,S) &= |H| + \sum_{h\in H}\sum_{y\in S}\delta_h(x,y) \\
&= |H| + \sum_{y\in S}\sum_{h\in H}\delta_h(x,y) \\
&\leq |H| + \sum_{y\in S\setminus\{x\}} \frac{c\cdot|H|}{m}, \text{ da } H \text{ c} - \text{universell} \\
&= \begin{cases} |H|(1+c\cdot\frac{n}{m}) & \text{falls} \quad x\notin S \\ |H|(1+c\cdot\frac{n-1}{m}) & \text{falls} \quad x\in S \end{cases}
\end{aligned}$$

Beachte: vorher wurde über S gemittelt. Dieser Satz gilt $\forall S$.

Jetzt Aussage, mit welcher Wahrscheinlichkeit „schlechte“ Hashfunktion vorkommen.

Satz 2.6:

Mit obigen Voraussetzungen sei $\mu = \frac{1}{|H|}\sum_{h\in H}\delta_h(x,S)$.

Dann gilt: $\forall t \geq 1:\ \text{prob}(\delta_h(x,S) \geq t\cdot\mu) \leq \frac{1}{t}$, d.h. $|\{h|\delta_h(x,S)\geq t\cdot\mu\}| \leq \frac{|H|}{t}$

Beweis:

Sei $H' = \{h|\delta_h(x,S)\geq t\cdot\mu\}$. Dann ist

$$\begin{aligned}
\mu = \frac{1}{|H|}\sum_{h\in H}\delta_h(x,S) \geq \frac{1}{|H|}\sum_{h\in H'}\delta_h(x,S) &\geq \frac{1}{|H|}\sum_{h\in H'} t\cdot\mu \\
&= \frac{|H'|}{|H|}t\cdot\mu
\end{aligned}$$

$\Rightarrow \frac{|H|}{t} \geq |H'|$

2.4 Perfektes Hashing

Wir wollen eine Hashfunktion h für S wählen, so daß keine Kollisionen vorkommen, d.h. $h|_S$ injektiv. Wir nehmen an, S sei bekannt. Zweistufiges Verfahren:
Stufe 1: Hashing mit Verkettung, aber *kurze* Listen
Stufe 2: Jede Liste wird mit eigener injektiver Hashfunktion behandelt

Zunächst: Wahl einer injektiven Hashfunktion.
Sei $U = \{0,\ldots,N\}$, sei $h_k : \{0,\ldots,N-1\} \to \{0,\ldots,m-1\}$ Hashfunktion für $k \in \{1,\ldots,N-1\}$ mit $h_k = ((kx) \bmod N) \bmod m$.

Sei $S \subseteq U$ bekannt. Für welches k ist h_k injektiv? Maß für „Injektivität“:
Sei $b_{i,k} = |\{x\in S|h_k(x)=i\}|$ für $1\leq k\leq N-1$, $0\leq i\leq m-1$, die Anzahl der $x\in S$ mit gleichem Hashwert i für h_k,
$b_{i,k}(b_{i,k}-1) = |\{(x,y)\in S^2|x\neq y,\ h_k(x)=h_k(y)=i\}|$,
die Anzahl der Paare, die mit Hashwert i die Injektivität verletzen.

$B_k = \sum_{i=0}^{m-1} b_{i,k}(b_{i,k}-1)$ ist die Anzahl der Paare in S^2, die die Injektivität verletzen.

Lemma 2.1:

1. Mit obigen Voraussetzungen gilt: $B_k < 2 \Leftrightarrow h_k|_S$ injektiv.
2. Ist $b_{i,k} = \frac{n}{m}\ \forall 0 \le i \le m-1$, dann ist

$$B_k = \sum_{i=0}^{m-1} \frac{n}{m}\left(\frac{n}{m}-1\right) = n \cdot \left(\frac{n}{m}-1\right)$$

Beweis (von 1.):
„$\Rightarrow$“: $B_k < 2 \Leftrightarrow \sum b_{i,k}(b_{i,k}-1) \in \{0,1\} \Rightarrow \forall i : b_{i,k} \in \{0,1\} \Rightarrow h_k|_S$ injektiv
„$\Leftarrow$“: $h_k|_S$ injektiv $\Rightarrow \forall i : b_{i,k} \in \{0,1\} \Rightarrow B_k = 0$ ❏

Lemma 2.2:
Mit obigen Voraussetzungen und N Primzahl gilt:

$$\sum_{k=1}^{N-1} \underbrace{\sum_{i=0}^{m-1} b_{i,k}(b_{i,k}-1)}_{B_k \approx 2 \cdot \#\text{Kollisionen für } h_k} \le 2 \cdot \frac{n \cdot (n-1)}{m} \cdot (N-1)$$

$\Rightarrow$ mittlere Anzahl der Kollisionen $\le \frac{n \cdot (n-1)}{m}$

Beweis:
Mit $m > n \cdot (n-1)$ ist die mittlere Anzahl der Kollisionen < 1
$\Rightarrow \exists h_k$: Anzahl der Kollisionen< 1. h_k ist injektiv auf S.

$$\begin{aligned}
&\sum_{k=1}^{N-1}\sum_{i=0}^{m-1} b_{i,k}(b_{i,k}-1) \\
&\quad = \sum_{k=1}^{N-1}\sum_{i=0}^{m-1} |\{(x,y)|x \neq y,\ h_k(x) = h_k(y) = i\}| \\
&\quad = \sum_{\substack{(x,y)\in S^2 \\ x\neq y}} |\{k|h_k(x) = h_k(y)\}|.
\end{aligned}$$

Seien $x, y \in S$ fest. Wieviele Funktionen h_k mit $h_k(x) = h_k(y)$ gibt es?

$$\begin{aligned}
h_k(x) = h_k(y) &\Leftrightarrow ((kx) \bmod N) \bmod m = ((ky) \bmod N) \bmod m \\
&\Leftrightarrow \underbrace{((kx) \bmod N) - ((ky) \bmod N))}_{g(k)} \bmod m = 0
\end{aligned}$$

Es gilt:
$-N+1 \le g(k) \le N-1$ und
$g(k) = i \cdot m$ mit $i \in \left[\left\lfloor \frac{-N+1}{m} \right\rfloor, \left\lfloor \frac{N-1}{m} \right\rfloor\right]$

Behauptung:
Für jedes solches i gibt es höchstens ein k mit $g(k) = i \cdot m$.

$$\Rightarrow |\{k|h_k(x) = h_k(y)\}| \le 2 \cdot \left\lfloor \frac{N-1}{m} \right\rfloor$$

$\Rightarrow$ Gesamtsumme:

$$\sum_{k=1}^{N-1}\sum_{i=0}^{m-1} b_{i,k}(b_{i,k}-1) \leq 2 \cdot \frac{N-1}{m} \cdot n \cdot (n-1)$$

Beweis der Behauptung:
Seien $k_1, k_2 \in [1..N-1]$ mit $g(k_1) = g(k_2) = i \cdot m$
$g(k_1) = g(k_2) \Leftrightarrow ((k_1 x) \bmod N) - ((k_1 y) \bmod N) = ((k_2 x) \bmod N) - ((k_2 y) \bmod N)$
$\Leftrightarrow ((k_1 x - k_1 y) - (k_2 x - k_1 y)) \bmod N = 0$
$\Leftrightarrow (k_1 - k_2)(x - y) \bmod N = 0$
Da $Z|_N$ ein Körper ist, folgt $(k_1 - k_2) \bmod N = 0$ oder $(x - y) \bmod N = 0$
Mit $x \neq y$ und $k_1, k_2 \in [1..N-1]$ folgt $k_1 = k_2$. ❏

Korollar 2.1:

Mit obigen Voraussetzungen gilt:

1. $\exists k \in [1..N-1] : \; B_k \leq 2 \cdot \frac{n \cdot (n-1)}{m}$
2. Mindestens $\frac{N-1}{2}$ viele h_k haben B_k mit $B_k \leq \frac{4n \cdot (n-1)}{m}$

Beweis:

1. $\frac{2n \cdot (n-1)}{m}$ ist Mittelwert, ein h_k muß kleineren B_k haben.
2. Sei $A = \{k | B_k > \frac{4n \cdot (n-1)}{m}\}$. Annahme $|A| > \frac{N-1}{2}$
   $\Rightarrow \sum_{k=1}^{N-1} B_k > \sum_{k \in A} B_k > \frac{N-1}{2} \cdot \frac{4n \cdot (n-1)}{m} = \frac{2n \cdot (n-1)}{m} \cdot (N-1)$ Widerspruch zu Lemma.

❏

Korollar 2.2:

Mit obigen Voraussetzungen gilt:

1. Ein $k \in [1..N-1]$ mit $B_k \leq \frac{2 \cdot n(n-1)}{m}$ kann in Zeit $O(m + N \cdot n)$ berechnet werden.
2. Ein $k \in [1..N-1]$ mit $B_k \leq \frac{4 \cdot n(n-1)}{m}$ findet man in Zeit $O(m+n)$ (randomisiert).
3. Sei $m = n \cdot (n-1) + 1$. Dann gibt es k mit $h_k|_S$ injektiv. k kann in Zeit $O(n^2 + N \cdot n)$ gefunden werden.
4. Sei $m = 2n \cdot (n-1) + 1$. Dann sind mindestens die Hälfte der h_k injektiv auf S. Laufzeit $O(n^2)$ (randomisiert).
5. Sei $m = n$. Ein k mit $B_k \leq 2 \cdot (n-1)$ kann in Zeit $O(n \cdot N)$ gefunden werden.
6. Sei $m = n$. Ein k mit $B_k \leq 4 \cdot (n-1)$ findet man randomisiert in Zeit $O(n)$.

Beweis:

1. Teste alle k durch. Es gibt $(N-1)$ verschiedene k's und n verschiedene $x \in S$. x wird ins Fach $h_k(x)$ abgelegt. Berechne B_k's, d.h. schaue nach, ob Fächer zu voll sind.

2. prob$(B_k \leq \frac{4 \cdot n(n-1)}{m} \geq \frac{1}{2}) \Rightarrow$ randomisiert trifft man k in 2 Versuchen, 1 Test dauert $O(m+n)$.

3. Es gibt $B_k \leq \frac{2 \cdot n(n-1)}{m} = \frac{2 \cdot n(n-1)}{n(n-1)+1} < 2 \Rightarrow h_k$ injektiv auf S. Mit Korollar 2.2.1. gilt Laufzeit

Beweis von 4.,5. und 6. durch Einsetzen ❏

Realisierung:
Aus Korollar 2.2.6. folgt: Mit $m = n$ gibt es ein h_k, so daß die Länge der Listen $O(\sqrt{n})$ sind, denn $B_k \leq O(n)$. Wenden auf jede Liste nochmals Hashing an.

Genauer: Sei $S \subseteq [0..N-1]$, $|S| = n = m$

1. Wähle k mit $B_k \leq 4(n-1) \leq 4n$
   Sei $h_k(x) = ((kx) \bmod N) \bmod n$.

2. Sei $w_i = \{x \in S | h_k(x) = i\}$. Sei $b_i = |w_i|$ und $m_i \geq 2 \cdot b_i(b_i - 1) + 1$. Wähle k_i mit $h_{k_i}(x) = ((k_i x) \bmod N) \bmod m_i$ ist injektiv auf w_i.

3. Sei $s_i = \sum_{j<i} m_j$. Speichere $x \in S$ in $T[s_i + j]$ wenn
   $i = ((kx) \bmod N) \bmod n$
   $j = ((k_i x) \bmod N) \bmod m_i$

Platzbedarf:

$$m = \sum_{i=0}^{n-1} m_i = \sum_{i=0}^{n-1} 2b_i(b_i - 1) + 1 = n + 2B_k \leq n + 8n \leq O(n)$$

Laufzeit:

1. $O(n)$

2. $O(n)$ für w_i, b_i, m_i, s_i

3. Für jedes k_i : $O(m_i)$

Gesamt: $O(n+m) = O(n)$

Satz 2.7:

Mit obigen Voraussetzungen kann für S eine perfekte Hashtafel der Größe $O(n)$ und eine Hashfunktion mit $O(1)$ Zugriffszeit deterministisch in Zeit $O(n \cdot N)$ und randomisiert in Zeit $O(n)$ aufgebaut werden.

Bevorzuge **randomisierte** Version

Jetzt: Online-Hashing
Starte mit leerer Hashtafel, $S = \{x_1, \ldots, x_n\}$. Füge nacheinander $x_1, x_2, \ldots$ ein. Wir kennen im i-ten Schritt nur $x_1, x_2, \ldots, x_i$.

