Zusammenfassung: Algorithmen und Datenstrukturen

Wachstum der elementaren Funktionen

Definition *Ordnung*: g ist in Ordnung von $f \rightarrow g$ wächst asymptotisch nicht schneller als f
Schreibweise: $g(n)=O(f(n))$ (g ist in Ordnung von f)
für Konstanten c gibt mit $|g(n)| \le c|f(n)|$ gilt: ist g in Ordnung von f

Transitivität $g(n)=O(f(n))$ genau dann, wenn $\left(\frac{g(n)}{f(n)}\right)$ beschränkt ist

Beschränktheit $\lim_{n\to\infty}\frac{g(n)}{f(n)}=\begin{cases}0 & \rightarrow g(n)=O(f(n)) \text{ bzw. } g(n)<f(n)\\ \infty & \rightarrow f(n)=O(g(n)) \text{ bzw. } f(n)<g(n)\\ \text{sonst} & \rightarrow g(n)=O(f(n)) \text{ und } f(n)=O(g(n))\end{cases}$

Sonderfälle

$\log_a(n)=O(n^k)$ für $k>0, a>1$ $\quad n^k=O(a^k)$ für $k\ge 0, a>1$ $\quad a^n=O(n!)$ für $a>0$
$n!=O(n^n)$
$\log_{10}(n)=O(\log_2(n))$ $\quad \log_2(n)=O(\log_{10}(n))$ $\quad \log(n!)=O(n\log(n))$
$2^{\sqrt{\log_2(\log_2(n))\log_2(n)}}=O\left(\frac{n}{\sqrt{n}}\right)$ $\quad \frac{n}{3\ln(\sqrt{n})}\neq O\left(\frac{n}{\ln^2(n)}\right)$ $\quad 10^n\neq O(2^n)$

Beispiel

$$f_1(n)=\frac{\sqrt{n}}{\log_2(\sqrt{n})}, f_2(n)=\frac{n}{\log_2(n^2)} \rightarrow \frac{f_1(n)}{f_2(n)}=\frac{n^{0.5}\cdot 2\log_2(n)}{0.5\log_2(n)\cdot n}=\frac{4}{n^1-n^{0.5}}=\frac{4}{\sqrt{n}} \xrightarrow[n\to\infty]{} =0 \rightarrow f_1(n)<f_2(n)$$

Rechenregeln falls $g_1=O(f_1)$ und $g_2=O(f_2)$ dann gilt
$g_1+g_2=O(\max(f_{1,}f_2))$
$g_1\cdot g_2=O(f_1\cdot f_2)$
$c\cdot O(f)=O(f)$

Formeln für Differenzengleichungen

Fibonacci Zahlen

$$\begin{cases}x_0=0\\ x_1=1\\ x_n=x_{n-1}+x_{n-2}\end{cases} \quad \text{Lösung: } x_n=\text{round}\left(\frac{g^n}{\sqrt{5}}\right) \quad g=\frac{1+\sqrt{5}}{2}$$

Lineare Differenzengleichungen *erster* Ordnung

„Ausgangssituation“: $\begin{cases}x_1=b\\ x_n=a_n\cdot x_{n-1}+b_n & n\ge 2\end{cases}$
b: Ausgabe des Aufrufs für $n=1$
a_n: Anzahl rekursiver Aufrufe pro Aufruf
b_n: Anzahl konstanter Aufrufe pro Aufruf

(um auf diese Form zu kommen evtl. erst x_n-x_{n-1} berechnen und umformen, siehe 1. Beispiel)

Lösungsformel:

$\pi_i=\prod_{j=2}^{i}a_j \quad i\ge 2,\ \pi_1=1$ Produkt aller Koeffizienten a_j ab zweiten bis zum i-ten

$$x_n=\pi_n\left(b+\sum_{i=2}^{n}\frac{b_i}{\pi_i} \qquad n\ge 1\right)$$

Beispiel:

$x_1=0,\ x_n=\sum_{i=1}^{n-1}(x_i)+n-1 \rightarrow x_{n-1}=\sum_{i=1}^{n-2}(x_i)+n-2 \rightarrow x_n-x_{n-1}=x_{n-1}+1 \Leftrightarrow x_n=2x_{n-1}+1$

$\rightarrow a_n=2,\ b=0,\ b_i=1,\ \pi_i=2^{i-1},\ \pi_n=2^{n-1}$

$$\rightarrow x_n=2^{n-1}\cdot\left(0+\sum_{i=2}^{n}\frac{1}{2^{i-1}}\right)=2^{n-1}\cdot\sum_{i=2}^{n}2^{1-i}=\underbrace{\sum_{i=2}^{n}2^{n-i}=\sum_{i=0}^{n-2}2^i}_{2^{n-2}+2^{n-3}+\ldots+2^{n-n}=2^0+2^1+\ldots+2^{n-2}}=\frac{2^{n+1-2}-1}{2-1}=2^{n-1}-1$$

Weiteres Beispiel:

$x_0=0,\ x_n=2\cdot x_{n-1}+3^n \rightarrow \pi_i=2^i,\ \pi_n=2^n$

$$\rightarrow x_n=2^n\cdot\left(3+\sum_{i=2}^{n}\frac{3^i}{2^i}\right)=2^n\cdot\left(\sum_{i=1}^{n}\frac{3^i}{2^i}\right)=2^n\cdot\left(\sum_{i=1}^{n}\left(\frac{3}{2}\right)^i\right)=2^n\cdot\left(\frac{1.5^{n+1}-1}{1.5-1}-1\right)=\ldots=3\cdot(3^n-2^n)$$

