DSAL Panikzettel

Caspar Zecha, Luca Oeljeklaus, Philipp Schröer

Version 8 — 14.07.2020

Inhaltsverzeichnis

1 O-Notation **2**
1.1 O-Notation-Rechenregeln . . . 2
1.2 log-Rechenregeln . . . 3

2 Elementare Datenstrukturen **3**
2.1 Listen . . . 3

3 Suchen und Sortieren **4**
3.1 Überblick . . . 4
3.2 Lineare Suche . . . 4
3.3 Binäre Suche . . . 4
3.4 Binäre Suchbäume . . . 4
3.5 Optimale Suchbäume . . . 5
3.6 Rotationen . . . 5
3.7 AVL-Bäume . . . 5
3.8 Treaps . . . 5
3.9 Splay-Bäume . . . 6
3.10 (a,b)-Bäume . . . 6
3.11 B-Bäume . . . 7
3.12 Tries . . . 7
3.13 Hashtabellen . . . 7

4 Amortisierte Analyse **7**

5 Hashing **7**

6 Skip List **8**

7 Mengen **8**

8 Sortieren **8**
8.1 Überblick . . . 8
8.2 Insertion Sort . . . 8
8.3 Vergleichsbäume . . . 9
8.4 Mergesort . . . 9
8.5 Quicksort . . . 9

8.6 Heap . . . 9
8.7 Heapsort . . . 10
8.8 Radixsort-Exchange . . . 10
8.9 Straight-Radixsort . . . 11
8.10 Quickselect . . . 11

9 Graphen **11**
9.1 Darstellung . . . 11
9.2 Tiefensuche . . . 11
9.3 Starke Zusammenhangskomponenten . . . 12
9.4 Kreise . . . 12
9.5 Kosaraju . . . 12
9.6 Topologisches Sortieren . . . 12
9.7 Dijkstra . . . 12
9.8 Bellman und Ford . . . 12
9.9 Floyd-Warshall . . . 13
9.10 Transitive Hülle . . . 13
9.11 Breitensuche . . . 13
9.12 Prioritätswarteschlangen . . . 13
9.13 s-t-Netzwerke . . . 13
9.14 Schnitte in Netzwerken . . . 14
9.15 Bipartites Matching . . . 14
9.16 Minimaler Spannbaum . . . 14
9.17 Algorithmus von Prim . . . 15
9.18 Greedy-Algorithmen . . . 15
9.19 Matroid . . . 15
9.20 Kruskal . . . 15
9.21 Union Find . . . 15

1 O-Notation

Die O-Notation schätzt Funktionen in Klassen ein, etwa für Komplexitätsabschätzungen.

$$g = O(f) \approx g \text{ wächst langsamer als } f$$
$$g = \Omega(f) \approx g \text{ wächst schneller als } f$$
$$g = \Theta(f) \approx g \text{ wächst so schnell wie } f$$

$$O(f) = \{g : \mathbb{N} \to \mathbb{R} | \exists c \in \mathbb{R}_{\geq 0}, \exists N \in \mathbb{N}, \forall n \geq N : |g(n)| \leq c|f(n)|\}$$
$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R} | \exists c \in \mathbb{R}_{\geq 0}, \exists N \in \mathbb{N}, \forall n \geq N : |g(n)| \geq c|f(n)|\}$$
$$\Theta(f) = \{g : \mathbb{N} \to \mathbb{R} | \exists c_1, c_2 \in \mathbb{R}_{\geq 0}, \exists N \in \mathbb{N}, \forall n \geq N : c_1|f(n)| \leq |g(n)| \leq c_2|f(n)|\}$$

1.1 O-Notation-Rechenregeln

- $f(O(1)) = O(1)$
- $O(c \cdot f(n)) = O(f(n))$

- $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
- $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
- $O(f(n))O(g(n)) = O(f(n)g(n))$
- $O(f(n)g(n)) = f(n)O(g(n))$

1.2 log-Rechenregeln

Diese Regeln sind ganz praktisch und sollten für die O-Notation im Hinterkopf behalten werden. Natürlich sind diese nicht spezifisch für die O-Notation.

- $x = \log_a y \Leftrightarrow y = a^x$
- $\log(1) = 0$
- $\log(x) + \log(y) = \log(xy)$
- $-\log(x) = \log(\frac{1}{x})$
- $\log(x) - \log(y) = \log \frac{x}{y}$
- $n \cdot \log(x) = \log(x^n)$
- $\frac{\log(x)}{\log(a)} = \log_a(x)$

Schätze $\log(n^2 + 3n + 5)$ ab:

$$\log(n^2 + 3n + 5) = 2 \cdot \log(n) + \log(1 + \frac{3}{n} + \frac{5}{n^2}) = 2 \cdot \log(n) + O(\frac{1}{n})$$

2 Elementare Datenstrukturen

2.1 Listen

2.1.1 Doppelt verkettete Liste

- Daten in Knoten gespeichert
- Zeiger auf Nachfolger und Vorgänger
- Zyklisch geschlossen
- Leerer Kopf-Knoten (um zu wissen, wann wir am Ende der Liste sind)

Arrays lassen sich mittels Listen simulieren.