Gibt es zu viele Kollisionen auf Stufe 1 oder eine nicht injektive Hashfunktion auf Stufe 2 $\rightarrow$ Umbau

```
procedure Tafelaufbau
    wähle k ∈ [1..N − 1] zuällig;
    for i = 0 to n − 1 do
        w_i ← ∅, b_i ← 0, m_i ← 1, s_i ← i;
        wähle k_i ∈ [0..n − 1] zufällig;
    od
    m ← n; B ← 0;
    for l = 1 to n do Insert(x_l) od;

procedure Insert(x)
    i ← ((kx) mod N) mod n; j ← ((k_i y) mod N) mod m_i;
    w_i ← w_i ∪ {x}; B ← B + 2b_i; b_i ← b_i + 1;
    if B > 4 · (n − 1) then
       Tafelaufbau
    else
       if T[s_i + j] frei then
          speichere x in T[s_i + j]
       else
          (*h_{k_i}|_{w_i} nicht injektiv *)
          if m_i < 2 · b_i(b_i − 1) + 1 then
             (* vergrößere m_i*)
             m_i ← 4 · b_i(b_i − 1) + 1;
             s_i ← m; m ← m + m_i;
             (*T[s_i..m − 1] reserviert für w_i*)
          fi;
Marke:    versuche_wieder
          wähle k_i ∈ [0..N − 1] zufällig;
          lösche T[s_i, ..., s_i + m_i − 1]
          for all x ∈ w_i do
             j ← ((k_i x) mod N) mod m_i;
             if T[s_i + j] frei then
                speichere x dort ab;
             else
                goto versuche_wieder
             fi;
          od;
       fi;
    fi;
```

Analyse:
evtl. wiederholtes *versuche_wieder* bzw. Tafelaufbau

versuche_wieder:
Test, ob gewähltes k_i gut ist in $O(b_i)$
m_i wird groß genug gehalten, so daß k_i mit Wahrscheinlichkeit $\leq \frac{1}{2}$ injektiv für w_i, d.h. im Mittel $2\times$ *versuche_wieder* für i, also $O(b_i)$ Zeit, um passendes k_i zu finden.
Sei b_i^f das endgültige b_i nach allen Einfügungen.

$$\text{Gesamtzeit} \leq O(\sum_{j=1}^{b_i^f} j) = O((b_i^f)^2)$$

Tafelaufbau passiert, wenn $B > 4 \cdot (n-1)$.

Für zufälliges k gilt nach Korollar

$$B^f = \sum_{i=0}^{n-1} b_i^f (b_i^f - 1) \leq 4 \cdot (n-1)$$

mit Wahrscheinlichkeit$\geq \frac{1}{2}$. Da das aktuelle $B \leq B^f$, brauchen wir ≤ 2 Versuche für k im Mittel

$$O(n) + O(\sum_{i=0}^{n-1} (b_i^f)^2) = O(n) + O(b^f)$$

Platz:
m_i wächst: $m_{i,1}, m_{i,2}, \ldots, m_i^f$.
Es gilt: $m_{i,(p+1)} \geq 2 \cdot m_{i,p} + 1$ für $p \geq 1$.

$$\Rightarrow \sum_{p+1} m_{i,p} \leq \sum_{p \geq 0} \frac{m_i^f}{2^p} = m_i^f \cdot \sum_{p \geq 0} \frac{1}{2^p} \leq 2 \cdot m_i^f$$

Gesamtplatzbedarf

$$= \sum_{i=0}^{n-1} 2 \cdot m_i^f \leq \sum_{i=0}^{n-1} 8 b_i^f (b_i^f - 1) + 2 \leq 8B^f + 2 = 8 \cdot 4(n-1) + 2 \leq 32n$$

Satz 2.8:

Sei N Primzahl, $S \subseteq U$, $|S| = n$. Eine perfekte Hashtafel der Größe $O(n)$ kann online in $O(n)$ erwarteter Zeit berechnet werden. Zugriffszeit: $O(1)$.

3 Höhenbalancierte Bäume, Union Find

3.1 (2,4)-Bäume

Definition 3.1:

Seien $a, b \in \mathbb{N}$, mit $a \geq 2$, $b \geq 2a - 1$.
T heißt (a, b)-Baum, falls

a) alle Blätter haben die gleiche Tiefe (Abstand zur Wurzel)

b) alle inneren Knoten v haben $\leq b$ Kinder

c) alle inneren Knoten v (außer Wurzel) haben $\geq a$ Kinder

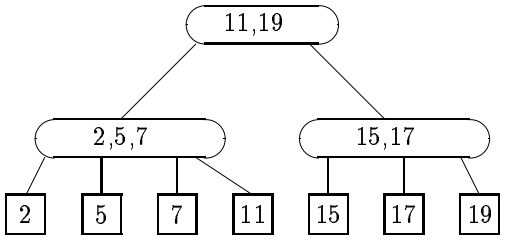
d) Wurzel r hat ≥ 2 Kinder

Jetzt: $(2, 4)$-Bäume

Abspeicherung:
Sei $S = \{x_1 < x_2 < \ldots < x_n\}$

1. Speichere S von links nach rechts in den Blättern.
2. In den inneren Knoten v mit d Kindern, speichere $d - 1$ Elemente $k_1(v), \ldots, k_{d-1}(v)$, wobei $k_i(v)$ Inhalt des rechtesten Blattes im i-ten Baum.
3. Speichere zusätzlich maximales Element von S in der Wurzel.

Beispiel 3.1: $S = \{2, 5, 7, 11, 15, 17, 19\}$



Bemerkung:
Die Tiefe t eines $(2, 4)$-Baumes mit n Blättern liegt zwischen $\frac{1}{2} \log n$ und $\log n$.

Beweis: $2^t \leq n \leq 4^t$

$\Rightarrow \log_4 n \leq t \leq \log_2 n = \frac{1}{2} \log_2 n \leq t \leq \log_2 n$ ❏

Operationen:

Zugriff(k): liefert Blatt k' mit $k' = min\{x \in S | k \leq x\}$

```
if k > max(S) then kein Resultat
   else v ←wurzel;
        while v kein Blatt do
              wähle i so, daß k_{i-1}(v) < k ≤ k_i(v);
              v ←i-tes Kind von v;
        od;
        return(v);
fi;
```

Laufzeit: O(Tiefe des Baumes) $= O(\log n)$

Einfügen(k): gegeben

- Schlüssel k
- Blatt v_i mit key($v_{i-1} < k <$key(v_i)
- Pfad von v_i zur Wurzel

Füge neues Blatt ein:

```
while w hat 5 Kinder do
      Spalte(w);
      w ←elter(w);
od;
```

Laufzeit: $O(1 + \#\text{Spaltungen})$

Lemma 3.1:
In einem $(2,4)$-Baum sind die armortisierten Kosten von Einfügen bzw. Streichen $O(1)$

Beweis:
Wähle Potential wie folgt:

$$\begin{aligned}\text{pot}(T) = {} & 2 \cdot \#\text{ Knoten vom Grad 1} \\ & + 1 \cdot \#\text{ Knoten vom Grad 2} \\ & + 0 \cdot \#\text{ Knoten vom Grad 3} \\ & + 2 \cdot \#\text{ Knoten vom Grad 4} \\ & + 4 \cdot \#\text{ Knoten vom Grad 5}\end{aligned}$$

Folgende Invariante gelte:

a) $\text{pot}(T) \geq 0$

b) bei Anwendung von Stehle/Verschmelze/Spalte auf Knoten w, haben alle anderen Knoten 2,3 oder 4 Kinder

c) Vor Einfügen/Streichen haben alle Knoten w 2,3 oder 4 Kinder

tatsächliche Kosten für die Operationen Stehle/Verschmelze/Spalte: $O(1)$

Behauptung: Die Operationen Spalten/Verschmelzen senken das Potential, Stehlen erhöht es nicht.

Beweis:

a) Stehlen:
Knoten w trägt 2 Einheiten bei, w' trägt p bei. Nach dem Stehlen trägt w 1 Einheit bei, $w'' \leq p + 1$ Einheiten $\Rightarrow$ Potential steigt nicht

b) Spalten:
w trägt 4 Einheiten bei, danach tragen w, w' 1 und 0 bei. Der Elternknoten trägt vorher p Einheiten und nachher $\leq p + 2$ Einheiten bei.

c) Verschmelzen:
w trägt 2 Einheiten bei, w' trägt 1 bei, Elternknoten w'' trägt p bei. Danach: w trägt 0 bei, w'' trägt $\leq p + 1$ bei

Amortisierte Laufzeit für Einfügen: tatsächliche Kosten + Potentialerhöhung

Schritt 1: Tatsächliche Kosten + Potentialerhöhung ≤ 3
Schritt 2: Folge der Spaltungen:
Tatsächliche Kosten f, Potentialerhöhung $\leq -f$
amortisierte Kosten ≤ 3

Ähnlich fürs Streichen.

Anwendung: Sortieren vorsortierter Folgen

Sortieren durch Einfügen, startdend mit leerem Baum.
n Zugriffe kosten $O(n \log n)$, $n\times$ Einfügen $O(n)$. Wir machen nun Lokalisieren (Zugriff) billiger. Das geht, wenn die Folgen vorsortiert sind. Sei $x_1, \ldots, x_n$ eine Folge reeller Zahlen. $F = |\{(i,j) | i < j \wedge x_i > x_j\}|$ heißt Anzahl der Inversionen mit $0 \leq F \leq \frac{n^2}{2}$

Satz 3.1:

$x_1, \ldots, x_n$ können sortiert werden in Zeit

$$O(n \log(\frac{F}{n} + 1))$$

Bemerkung:

Mit $F \approx \frac{n^2}{2} \Rightarrow T = O(n \log n)$
Mit $F \approx n \Rightarrow T = O(n)$
Mit $F \approx n \log n \Rightarrow T(n \log \log n)$
Mit $F \approx n^{\frac{3}{2}} \Rightarrow T = O(n \log(n)^{\frac{1}{2}}) = O(n \cdot \frac{1}{2} \cdot \log n) = O(n \log n)$

Beweis:

$f_i = |\{j | i < j \wedge x_i > x_j\}|$. Dann $F = \sum_{i=1}^{n} f_i$
Idee: Füge x_i's in umgekehrter Reihenfolge in den Baum, startend mit x_n.
Bei Add(x_i):
starte beim linken Blatt, laufe hoch, drehe an richtiger Stelle um, laufe runter, füge ein.
Intuition: Ist x_i klein im Vergleich zu $x_{i+1}, \ldots, x_n$, so läuft man nicht sehr hoch.

$T = \{x_n\}$
`for` $i = n-1$ `downto` 1 `do` Add(x_i) `od`;
mit Add(x_i): $v \leftarrow$ linkstes Blatt von T
`while` $x_i >$ key(elter(v)) `do`
$v \leftarrow$ elter(v);
`od`;

Danach kennen wir den Umkehrpunkt, laufen runter und können Einfüge(x_i) machen.

Behauptung: Amortisierte Kosten fürs Add(x_i) sind $O(1 + \log f_i)$

Einfüge(x_i) geht amortisiert in $O(1)$
Lokalisieren: amortisierte Kosten = tatsächlche Kosten (keine Potentialänderung)
tatsächliche Kosten = o(Höhe h des Umkehrpunktes)
x_i wird nach dem f_i-ten Blatt in den Baum eingefügt
v_0 ist Umkehrpunkt
v_1 ist linkestes Kind von v_0
v_2 ist linkestes Kind von v_1

Es gilt: Alle Blätter unterhalb von v_2 sind kleiner als x_i
v_2 hat Höhe $h-2$ und $\geq 2^{h-2}$ Blätter im Unterbaum
$\Rightarrow 2^{h-2} \leq f_i$
$\Rightarrow h-2 \leq \log f_i$
$h \leq 2 + \log f_i$ ❏

Gesamtkosten: $\sum_{i+1}^{n} O(1+\log f_i) = O(n + \sum_{i=1}^{n} \log f_i) = O(n + \log \prod_{i=1}^{n} f_i) \leq O(n + \log(\frac{F}{n})^n) = O(n + n\log(\frac{F}{n}))$

3.2 Union Find

Partitionen von $\{0,\ldots,N-1\} = U$ soll verwaltet werden unter **Operationen:**

1. Find(x): liefert Namen des Blocks, der x enthält
2. Union(A,B,C): vereinige Blöcke mit Namen A und B und gibt neuen Namen C ($C = A \cup B$)

Anwendungen:

- Konstruktion von minimal aufspannenden Bäumen
- Berechnung des kleinsten gemeinsamen Vorfahren in Bäumen
- (Fortran)

Starte mit Partitionen in Mengen der Größe 1. Menge(i) hat den Namen i.

1. Lösung:
Feld Ist_In$[0..N-1]$.
Ist_In$[i]$ = Name des Blocks, der i enthält
Find(x) = Ist_In$[x]$ geht in Zeit $O(1)$
Union(A,B,C): laufen durch Feld Ist_In und ändern alle Einträge A,B in C. Zeit $O(n)$

Verbesserung:
Für jeden Namen halte Liste der darin enthaltenen Elemente.
Union(A,B,C): durchlaufe Listen für A und B, ändere Einträge dort.
Zeit $O(|A|+|B|)$, konkatiniere Listen.

Beispiel 3.2:
$\{0\},\{1\},\ldots,\{N-1\}$

$$\left.\begin{array}{l}\text{Union}(0,1,0)\\ \text{Union}(0,2,0)\\ \vdots\\ \text{Union}(0,N-1,0)\end{array}\right\} \text{Laufzeit} : O(2+3+4+\cdots+N) = O(N^2)$$

Verbesserung:
Speichere für jeden Block seine Größe. Ändere bei Unions nur Einträge in kürzerer Liste. Laufzeit $O(min(|A|,|B|))$

Problem:
Menge sollte C heißen

Ausweg:
Vergieb *interne* und *externe* Namen (oben konstuierte Menge heißt intern B, extern C). Wir brauchen zwei Abbildungen:

1. MAPOUT: interner Name → externer Name
2. MAPIN: externer Name → interner Name

MAPIN, MAPOUT sind als Felder realisiert, Namen sind Zahlen zwischen 0 und $N-1$.

