1. Analyse von Algorithmen

Definition: Ein Algorithmus, bekommt irgendwelche Werte und produziert irgendwelche Werte.

Laufzeitfunktionen:

$\log\log n < \log n < log^2 n < \log n! < n\log n < n^2 < (\log n)! < 2^n < n\,2^n < e^n < n! < n^n < 2^{2^n}$

1.1 O-Notation

O → wächst asymp. nicht schneller	$g(n) = O(f(n)) \Leftrightarrow \exists c > 0\ \forall n \geq n_0 : g(n) \leq c * f(n)$
Ω → wächst asymp. nicht langsamer	$g(n) = \Omega(f(n)) \Leftrightarrow O(g(n)) = f(n)$
Θ → wächst asymp. gleich schnell	$g(n) = \Theta(f(n)) \Leftrightarrow g(n) = O(f(n)) \wedge f(n) = O(g(n))$

1.2 Eigenschaften

Transitivität $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \rightarrow f(n) = O(h(n))$ *(ebenso für Ω, Θ)*

Reflexivität $f(n) = O(f(n))$ *(ebenso für Ω, Θ)*

Symmetrie bzw. Transponierte Symmetrie

$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$

1. Aus $a_1 = b_1,\ a_{n+1} = b_{n+1} + c_{n+1}a_n$ folgt

$$a_n = \sum_{i=1}^{n}\left(\prod_{j=i+1}^{n} c_j\right)\cdot b_i$$

2. Aus $a_1 = b_1,\ a_{n+1} = b + c\cdot a_n$ mit $c \neq 1$ folgt

$$a_n = b_1 \cdot c^{n-1} + b\frac{c^{n-1}-1}{c-1}$$

3. Sei $a, b \in N,\ f : N \rightarrow N,\ b > 1$

$T(1) = f(1), T(n) = a\cdot T(n/b) + f(n)$

Dann gilt für $n = b^k$ (also $k = \log_b n = \frac{\log n}{\log b}$)

$$T(n) = \sum_{i=0}^{k} a^i \cdot f\left(n/b^i\right)$$

Beweis: Induktion nach k,

4. Für den Spezialfall $f(n) = c\cdot n$ ergibt sich

$$T(n) = \begin{cases} c\cdot n\cdot(k+1) & \text{für } a = b \\ c\cdot n\cdot\dfrac{(a/b)^{k+1}-1}{(a/b)-1} & \text{für } a \neq b \end{cases}$$

4. ist Spezialfall von 3.

Sei $a,b,c \in N, \quad b>1, \quad$ sei $T(1)=c$

$T(n)=a\cdot T(n/b)+c\cdot n$

mit $n=b^k \Leftrightarrow k=\log_b n = \frac{\log n}{\log b}$

so hat $T(n)$ folgendes Wachstumsverhalten:

$$T(n)=\begin{cases}\Theta(n) & \text{für } a<b\\ \Theta(n\log n) & \text{für } a=b\\ \Theta\left(n^{\frac{\log a}{\log b}}\right) & \text{für } a>b\end{cases}$$

1.3 Master-Theorem

Seien $a \geq 1$und $b > 1$ Konstanten, $f(n)$ eine Funktion und $T(n)$ auf den nichtneg. ganzen Zahlen definiert durch die Rekurrenz $T(n) = a * T\left(\frac{n}{b}\right) + f(n)$ wobei $\frac{n}{b}$ entweder $\left\lfloor \frac{n}{b} \right\rfloor$ oder $\left\lceil \frac{n}{b} \right\rceil$ bedeutet.