2.1.2 Einfach verkettete Liste

- Zeiger nur auf Nachfolger, nicht auf Vorgänger
- kompliziertere Operationen
- zyklisch wie oben, mit leerem Kopf-Knoten

2.1.3 Laufzeiten von Listen

- `append`() = $\Theta(1)$
- `delete`() = $\Theta(n)$
- `find`() = $\Theta(n)$
- `insert`() = $\Theta(n)$

3 Suchen und Sortieren

3.1 Überblick

	Liste	Sortierte Liste Array	Sortiertes Array	Binärer Suchbaum	AVL-Baum	Splay-Baum	Treap	Skip-List	Hashtabelle	Min-Heap
Randomisiert	N	N	N	N	N	N	N	J	J	J	N
Einfügen	1	n	1	n	n	$\log n$	$\log n$	$\log n$	$\log n$	1	n
Suchen	n	n	n	$\log n$	n	$\log n$	$\log n$	$\log n$	$\log n$	1	n
Löschen	n	n	n	n	n	$\log n$	$\log n$	$\log n$	$\log n$	1	n
Minimum	n	1	n	1	n	$\log n$	$\log n$	$\log n$	1	n	1
Maximum	n	1	n	1	n	$\log n$	$\log n$	$\log n$	$\log n$	n	n
Sort. Ausgeben	$n \log n$	n	$n \log n$	n	n	n	n	n	n	$n \log n$	$n \log n$

3.2 Lineare Suche

Suche x im Array, indem alle Elemente durchlaufen werden.
Mittelwert: $\frac{(n+1)}{2}$ Vergleiche.

3.3 Binäre Suche

Suche x im **sortierten** Array, aber halbiere den Suchraum in jedem Durchlauf.
Mittelwert: $\lfloor log(n) \rfloor + 1 = log(n) + O(1)$ Vergleiche im Worst-Case.
Laufzeit: $O(log(n))$.
Praktisches Lemma für Gauß-Klammern:

- $\lfloor a + n \rfloor = \lfloor a \rfloor + n$
- $\lceil a + n \rceil = \lceil a \rceil + n$
- $\lfloor -a \rfloor = -\lceil a \rceil$

3.4 Binäre Suchbäume

- Es gilt für alle Knoten: Linke Kinder kleiner, rechte Kinder größer als Elternknoten
- Interne und Externe Knoten

3.4.1 Einfügen

- In den richtigen externen Knoten

3.4.2 Löschen

Hier ist eine Fallunterscheidung nötig, abhängig vom Typ des zu löschenden Knotens.
Blatt:

1. Einfaches Zeigerverbiegen

Knoten hat kein linkes Kind:

1. Kopieren oder Zeiger verbiegen

Knoten hat ein linkes Kind:

1. Finde den größten Knoten im linken Unterbaum
2. Kopiere seinen Inhalt
3. Lösche ihn

3.4.3 Höhe

Werden in einen leeren binären Suchbaum n verschiedene Schlüssel in zufälliger Reihenfolge eingefügt, dann ist die erwartete Höhe des entstehenden Suchbaums $O(\log n)$.
Laufzeit von Operationen: Einfügen, Löschen und Suchen benötigen $O(h)$ Zeit, bei einem binären Suchbaum mit Höhe h.

3.5 Optimale Suchbäume

- Es seien $k_1 < ... < k_n$ Schlüssel
- $\sum_{i=1}^{n} p_i = 1$
- hat eine minimale Anzahl von Vergleichen im Erwartungswert
- Konstruktion in $O(n^3)$
- Wir suchen k_i mit Wahrscheinlichkeit p_i:
  1. Stelle Tabelle mit Wahrscheinlichkeiten auf
  2. Stelle gleichgroße Tabelle mit Erwartungswerten auf:

$$e_{i,j} = \min_{i \leq r \leq j} (e_{i,r-1} + e_{r+1,j}) + w_{i,j}$$

  3. Den Baum von der Tabelle ablesen

3.6 Rotationen

Rotationen können einen binären Suchbaum so umstrukturieren, dass sich die Höhe einzelner Teilbäume verändert, aber die Suchbaumeigenschaft erhalten bleibt.
Die Rotationen sind ähnlich zur *Splay*-Operation.

3.7 AVL-Bäume

AVL-Bäume sind binären Suchbaume mit der Eigenschaft, dass die Höhe des rechten und linken Unterbaums sich höchstens um 1 unterscheiden. Falls die Eigenschaft nicht erfüllt ist, kann mithilfe von Rotationen die Höhe so verändert werden, dass die Suchbaumeigenschaften erhalten bleiben.

