Datenstrukturen & Algorithmen

1 Laufzeitanalyse & Asymptotik

Schranken

$O(g) = \{f : \mathbb{N} \to \mathbb{R} | \exists c > 0, \exists n_0 \in \mathbb{N}; \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$

$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R} | \exists c > 0, \exists n_0 \in \mathbb{N}; \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\}$

$\Theta(g) = \{f : \mathbb{N} \to \mathbb{R} | \exists c_1, c_2 > 0, \exists n_0 \in \mathbb{N}; \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$

» $O(n) \subseteq O(n^2)$ stimmt, aber $\Theta(n) \subseteq \Theta(n^2)$ stimmt nicht.

Dominanz von Funktionen

» $log(log(...)) \leq log(n) \leq n^\alpha \leq n \log n \leq n^\beta \leq a^n < n! < n^n$ $(a, \alpha, \beta \in \mathbb{R}, |a| > 1, \alpha \leq 1 < \beta)$

» Binomialkoeffizient: $(\frac{n}{k})^k \leq \binom{n}{k} \leq (\frac{ne}{k})^k \Longrightarrow \binom{n}{3} \in O(n^3)$

» $log\, n\,! \in \Theta(n \log n)$

Grenzwerte

- Falls $\lim_{x\to\infty} \frac{f}{g} = \infty$, $g \in O(f)$ und $f \in \Omega(g)$;
- Falls $\lim_{x\to\infty} \frac{f}{g} = C \in \mathbb{R}^+ \setminus \{0\}$, $f \in \Theta(g)$ und $g \in \Theta(f)$;
- Falls $\lim_{x\to\infty} \frac{f}{g} = 0$, $f \in O(g)$ und $g \in \Omega(f)$.

Summen

» $\sum_{i=0}^{n} i = n(n+1)/2 \in \Theta(n^2)$

» $\sum_{i=0}^{n} i^2 = n(n+1)(2n+2)/2 \in \Theta(n^3)$

» $\sum_{i=0}^{n^2} i \in \Theta(n^4)$

» $\sum_{i=0}^{n} p^i = (1-p^{n+1})/(1-p)$

» $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

Logarithmen

» $\log_b x = \log_b a \cdot \log_a x$

» $a^{log_b x} = x^{log_b a}$

» $log(xy) = log\, x + log\, y$

» $log\, x^y = y \cdot log\, x$

Rekursionsgleichung

Master-Theorem

$$T(n) = \begin{cases} aT(n/b) + f(n), & n > 1 \\ f(1), & n = 1 \end{cases}$$

1. $f(n) = O(n^{\log_b a - \epsilon})$ für $\epsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b a + \epsilon})$ für $\epsilon > 0$ und $af(n/b) \leq cf(n)$ für $c < 1$ und n gross genug $\Rightarrow T(n) = \Theta(f(n))$

Amortisierte Laufzeitanalyse

» Aggregatsanalyse: Obere Schranke für Gesamtzahl Operationen durch Anzahl Operationen teilen

» Kontomethode: Jeder Operation kostet, bei jeder Operation op_k den Betrag a_k einzahlen auf Konto A ($A_k = A_{k-1} + a_k - t_k$) wobei t_k die realen Kosten sind. $A_k - t_k \geq 0 \forall k$ muss gelten.

» Potentialmethode: Potential definieren, das den Zustand einer Datenstrukur zu einem Zeitpunkt beschreibt. Wird bei häufigen, günstigen Operationen erhöht, um damit teure Operationen zu bezahlen.

Laufzeiten verschiedener Algorithmen

» Max-Subarray-Problem: naiv: $\Theta(n^3)$, Präfix-Summe: $\Theta(n^2)$, induktiv: $\Theta(n)$, Komplexität: $\Theta(n)$

2 Select, Find, Sort

Binary Search

» Geg: Sortiertes Array, benötigt $\Theta(\log n)$ Elementarschritte

Theorem: Jeder vergleichsbasierte Algorithmus zur Suche in unsortierten Daten der Länge n benötigt im schlechtesten Fall $\Omega(n)$ Vergleichsschritte.

Auswahlproblem

» Ziel: k-kleinstes Element in einem unsortierten Array der Länge n finden.

» Naiver Ansatz: k-mal das Min. entfernen. ($\Theta(k \cdot n)$)

» Pivotieren: Zufälligen Pivot wählen (worst case p = min, dann $\Theta(n^2)$), dessen Rang bestimmen (guter Pivot liegt in der Mitte, W'keit dafür 0.5), dann Rekursion auf dem relevanten, übrigbleibendem Teilarray. Base case: Rang des Pivots = k.

» Quickselect: Worst-case $\Theta(n^2)$, im Mittel $\Theta(n)$

» Median der Mediane: $\Theta(n)$ Worst-case-Laufzeit, um das k-te Element zu finden.

Bubblesort: Von links nach rechts, 2 El. vergleichen, vertauschen falls nötig, repeat.

» $\Theta(n^2)$ Vergleiche, $\Theta(n)$ Vertauschungen

» Worst-case: reverse sorted

Selectionsort: Kl. Element an 1. Stelle tauschen, Array um 1 Stelle einrücken und dann wiederholen.

» $\Theta(n^2)$ Vergleiche, $\Theta(n)$ Vertauschungen

Insertionsort: i-tes El. wird an i-ter Position eingefügt. (Jasskarten)

» $\Theta(n \log n)$ Vergleiche, $\Theta(n^2)$ Vertauschungen

» Worst-case: reverse sorted

Shellsort: Insertionsort auf Teilfolgen mit versch. Abständen

» $\Theta(n^{3/2})$ Laufzeit bei Folge $(2^k - 1)_k$, $\Theta(n log^2 n)$ bei Folge $(2^p 3^q)$

Quicksort: (Zufälliger) Pivot, partitionieren in $[\leq, p, <]$, wiederholen auf den Teilarrays

» $\mathcal{O}(n^2)$ Laufzeit (selten), $T(n) = T(0) + T(n-1) + cn$; average: $T(n) = 2T(\frac{n}{2}) + cn \Rightarrow \mathcal{O}(n \log n)$

» Worst-case: (reverse) sorted (Partition in 0 und n-1 Elem.)

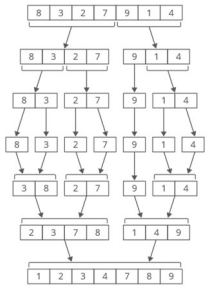
Mergesort: Aufteilen & Verschmelzen

» Vergleiche & Vertauschungen

» Worst-case:

» Nachteil: Benötigt zusätzlich $\Theta(n)$ Speicherplatz für das Verschmelzen; Vorteil: Parallelisierbar

» Ohne Rekursion: StraightMergesort; Teilweise Vorsortierung: NaturalMergeSort



src: interviewcake

Heapsort: El. in Max-/Min-Heap, dann immer Max/Min entfernen und heapify.

