# Algorithmen & Datenstrukturen

Martin Steffen

Wintersemester 2000/2001

# Lektion I

# Einleitung

**Literatur:** Baut auf Kapitel 1 von [CLR90] auf.

**Inhalt:** Algorithmus · Analyse von Algorithmen · Zeitkomplexität und Speicherkomplexität · Algorithmenentwurf

# Aufbau

- Einleitung und Überblick
- Sortieren
- Suchen
- dynamische Datenstrukturen
- Bäume, Graphen . . .
- Problemlösungsstragegien:
  - Divide and Conquer
  - Rekursion und Iteration
  - Greedy-Algorithmen
  - Relaxation
  - . . .

# Algorithmus

- Namensgeber: Abu Ja'far Mohammed ibn Mûsâ al-Kwoarizmî: *Kitab al jabr w'al-muqabala* (Regeln zur Wiederherstellung und Reduktion)
- heutige (informelle) Bedeutung ([Knu73a])
  - (endliche Beschreibung)
  - Definiertheit
  - Terminierung
  - besitzt Input[1] und Output
  - Effektivität/*effectivness*, jeder Einzelschritt ist effektiv ausführbar
  - zur Lösung eines wohlspezifizieren Berechnungsproblems

[1] eventuell leer

# Berechnungsproblem: Sortieren

- Eingabe: endliche Sequenz von $s = (a_1, a_2, \ldots , a_n)$
- Ausgabe: Permutation $(a'_1, \ldots a'_n)$ von $s$ mit $a'_i \leq a'_{i+i}$ für alle $i \in \{1, \ldots n\}$
- Algorithmus korrekt/löst das Problem, falls für alle Eingaben
  - Algorithmus hält nach endlicher Zeit
  - liefert das korrekte Ergebnis

# Beispiel: Insertion-Sort

- Sortieren durch Einfügen
- Imperative Lösung:
  - Eingabe: Array $A$ fester Länge $n$ von Zahlen
  - Ausgabe: monoton-steigend sortierter Array, Permutation von $A$.
  - direktes Einfügen ("in situ" sortieren)
- inkrementelle Lösung
- Bild: Verhalten von Insertion sort

```
Insertion_sort (A)

  for j = 2 to length[A]
    do key := A[j];
          i := j-1;
       while i > 0 and A[i] > key
          do
                A[i+1] := A[i]
                     i := i-1;

       A[i+1] := key;
```

# Analyse von Algorithmen

- Analyse: mathematische Abschätzung des (vorraussichtlichen) Resourcenverbrauches abhängig von der Größe der Eingabe

- Fragestellungen:
  - (Korrektheit, Terminierung)
  - Zeitkomplexität: Anzahl elementarer Schritte
  - Speicherkomplexität:
  - Sonstiges (Kommunikationsbandbreite, . . . )

- Idealisierungen:
  - abstraktes Maschinenmodell (Turingmaschine, RAM-Maschine)
  - asymptotisches Verhalten ($O$-Notation).
    - Abschätzung nach oben: worst-case
    - Abschätzung nach unten: best-case
    - mittlere Komplexität: average-case

- Untersuchung/Vergleich/Klassifizierung der Komplexität von Algorithmen: Komplexitätstheorie (s. [Pap94], auch [AHU89])

# Analyse des Sortierens durch Einfügen

- Abhängig vom Input
- sei $t_j$ die Anzahl der *while*-tests für den Wert von $j$

```
Insertion_sort (A)

for  j = 2 to length[A]                    // c1    n
  do key := A[j];                          // c2    n-1
       i := j-1;                           // c4    n-1
     while  i > 0 and A[i] > key           // c5    sum(j=2 to n) t_j
        do
              A[i+1] := A[i]               // c6    sum(j=2 to n) (t_j-1)
                    i := i-1;              // c7          -"-
        od
      A[i+1] := key;                       // c8    n-1
```

# Analyse (2)

```
Insertion_sort (A)

for  j  = 2  to  length [A]                 // c1      n
  do  key  :=  A[ j ];                      // c2      n-1
        i  :=   j -1;                       // c4      n-1
      while   i  > 0  and  A[ i ] >  key    // c5      sum(j=2 to n) t_j
         do
               A[ i +1]  :=  A[ i ]         // c6      sum(j=2 to n) ( t_j -1)
                     i  :=  i -1;           // c7            -"-
         od
       A[ i +1]  :=  key ;                  // c8      n-1
```

$$
\begin{aligned}
T(n) &= c_1 n + (c_2 + c_4 + c_8)(n-1) \\
&\quad c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1)
\end{aligned}
$$

# Analyse (3)

- Zusammenfassen aller Konstanten $c_i$:

| | | $t_j$ | $T(n)$ |
|---|---|---|---|
| bester Fall | sortiert | 1 | $an + b$ |
| schlechtester Fall | rückwärts sortiert | j | $an^2 + bn + c$ |

- für große $n$: nur der höchste Exponent des Polynoms zählt:
  - Bester Fall: lineare Zeitkomplexität ($O(n)$)
  - schlechtester Fall quadratische Zeitkomplexität ($O(n^2)$)

- Komplexität darstellbar durch ein Polynom über der Problemgröße: polynomielle Komplexität ($O(n^c)$)

# Algorithmenentwurf

- Inkrementell: vgl. Sortieren durch Einfügen.
- Divide-and-Conquer: rekursive Zerlegung des Problems:
  1. Divide: Zerlegen des Problem in (gleichartige, aber kleinere) Unterprobleme
  2. Löse die Teilprobleme rekursiv
  3. Conquer: Zusammenfügen zur Gesamtlösung.

# D&C: Mergesort

- klassisches Beispiel für D&C
  1. Divide: Zerlege die Sequenz in zwei Hälften
  2. Sortiere sie rekursiv
  3. Conquer: Verschmelzen ("*merge*") der sortierten Sequenzen
- Aufruf: $Merge\_Sort(A, 1, length(A))$

# Mergesort (code)

```
Merge_Sort(A, p, r)

  if p < r
    then   q := floor(p+r/2);    // the ''middle''
        Merge_Sort(A, p, q);
        Merge_Sort(A, q+1, r);
        Merge(A, p, q, r);
```

# Analyse von Merge-Sort

- rekursiver Algorithmus $\Rightarrow$ Komplexität durch Rekurrenzgleichung
- im schlimmsten Fall

$$T(n) = \begin{cases} O(1) & \text{falls } n = 1 \\ 2T(n/2) + O(n) & \text{falls } n > 1 \end{cases}$$

- Vereinfachung: Länge $= 2^n$
- Ansatz:

$$T(n) = \begin{cases} 0 & \text{falls } n = 1 \\ 2T(n/2) + n & \text{falls } n = 2^k, k \geq 1 \end{cases}$$

$\Rightarrow$ $T(n) = n \log_2 n$

24. Oktober 2000 (1 Wochen)

# Lektion II

# Sortieren

**Literatur:** Kapitel 7 und 8 aus [CLR90]. Knuth [Knu73b] enthält mehr über Sortieren als man wissen will. Heapsort stammt von Williams [Wil64], Quicksort von Tony Hoare [Hoa62]

**Inhalt:** Heaps · Heapsort · Quicksort

# Heap-Datenstruktur

- (gerichteter) Graph $G = (V, E)$:
  - $V$: Menge von Knoten ($vertices$),
  - $E$: Kanten, binäre Relation auf $V$ ($edges$)
- ungerichteter Graph: $E$: Menge ungeordneter Paare $\{u, v\}$ von Knoten mit $u \neq v$ (keine self-loops)
- Wurzelbaum: (rooted tree): verbundener, azyklischer, ungerichteter Graph mit einem ausgezeichnetem Knoten = Wurzel
- Heap: voller, geordneter, binärer Wurzelbaum.
  - Binär: Anzahl der Kinder $\leq 2$.
  - voll: entweder Blatt oder genau zwei geordnete Kinder

  - geordnet: Reihenfolge der Kinder spielt eine Rolle, d.h. hier man kann linkes und rechtes Kind unterscheiden.[2]

- $+$ Heap-Eigenschaft: Für alle Teilbäume $t$ gilt: $t = Node(l, a, r) \Rightarrow l = Leaf$ (und $r = Leaf$) oder $a \geq Key(l)$ und $a \geq Key(r)$

- Bild: Heap als Baum

[2] `('a Btree = Leaf | Node of ('a Btree * 'a * 'a Btree)`

# Implementierung: Heaps als Array

- Implementierung von Heaps als Array: Knoten als Arrayelemente

⇒ weitere Zusatzbedingung: alle Ebenen des Baums gefüllt bis auf evtl. die letzte, diese kann von links nach rechts teilgefüllt sein.[3]

- Füllung des Arrays von 1 bis *heapsize* (Bild)

- Repräsentierung des Binärbaumes:

$$\begin{aligned} Parent(i) &= \lfloor \tfrac{i}{2} \rfloor \\ Left(i) &= 2i \\ Right(i) &= 2i+1 \end{aligned}$$

- Ein weiteres Mal das Bild: Heap + Implementierung

[3] Länge des Arrays $l = 2^n - 1$ mit $2^n - 1 \geq heapsize$ für minimales $n$.