3.8 Treaps

Ein Treap ist ein binärer Suchbaum mit Heap-Eigenschaft, wobei diese auf zufällig gewählte Schlüssel angewendet wird. Dadurch wird bei zufälligen Operationen die Höhe des Baumes wahrscheinlich logarithmisch (und das ist gut).

3.9 Splay-Bäume

3.9.1 Definition

- Selbstorganisierende Datenstruktur; selbstlernend
- armortisiert effizient
- Restrukturierung durch Rotationen; ohne Zusatzinformationen

3.9.2 `splay`-Operation

- zig auf x = Rechtsrotation auf den Vater von x, wenn x das linke Kind ist
- zag auf x = Linksrotation auf den Vater von x, wenn x das rechte Kind ist
- zig-zag = Rechtsrotation auf den Vater von x, gefolgt von Linksrotation auf den Opa
- zag-zig = Linksrotation auf den Vater von x, gefolgt von Rechtsrotation auf den Opa
- zig-zig auf x = Rechtsrotation auf den Opa von x, gefolgt von Rechtsrotation auf den Vater von x
- zag-zag auf x = Linksrotation auf den Opa von x, gefolgt von Linksrotation auf den Vater von x

3.9.3 Suchen

1. Normale Suche von x im Baum
2. Wende `splay`(x) auf den gefundenen Knoten an
3. Prüfe ob x nun in der Wurzel steht

3.9.4 Einfügen

1. Normale Suche von x im Baum, falls gefunden weiter bei 3.
2. Füge x als Blatt ein
3. Wende `splay`(x) an

3.9.5 Löschen

1. Suche x im Baum
2. Prüfe ob x nun in der Wurzel steht
3. `splay`(x^-) auf den größten Knoten im linken Unterbaum
4. Klassisches Löschen von x

3.10 (a,b)-Bäume

3.10.1 Definition

- Jeder Knoten hat höchstens b Kinder
- Jeder innere Knoten außer der Wurzel hat mindestens a Kinder
- Alle Blätter haben die gleiche Tiefe
- Als assoziatives Array: Schlüssel nur in den Blättern, von links nach rechts geordnet
- Als Suchhilfe enthält ein innerer Knoten mit m Kindern genau $m - 1$ Schlüssel
- Hilfsschlüssel in den inneren Knoten müssen nicht in den Blättern stehen

3.10.2 Einfügen

Neues Blatt an richtiger Stelle einfügen.
Falls Elternknoten mehr als b Kinder: Aufteilen in 2 Knoten mit $\lfloor \frac{b+1}{2} \rfloor$ und $\lceil \frac{b+1}{2} \rceil$. Eventuell ist der Vater überfüllt und muss auch behandelt werden.

3.10.3 Löschen

Blatt mit Schlüssel entfernen.
Falls Elternknoten weniger als a Kinder: Zusammenfügen mit Geschwisterknoten und ggf. wieder teilen. Eventuell ist der nächste Elternknoten auch unterbelegt.

3.11 B-Bäume

B-Bäume sind $(m, 2m)$-Bäume, d.h. spezielle (a, b)-Bäume.
Dann beträgt die Zugriffszeit nur $O(\log_m(n))$.

3.12 Tries

In *Tries* entspricht die $i$te Verzweigung dem $i$ten Zeichen des Schlüssels, es wird nicht in die Knoten hineingeschaut.

3.13 Hashtabellen

Hashtabellen bestehen einer Tabelle/Array von Listen. Die Einträge sind dann in der Liste der Position, die durch die Hashfunktion angegeben wird. In dieser Liste wird dann mit linerarer Suche gesucht.

- Lastfaktor $\alpha = \frac{n}{m}$, bei einer Hashtabelle der Größe m mit n Elementen
- Erfolglose Suche $= O(\alpha)$
- Erfolgreiche Suche $= O(1 + \frac{\alpha}{2})$

Wir haben als Hash-Funktion immer eine der folgenden Familie benutzt, wobei m die Größe der Hashtabelle ist:

$$f_{a,b}(x) = (ax + b) \mod p) \mod m, \text{ mit } x \in [0, p-1]$$

4 Amortisierte Analyse

Bei der amortisierten Analyse sollen Komplexitäten von Algorithmen abgeschätzt werden, deren Operationen sich eine Art Bankkonto für Laufzeit bedienen; somit können wir Datenstrukturen abschätzen, wo die durchschnittlichen Kosten gering sind, obwohl wenige Operationen sehr teuer sind.
Dazu suchen wir uns zuerst eine Art Bankkonto-Funktion oder Energie-Funktion aus. Diese hängt von der Zeit und von der Datenstruktur ab und muss zu Beginn 0 sein. Damit können wir dann die Komplexität der einzelnen Operationen ausrechnen, indem wir zu jeder Operation die Differenz der Potenzial-Funktion vom Zeitpunkt vor der Operation und nach der Operation hinzuaddieren.