	Erster Fall	**Zweiter Fall**	**Dritter Fall**
Allgemein Falls gilt:	$f(n) \in \mathcal{O}\left(n^{\log_b a-\varepsilon}\right)$ für ein $\varepsilon > 0$	$f(n) \in \Theta\left(n^{\log_b a}\right)$	$f(n) \in \Omega\left(n^{\log_b a+\varepsilon}\right)$ für ein $\varepsilon > 0$ und ebenfalls für ein c mit $0 < c < 1$ und alle hinreichend großen n gilt: $af(\frac{n}{b}) \leq cf(n)$
Dann folgt:	$T(n) \in \Theta\left(n^{\log_b a}\right)$	$T(n) \in \Theta\left(n^{\log_b a}\log(n)\right)$	$T(n) \in \Theta(f(n))$
Beispiel	$T(n) = 8T(\frac{n}{2}) + 1000n^2$	$T(n) = 2T(\frac{n}{2}) + 10n$	$T(n) = 2T(\frac{n}{2}) + n^2$
Aus der Formel ist folgendes abzulesen:	$a = 8, b = 2$ $f(n) = 1000n^2$ $\log_b a = \log_2 8 = 3$	$a = 2, b = 2$ $f(n) = 10n$ $\log_b a = \log_2 2 = 1$	$a = 2, b = 2$ $f(n) = n^2$ $\log_b a = \log_2 2 = 1$
1. Bedingung:	$f(n) \in \mathcal{O}\left(n^{\log_b a-\varepsilon}\right)$ für ein $\varepsilon > 0$	$f(n) \in \Theta\left(n^{\log_b a}\right)$	$f(n) \in \Omega\left(n^{\log_b a+\varepsilon}\right)$ für ein $\varepsilon > 0$
Werte einsetzen:	$1000n^2 \in \mathcal{O}\left(n^{3-\varepsilon}\right)$	$10n \in \Theta\left(n^1\right)$	$n^2 \in \Omega\left(n^{1+\varepsilon}\right)$
Wähle $\varepsilon > 0$:	$1000n^2 \in \mathcal{O}\left(n^2\right)$ mit $\varepsilon = 1$ ✓	$10n \in \Theta(n)$ ✓	$n^2 \in \Omega\left(n^2\right)$ mit $\varepsilon = 1$ ✓
2. Bedingung: (nur im 3. Fall)			$af(\frac{n}{b}) \leq cf(n)$ Setze auch hier obige Werte ein: $2(\frac{n}{2})^2 \leq cn^2 \Leftrightarrow \frac{1}{2}n^2 \leq cn^2$ Wähle c = ½: $\forall n \geq 1 : \frac{1}{2}n^2 \leq \frac{1}{2}n^2$ ✓
Damit gilt für die Laufzeitfunktion:	$T(n) \in \Theta(n^3)$	$T(n) \in \Theta(n\log(n))$	$T(n) \in \Theta(n^2)$

2. Suchen

2.1 Binäre Suche

Algorithmus: (S geordnet, a zu suchendes Element)

```
search(unten, oben)
	while (unten < oben und nicht gefunden) do
		next <- upper((unten+oben)/2
		if (a < S(next)) then oben <- next – 1
		elseif (a > S(next)) then unten <- next + 1
		else return next
```

Laufzeit: O(log n)

2.2 Interpolationssuche

Algorithmus: (wie Binär)

$$\texttt{next <- } lower\left(\frac{(a-S[unten])*(oben-unten)}{S[oben]-S[unten]}\right) + unten$$

Laufzeit: O(log log n) (im Mittel) (Wort-case: O(n))

2.3 Quadratische Binärsuche

Algorithmus: (wie Interpolation, am Ende der Schleife mit lineare Suche i bestimmen)

$$A\left(next + \left\lceil (i-1)\sqrt{n}\right\rceil\right) \leq k < A\left(next + \left\lceil i\sqrt{n}\right\rceil\right)$$

Laufzeit: O(log log n) (im Mittel) (Worst-case: O($\sqrt{n}$)

3. Sortieren

3.1 Sortieren durch Auswahl (Selectionsort)

Minimum in finden und tauschen. (in-place, stabil) O(n^2)

3.2 Quicksort

Algorithmus: (Teile und Beherrsche-Prinzip, in-place, instabil)

```
Quicksort(unten, oben)
	i <- unten + 1; j <- oben; k <- S[unten];
	repeat
		while S[i] ≤ k do i++;
		while S[j] > k do j--;
		if i ≤ j then exchange(S[i], S[j]); i++; j++;
	until i > j;
	exchange(S[1], S[j])
	quicksort(unten,j); quicksort(j+1,oben);
```

Laufzeit: O(n log n) (im Worst-Case O(n^2))

Laufzeitbetrachtung: Selbst wenn stets eine 90:10 Aufteilung stattfindet, liegt Quicksort noch in O(n log n), ebenso können komplett unbalancierte Splits vorkommen (sofern nicht häufig hintereinander).

Verbesserungen: Pivotelement (Median of 3, echter Median), für kleine Teilfolgen direktes Verfahren, anstatt rekursiver Aufrufe

3.3 Heapsort

Algorithmus: (In Min-Heap alle Elemente einfügen und dann entnehmen, stabil)

```
h <- Emptyheap;
for i = 1 to n do insert(A[i], h);
for i = 1 to n do B[i] <- extractMin(h);
```

Laufzeit: O(n log n) (insert/del benötigt jeweils log n)

Lemma: Jedes Sortierverfahren, das nur auf dem Vergleich beliebiger Schlüsselwerte basiert, benötigt im schlechtesten Fall mindestens Zeit Ω(n log n).