Realisierung:
SIZE[A] = Größe des Blocks mit externem Namen A.
Find(x) = MAPOUT[Ist_In[x]]
Union(A, B, C):

```
if SIZE[A] ≤ SIZE[B]
   then X←MAPIN[A]; Y←MAPIN[B]
   else Y←MAPIN[A]; X←MAPIN[B]
for all Elemente x in Liste X do Ist_In[x]←Y od;
```
konkatiniere Liste X und Y, speichere sie unter Listenkopf Y.
MAPOUT[Y]←C;
MAPIN[C]←Y;
SIZE[C]←SIZE[A]+SIZE[B];

Satz 3.2:

Eine Folge von $n-1$ Union() und m Find() hat Gesamtkosten $O(m + n \log n)$.

Beachte: Einzelne Unions können $O(n)$ kosten.

Beweis:

Lasse Element x 1 DM bezahlen, wenn Ist_In[x] geändert wird. Kosten der Unions sind proportional zur Anzahl der Änderungen in Ist_In.

Behauptung:
kein Element bezahlt mehr als $\log n$ DM und höchstens $2n$ Elemente bezahlen überhaupt.

Beobachtung:

1. Bezahlt x 1 DM, wechselt x zu einem Block mindestens doppelter Größe
2. Mit $n-1$ Unions können Blöcke der Größe höchstens n aufgebaut werden (Beweis durch Induktion)
3. Nur $\leq 2n$ Elemente können an Unions teilnehmen

aus 1. und 2. folgt Teil 1 der Behauptung, aus 3. Teil 2 ⇒ Satz. ❏

Alternative: Stelle Blöcke als Bäume dar. Die Wurzel trägt den Namen des Blocks. Dies wird realisiert durch das Feld Elter[0..N-1] mit

$$\text{Elter}(i) = \begin{cases} j & \text{falls } j \text{ Elter von } i \text{ im Baum} \\ nil & \text{sonst} \end{cases}$$

Name[i]=Name des Blocks, der durch den Baum mit Wurzel i dargestellt wird, falls Elter[i]=nil, undefiniert sonst.
Wurzel[A]=Wurzel des Blocks mit Namen A.

```
Find(x): while Elter(x) ≠nil do x ← Elter(x) od;
         drucke(Name[x]);
```

Laufzeit: O(Tiefe von x im Baum)

```
Union(A, B, ...): i ← Wurzel(A); j ← Wurzel(B);
                  Elter[i]← j; (* oder Elter[j] ← i; *)
```

Idee: Vermeide große Tiefe, in dem die Wurzel des größeren Baumes zur Wurzel des neuen Baumes wird.

Satz 3.3: Folge von $n-1$ *Unions()* und m *Finds()* kostet $O(n + m \log n)$.

Beweis:

Sei F der Wald, den man nach $n-1$ Unions erhält. Höhe(x)=Höhe von x im Baum Gewicht(x)=Anzahl der Nachfolger von x in F, inklusive x

Behauptung: Das Gewicht(x) $\geq 2^{Höhe(x)}$, insbesondere Höhe(x) $\leq \log n$

Beweis: Induktion über die Höhe:

Höhe(x) $= 0$: $\Rightarrow 2^{Höhe(x)} = 1 \leq$ Gewicht(x)
Höhe(x) > 0: Sei y ein Kind mit Höhe(y)=Höhe(x) -1
Als y Kind von x wurde, war Gewicht(x) >Gewicht(y)
Danach änderte sich Gewicht(y) nicht mehr.
Evtl. hat sich das Gewicht(x) noch erhöht. Zum Schluß gilt:
Gewicht(x) $\geq 2 \cdot$Gewicht(y) $\geq 2 \cdot 2^{Höhe(x)-1} = 2^{Höhe(x)}$

Mit $n-1$ Unions kann ein Baum höchstens n Knoten haben, d.h. $n \geq$ Gewicht(x) $\Rightarrow$ $\log n \geq$Höhe(x).

Also: Find(x) kostet $\leq O(\log n)$ Zeiteinheiten ❏

Idee: Pfadkomprimierung
Nachfolgende *Finds()* werden billiger

Jetzt: Union-Find mit *Weighted-Uion Rule + Pfadkomprimierung*

Definition 3.2: $A : I\!N_0 \times I\!N_0 \to I\!N_0$

1. A$(i, 0) = 0 \; \forall i$
2. A$(i, 1) = 2$
3. A$(0, x) = 2x \; \forall x$
4. A$(i, x) =$A$(i-1,$A$(i, x-1))$ für $i \geq 1, x \geq 2$

ist Variante der Ackermannfunktion

Fkt.:	x	$\to$						
i	A	0	1	2	3	4	...	x
$\downarrow$	0	0	2	4	6	8	...	$2x$
	1	0	2^1	2^2	2^3	2^4	...	2^x
	2	0	2	4	2^4	2^{16}		
	3	0	2					
	4	0	2					

Definition 3.3: $\alpha(m, n) := \min\{z | A(z, \lceil 4\frac{m}{n} \rceil) > \log n\}$
realistisch: $\alpha(m, n) \leq 3$ für $m \geq n$

Satz 3.4:

Gesamtkosten einer Folge von $n-1$ Union() und m Find() sind nach der neuen Startegie $O(m \cdot \alpha(m, n))$. Dabei sei $m \geq n$.

Bemerkung: Schranke kann nicht verbessert werden.

Beweis:
Zuerst alle Unions $\rightarrow$ Wald T. Interpretiere Finds als partielle Finds. Laufen-/Komprimieren einen Pfad entlang von v_0 zu v_k (v_k nicht unbedingt Wurzel).
jetzt: Union zuerst
wir zeigen nun: Schranke für $n-1$ Unions, gefolgt von m partiellen Finds. F sei eine Multimenge der Kanten, die in partiellen Finds benutzt werden. Zeigen: $F \leq O(m \cdot \alpha(m,n))$
teilen F in Gruppen, eingeteilt nach Höhen der Endpunkte der Kanten.

$$G(i,j) = \{x | A(i,j) \leq \text{Höhe}(x) < A(i,j+1) \text{ für } 0 \leq i \leq z, j \geq 0\}$$

z wird später bestimmt
F wird partitioniert: für $0 \leq k \leq z$

$$N_k = \{(x,y) \in F | k = \min\{i | \exists j : x,y \in G_{i,j}\}\}$$

Setze $N_{z+1} := F - \bigcup_{k=0}^{z} N_k$
$L_k = \{(x,y) \in N_k |$ auf dem partiellen Feld, das (x,y) für $0 \leq k \leq z+1$ enthält, ist (x,y) die letzte Kante in $N_k\}$ ❏

Lemma 3.2:

a) $|L_k| \leq m$ für $0 \leq k \leq z+1$

b) $|N_0 - L_0| < n$

c) $|N_k - L_k| \leq 6n$ für $1 \leq k \leq z$

d) $|N_{z+1} - L_{z+1}| < n \cdot a(z,n)$ mit $a(z,n) = \min\{i | A(z,i) > \log n\}$

$\Rightarrow$ Summieren

$$\begin{aligned} |F| &= \sum_{k=0}^{z+1} |L_k| + |N_0 - L_0| + \sum_{k=1}^{z} |N_k - L_k| + |N_{z+1} - L_{z+1} < \\ &\leq (z+2) \cdot m + n + z \cdot 3n + n \cdot a(z,n) \end{aligned}$$

wähle $z = \alpha(m,n) \Rightarrow \leq (\alpha(m,n)+2) \cdot m + n + 3 \cdot \alpha(m,n) \cdot n + 8m = O(n \cdot \alpha(m,n))$
wegen $a(z,n) = a(\alpha(m,n),n) = \min\{i | A(\alpha(m,n),i) > \log n)$
es gilt $\alpha(m,n) = \min\{z | A(z, \lceil \frac{4m}{n} \rceil) > \log n\}$
d.h.

$$a(\alpha(m,n),n) \leq \left\lceil \frac{4m}{n} \right\rceil \leq \frac{8m}{n}$$

d.h.

$$n \cdot a(z,n) \leq n \cdot \frac{8m}{n} = 8m$$

Beweis des Lemmas:

a) Für jedes P.F. gibt es höchstens eine Kante in L_k. Es gibt nur m P.F.s

b) Sei $(x,y) \in N_0 - L_0$ $\exists j$ mit A$(0,j) \leq$Höhe$(x) <$Höhe$(y) <$A$(0,j+1)$
Da $(x,y) \in N_0 - L_0$, gibt es eine Kante (s,t) auf demselben P.F. mit $s,t) \in N_0$
Es gilt: Höhe$(y) \leq$Höhe$(s) <$Höhe(t)
Nach der Pfadkomprimierung hat x einen neuen Elter y' mit Höhe$(y') \geq$ Höhe(t), da y' Vorfahr von t. Außerdem Höhe$(t) \geq 2j+2$, d.h. (x,y') gehört nicht zu N_0. Also gibt es für jedes x maximal ein y mit $(x,y) \in N_0 - L_0$, also $|N_0 - L_0| \leq n$.

c)+d) Sei x beliebig und j so gewählt, daß $\mathrm{A}(k,j) \leq$Höhe$(x) <\mathrm{A}(k,j+1)$, d.h. $x \in G_{i,j}$.
Seien $y_1, y_2, \ldots, y_q$ alle Endpunkte mit $(x,y_i) \in N_k - L_k$.
Seien Höhe$(y_1) \leq$Höhe$(y_2) \leq \cdots \leq$Höhe(y_q). Für alle $i, 1 \leq i \leq q$ gilt:
Da $(x,y) \in N_k - L_k$, gibt es $(s_i,t_i) \in N_k$. (s_i,t_i) liegt auf gleichem P.F.-Pfad oberhalb von (x_i,y_i). Es gilt:
Höhe$(x) <$Höhe$(y_1) \leq$Höhe$(s_i) <$Höhe$(t_i) <$Höhe(y_{i+1}), da nach Pfadkomprimierung x einen Elter höher als t_i hat. (x,y_{i+1}) wird erst danach betrachtet.
Da $(s_i,t_i) \notin N_{k-1}$, gibt es ein j', so daß Höhe$(s_i) \leq \mathrm{A}(k-1,j') <$Höhe$(t_i)$ und damit Höhe$(y_i) \leq \mathrm{A}(k-1,j') <$Höhe$(y_{i+1})$. Damit gibt es ein j'' mit Höhe$(y_1) <\mathrm{A}(k-1,j'') \leq \mathrm{A}(k-1,j''+q-1) <$ Höhe(y_q)

c) und d) getrennt weiter

c) Es gilt: $(x,y_1),\ldots,(x,y_q) \in N_k$, d.h. $x, y_1, y_2, \ldots, y_q \in G_{k,j}$
Also: $\mathrm{A}(k,j) \leq$Höhe$(y_1) \leq$Höhe$(y_q) <\mathrm{A}(k,j+1) =\mathrm{A}(k-1,\mathrm{A}(k,j))$
Mit $\mathrm{A}(k-1,j''+q-1) <\mathrm{A}(k-1,\mathrm{A}(k,j))$ gilt $j''+q-1 <\mathrm{A}(k,j)$. Also: $q \leq \mathrm{A}(k,j)$. Wir haben gezeigt:
Für jedes $x \in G_{i,j}$ gibt es höchstens $\mathrm{A}(k,j)$ Kanten $(x,y_i) \in N_k - L_k$. Also:

$$|N_k - L_k| \leq \sum_{j \geq 0} |G_{k,j}| \cdot A(k,j)$$

Behauptung: $|G_{k,j}| \leq \dfrac{2n}{2^{A(k,j)}}$

Beweis: Sei $G_{k,j,l} = \{x | x \in G_{k,j}$ mit Höhe$(x) = l\}$ für $\mathrm{A}(k,j) \leq l <\mathrm{A}(k,j+1)$.
Dann ist $|G_{k,j,l}| \leq \dfrac{n}{2^l}$, denn Knoten von Höhe l hat $\geq 2^l$ Nachfolger (L: Gewicht$(x) \geq 2^{Höhe(x)}$). Diese Nachfolgermengen sind disjunkt. Somit:

$$|G_{k,j}| = \sum_{l=A(k,j)}^{A(k,j+1)} |G_{k,j,l}| < \sum_{l=A(k,j)} \frac{n}{2^l} \leq n \cdot 2 \cdot \frac{1}{2^{A(j,k)}}$$

Einsetzen:

$$|N_k - L_k| \leq \sum_{j \geq 0} \frac{2n}{2^{A(k,j)}} \cdot A(k,j) \leq 6 \cdot n$$

d) nachzulesen in Mehlhorn, Datenstrukturen + effiziente Algorithmen I

4 Lineare Programmierung

betrachten Probleme der Form:
max $c_1x_1 + c_2x_2 + \cdots + c_nx_n$, wobei

$$\begin{array}{ccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n \leq b_1 \\ \vdots & + & \vdots & + & \cdots & + & \vdots \leq \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \cdots & + & a_{mn}x_n \leq b_m \end{array}$$

$c_1, \ldots, c_n, a_{11}, \ldots, a_{mn}, b_1, \ldots, b_m \in \mathbb{R}$
$x_1, \ldots, x_n$ reellwertige Variablen

Problem heißt **lineares Programm** (LP)

Beispiel 4.1: Diät-Problem

- n = Nahrungsmittel
- c_j = Preis von Nahrungsmittel j
- k = Nährstoffe
- $\tilde{a}_{ij}$ = Gehalt an Nährstoff i im Nahrungsmittel j
- b_i^M = Tagesmindestbedarf an Nährstoff i
- b_i^H = Tageshöchstbedarf an Nährstoff i

Ziel: Stelle Tagesration aus vorhandenen Nahrungsmitteln zusammen, minimiere Kosten

Schreibweisen:
Sei x_j der Anteil an Nahrungsmitteln j

$\rightarrow$ minimiere $c_1x_1 + c_2x_2 + \cdots + c_nx_n$, wobei
$b_i^M \leq \tilde{a}_{i1}x_1 + \tilde{a}_{i2}x_2 + \cdots + \tilde{a}_{in}x_n \leq b_i^H \quad \forall 1 \leq i \leq k,\ x_j \geq 0$

$\rightarrow$ Umformulieren in Standardform
$c = (c_1, \ldots, c_n)$ Zeilenvektor

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix},\ b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \text{ Spaltenvektoren}$$

$A = (a_{ij})_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}}$ $m \times n$ Matrix.
$a_i = (a_{i1}, \ldots, a_{in})$ Zeilenvektor

$\rightarrow$ Darstellung $\max cx$,
wobei $a_ix \leq b_i$ für $i = 1, \ldots, m$
bzw. $\max\{xc | Ax \leq b\}$ („$\leq$“ komponentenweise)

$\max -cx$:

$$\begin{bmatrix} -\tilde{A} \\ \tilde{A} \\ I \end{bmatrix} \cdot x \leq \begin{bmatrix} -b^M \\ b^H \\ 0 \end{bmatrix},\ \text{dabei ist}$$

$$b^M = \begin{pmatrix} \vdots \\ b_i^M \\ \vdots \end{pmatrix}, b^H = \begin{pmatrix} \vdots \\ b_i^H \\ \vdots \end{pmatrix}, \tilde{A} = (\tilde{a}_{ij})$$

Sei $\max\{cx | Ax \leq b\}$ ein LP.