# Heapsort: Aufbau

- Ausgangspunkt: Unsortierter Array
- zwei Schritte:
  1. „Teilsortierung“: Bilden eines Heaps
  2. inkrementelle Sortierung des Heaps

unsortiert $\longrightarrow$ Heap $\longrightarrow$ sortiert

# Einfügen in den Heap

- *Heapify*: Einfügen, Erhalt der Heapbedingung
- Annahme: der rechte und der linke Teilbaum erfüllen die Heapbedingung!
- rekursive Prozedur: “Einsickern” des neuen Elementes

```
Heapify (A, i)
  l := Left(i); r := Right(i);   // Left(i), Right(i) are heaps

  if l <= heapsize(A) and A[l] > A[i]
     then largest := l  else largest := i;
  if r <= heapsize(A) and  A[r] > A[largest]
     then largest := r;

  if largest != i                    // Verletzung der Heapbedingung?
     then exchange (A[i] , A[largest]);
          Heapify(A, largest);
```

# Aufbau des Heaps

- repariere alle Verletzungen der Heapbedingung mit `Heapify`
- Bottom-up mittels `Heapify`
- Blätter erfüllen Heap-Eigenschaft $\Rightarrow$ Start bei den untersten nicht-Blatt Knoten ($\lfloor length(A)/2 \rfloor$).

```
Build-Heap(A)
  heapsize[A]   := length[A];
  for  i= floor(length(A)/2))  downto 1
      do
          heapify(A,i);
```

# Heapsort

- Heap als unsortiertes, aber teilsortiertes Reservoir, der Rest des Arrays ist bereits sortiert
- Wurzel des Heaps: Maximum

$\Rightarrow$ Aufbau der sortierten Elemente von hinten

- Entfernen aus dem Heap ist billig (`Heapify` $\log_2(n)$)

Heapsort (A)

```
build_heap (A);
for  i  =  length (A)  downto  2
do   exchange  A[1]  A[ i ];
     heapsize [A]  :=  heapsize [A]  −  1;
     Heapify (A,1);
```

# Quicksort

- Divide & Conquer
- im Gegensatz zu Mergesort: Verschmelzen trivial, investiere dafür mehr in das Divide
- D&C-Schritte:

  1. Divide: Teile die Sequenz $\sigma$ in zwei Teilsequenzen $\sigma_1$ und $\sigma_2$, sodaß $\sigma_1 \leq \sigma_2$ (punktweise)
  2. Sortiere Teilsequenzen $\sigma_1$ und $\sigma_2$
  3. Conquer: Zusammenfügen trivial

# Quicksort (2)

---

```
(* Id : quicksort.ml,v 1.6 1999/11/11 10:43:28 msExp *)

open Basic;;
open Sort;;

module QuicksortFun (E: ORDERED) : (SORT with type elem = E.t) =
  struct
    type elem = E.t

    let rec filter p l = match l with
      []         -> []
    | x :: tl ->
        if    (p x)
        then  x::(filter p tl)
        else  filter p tl

    let rec sort (l : elem list) =   match l with
```

```
      []    -> []
    | x :: tl   ->
        sort (filter (function y -> E.lt(y,x)) tl)
        @ [x] @                                          (* Pivot *)
        sort (filter (function y -> E.geq(y,x)) tl)
  end
```

# Quicksort mit arrays (1)

- Zwei rekursive Aufrufe
- das Divide: `Partition` liefert den Index für die Trennung in die kleinere und größere Hälfte

```
Quicksort(A,p,r)
if  p < r
    then   q := Partition(A,p,r)
           Quicksort(A,p,q);
           Quicksort(A,q+1,r);
```

# Quicksort mit arrays: Partitionieren

- Wähle ein Element des Arrays: das Pivot $x$
- Partitionieren: trenne/ordne $A$ in 2 Hälften, links alle $\leq x$, rechts alle $\geq x$.
- Invariante:

$$\begin{array}{lll} \mathbf{I_1} & A[i'] \leq x \leq A[j'] & \text{für alle } i' < i,\ j' > j. \\ \mathbf{I_2} & A[i'] \leq x \leq A[j'] & \text{für alle } i' \leq i,\ j' \geq j. \end{array}$$

- Abbruch: $(i \geq j) \wedge \mathbf{I_1}$, d.h., $A[p \ldots i-1] \leq x \leq A[j+1 \ldots r]$, daraus folgt $A[p \ldots j] \leq x \leq A[j+1 \ldots r]$

```
Partition (A, p, r)                             /* p <= r              */
  x := A[p];
  i := p-1;
  j := r+1;

  while true                                    /* Invariante I_2 */
  do      repeat j := j-1
            until A[j] <= x;
          repeat i := i+1
            until A[i] >= x;
          if    i < j                           /* Invariante I_1 */
          then exchange A[i] <-> A[j]           /* Invariante I_2 */
          else return   j;                      /* j <= i              */
  od;
```

# Analyse von Quicksort: worst case

- Schlechtester Fall: Partitionierung trennt den Ausschnitt $[p \ldots r]$ der Länge $n = r - p + 1$ in einen Teile der Größe $1$ und $n - 1$.
- Kosten der Partitionierung: $O(n)$
- Rekurrenz: $T(n) = T(n-1) + O(n) \Rightarrow T(n) = \sum_{k=1}^{n} O(k) = O(n^2)$
- Schlimmster Fall: Falls $A$ bereits sortiert

# Analyse von Quicksort: best case

31. Oktober 2000 (2 Wochen)

# Lektion III

# Lineares Sortieren

**Literatur:** Kapitel 9 [CLR90].

**Inhalt:** $O$-Notation · Einführung · Vergleichssortieren · Counting sort · Bucket sort

# Einführung

- Bisher: die Algorithmen $O(n \log n)$ oder $O(n^2)$
- Gemeinsamkeit:

  Die Ordnung die die Algorithem bestimmen, beruht *ausschließlich* auf dem *Vergleich* zwischen Elementen

$\Rightarrow$ $\boxed{\text{Vergleichssortieren}}$

# Sortieren: Überblick

| Verfahren | $T_{min}$ | $T_{max}$ | $T_{av}$ |
|---:|:---:|:---:|:---:|
| Auswahl | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Einfügen | $O(n)$ | $O(n^2)$ | $O(n^2)$ |
| Bubblesort | $O(n)$ | $O(n^2)$ | $O(n^2)$ |
| Mischen | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ |
| direktes Mischen | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ |
| natürliches Mischen | $O(n)$ | $O(n \log n)$ | $O(n \log n)$ |
| Quicksort | $O(n \log n)$ | $O(n^2)$ | $O(n \log n)$ |
| Heapsort | $O(n \log n)$ | | |

Tabelle 1: Vergleich

# Abschätzung für Vergleichssortieren

- Vorraussetzung: Sortieren einer Sequenz $(a_1, \ldots a_n)$ (oBdA: alle Elemente verschieden, wichtig ist nur $<$ oder das Gegenteil.)

- mögliche Eingaben der Länge $n$: alle Permutationen

- Entscheidungsbaum: Darstellung der Entscheidungen eines Sortieralgorithmusses
  - interner Knoten: $a_i : a_j$ wobei $1 \leq i < j \leq n$, repräsentiert den Vergleich von $a_i$ mit $a_j$, seine Unterbäume die Vergleiche, je nachdem wie sein Vergleich ausgefallen ist.
  - Blatt: Permutation
  - Lauf der Sortierung: Pfad durch den Baum

$\Rightarrow$ Höhe des Baumes $\cong$ worst-case der Vergleiche $\cong$ worst-case der Laufzeit

- Folie: *insertion sort*

# Vergleichssortieren: worst-case

- gegeben: Entscheidungsbaum für einen Algorithmus
- $\boxed{\text{worst-case} = \text{Tiefe des Baumes}}$

$\Rightarrow$ Abschätzung für die Laufzeit:

**Theorem 1 (Untere Schranke für worst-case)** Die Höhe ein Entscheidungsbaums der $n$ Elemente sortiert besitzt eine asymptotische *untere* Schranke der Größe $O(n \lg n)$.

- Mischsortierung und Heapsort sind asymptotisch optimal.

# Abschätzung der Laufzeit(2)

**Beweis von 1:**

- Es gibt $\geq n!$ Blätter
- Eigenschaft eines binären Baumes: $n! \leq n^h$, d.h., $h \geq \lg(n!)$
- mit Stirling-Approximation $n! > \left(\frac{n}{e}\right)^n$

$\Rightarrow$ $h \geq n \lg n - n \lg e$

$\Rightarrow$ Untere Schranke $O(n \lg n)$.

$\square$

# Lineares Sortieren: Counting Sort

- Beispiel für einen Algorithmus, der kein Vergleichssortieren ist
- man braucht zusätzliche Information
- Counting Sort: Information, daß der Wertebereich der Elemente endlich ist: $1 \ldots, k$ mit $k = O(n)$.

$\Rightarrow$ Zählen der Elemente $= i$ möglich (im Hilfsarray $C[1 \ldots k]$)

- kein *Vergleich* im Algorithmus

# Counting Sort

```
Counting-Sort(A,B,k)
  for i := 1 to k do C[i] := 0;

  for j := 1 to length[A]
  do  C[A[j]] := C[A[j]] + 1   /* C[i]: Anzahl der Elemente = i */

  for i := 2 to k              /* Erlaubter Input von 1...k      */
  do  C[i] := C[i] + C[i-1];   /* C[i]: Anzahl Elemente <= i     */

  for j := length[A]  downto  1
  do
        B[C[A[j]]] :=    A[j];  /* C[A[j]] gibt Pos. in B         */
          C[A[j]]   := C[A[j]] - 1;
  od;
```

# Analyse von Counting Sort

1. erste Schleife: $O(k)$
2. zweite Schleife:$O(n)$
3. dritte Schleife: $O(n)$

- Insgesamt: $O(k + n)$, und falls $k = O(n) \Rightarrow$ lineare Laufzeit: $O(n)$

# Bucketsort

- Annahme von Bucketsort: uniforme Verteilung der Eingabe im Intervall $I = [0, 1[$
- Teilung von $I$ in n Unterintervalle (Buckets)

1. Verteilen der Elemente in die Buckets
2. Sortieren der Buckets
3. Aneinanderhängen der sortierten Teillisten

# Bucketsort (2)

```
BucketSort(A)
  n := length(A);

  for i=1 to n
    do insert A[i] into list B[floor (n * A[i])];

  for i=0 to n-1
    do sort list B[i] with insertion sort;

  concatenate lists B[0], B[1], B[n-1] together;
```

# Bucketsort: Analyse

- Alles linear bis auf Insertionsort = quadratisch
- Zufallsvariable $n_i$: Anzahl der Elemente in $B[i]$.
- Insertionsort quadratisch $\Rightarrow$ erwarteter Gesamtaufwand:

$$\sum_{i=0}^{n-1} O(\mathcal{E}(n_i^2)) = O\left(\sum_{i=0}^{n-1} (\mathcal{E}(n_i^2))\right)$$

- Verteilung von $n_i$: Binomialverteilung $B(k, n, p = 1/n)$

$$P(n_i = k) = \binom{n}{k} p^k q^{n-k}$$

- $\mathcal{E}(n_i) = 1$ und $Var(n_i) = 1 - \frac{1}{n}$.