5 Hashing

Für eine Familie von Hashfunktionen muss gelten:
$\mathcal{H}$ sei eine nichtleere Menge von Funktionen $U \to \{1, \ldots, m\}$.
Dann ist $\mathcal{H}$ eine universelle Familie von Hashfunktionen, wenn für jedes $x, y \in U, x \neq y$ gilt:

$$\frac{|\{h \in \mathcal{H} \mid h(x) = h(y)\}|}{|\mathcal{H}|} \leq \frac{1}{m}$$

Wenn $S \subseteq U$ eine beliebige Untermenge ist und $x \in U, x \notin S$ und $h \in \mathcal{H}$ eine zufällig gewählte Hashfunktion ist, gilt:

$$E(|\{y \in S | h(x) = h(y)\}|) \leq \frac{|S|}{m}$$

Weiterhin gilt für ein beliebiges $k \in \{1, \ldots, m\}$: $\mathrm{P}(h(x) = k) = \frac{1}{m}$, falls h zufällig aus $\mathcal{H}$.

6 Skip List

- Schlüssel nach Größe sortiert
- Suchen: Von Oben nach Unten
- Anzahl ist geometrisch verteilt
- Einfügen, Löschen und Suchen in $O(\log n)$
- Speicherverbrauch = $O(n)$

7 Mengen

- Bitarrays: Universum U kann als Menge durch Bitarrays implementiert werden
- Suchen, Einfügen und Löschen: $O(1)$
- Vereinigung und Schnitt: $O(|U|)$
- Auswahl, begrenzte Teilmenge: $O(|U|)$

8 Sortieren

8.1 Überblick

	Quicksort	Heapsort	Mergesort	Insertion-Sort	Straight-Radix	Radix-Exchange
in-place	I	J	N	J	N	I
stabil	N	N	J	J	J	N
Laufzeit (worst-case)	n^2	$n \log n$	$n \log n$	n^2	nw	nw
Laufzeit (Durchschnitt)	$n \log n$	$n \log n$	$n \log n$	n^2	nw	nw
vergleichsbasiert	J	J	J	J	N	N

in-place = Speicherverbrauch von $O(1)$
stabil = Elemente mit gleichem Schlüssel bleiben in der ursprünglichen Reihenfolge
I = Implementationsabhängig

8.2 Insertion Sort

- Beispiel
  (2,4,1,3) → (**2**,4,1,3) → (**2**,**4**,1,3) → (**1**,**2**,**4**,3) → (**1**,**2**,**3**,**4**)
- Vorgehen
  1. Die ersten n Elemente sind eine sortierte Teilliste.
  2. nehme das n+1. Element, und füge es an der richtigen Stelle ein.
  3. und zack fettig, Aixzellenz.

- Inversionen
  Die Anzahl der Inversionen (Paare der (a_i, a_j), wobei $i < j$, sodass $a_i > a_j$) bestimmt, wieviele Schritte der Algorithmus benötigt (hier vertiefen).

8.3 Vergleichsbäume

Vergleichsbäume gibt es, um darzustellen, über welche Vergleichskombinationen man letztendlich eine Sortierung erreicht. Der Vergleichsbaum hat dann mindestens $n!$ Blätter. Jeder vergleichsbasierte Algorithmus benötigt für das Sortieren einer zufällig permutierten Eingabe im Erwartungswert mindestens $\log(n!)$ Schritte.

8.4 Mergesort

- Beispiel
  (4,2,1,3) → (4,2),(1,3) → (4),(2),(1,3) → (2,4),(1,3) → (2,4),(1),(3) → (2,4),(1,3) → (1,2,3,4)
- Vorgehen
  1. Liste in zwei Teillisten aufteilen
  2. Rekursiv Mergesort auf dem linken, danach auf dem rechten Teilliste aufrufen.
  3. Die Teillisten *mergen*, d.h. so zusammenfügen, dass immer das kleinere Element der beiden Teillisten in die neue Liste eingefügt wird.