Beweis: n! Permutationen, daraus folgt

$$n! \leq 2^n \Leftrightarrow h \geq \log(n!) = \Omega(n \log n)$$

Heapsort und Mergesort sind daher asymptotisch optimale Sortierverfahren durch Schlüsselvergleich

3.4 Mergesort

Algorithmus: (stabil, mit Arrays in der Regel out-of-place)

1. Splitten bis es „Runs“ der Länge 1 gibt.
2. Jeweils 2 Runs rekursiv mergen (zusammen sortieren)

Laufzeit: O(n log n)

3.5 Counting-Sort

Algorithmus: (out-of-place, stabil)

```
for i = 0 to k do C[i] = 0
for j = 1 to n do C[A[j]]++;
for i = 1 to k do C[i] = C[i] + C[i - 1];
for j = n downto 1 do
      B[C[A[j]]] = A[j]
      C[A[j]]--;
```

A Inputarray
C Merkt sich die Positionen
B Outputarray

Laufzeit: O(n) (falls k = O(n))

3.6 Radixsort

Algorithmus: (out-of-place, stabil)

1. sortiere nach Ziffer in die Buckets ein
2. füge von vorne nach hinten die Elemente zusammen
3. Wiederhole für die nächste Ziffer

Laufzeit: O(n * m) (m Anzahl der Buckets)

3.7 Bucketsort

Algorithmus: (out-of-place, stabil)

```
for i = 1 to n - 1 make B[i] an empty list
for i = 1 to n do insert A[i] into list B[⌊n * A[i]⌋]
for i = 0 to n - 1 do sort list B[i] with insertion sort
concatenate lists B[0], … B[n - 1] together in order
```

Laufzeit: O(n)

4. Auswählen

Menge S von n Schlüsseln und natürliche Zahl k ($1 \leq k \leq n$). Gesucht ist k-kleinster Schlüssel in S.

4.1 Randomisierte Auswahl

Möglichst gleichmäßige, rekursive Partitionierung der Eingabe

Algorithmus:

```
if n = 1 then return S[1] (gefunden)
r = randomInt aus {1, …, n}
L = {x ∈ S | x ≤ S[r]}
R = { x ∈ S | x > S[r]}
if (k ≤ |L|) then wähle k-kleinstes Element in L
else wähle (k-|L|)-kleinstes Element in R
```

Laufzeit: O(n) (da Partition mit Wahrscheinlichkeit ½ gut ist)

4.2 Deterministische Auswahl

Bestimme Median (Pivotelement per Sampling)

Algorithmus:

1. Unterteile Schlüsselmenge S in $\frac{n}{5}$ Mengen mit je (bis zu 5 Schlüsseln)
2. Finde Median jeder Fünfermenge in O(1) Zeit
3. Wende det. Auswahl rekursiv auf Menge der $\frac{n}{5}$-Fünfer-Mediane an, um x zu finden
4. Partitioniere nun S bezgl. x in L und R (analog zur rand. Auswahl)
5. Wende det. Auswahl rekursiv auf L an, falls $|L| \geq k$ und sonst auf R mit $(k-|L|)$

Laufzeit: O(n)

5. Balancierte Bäume

Speicherungsarten

Knotenorientierte Speicherung: Element im Knoten (AVL-Bäume)
Blattorientierte Speicherung: Elemente nur in Blättern (B*-Bäume)

Operationen: (da nicht balanciert stets O(n))

- suchen, einfügen → trivial
- löschen (von Element u): Fall 1: u ist Blatt; Fall 2: u hat ein Kind; Fall 3: u hat zwei Kinder

5.2 Balancierte Bäume: AVL-Baum

Definition: Ein binärer Baum heißt AVL-Baum, falls $\forall u: |Bal(u)| \leq 1$. (Bal := h(u.right) – h(u.left))

Lemma: Ein AVL-Baum mit n Knoten hat Höhe O(log n).

Balancingoperationen: O(1) (da nur lokal wenige Zeiger umgehängt werden)

Bal(u) = 2,	Bal(v) = 1:	Rotation nach links an u
Bal(u) = -2,	Bal(v) = -1:	Rotation nach rechts an u
Bal(u) = 2,	Bal(v) = -1:	Doppelrotation nach links an u (rechts(v), links(u))
Bal(u) = -2,	Bal(v) = 1:	Doppelrotation nach rechts an u (links(v), rechts(u))

Operationen: (O(log n))

- Einfügen: Rekursiv nach zur Wurzel hin balancieren
- Löschen: Balanciere wie beim Einfügen

Beispiel: Sweepline-Algorithmus (verwendet AVL)

Vertikale „Sweepline“ geht Ebene von links nach rechts durch; jedes aktuell von Sweepline abgedeckte vertikale Segment kann höchstens Schnittpunkte mit gerade „aktiven“ horizontalen Segmenten haben.