- $\mathcal{E}(n_i^2) = Var(n_i) + \mathcal{E}(n_i)^2 = 2 - \frac{1}{n} = O(1)$

$\Rightarrow$

$$T(n) = O(n)$$

# Lektion IV

## Analyse

**Literatur:** Verschiedene Kapitel aus [CLR90]. Daneben Abschnitt 2.5 aus [Heu97].

**Inhalt:** Analyse von Divide and Conquer · lineare, $O(n \log n)$, polynomielle Komplexität

# Laufzeitanalyse für Divide & Conquer

- Aufstellen einer Rekurrenzgleichung:

$$\begin{aligned} T(n) &= O(1) \quad \text{für } n \leq n_0 \\ T(n) &= \sum_{i=1}^{a} T(n_i) + D(n) + C(n) \quad \text{für } n > n_0 \end{aligned}$$

- $T(n)$: Laufzeit für Problem der Größe $n$, $D$ und $C$ Kosten für das Divide resp. Conquer

- Vereinfachung: Teilprobleme gleichgroß, $n_0 = 1$. ferner $f(n) := D(n) + C(n)$.

$\Rightarrow$ $\boxed{T(n) = a\,T(n/b) + f(n)}$ $a \geq 1$, $b > 1$ konstant

- Beispiel: Merge-Sort:
  - Zwei rekursive Aufrufe $a = 2$
  - Halbierung der Problemgröße: $b = 2$
  - Kosten des Mergens: $O(n)$.

# Lösen der Rekurrenzgleichung

Sei $n = b^k$. Auflösen durch Iteration.

$$\begin{aligned}
T(n) &= aT(\frac{n}{b}) + f(n) \\
&= a^2T(\frac{n}{b^2}) + af(\frac{n}{b}) + f(n) \\
\ldots \\
&= a^kT(\frac{n}{b^k}) + \sum_{i=0}^{k-1} a^i f(\frac{n}{b^i}) \\
&= a^{\log_b n}\, T\left(\frac{n}{b^{\log_b n}}\right) + \sum_{i=0}^{(\log_b n)-1} a^i f(\frac{n}{b^i}) \\
&= a^{\log_b n}\, T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(\frac{n}{b^i}) \\
&= d\, n^{\log_b a} + \sum_{i=0}^{(\log_b n)-1} a^i f(\frac{n}{b^i})
\end{aligned}$$

- Sei $f(n) = D(n) + C(n)$ linear, $f(n) = cn$.

# Spezialfall: lineare Zeitkomplexität

- Summe der Größen der Teilprobleme $(a\frac{n}{b})$ echt kleiner

$\Rightarrow a < b$

$$
\begin{aligned}
T(n) &= d\, n^{\log_b a} + \sum_{i=0}^{(\log_b n)-1} a^i f(\frac{n}{b^i}) \\
&= d\, n^{\log_b a} + \sum_{i=0}^{(\log_b n)-1} ca^i \frac{n}{b^i} \\
&= d\, n^{\log_b a} + cn \sum_{i=0}^{(\log_b n)-1} \left(\frac{a}{b}\right)^i \\
&\leq O(n)
\end{aligned}
$$

# Spezialfall: $O(n \log n)$

- Summe der Größen der Teilpobleme $=$ Größe des Ausgangsproblems:$\Rightarrow$ $a = b$

$$
\begin{aligned}
T(n) &= d\, n^{\log_b a} + \sum_{i=0}^{(\log_b n)-1} ca^i \frac{n}{b^i} \\
&= d\, n^{\log_b a} + cn \sum_{i=0}^{(\log_b n)-1} \left(\frac{a}{b}\right)^i \\
&= d\, n + cn \sum_{i=0}^{(\log_b n)-1} 1 \\
&\leq O(n \log n)
\end{aligned}
$$

- Beispiel: Mergesort

# Spezialfall: polynomiale Laufzeit

- Summe der Größen der Teilpobleme größer als das Ausgangsproblem $\Rightarrow a > b$

$$
\begin{aligned}
T(n) &= d\, n^{\log_b a} + \sum_{i=0}^{(\log_b n)-1} c \left(\frac{a}{b}\right)^i \\
&\leq d\, n^{\log_b a} + cn \frac{\left(\frac{a}{b}\right)^{\log_b n} - 1}{\frac{a}{b} - 1} \\
&\leq O\left(n^{\log_b a} + n \left(\frac{a}{b}\right)^{\log_b n}\right) \\
&\leq O\left(n^{\log_b a} + n \frac{a^{\log_b n}}{b^{\log_b n}}\right) \\
&\leq O\left(n^{\log_b a}\right)
\end{aligned}
$$

# Lektion V

# Elementare Datenstrukturen

**Literatur:** Kapitel 11 [CLR90].

**Inhalt:** Stacks und Queues · verzeigerte Listen · Bäume · Implementierungsmöglichkeiten

# Einleitung

- „dynamische Mengen“: veränderbar
- viele verschiedene Varianten
- Unterschieden nach
  - Zugriffsmöglichkeiten (Schnittstelle)
  - Repräsentierung
- zwei Arten von Zugriffen
  - modifizend
  - lesend ($queries$)
- Beispiele für Zugriffe:
  - Suchen

- Einfügen, Löschen
- Minumium, Maximum bestimmen
- Vorläufer, Nachfolger bestimmen
- ...

- Beispiele für Repräsentierungen:
  - (einfach o. doppelt) verzeigerte Listen
  - Bäume unterschiedlichster Art (balanciert, Heaps ... )
  - Hashstrukturen

# Stacks und Queues

- Stack/Queue: dynamische (Multi-)Mengen mit spezifischer Strategie zum Einfügen und Löschen:
  - Stack: Lifo: *last-in, first-out*
    - *Push*: Einfügen
    - *Pop*: Entfernen
  - Queue: Fifo: *first-in, first-out*
    - *Enqueue*: Einfügen
    - *Dequeue*: Entfernen

# Stack als Array

- Implementiert als Array $1 \ldots n$
- dynamischer Inhalt: Elemente $S[1], \ldots S[top]$
- Komplexität: $O(1)$

```
Stack-empty(S)
  if top == 0 then return true else false;

Push(S,x)
  top    := top + 1;
  S[top] := x;

Pop(S)
  if Stack-empty(S)
  then error "underflow"
  else top  := top  -1;
       return S[top+1];
```

# Queues als Array

- Array als Ringpuffer ($circular\ buffer$)
- zwei Pointer: $head$ and $tail$: erste benutzte und erste freie
- Komplexität: $O(1)$

```
Enqueue(Q, x) =
  Q[tail] := x;
  if tail == length(Q)
     then tail := 1          // start again with 1
     else tail := tail + 1;


Dequeue(Q) =
  x := Q[head]
  if head == length(Q)
     then head := 1          // start again with 1
     else head := head + 1
  return x;
```

## 7. November 2000 (3 Wochen)

# Verzeigerte Listen

- lineare Datenstuktur
- keine wahlfreier Zugriff wie beim Array, sonder über Zeiger
- Verschiedene Varianten:
  - *einfach* verzeigert: nur $prev$
  - *doppelt* verzeigert: $prev$ und $next$
  - *zirkulär*

# Verzeigerte Listen (2)

| Operation | Komplexität |
|---|---|
| Einfügen | $O(1)$ |
| Löschen (bei geg. Element) | $O(1)$ |
| Suchen | $O(n)$ |

# verzeigerte Listen (3)

```
List-search(L,k)        // k: key
  x   := head
  while  x != nil  and  x.key != k
     do  x := next(x);

  return x;
```

```
List-insert(L,x)                  // vorne Einfuegen
  next(x) := head;
  if  head != nil  then  prev(head) := x
      head := x
  prev(x) := nil
```

```
List-delete(L,x)
  if prev(x) != nil
     then next(prev(x)) := next(x)
     else          head := next(x)
  if next(x) != nil
     then prv(next(x)) := prev(x);
```

# Lektion VI

# Hashstrukturen

**Literatur:** Kapitel 12 aus [CLR90]. Hashfunktionen werden auch in [Knu73b] diskutiert.

**Inhalt:** Hashstrukturen · Hashing mit externer Verkettung · Hashing mit offener Adressierung · Hashfunktionen

# Einleitung

- drei Operationen eines Wörterbuches (*dictionaries*)
  - Suchen
  - Einfügen
  - Entfernen
- Hashtabellen: effizient zur Implementierung von Dictionaries