» Einfügen: $\mathcal{O}(log\, n)$, Löschen: $\mathcal{O}(log\, n)$, Sortieren $\mathcal{O}(n \log n)$

» Worst-case: $2n \log n$ Vergleiche

» Nachteile: Kein guter Cache, viele Bewegungen im Array → in guten Fällen relativ schlecht (wegen vielen Bewegungen)

Radixsort: Nach Ziffer sortieren

» $\mathcal{O}(p \cdot n)$ Laufzeit (p = #Ziffern)

» Worst-case:

Bucketsort: In Buckets anhand letzer Ziffer auteilen, dann in-order-merge und Buckets mit vorletzter Ziffer usw.

» $\mathcal{O}(n)$ Laufzeit

» Worst-case: n^2, falls alle Keys in einem Bucket

std::sort (Introsort)

» $\mathcal{O}(n \log n)$, bei wenig Elementen Insertionsort, sonst zuerst Quicksort und ab einer gewissen Rekursionstiefe Heapsort, um worst case zu verhindern

Radix- und Bucketsort können nur schneller als $\Theta(n \log n)$ sein, da sie zusätzliche Information über die Zusammensetzung der Keys haben.

In Situ: Algorithmus benötigt konst. Speicherplatz, bzw. arbeitet in place. Selection, Insertion, Bubble, Quick und Heap
Stable: Relative Reihenfolge zweier identischer Elemente bleibt gleich, wird nicht vertauscht. Insertion, Bubble. Nicht: Selection, Quick, Heap.

Theorem: Vergleichsbasierte Sortierverfahren benötigen im schelchtesten Fall und im Mittel mindestens $\Omega(n \log n)$ Schlüsselvergleiche.

3 Datenstrukturen

Stack (LIFO)

» push(e), pop(), top(), isEmpty(), emptyStack()
» implementiert als linked list mit Head-Pointer
» *push*: Neues Element hinzufügen mit Ptr next auf Wert von top, dann top auf neues Element zeigen lassen
» *pop*: Falls top = nullptr => return null, sonst: Ptr p = top; top = top.next

Queue (FIFO)

» enqueue(e), dequeue(), head(), isEmpty(), emptyQueue()
» implementiert als Linked List mit Head- und Tail-Pointer
» enqueue: Neues Element mit next = nullptr;

Skiplist

» Sortierte Linked List mit $n = 2^k$ Elementen und k Ebenen
» Perfekte Skipliste (E1 jedes Element, E2 jedes 2. Element, E3 jedes 4. Eelement etc.): Suchen $\mathcal{O}(log\, n)$, Einfügen $\mathcal{O}(n)$ (weil Höhen angepasst werden müssen)
» Randomisierte Skipliste ($\mathbb{P}(H = i) = \frac{1}{2^{i+1}}$): *Erwartungswert* für Suchen/Einfügen/Löschen eines Elements $\mathcal{O}(log\, n)$

Hashtables

» Problem: z.B. String → grosser Schlüsselbereich und somit grosses Array notwendig. Lösung: Hashing, Abbildung auf eine beschränkte Menge (nicht mehr injektiv).
» Nicht injektiv → Kollisionen
» Behandlung von Kollisionen: Verkettung. Überall, wo eine Kollision auftritt mit einer Linked List die anderen Einträge anhängen (worst case: alle Schlüssel werden auf denselben Index abgebildet → $\Theta(n)$ pro Operation)
» Einfaches gleichmässiges Hashing
  » Jeder Schlüssel wird mit gleicher W'keit und unabhängig von den anderen Schlüsseln auf einen der m verfügbaren Slots abgebildet.
  » Füllgrad/Belegungsfaktor: $\alpha = n/m$
  » Eine verkettete Hashtabelle habe Füllgrad α. Die erwarteten Kosten der nächsten Operation betragen $\Theta(1 + \alpha)$
» Open Addressing
  » Speichere Überläufer direkt in der Hashtabelle mit einer Sondierungsfunktion
  » Lineares Sondieren: $s(k, j) = h(k) + j$. Da viele Datensätze oft Muster aufweisen (→ lange zsh. Bereiche), viele Kollisionen, bevor ein freier Slot gefunden wird.
  » Quadratisches Sondieren: $s(k, j) = h(k) + \lceil j/2 \rceil^2 (-1)^{j+1}$
  » Double Hashing: $s(k, j) = h(k) + j \cdot h'(k)$. h und h' sollten möglichst unabhängig voneinander sein.
  » Analyse gleichm. Hashing mit open addressing: Habe eine Hashtabelle Füllgrad $\alpha < 1$, dann hat die nächste Operation *erwartete* Kosten $\leq \frac{1}{1-\alpha}$. Erfolglose Suche $\leq \frac{1}{\alpha} \cdot \log \frac{1}{1-\alpha}$

Fibonacci-Heap

» Operationen: makeHeap(), insert(e), min(), extractMin(), union(h1, h2), decreaseKey(e, k), delete(e).
» Laufzeit: extractMin und delete $\Theta(\log n)$, Rest $\Theta(1)$
» Struktur: Linked List von Min-Heaps mit Ptr. auf Min.

4 Trees

Bäume haben keine Zyklen.

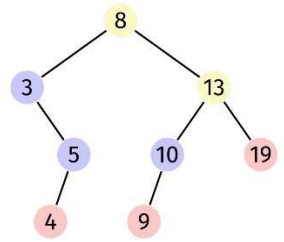
Anwendungen

» Entscheidungsbäume: Hierarchische Darstellung von Entscheidungsregeln
» Syntaxbäume: Parsen und Traversieren von Ausdrücken, z.B. im Compiler oder von math. Termen
» Codebäume: Darstellung eines Codes, z.B. Morsealphabet, Huffman Code
» Suchbäume: effizientes Suchen eines Elementes

Ordnung eines Baumes: Max. Anzahl Kinderknoten
Höhe eines Baumes: Max. Pfadlänge Wurzel – Blatt

Binary Search Tree (BST)

» Jeder Knoten speichert einen Key, Keys im Teilbaum `v->left` < `v` < `v->right`
» Da der Baum nicht balanciert ist, hat eine Suche die worst-case-Laufzeit von $\mathcal{O}(T(v))$ (mit $T(v)$ = Tiefe des (Teil)Baumes mit Root v), falls balanicert, dann $\mathcal{O}(log\, n)$.
» Traversierungsarten (pre/post sind strukturerhaltend)
  » Preorder: `v, left, right`
  » Postorder: `left, right, v` (ideal zum Löschen)
  » Inorder: `left, v, right`