5.2 Gewichtsbalancierte Bäume

Definition: Sei T ein binärer Suchbaum mit n Knoten und l Knoten im linken Teilbaum. Dann heißt $p(T) = \frac{l+1}{n+1}$ die Wurzelbalance von T. T heißt von beschränkter Balance α oder ein BB[α]-Baum, falls für jeden Unterbaum T‘ von T gilt: $\alpha \leq p(T') \leq 1 - a$

Höhe: Sei T ein BB[α]-Baum mit n Knoten. Dann gilt: $Höhe(T) \leq \frac{\log(n+1)-1}{\log\left(\frac{1}{1-\alpha}\right)}$

BB[α]-Bäume haben logarithmische Höhe und eine logarithmische mittlere Weglänge.

Algorithmus zur Balancierung: (vi der Weg von Wurzel zur Stelle an der der Baum geändert wurde)

```
for i = k to 0 do
    if p(T_i) ≤ □ then // T' sei rechter Teilbaum
        if p(T') ≤ (1−2α)/(1−α) then Rotation.Links(v_i)
        else DoppelRot.Links(v_i)
    if p(T_i) > 1 − □ then // T' sei linker Teilbaum
        if p(T') > □/(1−α) then Rotation.Rechts(v_i)
        else DoppelRot.Rechts(v_i)
```

Zeitbedarf für suchen, einfügen, löschen O(log n) (n Anzahl Einträge)

5.3 B-Bäume

Definition: Ein B-Baum der Ordnung $k \geq 2$ ist ein Baum,
dessen Blätter alle die gleiche Tiefe haben
die Wurzel mind. 2 Kinder und jeder andere innere Knoten mindestens k Kinder hat
jeder innere Knoten höchstens 2k – 1 Kinder besitzt

Lemma: Sei T ein B-Baum der Ord. k mit Höhe h und n Blättern dann gilt: $2 * k^{h-1} \leq n \leq (2k-1)^h$

Realität: Wähle bei großen Datenmengen k so, dass auf eine Seite (page) des Hauptspeichers ein Knoten des B-Baums passt. (bei Knoten orienterter Speicherung (leere Blätter))

Satz: Zugriff/Einfügen/Streichen kann in B-Bäumen der Ordnung k, welche n Schlüssel verwalten, in Zeit $O(k \log_k n)$ durchgeführt werden.

Alternative Definition: Wie bisher nur, jeder innere Knoten höchstens *2k+1* Kinder besitzt.
➔für die Höhe gilt dann: $log_{(2k+1)}(N+1) - 1 \leq h \leq log_{k+1} \frac{N+1}{2}$ (N gespeich. Elemente)
➔hierbei werden Datensätze dann auch in den Blättern gespeichert.

5.4 B*-Bäume

Definition: Ein B*-baum der Ordnung $e^* \geq 2$ ist ein Baum,
- bei dem alle Einträge in den Blattknoten gespeichert sind,
- jedes Blatt hat mindestens k und höchstens 2k Einträge
- jeder Pfad vom Wurzelknoten zu einem Blatt hat die Länge h*,
- jeder innere Knoten hat mind. k*+1 Söhne (die Wurzel hat mind. 2 oder ist Blatt)
- jeder Knoten hat höchstens 2k*+1 Söhne

Prinzipien:
- möglichst hohes fan-out, um die Zahl der Knotenzugriffe zu minimieren
- Blätter werden durch Zeiger nach links und rechts verbunden (erlaubt schnelle sequentielle Bearbeitung)
- Einschränkungen der Höhe: $log_{(2k^*+1)} \frac{N}{2k} \leq h \leq 1 + log_{k^*+1} \frac{N}{2k}$

Sonstiges:
- Balancierung analog zu B-Bäumen
- Relevant für Datenbanksysteme (meist dann aber nicht streng eingehaltene Balance, da dies meist durch Einfügungen wieder kompensiert wird

5.5 Mengenoperationen mit Suchbäumen

Lemma: Bei Speicherung von Mengen als Suchbäume können die Operationen IstTeilmenge, Vereinigung, Durchschnitt und Differenz in Zeit $O(|S_1|+|S_2|)$ ausgeführt werden.

Anhängen in AVL-Bäumen

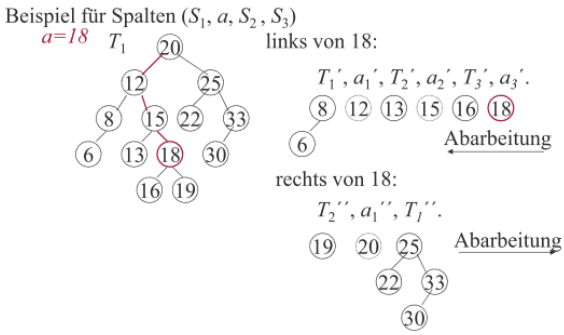
Algorithmus: Anhängen*(T_1, T_2, a, T_3) (Bed: $Höhe(T_1) \geq Höhe(T_2)$)

1. Steige in T_1 ganz rechts ab bis zum ersten Knoten u mit $h_u \leq Höhe(T_2) + 1$
2. Mache a zum Vater von u und T_2. Fall u in T_1 einen Vater v hatte, mache a zum rechten Sohn von v
3. Rebalanciere ab v aufwärts

Spalten von AVL-Bäumen

Algorithmus: (AVL-Baum T an Knoten a spalten)

1. Baum in zwei Listen aus Knoten/Bäumen aufspalten
2. Durch Anhängen* die Knoten von a ausgehend zusammenfügen



5.6 Optimale Suchbäume

Bedingung: Die verwaltete Menge S sei statisch (Wörterbuch), auf die Elemente x aus S werde mit unterschiedlicher Häufigkeit zugegriffen.