# Adreßtabellen mit direktem Zugriff

- implementiert mit Arrays, Zugriff direkt über Index = Schlüssel ($key$, eindeutig)
- Suchen, Einfügen, Entfernen: in konstanter Zeit ($O(1)$).

```
Search(T,k)     = return T[k];

Insert(T,x)     = T[key(x)] := x;

Delete(T,x)     = T[key(x)] := nil;
```

- Problem:
  - machbar nur für kleine Bereiche der Schlüssel, Platzverschwendung
  - nur destruktives Einfügen möglich

$\Rightarrow$ Hashtabellen

# Hashtabelle

- Verallgemeinerung von Arrays mit direkten Zugriff
- gegeben: Bereich $U$ der Schlüssel ("Universum")
- hash-Funktion: $h : U \to \{0, \ldots, m-1\}$
  - $|U| > m$

  $\Rightarrow$ $h$ nicht injektiv $\Rightarrow$ Kollision
  - $h(k)$: Hashwert von $k$
  - "zufällige" Funktion

# Kollisionsauflösung: Verkettung

- Verkettung (*chaining*, auch *external chaining*)
- Bild
- Varianten: Eindeutiger Schlüssel, implementierbar durch
  - Überschreibendes Einfügen, oder
  - Löschen nur mittels key (`Delete(k)`) und Löschen von allen Elementen mit passendem key, search gibt nur den ersten Treffer zurück.

```
Insert(T,x) = T[h(key(x))] := insert x at the head of T[h(key)];

Search(T,k) = Search for an element with key k in T[h(k)];

Delete(T,x) = Delete x from list T[h(k)];
```

# Analyse

**Definition 2 (Belegungsfaktor)** Der *Belegungsfaktor* $\alpha$ (*load factor*) einer Hashtabelle $T[0, \ldots, m-1]$ ist definiert als

$$\alpha = \frac{n}{m},$$

wobei $n$ gleich die Anzahl der gespeicherten Elemente ist.

- Worst-case: alle Elemente mit dem selben Schlüssel $\Rightarrow O(n)$
- im Mittel:
  - Annahme: Gleichverteilung = einfaches, uniformes Hashing
  - Hashfunktion $h(k)$ mit konstantem Aufwand $O(1)$

  $\Rightarrow$ Suchen nach Element mit Schlüssel k: linear in der Länge von $T[h(k)]$

# Analyse (2)

**Satz 3** Das erfolgreiche sowie erfolglose Suchen in einer Hashtabelle mit Verkettung und unter der Annahme einfachen, uniformen Hashings benötigt im Mittel $O(1 + \alpha)$ Zeitaufwand.

**Beweis:** Unterscheidung in *erfolglose* und *erfolgreiche* Suche.

**Fall 1:** erfolglos

- eine Liste wird bis *zum Ende* durchsucht
- durchschnittliche Länge der Listen: $\alpha = \frac{n}{m}$

$\Rightarrow O(1 + \alpha)$

**Fall 2:** erfolgreich

Annahmen:

- alle Schlüssel gleichwahrscheinlich, kein Schlüssel doppelt, Anfügen ans Ende

Ansatz:

$$|\text{Zugriffe beim Finden von } e| = |\text{Zugriffe beim Einfügen von } e| +1$$

Durschnittliche Länge beim Einfügen des $i$-ten Elements: $\frac{i-1}{m}$

$\Rightarrow$ im Durchschnitt: $\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) = \ldots 1 + \frac{\alpha}{2} - \frac{1}{2m}$

$\Rightarrow O(1+\alpha)$ $\square$

# Hashfunktionen

- Ziel: Gleichverteilung der Ergebnisse $j$ von $h(k)$ (aus dem Bereich $0$ bis $m-1$):

$$\sum_{\{k|h(k)=j\}} P(k) = \frac{1}{m}$$

- falls keys gleichverteilt aus $[0,1[ \Rightarrow h(k) = \lfloor km \rfloor \in \{0, \ldots, m-1\}$
- ansonsten: Heuristiken

| | |
|---|---|
| Divisionsmethode | $h(k) = k \mod m$ |
| Multiplikationsmethode | $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ (mit $0 < A < 1$) |
| Universal hashing | “zufällige” Auswahl von $h$ |

- wichtig: Auswahl der Parameter, abhängig auch von den keys
  - Divisionsmethode: $m$ prim, möglichst nicht nahe 2er-Potenz
  - Multiplikation: $m$ egal

# Offene Adressierung

- alle Elemente in der Hashtabelle gespeichert (keine Pointer, keine *externen* Listen)
  $\Rightarrow$ Verkettung wird errechnet
- Belegungsfaktor $\leq 1$
- Sondierung ($probe$): Suche nach freiem Slot

$\Rightarrow$ Hashfunktion

$$h : U * \{0, \ldots m-1\} \to \{0, \ldots, m-1\}$$

- Sondierungs-Sequenz ($probe\ sequence$) für Schlüssel $k$:

$$(h(k,0), h(k,1), \ldots, h(k,m-1))$$

- gute Hashfunktion: Vermeidung von Häufungen ($clusters$)

# Offene Adressierung: Hashfunktionen

- lineares Probing
  - gegeben: $h' : U \to \{0, \ldots m-1\}$: gewöhnliche Hashfunktion

$$h(k,i) = ((h'(k) + i) \mod m)$$

  - Probe-Sequenz: $T[h'(k), h'(k)+1, \ldots]$
  - Problem: primäre Häufung

- Quadratisches Probing
  - $h'$ wie oben

$$h(k,i) = ((h'(k) + c_1 i + c_2 i^2) \mod m)$$

- Doppeltes Hashing

– gute Methode
– $h_1$, $h_2$: zwei gewöhnliche Hashfunktionen

$$h(k,i) = (h_1(k) + ih_2(k)) \mod m$$

– $h_2(k)$ sollte relativ prim zu $m$ sein!

# Lektion VII

# Binäre Bäume

**Literatur:** Kapitel 13 aus [CLR90]. *Balancierte* Bäume stammen aus [AVL62].

**Inhalt:** Binäre Bäume · Baumoperationen

# Einleitung

- Unterstützung vieler Wörterbuchoperationen
- Baumstrukturen *effizient*
- Effizienz der Operationen proportional Höhe des Baumes $= O(\log n)$ (falls der Baum “ausgeglichen”)
- Viele Varianten, abhängig von der Anwendung
  - Rot/Schwarz-Bäume [Bay72]
  - B-Bäume
  - AVL-Bäume [AVL62]
  - Splay-Trees
  - $(a, b)$-Bäume, z.B. $(2, 3)$-Bäume (2-3-Bäume)
  - selbst-balancierende Bäume
  - ...

# Binäre Suchbäume

**Definition 4 (Binärer Suchbaum)** Ein *binärer Suchbaum* ist ein binärer Baum (*left*, *right*, (evtl. *parent*)) bei dem für alle Knoten $n$ gilt:

$$key(l) \leq key(n) \leq key(r).$$

wobei $l$ und $r$ beliebige Knoten aus dem rechten bzw. linken Unterbaum von $n$ sind

- Die Elemente des Baumes sind "sortiert"

$\Rightarrow$ Ausgeben der sortierten Elemente = rekursives Durchlaufen des Baumes (*in-order tree walk*)[4]

[4]es gibt daneben Durchlaufen in *Präordnung* und *Postordnung*.

# Durchlaufen

```
InOrder-Tree-Walk(t) =
  if    t != nil
  then  InOrder-Tree-Walk(left(t));
        print (key(t));
        InOrder-Tree-Walk(right(t));
```

# Baumoperationen: Suchen

- wichtige Operation auf Suchbäumen
- Komplexität: $O(h)$, wobei $h$ die Höhe
- Minimum/Maximum bestimmen: genauso einfach (der linkeste/rechteste Knoten im Baum)

```
Search(t,k) =                       // t : tree, k : key
  if           t == nil    then raise error("Not found")
  else if      k == key(t) then return t
  else if      k <  key(t)
         then  Search( left(t),k)
         else  Search(right(t),k)
```

# Baumoperationen: Suchen (2)

Hier das Ganze nochmals iterativ

```
Search(t,k) =                         // t : tree, k : key
  while (t != nil) and (k != key(t))
  do
    if k < key(t)
      then t :=  left(t)
      else t := right(t)
  od;
  if   (t == nil)
  then raise error("Not␣Found")
  else return t
```

# Baumoperationen: Einfügen

- Einfügen immer an den Blättern
- Komplexität: $O(h)$

# Einfügen: code

```
Insert(T,z) =                    // T: tree, z: node to insert
  y := nil; x := root(T);        // zwei Hilfsvariablen
  while  x != nil                // finde das passende Blatt
  do
    y := x;                      // y zum merken des Vaters
    if    key(z) < key(x)
    then x := left(x)
    else x := right(x)
  od;
  p(z) := y;                     // => x ist nil, y sein Vorgaenger
  if    y = nil                  // Sonderfall: y = Wurzel;
  then root(T) := z
  else if    key(z) < key(y)// f"uge an der passenden Stelle
       then  left(y) := z        // z als neues Blatt ein.
       else right(y) := z
```

# Baumoperationen: Löschen

- Problem (s. Bild):
  - Erhalt der Baumeigenschaft: innere Knoten müssen ersetzt werden
  - Erhalt der Sortierungseigenschaft: der Ersatz muß passen

- Baumnachfolger eines Knotens $x$: Nachfolger in der in-order-Baumordnung
  - falls Knoten mit rechtem Kind:

  $\Rightarrow$ Nachfolger $=$ der “linkeste” Knoten im rechten Teilbaum.

  $\Rightarrow$ Nachfolger hat kein linkes Kind!