Pre: 8, 3, 5, 4, 13, 10, 9, 19
Post: 4, 5, 3, 9, 10, 19, 13, 8
In: 3, 4, 5, 8, 9, 10, 13, 19

» Knoten entfernen
  » Knoten hat keine Kinder → Knoten durch Blatt (nullptr) ersetzen
  » Knoten hat ein Kind → Knoten durch sein Kind ersetzen
  » Knoten hat zwei Kinder → Knoten durch seinen symmetrischen Vorgänger/Nachfolger ersetzen (rechtestes Element von v.left bzw. linkstes Element von v.right). Bsp.: vgl. Blid oben: 5 ist der sym. Vorgänger von 8, 9 der sym. Nachfolger

```
SymmetricSuccessor(v) {
 w = v.right; x = w.left;
 while(x != nullptr) {
  w = x; x = x.left }
 return w; }
```

  » Entfernen von v hat Kosten $\mathcal{O}(h(T))$: $\mathcal{O}(h(T))$ für das Suchen von v, $\mathcal{O}(h(T))$ für das Suchen vom sym. Nachfolger, Entfernen/Einfügen in $\mathcal{O}(1)$
» Weitere Laufzeiten: Auslesen (+ Entfernen) des Minimums in $\mathcal{O}(h(T))$. Baum hat im best case logarithmische Höhe, im worst case eine lineare Höhe (Linked List)

Min/Max-Heap

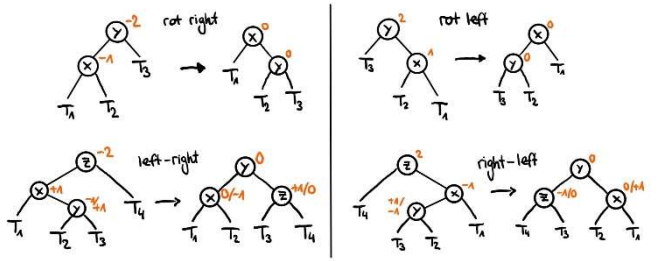
» Eigenschaften
  » vollständiger Binary Tree bis auf die letzte Ebene
  » Lücken des Baumes in der letzten Ebene höchstens rechts
  » Heap-Bedingung: Schlüssel eines Kindes kleiner/grösser als der des Elternknotens
» Baum → Array (Startindex 0)
  » children(i) = $\{2i + 1, 2i + 2\}$
  » parent(i) = $\lfloor (i - 1)/2 \rfloor$
» Höhe $H(n) = \lfloor log\, n \rfloor + 1$

» Einfügen: Element an 1. freier Stelle einfügen, dann Heap-Eigenschaft durch Aufsteigen wiederherstellen (worst case $\mathcal{O}(\log n)$)
» Max. entfernen: Ersetze Max. durch letztes (unten rechts) Element, dann Root versickern lassen, um Heap-Eigenschaft wiederherzustellen
» vgl. Pseudo-Code, Kap. 10

	Binary Heap (worst-Case)	Fibonacci Heap (amortisiert)
MakeHeap	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\log n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$\Theta(1)$
ExtractMin	$\Theta(\log n)$	$\Theta(\log n)$
Union	$\Theta(n)$	$\Theta(1)$
DecreaseKey	$\Theta(\log n)$	$\Theta(1)$
Delete	$\Theta(\log n)$	$\Theta(\log n)$

AVL Trees

» Ziel: Verhinderung der Degenerierung zur Linked List (weil $\Theta(n)$) Balancierung garantiert, dass ein Baum mit n Knoten stets eine Höhe von $\mathcal{O}(log\, n)$ hat. $\text{bal}(v) \coloneqq h(T_r(v)) - h(T_l(v))$
» AVL Bedingung: $\forall v : \text{bal(v)} \in \{-1,0,1\}$
» AVL Baum ist asymptotisch nicht mehr als 44% höher als ein perfekt balancierter Baum (hat Höhe $\lfloor \log n \rfloor + 1$)
» Min. #Blätter in Abh. der Höhe ($h > 2$): $N(h) = N(h-1) + N(h-2) = F_{h+2}$ mit Fibonacci Zahlen $F_0 \coloneqq 0, F_1 \coloneqq 1$, wobei ein Blatt ein Knoten mit `left = right = nullptr` ist.
» Einfügen: jeweils Balance speichern, einfügen wie beim BST, dann Balance für alle Knoten aufsteigend bis root prüfen, AVL Bedingung allenfalls durch Rotation wiederherstellen, Balance updaten



» Löschen: a) Node hat keine Kinder: Balance nach oben anpassen, allenfalls Teilbaum rebalancieren. b) Node hat ein Kind: Node durch Kind ersetzen, Balance anpassen. c) Node hat zwei Kinder: Node durch symmetrischen Nachfolger ersetzen, dann Node löschen.
» Laufzeit für Suchen, Einfügen und Löschen: $\mathcal{O}(\log n)$

Huffman (greedy)

» Liste mit relativen Häufigkeiten sortieren. Jeweils 2 Elemente mit der geringsten Häufigkeit zu einem Element zusammenfassen und in die Liste einsortieren. Konstruktion des Baumes bottom-up. $O(n \log n)$

5 Dynamic Programming

Memoization

» Abspeichern von Zwischenergebnissen
» Bevor ein Teilproblem gelöst wird, wird die Existenz eines entsprechenden Zwischenergebnisses geprüft und verwendet.
» Tausch: Laufzeit vs. Speicherplatz
» Bei Divide-and-Concquer (z.B. Mergesort) werden Teilprobleme nur 1x benötigt, bei DP jeweils mehrfach.

DP Ansatz

1. DP-Tabelle mit Information zu Teilproblemen
   → Dimension der Tabelle? Bedeutung der Einträge?
2. Berechnung der Randfälle (base cases)
   → Welche Einträge sind nicht von anderen abhängig?
3. Berechnungsreihenfolge bestimmen
   → Wie berechnet man die nächsten Einträge?
4. Auslesen der Lösung
   → Wie konstruiert man die Lösung aus der Tabelle?

Laufzeit: Typischerweise #Teilprobleme · Zeit/TP
Bottom Up («Tabulation»): Starte bei dp[0], berechne von dort aus bis dp[n]. (eher iterativ)
Top Down («Memoization»): Starte bei dp[n], überlege, wie man auf das Ergebnis kommt, bis unten base cases erreicht werden. Etwas schneller, falls nicht jedes Teilproblem benötigt wird, da nur jene berechnet werden, die wirklich für dp[n] benötigt werden. (eher rekursiv)
Häufige DP-Ansätze: Prefix-Sum.