Definition: Sei $S = \{x_1,\dots,x_n\}$. Sei β_i die Wahrscheinlichkeit für die Suche nach x_i, und sei α_i die Wahrscheinlichkeit für die Suche nach einem x mit $x_i < x < x_{i+1}$.
Dann heißt $(\alpha, \beta) = (\alpha_0, \beta_1, \dots, \beta_n, \alpha_n)$ die Zugriffswahrscheinlichkeit für die Suche in S.

Lemma:

1. Sei T_{ij} ein opt. Suchbaum für $\{x_i,\dots,x_j\}$ mit Wurzel x_m und linkem bzw. rechtem Teilbaum T_l bzw. T_r und gewichteten Weglängen P_l bzw. P_r. Dann gilt:
   a) $P_{ij} = P_l + P_r + w_{ij}$
   b) T_l ist opt. Suchbaum für $\{x_i,\dots,x_{m-1}\}$
   c) T_r ist opt. Suchbaum für $\{x_{m+1},\dots,x_j\}$
2. Es gibt optimale Suchbäume T_{ij} für $\{x_i,\dots,x_j\}$ mit Wurzel r_{ij}, so dass $i \le r_{i,j-1} \le r_{i,j} \le r_{i+1,j} \le j$

Algorithmus:

```
for i = 0 to n do P_{i+1,i} = 0; w_{i+1,i} = α_i
for k = 0 to n − 1 do
        for i = 1 to n − k do
                j = i + k
                w_ij = w_{i,j-1} + α_j + β_j
                Bestimme m so, dass P_{i,m-1} + P_{m+1,j} minimal  (r_{i,j-1} ≤ m ≤ r_{i+1,j})
                r_ij = m
                P_ij = w_ij + P_{i,m-1} + P_{m+1,j}
```

Laufzeit: $O(n^2)$

Bemerkung: „nahezu optimale“ Suchbäume lassen sich in linearer Zeit berechnen.

5.7 Kantenmarkierte Suchbäume (Tries/Patricia)

Definition: Sei Σ ein festes, endliches Alphabet. Die einzelnen Buchstaben dienen dann zur Orientierung im Suchbaum. Das gesuchte Wort wird blattorientiert gespeichert, wobei der Weg zum Wort von der Wurzel genau das Wort ergibt.

Nachteile:

- Meist sehr dünn besetzt, d.h. fast alle Knoten haben weniger als |Σ| Söhne.
  ➔ aufwendige Speicherung und viele leere Felder
- Es kann lange Wege ohne Verzweigungspunkte geben

Patricia (Practical Algorithm To Retrieve Information Coded in Alphanumerics)

Definition: Binärer Baum, deren linker/rechter Sohn immer der 0/1-Sohn ist. Jeder innere Knoten hat ein Feld Skip, das angibt, wie lang der Weg ist, den dieser Knoten repräsentiert. In den Blättern stehen Schlüssel oder Verweise auf das vollständige Wort.

6. Hashing

6.1 Hashing mit Verkettung

Die i-te Liste ($0 \leq i < m$) enthält alle Elemente x aus S mit h(x) = i.

Worst-Case: Alle Elemente in einer Liste abgespeichert ➔ O(|S|)
Average-Case: $O(1 + \beta)$

Problem: Bestimmung der Größe von m (durch einfügen und streichen kann β zu groß werden)

1. m (zu) groß: hohe Speicherkosten und evtl. erhöhte Laufzeit
2. m (zu) klein: erwartete Kosten (zu) hoch

Lösung durch rehashing

6.2 Hashing mit offener Adressierung

Vorteil: Kein zusätzlicher Speicherbedarf
Nachteil: Schlechte Leistung bei Belegungsfaktor nahe 1 und keine Unterstützung des Streichens.