Nützliche Formeln:

$$\sum_{i=m}^{n} x^i = \frac{x^{n+1}-x^m}{x-1} \quad \sum_{i=0}^{n} i\cdot x^{i-1} = \frac{(n+1)x^n(x-1)-(x^{n+1}-1)}{(x-1)^2} \quad \sum_{i=0}^{n} i\cdot x^i = \frac{(n+1)x^{n+1}(x-1)-x(x^{n+1}-1)}{(x-1)^2}$$

$$H_n := \sum_{i=1}^{n} \frac{1}{i} \approx \ln(n) + \underbrace{\gamma}_{0.5772156649} + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \underbrace{\epsilon}_{0<\epsilon<\frac{1}{252n^6}} \quad \sum_{i=1}^{n} H_i = (n+1)H_n - n \quad e^x := \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

$$\sum_{i=1}^{n} 1 = n \qquad \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \qquad \sum_{i=1}^{n} 2i = n(n+1)$$

$$\sum_{i=1}^{n} (2i-1) = n^2 \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \qquad \sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2 \qquad \frac{1}{2}\binom{n}{2} = \frac{n(n+1)}{4}$$

$$\sum_{i=1}^{n} i\,2^{n-1} = 2^{n+1} - n - 2 \quad \sum_{i=0}^{n} a^k b^{n-k} = \frac{a^{n+k}-b^{n+1}}{a-b} \quad \sum_{i=1}^{n} 2^{i-1} = 2^n - 1 \quad \sum_{i=1}^{n} 2^i = 2^{n+1} - 2$$

Nützliche Umformungen: $\sum_{i=1}^{n-1} (n-i) = \sum_{i=1}^{n-1} i \quad \sum_{i=2}^{n} \left(\frac{i}{2^{i-1}}\right) = 2\cdot\sum_{i=2}^{n} \left(\frac{i}{2^i}\right) = 2\cdot\sum_{i=0}^{n} \left(i\cdot\left(\frac{1}{2}\right)^i\right) - 1$

Partialbruchzerlegung: $\frac{i}{(i+1)(i+2)} = \frac{A}{i+1} + \frac{B}{i+2} \Leftrightarrow i = (A+B)i + 2A + B$

dabei muss gelten: $A+B=1$ und $2A+B=0$ (damit $i=i$) $\rightarrow A=-1$ und $B=2$

Sortieren

- Motivation: In sortierten Arrays kann binär gesucht werden.
- Selection Sort: Sortieren durch Auswahl
- Insertion Sort: Sortieren durch Einfügen
  - bei Index 2 Anfangen → jede Zahl so lange mit linken Nachbarn vertauschen, wie sie kleiner ist
  - Laufzeit: Best Case: $O(n)$ (schon sortiert)
    Worst Case: $O(n^2)$ (umgekehrt)
- Bubblesort: Sortieren durch Austauschen
  - Liste durchlaufen bis $i = n-1$, Element mit nächstem vergleichen und tauschen, wenn kleiner (größtes wandert nach rechts)
  - nächster Durchlauf mit $i = n-2$, … bis $i = 1$
  - Laufzeit: Best Case: $O(n)$ (vorsortiert, wenn Vertauschungen ausgewertet werden, sonst wie Worst Case)
    Worst Case: $O(n^2)$ (umgekehrt oder ohne merken $(n^2-n)/2$)
    Average Case: $O(n^2) \rightarrow (n^2-n)/4$
- Quicksort: Divide and Conquer Strategie
  - Pivot Element P wählen → an stelle P Folge F in F1 und F2 teilen
  - alle in F1 < P und F2 > P
  - Quicksort rekursiv für F1 und F2 aufrufen
  - geeignet nur für hinreichend große Folgen
  - Laufzeit: Best Case: $O(n\log_2 n) \rightarrow b\cdot 2^{c\lfloor\log_2 n\rfloor} + c\lfloor\log_2 n\rfloor n$
    Worst Case: $O(n^2) \rightarrow c/2\cdot n^2 + (c/2+b)n - c$
    Vergleiche (Durchschnitt): $2(n+1)H_n - 4n \rightarrow O(n\log n)$
    Umstellungen (Durchschnitt): $(n+4)/6$
    Avg. Case: $O(n\ln n) \rightarrow 2c(n+1)H_{n+1} + 1/3(2b-10c)n + 1/3(2b-7c)$
    Stacktiefe (Best Case): $\leq\lfloor\log_2 n\rfloor + 1$
    Stacktiefe (Worst Case): $= n$
  - probabilistischer Quicksort: Pivotelement wird zufällig gewählt → Wahrscheinlichkeit für Worst Case 1 / $n!$ → Laufzeit wie Average Case
- Heapsort
  - Sortieren durch Auswählen, aber mit Hilfe eines binären Heaps
  - Array wird über einen Baum gelegt
  - Heapbedingung (für Mini-Heap): Wurzel ist kleiner als Nachfolger → entlang eines Pfades ist Heap sortiert
  - erste Phase: Heap aufbauen
    - rechts → links / unten → oben: d. h. rechter Teilbaum, linker Teilbaum, Wurzel
    - Wurzel: $a[i]$, Kind-Links: $a[2i]$, Kind Rechts: $a[2i+1]$, Vater: $a[\text{floor}(i/2)]$
  - zweite Phase: Sortieren
    - Minimum steht an $a[1]$ (Root)
    - erstes und letztes Element vertauschen und nur noch $n-1$ betrachten

