

# Algorithmen & Datenstrukturen

YASSINE LAKHNECH, MARTIN STEFFEN

Sommersemester 1999

## Lektion I

### Einleitung

**Literatur:** Baut auf Kapitel 1 von [CLR90] auf. Etymologie des Wortes *Algorithmus* aus [Knu73a]

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

Sommer 1999

1 Einleitung

### Aufbau

- Einleitung und Überblick
- Sortieren
- Suchen
- dynamische Datenstrukturen
- Bäume, Graphen
- ...

Sommer 1999

1 Einleitung

### 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)
  - Definiiertheit
  - Terminierung
  - besitzt *Input*<sup>1</sup> und *Output*
  - *effektivness*, jeder Schritt ist effektiv ausführbar
  - zur Lösung eines wohlspezifizieren *Berechnungsproblems*

<sup>1</sup>eventuell leer

## Berechnungsproblem: Sortieren

---

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

4

## Beispiel: Insertion-Sort

---

- Sortieren durch Einfügen
- Imperative Lösung:
  - Eingabe: **Array**  $A$  fester Länge  $n$  von Zahlen
  - Ausgabe: monoton-steigen sortierter **Array**, Permutation von  $A$ .
  - direktes Einfügen ("in situ" sortieren)
- Bild
- Code

5

---

```
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;
```

---

6

## Analyse von Algorithmen

---

- **Analyse:**
  - **Zeitkomplexität:** Anzahl elementarer **Schritte**
  - **Speicherkomplexität:**
  - Sonstiges (Kommunikationsbandbreite ...)
- **Komplexität:** hängt von der **Eingabe** ab
- maßgebend: **asymptotisches** Verhalten ( $O$ -Notation)
  - worst-case, best-case, average-case

7

## Analyse des Sortierens durch Einfügen

- Sei  $t_j$ : Anzahl der *while*-tests für den Wert von  $j$
- Bild aus [CLR90]
- Abhängig vom Input

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

$$T(n) = c_1 + c(n - 1) + c_4(n - 1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j -$$

8

## 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.

9

## 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:  $Merge\_Sort(A, 1, length(A))$

10

```
Merge_Sort(A, p, r)
```

```

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

```

11

## Analyse von Merge-Sort

Merge(A,p,q,r)

```

/* A[p..q] und A[q+1..r] sind sortiert */
/* B Array[1..r-p], d.h. der Länge r-p+1 */

i := p; j:=q+1; k:=1; /* init */

k := 1;
loop exit when i==(q+1) or j==(r+1)
do
  if A[i] <= A[j]
  then B[k] := A[i];
  i := i+1
  else B[k] := A[j];
  j := j+1
  fi;
  k := k+1;
od;

if i==(q+1)
then
  B[k..r-p+1] = A[j..r];
else B[k..r-p+1] = A[i..q];
fi;

return B;

```

12

- rekursiver Algorithmus  $\Rightarrow$  Komplexität durch [Rekurrenzgleichung](#)
- Vereinfachung: Länge =  $2^n$
- im schlimmsten Fall

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

- $T(n) = n \log n$

13

## Lektion II

### Sortieren

**Literatur:** Kapitel 7 und 8 aus [CLR90]. Knuth [Knu73b] enthält mehr über Sortieren als man wissen will.

**Inhalt:** Heapsort · Quicksort

## Heap sort

- **Heap:** vollständiger binärer Baum.<sup>2</sup>
- Bild
- **Heap-Eigenschaft:** Für alle Teilbäume  $t$  gilt:  $t = \text{Node}(l, a, r) \Rightarrow l = \text{Leaf}$  (und  $l = \text{Leaf}$ ) oder  $a \geq \text{Key}(l)$  und  $a \geq \text{Key}(r)$

22. April 1999

<sup>2</sup>Binäre Bäume sind *geordnet*, *vollständig*: entweder Blatt oder genau zwei geordnete Kinder ( $'a \text{ Btree} = \text{Leaf} \mid \text{Node of } ('a \text{ Btree} * 'a * 'a \text{ Btree})$ )

## Heaps als Array

- Implementierung von Heaps als **Array**: Knoten als Arrayelemente
- ⇒ **Zusatzbedingung**: alle Ebenen des Baums **gefüllt** bis auf evtl. die letzte, diese kann von **links nach rechts** teilgefüllt sein.<sup>3</sup>
- Füllung des Arrays von 1 bis *heapsize* (Bild)
  - **Repräsentierung** des Binärbaumes:

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

<sup>3</sup>Länge des Arrays  $l = 2^n - 1$  mit  $2^n - 1 \geq \text{heapsize}$  für minimales  $n$ .

