# Algorithmen 1 Zusammenfassung

Jean-Pierre v. d. Heydt

19. Juli 2018

# Inhaltsverzeichnis

# 1 Einführung

## 1.1 Pseudocode

„Mir wäre lieber, der Programmcode würde aussehen wie Pseudocode.“ -Müller Quade

**Zuweisung** $:=$

**Kommentar** //

**Ausdrücke** Reguläre Verwendung mathematischer Ausdrücke

**Deklaration** $x = c$ : Digit $(c \in \mathbb{N})$

**Array** $a$ : Array$[0 \dots n-1]$ of Digit

**Schleifen** for, while, repeat… until,…

**assert** Aussage über den Zustand der Programmausführung (Z.b. Schleifeninvarianten)

**invariant** Aussage, die an „vielen“ Stellen im Programm stimmt

## 1.2 O-Kalkül

**Definitionen:**

1. $O(f(n)) := \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : g(n) \leq c * f(n)\}$
2. $\Omega(f(n)) := \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : g(n) \geq c * f(n)\}$
3. $\Theta(f(n)) := O(f(n)) \cap \Omega(f(n))$
4. $o(f(n)) := \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : g(n) \leq c * f(n)\}$
5. $\omega(f(n)) := \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : g(n) \geq c * f(n)\}$

Die Anwendung dieser Definitionen hängt unter anderem von der konkreten Verwendung eines Maschinenmodells ab. Dies wird in der Vorlesung aber ausdrücklich nicht behandelt.

**Beweismöglichkeiten:**

Für nicht-negative $f, g \colon \mathbb{N} \to \mathbb{R}$ gelten folgende Aussagen:

- $f(n) \in o(g(n)) \iff \lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- $f(n) \in \omega(g(n)) \iff \lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ Notiz: Folien sehen hier limsup vor. Ich halte das für falsch.
- $f(n) \in \Theta(g(n)) \impliedby 0 < \lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$
- $f(n) \in O(g(n)) \iff 0 \leq \limsup\limits_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$

- $f(n) \in \Omega(g(n)) \iff 0 < \liminf_{n\to\infty} \frac{f(n)}{g(n)} = c \leq \infty$

Besonders beim Master Theorem anwendbar ist:

- $f_1, f_2 \in \Theta(g(n)), \forall n \in \mathbb{N} : f_1(n) \leq f(n) \leq f_2(n) \implies f \in \Theta(g(n))$

### 1.3 Master Theorem

Mit positive konstanten $a, b, c, d$ und $n = b^k, k \in \mathbb{N}$ gilt für:

$$T(n) = \begin{cases} a & \text{falls } n = 1 \\ cn + dT(n/b) & \text{falls } n > 1 \end{cases}$$

,

$$T(n) \in \begin{cases} \Theta(n) & \text{falls } d < b \\ \Theta(n \log(n)) & \text{falls } d = b \\ \Theta(n^{\log_b(d)}) & \text{falls } d > b \end{cases}$$

Dies gilt auch, wenn $x$ in $T(x)$ zur entsprechenden k-ten Potenz aufgerundet wird.

### 1.4 Invarianten

Beweis von Schleifeninvarianten:

1. **Initialisierung:** Invariante gilt vor der ersten Iteration.
2. **Fortsetzung:** Invariante gilt vor Iteration i. ⇒ Invariante gilt vor Iteration i+1.
3. **Terminierung:** Abbruchbedingung erfüllt ∧ Invariante gilt ⇒ richtiges Ergebnis.

## 2 Folgen, Felder, Listen

| Operation | List | SList | UArray | CArray | Erklärung'*' |
|---|---|---|---|---|---|
| $[\cdot]^*$ | n | n | 1 | 1 | heraussuchen eines Elements |
| $\lvert\cdot\rvert$ | $1^*$ | $1^*$ | 1 | 1 | nicht bei splices zwischen verschiedenen Listen |
| first | 1 | 1 | 1 | 1 | |
| last | 1 | 1 | 1 | 1 | |
| insert | 1 | $1^*$ | n | n | nur insertAfter |
| remove | 1 | $1^*$ | n | n | nur removeAfter |
| pushBack | 1 | 1 | $1^*$ | $1^*$ | armortisiert |
| pushFront | 1 | 1 | n | $1^*$ | armortisiert |
| popBack | 1 | n | $1^*$ | $1^*$ | armortisiert |
| popFront | 1 | 1 | n | $1^*$ | armortisiert |
| concat | 1 | 1 | n | n | |
| splice | 1 | 1 | n | n | |
| findNext | n | n | $n^*$ | $n^*$ | Cache-effizient |

## 2.1 Listen

**Dummy-Element** Der Vorteil von Dummy-Elementen, ist die Vermeidung vieler Sonderfälle und das Einhalten der Invariante. Allerdings verbraucht ein Dummy-Element in der Regel mehr Speicherplatz.