6.2.1 Lineares Sondieren

Gehe solange weiter ausgehend von der Hashposition, bis ein leeres Feld erreicht wurde. Es entstehen lange Blöcke belegter Positionen, selbst wenn die Schlüssel durch h(x) eigentlich auf versch. Positionen der Hash-Tafel abgebildet werden. (→ stark erhöhte Suchzeit)

6.2.2 Quadratisches Sondieren

Wie linear, nur mit abgewandelter Form: $h(x,i) = (h(x) + i^2) \bmod m$
Vorteil: kein Primary clustering; **Nachteil**: gleiche Positionsfolge (secondary clustering)

6.3 Perfektes Hashing

Beobachtung: Hash-Funktion sollte injektiv sein, falls S bekannt ist. Dadurch konstant O(1) Zugriffszeit anstatt amortisiert. Effektiv nur mit fester Schlüsselmenge möglich eine perfekte Hashfunktion zu bestimmen.

6.3.1 Zweistufiges Hashing

Algorithmus: (Bedingung: $|S| = m \leq n$)

1. Wähle $h \in H_{2,n}$ zufällig. Berechne h(x) für alle $x \in S$.
2. Falls $4m \leq \sum_i |B_i|^2$, dann wiederhole 1.
3. Konstruiere die Mengen B_i für alle $0 \leq i \leq n$.
4. for i = 0 to n -1 do
    a. wähle $h_i \in H_{2,|Bi|^2}$ zufällig
    b. falls h_i auf B_i nicht injektiv ist, wiederhole (a)

Laufzeit: O(n) (da die Wahrscheinlichkeit für das Wiederholen von 1. ≤ ½)

7. Graphenalgorithmen

7.1 Darstellungsformen von Graphen

- Adjazenzmatrix (Start, Ziel) := Kosten
- Adjazenzlisten
  - Out: Start -> Ziel := Kosten
  - In: Ziel -> Start := Kosten

7.2 Topologisches Sortieren

Sei G = (V, E) gerichteter Graph. Abbildung num: V → {1, …, n} mit n = |V| heißt topologische Sortierung von G, falls num(v) < num(w) für alle (v, w) ∈ E. (umgangssprachlich: Pfeile dürfen nur nach rechts zeigen)

Algorithmus:

1. Merke die eingehenden Kanten.
2. Füge einen Knoten der keine eingehenden Kanten besitzt zur Zielmenge hinzu.
3. Lösche seine ausgehenden Kanten. Gehe zu 2.

Laufzeit: O(n + m)

7.3 Kürzeste Wege in Graphen

7.3.1 Single Source Shortest Paths

7.3.1.1 SSSP für azyklische Graphen

Algorithmus:

1. Topologisch sortieren
2. Start bei start + 1 -> n
3. d(start) = 0; rest infinity
4. d(n) = Minimum aller eingehenden Kantenkosten + Wegkosten zum Knoten
5. Wiederhole 4. mit erhöhtem n

Laufzeit: O(n + m)

7.3.1.2 Dijkstra (nur positive Kanten)

Ausgehend vom Start jeweils die nächste kürzeste Strecke festmachen. Daraus resultierende neue Wege merken und vorgehen wiederholen.

Laufzeit: $O(n^2 + m)$ (mit Fibonacci-Heaps auf O(n log n + m) minimierbar)

7.3.1.3 Bellman-Ford (negative Kosten, aber keine negativen Zyklen)

Algorithmus:

1. d(start) = 0; rest infinity
2. wiederhole (3.) n – 1 mal
3. für alle Kanten (u, v) aus E: überprüfe ob mit aktueller Kante der Weg zu v kürzer wird
4. für alle (u, v): wenn d(u) + d(u,v) < d(v), dann negativer Zyklus vorhanden

Laufzeit: O(n * m)

7.3.2 All Pairs Shortest Paths (Floyd-Warshall)

Algorithmus:

```
for k = 1 to n do
      for I = 1 to n do
            for j = 1 to n do
                  d_k(i,j) ←min{d_k-1(i,j), d_k-1(i,k) + d_k-1(k,j)}
```

Laufzeit: $O(n^3)$ (alternativ n-mal Dijkstra: O(n * nm))

7.3.3 All Pairs Shortest Paths (Matrixmultiplikation)

Lemma: Wenn ein Pfad von i über k nach j der kürzeste Pfad von i nach j ist, dann sind auch die Pfade von i nach k und von k nach j jeweils kürzeste Pfade.

Algorithmus: (bereits optimierte Matrixmultiplikation))

```
m = 1
while m < n-1 do
      for i = 1 to n do
            for j = 1 to n do begin
                  l_ij^(new) = ∞
                  for k = 1 to n do
                        l_ij^(new) = min(l_ij^(new), l_ik^(old) + l_kj^(old))
m = 2m
L^(old) = L^(new)
```

Laufzeit: $O(n^3 \log n)$ (Floyd-Warshall schneller!!)