## Einfügen in den Heap

- **Heapify**: Einfügen, Erhalt der **Heapbedingung**
- **rekursive** Prozedur: "Einsickern" des neuen Elementes

```

Heapify (A, i)
  l := Left(i); r := Right(i);

  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
    then exchange A[i], A[largest];
       Heapify(A, largest);

```

## Aufbau des Heaps

- **Bottom-up** mittels Heapify
- Blätter erfüllen **Heap-Eigenschaft** ⇒ Start bei den **untersten nicht-Blatt Knoten**

```

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

```

## Heapsort

- Heap als "teilsortiertes" **Reservoir**
- **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

20

## Quicksort (2)

```

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

let rec qsort (l : int list) = match l with
  [] -> []
| x :: tl ->
  qsort (filter (function y -> y < x) tl)
  @ [x] @
  qsort (filter (function y -> y >= x) tl);;

```

21

## 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);

```

22

## 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**:

$$\begin{aligned}
 \mathbf{I}_1 & \quad A[i'] \leq x \leq A[j'] \quad \text{für alle } i' < i, j' > j. \\
 \mathbf{I}_2 & \quad A[i'] \leq x \leq A[j'] \quad \text{für alle } i' \leq i, j' \geq j.
 \end{aligned}$$

- **Abbruch**:  $(i \geq j) \wedge \mathbf{I}_1$ , d.h.,  $A[p \dots i-1] \leq x \leq A[j+1 \dots r]$ , daraus folgt  $A[p \dots j] \leq x \leq A[j+1 \dots r]$

23

---

```

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

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

```

---

24

---

## Analyse von Quicksort: worst case

---

- **Schlechtester Fall:** Partitionierung trennt den Ausschnitt  $[p \dots r]$  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*

25

---

## Analyse von Quicksort: best case

---

## Lektion III

### Lineares Sortieren

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

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

26

## Einführung

- Bisher: die Algorithmen  $O(n \log n)$  oder  $O(n^2)$
- Gemeinsamkeit:  
Die Ordnung die die Algorithmen bestimmen, beruht *ausschließlich* auf dem *Vergleich* zwischen Elementen

⇒ **Vergleichssortieren**

28

## 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	$O(n \log n)$	$O(n^2)$	$O(n \log n)$

Tabelle 1: Vergleich

29

## Abschätzung für Vergleichssortieren

- Voraussetzung: Sortieren einer Sequenz  $(a_1, \dots, a_n)$  (oBdA: alle Elemente verschieden, wichtig ist nur  $\leq$  oder das Gegenteil.)
- Entscheidungsbaum: Darstellung der Entscheidungen eines Sortieralgorithmusses
  - interner Knoten:  $a_i : a_j$  wobei  $1 \leq i, j \leq n$
  - Blatt: Permutation
- Lauf der Sortierung: Pfad durch den Baum
- Folie: *insertion sort*

29. April 1999

30

## Vergleichssortieren: worst-case

- gegeben: Entscheidungsbaum für einen Algorithmus
  - **worst-case = Tiefe des Baumes**
- ⇒ Abschätzung für die Laufzeit:

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

- Mischsortierung und Heapsort sind *asymptotisch optimal*.

31



## Abschätzung der Laufzeit(2)

### Beweis:

- Es gibt  $\geq n!$  Blätter
  - $n! \leq n^h$ , d.h.  $h \geq \lg(n!)$
- $\Rightarrow n! \geq \left(\frac{n}{e}\right)^n$
- $\Rightarrow h \geq n \lg n - n \lg e$
- $\Rightarrow$  Untere Schranke  $O(n \lg n)$ .

□

32

## 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:  $1 \dots k$  mit  $k = O(n)$ .
- $\Rightarrow$  Zählen der Elemente =  $i$  möglich (im Hilfsarray  $C[1 \dots k]$ )
- kein Vergleich im Algorithmus

33

## 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;