- cx heißt *Zielfunktion*
- x heißt zulässige Lösung, falls $Ax \leq b$
- $P := \{x \in \mathbb{R}^n | Ax \leq b\}$ heißt *Zulässigkeitsbereich*.

Es gibt drei Möglichkeiten

1. $P = \emptyset$
2. $P \neq \emptyset$ und es gibt $x^* \in P$ mit $cx^* = \max\{cx | Ax \leq b\}$
3. $P \neq \emptyset$ und $\sup\{cx | x \in P\} = +\infty$
   Zielfunktion kann beliebig wachsen

Lösen eines LPs

a) welcher Fall trifft zu ?

b) Nachweis: warum ?

Wiederholung:
$\mathbb{R}^n$ ist ein $\mathbb{R}$-Vektorraum
$a_{i1}x_1 + \cdots + a_{in}x_n = b_i$ definiert eine Hyperebene $h_i = \{x \in \mathbb{R}^n | a_i x = b_i\}$
a_i steht senkrecht auf h_i (Normalvektor, falls $\|a_i\| = 1$). Sei $v \in \mathbb{R}^n$
$a_i v = \cos(a_i, v) \cdot \|a_i\| \cdot \|v\| = \|a_i\| \cdot$(Absatnd von v zur Hyperebene $a_i x = 0$)

Geometrische Interpretation

$P = \bigcap_{i=1}^{n} H_i = \{x | Ax \leq b\}$ ist konvexer Polyeder (kann leer oder unendlich sein). Im Fall 1. ist P leer.

Maximiere cx durch verschieben der Hyperebene $cx = 0$ möglichst weit in Richtung c, so daß der Schnitt mit P nicht leer ist.

Falls 2: Ann. $\exists \geq 1$ Ecken

Optimale Lösung ist Ecke von P, es gibt Hyperebene $cx = f$, die P in Punkt x^* berührt. Punkt x_0 ist Ecke von P gdw. er Schnittpunkt von n linear unabhängigen Hyperebenen ist aus $\{H_1, \ldots, H_m\}$, d.h. n der Nebenbedingungen gelten mit Gleichheit.

Seien o.B.d.A. die ersten n Ungleichungen mit „=“ erfüllt. $a_1 x^* = b_1, \ldots, a_n x^* = b_n$. Betrachte Kegel $K = \{x | a_i x \leq b, 1 \leq i \leq n\}$, x^* ist Spitze des Kegels. Für die Kanten des Kegels gilt:
Für jedes $j, 1 \leq j \leq n$ gibt es eine Kante e_j, die von den Geraden $g_j = h_1 \cap h_2 \cap \ldots \cap h_{j-1} \cap h_{j+1} \cap \ldots \cap h_n$ unterstützt wird und in Halbraum H_j hineinführt.
Für den Richtungsvektor y dieser Kante e_j gilt: $a_i y = 0 \; \forall 1 \leq i \leq n, \; i \neq j$
$a_j y < 0$, etwa $a_j y = -1$, denn e_j führt nicht aus dem Halbraum $cx \leq cx^*$ heraus gdw. $c(x^* + \lambda y) \leq cx^* \Rightarrow cy \leq 0$. Sei

$$A_0 = \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix}$$

, Richtungsvektor y ist definiert durch

$$A_0 y = \begin{pmatrix} 0 \\ 0 \\ -1 \\ 0 \\ 0 \end{pmatrix}$$

, d.h. y ist die j-te Spalte von $-A_0^{-1}$, denn

$$[A_0] \cdot [-A_0^{-1}] = \begin{pmatrix} -1 & & 0 \\ & \ddots & \\ 0 & & -1 \end{pmatrix}$$

Bemerkung: A_0^{-1} definiert, da $h_1, \cdots, h_n$ linear unabhängig. Es gilt also:
K ist im Halbraum $cx \leq f$ enthalten, $f = cx^* \Leftrightarrow cy \leq 0 \ \forall$ Spalten y von $-A_0^{-1} \Leftrightarrow cy \geq 0 \ \forall$ Spalten y von $A_0^{-1} \Leftrightarrow c \cdot A_0^{-1} \geq 0$
$\Leftrightarrow c = u_1 a_1 + \cdots + u_n a_n$ mit $u_1, u_2, \ldots, u_n \geq 0$
$\Leftrightarrow c$ ist in dem von $a_1, \ldots, a_n$ aufspannenden Kegel enthalten, c ist Konvexkombination von $a_1, \ldots, a_n$.

Nachweis:

Fall 3: $x_0 \in P, y \in \mathbb{R}^n$, so daß $x_0 + \lambda y \in P \ \forall \lambda \geq 0$ und $cy > 0$

Fall 2: optimale Lösung $x^* \in P$
Zahlenvektoren $a_{i_1}, \ldots, a_{i_n}$ aus A mit $u_{i_1} \geq 0, \ldots, u_{i_n} \geq 0$, so daß $a_j x^* = b_j \ \forall j \in \{i_1, \ldots, i_n\}, (a_{i_1}, \ldots, a_{i_n}$ linear unabhängig. $c = (u_{i_1} a_{i_1} + \cdots + u_{i_n} a_{i_n}), c$ linear abhängig von $a_{i_1}, \ldots, a_{i_n}$. Erweitere $u_{i_1}, \ldots, u_{i_n}$ zu Zeilenvektor der Länge m. Setze $u_k = 0$ für $k \in \{1, \ldots, m\} \backslash \{i_1, \ldots, i_m\}$, dann ist $uA = c$. Für $x \in P$ gilt:

$$\begin{aligned} cx &= (uA)x = u(Ax) \\ &\leq ub \\ &= (u_1, \ldots, u_m) \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \\ &= (u_{i_1}, \ldots, u_{i_n}) \begin{pmatrix} b_{i_1} \\ \vdots \\ b_{i_n} \end{pmatrix}, u_k = 0 \text{ für } k \notin \{i_1, \ldots, i_n\} \\ &= (u_{i_1}, \ldots, u_{i_n}) \begin{pmatrix} a_{i_1} \\ \vdots \\ a_{i_n} \end{pmatrix} \cdot x^* \\ &= c \cdot x^* \end{aligned}$$

Fall 1: Ein Vektor $y \geq 0$ mit $yA = 0$ und $yb < 0$
Wäre $Ax \leq b$ für ein $x \in \mathbb{R}^n$, so wäre $\underbrace{yAx}_{=0} \leq \underbrace{yb}_{<0}$, da $y \geq 0$ (Widerspruch)

später: y ex. immer im Fall 1

4.1 Simplexalgorithmus

Wir nehmen an, wir kennen die Ecke x_0 von P, d.h. wir kennen ein Teilsystem $A_0 x \leq b_0$ der Größe n mit

$$\left.\begin{array}{l} 1)\ A_0 x_0 = b_0 \\ 2)\ A_0 \text{ nicht singulär} \end{array}\right\} x_0 \text{ Schnittpunkt von } n \text{ Hyperebenen}$$

Stelle c als Linearkombination der Zeilen aus A_0 dar. Sei $c = u_0 A_0$, d.h. $u_0 = A_0^{-1} c$. Erweitere u_0 zu $u \in \mathbb{R}^m$, indem die Komponenten, die nicht Zeilen aus A_0 entsprechen, auf 0 gestetzt werden. Dann ist $c = uA$.

Fall 1: $u_0 \geq 0$
Dann ist x_0 optimal, u_0 und Zeilen von A dienen als Nachweis. c ist Konvexkombination der Zeilen von A_0.

Fall 2: $\neg(u_0 \geq 0)$
(*) wähle i^* mit $u_{i^*} < 0$ (existiert)
c ist keine Konvexkombination der Zeilen von A_0, Hyperebene $cx = cx_0$ ist also nicht tangential im Punkt x_0.
Sei I_0 die Menge der Indizes der Zeilen von A_0. Betrachte Kante $e_{i^*} \in \bigcap_{\substack{i \in I \\ i \neq i^*}} h_i$, die in H_{i^*} hineinführt. Sei y Richtungsvektor mit $a_{i^*}y = -1$, $a_i y = 0$ mit $i \in I_0, i \neq i^*$ (y ist Spalte i^* von $-A_0^{-1}$). Dann gilt: $cy = u_0 A_0 y = -u_{i^*} > 0$.

Fall 2.1: $Ay \leq 0$, d.h. für alle Zeilen von A ist $a_i y \leq 0$
$\Rightarrow x_0 + \lambda y \leq P$ für alle $\lambda > 0$, denn $A(x_0 + \lambda y) = Ax_0 + \lambda \underbrace{Ay}_{\leq 0} \leq Ax_0 \leq b$.
Ferner $cy > 0$, d.h. $c(x_0 + \lambda y) = cx_0 + \lambda \underbrace{cy}_{>0}$ wird beliebig groß, unendliches Maximum.

Fall 2.2 Für mindestens eine Zeile von A ist $a_i y > 0$.
Sei λ_0 maximal, so daß $x_0 + \lambda_0 y \in P$, d.h.

$$\begin{aligned} \lambda_0 &= \max\{\lambda | a_j(x_0 + \lambda y) \leq b_j \ \forall j = 1, \ldots, m\} \\ &= \max\{\lambda | a_j(x_0 + \lambda y) \leq b_j \ \forall j : a_j y > 0\} \\ &= \min\left\{ \frac{b_j - a_j x_0}{a_j y} | a_j y > 0, \ 1 \leq j \leq m \right\} \end{aligned}$$

(**) wähle j^*, für das das Minimum angenommen wird.
Sei $x_1 = x_0 + \lambda_0 y$ und sei A_1 die Martix, die man aus A_0 erhält beim Ersetzen von Zeile i^* durch Zeile j^*. Sei b_1 der Teil von b, der A_1 entspricht.

Iteriere:

Wir erhalten eine Folge von Subsystemen $(A_0, x_0, b_0), (A_1, x_1, b_1), \ldots$ mit $cx_0 \leq cx_1 \leq \ldots$ ($\lambda_0 \neq 0$ kommt vor, wenn mehr als n Hyperebenen durch eine Ecke gehen).

Achtung: Endlosschleife möglich

Bei allgemeiner Lage der Hyperebenen ist obiges nicht möglich. Das Verfahren terminiert, daraus folgt Korrektheit.

Abhilfe: *Bland*'s kleinste Indexmethode

Wähle i^* und j^* in (*) und (**) minimal

Satz 4.1: Bei minimaler Wahl der Indices terminiert das Simplexverfahren.