- DownHeap anwenden und wiederholen
- Array wird in umgekehrter Reihenfolge sortiert

◦ Laufzeit: Iterationen in DownHeap (Worst Case): $\lfloor \log_2(r/l) \rfloor$
BuildHeap (Worst Case): $\approx 1.22 \cdot n \rightarrow O(n)$
Mit Sort (Worst Case): $O(n \log_2 n)$

Suchen

Im Baum: Suche in Bäumen mit n Blättern

Länge eines Pfades von Wurzel zum Blatt: $\log_2 n$

Anzahl der Vergleiche: $\frac{n}{2}(\log_2(n)-1)$

Im sortierten Array (binär): $\lfloor \log_2 n \rfloor + 1 \rightarrow O(\log_2 n)$ Quick Select: $4cn \rightarrow O(n)$

Für das Eindeutigkeitsproblem gibt es einen Algorithmus mit der Laufzeit $O(n)$.

Hashverfahren

sollen effizient sein und Kollisionen vermeiden

X: Menge der Schlüssel
m: Anzahl der Zellen/Slots (im Primärbereich)
h: Hashfunktion $X \rightarrow \{1, \ldots, m\}$
B: Belegungsfaktor $B = n/m$

H: Hashtabelle
n: Anzahl der Elemente in Hashtabelle (inkl. Überlaufbereich)
$h(s)$: berechneter Tabellenindex

Anzahl der Vergleiche im Mittel: $V = 1 + B/2$ (Annahme: Hashfunktion zufällig gewählt)

Division: $s \rightarrow s \bmod m$

Multiplikation: $s \rightarrow \lfloor m\{s \cdot c\} \rfloor$ (Ganzzahliger Anteil von m mal gebrochener Anteil von $s\,c$)

Universelle Familien

Liefern ähnlich niedrige Kollisionswahrscheinlichkeiten wie Zufallsfunktionen, sind aber effizient implementierbar → Hashverfahren wird zu probabilistischem Verfahren

Familie von Funktionen heißt universelle Familie, wenn für jede Funktion die Kollisionswahrscheinlichkeit kleinergleich (1 / m) ist.

Verfahren

1. Hashfunktion $h \in H$ zufällig wählen und über Verfahren beibehalten
2. Primzahl p, Zahlen: $a, b \in \mathbb{Z}_p, \quad 2 \le m \le p$
3. $h: x \rightarrow ((ax+b) \bmod p) \bmod m$

Kollisionsauflösung

Hashverfahren mit Primär- und Überlaufbereich

- Schlüssel, die auf den selben Hashwert abgebildet werden, werden als Liste verkettet
- **Überlaufbereich**: doppelte Schlüssel können im Überlaufbereich der Tabelle gespeichert werden; Schlüssel im Primärbereich haben zusätzlich Index, der auf Schlüssel im Überlaufbereich verweist
- **separate Verkettung**: Tabelle nur Anker für verkettete Listen → kann zu schlechterem Laufzeitverhalten führen

Offene Adressierung

- im Falle einer Kollision wird Ersatzplatz in Tabelle gesucht
- Adresse steht nicht von Anfang an fest → Sondierfolge wird verfolgt, jeder Schlüssel hat seine eindeutige Sondierfolge
- jede Sondierfolge sollte auf alle möglichen Plätze der Hashtabelle abbilden
- Suchen: verfolge Sondierfolge bis s gefunden wird
- Einfügen: verfolge Sondierfolge bis Zelle frei ist
- Löschen: suche s und kennzeichne als gelöscht/frei

Sondieren (Suchen eines alternativen Indexes im Falle einer Kollision)

m: Anzahl der Zellen/Slots j: aktueller Sondierindex, initial 0, wird bei Kollision inkrementiert

- **uniformes Hashing** h und h' unabhängig wählen ($i = 0, \ldots, m-1$)

  mittlere Länge der Sondierfolge beim Suchen eines Elements: $\frac{1}{B} \cdot \ln\left(\frac{1}{1-B}\right)$

  mittlere Länge der Sondierfolge beim Einfügen eines Elements: $\frac{1}{1-B}$

- **lineares Sondieren** $i(s)_j = (h(s) \pm j^2) \bmod m$

  mittlere Länge der Sondierfolge beim Suchen eines Elements: $\frac{1}{2} \cdot \left(1 + \frac{1}{1-B}\right)$

mittlere Länge der Sondierfolge beim Einfügen eines Elements: $\frac{1}{2}\cdot\left(1+\left(\frac{1}{1-B}\right)^2\right)$

- **quadratisches Sondieren** $i(s)_j=(h(s)\pm j^2)\bmod m$
  wenn alle Zellen erreicht werden sollen, muss gelten
  $i(s)_j=(h(s)\pm j^2 \bmod p)\bmod m$ mit $p=3 \bmod 4$
- **Doppelhashing** $i(x)=(h(x)+h'(x)\cdot j) \bmod m$
  alle Zellen werden erreicht, wenn gilt $h'(s)$ ist teilerfremd zu m

Beispiel
n = 12000 Datensätze sollen verwaltet werden
Im Mittel soll ein Datensatz mit V = 2 zugriffen gefunden werden.