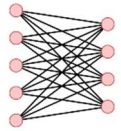
Greedy Algrorithms

» In jeder Iteration wird der aktuell beste Schritt gewählt. Probleme auf diese gierige Weise zu lösen ist meist schnell, aber nicht immer die beste Lösung.
» Ein Optimierungsproblem kann damit gelöst werden, wenn
  » es eine optimale Substruktur hat (Lösung ergibt sich durch Kombination optimaler Teillösungen)
  » die "greedy choice property" gilt: Die Lösung eines Problems kann konstruiert werden, indem ein lokales Kriterium herangezogen wird, das nicht von der Lösung der Teilprobleme abhängt.

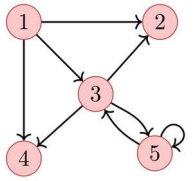
6 Graph Theory

Graph

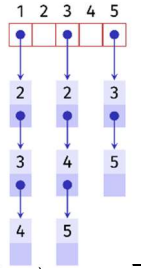
» besteht aus Knoten & Kanten (vertices/edges) ($G = (V, E)$)
» Eulerzyklus: Zyklus, der alle Kanten eines Graphen genau einmal enthält. Eulerzyklus ⇔ jeder Knoten hat gerade Anzahl Kanten (gerader Grad)
» Gerichteter Graph: $V = \{v_1, \dots, v_n\}$ und $E \subseteq V \times V$
» Ungerichteter Graph: V gleich, $E \subseteq \{\{u, v\} \mid u, v \in V\}$
» Ein ungerichteter Graph ohne Schleifen, in dem jeder Knoten mit jedem anderen Knoten verbunden ist, heisst vollständig (full)
» Bipartiter Graph: V kann in in disjunkte U und W aufgeteilt werden, so dass alle Kanten einen Knoten in U und einen W haben.



» Gewichteter Graph $G = (V, E, c)$ mit Gewichtsfkt. $c : E \to \mathbb{R}$
» Für gerichtete Graphen
  » Vorgängermenge von v: $\text{N}^-(v) \coloneqq \{u \in V \mid (u, v) \in E\}$
  » Nachfolgermenge von v: $\text{N}^+(v) \coloneqq \{u \in V \mid (v, u) \in E\}$
  » Eingangsgrad/Ausgangsgrad: Kardinalität der Mengen
  » w heisst adjazent zu v, falls $w \in \text{N}^+(v)$
  » $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v)$
» Für ungerichtete Graphen
  » Nachbarschaft von v: $N(v) = \{w \in V \mid \{v, w\} \in E\}$
  » $\deg(v) = |N(v)|$. Spezialfall Schleifen: erhöhen Grad um 2(!)
  » $\sum_{v \in V} \deg(v) = 2|E|$
» Adjazenzmatrix: Boolsche Matrix mit $a_{ij} = 1$, falls $(i, j) \in E$. Symmetrisch, falls G ungerichtet. Speicherbedarf $\Theta(n^2)$.
Adjazenzliste: Array mit $A[i]$ = Linked List mit Einträgen von $\text{N}^+(\text{v}_\text{i})$. Speicherbedarf $\Theta(|V| + |E|)$



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



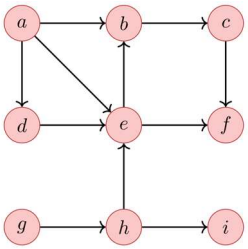
Operation	Matrix (Θ)	Liste (Θ)
Nachbar/Nachfolger von v finden	n	$deg^+(v)$
v ohne Nachbar/Nachfolger finden	n^2	n
$(v, u) \in E$?	1	$deg^+(v)$
Kante einfügen	1	1
Kante (v, u) löschen	1	$deg^+(v)$

Tiefensuche (DFS)

» Verfolge Pfad, bis nichts mehr besucht werden kann
» Man kann Zyklen damit finden

Breitensuche (BFS)

» Verfolge Pfad zuerst in Breite, dann in Tiefe



DFS: $a, b, c, f, d, e, g, h, i$
BFS: $a, b, d, e, c, f, g, h, i$
Top. Sort (Bsp.): $a, d, g, h, i, e, b, c, f$

Bemerkungen:
» DFS entspricht BFS, wenn G wie ein Stern/Strahl aussieht «*»

Topologisches Sortieren

» $G = (V, E)$ gerichtet, kreisfrei, $\Theta(|V| + |E|)$
» Gibt eine Reihenfolge der Abhängigkeiten der Knoten an (nicht eindeutig), z.B. für eine Auswertungsreihenfolge

Shortest Path

» Beobachtungen/Voraussetzungen
  » Bei negativen Zyklen existiert kein kürzester Weg
  » Dreiecksungleichung: Ein kürzester Weg von s nach v kann nicht länger sein als ein kürzester Weg von s nach v via u.
  » Optimale Substruktur: Teilpfade von kürzesten Pfaden sind kürzeste Pfade.
  » Kürzeste Pfade enthalten keine Zyklen
» Allgemeiner Algorithmus: Weg zu allen Knoten überschätzen ($d_s[v] = \infty$), Start wählen ($d_s = 0$), eine Kante wählen und relaxieren (falls sich Distanz zum neuen Knoten verbessert: $d_s[v]$ aktualisieren, Vorgänger merken ($\pi_s[v] = u$)). Wiederholen, bis nichts mehr relaxiert werden kann. Relaxieren ist sicher wegen obigen Voraussetzungen. Kanten wählen: DAG → top. Sort
» **Dijkstra** (greedy)
  » Mengen: M (Knoten, deren kürzester Weg bereits bekannt ist), R = $\bigcup_{v \in M} N^+(v) \setminus M$ (Weg bekannt, aber nicht zwingend kürzester), $U = V \setminus (M \cup R)$ (unbekannte Knoten)
  » Laufzeit $\mathcal{O}(|E| \log |V|)$, besser Fib.-Heap: $\mathcal{O}(|E| + |V| \log |V|)$
  » Funktioniert mit negativen Kantengewichten (keine Zyklen!), kann dann aber exponentielle Laufzeit haben
» **Bellman-Ford** (DP)
  » Induktion über Anzahl Kanten: $d_s[i, v]$ kürzeste Weglänge von s nach v über max. i Kanten ($i \leq n - 1$)
  » $d_s[i, v] = \min\{ d_s[i-1, v], \min_{(u,v) \in E}(d_s[i-1, u] + c(u, v))\}$
  » $d_s[0, s] = 0, d_s[0, v] = \infty \forall v \neq s$
  » $\mathcal{O}(|E| \cdot |V|)$,
  » Funktioniert mit negativen Kantengewichten, kann negative Zyklen detektieren → return false
» **Floyd-Warshall** (DP)
  » Für DAG oder zyklisch mit positiven Gewichten
  » Induktion über Knoten
  » $\mathcal{O}(|V|^3)$, kann auf einer Matrix (in place) ausgeführt werden
  » retourniert kürzeste Pfade von bel. Knoten zu bel. anderem, kein fixert Startpunkt
» **Johnson**
  » Zuerst Bellman-Ford, um negative Zyklen zu entfernen, dann mit einer Potentialfunktion negative Kantengewichte so anpassen, dass Dijkstra angewendet werden kann.
  » $\mathcal{O}(|V| \cdot |E| \cdot log|V|)$, Fib.-Heap: $\mathcal{O}(|V|^2 log|V| + |V| \cdot |E|)$
» **A*-Algorithmus**
  » Erweiterung von Dijkstra, zusätzliche heuristische Funktion, die die Richtung des Zielknotens miteinbezieht.
» Bemerkungen
  » Fib.-Heap-Laufzeiten sind amortisiert
  » F/W, Johnson berechnen kürzeste Pfade von allen zu allen Knoten (sind obv. langsamer als Dijkstra, B/F, A*)
  » Johnson besser als F/W für dünn besetzte Graphen