Beweis (idirekt):

Annahme: terminiert nicht, d.h. es liegt eine unendliche Folge von Subsystemen vor $((A_0, x_0, b_0), \ldots)$
Es gibt nur $\leq \binom{m}{n}$ verschiedene Systeme
$\Rightarrow s, t \in \mathbb{N}, s < t$ mit $A_s = A_t$
$\Rightarrow cx_s = cx_{s+1} = \ldots = cx_t \Rightarrow x_s = x_{s+1} = \ldots = x_t$
Sei r der maximale Index, so daß a_r in einer der Iterationen $s, \ldots, t-1$, etwa p, gestrichen wird. Dann muß a_r in einer der Iterationen, etwa q, eigefügt werden.
Für $j > r$ und $s \leq k \leq t$: Zeile a_j in A_s = Zeile a_j in A_k

Behauptung: In Iteration k wird der Richtungsvektor y_k gewählt, mit $cy_k > 0$. $x_{k+1} = x_k + \lambda y_k$, also $cx_{k+1} = cx_k + \lambda \underbrace{cy_k}_{>0} \Rightarrow \lambda = 0$ ❏

Betrachte die Iterationen p und q. Sei $\bar{u}_k$ der Vektor u_k erweitert mit Nullen, d.h. $c = \bar{u}_k A$. Es gilt:

$$\begin{aligned} 0 &< cy_k \\ &= \bar{u}_p A y_q \\ &= \bar{u}_p \begin{pmatrix} a_1 y_q \\ a_2 y_q \\ \vdots \\ a_m y_q \end{pmatrix} = \sum_{k=1}^{m} \bar{u}_{p_k}(a_k y_q) \end{aligned}$$

$\Rightarrow$ Es gibt einen Index k mit $\bar{u}_{p_k}(a_k y_q) > 0$. a_k ist eine Zeile von A_p, da sonst $\bar{u}_{p_k} = 0$. Für $j > r$ ist $a_j y_q = 0$, da Zeile a_j in der Iteration q nicht gestrichen wird.
$\Rightarrow k \leq r$. Zeile r wird in einer Iteration p gestrichen, d.h. $\bar{u}_{p_r} < 0$ und in der Iteration q hinzugefügt, d.h. $a_r y_q > 0$.
$\Rightarrow k \neq r$. a_k ist eine Zeile von A_p mit $k < r$. Wenn r in A_q eingefügt wird, gilt:

$$\begin{aligned} r &= \min\{j | a_j y_q > 0 \wedge \frac{b_j - a_j x_q}{a_j y_q} = \underbrace{\lambda_0^{(q)}}_{=0}\} \\ &= \min\{j | a_j y_q > 0 \wedge a_j x_q = b_j\} \end{aligned}$$

$\Rightarrow a_k y_q \leq 0$, da a_k Zeile von A_p und damit $a_k x_p = a_k x_q = b_k$. Wenn r in A_p gestrichen wird, dann gilt: $r = \min\{i | \bar{u}_{p_i} < 0\}$
$\Rightarrow \bar{u}_{p_k} \geq 0 \Rightarrow \bar{u}_{p_k} a_k y_q \leq 0$. Widerspruch zu $\bar{u}_{p_k}(a_k y_q) > 0$ ❏

Wie findet man eine Startecke x_0?

$$\max\{cx | Ax \leq b\}, b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

o.B.d.A.: $b_1, \ldots, b_k \geq 0, b_{k+1}, \ldots, b_m < 0$. Sei

$$b^{(1)} = \begin{pmatrix} b_1 \\ \vdots \\ b_k \end{pmatrix}, b^{(2)} = \begin{pmatrix} b_{k+1} \\ \vdots \\ b_m \end{pmatrix}$$

Spalte A analog in $A^{(1)}$ und $A^{(2)}$

(I) $\max\{cx | A^{(1)} x \leq b^{(1)} \wedge A^{(2)} x \leq b\}$
Seien x', x'' n-Vektoren, x''' $(m-k)$-Vektor

(II) $\max\{-(x'''_1 + \cdots + x'''_{m-k} | x', x'', x''' \geq 0, A^{(1)}(x' - x'') \leq b^{(1)}, A^{(2)}(x' - x'') - x''' \leq b^{(2)}\}$
$c = (\underbrace{0, \ldots, 0}_{2n}, \underbrace{-1, \ldots, -1}_{m-k})$

$$\begin{pmatrix} -I & 0 & 0 \\ 0 & -I & 0 \\ 0 & 0 & -I \\ A^{(1)} & -A^{(2)} & 0 \\ A^{(1)} & -A^{(2)} & -I \end{pmatrix} \cdot \begin{pmatrix} x' \\ x'' \\ x''' \end{pmatrix} \leq \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ b^{(1)} \\ b^{(2)} \end{pmatrix}$$

Es gilt:

1. (I) hat Lösung $\Leftrightarrow$ (II) hat Lösung mit $x''' = 0$

   Beweis:

   $\Rightarrow$: Sei $\tilde{x} = (\tilde{x}_1, \ldots, \tilde{x}_n)$ Lösung vom (I)
   Setze $x_i' = \tilde{x}_i$, $x_i'' = 0$, falls $\tilde{x}_i \geq 0$
   $x_i' = 0$, $x_i'' = -\tilde{x}_i$, falls $\tilde{x}_i < 0$
   und $x_i''' = 0$.
   Mit $\tilde{x}_i = x_i' - x_i''$ folgt $A^{(1)}(x_i' - x_i'') \leq b^{(1)}$
   $A^{(2)}(x_i' - x_i'') \leq b^{(2)}$
   $\Leftarrow$: Sei $\tilde{x} = (\tilde{x}', \tilde{x}'', 0)$ Lösung von (II) mit $x''' = 0$
   (I) hat Lösung $\tilde{x}' - \tilde{x}''$.

2. (II) hat Lösung mit $x''' = 0 \Leftrightarrow \max_{(II)} = 0$
3. $x^* = (0, \ldots, 0, 0, \ldots, 0, -b_{k+1}, \ldots, -b_m)$ ist Ecke von $P_{(II)}$.

   Beweis:

   Betrachte $(2n + m - k)$-dimensionales System:

$$\begin{pmatrix} -I & 0 & 0 \\ 0 & -I & 0 \\ A^{(1)} & A^{(2)} & -I \end{pmatrix} \cdot x \leq \begin{pmatrix} 0 \\ \vdots \\ 0 \\ b^{(2)} \end{pmatrix}$$

   Rang $2n+m-k \Leftrightarrow$ Matrix regulär, d.h. x^* ist Ecke von $P_{(II)}$. Für (II) ist eine Ecke bekannt. Löse (II) mit Simplexverfahren. Sei $x_0 = (x_0', x_0'', x_0''')$ optimale Lösung. $x''' \neq 0 \Rightarrow$ Problem (I) hat keine Lösung.
   $x_0''' = 0$: Betrachte
   (I') $\max\{c(x' - x'') | A(x' - x'') \leq b, x', x'' \geq 0\}$
   Behauptung: (x_0', x_0'') ist Ecke von $P_{(I')}$.

   Beweis:

   Zu $x_0 = (x_0', x_0'', 0)$ gehört nichtsinguläres Subsystem von (II) der Größe $2n + m - k$ mit Gleichheit, etwa $\tilde{A}x \leq \tilde{b}$.

$$\tilde{A} \begin{pmatrix} x_0' \\ x_0'' \\ 0 \end{pmatrix} = \tilde{b}$$

   Sei $\tilde{B}$ eine Teilmatrix von $\tilde{A}$ bestehend aus den ersten $2n$ Spalten $\Rightarrow$ Rang $\tilde{B} + 2n$
   Wähle $2n$ linear unabhängige Zeilen von $\tilde{B} \rightarrow$ Matrix $\tilde{\tilde{B}}$ ($2n \times 2n$ Matrix). $\tilde{\tilde{B}}$ ist Subsystem von (I'):

$$\begin{pmatrix} -I & 0 \\ 0 & -I \\ A^{(1)} & -A^{(1)} \\ A^{(2)} & -A^{(2)} \end{pmatrix} \cdot \begin{pmatrix} x' \\ x'' \end{pmatrix} = \begin{pmatrix} -x' \\ -x'' \\ A^{(1)}(x' - x'') \\ A^{(2)}(x' - x'') \end{pmatrix} \leq \begin{pmatrix} 0 \\ \vdots \\ 0 \\ b^{(1)} \\ b^{(2)} \end{pmatrix}$$

Es gilt:
$\tilde{\tilde{B}}\binom{x'_0}{x''_0} = \tilde{\tilde{b}}$, wobei $\tilde{\tilde{b}}$ Teil von $\tilde{b}$, der $\tilde{\tilde{B}}$ entspricht. $\Rightarrow (x'_0, x''_0)$ Ecke von (I').

Es gilt:
Falls $(\tilde{x}'_0, \tilde{x}''_0)$ optimale Lösung von (I'), dann ist $(\tilde{x}'_0 - \tilde{x}''_0)$ optimale Lösung von (I). ❏

Was tun im Fall 1:

Lemma 4.1: (Lemma von Farkas)
$P = \{x | Ax \leq b\} = \emptyset \Rightarrow \exists y \geq 0$ mit $yA = 0$ und $yb > 0$.

Beweis:
Annahme: $\forall y$ mit $y \geq 0$ und $yA = 0 \Rightarrow yb \geq 0$. Dann ist

$$\begin{aligned} 0 &= \min\{yb | yA = 0, y \geq 0\},\ y^T = v \\ &= \min\{b^T v | A^T v = 0, v \geq 0\} \\ &= \max\{-b^T v | -A^T v \leq 0, A^T v \leq 0, -v \leq 0\} \end{aligned}$$

$v_0 = 0$ ist optimale Ecke des Polyeders $\{v | A^T v \leq 0, -A^T v \leq 0, -v \leq 0\}$ bezüglich einer Zielfunktion $-b^T v$.
Also gibt es einen Vektor u mit

$$u \begin{bmatrix} -A^T \\ A^T \\ I \end{bmatrix} = -b^T$$

bzw. $[-A\ A\ I]u^T = b$.
Sei $u^T = (u', u'', u''')$ und $\tilde{u}^T = (u', u'')^T$. Da $u^T \geq 0$, also $u''' \geq 0$, gilt:
$[A\ -A]\tilde{u}^T \leq b$ und somit $A(u'^T - u''^T) \leq b$, also $u'^T - u''^T \in P$. ❏

4.2 Dualität

Satz 4.2:

Falls $F_p = \{x \in \mathbb{R}^n | Ax \leq b\}$ und $F_q = \{y \in \mathbb{R}^m | yA = c, y \geq 0\}$ nicht leer sind, gilt: $\max\{cx | Ax \leq b\} = \min\{yb | yA = c, y \geq 0\}$

Beweis:

Seien x und y Vektoren mit $Ay \leq b, yA = c$ und $y \geq 0$. Dann gilt: $cx = yAx \leq yb$, da $Ax \leq b$ und $y \geq 0$, d.h.
falls $\sup\{cx | Ax \leq b\} = \infty$, so ist $F_q = \emptyset$
falls $\inf\{yb | yA = c, y \geq 0\} = -\infty$, so ist $F_p = \emptyset$
falls $F_p, F_q \neq \emptyset$: es gibt endliche Lösungen.
Sei x^* optimale Lösung in F_p. Simplexmethode liefert $u \geq 0$ mit $uA = c$ und $u_i = 0$ falls $(Ax^*)_i < b_i$. Dann gilt: $ub = \sum u_i b_i = \sum u_i (Ax^*)_i = uAx^* = cx^*$. ❏

4.3 Alternative Darstellung

(LP) $\min \sum c_i, x_i$ mit $\sum a_{ij} x_j \leq b, i = 1, \ldots, m\ x_j \geq 0$
Mache aus Ungleichungen Gleichungen durch Schlupfvariablen $x_{n+1} \geq 0$
$\rightarrow (\min cx$ mit $Ax = b, x \geq 0)$ mit $x = (x_1, \ldots, x_{n+m}^T), c = (c_1, \ldots, c_n, 0, \ldots, 0)^T$

$$A = [\tilde{A}\, I^m],\ \tilde{A} = (a_{ij})_{\substack{i=1,\ldots,m \\ j=1,\ldots,n}} = [A_1 \ldots A_{n+m}]$$

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

Sei $P = \{\hat{x} \in \mathbb{R}^n | \tilde{A}\hat{x} \leq b,\ \hat{x} \geq 0\}$
$F = \{x \in \mathbb{R}^{m+n} | Ax = b,\ x \geq 0\}$
Es gilt:

1. $\hat{x} \in P \Rightarrow x \in F$ mit $x_i = \hat{x}_i$ für $1 \leq i \leq n$, $x_{n+i} = b_i - \tilde{a}_i\hat{x}_i$
2. $x \in F \Rightarrow (x_1, \ldots, x_n) \in P$.

$x = (x_1, \ldots, x_n) \in P$ heißt *Basislösung*, falls es eine nichtsinguläre Teilmatrix $A_B = [\underbrace{A_{j_1}, \ldots, A_{j_m}}_{\text{Basis B}}]$ von A gibt, so daß

$$A_B \cdot \begin{pmatrix} x_{j_1} \\ \vdots \\ x_{j_m} \end{pmatrix}$$

und $x_j = 0$ für $j \notin \{j_1, \ldots, j_m\}$
x heißt zulässige Basislösung (bfs), falls außerdem $x_{j_k} \geq 0$ für alle x_{j_k} : k-te Komponente von $A_B^- 1 b$.

Beispiel 4.2:

$$\begin{array}{c} x_1 + x_2 + x_3 + x_4 + 0 + 0 = 4 \\ x_1 + 0 + 0 + 0 + x_5 + 0 = 2 \\ 0 + 0 + x_3 + 0 + 0 + x_6 = 3 \end{array}$$

$A_B = [A_3\ A_4\ A_5\ ]$

$$A_B \cdot \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix},\ x = A_B^{-1} b$$

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \\ 3 \end{pmatrix} \Rightarrow \begin{array}{l} x_5 = 2 \\ x_3 = 3 \\ x_4 = 1 \end{array}$$

$$\text{bfs} \rightarrow \begin{pmatrix} 0 \\ 0 \\ 3 \\ 1 \\ 2 \\ 0 \end{pmatrix}$$

$$A_B = [A_1 A_2 A_3] \rightarrow \text{Basislösung} \begin{pmatrix} 2 \\ -1 \\ 3 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Lemma 4.2:

$\hat{x}$ ist Ecke von $P \Leftrightarrow$ zugehöriger Vektor $x \in F$ ist zulässige Basislösung (bfs)

Zum Simplex: Wie wechselt man die Ecke (Basis)?