  $\Rightarrow$ $key(left(x)) \leq key(x) \leq key\ succ(x) \leq key(right(x))$[5]

$\Rightarrow$ Nachfolger ersetzt zu löschenden inneren Knoten

- Komplexität: $O(h)$

[5] Wenn die Schlüssel alle unterschiedlich sind, gilt für die beiden ersten Ungleichungen schärfer $<$ anstelle $\leq$. Die erste Ungleichung gilt nur falls das linke Kind existiert.

```
Delete(T,z)                 // T = tree, z = zu loeschender Knoten
  if    left(z) = nil  or  right(z) = nil
    then y := z              // y tritt an die Stelle von x
    else y := Successor(z);  // wobei: y hat <= 1 Kind!

  if left(y) != nil          // Bestimmung des Kindes x
     then x :=  left(y)      // von y.
     else x := right(y);
                                   // Ver"andern der Verzeigerung
  if x != nil then p(x) = p(y);    // y herausgenommen => x mit
                                   // neuem Vater
  if p(y) = nil                    // Spezialfall: wurzel
    then root(t) = x
    else if   y = left(p(y))
            then  left(p(y)) := x
            else right(p(y)) := x   // y ist nun  vom
                                    //  urspr. Platz entfernt

  if y != z then key(z) := key(y); // z durch y ersetzen

  return y;
```

## 15. November 1999 (4 Wochen)

# Lektion VIII

# Rot-schwarz-Bäume

**Literatur:** Kapitel 13 aus [CLR90]. Die Rot/Schwarz-Bäume wurden von [Bay72] eingeführt.

**Inhalt:** Rot/schwarz-Bäume · Rotation · Erhalt der Rot-Schwarzeigenschaft bei Einfügen und Löschen

# Einführung

- Hauptproblem der Suchbäume: Effizienz hängt von der Höhe ab

$\Rightarrow$ je balanzierter desto „besser“ der Baum

- für die Suche: ja; andererseits
  - dauerndes Rebalanzieren kostet auch Zeit $\Rightarrow$Balanciertheitsanforderungen am besten nicht zu strikt (vor allem bei vielen Lösch- und Einfügeoperationen
  - Wie stellt man fest ob ein Baum balanciert ist? (Laufzeit!)
  - speichert man Zusatzinformationen darüber ab? (Speicherplatz!)

# Definition

- zwei Arten von Knoten: Schwarze werden strikt balanciert, rote dienen als Schlupf.

- Anteil des Schlupfes muß beschränkt sein

**Definition 5** rot/schwarz-Bäume sind binäre Suchbäume mit einem Bit Zusatzinformation: der Farbe und folgenden Bedingungen:

- jeder Knoten ist entweder rot oder schwarz

- Blätter (nil) sind schwarz

- die Kinder eines roten Knotens sind schwarz

- jeder einfache Pfad von einem gegebenen Knoten zu einem Blatt enthält die selbe Anzahl an schwarzen Knoten[6]

- Bild

- Schwarzhöhe $bh(x)$ eines Knotens: die Anzahl der schwarzen Knoten auf den Pfaden (ausschließlich $x$) zu den Blättern

**Lemma 6** Ein rot-schwarz Baum mit $n$ internen Knoten hat eine Höhe von höchstens $2\lg(n+1)$.

$\Rightarrow$ die Operationen Suchen, Minimum, Maximum, Successor, Predecessor: $O(\lg n)$.

- Einfügen und Löschen?

[6] einfacher Pfad: alle Knoten (des Pfades) unterschiedlich. Falls die Pfade nur von der Eltern zu den Kindern erlaubt sind, sind alle Pfade einfach!

# Rotation

- Rotation
- Erhalt der In-Ordnung der Schlüssel
- Komplexität $O(1)$
- Bild

```
Rotate−l (t,x) =                          // t = Baum, x = Knoten
  if     (x = nil) or (right(x) = nil)
  then   raise error "illegal␣rotation"
  else   y := right(x);                   // rette y
         p(y) := p(x);                    // ver"andere y's Verwandschaft
         if    y != nil
         then  right(x) := left(y);       // beta
         if    (left(y)) != nil
         then  p(left(y)) := x;

         left(y):= x                      // x != nil
         if    root(t) = x                // Spezialfall
         then  root(t) := y
         else  if    x = left(p(x))       // p(x) definiert
               then  left(p(x)) := y
               else  right(p(x)) := y

         left(y) := x                     // zum Schluss: x als Kind von y
         p(x)  := y;
```

# Einfügen

- Zunächst Einfügen wie im gewöhnlichen binären Suchbaum

$\Rightarrow$ neues rotes Blatt.

  - keine Schwarz-Verletzung, aber u. U.
  - Rot-Verletzung

- Bei rot-Verletzung:
  - Reparieren durch Umfärben und Rotation
  - keine Neueinführung von Schwarzverletzungen

# Einfügen: drei Fälle (+ 3 symmetrische)

Die Graphik sind leider noch nicht nach HTML übersetzbar

```
insert(T,x)
  tree_insert(T,x);                          // normal einfuegen
  color(x)   := red                          // am Anfang rot

  while x != root(t) and color(p(x)) = red   // Rot-verletzung
  do
    if  p(x) = left(p(p(x));                 // Vater ist linker Sohn
      then  y := right(p(p(x))               // y = Onkel, d.h.
                                             // rechter Sohn des Opas
          if  color(y) = red                 // Onkel rot
            then  color(p(x))   := black;    // => umfaerben
                  color(y)      := black;
                 color(p(p(x)) := red;
                             x := p(p(x))
            else if x = right(p(x))          // Onkel schwarz
                 then x := p(x);             // x ist rechtes Kind
                      left_rotate(T,x)

                 color(p(x)) := black;       // Vater <- schwarz
                 color(p(p(x)) := red;       // Opa    <- rot
               right_rotate(T,p(p(x)));      // Ausgleich
```

```
        else  .......  // analog
  done;
  color(root(T)) := black;
```

# Löschen

Siehe Übung

21. November 2000 (5 Wochen)

# Lektion IX

# Graphen

**Literatur:** Kapitel 23 aus [CLR90]. Breitensuche stammt von Moore [Moo59]. Der Erfinder der Tiefensuche scheint unbekannt. Der angegebene Algorithmus für die starken Zusammenhangskomponenten ist von Tarjan [Tar72].

**Inhalt:** Graphen · Repräsentierungen · Erreichbarkeit · Breitensuche · Tiefensuche · backtracking · topologisches Sortieren · Berechnung starker Zusammenhangskomponenten

# Suchen in Graphen

- Wichtige Klasse von Algorithmen, viele Anwendungen
- Problem: Durchlaufen eines gegebenen Graphen, mit Besuch der Knoten + Berechnung von Strukturinformation über den Graphen
- Unterscheidung nach
  - Durchlaufordnung und (davon abhängig)
  - Art der Zusatzinformation über den Graphen und seine Struktur: z. B.
    * gegenseitige Erreichbarkeit von Knoten
    * Zyklen
    * Bäume als Teilstrukturen im Graphen
    * (stark) zusammenhängende Teilgraphen
- Zwei Klassiker:

1. Tiefensuche
2. Breitensuche

# Graphen und ihre Repräsentierung

**Definition 7 (Graph)** Ein (gerichteter) *Graph* $G = (V, E)$: $V$ Menge von Knoten, $E \subseteq V \times V$ Kanten.

- Zwei Standardmöglichkeiten:
  1. Adjazenzlisten
  2. Adjazenzmatrix
- Speicherbedarf:
  - Adjazenzliste: $O(V + E)$
  - Adjazenzmatrix: $O(V^2)$

# Breitensuche

- klassischer Graphsuchalgorithmus, Reihe von Verallgemeinerungen
- ursprüngliches Problem [Moo59]: kürzeste Wege in Labyrinthen
- Problem:
  - gegeben: $G = (V, E)$ + einen ausgezeichneten Knoten (Quelle) $s \in V$.
  - gesucht: Alle von $s$ aus erreichbaren Knoten
  - Zusatzinfo:
    * Abstand $\delta(s, u)$ jeden Knotens $u$ zu $s$[7]
    * Breitensuchbaum: Baum mit kürzesten Pfaden von $s$ zu jedem Knoten.

[7] = Länge des kürzesten Pfades von $s$ zum Knoten, falls unerreichbar, Abstand $= \infty$.

# Prinzip

- Grundgedanke der Breitensuche:

  Suche “in die Breite”: Bevor Knoten mit Abstand $k+1$ von der Quelle entdeckt (bzw. abgehandelt) werden, sind alle Knoten mit Abstand $k$” entdeckt (bzw. abgehandelt)

- Sicherstellung der Breitendurchlaufordnung:

  Behandlung der entdeckten Knoten in Fifo-Manier (Queue)

# Breitensuche (2)

- Zur Erkennen von bereits gesehenen Knoten: 3 Farben:[8]

  1. weiß: unentdeckt
  2. grau: entdeckt, aber noch nicht fertigbehandelt
  3. schwarz: fertig (und damit auch entdeckt)

- Reihenfolge: weiß $\Rightarrow$ grau $\Rightarrow$ schwarz[9]

- Zusatzinfo:

  – Distanz: Distanz des Knotens, von dem man entdeckt wird + 1
  – Breitensuchbaum: Merken des Knotens, von dem man entdeckt wird.

[8] Daß man 2 Farben braucht, sollte klar sein, die dritte kennzeichnet Knoten die man bereits entdeckt hat, deren adjazierene Knoten man aber noch nicht kennt. Wg. Breitenstrategie arbeitet man die Nachbarn eines neuentdeckten Knotens nicht sofort ab. Das sind die grauen Knoten.