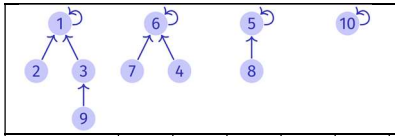
Problem	Methode	Laufzeit	dicht $m \in \mathcal{O}(n^2)$	dünn $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
$c \geq 0$	Dijkstra	$\mathcal{O}((m+n) \log n)$	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n \log n)$
allgemein	Bellman-Ford	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$

Algorithmus			Laufzeit
Dijkstra (Heap)	$c_v \geq 0$	1:n	$\mathcal{O}(\|E\| \log \|V\|)$
Dijkstra (Fibonacci-Heap)	$c_v \geq 0$	1:n	$\mathcal{O}(\|E\| + \|V\| \log \|V\|)$ *
Bellman-Ford		1:n	$\mathcal{O}(\|E\| \cdot \|V\|)$
Floyd-Warshall		n:n	$\Theta(\|V\|^3)$
Johnson		n:n	$\mathcal{O}(\|V\| \cdot \|E\| \cdot \log \|V\|)$
Johnson (Fibonacci-Heap)		n:n	$\mathcal{O}(\|V\|^2 \log \|V\| + \|V\| \cdot \|E\|)$ *

* amortisiert
Johnson ist besser als Floyd-Warshall für dünn besetzte Graphen ($|E| \approx \Theta(|V|)$).

Minimum Spanning Tree (MST)

» Sei $T = (V, E', c)$ mit $E' \subset E$ ein MST, also ein zusammenhängender, zyklenfreier Graph, so dass $\sum_{e \in E'} c(e)$ minimal ist.
» Konstruktion mittels **Kruskal**-Algorithmus. Immer billigste Kante hinzufügen, welche keinen Zyklus erzeugt. (greedy, nicht eindeutig). Laufzeit $\Theta(|E| log|V|)$.
» Implementation mit **Union Find** (Partitionen vereinigen)
  » Idee: Jede Partition als Baum, Wurzel = Name der Menge



{{1, 2, 3, 9}, {7, 6, 4}, {5, 8}, {10}}

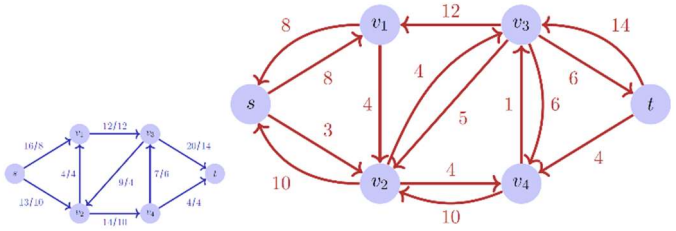
Index	1	2	3	4	5	6	7	8	9	10
Parent	1	1	1	6	5	6	6	5	3	10

  » makeSet(i): `p[i] = i; return i;`
  » find(i): `while(p[i] != i) i = p[i]; return i;`
  » union(i, j): `p[j] = i;` (i und j müssen Wurzeln sein, sonst union(find(i), find(j)))
  » Kann im worst case zu einer Kette entarten → `Θ(n)`
  » Kann verschieden optimiert werden, z.B. kleineren Baum unter grösseren Baum hängen (→ benötigt zusätzliche Info über Grösse) oder bei jedem find alle Knoten an Wurzel hängen.
» Jarnik/Prim/Dijkstra: Wie Dijkstra, aber so, dass nur Kanten hinzugefügt werden, die keine Zyklen bilden. $O(|E| log|V|)$, Fib.-Heap: $O(|E| + |V| log|V|)$

Flussnetzwerke

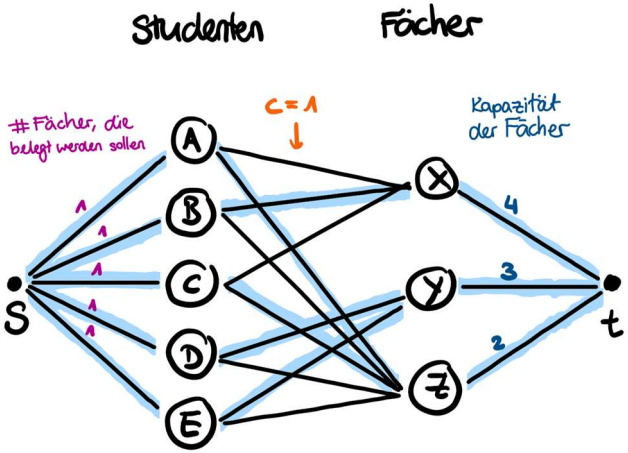
» $G = (V, E, c)$ gerichteter Graph mit Kapazitäten
» Antiparallele Kanten verboten: $(u, v) \in E \Rightarrow (v, u) \notin E$
» Fehlen einer Kante durch Kapazität $c = 0$ modelliert
» Quelle s, Senke t, jeder Knoten v dazwischen muss auf einem Pfad liegen
» Fluss $f: V \times V \to \mathbb{R}, f(u, v) \leq c(u, v), f(u, v) = -f(v, u)$
» Summe des Gesamtflusses (inkl. neg. Gegenfluss) muss 0 sein
» Schnitt: Partitionierung von V in S und T, $s \in S$ und $t \in T$
  » Kapazität eines Schnittes: Summe der Kapazität aller Kanten von S nach T
  » Minimaler Schnitt: Schnitt mit minimaler Kapazität
  » Flusserhaltung über Schnitt
  » $|f| = f(s, V), f(U, U) = 0, f(U, U') = -f(U', U), f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ (falls $X \cap Y = \emptyset$)
  » Für jeden Fluss und Schnitt gilt $f(S, T) = |f| \leq c(S, T)$ mit $c(S, T)$ = Summe der Kapazitäten von S nach T, jene T nach S nicht beachten.
» Max Flow greedy zu lösen funktioniert nicht