A war $m \times (m+n)$ Matrix, nenne $m+n$ nun n $(m < n)$
m linear unabhängige Spalten bilden eine Basis. Sei A_j die neu gewählte Spalte (Pivotspalte). Da $A_j \notin B : \sum_i x_{ij} A_{B(i)} = A_j \ \sum_i x_{i0} A_{B(i)} = b$. x_0 gibt die Werte der Basisvariablen von x_0. x_0 Basislösung.

$(**) - \lambda \cdot (*)$ (aus Ecke x_0 gehe um λ in Richtung A_j)
$\Rightarrow \sum_{i=1}^{m} x_{i_0} A_{B(i)} - \lambda \cdot x_{i_j} A_{B(i)} + \lambda A_j = b$
$\sum_{i=1}^{m} (\underbrace{x_{i_0} - \lambda x_{i_j}}_{x_{(\lambda)}}) A_{B(i)} + \lambda A_j = b$

$x_{(\lambda)}$ zulässig, wenn $x_{i_0} - \lambda x_{i_j} \geq 0$ für $i = 1, \ldots, m \Rightarrow \lambda \leq \min_i \frac{x_{i_0}}{x_{i_j}}$

Beispiel 4.3:

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

$\overset{Tableau}{\Rightarrow}$

	x_1	x_2	x_3	x_4	x_5
1	3	2	1	0	0
3	5	1	1	1	0
4	2	5	1	0	1

1	3	2	1	0	0
2	2	-1	0	1	0
3	-1	3	0	0	1
	↑				
	Pivotspalte				

Pivotelement mit $\max \lambda$
$\min \left\{ \frac{1}{3}, \frac{2}{2} \right\}$

Wie kommt man zu einer besseren bfs? (wahl von A_j)

$$c_1 x_1 + \cdots + c_n x_n = c^T x$$

stelle Zielfunktion als Funktion der Nichtbasisvariablen dar.

$$x_{B(i)} = x_{i_0} - \sum_{j \notin B} x_{i_j} x_j$$

$$z = \sum_{i=1}^{n} c_i x_i = \sum_{i \notin B} c_i x_i + \sum_{i=1}^{m} c_{B(i)} \cdot \underbrace{x_{B(i)}}_{x_{i_0} - \sum_{j \notin B} x_{i_j} x_j}$$

$$z = \underbrace{\sum_{i=1}^{m} c_{B(i)} \cdot x_{i_0}}_{z_0} + \sum_{j \notin B} \bar{c}_j x_j, \bar{c}_j = c_j - \sum x_{i_j} c_{B(i)}$$

z_0 = Wert der Zielfunktion an bfs zur Basis B. Ist $\bar{c}_j < 0$ und wird x_j in Basis eingeführt, wird z kleiner.

Bland'sche Regel

- Wähle $\bar{c}_j < 0$ mit größtem Betrag und kleinstem Index.
- Als Pivotelement wähle das mit kleinstem λ und kleinstem Zeilenindex
  $\rightarrow$Terminierung

Phase 1: (Finden einer Ecke)

Dualität

Satz 4.3: Primales LP hat den selben Optimalwert wie das duale.

Satz 4.4: Das duale Problem des dualen ist primal

Satz 4.5: (komplementärer Schlupf)
Paar x, Π ist optimal $\Leftrightarrow$
$(*)$ $u_i = \Pi_i(a_i^T x - b_i) = 0 \ \forall i$
$(**)$ $v_i = (c_j - |pi^T A_j)x_j = 0 \ \forall j$

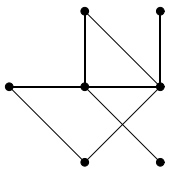
Beweis:

Es gilt: $u_i, v_j > 0 \ \forall i, j$ nach Problemdefinition (Primal$\leftrightarrow$dual)
Sei $u = \sum_i u_i \geq 0, v = \sum_i v_i \geq 0$
$u = 0 \Leftrightarrow (*)$
$v = 0 \Leftrightarrow (**)$
Es gilt: $u + v = c^T x - \Pi^T b$, d.h. $u + v = 0 \Leftrightarrow c^T x = \Pi^t b$, d.h. x, Π sind optimale Lösungen. ❏

5 NP-Vollständigkeit

Wir betrachten das Problem CLIQUE.

Gegeben: Graph $G = (V, E)$ mit n Knoten, Zahl k
Gibt es einen vollständigen Teilgraph von G der Größe k?



Algorithmus: Teste alle $\binom{n}{k}$ k-elementige Teilmengen.

Laufzeit: Für $k = \frac{n}{2}$: $\binom{n}{\frac{n}{2}} = \frac{n!}{\frac{n}{2}!\frac{n}{2}!} \geq \frac{2^n}{(n+1)}$

Sei der Graph binär codiert durch eine Adjazenzmatrix (Bitstring der Länge n^2, k hat Darstellung der Länge $\log \frac{n}{2}(k = \frac{n}{2})$). Der Algorithmus akzeptiert die Sprache CLIQUE=$\{w\#v | w, v \in \{0,1\}^*, |w| = n^2$ für ein n, Graph mit Adj.matrix w hat CLIQUE der Größe k (v Binärdarstellung von k) $\}$
in Zeit $\Omega(\frac{2^{\sqrt{m}}}{\sqrt{m}})$, wobei m Inputlänge, $m = w\#v$

Wir beschreiben das Durchlaufen aller Möglichkeiten durch einen nichtdeterministischen Algorithmus.
$\hookrightarrow$ Operation: Wähle Marke1, Marke2
lassen einen Dämon die Wahl machen.

Der nichtdeterministische Algorithmus akzeptiert Input, wenn es mindestens eine akzeptierte Berechnung auf diesem Input gibt. Laufzeit: Länge der kürzesten akzeptierten Berechnung.

Nichtdeterministischer Algorithmus für CLIQUE

1. lese Input $w\#v$, berechne $n = \sqrt{|w|}$ und k
   bei Inkonsistenz wird abgebrochen. $O(n^2)$

2. wähle nichtdeterministische Teilmenge $V' \subseteq V$ der Größe k durch generieren des Bitvektors $A[1 \ldots n]$ mit genau k Einsen. $A[i] = 1 \Leftrightarrow i \in V$

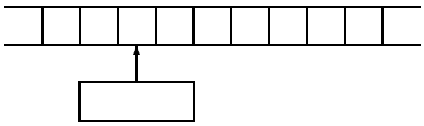
   ```
   for i = 1 to n do
       wähle m1, m2;
       m1 : A[i] ← 0;
            goto m;
       m2 : A[i] ← 1;
            k ← k − 1;
       m : od;
   if k ≠ 0 then Abbruch
   ```

   $O(n)$

3. Teste V' auf Cliqueneigenschaft $O(n^2)$

NP ist eine Klasse von Problemen, die auf einer *nichtdeterministischen* Maschine in polynomieller Zeit gelöst werden können.
P ist die Klasse von Problemen, die auf einer *deterministischen* Maschine in polynomieller Zeit gelöst werden können.

Einfaches Maschinenmodell: **Turingmaschine**



Kontrolleinheit mit endlicher Anzahl von Zuständen liest/schreibt, bewegt Kopf (versucht in akzeptierenden Zusatnd zu gelangen).

Definition 5.1: (Turingmaschine)

Eine nichtdeterministische Turingmaschine (TM) ist ein 4-Tupel $M = (Z, \Gamma, \delta, F)$, mit
$Z = \{Z_1, \ldots, Z_s\}$ endliche Menge der Zustände
$\Gamma = \{A_1, \ldots, A_v\}$ ist endliches Bandalphabet
Z_1 : Anfangszustand
$F \subseteq Z$: Menge der akzeptierenden Zustände
$\delta : Z \times \Gamma \to 2^{Z \times \Gamma \times \{-1,0,1\}}$ Übergangsfunktion

Die TM ist deterministisch, falls $|\delta(z,a)| \leq 1$ für jeden Zusatnd z und jeden Bandeintrag a.

A_1 repräsentiere den leeren Bandeintrag. Eine Konfiguration $C \in \Gamma^*(Z \times \Gamma)^*$ beschreibt einen Schnappschuß einer Berechnung. Für $C = w(z,a)v$ ist der Bandinhalt wav, z Zusatnd, Lesekopf steht auf a.

$T_M(n) = \max_{\substack{x \in \Gamma^* \\ |x|=n}} \{k | k$ Länge einer Halteberechnung an $x\}$, falls M deterministisch.
$T_M(n) = \infty$, falls $\exists x \in \Gamma^* : |x| = n$, so daß M an x nicht hält.
$T_M(n) = \max_{\substack{x \in \Gamma^* \\ |x|=n}} \min_{Berechnung}\{k | k$ Länge einer akzeptierenden Berechnung an $X\}$, falls M nichtdeterministisch.

Definition 5.2: (P,NP)

$P = \{L | L \subseteq \Gamma^*$ für ein endliches Alphabet Γ und es gibt eine DTM M und ein Polynom p, so daß $L = L(M)$ und $T_M(n) \leq p\}$

$NP = \{L | L \subseteq \Gamma^*$ für ein endliches Alphabet Γ und es gibt eine NTM M und ein Polynom p, so daß $L = L(M)$ und $T_M(n) \leq p\}$

Es ist klar, daß $P \subseteq NP$
Frage: Ist $P \neq NP$?

Es gibt vile Probleme, die etwa gleich schwierig sind:
CLIQUE, Travelling Salesman, Knotenüberdeckung, ...

Definition 5.3: (Reduktion)

a) Seien Σ, Γ endliche Alphabete.
Abb: $f : \Sigma^* \to \Gamma^*$ ist eine Transformation, die in polynomieller Zeit auf einer TM M berechenbar ist

b) Seien $L_1 \subseteq \Sigma^*$, $L_2 \subseteq \Gamma^*$ Sprachen.
L_1 ist reduzierbar auf L_2, wenn es eine Transformation f bibt, mit $x \in L_1 \Leftrightarrow f(x) \in L_2 \; \forall x \in \Sigma^*$

c) L ist NP-vollständig, falls

   i) $L \in$NP

   ii) $L' \leq L \; \forall L' \in$NP

$A \leq B$ bedeutet: „A reduzierbar auf B“

Satz 5.1:
Sei L_0 NP-vollständig

1. $L_0 \in$P$\Leftrightarrow$P=NP
2. Falls $L_0 \leq L_1$ und $L_1 \in$NP, so ist L_1 NP-vollständig.

Beweis:

1. $\Leftarrow$: $P = NP \Rightarrow L_0 \in P$
   $\Rightarrow$: Sei $L_0 \in P$ und L beliebig aus NP.
   L_0 wird von einer DTM N in Polynomzeit(p) akzeptiert.
   L_0 NP-vollständig und $L \in NP \Rightarrow L \leq L_0$,
   d.h. es gibt eine Transformation f mit $L = f^{-1}(L_0)$.
   f kann in Polynomzeit(q) berechnet werden, durch DTM N.
   Wir wollen eine DTM konstruieren, die jedes $x \in L$ akzeptiert.
   1. Aus Eingabe x berechne $f(x)$ mit N in Zeit $q(|x|)$.
   2. schreiben $f(x)$ auf Band, setze Kopf auf ersters Zeichen.
      Zeit$|f(x)|$
   3. Entscheide mit M, ob $f(x) \in L_0$
      Ist $f(x) \in L_0$, akzeptiere x sonst lehne ab.
   Gesamtzeit: $q(|x|) + |f(x)| + p(|f(x)|) = r(x)$

   ❏

2. z.z.: $\forall L \in NP : L \leq L_1$
   Da L_0 NP-vollständig, gibt es Transformation f mit $L = f^{-1}(L_0)$. Da $L_= \leq L_1$, gibt es Transformation g mit $L_0 = g^{-1}(L_1)$. Bilde $h = g \circ f$. Es gilt: $L = f^{-1}(L_0) = f^{-1}(g^{-1}(L_1)) = (g \circ f)^{-1}(L)$.
   z.z.: h kann von DTM berechnet werden (geht analog zu a). ❏

Aus Satz folgt:

a) NP-vollständige Probleme sind die schwersten in NP. Ist eines davon in P, dann alle.

b) $\rightarrow$Beweisschema zum Beweis von NP-vollständig

   1. Zeige NP-Vollständigkeit für eine Sprache direkt
   2. Für alle anderen (L_1) zeige: $L_1 \in NP \wedge L_0 \leq L_1$

Als L_0 wähle Erfüllungsproblem SAT.
Seien $V = \{x_1, x_2, \ldots\}$ Variablen, x_i und $\bar{x}_i$ sind Literale. $(y_1 \vee y_2 \vee \ldots \vee y_k)$ = Klausel vom Grad k, wenn $y_1, \ldots, y_k$ Literale. $(c_1 \wedge c_2 \wedge \ldots \wedge c_m)$ Formel in konjunktiver Normalform, falls $c_1, \ldots, c_m$ Klauseln.