8.5 Quicksort

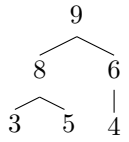
- Beispiel
  (4,7,5,6,9,1,2,8,3) $\xrightarrow{\text{Pivot 4, } 7 \geq 4,\ 3 \leq 4, \text{ vertauschen}}$ (4,**3**,5,6,9,1,2,8,**7**)
  $\xrightarrow{5 \geq 4,\ 2 \leq 4, \text{ vertauschen}}$ (4,**3**,**2**,6,9,1,**5**,**8**,**7**)
  $\xrightarrow{6 \geq 4,\ 1 \leq 4, \text{ vertauschen}}$ (4,**3**,**2**,**1**,9,**6**,**5**,**8**,**7**)
  $\xrightarrow{9 \geq 4,\ 1 \geq 4, \text{ vertauschen}}$ (4,**3**,**2**,**9**,**1**,**6**,**5**,**8**,**7**)
  $\xrightarrow{\text{Tausch rückgängig machen, und Pivot mit dem dann linken Element tauschen}}$ (1,3,2,**4**,9,6,5,8,7)
  Vorgang für (1,3,2) und (9,6,5,8,7) wiederholen.
- Vorgehen:
  1. Pivot Element wählen (bei uns immer das linkeste Element in dem Teil-Array).
  2. Von beiden Rändern nach innen gehen, bis links auf das erste Element gezeigt wird, welches größer als der Pivot ist, und rechts auf das erste Element gezeigt wird, welches kleiner als der Pivot ist, beide Elemente vertauschen.
  3. Wiederhole Schritt 2, bis der rechte Zeiger links vom linken Zeiger steht. Dann den letzten Tausch rückgängig machen, und den Pivot mit dem letzen Element aus der linken Teilliste tauschen.
  4. Erst die linke, dann die rechte Teilliste jeweils mit Quicksort sortieren.

8.6 Heap

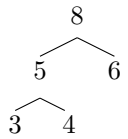
Ein Heap ist ein Baum, bei dem die sog. Heap-Eigenschaft gilt: Jedes Element muss größer/gleich als sein Kindknoten (Max-Heap) sein (analog für Min-Heap). Außerdem sind alle Zeilen gefüllt, außer die unterste, wobei in dieser die Knoten von links aufgefüllt werden.

8.7 Heapsort

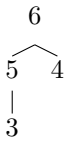
- Beispiel (an einem Maxheap)



Array : $[\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing]$
$\xrightarrow{\text{Oberstes Element ins Array, Heapeigenschaft fixen}}$



Array : $[\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, 9]$
$\xrightarrow{\text{Oberstes Element ins Array, Heapeigenschaft fixen}}$



Array : $[\varnothing, \varnothing, \varnothing, \varnothing, 8, 9]$
$\xrightarrow{\text{Oberstes Element ins Array, Heapeigenschaft fixen}}$

Array : $[\varnothing, \varnothing, \varnothing, 6, 8, 9]$
$\xrightarrow{\text{Oberstes Element ins Array, Heapeigenschaft fixen}}$

Array : $[\varnothing, \varnothing, 5, 6, 8, 9]$
$\xrightarrow{\text{Oberstes Element ins Array, Heapeigenschaft fixen}}$

Array : $[\varnothing, 4, 5, 6, 8, 9]$
$\xrightarrow{\text{Oberstes Element ins Array, Heapeigenschaft fixen}}$

Array : $[3, 4, 5, 6, 8, 9]$

- Vorgehen (für einen Maxheap)
  1. Maxheap bauen.
  2. Oberstes Element rausnehmen, und in ein Array packen.
  3. Heapeigenschaft wiederherstellen.
  4. Ab 1. wiederholen, bis der Heap leer ist.

8.8 Radixsort-Exchange

8.8.1 Vorgehen

Sortiere Elemente nach der ersten Stelle in-place mit dem Vertauschungsprinzip von Quicksort. Iteriere nach diesem Prinzip alle Stellen durch.

8.8.2 Beispiel

(010,110,111,001,101,011)
$\xrightarrow{\text{sortieren nach Stelle 1}}$ (010,011,001|111,101,110)
$\xrightarrow{\text{sortieren nach Stelle 2}}$ (001|011,010|101|111,110)

$\xrightarrow{\text{sortieren nach Stelle 3}}$ (001|010|011|101|110|111)
$\rightarrow$ (001,010,011,101,110,111)

8.9 Straight-Radixsort

- Beispiel
  (010,110,111,001,101,011)
  $\xrightarrow{\text{sortieren nach Stelle 3}}$ (010,110)(111,001,101,011)
  $\xrightarrow{\text{sortieren nach Stelle 2}}$ (001,101)(010,110,111,011)
  $\xrightarrow{\text{sortieren nach Stelle 1}}$ (001)(010)(011)(101)(110)(111)
  $\rightarrow$ (001,010,011,101,110,111)
- Vorgehen
  1. Zahlen nach der letzten Stelle sortieren.
  2. Zahlen nach der vorletzten Stelle sortieren, wobei die vorherige Reihenfolge unter den Zahlen mit gleicher vorletzter Ziffer die gleiche wie vorher sein soll (*stabil* nach Ziffer sortieren).
  3. Wiederholen, bis nach der ersten Stelle sortiert wurde.

8.10 Quickselect

Eine Art Quicksort, bei der man aber nur das k-te Element sucht, bspw. den Median. Es wird also nur rekursiv Quicksort auf dem Teil gemacht, in dem das k-te Element ist. Am Ende wird dann das k-te Element zurückgegeben, wenn es das Pivot-Element ist.