```

34

## 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:  $O(n)$

35

## 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(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;

```

36

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

 $\Rightarrow$ 

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

37

## Lektion IV

### Analyse

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

**Inhalt:** Analyse von Divide and Conquer

## Laufzeitanalyse für Divide & Conquer

- Aufstellen einer **Rekurrenzgleichung**:

$$T(n) = O(1) \quad \text{für } n \leq n_0$$

$$T(n) = \sum_{i=1}^a T(n_i) + D(n) + C(n) \quad \text{für } n > n_0$$

- $T(n)$ : Laufzeit für Problem der Größe  $n$ ,  $D$  und  $C$  Kosten für das **Divide** resp. **Conquer**
- Vereinfachung: Teilprobleme **gleichgroß**,  $n_0 = 1$ . ferner  $f(n) := D(n) + C(n)$ .

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

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

39

## Lösen der Rekurrenzgleichung

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

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

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

40

## Spezialfall: lineare Zeitkomplexität

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

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

41

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

- Summe der Größen der Teilprobleme gleich 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

42

## Spezialfall: polynomiale Laufzeit

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

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

43

---

## Einleitung

---

## Lektion V

### Elementare Datenstrukturen

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

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

- „dynamische Mengen“: veränderbar
- viele verschiedene Variante
- Unterschieden nach
  - Zugriffsmöglichkeiten (Schnittstelle)
  - Repräsentierung
- zwei Arten von Zugriffen
  - modifizierend
  - lesend (*queries*)
- Beispiele für Zugriffe:
  - Suchen
  - Einfügen, Löschen
  - Minimum, Maximum bestimmen
  - Vorläufer, Nachfolger bestimmen
  - ...
- Beispiele für Repräsentierungen:
  - (einfach o. doppelt) verzeigerte Listen
  - Bäume unterschiedlichster Art (balanciert, Heaps ...)
  - Hashstrukturen

---

 45

---

## 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

---

 46

---

## Stack als Array

---

- Implementiert als Array  $1 \dots n$
- dynamischer Inhalt: Elemente  $S[1], \dots, S[top]$
- 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];
```

---

47

## 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;

```

48

## Verzeigerte Listen

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

49

## Verzeigerte Listen (2)

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

50

## 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);

```

51

6. Mai 1999

## 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

52

### Einleitung

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

54

### Adreßtabellen mit direktem Zugriff

- implementiert mit *Arrays*
- Zugriff *direkt* über den *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

⇒ **Hashtabellen**

55

## Hashtabelle

- Verallgemeinerung von Arrays mit direkten Zugriff
  - gegeben: Bereich  $U$  der Schlüssel ("Universum")
  - **hash-Funktion:**  $h : U \rightarrow \{0, \dots, m - 1\}$ 
    - $|U| > m$
- $\Rightarrow h$  nicht injektiv  $\Rightarrow$  Kollision
- $h(k)$ : Hashwert von  $k$
  - "zufällige" Funktion

56

## 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)];`

---

57

## Analyse

**Definition 2 (Belegungsfaktor)** Der *Belegungsfaktor*  $\alpha$  (*load factor*) einer Hashtabelle  $T[0, \dots, m - 1]$  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  $T[h(k)]$

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

58

## Analyse (2)

**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
  - Einfügen am Ende
  - **Ansatz:** Suchen nachdem der Schlüssel eingefügt ist kostet **eins mehr** als davor
- $\Rightarrow$  Durchschnitt:  $\frac{1}{n} \sum_{i=1}^n (1 + \frac{i-1}{m}) = \dots 1 + \frac{\alpha}{2} - \frac{1}{2m}$

$\Rightarrow O(1 + \alpha)$  □

59

## Hashfunktionen

- Ziel: Gleichverteilung von  $h(k)$ :

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

- falls keys gleichverteilt aus  $[0, 1[ \Rightarrow$

$$h(k) = \lfloor km \rfloor \in \{0, \dots, m-1\}$$

- ansonsten: Heuristiken

Divisionsmethode  $h(k) = k \bmod m$

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

60

## 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

$$h : U * \{0, \dots, m-1\} \rightarrow \{0, \dots, m-1\}$$

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

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

- gute Hashfunktion: Vermeidung von Häufungen (clusters)

61

## Offene Adressierung: Hashfunktionen

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

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

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

- Quadratisches Probing
  - $h'$  wie oben

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

- Doppeltes Hashing
  - gute Methode
  - $h_1, h_2$ : zwei gewöhnliche Hashfunktionen

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m$$

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

20. Mai 1999

62

## 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
  - ...

64

## 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**"
- ⇒ Ausgeben der sortierten Elemente = rekursives Durchlaufen des Baumes (*in-order tree walk*)<sup>4</sup>

---

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

---

<sup>4</sup>es gibt daneben Durchlaufen in *Präordnung* und *Postordnung*.