[9] graue Knoten: auf der “Grenze” der Breitensuche: ein schwarzer Knoten kann nie einem weißen benachbart sein.

```
bfs (G,s) =          // Graph und Quellknoten
  for each vertex u in V[G] - {s}
  do color[u] := white
     d[u]     := ∞;
     p[u]     := nil
  done;
  color[s]    := gray;
  d[s]        := 0;
  p[s]        := nil;
  Q.enqueue(s);                      // beginne mit der Quelle

  while not Q.isempty()
  do
     u := Q.head();                  // noch kein dequeue!
     for each v in Adj[u]            // alle Nachbarn.
     do
        if   color[v] = white
        then color[v] := gray;
             d[v]     := d[u] + 1;
             p[v]     := u
             Q.enqueue(v)           // speichere den Nachbarn
        fi;
     done;
     Q.dequeue()                     // u ist abgearbeitet
```

```
    color(u) := black;        // und als fertig markiert
done
```

# Breitensuchalgorithmus: Analyse

- Zeitkomplexität:
  - jeder Knoten wird höchstens einmal in $Q$ eingereiht, und damit auch höchstens einmal aus ausgereiht: $O(V)$[10]
  - für jeden Knoten wird die Adjazenzliste durchlaufen, insgesamt höchstens $O(E)$

$\Rightarrow O(E+V)$

5. Dezember 2000 (6 Wochen)

[10] Queue-Operationen `enqueue` und `dequeue` kosten $O(1)$, siehe Kapitel .

# Korrektheitsargument

Siehe Vorlesung

# Tiefensuche

- Suche zunächst in die Tiefe[11]
- Wie bei Breitensuche: Farben zur Suchsteuerung
  1. Weiß: ungesehen
  2. Grau: entdeckt
  3. Schwarz: fertig
- anstelle von Queue: Stack
- “Backtracking”
- Teilgraph der Vorgänger $G_\pi = (V, E_\pi)$:

$$E_\pi = \{(\pi(v), v) \mid v \in V, \pi(v) \neq \perp\}$$

[11] der vorgestellte Algorithmus wendet dies auf alle Knoten der Reihe nach an, das ist aber nicht zentral. Die eigentliche Tiefensuche ist *DFS-Visit*.

$\Rightarrow$ Menge von Tiefensuchbäumen (= Tiefensuchwald, *depth-first forest*)

- Zusatzinformation: Zeitstempel $d[v]$ *(discovered)* und $f[v]$ *(finished)* wobei $d[v] < f[v]$.

# Algorithmus (1)

```
DFS(G)                                  // Gerichteter Graph
  for all vertices v ∈ V[G]   // Initialisierung
  do
      color[u] := white;
       pred[u] := nil;
  od
  time := 0;                            // Ende der Initialisierung
  for all vertices  u ∈ V[G]
  do
       if color[u] = white then DFS—Visit(u); // eigentl. Tiefensuche
  done;
```

# Algorithmus (2)

```
DFS—visit (u)                                   // Eigentliche Tiefensuche
  color [u] := gray;                            // Knoten entdeckt
  time := time + 1;  d[u] := time;              // merke Entdeckungszeit
  for each v in Adj[u]                          // erkunde Kante (u, v)
  do  if    color [v] = white
      then  pred [v] := u                       // pred = Entdeckerknoten
            DFS—visit (v)                       // rek. Abstieg
  done;
  color [u] := black;                           // u ist fertig, merke
  time := time + 1; f[u] := time;               // die Fertigstellzeit
```

# Tiefensuche: Analyse

- Bild 23.5(1)
- Komplexität:
  - $O(V)$ für $DSF$
  - $O(E)$ für $DFS\text{-}Visit$

$\Rightarrow$ Laufzeit für Tiefensuche: $O(E+V)$

# Weitere Eigenschaften

**Proposition 8 (Klammerung)** Gegeben zwei Knoten $u, v$ aus $G = (V, E)$. Es gilt genau eine der folgenden Bedingungen:

1. $[d[u], f[u]]$ ist echtes Teilintervall von $[d[v], f[v]]$, $u$ ist Nachfahre von $v$ in Tiefensuchbaum

2. Symmetrisch.

3. $[d[u], f[u]]$ und $[d[v], f[v]]$ disjunkt

$\Rightarrow$ Verschachtelung der Intervalle:

**Korollar 9** Knoten $v$ ist echter Nachfolger von $u$ im Tiefensuchwald gdw. $d[u] < d[v] < f[v] < f[u]$

- Klassifikation von Kanten:

– Baumkante: Kanten des Tiefensuchwaldes
– Rückwärtskante: Kanten von Nachfahren zu Vorfahren in einem Tiefensuchbaum (inkl. Selbst-Schleifen)
– Vorwärtskante: Kante von Vorfahren zu Nachfahren gemäß einen Tiefensuchbaumes, die nicht im Baum sind
– Querkante: alle anderen

# Topologisches Sortieren

- Anwendung der Tiefensuche
- algorithmische Umsetzung der bekannten Tatsache: jede Halbordnung läßt sich zu einer totalen/linearen Ordnung erweitern.
- Graphdarstellung einer Halbordnung: DAG = „Gerüst“ der Ordnung

Zunächst ein paar Definitionen:

**Definition 10** Gegeben eine binäre Relation $R$. Die *reflexive, transitive Hülle* $R^*$ of $R$ ist gegeben als die kleinste binäre Relation sodaß $xR^*x$ und $xRyR^*z$ impliziert $xR^*z$. Mit $R^+$ ist die Relation $RR^*$ gemeint.

**Definition 11** Eine binäre Relation $R \subseteq S \times S$ auf einer Menge $S$ ist eine *Halbordnung* (Notation dann oft $\leq$ für $R$) wenn sie reflexiv, transitiv, und antisymmentrisch ist. Eine *totale* oder *lineare* Ordnung ist eine Halbordnung mit der zusätzlichen Eigenschaft der *Vergleichbarkeit*. $(\forall x, y . x \leq y$ oder $y \leq x)$

**Definition 12 (DAG)** Ein gerichteter, azyklischer Graph (DAG) ist ein gerichteter Graph ohne Zyklen, d.h., es für alle Knoten $u, v \in V$ gilt: Wenn $u \to^+ v$, dann $u \neq v$.

- Eine topologische Sortierung eines DAGs $G$ ist eine lineare Ordnung der Knoten von $G$, Kompatibilität: falls $(u, v)$ eine Kante auf $G$, dann $u < v$ in der linearen Ordnung.

- d.h., kein Sortieren wie bei den Sortieralgorithmen

- Bild

- Lineare Ordnung = Zeitstempel der Schwarzfärbung = f-Zeit.

# Topologisches Sortieren (2)

- Eingabe: DAG („=“ Halbordnung), Ausgabe: verzeigerte Liste („=“ lineare Ordnung)

```
Topological-Sort(G)                              // G ist ein DAG

  call DFS(G) to compute finishing-times f[v] for all v;
  as each vertex is finished, insert it onto the front of a list;
  return the linked list
```

- Komplexität: Laufzeit $O(V+E)$.

12. Dezember 2000, Beweis nächstes Mal (7 Wochen)

# Topologisches Sortieren (3)

**Satz 13 (Weiße Pfade)** Gegeben Graph $G$. Im Tiefensuchwald für $G$ gilt: $v$ ist Nachfahre von $u$ gdw. zur Zeit $d[u]$ der Knoten $v$ durch $u$ auf einem weißen Pfad erreichbar ist.[12]

**Lemma 14** Ein gerichteter Graph ist azyklisch gdw. die Tiefensuche keine Rückwärtskanten produziert.

**Beweis:** **Fall 1:** $\Leftarrow$
Sei $G$ zyklisch $\Rightarrow$ es gibt einen Zyklus $c$: $w \rightarrow^{+} w$. Sei $v$ der erste entdeckte Knoten aus $c$ und $u$ sein Vorgänger im Zyklus. Zur Zeit $d[v]$: weißer Pfad $v \rightarrow^{*} u \Rightarrow u$ wird Nachfolger im Tiefensuchwald $\Rightarrow$ Behauptung ($(u, v)$ ist Rückwärtskante).

**Fall 2:** $\Rightarrow$
$DFS$ findet Rückwärtskante $(v, u)$, andererseits $u \rightarrow^{*} v$ mittels Baumkanten (weil $u$ Vorgänger von $v$ im Tiefensuchbaum) □

**Lemma 15 (Korrektheit)** Der Algorithmus angewandt auf einen DAG, liefert eine topologische Sortierung.

[12] Ein weißer Pfad ist ein Pfad, der nur weiße Knoten enthält.

**Beweis:** Prüfen der Kompatibilität: falls $u \to^+ v$ in $G$, dann $f[u] < f[v]$. Sei $(u, v)$ eine Kante. Wenn sie mit $DFS$ erkundet wird, ist $v$ nicht Grau (sonst wäre $v$ ein Vorfahre $\Rightarrow$ Rückwärtskante) $\Rightarrow$ $v$ ist weiß oder schwarz.

**Fall 1:** $v$ weiß
$v$ wird Nachfahre von $u \Rightarrow f[v] < f[u]$

**Fall 2:** $v$ schwarz
Dann sofort $f[v] < f[u]$ □

# Starke Zusammenhangskomponenten

- klassische Anwendung der DFS: Zerlegen eines Graphen in stark-zusammenhängende Komponenten

**Definition 16 (Starker Zusammenhang)** Gegeben gerichteter Graph $G = (V, E)$. Eine *starke Zusammenhangskomponente* von $G$ ist eine maximale Knotenmenge $U \subseteq V$ mit: für alle $u_1, u_2 \in U$ gilt $u_1 \rightarrow^* u_2$ und $u_2 \rightarrow^* u_1$.