» Max Flow: **Ford-Fulkerson**
  » Für ganzzahligen Fluss max. $|f_{\max}|$ Durchläufe. Laufzeit insg. $\mathcal{O}(f_{\max} \cdot |E|)$ (E sind alle Kanten, auch die von der Quelle und zur Senke)
  » Muss für irrationale Kapazitäten nicht terminieren
  » Vorgehen: $f(u,v) = 0\ \forall u, v \in V$. Restnetzwerk $G_f = (V, E_f, c_f)$ und Erweiterungspfade ($s \rightsquigarrow t$) bestimmen mit $c_f(u,v) = c(u,v) - f(u,v)$ und $E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$ (Kanten, wo der Fluss erhöht oder verringert werden kann). Fluss über dem Erweiterungspfad erhöhen, wiederholen, bis kein Erweiterungspfad mehr vorhanden ist (DFS).



» Max Flow: **Edmonds-Karp**: Wähle bei Ford-Fulkerson zum Finden eines Pfades in G_f jeweils einen Erweiterungspfad kürzester Länge (durch BFS) (Laufzeit $\mathcal{O}(|V| \cdot |E|^2)$) (max. $|V| \cdot |E|$ Durchläufe)
» **Max-Flow/Min-Cut Theorem**: Folgendes ist äquivalent:
1. f ist ein maximaler Fluss in G
2. Restnetzwerk G_f enthält keine Erweiterungspfade
3. Für einen Schnitt von S nach T gilt $|f| = c(S,T)$

Anwendung: 5 Studenten wählen 2 von 3 möglichen Fächer, bekommen 1 zugeteilt (Max-Flow auch durch Erweiterungsnetzwerk bestimmen).

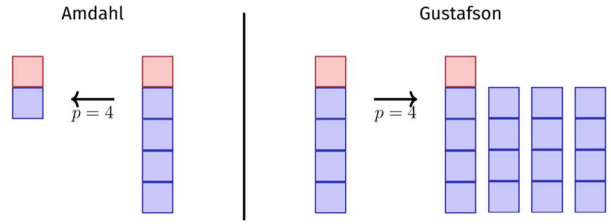


7 Parallel Programming

Amdahl $S_p = \frac{T_1}{T_p} \leq \frac{W_s+W_p}{W_s+W_p/p}$ oder mit λ = serieller Anteil (%):
$S_p \leq \frac{1}{\lambda + \frac{1-\lambda}{p}}$ bzw. $S_\infty \leq 1/\lambda$.

Gustafson: $W_s + p \cdot W_p = \lambda \cdot T + p(1-\lambda)T$ mit Speedup
$S_p = \frac{W_s+p \cdot W_p}{W_s+W_p} = p - \lambda(p-1)$.

Amdahl geht von relativem sequentiellem Teil aus, Gustafson hält sequentiellen Teil konstant und variiert parallelen Teil.



Greedy Scheduler: Teilt zu jeder Zeit so viele Tasks zu Prozessoren zu wie möglich. Auf einem idealen Parallelrechner mit p Prozessoren führt ein Greedy Scheduler eine mehrfädige Berechnung mit Arbeit T_1 und Zeitspanne T_∞ in Zeit $T_p \leq T_1/p + T_\infty$ aus.

Race condition: Resultat einer Berechnung hängt vom Scheduling ab. Man unterscheidet zwischen Bad Interleavings und Data Races.

Bad interleaving: Fehlerhaftes Programmverhalten verursacht durch eine unglückliche Ausführungsreihenfolge. z.B. Variable wird auf einem Thread zuerst gelesen, dann auf einem anderen Thread gelesen und verändert. Wenn der erste Thread die Variable verändert, wird ein veralteter Wert verändert, die Änderung des zweiten Threads ging verloren (Bsp.: Bank withdrawl). Annahmen, die man über die Atomizität von Operationen trifft, sind praktisch immer falsch, daher gar nicht probieren, sondern Locks verwenden.

Data Race: Fehlerhaftes Programmverhalten verursacht durch ungenügend synchronisierten Zugriff zu einer gemeinsam genutzten Ressource. z.B. gleichzeitiges Lesen/Schreiben

Memory Reordering: Compiler und Hardware dürfen die Ausführung des Codes so verändern, dass die Semantik einer sequentiellen Ausführung nicht geändert wird. Folgendes ist sequentiell äquivalent:

```
void f() {
  x = 1;
  y = x+1;
  z = x+1; }
```

```
void f() {
  x = 1;
  z = x+1;
  y = x+1; }
```

Ausserdem Gefahr bei Optimierungen durch den Compiler:

```
x = 1; while(x == 1);    ⟹    x = 1; while(true);
```

Ein zweiter Thread kann also nicht mehr die Variable ändern und den Loop abbrechen.

Deadlock: Zwei oder mehr Prozesse sind gegenseitig blockiert, weil jeder Prozess auf einen anderen warten muss, um fortzufahren. Erkennung: Finde Zyklen im Abhängigkeitsgraphen, kann normalerweise nicht einfach vermieden werden. → Lock order

Starvation: Wiederholter, erfolgloser Versuch, eine zwischenzeitlich freigegebene Ressource zu erhalten, um die Ausführung fortzusetzen.

Livelock: Konkurrierende Prozesse erkennen einen pot. Deadlock, machen aber keine Fortschritte beim Auflösen des Problems.

Condition Variables: Wenn ein Prozess wegen eines Locks warten muss, kann man ihn per Condition Variable «schlafen» lassen und dann per Notification aufwecken, um die Condition zu prüfen.
  » `notify_one()`, wenn alle Threads auf dieselbe Bedingung warten (effizienter)
  » `notify_all()`, wenn die Threads auf versch. Bedingungen warten und daher jeder Thread die Condition überprüfen muss.

Sobald eine Condition Variable im Spiel ist, muss `std::unique_lock` verwendet werden (kann aber grundsätzlich immer verwendet werden). Wenn keine CV im Spiel ist oder ein Thread nur notified (und nie selbst wartet), kann `std::lock_guard` verwendet werden.

Recursive Mutex: Wird eine gelockte Recursive Mutex von einem Thread nochmals gelockt, entsteht kein Fehler/Deadlock. Um die Mutex zu entsperren, wird dieselbe Anzahl Unlocks wie Locks benötigt.

C++ Threads: Parameter beim Erstellen jeweils als Reference übergeben.