65

## Baumoperationen: Suchen

- wichtige Operation auf Suchbäumen
- Komplexität:  $O(h)$ , wobei  $h$  die **Höhe**
- Minimum/Maximum bestimmen: genauso einfach (der linke/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)
```

66

## 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
```

67

## Baumoperationen: Einfügen

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

---

```

Insert(T,z) =           // T : tree, z : node to insert
y := nil; x := root(T); // zwei Hilfsvariablen
while x != nil
do
  y := x;
  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
then root(T) := z
else if key(z) < key(y)
  then left(y) := z
  else right(y) := z

```

---

68

## Baumoperationen: Löschen

- Komplexität:  $O(h)$
- Problem:
  - Erhalt der Baumeigenschaft: innere Knoten müssen ersetzt werden
  - Erhalt der Sortierungseigenschaft: der Ersatz muß passen
- Bild
- Baumnachfolger eines Knotens  $x$ : Nachfolger in der in-order-Baumordnung
  - falls Knoten mit rechtem Kind:
$$\Rightarrow \text{Nachfolger} = \text{der "linkeste" Knoten im rechten Teilbaum.}$$

$$\Rightarrow \text{Nachfolger hat kein linkes Kind!}$$

$$\Rightarrow \text{key(left(x))} \leq \text{key(x)} \leq \text{key(succ(x))} \leq \text{key(right(x))}^5$$

$$\Rightarrow \text{Nachfolger ersetzt zu löschenden inneren Knoten}$$

---

<sup>5</sup>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.

69

---

```

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 nur ein Kind

if left(y) != nil     // Bestimmung des Kindes x
  then x := left(y)
  else x := right(y);

if x != nil then p(x) = p(y);

if p(y) = nil
  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);

return y;

```

---

70

## 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
- ⇒ je balanzierter desto besser der Baum

72

## Definition

**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
- 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  $2 \lg(n + 1)$ .

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

- Einfügen und Löschen?

73

## Rotation

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

74

## Rotation(2)

```

Rotate-l(t,x) = // t = Baum, x = Knoten
s if (x = nil) or (right(x) = nil)
  then
    raise error "illegal rotation"
  else
    y := right(x); // rette y
    p(y) := p(x);
    if y != nil
    then right(x) := left(y);
    if (left(y)) != nil
    then p(left(y)) := x;

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

    p(x) := y; // zum Schluss

```

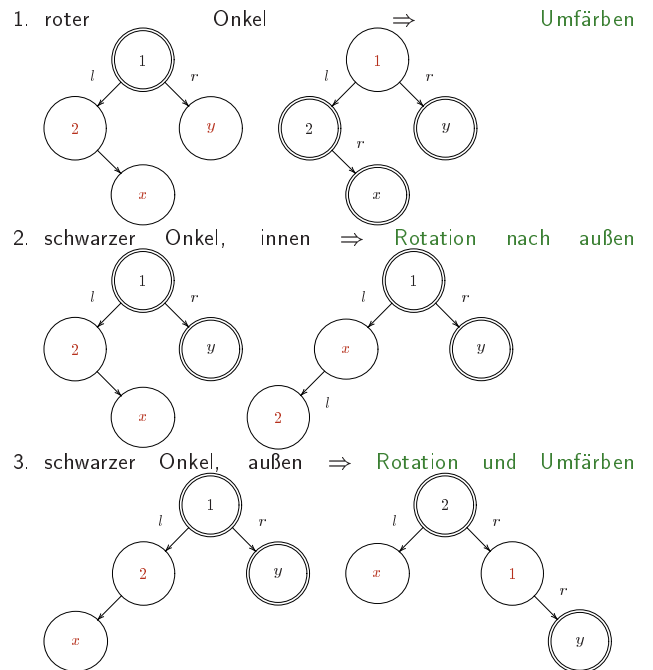
75

## Einfügen

- Zunächst Einfügen wie im gewöhnlichen **binären Suchbaum**
- ⇒ neues **rotes** Blatt.
- keine **Schwarz-Verletzung**, aber u. U.
  - **Rot-Verletzung**
- Bei **rot-Verletzung**:
  - Reparieren durch **Umfärben** und **Rotation**
  - keine Neueinführung von **Schwarzverletzungen**

76

## Einfügen: drei Fälle (+ 3 symmetrische)



77

## Einfügen (3)

```

insert(T,x)
tree_insert(T,x);           // normal einfüegen
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;

```

78

79

---

## Löschen

---

---

27. Mai 1999

---

---

80

---

## Graphen

---

---

82

## Lektion IX

### Graphen

**Literatur:** Kapitel 23 aus [CLR90]. Der Algorithmus für die starken Zusammenhangskomponenten ist von Tarjan [Tar72].

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

---

## Repräsentierung von Graphen

---

---

83

---

## Erreichbarkeitsproblem

---

---

## Suchstrategiere

---

---

## Breiten- & Tiefensuche

---

---

## Breitensuche

---

---

---

## Korrekteitsargument

3. Juni 1999

88

## Tiefensuche

- Suche zunächst in die **Tiefe**<sup>6</sup>
- Wie bei Breitensuche: **Farben** zur Suchsteuerung
  1. Weiß: ungesehen
  2. Grau: entdeckt
  3. Schwarz: fertig
- anstelle von **Queue**: **Stack**
- "Backtracking"
- **Teilgraph der Vorgänger**  $G_\pi = (V, E_\pi)$ :