Zuweisung ist eine Abb. $\Psi : V \rightarrow \{0, 1\}$ mit

$$\Psi(y) = \begin{cases} \Psi(x_i) & \text{falls} \quad y = x_i \\ 1 - \Psi(x_i) & \text{falls} \quad y = \bar{x}_i \end{cases}$$

$\Psi(y_1 \vee \ldots \vee y_k) = \max\{\Psi(y_i) | 1 \leq i \leq k\}$
$\Psi(x_1 \wedge \ldots \wedge c_m) = \min\{\Psi(c_i) | 1 \leq i \leq m\}$

Formel α erfüllabr, wenn es Ψ gibt mit $\Psi(\alpha) = 1$. Problem heißt 3SAT (2SAT), falls $k = 3$ ($k = 2$) für alle Klauseln.

Satz 5.2: SAT ist NP–vollständig

Beweis:

1. SAT∈NP. α sei Formel in konjunktiver Normalform,
   V_α Variablenmenge $|V_\alpha| = m \leq |\alpha|$.
   Folgende NTM N akzeptiere SAT:
   N wählt Zuweisung $\Psi : V_\alpha \to \{0,1\}$ durch Hinschreiben eines Bitstrings der Länge m.
   $\Psi(\alpha)$ wird ausgewertet. α wird akzeptiert $\Leftrightarrow \Psi(\alpha) = 1$. N abrbeitet in polynomieller Zeit und akzeptiert SAT

2. Sei L∈NP beliebig, L≤SAT
   Sei M eine NTM, die L in polynomieller Zeit p akzeptiert. Konstruieren für jede Eingabe x für M eine Formel $\alpha(x)$, die das Verhalten von M an x beschreibt. $\alpha(x)$ ist erfüllbar, wenn NTM M die Eingabe x akzeptiert. $x \to \alpha(x)$ ist gewünschte Transformation. ❑

Beispiel 5.1: für ein NP–vollständiges Problem: Knotenüberdeckung VC

Problem: Graph $G = (V, E)$. Zahl $k = |V|$.
Frage: Gibt es Knotenüberdeckung der Größe$\leq k$ für G, d.h. ein $V' \subseteq V$, so daß $|V'| \leq k$ und für jede Kante $(u, v) \in E$: u oder v oder beide $\in V'$?

Satz 5.3: Knotenüberdeckung (VC) ist NP–vollständig

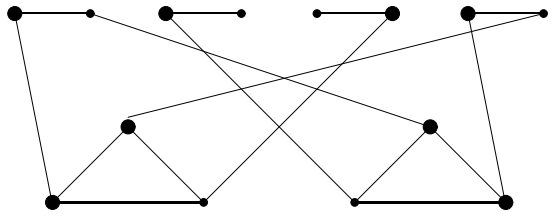
Beweis:

a) z.z.: VC∈NP, da eine NTM ein V' rät und die Korrekheit in polynomialer Zeit überprüfen kann.

b) wir werden sehen: 3SAT ist NP–vollständig. z.z.: 3SAT≤VC
   wir transformieren 3SAT-Problem zu VC-Problem und zeigen, daß mit Lösung des VC-Problems auch 3SAT gelöst ist.

Beispiel 5.2: für 3SAT

$\alpha = (x_1 \vee \bar{x}_3 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee x_2 \vee x_4)$
Für jede Variable $x_i \in V$ bilde Graph T_i.
Für jede Klausel $c_j \in C$ bilde Komponente S_j.
Seien x_j, y_j, z_j Literale in Klausel c_j. Füge Kanten $\{a_1[j], x_j\}, \{a_2[j], y_j\}, \{a_3[j], z_j\}$ ein. Setze $k = n + 2m$



Graph kann in polynomieller Zeit konstruiert werden. Z.z.: α erfüllbar $\Leftrightarrow$ G hat VC der Größe k.

$\Leftarrow$: Sei $V' \subseteq V$ so ein VC. V' muß genau einen Konten in jedem T_i haben und genau zwei in jedem S_j. Setze $\Psi(x_i) = 1$, wenn $x_i \in V'$, auf 0 sonst. Von den beiden Knoten aus V' in der Klausel c_j werden nur 2 ausgehende Kanten überdeckt.

Eine muß von T_i her bedeckt werden. Diese setzt Klausel c_j auf 1.
Das gilt für alle Klauseln, also auch für α.
$\Rightarrow$: Sei $\Psi : V \to \{0,1\}$ eine erfüllbare Zuweisung.
Konstruiere analog die VC. Ist $\Psi(x_i) = 1$, setze $x_i \in V$ und die beiden anderen Knoten aus jeder Klausel, in der x_i vorkommt, $\in V' \Rightarrow$ VC komplett.

Ausführlicher Beweis für SAT ist NP–vollständig

Sei L$\in$NP. L$\leq$SAT. Sei M eine NTM, z_1 Anfangszustand, $z_r, \ldots, z_s$ akzeptierende Zustände. Wenn $\delta(z,a) = \emptyset$, setze $\delta(z,a) = (z,a,0)$. M hält niemals, Haltekonfigurationen werden wiederholt. M akzeptiert $x \Leftrightarrow$ es gibt eine Konfiguartionsfolge $C_1, C_2, \ldots, C_{p(n)}$, so daß C_1 Anfangskonfiguration für x, $C_i \vdash C_{i+1}$ für $1 \leq i \leq p(n)$. Zusatnd in $C_{p(n)}$ ist akzeptierend. δ definiert Relation $Z \times \Gamma \times Z \times \Gamma \times \{-1,0,1\}$. Nummeriere Übergänge durch. Sei m die Anzahl der Tupel in δ. Wir benutzen folgende Variablen:

$z_{t,k}$	$1 \leq t \leq p(n)$ $1 \leq k \leq s$	$z_{t,k} = 1$, wenn M zum Zeitpunkt t im Zusatnd k
$a_{t,i,j}$	$1 \leq t, i \leq p(n)$ $1 \leq j \leq v$	$a_{t,i,j} = 1$, wenn A_j Inhalt der i-ten Stelle auf Band zur Zeit t.
$s_{t,i}$	$1 \leq t \leq p(n)$	$s_{t,i}$, wenn M zur Zeit t die i-te Stelle auf Band liest.
$b_{t,l}$	$1 \leq t \leq p(n)$ $1 \leq l \leq p(n)$	$b_{t,l}$, wenn M zur Zeit $t \to t+1$ den l-ten Übergang benutzt.

Jetzt Formeln:

I: Anfangsbedingung: M in Zustand z_1, Kopf an erster Stelle, Bandinhalt: $x A_1^{p(n)-n}$. Sei $x = A_{j_1} A_{j_2} \ldots A_{j_n}$. Dann ist $I = (z_{1,1} \wedge s_{1,1} \wedge a_{1,1,j_1} \wedge a_{1,2,j_2} \ldots \wedge a_{1,n,j_n} \wedge a_{1,n-1,1} \wedge a_{1,p(n),1})$.

B: Randbedingung: zu jedem Zeitpunkt ist M in genau einem Zusatnd, jede Stelle enthält genau ein Symbol, Kopf ist auf genau einer Stelle, es wird genau ein δ-Übergang angewandt.

T: Übergangsbedingung: Konfiguration sind übereinstiimend mit Übergangstabelle.

Seien **I,B,T** Formeln. Dann ist $\alpha(x) = I \wedge B \wedge T \wedge (z_{p(n),r} \vee \ldots \vee z_{p(n),s})$.

B: Genau eine aus einer Menge von Variablen soll wahr sein. Seien $x_1, \ldots, x_n$ diese Variablen. Exact$(x_1, \ldots, x_n)$ =Mindest$(x_1, \ldots, x_n) \wedge$Höchst$(x_1, \ldots, x_n)$ mit:
Mindest$(x_1, \ldots, x_n) = x_1 \vee x_2 \vee \ldots \vee x_n$ und
Höchst$(x_1, \ldots, x_n) = \neg \bigvee_{1 \leq i < j \leq n} (x_i \wedge x_j) = \bigwedge_{1 \leq i < j \leq n} (\bar{x}_i \wedge \bar{x}_j)$

Exact ist in konjunktiver Normalform, n^2 Vorkommen von Variablen.
Exact$(x_1, \ldots, x_n)$ ist erfüllt von Ψ, wenn $\Psi(x_i) = 1$ für genau ein $i, 1 \leq i \leq n$.
$B = \bigwedge_{1 \leq t \leq p(n)} (B_{state}(t) \wedge B_{Bandinh.}(t) \wedge B_{Überg.}(t))$ mit
$B_{State}(t) = Exact(z_{t,1}, z_{t,2}, \ldots, z_{t,s})$
$B_{Posit}(t) = Exact(s_{t,1}, \ldots, s_{t,p(n)})$
$B_{Bandinh}(t) = \bigwedge_{1 \leq i \leq p(n)} Exact(a_{t,i,1}, \ldots, a_{t,i,v})$
$B_{Überg}(t) = Exact(b_{t,1}, \ldots, b_{t,m})$

Formel für **B** hat Größe polynomiell in $p(n), v, s, m$

T: $T = \bigwedge_{1 \le t \le p(n)} T(t)$ T(t) drückt aus, daß Übergang im Zeitpunkt t nach $t+1$ mit Eintrag in Turingtabelle kompatibel ist, der durch $b_{t,1}, \ldots, b_{t,m}$
Sei $z_{k_l}, A_{j_l}, z_{\tilde{k_l}}, A_{\tilde{j_l}}, R_l$ der l-te Eintrag in Tabelle, dabei ist z_{k_l} aktueller Zustand, A_{j_l} wird gerade gelesen, $z_{\tilde{k_l}}$ neuer Zustand, $A_{\tilde{j_l}}$ wird geschrieben, Kopf wird um R_l Einheiten bewegt.

$$
\begin{aligned}
T(t) = \bigwedge_{1 \le i \le p(n)} \{ \bigwedge_{1 \le j \le v} \quad & (\quad s_{t,i} \vee \bar{a_{t,i,j}} \vee a_{t+1,i,j}) \wedge \\
\bigwedge_{1 \le l \le m} [\quad & (\quad \bar{s_{t,i}} \vee \bar{b_{t,l}} \vee z_{t,k_l}) \vee \\
& (\quad s_{t,i} \vee \bar{b_{t,l}} \vee a_{t,i,j_l}) \vee \\
& (\quad \bar{s_{t,i}} \vee \bar{b_{t,l}} \vee z_{t+1,\tilde{k_l}}) \vee \\
& (\quad \bar{s_{t,i}} \vee \bar{b_{t,l}} \vee a_{t+1,i,\tilde{j_l}}) \vee \\
& (\quad s_{t,i} \vee \bar{b_{t,l}} \vee s_{t+1,i+R_l})]\}
\end{aligned}
$$

$(s_{t,i} \vee \bar{a_{t,i,j}} \vee a_{t+1,i,j})$ bedeutet:
Ist $s_{t,i}$ falsch, $a_{t,i,j}$ wahr, dann muß $a_{t+1,i,j}$ wahr sein. Interpretation: Hat M die i-te Stelle des Bandes nicht berührt zum Zeitpunkt t, sollte sich das Symbol dort nicht ändern.
$\bar{s_{t,i}} \vee \bar{b_{t,l}} \vee a_{t,i,j_l}$ bedeutet:
Ist $s_{t,i}$ wahr, $b_{t,l}$ wahr, so muß a_{t,i,j_l} wahr sein, d.h. stehen wir zur Zeit t an der Stelle i und wollen l-ten Übergang anwenden, sollte dort A_{j_l} stehen.
T ist in konjunktiver NF und hat polynomielle Größe.

1. $\alpha(x)$ kann in Polynomzeit aus x konstruiert werden, d.h. haben $x \to \alpha(x)$ Transformation.

2. Ist $x \in L$, so ist $\alpha(x)$ erfüllbar:

   Beweis:

   Ist $x \in L$, so gibt es eine Folge $C_1, C_2, \ldots, C_{p(n)}$ von Konfigurationen, wobei C_1 Anfangskonfiguration, $C_i \vdash C_{i+1}$ für $1 \le i \le p(n)$ und Zustand von $C_{p(n)}$ ist akzeptiert.
   Zuweisung Ψ wird wie folgt gesetzt:

$$
\begin{aligned}
\Psi(z_{t,k}) &= \begin{cases} 1 & \text{falls } z_k \text{ Zustand von } C_t \\ 0 & \text{sonst} \end{cases} \\
\Psi(a_{t,i,j}) &= \begin{cases} 1 & \text{falls } A_j \text{ Symbol auf i-ter Stelle von } C_t \text{ ist} \\ 0 & \text{sonst} \end{cases} \\
\Psi(z_{t,k}) &= \begin{cases} 1 & \text{falls i-te Stelle in } C_j \text{ betrachtet wird} \\ 0 & \text{sonst} \end{cases} \\
\Psi(z_{t,k}) &= \begin{cases} 1 & \text{falls l-ter Eintrag der Turingtabelle beim} \\ & \text{Übergang } C_t \vdash C_{t+1} \text{ benutzt wird} \\ 0 & \text{sonst} \end{cases}
\end{aligned}
$$

   Nach Konstruktion von α ist klar, daß diese Zuweisung Ψ α erfüllt.