- Idee: $G$ und $G^t$ haben die selben starken Zusammenhangskomponenten $\Rightarrow$ zweimaliges Anwenden von $DFS$, auf $G$ und auf den transponierten Graphen[13] $G^t$

- Komplexität: lineare Zeitkomplexität $O(E + V)$

[13] Sei $G = (V, E)$ gegeben, dann ist $G^t = (V, E^t)$ wobei $(v, u) \in E^t$, gdw. $(u, v) \in E$. Die starken Zusammenhangskomponenten von $G$ und $G^t$ stimmen überein.

# Algorithmus

```
Strongly-connected-components(G)      // G: gerichteter Graph

  call DFS(G) to obtain finishing times f[u] for the vertices;

  compute G^t = transpose(G)           // transponiere G

  call DFS(G^t), but in the main loop of DFS,
       consider the vertices in order of
       decreasing f[u]

  output the vertices of each tree in the depth-first
      forest from the previous step as a separated strongly
      connected component.
```

19. Dezember 2000 (8 Wochen)

# Analyse

- Vorvater $\phi(u)$ eines Knoten $u$ ist derjenige Knoten $w$ mit $u \rightarrow^* w$ und maximalen $f[w]$.

**Lemma 17** Der Vorvater $\phi(u)$ bzgl. einer Tiefensuche ist ein Vorgänger von $u$,

**Korollar 18** Für alle Knoten $u$ gilt: $u$ und $\phi(u)$ liegen in der selben starken Zusammenhangskomponente.

**Beweis:** Folgt direkt aus dem vorangegangenen und der Definition von Vorvater. □

# Analyse (2)

- es gilt also: für jede SCC ist der Vorvater
  - der erste Knoten der entdeckt wird und
  - der letzte Knoten der beendet wird!

$\Rightarrow$ die starke Zusammenhangskomponente von $r$ sind diejenigen Knoten die von $r$ in $G^t$ erreichbar sind.

9. Januar 2001 (9 Wochen)

# Lektion X

## Spannbäume

**Literatur:** Kapitel 24 aus [CLR90].

**Inhalt:** Kantengewichte · minimale Spannbäume · *greedy*-Strategien · Kruskals Algorithmus · Prims Algorithmus

# Minimaler Spannbaum

- Motivation: Verknüpfung[14] einer geg. Anzahl von Knoten mit minimalen Kosten (Verkabelung, Routing, ... )

- Verknüpfung von $n$ Knoten: mit $n-1$ Kanten

- Kosten: modelliert als „Kantengewicht“

- Gegeben:
  - $G = (V, E)$ ungerichtet, verbunden, und mit $E \subseteq V \times V$ Menge potentieller Verbindungen
  - Kantengewicht: $w : V^2 \to \mathbb{R}$.

- Gesucht: $T \subseteq E$ sodaß

[14] jeder von jedem erreichbar. Zum Vergleich mit den starken Zusammenhangskomponenten: der Graph ist hier *ungerichtet*!

- azyklisch (Baum)
- Verbindung aller Knoten (*aufspannend*, *spanning*)
- minimale Kosten

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

minimal.

$\Rightarrow$ Problem des minimalen Spannbaums

- Beispiel für greedy-Strategie
- Bild

3. Januar 1999 (9 Wochen)

# Spannbaumalgorithmus

- greedy-Strategie:
  - Heuristik für Optimierungsprobleme
  - falls eine Entscheidung zu treffen ist: entscheide dich für die augenblicklich beste Alternative (ohne auf mögliche Nachfolge-Vorteile zu achten)

- iterativer Aufbau eines minimalen Spannbaumes

$\Rightarrow$ schrittweises Hinzufügen von Kanten.

**Definition 19** Sei $A$ Teilmenge eines minimalen Spannbaumes für $G$. Eine Kante $(u, v)$ ist sicher für $A$, falls $A \cup \{(u, v)\}$ Teilmenge eines minimalen Spannbaumes ist.

```
Generic-MST(G,w) =
  A := 0;
  while A does not form a spanning tree
  do
    find an edge (u,v) that is safe wrt. A
    A := A ∪ {(u,v)}
  done;
  return A
```

# Sichere Kanten

- Problem: wie findet man sichere Kanten?

**Definition 20** Ein Schnitt $(S, V - S)$ (*cut*) eines ungerichteten Graphen ist eine Partition von $V$. Eine Kante kreuzt den Schnitt, falls ein Endpunkt in $S$ ist und der andere in $V - S$. Ein Schnitt respektiert eine Menge von Kanten, wenn keine Kante den Schnitt kreuzt. Eine Kante, die einen Schnitt kreuzt, ist leicht falls ihr Gewicht von allen kreuzenden Kanten minimal ist.[15]

**Theorem 21** • $G = (V, E)$ zusammenhängend, ungerichtet, mit reellwertiger Gewichtsfunktion $w : E \to \mathbb{R}$.

- $A \subseteq E$, $A$ Teil eines min. Spannbaumes
- $(S, V - S)$: Schnitt der $A$ respektiert
- $(u, v)$: leichte Kante, die den Schnitt $(S, V - S)$ kreuzt

$\Rightarrow$ $(u, v)$ sicher für $A$.

[15] Es kann natürlich mehrere leichte/minimale Kanten geben.

# minimaler-Spannbaum-Algorithmus

- während der Algo läuft: $A$ immer azyklisch (Invariante)

$\Rightarrow$ $G_A = (V, A)$ Wald

- zu Beginn: Wald mit $|V|$ (einknotigen) Bäumen

- jede hinzugefügte sichere Kante verknüpft zwei Bäume

$\Rightarrow$ Schleife $|V| - 1$-mal durchlaufen

**Korollar 22** • $G = (V, E)$ zusammenhängend, ungerichtet, Gewichtungsfunktion $W : E \to \mathbb{R}$.

- $A \subseteq E$, $A$ Teil eines min. Spannbaumes

- $C$ eine Zusammenhangskomponente (hier Baum) im Wald $G_A = (V, A)$.

Es gilt: Wenn $(u, v)$ eine leichte Kante ist die $C$ mit einer anderen Komponente verbindet, dann ist $(u, v)$ sicher für $A$.

# Kruskals Algorithmus

- Spezialisierung des generischen Algorithmus'
- greedy-Strategie
- direkt Umsetzung des Korollars 22

$\Rightarrow$ sichere Kante = die mit dem geringsten Gewicht, die zwei Bäume verbindet.

$\Rightarrow$ nimm immer die Kante mit geringstem Gewicht: wenn sie zwei Komponenten verbindet: hinzufügen, wenn nicht, ignorien

- Hilfsfunktionen + Hilfsdatenstrukturen:
  - Wald = disjunkte Mengen/Partition (von Knoten)
  - `find-set`: finde Repräsentanten
  - `Union`: Vereinigung
  - `Makeset`: ein-elementige Menge

# Kruskal (2)

```
mst-kruskal(G,w)
  A := 0;                            // A: Kantenmenge
                                     // Invariante: A Teil eines min. S–B
  for all v ∈ V[G]
  do Make–Set(v) done;               // Wald aus lauter Knoten

  sort the edges of E by non-decreasing weight w;

   for all edges (u,v) ∈ E (in order)
   do
     if    Find-Set(u) ≠ Find-Set(v) // nicht im selben Baum?
     then
       A := A ∪ {(u,v)}; // fuege Kante hinzu
       Union(u,v);                   // vergroebere Partition
     fi;
   done;
   return A;
```

# Kruskal (3)

- Laufzeit von Kruskal:

- hängt von der Implementierung der Hilfsstukturen ab

| | |
|---|---|
| Initialisierung | $O(V)$ |
| Sortieren | $O(E\log(E))$ |
| innere Schleife | $O(E)$ |
| Datenrepräsentierung | $\alpha(E,V)$. |

$\Rightarrow O(E\log(E))$[16]

[16] unter der Voraussetzung, daß $\alpha(E,V) = O(\log E)$, z.B., heap-Implementierung.

# Prims Algorithmus

- Spezialisierung des generischen Algorithmus'
- $A$: kein Wald, sondern ein einzelner Baum[17]
- Wachsen des Baumes startend von beliebiger Wurzel $r$
- Baum bestimmt den Schnitt

$\Rightarrow$ Iterationsschitt: füge leichte Kante vom Baum nach außerhalb des Baumes hinzu

- greedy: mit jedem Schritt wird der Baum minimal schwerer
- Datenstruktur
  - Ordnen der Knoten außerhalb des Baumes

  $\Rightarrow$ Priority Queue (z.B. wieder mittels Heap)

[17] genauer: alle anderen Bäume des Waldes sind nur Einzelknoten.

```
mst-prim (G, w, r )                 // G = Graph, w = Gewichtung, r: Wurzel
  Q := V(G);                        // priority queue von Knoten
                                    // Q : Knoten noch nicht im Baum

  for all u ∈ Q do key[u] := ∞ done;
  key[r] := 0;
  p(r)   := nil;                    // r ohne parent
  while Q ≠ ∅
  do
    u := extract_min(Q);            // u dem Baum hinzugefuegt
    for all v ∈ Adj(u) // bewerte alle Nachbarn
    do
       if   v ∈ Q and w(u,v) < key(v)
       then p(v)   := u;            // parent
            key(v) := w(u,v)
       fi
    done
  done
```

# Prim (3)

16. Januar 2001 (10 Wochen)

# Lektion XI

# Kürzeste Pfade

**Literatur:** Teile von Kapitel 25/26 aus [CLR90]. [Ski97]

**Inhalt:** Einleitung · Varianten · Relaxation · Dijkstras Algorithmus · Bellman-Ford · lineare Programmierung

# Einleitung

- Motivation: Routing, Streckenplanung
- klass. Beispiel für ein (gutartiges) Optimierungsproblem
- gegeben: gerichteter Graph mit Gewichtungsfunktion $w : E \to \mathbb{R}$
- Problem: finde einen Verbindung/Pfad mit minimalen Kosten

# Kürzeste Pfade: Definition

**Definition 23 (Kürzester Pfad)** Gegeben $G = (V, E)$ gerichteter Pfad, $w : E \to \mathbb{R}$. Das Gewicht eines Pfades $p = (v_0, v_1, \ldots, v_k)$ ist definiert:

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

Das Gewicht des kürzesten Pfades von $u$ nach $v$ ist das Minimum:

$$\delta(u, v) = \begin{cases} \min\{w(p) \mid u \xrightarrow{p} v\} & \text{falls } \exists \text{ Pfad von } u \text{ nach } v \\ \infty & \text{sonst} \end{cases}$$

Ein *kürzester Pfad* von $u$ nach $v$ ist ein Pfad von $u$ nach $v$ mit $w(p) = \delta(u, v)$.