Hashverfahren mit Primär- und Überlaufbereich
Belegungsfaktor: $1+B/2=V=2\Leftrightarrow B=2$ Primärbereich: $m=n/B=12000/2=6000$
Überlaufbereich: $p_0=e^{-B}$, $E(\text{Kol})=n-m(1-p_0)\approx 6812.011699$

Hashtabelle für Uniformes Hashing
Belegungsfaktor: $\bar{L}=2=\frac{1}{B}\ln\left(\frac{1}{1-B}\right)\Leftrightarrow B\approx 0.8$ → Mindestgröße: $m=\frac{12000}{0.8}=15000$

Beispiel zum Multiplikationsverfahren
ges.: Hashwert für 2^k mit k von 0 bis 10 geg.: m = 2048, c = 0.618, 16-Bit Wortbreite
→ 11-Bit-Hashwerte, da $m = 2048 = 2^{11}$
→ $h(s)=\lfloor 2048\cdot\{2^k\cdot 0.618\}\rfloor$ wobei $\{2^k\cdot 0.618\}$ gebrochener Anteil von $2^k\cdot 0.618$
$k=0:\lfloor 2048\cdot\{2^0\cdot 0.618\}\rfloor=2165$
$k=1:\lfloor 2048\cdot\{2^1\cdot 0.618\}\rfloor=283$ …

Beispiel zum linearen Sondieren Sondierfolge: $i(s)_j:=(s+j\cdot c) \bmod m$
für c = 7 und m = 21: $i(261)_0 = 9$, $i(321)_0 = 6$, $i(781)_0 = 4$, $i(109)_1 = 4$, $i(800)_0 = 3$, $i(235)_2 = 18$

- beim Einfüge von 109 und 235 Kollision → j wird inkrementiert; bei nächstem Einfügen j wieder auf 0 setzen
- c sollte teilerfremd zu m sein, damit alle Plätze in der Hash-Tabelle sondiert werden (hier z. B. 5)
  Begründung: angenommen: $i\cdot c\equiv j\cdot c \bmod m \rightarrow m \text{ teilt } (i-j)\cdot c$
  $\gcd(m,c)=1\rightarrow m \text{ teilt } (i-j)$, ein Widerspruch, da $|i-j|<m$

Beispiel zum Multiplikationsverfahren mit quadratischen Sondieren
geg.: 8-Bit Schlüssel, 6-Bit Hashwerte, Multiplikation mit c = 0.608, p > max. Hashwert aber kleinste Zahl, sodass alle Indizes der Tabelle in jeder Sondierfolge auftreten
→ Hashwerte für Schlüssel 4, 5, …: $h(4)=\lfloor 64\cdot\{0.608\cdot 4\}\rfloor=27, h(5)=\lfloor 64\cdot\{0.608\cdot 5\}\rfloor=\ldots$
p muss Primzahl sein, die kongruent 3 modulo 4 ist und größer gleich 63 ist, da größter Hashwert $2^6 - 1 = 63$ → p = 67 (Diese Kongruenzbedingung muss immer gelten bei quadratischen Sondieren wenn alle Werte auftreten sollen.)
→ Formel für Sondieren: $i(s)_j=(\lfloor 64\cdot\{0.608\cdot s\}\rfloor\pm j^2)\bmod 67$ für $j=0,\ldots,p-1$
(mit Plus anfangen, dann Minus versuchen, dann Index erhöhen, …)

Analyse und weitere Formeln

- Verkettungen: für Dimensionierung Anzahl der Kollisionen entscheidend
  offene Adressierung: Länge der Sondierfolgen entscheidend
- Wahrscheinlichkeit für *keine* Kollision bei n belegten Zellen: $p_0=(1-1/m)^n$
- Wahrscheinlichkeit, dass auf einen Wert i Schlüssel abgebildet werden: $p_i=\binom{n}{i}\cdot\left(\frac{1}{m}\right)^i\left(1-\frac{1}{m}\right)^{n-i}$
- Erwartungswert für Kollisionen: $E(\text{Koll})=n-m(1-p_0)$
- Anzahl Vergleiche im Mittel bei *erfolgreicher* Suche: $1+0.5\cdot B=2$
- Anzahl Vergleiche im Mittel bei *erfolgloser* Suche: $B+e^{-B}$
- Kollisionswahrscheinlichkeit für Zufallsfunktionen: 1 / m
- Wahrscheinlichkeit für Sondierfolge sl_n der Länge r: $p_r=\binom{m-r}{n-n-1}/\binom{m}{n}$

Bäume

Allg. Definition von Wurzelbäumen: $B=(V,E,r)$ wobei V:Knoten, E:Kanten, r:Wurzel
Höhe eines Knotens v: Länge aller Pfade von v; Tiefe: Länge des Pfads zur Wurzel
(leerer Raum hat Höhe -1; nur Wurzel hat Höhe 0)

max. Anzahl von Knoten: $\frac{d^{h+1}-1}{d-1}$ wobei d:max. Nachfolger der Wurzel , h:Höhe

Binäre Suchbäume

Elemente im linken Teilbaum kleiner als Wurzel; Elemente im rechten Teilbaum größer

Laufzeit: Vergleiche: $O(\log n)$ (Average Case) bzw. $O(n)$ (Worst Case, Baum ist Liste)
Anzahl der Ebenen: $\lceil \log_2 n \rceil$ wobei *n*: Anzahl der Elemente
Höhe: $\log_2 n$ (Best Case) bzw. n (Worst Case)
Knoten im Suchpfad (Average Case): $P(n)=2\cdot(n+1)/n\cdot H_n-3$