3. Ist $\alpha(x)$ erfüllbar, so ist $x \in L$

   Beweis:

   Sei Ψ Zuweisung, die $\alpha(x)$ erfüllt. Es gilt: $\Psi(B) = 1$ und z.B.
   $\Psi(B_{State}(t)) = \Psi(Exact(z_{t,1}, \ldots, z_t)) = 1$ für alle t. Also gibt es für alle t

genau ein k, z.B. k(t), so daß $z_{t,k} = 1$.
Ähnlich gibt es für alle t genau ein i, z.B. i(t), so daß $\Psi(s_{t,i}) = 1$ und genau ein l, z.B. l(t), so daß $\Psi(b_{t,l}) = 1$. Für alle t und i gibt es genau ein j, z.B. $j(t,i)$, so daß $\Psi(a_{t,i,j}) = 1$.
Die true-Variablen definieren die Konfiguration C_t für alle t. $z_k(t)$ ist Zusatnd, i(t) Position des Leseskopfs, I muß erfüllt sein. k(1) = 1, falls i(1) = 1
j(t, 1) = j_1, j(t, 2) = j_2, ...
wobei $x = A_{j_1}, A_{j_2}, \ldots C_1$ ist Anfangskonfiguration von M an Eingabe x.
Außerdem ist $\Psi(z_{p(n),s} \vee z_{p(n),s+1} \vee \ldots \vee z_{p(n),r} = 1$, d.h. $r \leq k(p(n)) \leq s$.
Also ist Zustand der Konfiguration $C_{p(n)}$ akzeptierend. Noch z.z.: $C_t \vdash C_{t+1}$ für $1 \leq t \leq p(n)$. Ähnlich wie oben durch Betrachtung von T
$\rightarrow$ l(t)-ter Eintrag kann auf C_t angewandt werden und ergibt C_{t+1}
$\Rightarrow$ es gibt akzeptierende Berechnung an x, d.h. $x \in L$
$\Rightarrow$ L$\leq$SAT, d.h. SAT NP-vollständig. ❏

Satz 5.4: 3SAT ist NP-vollständig

Eingabe: Formel α in konjunktiver NF mit ≤ 3 Literalen pro Klausel.
Frage: Ist α erfüllbar?

1. 3SAT$\in$NP. Klar
2. zeigen: SAT$\leq$3SAT
   Wir ersetzen lange Klauseln durch kurze. Sei $x_1 \vee \ldots \vee x_n$ eine Klausel.
   Mit neuen Variablen $y_1, \ldots, y_n$ betrachte $\alpha = (x_1 \vee \bar{y_1} \wedge (y_1 \vee x_2 \vee \bar{y_2}) \wedge \ldots \wedge (y_{n-1} \vee x_n \vee \bar{y_n}) \wedge y_n$

Behauptung: Sei $\Psi : \{x_1, \ldots, x_n\} \rightarrow \{0,1\}$ eine erfüllbare Zuweisung. Dann gibt es Erweiterung $\Psi' : X \cup Y \rightarrow \{0,1\}$ mit $\Psi'(\alpha) = 1 \Leftrightarrow \Psi(x_1 \vee \ldots \vee x_n) = 1$.

Beweis:
$\Leftarrow$: Sei $\Psi(x_1, \ldots, x_n) = 1$. Sei i_0 der kleinste Index i, so daß $\Psi(x_i) = 1$. Dann

$$\Psi'(z) = \begin{cases} \Psi(x_j) & \text{wenn } z = x_j \text{ für ein } j \\ 0 & \text{wenn } z = y_j \text{ für } j < i_0 \\ 1 & \text{sonst} \end{cases}$$

Dann ist $\Psi'(\alpha) = 1$.

$\Rightarrow$: Sei $\Psi' : X \cup Y \rightarrow \{0,1\}$ erfüllend, z.z.: $\Psi'(x_1 \vee \ldots \vee x_n) = 1$
Nehmen an: $\Psi'(x_i) = 0$ für alle i
wir zeigen: $\Psi'(y_i) = 0$ für alle i
$\Psi'(x_1 \vee \bar{y_1}) = 1 \Rightarrow \Psi'(y_1) = 0$
$\Psi'(y_1 \vee x_2 \vee \bar{y_2} = 1 \Rightarrow \Psi'(y_2) = 0$
$\vdots$
$\Psi'(y_{n-1} \vee x_n \vee \bar{y_n}) = 1 \Rightarrow \Psi'(y_n) = 0 \Rightarrow \Psi'(\alpha) = 0$
Widerspruch

Ersetze jede Klausel wie oben, durch mehrere Klauseln der Länge$\leq$3 ❏

Beachte: 2SAT ist effizient lösbar.

Satz 5.5: $(0,1)$-LP ist NP-vollständig (IP)

Eingabe: Ganzzahlige Matrix A, Vektor b.
Frage: Gibt es einen (0,1)-Vektor c, so daß $Ac \geq b$?

Beweis: IP∈NP

Raten c nichtdeterministisch und prüfen $Ac \geq b$. Jetzt: SAT≤IP.
Sei $\alpha = z_1 \wedge \ldots \wedge z_k$ Formel in konjunktiver NF und seien $x_1, \ldots, x_n$ die vorkommenden Variablen
Konstruieren $A = (a_{ij})_{\substack{1 \leq i \leq k \\ 1 \leq j \leq n}}$, $b = (b_i)_{1 \leq i \leq k}$

$$a_{ij} = \begin{cases} 1 & \text{falls } x_j \text{ in } z_i \text{ vorkommt} \\ -1 & \text{falls } \bar{x_j} \text{ in } z_i \text{ vorkommt} \\ 0 & \text{sonst} \end{cases}$$

Behauptung: α erfüllbar $\Leftrightarrow$ es gibt (0,1)-Vektor c mit $Ac \geq b$.

Beweis:

$\Rightarrow$: Sei Ψ eine Zuweisung, so daß $\Psi(\alpha) = 1$.
Wähle $c_j = \Psi(x_j)$ für $1 \leq j \leq n$.
Dann ist
$(Ac)_i = \sum_{j=1}^{n} a_{ij} c_j = \sum_{x_j \in z_i} \Psi(x_j) - \sum_{x_j \bar{\in} z_i} \Psi(x_j) \geq 1 - \sum_{\bar{x_j} \in z_i} 1 = b_i$
d.h. $\Psi(x_j) = 0$.
$\Leftarrow$: Sei c ein (0,1)-Vektor mit $Ac \geq b$
Sei $\Psi(x_j) = c_j$.
Beh.: $\Psi(\alpha) = 1$
Beweis:
Nimm an, es gibt Klausel z, mit $\Psi(z_i) = 0$,
d.h. $\Psi(x_j) = c_j = 0$ für $x_j \in z_i$
$(Ac)_i = \sum_{x_j \in z_i} c_j - \sum_{\bar{x_j} \in z_i} c_j = - \sum_{x_j \in z_i} < b_i$
Widerspruch

5.1 Weitere Komplexitätsklassen

Sei $f : \mathbb{N} \to \mathbb{N}$ eine Funktion. DTIME(f) (NTIME(f), DSPACE(f)) bezeichne Menge der Sprache L, so daß es deterministische (nichtdeterministische, deterministische) Turingmaschine M gibt, die L akzeptiert und deren Laufzeit (Laufzeit, Platz) durch f beschränkt ist. Weitere Bezeichnungen:
P= $\bigcup_{i \geq 1}$ DTIME(n^i)
NP= $\bigcup_{i \geq 1}$ NTIME(n^i)
PSPACE= $\bigcup_{i \geq 1}$ DSPACE(n^i)
EXPTIME= $\bigcup_{i \geq 1}$ DTIME(2^{n^i})
EXPSPACE= $\bigcup_{i \geq 1}$ DSPACE(2^{n^i})

Beispiel 5.3: QBF (quantifizierte boolsche Formel)

Variablen $x_1, \ldots, x_n$
$Q_1 x_1 Q_2 x_2 \ldots Q_n x_n : F(x_1, \ldots, x_n)$, $Q_i \in \{\exists, \forall\}$
QBF∈PSPACE

EXPSPACE: Äquivalenzproblem für reguläre Ausdrücke mit den Operatoren $\{+, \cdot, ^2, ^*\}$

Satz 5.6: P⊆NP⊆PSPACE⊆EXPTIME⊆EXPSPACE

Beweis:

klar: P⊆NP, EXPTIME⊆EXPSPACE

NP⊆PSPACE: Sei L∈NP.
L wird in Zeit$\leq n^i$ von NTM M akzeptiert. M habe o.B.d.A. in jedem Schritt Wahl zwischen zwei möglichen Zügen. Sei M' eine DTM, so daß

1. Eingabe von M' ist (a, α), wobei x Eingabe von $M, \alpha \in \{0,1\}^{n^i}$, wobei $n = |x|$. M' prüft, ob M bei Zugauswahl α an der Eingabe x nach n^i Schritten anhält.
2. M' läuft in Zeit/Platz $O(n^i)$.

M'' sei eine aus M' konstruierte DTM, die L in Platz $O(n^i)$ akzeptiert. M'' hat ein Band mehr als M'. Schreibe auf Band 2 $0^{n^i} = n^i$ mal „0". M'' simuliert M' an (x, α) und testet Aktzeptanz. Falls nicht, erhöhe α um 1, teste wieder. Abbruch bei Akzeptanz oder $\alpha = 10^{n^i}$. ❏

PSPACE⊆EXPTIME: Sei L∈PSPACE
DTM M akzeptiere L in Platz n^i. Sei a = Anzahl der Zustände in M, b = Größe der Alphabets und k = Anzahl der Bänder.
Dann gibt es höchstens $a \cdot n^{ik} \cdot b^{n^i}$ verschiedene Konfigurationen für M bei Eingabelänge n.
n^{ik} = # mögl. Kopfpositionen
b^{n^i} = # mögl. Platzbelegungen
M braucht also $\leq a \cdot n^{ik} \cdot b^{n^i}$ Schritte zum akzeptieren, ohne in Zykel zu geraten. Wir nehmen DTM M', die M an Eingabe x der Länge n simuliert für $\leq a \cdot n^{ik} \cdot b^{n^i}$ Schritte. Ist x bis dahin akzeptiert, so akzeptiert M' x, andernfalls lehnt M' x ab. M' akzeptiert L in Zeit $O(a \cdot n^{ik} \cdot b^{n^i}) \leq 2^{n^j}$ für geeignetes j. ❏

Bemerkung: Man weiß nicht, ob einzelne ⊆ bedeuten = oder ⊂, aber man kann zeigen:

1. P≠EXPTIME und
2. PSPACE≠EXPSPACE

Beweis von 2.:

Funktion $f : \mathbb{N} \to \mathbb{N}$ heißt Platzzählfunktion, falles es eine DTM gibt, die für jede Eingabe x hält und genau f($|x|$) Platz braucht.

Beispiel: $n^i, 2^{n^i}, i \geq 1$ ❏

Lemma 5.1:

Sei f Platzzählfunktion. Sei M eine DTM, die an Eingabe der Länge $n \leq f(n)$ Platz braucht. Danach gibt es DTM M', die in $\leq f(n)$ Platz die gleiche Sprache wie M akzeptiert und an jeder Eingabe hält.

Satz 5.7:

Seien f und F Platzzählfunktionen, mit

$$\lim_{n \to \infty} \frac{f(n)}{F(n)} = 0$$

Dann gilt: DSPACE(f) ⊂DSPACE(F).

Beweis: Nur $\neq$

DTM's können als Strings über $\{0,1\}^*$ codiert werden. Dann gibt es eine DTM, die überprüft, ob ein gegebener String eine DTM codiert und gegebenfalls dir TM konstruiert.
Sei M_F eine DTM, die für jede Eingabe w terminiert und genau F($|w|$) Platz braucht. M_F existiert.

Idee: konstruiere TM M, die in Platz F(n) eine best. Teilmenge aller TM konstruiert. Wir zeigen, daß die von M akzept. Sprache nicht von DTM mit f(n) Platz akzept. werden kann.
M: sei $w \in \{0,1\}^*$ Eingabe der Länge n.

1. benutze M_F, um F(n) Zellen zu reservieren.
2. Bestimme j mit $w = 1^j 0 u$, $u \in \{0,1\}^*$.
   Existiert j nicht, weist M w zurück und terminiert.
3. M testet, ob u DTM codiert.
   Wenn nicht, oder M will reservierten Platz verlassen, weist M w zurück und terminiert.
4. Konstruiere DTM M_w (durch u kodiert), simuliere M_w an Eingabe w.
   Verläßt Simulation reservierten Platz nicht und terminiert sie in einem nichtakzeptierenden Zustand von M_w
   `then` M akzeptiert w
   `else` M weist w zurück

Sei L Sprache, die M akzeptiert. Klar: $L \in$DSPACE(F). Z.z.: $L \notin$ DSPACE(f)
Annahme: $L \in$DSPACE(f). Sei M' eine DTM, die L mit Platz f($|w|$) an Eingabe w akzeptiert. M' terminiere immer. Wir konstruieren w so, daß M akzeptiert $w \Leftrightarrow M'$ weist w zurück $\Rightarrow$ Widerspruch.

Index

Symbols
(a,b)-Baum 35

B
Basislösung 50
    zulässige 50
Baum 1
    binomisch 7

D
Dijkstra-Algorithmus 6

E
Entfernung 3

F
Fibunacci-Heaps 6

G
Graph 1
    azyklisch 1
    gerichtet 1
    zyklisch 1

K
Kante 1
    Baumkante 11
    Querkante 11
    Rückwärtskante 11
    Vorwärtskante 11

P
Pfad 1

S
Sortierung
    topologisch 1, 2

T
TopSort 2