9 Graphen

9.1 Darstellung

- Ungerichteter Graph $G = (V, E)$, V ist die Menge der Knoten und $E \subseteq \binom{V}{2}$ die der Kanten
- Gerichteter Graph $G = (V, E)$, mit $E \subseteq V \times V$
- Speicherung als Adjazenzmatrix mit Speicherbedarf von $\Theta(|V|^2)$
- Als Adjazenzliste mit Speicherbedarf $\Theta(|V| + |E|)$
- Wechsel zwischen beiden Darstellungen: $O(n^2)$

9.2 Tiefensuche

Ein Algorithmus in Linearzeit, welcher viele Anwendungen hat.

- Vorgehen
  1. Alle Knoten weiß markieren, und einen Knoten als Startpunkt auswählen, seine Discovery-Zeit auf 1 setzen. Weiter zu Schritt 3.
  2. Falls vorhanden, einen weiteren weißen Knoten auswählen, seine Discovery-Zeit setzen, ihn grau färben und weiter zu Schritt 3. Falls nicht, weiter zu Schritt 6.
  3. Falls es einen weiß markierten inzidenten Knoten gibt, diesen auswählen und weiter zu Schritt 4, sonst zu Schritt 5.
  4. Die Discovery-Zeit setzen, ihn grau markieren. Weiter zu Schritt 3.
  5. Die Finish-Zeit setzen, den Knoten schwarz markieren, und falls vorhanden, den Vorgänger auswählen und weiter zu Schritt 3. Falls nicht vorhanden, weiter zu Schritt 2.
  6. Jetzt sollten alle Knoten schwarz markiert sein und eine Discovery- und Finish-Zeit besitzen. Jede Kante (u,v) nun wie folgt kategorisieren:

- falls $d(u) + 1 = d(v)$ und $f(u) > f(v) \Rightarrow$ Baumkante
- falls $d(u) < d(v)$ und $f(u) > f(v) \Rightarrow$ Vorwärtskante
- falls $d(u) > d(v)$ und $f(u) < f(v) \Rightarrow$ Rückwärtskante
- sonst Querkante

9.3 Starke Zusammenhangskomponenten

Starke Zusammenhangskomponenten sind die Teilmengen der Knoten eines Graphen, die sich gegenseitig von jedem Knoten dieser Komponente über entsprechend gerichtete Kanten erreichen können.

9.4 Kreise

Wende Tiefensuche auf den Graphen an. Wenn eine Rückwärtskante exisitiert, dann ist der Graph nicht kreisfrei.

9.5 Kosaraju

Findet starke Zusammenhangskomponenten in gerichteten Graphen.
Vorgehen:

1. Kehrgraphen bilden (alle Kanten einfach umdrehen).
2. Eine Tiefensuche darauf ausführen.
3. Knoten nach Finishtime sortieren.
4. In dieser Reihenfolge auf dem Originalgraphen eine Tiefensuche ausführen (bei Schritt 1 und 2 der Tiefensuche den Knoten wählen, der die größte Finish-Zeit hat).
5. Jeder Baum, der im Wald aus den Baumkanten der Tiefensuche ist, spannt eine starke Zusammenhangskomponente auf.

9.6 Topologisches Sortieren

Topologische Sortierung bezeichnet in der Mathematik eine Reihenfolge von Dingen, bei der vorgegebene Abhängigkeiten erfüllt sind. G sei ein DAG. Wenn wir die Knoten von G nach den Finish-Zeiten einer Tiefensuche umgekehrt anordnen, sind sie topologisch sortiert.

9.7 Dijkstra

Dieser Algorithmus berechnet die kürzesten Pfade zu einem Startknoten in einem gewichteteten Graphen.

1. Wähle Startknoten und setze seine Distanz auf 0 und die der anderen Knoten auf ∞
2. Solange es unbesuchte Knoten gibt, wähle den mit der minimalen Distanz aus:
   - Markiere den Knoten als besucht
   - Aktualisiere für alle Nachbarknoten die Distanz, wenn jeweils mit diesem Knoten die Distanz verringert werden kann

9.8 Bellman und Ford

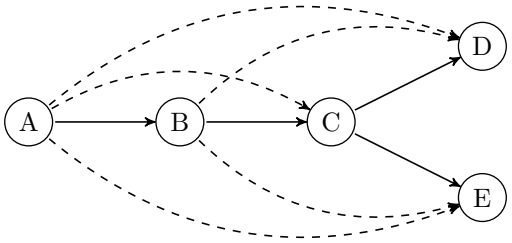
Berechnung der kürzesten Pfade mit negativen Kantengewichten. Dijkstra kann das nicht.
Die Idee ist einfach: Wir relaxieren alle Kanten soweit, bis es keine Änderung mehr gibt. Relaxieren ist einfach das Verringern der gespeicherten Distanz eines Nachbarknotens auf Basis des Gewichtes der Kante dorthin.
Bei Kreisen mit negativem Gewicht terminiert der Algorithmus nicht.