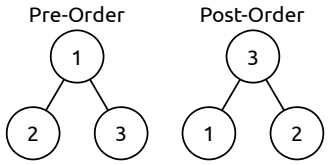
Operationen

Einfügen: Element an Stelle anhängen, sodass linker Teilbaum kleiner als rechter Teilbaum ist
Löschen:

- wenn Blatt: trivial
- wenn nur ein Nachfolger: entferne Knoten aus verketteter Liste
- sonst: Lücke mit vorherigem Element füllen; für vorheriges Element (auch *Symmetrischer Vorgänger*) einmal nach links, dann immer nach rechts bis Blatt

Traversierung

- pre-order: Wurzel, linker Teilbaum, rechter Teilbaum
- post-order: linker Teilbaum, rechter Teilbaum, Wurzel
- in-order: linker Teilbaum, Wurzel, rechter Teilbaum
  → Reihenfolge der Schlüssel
- reverse in-order: rechter Teilbaum, Wurzel, linker Teilbaum
  → umgekehrte Reihenfolge der Schlüssel



Alle Binären Suchbäume für bestimmte Menge an Elementen

Bäume erstellen, sodass jedes Element einmal Wurzel ist

Prüfen, ob Reihenfolge beim Pre-Order-Traversieren möglich ist

einfach Baum in Reihenfolge der Elemente aufbauen und Bedingungen prüfen, z. B.
e, b, c, a: nicht möglich, da a < b
e, b, a, c, d, i, j, f: nicht möglich, da f < i

Baum aus Pre-Order- und Post-Order-Ausgabe ableiten

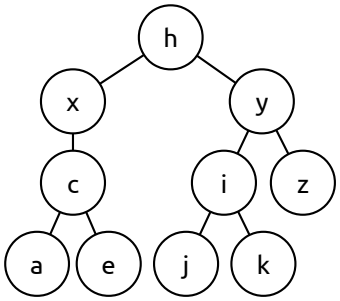
l1 := Pre-Order-Ausgabe, *l2* := Post-Order-Ausgabe
Wurzel: *l1*[0] bzw. *l2*[-1]
linker Nachfolger der Wurzel (wenn vorhanden): *x* := *l1*[1]
rechter Nachfolger der Wurzel (wenn vorhanden): *y* := *l2*[-2]
linker Teilbaum: alle Elemente in *l2* links von *x*
rechter Teilbaum: alle Elemente in *l1* rechts von *y*

Bemerkungen

- falls Knoten nur einen Nachfolger besitzt, ist Baum nicht eindeutig bestimmt
- bei diesen Aufgaben ist Sortierreihenfolge evtl. egal

Beispiel

preorder: h x c a e y i j k z
postorder: a e c x j k i z y h



B-Bäume

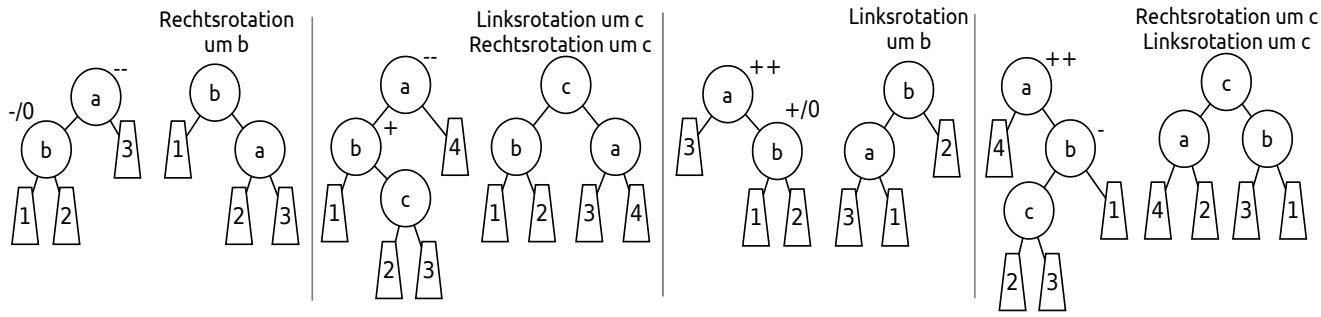
- speichern auf externen Speichermedien mit quasi wahlfreiem Zugriff
- Ziel: Anzahl der Zugriffe auf externes Speichermedium minimieren
- Baum mit Ordnung *O*: jeder Knoten hat höchstens *O* Nachfolger
- jeder Knoten (außer Wurzel) hat min. $\lceil O/2 \rceil = \lfloor (O-1)/2 \rfloor + 1$ Nachfolger
- Wurzel hat min. 2 Nachfolger (wenn sie kein Blatt ist)
- alle Blätter liegen auf einer Ebene
- jede Seite enthält max. $O-1$ Elemente (sortiert gespeichert)
- für Höhe *h* eines Baumes gilt: $\log_O(n+1)-1 \le h \le \log_{\lfloor (n-1)/2 \rfloor+1}((n+1)/2)$
- minimale bzw. maximale Anzahl der Knoten *k*: $1+\frac{\lceil O/2 \rceil^h-1}{\lceil O/2 \rceil-1}\cdot 2 \le k \le \frac{O^{h+1}}{O-1}$
- Ausnutzung pro Seite in B*-Baum: 2 / 3
- größtes und das kleinstes Element stehen in einem Blatt
- Suchen: wie üblich; gibt Page und Index zurück
- Einfügen: Blatt ermitteln, das *e* speichern soll
  - noch Platz: einfügen
  - voll: Seite bei mittlerem Element teilen und mittleres Element oben sortiert einfügen
  - bei B*-Bäumen ggf. noch ausgleichen
- Löschen: wg. Adressen = Elemente + 1 kann Element nur aus Blatt entfernt werden