7.4 Depth First Search (Tiefensuche)

Klasseneinteilung der Kante (v,w):

1. Baumkanten T, falls w noch nicht besucht
2. Vorwärtskanten F, falls w schon besucht und es Pfad v $\rightarrow_T^*$ w
3. Rückwärtskanten B, falls w schon besucht und es Pfad w $\rightarrow_T^*$ v
4. Querkanten C, falls w schon besucht, aber weder Vorwärts noch Rückwärtskante

Algorithmus:

```
z1 = 0; z2 = 0;
procedure DFS(v)
      besucht[v] = true; dfsnum[v] = ++z1;
      for all (v,w) ϵ E do
            if !besucht[w] then (v,w) ist Baumkante; DFS(w);
            elseif v →*T w then (v,w) ist Vorwärtskante;
            elseif w →*T v then (v,w) ist Rückwärtskante;
            else (v,w) ist Querkante;
      compnum[v] = ++z2;
```

Laufzeit: O(n + m)

Lemma (Eigenschaften, Teil 1):

a) Kantenklassen T,B,F,C bilden Partition der Kantenmenge E

b) T entspricht dem Wald der rekursiven Aufrufe

c) $v \xrightarrow[T]{*} w$ gdw. $\text{dfsnum}[v] \le \text{dfsnum}[w]$ und $\text{compnum}[w] \le \text{compnum}[v]$.

d) Seien $v,w,z \in V$ mit $v \xrightarrow[T]{*} w, (w,z) \in E$ und $\neg(v \xrightarrow[T]{*} z)$, dann
i) $\text{dfsnum}[z] < \text{dfsnum}[v]$
ii) $(w,z) \in B \cup C$
iii) $\text{compnum}[z] > \text{compnum}[v]$ gdw. $(w,z) \in B$?
iv) $\text{compnum}[z] < \text{compnum}[v]$ gdw. $(w,z) \in C$.

Lemma (Eigenschaften, Teil 2):

Sei $(v,w) \in E$:

e) $(v,w) \in T \cup F \Leftrightarrow \text{dfsnum}[v] \le \text{dfsnum}[w]$.

f) $(v,w) \in B \Leftrightarrow \text{dfsnum}[w] < \text{dfsnum}[v]$ und $\text{compnum}[w] > \text{compnum}[v]$.

g) $(v,w) \in C \Leftrightarrow \text{dfsnum}[w] < \text{dfsnum}[v]$ und $\text{compnum}[w] < \text{compnum}[v]$.

7.4.1 Starke Zusammenhangskomponenten

Ein gerichteter Graph G=(V,E) heißt stark zusammenhängend falls gilt: $\forall v, w \in V: v \rightarrow^* w$.
Die maximalen (bzgl. Mengeninklusion) stark zusammenhängenden Teilgraphen von G heißen starke Zusammenhangskomponenten (SZK's).

Algorithmus: DFS(v)

```
besucht[v] = true; dfsnum[v] = ++z1;
push(v, unfertig); push(wurzeln);
for all (v,w) ∈ E do
        if !besucht[w] then DFS(w)
        elseif w ∈ unfertig then
                while dfsnum[top(wurzeln)] > dfsnum[w] do pop(wurzeln)
compnum[v] = ++z2;
if v = top(wurzeln) then
        repeat w = pop(unfertig); print(w);
        until v = w;
        pop(wurzeln)
```

Laufzeit: O(n + m)
Bezeichnungen:

- SZK K heißt abgeschlossen, wenn alle DFS-Aufrufe für Knoten in K abgeschlossen sind.
- Wurzel von K ist Knoten mit kleinster dfsnum in K
- Folge von Knoten heißt unfertig, falls ihre DFS aufgerufen sind, aber ihre SZK noch nicht abgeschlossen ist. (Sortiert nach dfsnum!)

7.5 Minimal aufspannende Bäume (MST)

7.5.1 Kruskals Greedy-Algorithmus

Algorithmus:

```
d(e1) ≤ … ≤ d(em)
for i = 1 to m do
        if (V, ET mit {ei}) azyklisch) then ET = ET mit {ei}
```

Laufzeit: O(m log m)
Alternative: Repräsentiere jede Menge durch einen Baum (Knoten zeigt auf Eltern, Wurzel enthält Namen der Menge), dazu Pfadkomprimierung ➔ O(n + m * α(m+n,n)) (α Ackermann^{-1})

7.5.2 Prim

Algorithmus: Vorgehensweiße wie Dijkstra, nur dass stets die minimale nächste Kante gewählt wird
Laufzeit: O(m * n²)

8. Mustersuche in Zeichenketten

Zeichenkette (String) T ist endliche Folge von Symbolen aus endlichem Alphabet Σ. Besteht T aus n Symbolen, so werden diese in T[1], …, T[n] gespeichert. T[i..j] ist das an Position i beginnende und an der Position j endende Teilwort von T.