8 C++

Basics

» `std::vector<int> vec(10, 1);` *// 10 Elem. mit Wert 1*
» `std::vector<std::vector<unsigned>> M (x, std::vector<unsigned>(y,0));` *// y × x-Matrix, überall 0*
» `const T var` ⇔ `T const var` (gilt auch für `T&`)
» Deklaration von rechts nach links lesen:

`int const p1;`	p1 konstanter Int
`int const* p2;`	p2 Pointer auf konst. Int

`int* const p3;`	p3 konst. Pointer auf Int
`int const* const p4;`	p4 konst. Pointer auf konst. Int

» `const` ist nicht absolut:

```
int a = 5;
const int* p1 = &a; int* p2 = &a;
*p1 = 2; // Fehler
*p2 = 2; // ok, obwohl *p1 verändert wird
```

Advanced C++

» `auto` nutzen, um nicht den expliziten Typ einer Variable hinschreiben zu müssen.

» Range-based for-loop für Container, die Iterator implementiert haben.

```
for(auto& x : c) ++x;
for(auto x : c) cout << x;
```

x hat hierbei denselben Typ wie die Elemente im Container, x ist kein Iterator/Pointer. x muss eine Referenz sein, wenn die Elemente verändert werden sollen.

» **Templates**

» Ziel: generische Datentypen auf Klassen, Funktionen etc.

» Templates sind weitgehend Ersetzungsregeln zur Instanziierungszeit und während der Kompilation.

» Wenn der Typ bei der Instanziierung nicht inferiert werden kann, muss er explizit angegeben werden:

```
template <typename T>
std::pair<T, T> read(){
 T left; T right;
 std::cin >> left >> right;
 return std::pair<T, T>(left, right); }

auto p = read<double>();
std::pair<int, int> q = read<int>();
```

» **std::map**

```
std::map<int, char> m;
for(int i = 0; i < 26; ++i)
 m[i] = i+65;
for(auto& x : m)
 std::cout << x.first << "/" << x.second << "\n";
```

Funktoren

» Struct/Class, deren `operator()` überloden ist.

» Funktionen, die aber auch einen Zustand annehmen können.

» **Lambda-Expressions** sind Inline-Funktoren/anonyme Fkt.

```
[value] (int x) ->bool {return x > value;}
```
Capture · Parameter · return type · Anweisung

» Return type und Parameter können weggelassen werden

» Minimale Lambda-Expression: `[]{}` (Aufruf: `[]{}();`)

» Verschiedene Captures

» `[x]`: Zugriff auf kopierten Wert von x (nur lesend)

» `[&x]`: Zugriff auf Referenz von x

» `[&x, y]`: Zugriff auf Ref. x und Wert von y

» `[&]`: Default-Referenz-Zugriff auf alle Objekte im Kontext der Lambda-Expression

» `[=]`: Default-Werte-Zugriff auf alle Objekte im Kontext der Lambda-Expression.

» Achtung bei Funktoren innerhalb von Klassen

```
struct mutant {
 int i = 0;
 void do(){ [=]{i = 1;}(); } };
mutant m; m.do();
cout << m.i; // 1, weil bei der λ-Expr. der this-
Pointer (this->i) implizit kopiert wird, nicht
die Variable selbst.
```

Rule of Five

1. Destruktor `~T()`
2. Copy-Konstruktor `T(const T& x)`
3. Zuweisungsoperator
4. Move-Konstruktor `T(T&& x)`
5. Move-Zuweisungsoperator

» std::move produziert eine xvalue-Expression.

```
std::move(first, last, first_new); // moves [first,
last[ to first_new
```

Smart Pointers

```
std::shared_ptr<int> p = std::make_shared<int>(3);
```

9 Code Examples

Beispiel Mutex:

```
#include<thread>
#include<mutex>
int amount = 10;
std::mutex m;
void withdraw() {
 m.lock(); // block line
 --amount;
 m.unlock(); }
void withdraw() {
 std::lock_guard<std::mutex> guard(m); //block scope
 --amount; } //guard wird hier destructed, also
              auch bei exceptions wieder aufgelöst.
int main() {
 std::thread t1(withdraw);
 std::thread t2(withdraw);
 t1.join(); t2.join(); }
```

Beispiel: Brücke, auf der max. 3 Autos sein dürfen.

```
#include<mutex>
#include<condition_variable>
std::mutex m;
std::condition_variable cond;
int count = 0;
void enter_car() {
 std::unique_lock<std::mutex> lock(m);
 cond.wait(lock, [&]{ return count < 3; });
 ++count; }
void leave_car() {
 std::lock_guard<std::mutex> lock(m);
 --count; // ↑ auch unique_lock möglich
 cond.notify_all(); }
```

Beispiel: Reentrant Lock

```
#include <mutex>
#include <thread>
#include <condition_variable>
using mutex = std::mutex;
using guard = std::unique_lock<mutex>;
using condition = std::condition_variable;
class ReentrantLock {
 std::thread::id id;
 unsigned count;
 mutex m;
 condition cond;
public:
 ReentrantLock(): count(0) {}
 void lock(){
  guard g(m);
  std::thread::id curr = std::this_thread::get_id();
  cond.wait(g, [&]{ return (count == 0
   || id == curr);});
  if(count == 0) id = current;
  ++count; }
 void unlock(){
  guard g(m);
  std::thread::id curr = std::this_thread::get_id();
  cond.wait(g, [&]{ return id == current; });
  --count;
  cond.notify_all(); } };
```

DFS

```
// helper function for DFS
enum Color{ white, gray, black };
NodeP Graph::nextWhite(NodeP v, std::unordered_
 map<NodeP, Color>& c){
 for(Edge e : this->edges){
  if(e.source == v && c[e.target] == white)
   return e.target;
  else if(e.target == v && c[e.source] == white)
   return e.source; }
 return nullptr; }
```

```
// DFS main function (ignores edge length)
std::vector<NodeP> Graph::DFS(NodeP start) {
 std::unordered_map<NodeP, Color> c;
 std::vector<NodeP> dfs_nodes; // result path
 std::stack<NodeP> stack;
 for(NodeP n : this->nodes) c[n] = white;  // init:
  all nodes are white
 dfs_nodes.push_back(start);
 NodeP v = start;
 c[v] = gray;
 stack.push(v);
 while(stack.size() != 0){
  NodeP w = nextWhite(v, c);
  if(w != nullptr){
   dfs_nodes.push_back(w);
   c[w] = gray;
   stack.push(w);
   v = w; }
  else{
   c[v] = black;
   if(stack.size() != 0){
    v = stack.top();
    stack.pop();
    if(c[v] == gray) stack.push(v); }}}
 return dfs_nodes; }
```