- wenn Element kein Blatt, mit Nachfolger in Sortierreihenfolge (immer Blatt) vertauschen und dann löschen
- auf Underflow prüfen: Knoten hat weniger als $\lfloor (O-1)/2 \rfloor$ Elemente
  → zunächst mit Elementen der gleichen Seite austauschen, wenn nicht möglich muss zusammengefasst werden

Ausgeglichene Bäume (AVL-Bäume)

- Versuch Höhe nahe $\log_2(n)$ zu halten
- Höhe höchstens 45 % höher als im optimalen Fall: $h<1.45\log_2(n+2)-1.33$
- Höhe des rechten und linken Teilbaums jedes Knotens unterscheiden sich höchstens um eins
- Balacefaktor: Höhe rechter Teilbaum minus Höhe linker Teilbaum

Ausgleichen durch Rotation (Links immer Ausgangssituation, rechts Ausgleich)

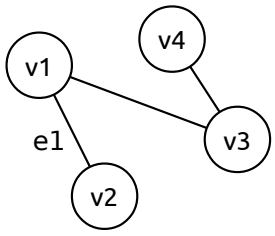


Probabilistische binäre Suchbäume (Treaps)

- zufällige Wahl von Baumelementen, um weniger Streuung zwischen Worst/Best Case zu haben
- Überlagerung von binärem Suchbaum mit Heap (deshalb Treap)
- Treapelemente bestehen aus Schlüssel *k* und Priorität *p*: $e=(k,p)$
- Elemente mit geringster Priorität (wird zufällig Gewählt) werden Wurzel
- sowohl Heap als auch Baumeigenschaft muss erfüllt sein (Schlüssel links sind kleiner als Schlüssel rechts, Priorität des Vaters ist kleiner als die der Kinder)
- Einfügen: Knoten wie normal anhängen, anschließend rotieren bis Heapbedingung erfüllt ist
  Es entsteht – unabhängig von Einfügereihenfolge – immer der selbe Treap.
  → Treaps, welche geordnete Menge speichern, sind eindeutig bestimmt.
- Löschen: Knoten so lange nach unten rotieren, bis Blatt; dann einfach löschen
  (hat Knoten 2 Nachfolger, beim nach unten rotieren den mit kleiner Priorität wählen, damit Heap-Eigenschaft nicht verletzt wird)
- Anzahl der Knoten im Pfad (Erwartungswert): $P(n)=2\cdot(n+1)/n\cdot H_n-3$
- Anzahl der Rotationen (Erwartungswert): $E(r)<2$

Graphen

Definition: $G=(V,E)$
V: Menge der Knoten, hier *V* = {v1, v2, v3, v4}
E: Menge der Kanten, hier *E* = { {v1, v2}, {v1, v3}, {v3, v4} }
Adjazente Knoten: benachbarte Knoten
Inzidente Kanten: verbundene Kanten, hier e1 inzident zu v1 und v2
Grad eines Knotens: Summe ein- und ausgehender Kanten
(Anzahl Knoten ungeraden Grades immer gerade)
Knotenfolge: z. B. v1, v2, v4 (Knoten müssen nicht verbunden sein)
Pfad/Weg: Knotenfolge *adjazenter* Knoten, z. B. v1, v3, v4
Länge des Wegs: Anzahl der durchlaufenden Kanten
einfacher Weg: kein Knoten kommt mehrfach vor
Zyklus/geschlossener Pfad: Weg, der im selben Knoten beginnt und endet
starke Zusammenhangskomponente: gegenseitig erreichbare Knoten (*Zyklus*)
min. Anzahl der Kanten in zusammenhängenden Graph: $|V|-1$
max. Anzahl der Kanten in *ungerichteten* Graph: $0\le|E|\le\binom{|V|}{2}$
max. Anzahl der Kanten in *gerichteten* Graph: $0\le|E|\le|V|(|V|-1)$



Adjazenz Matrix:

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Probleme

House-Utility-Problem: Graph, bei dem es keine Überkreuzung geben darf → Ebender Graph nur bis 4 Knoten möglich
Abstandsprobleme: kürzester Weg zwischen 2 Knoten
Chinese Problem: kürzester Weg zwischen 2 Knoten, der jede Kante mindestens 1 mal besucht
Travelling Salesman: kürzester Weg, der alle Knoten besucht

kritischer Pfad: längste Abhängigkeitskette in gerichtetem Graph

Breitensuche bzw. Breadth First Search (BFS)

V_T: betrachtete Knoten, initial leer
V_{ad}: unbesuchte Knoten, die mit Knoten in V_T benachbart sind, initial nur Startknoten
V_R: noch unendeckte Nachbarn des Startknotens, initial alle Knoten außer Startknoten

Algorithmus:
Für jeden Eintrag k in V_{ad}: verschiebe k nach V_T; verschiebe alle Nachbarn von k die noch in V_R sind nach V_{ad}
Wiederhole bis alle Knoten (der Zusammenfassungskomponente) in V_T sind

Ordnung (Adjazenzliste): $T(|V|,|E|)=O(|V|+|E|)$ Ordnung (Adjazenzmatrix): $T(|V|,|E|)=O(|V|^2)$