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

⇒ Menge von **Tiefensuchbäumen** (= **Tiefensuchwald**, *depth-first forest*)

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

<sup>6</sup>der vorgestellte Algorithmus wendet dies auf alle Knoten der Reihe nach an, das ist aber nicht zentral. Die eigentliche Tiefensuche ist *DFS-Visit*.

89

## Algorithmus

```

DFS(G)
  for each vertex v in V[G]
  do
    color[u] := white;
    pred[u] := nil;
  od
  time := 0; // Ende der Initialisierung
  for each vertex u in V[G]
  do
    if color[u] = white then DFS-Visit(u);

DFS-visit(u)
  color[u] := gray; // Knoten entdeckt
  time := time + 1; d[u] := time;
  for each v in Adj[u] // erkunde Kante (u,v)
  do if color[v] = white
    then pred[v] := u
       DFS-visit(v)
  od;
  color[u] := black; // u ist fertig
  time := time + 1; f[u] := time;

```

90

## Tiefensuche: Analyse

- Bild 23.5(1)
  - **Komplexität**:
    - $O(V)$  für *DSF*
    - $O(E)$  für *DFS-Visit*
- ⇒ Laufzeit für Tiefensuche:  $O(E + V)$

91

## Weitere Eigenschaften

**Proposition 7 (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

⇒ Verschachtelung der Intervalle:

**Korollar 8** 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 einem Tiefensuchbaumes, die nicht im Baum sind
  - Querkante: alle anderen

92

## Topologisches Sortieren

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

**Definition 9 (DAG)** Ein gerichteter, azyklischer Graph (DAG) ist ein gerichteter Graph ohne Zyklen, d.h., es für alle Knoten  $u, v \in V$  gilt: Wenn  $u \rightarrow^+ v$ , 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. keine Sortieren wie bei den Sortieralgorithmen
- Bild
- Lineare Ordnung = Zeitstempel der Schwarzfärbung

93

## 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  $O(V + E)$ .

94

## Topologisches Sortieren (3)

**Satz 10 (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.

**Lemma 11** Ein gerichteter Graph ist azyklisch gdw. die Tiefensuche keine Rückwärtskanten produziert.

**Beweis: Fall 1:** ⇐

Sei  $G$  zyklisch ⇒ 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 ⇒ Behauptung

**Fall 2:** ⇒

$DFS$  findet Rückwärtskante  $(v, u)$ , andererseits  $u \rightarrow^* v$  mittels Baumkanten (weil  $u$  Vorgänger von  $v$  im Tiefensuchbaum) □

**Lemma 12 (Korrektheit)** Der Algorithmus angewandt auf einen DAG, liefert eine topologische Sortierung.

**Beweis:** Prüfen der Kompatibilität: falls  $u \rightarrow^+ v$  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 ⇒ Rückwärtskante) ⇒  $v$  ist weiß oder schwarz.

95



**Fall 1:**  $v$  weiß

$v$  wird Nachfahre von  $u \Rightarrow f[v] < f[u]$

**Fall 2:**  $v$  schwarz

Dann sofort  $f[v] < f[u]$  □

17. Juni 1999

96

## Starke Zusammenhangskomponenten

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

**Definition 13 (Starker Zusammenhang)** Gegeben  $G = (V, E)$ . 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: zweimaliges Anwenden von DFS, auf  $G$  und auf den transponierten Graphen<sup>7</sup>  $G^t$
- Komplexität: lineare Zeitkomplexität  $O(E + V)$

<sup>7</sup>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.

97

## Algorithmus

```
Strongly-connected-components(G)    // G: gerichteter Graph
    call DFS(G) to compute finishing times f[u] for each vertex;

    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.
```

98

## Analyse

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

**Lemma 14** Der Vorvater  $\phi(u)$  bzgl. einer Tiefensuche ist ein Vorgänger von  $u$ ,

**Korollar 15** 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. □

99

## Analyse (2)

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

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

⇒

24. Juni 1999

100

## Lektion X

### Spannbäume

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

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

## Minimaler Spannbaum

- Motivation: **Verknüpfung**<sup>8</sup> 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:  $w : V^2 \rightarrow \mathbb{R}^+$ .
- **Gesucht:**  $T \subseteq E$  sodaß
  - **azyklisch** (Baum)
  - **Verbindung** aller Knoten (*aufspannend, spanning*)
  - **minimale Kosten**

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

minimal.