BFS:

```
// BFS main function (ignores edge length)
std::vector<NodeP> Graph::BFS(NodeP start){
 std::unordered_map<NodeP, Color> c;
 std::vector<NodeP> bfs_nodes;
 std::queue<NodeP> q;
 for(NodeP n : this->nodes) c[n] = white;  // init:
  all nodes are white
 NodeP v = start;
 c[v] = gray;
 q.push(v);
 while(!q.empty()) {
  NodeP w = q.front();
  q.pop();
  for(auto e : this->edges) {
   // look at target and source because edges are
    undirected
   if(e.target == w && c[e.source] == white) {
    c[e.source] = gray;
    q.push(e.source); }
   else if(e.source == w && c[e.target] == white) {
    c[e.target] = gray;
    q.push(e.target); }}
  c[w] = black;
  bfs_nodes.push_back(w); }
 return bfs_nodes; }
```

10 Pseudocode

Heap: $\text{Children}(i) = \{2i, 2i+1\}$, $\text{Parent}(i) = \lfloor i/2 \rfloor$

Aufsteigen:

Input: Array A mit mindestens m Elementen und Max-Heap-Struktur auf $A[1, \dots, m-1]$
Output: Array A mit Max-Heap-Struktur auf $A[1, \dots, m]$.

$v \leftarrow A[m]$ // Wert
$c \leftarrow m$ // derzeitiger Knoten (child)
$p \leftarrow \lfloor c/2 \rfloor$ // Elternknoten (parent)
while $c > 1$ and $v > A[p]$ **do**
  $A[c] \leftarrow A[p]$ // Wert Elternknoten → derzeitiger Knoten
  $c \leftarrow p$ // Elternknoten → derzeitiger Knoten
  $p \leftarrow \lfloor c/2 \rfloor$
$A[c] \leftarrow v$ // Wert → Wurzel des (Teil-)Baumes

Versickern:

Input: Array A mit Heapstruktur für die Kinder von i. Letztes Element m.
Output: Array A mit Heapstruktur für i mit letztem Element m.

while $2i \leq m$ **do**
  $j \leftarrow 2i$; // j linkes Kind
  if $j < m$ and $A[j] < A[j+1]$ **then**
    $j \leftarrow j+1$; // j rechtes Kind mit grösserem Schlüssel
  if $A[i] < A[j]$ **then**
    swap$(A[i], A[j])$
    $i \leftarrow j$; // weiter versickern
  else
    $i \leftarrow m$; // versickern beendet

Quicksort

Input: Array A, welches den Pivot p in $A[l, \dots, r]$ mindestens einmal enthält.
Output: Array A partitioniert in $A[l, \dots, r]$ um p. Rückgabe der Position von p.

while $l \leq r$ **do**
  while $A[l] < p$ **do**
    $l \leftarrow l+1$
  while $A[r] > p$ **do**
    $r \leftarrow r-1$
  swap$(A[l], A[r])$
  if $A[l] = A[r]$ **then**
    $l \leftarrow l+1$
return l-1

Floyd-Warshall

Input: Graph $G = (V, E, c)$ ohne Tyklen mit negativem Gewicht.
Output: Minimale Gewichte aller Pfade d

$d^0 \leftarrow c$
for $k \leftarrow 1$ **to** $|V|$ **do**
  for $i \leftarrow 1$ **to** $|V|$ **do**
    for $j \leftarrow 1$ **to** $|V|$ **do**
      $d^k(v_i, v_j) = \min\{d^{k-1}(v_i, v_j), d^{k-1}(v_i, v_k) + d^{k-1}(v_k, v_j)\}$

Johnson:

Input: Gewichteter Graph $G = (V, E, c)$
Output: Minimale Gewichte aller Pfade D.

Neuer Knoten s. Berechne $G' = (V', E', c')$
if BellmanFord$(G', s) =$ false **then** return "graph has negative cycles"
foreach $v \in V'$ **do**
  $h(v) \leftarrow d(s, v)$ // d aus BellmanFord Algorithmus
foreach $(u, v) \in E'$ **do**
  $\tilde{c}(u, v) \leftarrow c(u, v) + h(u) - h(v)$
foreach $u \in V$ **do**
  $\tilde{d}(u, \cdot) \leftarrow$ Dijkstra$(\tilde{G}', u)$
  foreach $v \in V$ **do**
    $D(u, v) \leftarrow \tilde{d}(u, v) + h(v) - h(u)$

Ford-Fulkerson

Input: Flussnetzwerk $G = (V, E, c)$
Output: Maximaler Fluss f.

for $(u, v) \in E$ **do**
  $f(u, v) \leftarrow 0$
while Existiert Pfad $p : s \rightsquigarrow t$ im Restnetzwerk G_f **do**
  $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \in p\}$
  foreach $(u, v) \in p$ **do**
    $f(u, v) \leftarrow f(u, v) + c_f(p)$
    $f(v, u) \leftarrow f(v, u) - c_f(p)$

Dijkstra

Input: Positiv gewichteter Graph $G = (V, E, c)$, Startpunkt $s \in V$
Output: Minimale Gewichte d der kürzesten Pfade und Vorgängerknoten für jeden Knoten.

foreach $u \in V$ **do**
  $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ null
$d_s[s] \leftarrow 0$; $R \leftarrow \{s\}$
while $R \neq \emptyset$ **do**
  $u \leftarrow$ ExtractMin(R)
  foreach $v \in N^+(u)$ **do**
    if $d_s[u] + c(u, v) < d_s[v]$ **then**
      $d_s[v] \leftarrow d_s[u] + c(u, v)$
      $\pi_s[v] \leftarrow u$
      $R \leftarrow R \cup \{v\}$

Bellman-Ford (DP)

Input: Graph $G = (V, E, c)$, Startpunkt $s \in V$
Output: Wenn Rückgabe true, Minimale Gewichte d der kürzesten Pfade zu jedem Knoten, sonst kein kürzester Pfad.

foreach $u \in V$ **do**
  $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ null
$d_s[s] \leftarrow 0$;
for $i \leftarrow 1$ **to** $|V|$ **do**
  $f \leftarrow$ false
  foreach $(u, v) \in E$ **do**
    $f \leftarrow f \vee \text{Relax}(u, v)$
  if $f =$ false **then return** true
return false;

Kruskal

Input: Gewichteter Graph $G = (V, E, c)$
Output: Minimaler Spannbaum mit Kanten A.

```
Sortiere Kanten nach Gewicht c(e_1) ≤ ... ≤ c(e_m)
A ← ∅
for k = 1 to |V| do
    MakeSet(k)
for k = 1 to m do
    (u, v) ← e_k
    if Find(u) ≠ Find(v) then
        Union(Find(u), Find(v))
        A ← A ∪ e_k
    else
return (V, A, c)
```