Tiefensuche bzw. Depth First Search (DSF)

Unterschied zu BSF: verfolge ersten Nachbarn so lange, bis es keine (unbekannten) Nachbarn mehr gibt, gehe dann wieder eins zurück und verfolge (unbekannte) Nachbarn bis es keine mehr gibt, …

Ordnung (Adjazenzliste): $T(|V|,|E|)=O(|V|+|E|)$

Gerichteter Graph

azyklisch: beim DFS treten keine Rückwärtskanten auf (Kanten auf bekannten Knoten)

topologisch sortiert: lineare Anordnung einer Menge V mit $w, v \in V$ für die gilt:
w kommt vor v (Abhängigkeiten aufgelöst)

Starke Zusammenhangskomponente

1. führe DFS aus; bei jeder Terminierung (letzter Knoten, bevor eine Ebene höher gegangen wird), nummeriere Knoten fortlaufend
2. drehe alle Kanten um
3. starte DSF bei kleinster Nummerierung; bei Terminierung erhält man Zusammenhangskomponente; wähle danach unbesuchten Knoten mit nächstkleineren Nummerierung und wiederhole

Min-Cut

Simple: Wähle zufällig Kanten aus und kontrahiere sie. Falls dabei keine Kante eines minimalen Schnittes kontrahiert wird und nur zwei Kanten übrig bleiben, entsteht so der minimale Schnitt des Graphen → wenig Erfolgsaussichten: $p \geq 1/p^2$

Mit „L": mache nur $p/\sqrt{2}$ viele Kontraktionen, setze rekursiv fort, mache zwei unabhängige Wiederholungen → bessere Erfolgswahrscheinlichkeit: $p \geq 1/(\log_2 p)$

Mit Standardverfahren für probabilistische Algorithmen lässt sich Erfolgswahrscheinlichkeit weiter erhöhen: $p > 1 - e^{-m}$ (m ist Anzahl der Wiederholungen)
dann ist Laufzeit: $O(p^2 \log_2(p)^2)$

Gewichtete Graphen

Priority Queue (PQ)

- wie Queue; beim Speichern wird Priorität vergeben (kann auch nach Einfügen erniedrigt werden)
- gibt immer das Element mit kleinster Priorität zurück
- Operationen: Init, Update/Insert, Remove, Empty
- Implementierung über binären Heap

Union Find

- dynamische Partitionierung einer Menge in Teilmengen, für jede Teilmenge wird Element als Repräsentant ausgewählt
- Operationen: Find(e) (gibt Repräsentant der Teilmenge von e zurück), Union(x, y) (vereinigt Teilmengen von x und y oder gibt true zurück, wenn sie in gleicher Menge sind), FindInit (legt Teilmenge für jedes Element an)
- Balancierung und Pfadkomprimierung: TODO
- Laufzeit für $(n-1)$ Union- und m Find-Aufrufe:

$$O((m+n)\cdot \min\{k \mid A_k(2) \geq n\}) \text{ mit } A_k(m,n) = \begin{cases} n+1 \text{ für } m=0 \\ A(m-1,1) \text{ für } n=0 \\ A(m-1, A(m, n-1)) \text{ sonst} \end{cases}$$

Kürzester Weg/Abstandsproblem: Dijkstra

Ordnung: $O(p^2)$

i: Indexvariable
N_s: Startknoten
T: bisher aufgebauter Baum, initial nur N_s
M: Menge schon betrachteter Knoten, initial nur N_s
d_i: bisher berechneter Abstand von K_q zu N_i

Initial enthalten M und T nur N_s; d_i wird nur für direkte Nachbarn ausgefüllt, sonst unendlich

Dann:

1. finde den noch nicht betrachteten Knoten mit dem kleinsten Abstand
2. füge die Kante, über die der Knoten mit diesem Abstand erreicht wird zu T hinzu
3. überprüfe, ob die Nachbarn vom neuen Knoten über diesen schneller erreicht werden können, als der bisherige Wert und passe die Distanz entsprechend an

Sind alle Knoten betrachtet, enthält T den Quellbaum; in d_i steht der kürzeste Abstand vom Startknoten zu N_i.

Minimaler aufgespannter Baum: Prim, Kruskal, Boruvka

Zusammenhängender Teilbaum mit allen Knoten ohne Zyklen und mit minimalen Summen der Kantengewichte (auch Minimal Spanning Tree bzw. MST)