⇒ Problem des **minimalen Spannbaums**

- Beispiel für **greedy-Strategie**
- Bild

<sup>8</sup>jeder von jedem erreichbar.

102

## Spannbaumalgorithmus

- **greedy-Strategie:**
    - **Heuristik** für Optimierungsprobleme
    - falls eine Entscheidung zu treffen ist: entscheide dich für den **augenblicklich beste** Alternative (ohne auf mögliche Nachfolge-Vorteile zu achten)
  - **iterativer** Aufbau eines minimalen Spannbaumes
- ⇒ schrittweises Hinzufügen von Kanten.

**Definition 16** 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 for A
    A := A + {(u,v)}
  od
  return A

```

103

## Sichere Kanten

- **Problem:** wie findet man sichere Kanten?

**Definition 17** Ein **Schnitt**  $(S, V - S)$  (*cut*) eines ungerichteten Graphen ist eine **Partition** von  $V$ . Eine Kante **kreuzt** den Schnitt, if ein Endpunkt ist in  $S$  und der andere in  $V - S$ . Eine Kante, die einen Schnitt kreuzt, ist **leicht** falls ihr Gewicht von allen kreuzenden Kanten **minimal** ist.<sup>9</sup>

**Theorem 18** •  $G = (V, E)$  zusammenhängend, ungerichtet, mit reellwertiger Gewichtsfunktion  $w : E \rightarrow \mathbb{R}$ .

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

⇒  $(u, v)$  **sicher für  $A$** .

<sup>9</sup>man kann

## minimaler-Spannbaum-Algorithmus

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

⇒  $G_A = (V, A)$  **Wald**

- zu Beginn: Wald mit  $|V|$  (einknotigen) Bäumen
- jede hinzugefügte sichere Kante **verknüpft zwei Bäume**

⇒ Schleife  $|V| - 1$ -mal durchlaufen

**Korollar 19** •  $G = (V, E)$  zusammenhängend, ungerichtet, Gewichtsfunktion  $w : E \rightarrow \mathbb{R}$ .

- $A \subseteq E$ ,  $A$  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<sup>1</sup>

- **greedy-Strategie**

- Umsetzung des Korollars 19

⇒ **sichere Kante** = die mit dem **geringsten Gewicht**, die zwei Bäume **verbindet**.

- **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. Spannbaumes
  for each v in V[G] do Make-Set(v) od; // Wald aus lauter Knoten

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

  for each edge (u,v) in E (in order)
  do
    if Find-Set(u) != Find-Set(v) // u,v nicht im selben Baum?
    then A := A + {(u,v)};        // fuege Kante u--v hinzu
      Union(u,v);                 // vergroebere die Partition
    fi;
  od;
  return A;
```

## Kruskal (3)

- **Laufzeit** von Kruskal:
  - hängt von der Implementierung der Hilfsstrukturen ab
 

Initialisierung	$O(V)$
Sortieren	$O(E \log(E))$
innere Schleife	$O(E)$
Datenrepräsentierung	$\alpha(E, V)$
- ⇒  $O(E \log(E))$ <sup>10</sup>

<sup>10</sup>unter der Voraussetzung, daß  $\alpha(E, V) = O(\log E)$ , z.B., heap-Implementierung.

## Prims Algorithmus

- **Spezialisierung** des generischen Algorithmus<sup>7</sup>
  - $A$ : kein Wald, sondern ein einzelner Baum<sup>11</sup>
  - Wachsen des Baumes startend von beliebiger **Wurzel**  $r$
  - Baum bestimmt den **Schnitt**
- ⇒ Iterationsschritt: 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
- ⇒ **Priority Queue** (z.b. wieder mittels Heap)

<sup>11</sup>genauer: alle andern Bäume des Waldes sind nur Einzelknoten.

## Prim (2)

```

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 each u in Q do key[u] := infinity od;

  key[r] := 0;
  p(r) := nil;            // r hat kein parent
  while Q != empty
  do
    u := extract_min(Q);   // u wird den Baum hinzugefügt
    for each v in Adj(u)
    do
      if v in Q and w(u,v) < key(v)
      then p(v) := u;      // parent
           key(v) := w(u,v)
      fi
    od
  od

```

## Lektion XI

### Kürzeste Pfade

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

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

## Einleitung

- Motivation: **Routing**
- gegeben: **gerichteter** Graph mit Gewichtungsfunktion  $w : E \rightarrow \mathbb{R}$
- **Problem**: finde einen Verbindung/Pfad mit **minimalen Kosten**

**Definition 20 (Kürzester Pfad)** Gegeben  $G = (V, E)$  gerichteter Pfad,  $w : E \rightarrow \mathbb{R}$ . Das **Gewicht** eines Pfades  $p = (v_0, v_1, \dots, v_k)$  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  $w(p) = \delta(u, v)$ .