9.9 Floyd-Warshall

Floyd Berechnet die kürzesten Wege zwischen allen Knotenpaaren in $O(|V|^3)$. Warshall ist der Spezialfall, bei dem nur auf Existenz eines Weges geprüft wird. Dies geht in $O(|V| + |E'| + k^3)$ wobei k die Anzahl der starken Zusammenhangskomponenten des Graphen mit Knoten V und E' die Kanten der transitiven Hülle sind.

9.10 Transitive Hülle

Beispiel:



Gegeben sei eine Relation „Direkter-Vorgesetzter“ mit folgenden Beziehungen:

C ist direkter Vorgesetzter von D und E
B ist direkter Vorgesetzter von C
A ist direkter Vorgesetzter von B

Die transitive Hülle dieser Relation enthält nun zusätzlich auch die indirekten Vorgesetzten:

A ist Vorgesetzter von B, C, D, E
B ist Vorgesetzter von C, D, E
C ist Vorgesetzter von D und E

Man verwendet die Warshall Version des Floyd-Warshall-Algorithmus' um sie zu bestimmen.

9.11 Breitensuche

Vorgehen:

1. Startknoten k wählen.
2. Für jede zu k inzidente Kante prüfen, obe der gegenüberliegende Knoten schon entdeckt wurde, bzw. ob es der gesuchte Knoten ist.
3. Falls nicht, Knoten in Warteschlange einreihen.
4. Schritt 2. für alle Elemente der Warteschlange durchführen, wobei immer erst alle direkten Nachfolger bearbeitet werden.

9.12 Prioritätswarteschlangen

Eine Warteschlange, bei der Elemente bestimmte Gewichte haben. Hier implementiert mit einem Heap: Das Einfügen ist unverändert wie beim Heap, das Auslesen ist `extract-min`. Weil die Warteschlange auf einem Heap basiert, liegen alle Operationen in $O(\log n)$.

9.13 s-t-Netzwerke

Gerichteter Graph mit gewichteten Kanten, mit einer Quelle s und einer Senke t, wobei man bequemlicherweise annimmt, dass jeder Knoten auf einem Pfad von s nach t liegt. Zwei Kanten (u, v) und (v, u) können verschiedene Gewichte haben.

9.13.1 Residualnetzwerk

"Netzwerk minus Fluß = Residualnetzwerk"
Im Residualnetzwerk stellen wir die Kanten folgendermaßen für alle $u, v \in V_G$ mit G als originales s-t-Netzwerk:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{falls } (u,v) \in E_G, \\ f(v,u) & \text{falls } (v,u) \in E_G, \\ 0 & \text{sonst} \end{cases}$$

Also für jede Kante schreibe in die ursprüngliche Richtung die Restkapazität und in die entgegengesetzte Richtung den Fluss. Es muss dann natürlich immer gelten, dass die Gewichte der beiden Kanten zwischen zwei Knoten zusammen die Kapazität ergeben.

9.13.2 Augmentierende Pfade

Ein augmentierender Pfad in G ist ein s-t-Pfad mit Restkapazität größer 0.

9.13.3 Ford-Fulkerson-Algorithmus

Initialisiere $f = 0$. Solange es einen augmentierenden Pfad f von s nach t gibt, augmentiere f entlang p. Mithilfe des Residualnetzwerks lässt sich feststellen, ob es noch augmentierende Pfade gibt. Gib f schließlich zurück.

9.14 Schnitte in Netzwerken

In einem s-t-Netzwerk mit maximalem Fluss alle Kanten als sicher betrachten, welche noch Restkapazitäten haben. Achtung, auch bei voller Kapazität einer Kante besitzt die Gegenrichtung noch Restkapazität und ist somit noch sicher. Alle Knoten, welche noch von s aus über sichere Kanten erreicht werden können gehören zur Menge S. Die restlichen gehören zu T. Alle Kanten, die S und T verbinden gehören zum minimalen Schnitt.

9.15 Bipartites Matching

Der Algorithmus von Edmond und Karp dient dem finden des maximalen Flusses zwischen zwei Knoten s und t in einem gerichteten Graphen.
Vorgehen:

1. Einen Pfad von s nach t finden.
2. das geringste Gewicht des Pfades durchschieben.
3. Wiederholen, bis keine Pfade mehr da sind.

9.16 Minimaler Spannbaum

Für einen zusammenhängenden, gewichteten Graphen $G(V, E)$ ist ein minimaler Spannbaum ein Teilgraph von G, sodass dieser kreisfrei ist, alle Knoten V enthält und für $F \subseteq E$ gilt, dass die Summe der Kantenlängen aus F die kleinstmögliche ist, die alle anderen Bedingungen erfüllt.

Kann auch bei Bipartitem Matching erweitert werden auf zwei Teilmengen von E, genannt S und T, wobei in dem Fall die Lösung allgemein nicht eindeutig ist.