# Problemvarianten

- spezifizierte Quelle (*single-source*): finde einen kürzesten Pfad von einem gegeben Knoten für jeden Zielknoten
- spezifiziertes Ziel: (*single-destination*): duales Problem
- spezifizierte Quelle und Ziel (*single-pair*)
- kürzeste Pfade für alle Knotenpaare.

# Kürzeste Pfade

- Hier: *single-source*
- Eingabe: Gerichteter Graph $G = (V, E)$ + Quelle $s \in V$.
- Ausgabe: nicht nur minimales Gewicht, sondern auch Pfad

$\Rightarrow$ Baum kürzester Pfade bei gegebenem Startknoten $s$:[18] $G' = (V', E')$ mit:

  - $V' \subseteq V$: Menge der von $s$ aus erreichbaren Knoten
  - $G'$: bewurzelter Baum mit Wurzel $s$
  - der (eindeutige, einfache) Pfad von $s$ nach $v$ in $V'$ ist ein kürzester Pfad von $s$ nach $v$ in $G$.

- Repräsentierung: Im Lauf des Graphtraversierung: merke den Vorgänger $\pi$ für jeden Knoten $\Rightarrow G_\pi = (V_\pi, E_\pi)$.[19]

[18] es kann mehrere Bäume mit kürzesten Pfaden geben. In der Regel will man einen davon.

[19] Vergleiche Breitensuche und Vorgängergraph. Der Vorgängergraph des Breitensuchalgorithmus entspricht für den Spezialfall, daß alle Kanten das Gewicht 1 haben, den Baum der kürzesten Pfade.

- Problem: Kanten mit negativen Kosten, insbesondere Zyklen mit negativen Kosten.

# Kürzeste Pfade (2)

**Hauptidee:** Der kürzeste Pfad zwischen zwei Knoten enthält kürzeste Pfade als Teile

$\Rightarrow$ gutartiges Problem, viele Techniken anwendbar

- greedy
- Relaxation
- dynamische Programmierung....

**Lemma 24** geg $G = (V, E)$, gerichtet, gewichtet mit $w : E \to \mathbb{R}$. $(v_1, \ldots, v_n)$ ist ein kürzester Pfad von $v_1$ nach $v_n$ und $w = (v_i, \ldots, v_j)$ mit $1 \le i \le j \le n$ ein Teilpfad. Dann ist $w$ ein kürzester Pfad von $v_i$ nach $v_j$.

$\Rightarrow$ Zerlegung: Sei $s \to {}^*v$ kürzester Pfad mit $s \to {}^*u \to v$, dann $\delta(s, v) = \delta(s, u) + w(u, v)$.

- „Dreiecksungleichung“
  - $\delta(s, v) \le \delta(s, u) + w(u, v)$ (mit $(u, v) \in E$)
  - $\delta(s, v) \le \delta(s, u) + \delta(u, v)$

# Relaxation

- Methode zur Optimierung bei gutartigen Probleme
- pessimistische (worst-case) Abschätzungen
- im Laufe des Algos immer weiter verbessert
- gutartig: "monoton", neue Information verbessern die Schätzung nur
- für kürzeste Pfade: für alle $v \in V$: $d(v)$ als obere Schranke für den Pfad von $s$ nach $v$.

# generische Relaxation

- generischer Relaxationsalgorithmus
  - Initialisierung: für all $v \in V$:
    - $d(v) = \infty$ (außer $s$)
    - $d(s) = 0$
    - $\pi(V) = \perp$
  - Relaxierung einer Kante: Anwendung der Dreiecksungleichung:

```
Relax(v1, v2) =
   if   d(v2) > d(v1) + w(v1, v2)
   then    π(v2) := v1;
           d(v2) := d(v1) + w(v1, v2)
   fi
```

- Unterschiedliche Algorithmen, je nachdem wann und wieoft Kanten relaxiert werden.

# Dijkstra

- Voraussetzung: nicht-negative Gewichtung
- $S$: Menge von Knoten, deren min. Gewicht bereits bestimmt ist: d.h. $v \in S \Rightarrow d(v) = \delta(s,v)$
- Hauptschritt:

  Nimm den Knoten $u$ mit der momentan geringsten Pfadabschätzung aus $V - S$, relaxiere alle an $u$ adjazierenden Knoten und stecke $u$ nach $S$.

$\Rightarrow$ Hilfsstruktur: Priority Queue

- Greedy-strategie: relaxiere immer eine Kante zu dem unbehandelten Knoten mit dem geringsten Abschätzung des Abstandes.
- Graphsuche, gesteuert nach den besten Abschätzungen, Ähnlich Breitensuche

- Laufzeitkomplexität:
  - Jeder Knoten einmal aus der Queue genommen: $O(V^2)$[20]
  - jede Kante genau einmal relaxiert: $O(E)$

  $\Rightarrow$ $O(V^2 + E) = O(V^2)$.

[20] extrahieren kostet $O(V)$ (z.B. Arrayimplementierung), wenn man nicht noch Zusatzannahmen macht.

# Dijkstra (2)

```
Dijkstra(G,w,s)
  Intitialise(G,s);          // siehe generischer Algo.
  S := ∅;                    // Knoten mit bekannter Distanz
  Q := V;                    // priority queue

  while Q ≠ ∅
  do
     u := extract-min(Q);
     S := S ∪ {u};
     for all vertices v ∈ Adj(u)
     do
           relax(u,v)
     done
  done
```

# Bellmann-Ford

- beliebige Kantengewichtung, entdeckt negativ-gewichtete Zyklen
- Relaxation der Abschätzung des Abstandes.
- Laufzeitkomplexität: $O(VE)$.
- Relaxation aller Kanten, ohne Berücksichtigung ihres Gewichtes, dafür genügend oft (d.h. keine greedy Strategie

# Bellmann-Ford (2)

```
Bellmann-Ford(G,w,s)                     // G = (V,E), Gewichtung w

  initialize(G,s);                       // s. generischer Algo.
  for i := 1 to size(V)-1
  do
    for all edges (u,v) ∈ E
    do
      relax(u,v)
    done
  done;
  for all edges (u,v) ∈ E          // Test auf negative Zyklen
  do
    if d(v) > d(u) + w(u,v)
       then return false
       else return true
  done;
```

# Lineare Programmierung

- spezielles Optimierungsproblem
- Gegeben:
  - lineare Zielfunktion $+$
  - Menge von linearen Ungleichungen $(constraints)$
- Gesucht: Eingabe, sodaß Zielfunktion mit optimaler/minimaler Wert.
- genauer:
  - gegeben
    * $Ax \leq b$
    * $f(x_1 \ldots x_n) = \sum_{i=1}^{n} c_1 x_i$
  - gesucht: minimiere $f(x_1 \ldots x_n)$
- berühmtes Lösungsverfahren: Simplex-Verfahren (und andere)

- Eingeschränkt hier: Differenzenungleichungen, d.h. Ungleichungen der Form:[21]

$$x_i - x_j < b_k$$

[21] jede Reihe der Matrix enthält eine $1$ und eine $-1$ und ansonsten nur $0$.

# Darstellung als Graphproblem

- Gegeben: Differenzenungleichungen: $Ax \leq b$.
- Definiere Constraintgraph $G = (V, E)$ mit
  - $V = \{v_1, \ldots v_n\}$ (ein Extraknoten $v_0$!)
  - $E = \{(v_i, v_j) \mid x_j - x_i \leq b_k\} \cup \{(v_1, v_1), (v_0, v_n)\}$. Die Gewichte $W(v_i, v_j) = b_k$ $(i, j \neq 0)$ und die Gewichte der von $v_0$ ausgehenden Kanten $= 0$.
-

# Literatur

[AHU89] Alfred Aho, John E. Hopcroft, and Jeffrey Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1989.

[AVL62] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[Bay72] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[CLR90] Thomas H. Cormen, Charles E. Leierson, and Ronald L. Rivest. *An Introduction to Algorithms*. MIT Press, 1990.

[Heu97] Volker Heun. Grundlegende Algorithmen. Vorlesungsskript TU München, October 1997.

[HJ89] C. A. R. Hoare and Cliff B. Jones, editors. *Essays in Computing Science*. International Series in Computer Science. Prentice Hall, 1989.

[Hoa62] C. A. R. Hoare. Quicksort. *BCS Computer Journal*, 5(1):10–15, 1962. Reprinted in [HJ89].

[Knu73a] Donald Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1973.

[Knu73b] Donald Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1973.

[Moo59] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Ski97] Steven S. Skiena. *The Algorithm Design Manual*. Springer, Telos, 1997.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[Wil64] J. W. J. Williams. Heapsort (Algorithm 232). *Communications of the ACM*, 7(6):347–348, 1964.