## Varianten

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

## Kürzeste Pfade

- Hauptidee:

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

⇒ gutartiges Problem, viele Techniken anwendbar

- greedy
- Relaxation
- dynamische Programmierung...

**Lemma 21** geg  $G = (V, E)$ , gerichtet, gewichtet mit  $w : E \rightarrow \mathbb{R}$ .  $v_1, \dots, v_n$  ist ein kürzester Pfad von  $v_1$  nach  $v_n$  und  $w = (v_i, \dots, v_j)$  mit  $1 \leq i \leq j \leq n$  ein **Teilpfad**. Dann ist  $w$  ein **kürzester Pfad** von  $v_i$  nach  $v_j$ .

⇒ Zerlegung: Sei  $s \rightarrow^* v$  kürzester Pfad mit  $s \rightarrow^* u \rightarrow v$ , dann  $\delta(s, v) = \delta(s, u) + w(u, v)$ .

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

## 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  $v \in V$ :  $\delta(v)$  als obere Schranke für den Pfad von  $s$  nach  $v$ .

## Relaxation (2)

- **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(v_1, v_2) =
  if d(v_2) > d(v_1) + w(v_1, v_2)
  then pi(v_2) := v_2
      d(v_2) := d(v_1) + w(v_1, v_2) // v
```
- Unterschiedliche Algorithmen, je nachdem wann und wieoft Kanten relaxiert werden.

116

## Dijkstra

- **Voraussetzung:** nicht-negative Gewichtung
  - **Greedy-strategie:** relaxiere immer eine Kante zu dem un-behandelten Knoten mit dem **geringsten** Abschätzung des Abstandes.
- ⇒ Hilfsstruktur: **Priority Queue**
- Graphsuche, gesteuert nach den besten Abschätzungen, Ähnlich **Breitensuche**
  - **Laufzeitkomplexität:**
    - Jeder **Knoten** einmal aus der Queue genommen:  $O(V^2)$ <sup>12</sup>
    - jede **Kante** genau einmal relaxiert:  $O(E)$
- ⇒  $O(V^2 + E) = O(V^2)$ .

<sup>12</sup>extrahieren kostet  $O(V)$ , wenn man nicht noch Zusatzannahmen macht.

117

## Dijkstra (2)

```
Dijkstra(G,w,s)
  Initialise(G,s);          /* siehe Folien          */
  S := emptyset;           /* Knoten mit bekannter Distanz */
  Q := V;                   /* priority queue          */

  while Q != empty
  do
    u := extract-min(Q);
    S := S + {u};
    for each vertext v in Adj(u)
    do
      relax(u,v)
    od
  od
```

118

## Bellmann-Ford

- **beliebige** Kantengewichtung, entdeckt **negativ-gewichtete Zyklen**
- **Relaxation** der Abschätzung des Abstandes.
- **Laufzeitkomplexität:**  $O(VE)$ .

119

## Bellmann-Ford (2)