9.17 Algorithmus von Prim

Verwendung: minimalen Spannbaum berechnen für zusammenhängende Graphen. Anleitung:

1. Beliebigen Knoten auswählen.
2. Kleinste zu diesem Knoten inzidente Kante in den Teilgraphen mit aufnehmen, durch die kein Kreis entsteht.
3. Kleinste zu dem Teilgraphen inzidente Kante in den Teilgraphen aufnehmen, durch die kein Kreis ensteht. Ende falls das nicht möglich ist.

9.18 Greedy-Algorithmen

Greedy-Algorithmen sind an jeder Stelle *gierig*, sie wählen immer die lokal am besten scheinende Option. In Matroiden (siehe nächster Abschnitt) ist diese Strategie nachweislich optimal, während als Gegenbeispiel bei Schach die nur im aktuellen Schritt beste Aktion (bspw. gemessen an eliminierten Figuren) zu kurzsichtig ist.

9.19 Matroid

Mit Matroiden lassen sich Greedy-Algorithmen beweisen. Dabei besteht der Matroid $M = (S, \mathcal{I})$ aus Basis S und Familie $\mathcal{I} \subseteq \text{Pot}(S)$ von *unabhängigen Mengen* mit

- Falls $A \subseteq B$ und $B \subseteq \mathcal{I}$, dann $A \in \mathcal{I}$ (M ist *hereditary*)
- Falls $A, B \subseteq \mathcal{I}$ und $|A| < |B|$, dann gibt es ein $x \in B \setminus A$ sodass $A \cup \{x\} \in \mathcal{I}$ (*Austauscheigenschaft*)

Dazu noch:

- Eine Menge ist *maximal*, wenn keine echte Obermenge von ihr unabhängig ist.
- Ein *gewichtetes* Matroid hat eine Gewichtsfunktion $w : S \to \mathbb{Q}$.
- Eine Menge maximalen Gewichts unter allen unabhängigen Mengen heißt *optimal*.

Jetzt gibt es mit diesen Axiomen einige schöne Lemmata, aber für uns ist ein Satz wichtig: Der Greedy-Algorithmus berechnet eine optimale Menge in einem gewichteten Matroid.
Praktisch können wir das Matroid etwa nutzen, um zu beweisen, dass der minimale Spannbaum sich durch einen Greedy-Algorithmus berechnen lässt, weil er die optimale Menge im graphischen Matroid ist.

9.20 Kruskal

Dieser Algorithmus ist geeignet für kantengewichtete, zusammenhängende Graphen. Er dient dem Aufstellen minimaler Spannbäume.
Führe den folgenden Schritt so oft wie möglich aus: Wähle unter den noch nicht ausgewählten Kanten von G die kürzeste Kante, die mit den schon gewählten Kanten keinen Kreis bildet.

9.21 Union Find

Praktisch für Dijkstra oder Prim, wo man viel mit Mengen arbeitet. Union-Find-Strukturen beinhalten **paarweise disjunkte** Mengen und haben drei Operationen:

1. `MakeSet`(x): Erstellt eine neue Menge, dass nur aus dem Element aus x besteht.
2. `Union`(x, y): Erstellt eine neue Menge aus Mengen x und y. In der Regel werden x und y dafür konsumiert, d.h. unbrauchbar, damit alle in der Struktur enthaltenen Mengen disjunkt bleiben.
3. `Find`(x): Findet die (eindeutige) Menge, die das Element x enthält.

Wir implementieren Union-Find mit Wäldern (es geht auch anders): Zunächst ist jedes einzelne Element eine frei stehende Wurzel; `Union` hängt zwei Bäume aneinander. Außerdem verwenden wir *union by rank* und *Pfadkompression*, um die Bäume der Mengen möglichst flach zu halten. Wir merken uns bei jedem Baum den sogenannten *Rang*, der einfach die Tiefe des Baumes ist.

1. `MakeSet`(x): Ein Baum bestehend nur aus der Wurzel x.
2. `Union`(x, y): Hänge den kleineren Baum `Find`(x) oder `Find`(y) unter die Wurzel im größeren Baum. Das ist *union by rank*: Wir hängen immer den kleineren Baum in den größeren hinein. Wenn beide Bäume gleichen Rang haben, hänge `Find`(x) in `Find`(y) und nur in diesem Fall müssen wir den Rang um eins erhöhen.
3. `Find`(x): Durchlaufe alle Bäume und suche darin nach dem Element. Dabei *Pfadkompression*: Setze den Elternknoten des gesuchten Knotens und die aller Knoten auf dem Weg zu x auf x.

Es ergibt sich schöne Performance: Ein Baum mit m Elementen in einer solchen Datenstruktur hat höchstens die Höhe $\log(m)+1$. Außerdem sind Operationen in einem anfangs leeren Baum in $O(m \log m)$; `Union` und `Find` sind dann in $O(\log m)$.