Naiver Algorithmus: (Worst Case: O(m*n))

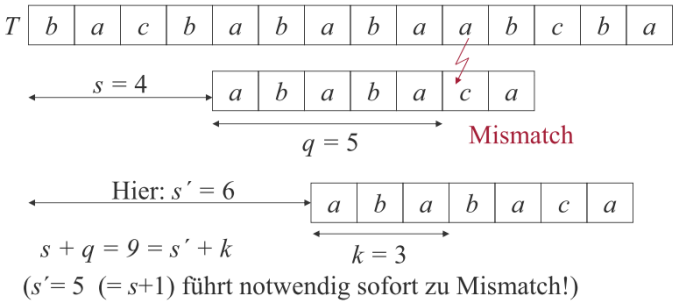
for i = 1 to n – m + 1 do überprüfe, ob P = T[i..i+m-1]

8.1 Algorithmus von Knuth-Morris-Pratt

Algorithmus: Präfixfunktion(P)

```
m = Länge von P;
π[1] = 0; k = 0;
for q = 2 to m do
    while k > 0 and P[k+1] ≠ P[q] do
       k = π[k]
    if P[k+1] == P[q] then k++
    π[q] = k
return π
```

Beispiel:



($s'=5$ $(=s+1)$ führt notwendig sofort zu Mismatch!)

Algorithmus: KMP-Matcher(T,P)

```
n = Länge von T; m = Länge von P;
π = Präfixfunktion(P);
q = 0;
for i = 1 to n do
    while q > 0 and P[q+1] ≠ P[i] do
       q = π[q]
    if P[q+1] == P[i] then q++
    if q == m then print Muster ab Stelle i-m+1
       q = π[q]
```

Laufzeit: O(n+m)

O(m) für berechnen der Präfixfunktion (amortisierte Betrachtung)

Lemma: (für den Korrektheitsbeweis der Präfixfunktion)

1. Sei $\pi^*(q) := \{q, \pi(q), \pi(\pi(q)), \dots\}$. Dann gilt für q = 1, …, m dass $\pi^*(q) = \{k | P[1..k] \; ist \; Suffix \; von \; P[1..q]\}$
2. $\forall \, q = 1, \dots, m: \pi(q) > 0 \rightarrow \pi(q) - 1 \in \pi^*(q-1)$
3. Definiere: $E_{q-1} := \{k | k \in \pi^*(q-1) und \; P[k+1] = P[q]\}$

$$\forall \, q = 2, \dots, m: \pi(q) = \begin{cases} 0, & falls \; E_{q-1} = \emptyset \\ 1 + \max\{k \in E_{q-1}\}, & sonst \end{cases}$$

8.1.2 Der KMP-Algorithmus als DFA

Falls ein festes Pattern häufig geprüft werden soll, so kann dafür ein deterministischer endlicher Automat (DFA, $M_P = (\Sigma, Q, q_0, d, F)$) erstellt werden. (dadurch Laufzeit von garantiert n)

8.2 Der Algorithmus von Boyer-Moore

Besonders vorteilhaft bei großen Alphabeten und relativ langen Mustern.

„schlechter Buchstabe"-Heuristik: Kommt ein Buchstabe nicht im Wort vor so springt man um m, sonst zum ersten Auftreten von hinten im Pattern.

„gutes Suffix"-Heuristik: Springe zum ersten Auftreten des bereits geprüften Suffixes von hinten im Pattern, ansonsten um die volle Länge

Laufzeit:

berechne gutes Suffix O(m)
Boyer-Moore (Worst Case): O((n-m)m+|Σ|)
Boyer-Moore (Average Case): O(n/m) (großes Alphabet und m << n)

9. Sonstiges

Programmiertechniken

- **Divide & Conquer**: Zerlege in disjunkte Teilprobleme und löse diese (Mergesort)
- **dynamisches Programmieren**: Zerlege das Problem in nicht disjunkte Teilprobleme, von denen die kleinsten Teilprobleme direkt gelöst und in eine Tabelle (Matrix) eingetragen werden. Größere Teilprobleme werden aus den Lösungen kleinerer Probleme mit Hilfe der Tabelle/Matrix bestimmt. (optimale Suchbäume)
- **Greedy Algorithmus**: Es wird schrittweise der Folgezustand gewählt, der zu diesem Zeitpunkt den höchsten Gewinn erzielt. (lösen meist schnell aber nicht zwingend optimal) (Dijkstrahl)

Laufzeit-Beweistechniken

- **amortisierte Analyse**: hier wird der Worst-Case ALLER Operationen betrachtet, dadurch fällt eine einzelne Operation die „teuer" ist aber nur einmal Auftritt nicht mehr ins Gewicht
- **Worst-Case**: Man muss vom teuersten Fall ausgehen, auch wenn dieser nur selten vorkommt
  Average-Case: Durchschnittlich übliche Laufzeit
  Best-Case: Kürzeste mögliche Laufzeit