Prim

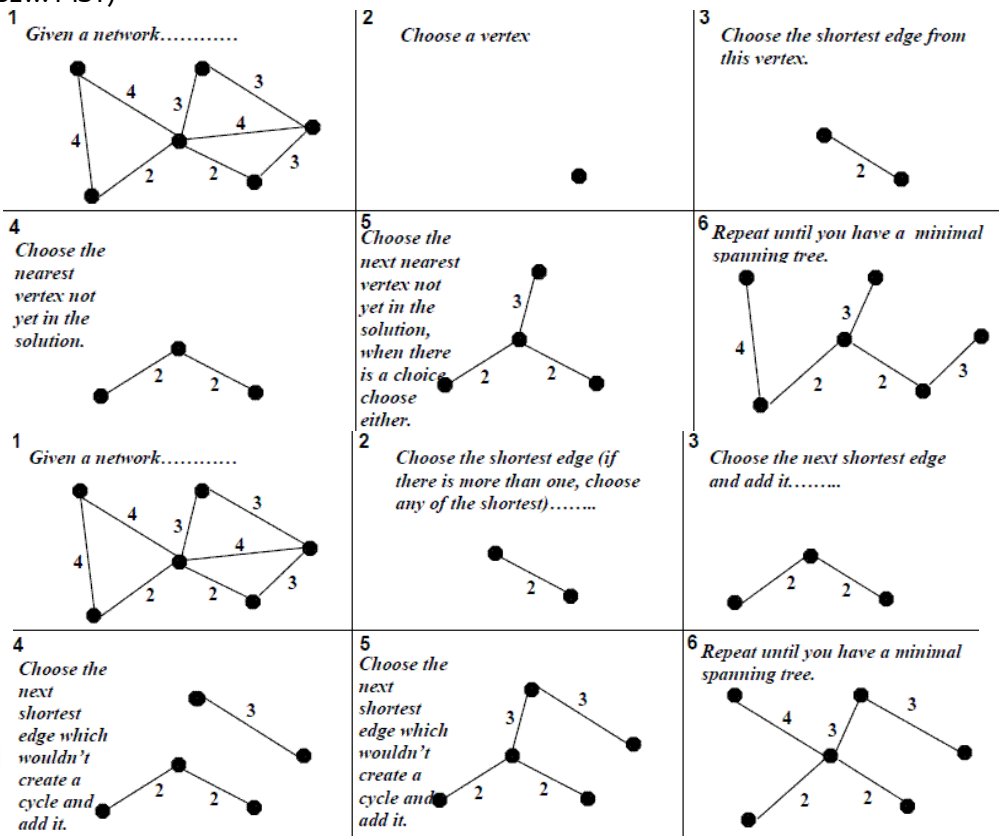
Ordnung:
$O((p+q)\cdot\log_2(p))$ Liste
$O(p^2)$ Matrix
benutzt Priority-Queue
Algorithmus: wähle Startknoten
füge dann Knoten/Kante der Auswahl hinzu, die kleinste Entfernung zur bisherigen Auswahl haben; so lange wiederholen, bis alle Knoten hinzugefügt sind

Kruskal

Ordnung: $O(p+q\log_2(q))$
benutzt Union-Find-Datentyp
Algorithmus: kleinste Kante, die *keinen* Zyklus bildet hinzufügen; so lange wiederholen, bis alle Knoten erreicht werden
Graph muss nicht zu jedem Zeitpunkt zusammenhängend sein!

Boruvka

Ordnung: $O(p+q\log_2(q))$
führt Min-Max-Ordnung zum Sortieren ein



Anzahl rekursiver Aufrufe: $\leq\lfloor\log_2(p)+1\rfloor$

1. identifiziere für jeden Knoten seine kleinste Kante (minimalinzidente Kante)
2. bilde aus den minimalinzidenten Kanten Zusammenhangskomponenten
3. gehe zurück zu 1. und identifiziere die minimalinzidenten Kanten der Komponenten (statt der einzelnen Knoten); wiederhole rekursiv
4. fertig, wenn alles eine große Zusammenhangskomponente ist

Transitiver Abschluss und Abstandsmatrix

- Floyd und Warshal: für gerichtete Graphen, Ordnung: $O(p^3)$
- transitive Abschlussmatrix: beschreibt Erreichbarkeit
  Abstandsmatrix: gibt zusätzlich Distanzen an
- Warshal produziert Folge von Matrizen a_n
  - *i* x *j* Matrix, 1: Verbindung von *i* nach *j*, 0: keine Verbindung
    zunächst nur direkte Verbindungen einzeichnen (a_0)
  - in jedem Schritt eine Kante hinzunehmen: in a_1 dürfen Pfade also auch über Knoten 1 laufen, in a_2 zusätzlich über Knoten 2, …
  - Es gibt so viele Matrizen wie Knoten im Graph.
- Floyd produziert Folge von zwei Matrizen a_n, p_n
  - *a* Matrizen enthalten Gesamtabstände oder ∞ falls noch keine Verbindung bekannt
  - *p* Matrizen enthalten entweder 0 für eine direkte Verbindung oder Hop-Knoten der zum gewünschten Knoten führt (evtl. muss diese Verbindung nochmal rekursiv aufgelöst werden, falls es zu diesem Knoten auch keine direkte Verbindung gibt).
  - Vorgehen: Ablauf wie bei Warshal, in *a* Matrix werden bessere/neue Verbindungen über den aktuellen Knoten geupdated. Beim Update wird an der selben Stelle, an der sich in *a* etwas geändert hat, die aktuelle Matrixnummer (neuer Knoten) eingefügt.

Netzwerke

Definition: $N=(V,E,s,t)$, wobei $G=(V,E)$ ein gewichteter Graph ist; $s\in V$ ist „Quelle“ und $t\in V$ ist „Senke“
Kantenbeschriftungen: Kapazität (maximaler Fluss), aktueller Fluss
Kantenbeschriftungen Restgraph: (Kapazität – Fluss) und in Gegenrichtung Fluss

Maximaler Fluss nach Algorithmus von Edmonds-Karp

1. Breitensuche in Restgraph: kürzester Pfad (minimale Anzahl an Kanten) von *S* nach *T* ohne Gewichte zu beachten
2. Flusserhöhung durchführen: vom kleinsten Gewicht von diesem Pfad
3. wiederholen, bis in Restgraph kein Pfad mehr von *S* nach *T* existiert