```

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

initialize(G,s);              // Siehe Vorlesung
for i := 1 to size(V)-1
do
  for each edge (u,v) in E
  do
    relax(u,v)
  od
od;
for each edge (u,v) in E      // Test auf negative Zyklen
do
  if d(v) > d(u) + w(u,v)
  then return false
  else return true
od;

```

120

## Lineare Programmierung

- spezielles Optimierungsproblem
- **Gegeben:**
  - lineare Zielfunktion +
  - Menge von linearen Ungleichungen (*constraints*)
- **Gesucht:** Eingabe, sodaß Zielfunktion mit optimaler/minimaler Wert.
- **genauer:**
  - gegeben
    - \*  $Ax \leq b$
    - \*  $f(x_1 \dots x_n) = \sum_{i=1}^n c_i x_i$
  - gesucht: **minimiere**  $f(x_1 \dots x_n)$
- berühmtes Lösungsverfahren: **Simplex-Verfahren** (und andere)
- Eingeschränkt hier: **Differenzungleichungen**, d.h. Ungleichungen der Form:<sup>13</sup>

$$x_i - x_j < b_k$$

<sup>13</sup>jede Reihe der Matrix enthält eine 1 und eine -1 und ansonsten nur 0.

121

## Darstellung als Graphproblem

- Gegeben: **Differenzungleichungen:**  $Ax \leq b$ .
- Definiere **Constraintgraph**  $G = (V, E)$  mit
  - $V = \{v_1, \dots, v_n\}$  (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.
- 

122

## Lektion XII

### Schluß

**Literatur:** Verschiedenes aus [CLR90] oder ähnlichem. Vorallem Kapitel 36. Dazu Kapitel 11 aus [HU79].

**Inhalt:**

## Harte Probleme

- Bisher nur: **leichte** Problem (Komplexitätsmäßig): **Laufzeit**  $O(n^k)$  (polynomiell, of quadratisch)
- **Techniken für leichte Probleme**
  - **Greedy**: entscheide Dich immer für den augenblicklich Größten Gewinn
  - **Divide-and-Conquer**: zerlege das Problem in unabhängige Teilprobleme und löse diese rekursiv
  - **Relaxation**
  - viele andere:
    - \* dynamische Programmierung: “Verallgemeinerung” von divide-an-conquer
    - \* branch-and-bound
- inhärent **harte Probleme**
  - einfachen Strategien scheitern, keine effizienten Lösungen bekannt
  - oft **Optimierungsprobleme**
  - praktisch relevant: Plazierungen, Scheduling, Kostenminimierung, Probleme aus der Logik, Netzwerkdesign, Zahlentheorie . . .

124

## NP-Vollständigkeit

- berühmte Klasse **schwerer Probleme**
- Status **unbekannt**, wichtiges offenes Problem
  - gibt es eine polynomielle Lösung für derartige Probleme oder
  - gibt es eine nicht-polynomielle untere Abschätzung

$$NP = P \text{ oder } NP \neq P$$

125

## Komplexitätsklasse NP

- Formales Modell: **Turingmaschinen** o.ä.
- Unterscheidung
  - determinisch (DTIME) oder
  - nicht-deterministische (NTIME) Komplexität
- Beispiel:  $DTIME(n)$ : **linear**

**Definition 22 (NP)** Die **Komplexitätsklasse NP** ist die Klasse von Problemen, die **nichtdeterministisch-polynomiell** gelöst werden kann.

- Alternativ:  $NP$  enthält die Probleme, die **polynomiell verifiziert werden können**
- $P = \bigcup DTIME(n^i)$
- $P = \bigcup NTIME(n^i)$
- ähnliche Klasse auch für **Speicherkomplexität** (DSPACE, NSPACE)

126

## Reduzierbarkeit und NP-Vollständigkeit

- **Reduktion**: Vergleich der “Schwere” von Algorithmen
- analog der Reduktion unentscheidbarer Probleme
- hier: **Komplexität** der Übersetzung muß mit berücksichtigt werden.

**Definition 23 (Polynomielle Reduktion)** Ein Problem  $L_1$  (Sprache) ist **reduzierbar in polynomieller Zeit** auf  $L_2$  ( $L_1 \leq_p L_2$ ), wenn es einen in polynomieller Zeit berechenbare Algorithmus (Turingmaschine, . . . )  $f$  gibt sodaß:  $x \in L_1$  gdw.  $f(x) \in L_2$ .

⇒ Definition von **NP-vollständigen** Probleme: die “maximalen” in NP.

**Definition 24 (NP-vollständig)** Eine Problem  $L$  aus der Klasse NP ist **NP-vollständig**, falls sich alles Probleme aus der Klasse NP sich **polynomiell** auf  $L$  reduzieren lassen.

127



## Beispiele

- berechnete Frage: gibt es NP-vollständige Probleme?
  - Erfüllbarkeit von Boolescher Logik/Schaltkreisen
  - Problem des Handlungsreisenden
  - Hamiltonsche Graphen, Max-Clique, . . . . .

## Problem des Handlungsreisenden

- TSP: berühmtes NP-vollständiges Problem
- gegeben:
  - vollständiger Graph (Knoten = "Städte")
  - $c : V \times V \rightarrow \mathbb{Z}$  ("Distanz")
- Gesucht: Teilgraph  $G' = (V', E')$  als Kreis mit  $V' = V$  und minimalen Kosten ("beste Rundreise") (alternativ: gibt es eine Rundreise mit Kosten  $\leq k$ )

---

15. Juli 1999

---

## Literatur

- [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. Leieron, and Ronald L. Rivest. *An Introduction to Algorithms*. MIT Press, 1990.
- [Heu97] Volker Heun. Grundlegende Algorithmen. Vorlesungsskript TU München, October 1997.
- [HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [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.

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