### Ausgewählte Operationen

- **head(): Handle** gibt den Kopf der Liste zurück
- **isEmpty(): boolean**
- **first(): Handle** gibt das erste Element in der Liste zurück
- $[i : \mathbb{N}]$**: Element** gibt das Element an dem gegebenen Index i zurück. Liegt in $O(n)$.
- **splice(a,b,t: Handle): void** Hilfsfunktion. Schneidet $\langle a, \ldots, b \rangle$ aus und fügt es hinter t ein. Ist in $O(1)$ möglich.
- **moveAfter(b,a: Handle): void** bewegt das Element b hinter a.
- **remove(b: Handle): void** entfernt b aus der List.
- **insertAfter(x: Element, a: Handle): Handle** Fügt das Element x in die Liste ein und gibt den neuen Handle zurück.
- **findNext(x: Element, from: Handle): Handle** sucht vom Handle from nach dem Element x. Liegt in $O(n)$.

**Vergleich einfach und doppelt verketteter Listen** Einfach verkettete Listen benötigen weniger Speicherplatz, was auch oft gleichbedeutend ist mit weniger Zeit. Sie sind jedoch von ihrer Funktionsweise jedoch eingeschränkter (z.B. kein remove).

## 2.2 Arrays

### Ausgewählte Operationen

- $[i : \mathbb{N}]$**: Element** gibt das Element an dem gegebenen Index i zurück. Liegt in $O(1)$.
- **pushBack(): void** Hängt Element ans Ende an und erhöht die Größe um eins. Wenn kein Platz mehr vorhanden ist, wird das komplette Array umkopiert und größer angelegt.

**Armortisierte Analyse** Jede Operationsfolge $\langle \sigma_1, \ldots, \sigma_m \rangle$ von pushBack und popBack Operationen auf ein anfänglich leeres, unbeschränktes Array liegt in $O(m)$.

$$O(\underbrace{c \cdot m}_{Gesamtzeit} / \underbrace{m}_{\#Operationen}) = O(1)$$

Bei der **Konto-Methode** werden für die günstigen Operationen Tokens eingezahlt, während bei den teuren Operationen Tokens ausgezahlt werden. Die Anzahl der Tokens dürfen dich nur um einen konstanten Faktor von der Zeitkomplexität der Operation unterscheiden. Ziel ist es, dass das

Konto bei jeglichen Eingaben nicht überzogen wird.
Mit anderen Worten: Die angenommene amortisierten Kosten sind korrekt, wenn

$$\sum_{i=1}^{n} T_{Op_i} \leq c \cdot \sum_{i=1}^{n} A_{Op_i}$$

wobei $T_{Op}$ die Tatsächlichen Kosten und $A_{Op}$ die amortisierten Kosten der Operationsfolge sind.

### 2.3 Weitere Datenstrukturen für Folgen

**Stack** Elemente können draufgelegt und runter genommen werden (Last in First out). Wird bei Tiefensuche verwendet.

**FIFO queue** (First in First out). Wird bei Breitensuche verwendet.

**Deque** Steht für double ended queue. An beiden Enden können Elemente verwendet werden.

**Cyclic Array** Besonders gut um FIFOS zu realisieren. hat aber eine feste Größe.

**Skip List** Randomisierte Datenstruktur mit Einfügen und Lookup erwartet in $O(\log(n))$.

**Hotlist** Mischung aus Array und Liste. Einfügen, Löschen und Suchen amortisiert in $O(\sqrt{n})$.

## 3 Hashing

**Bestandteile und Konventionen für Hashtabellen**

- Speichern einer Menge $M$ von Elementen.
- Statt Elementen aus $M$ verwenden wir Schlüssel aus der Menge $S$.
- für $e \in M$ gibt es die Funktion $key\colon M \to S$ welche $key(e)$ einen Schlüssel zuweist.
- wir verwenden $e \in M$ oft äquivalent zu $key(e) \in S$.
- Objekte werden in der HashTabelle t mit $m$ Einträgen gespeichert.
- Eine Hash-Funktion $h\colon S \to \{0, \ldots, m-1\}$ weist Schlüssel einem Platz in der Hashtabelle zu.
- Eine Hashtabelle stellt die Funktionen insert($e$), remove($e$) und find($e$) erwartet in $O(1)$ bereit.

### 3.1 Hash-Funktion

Eine perfekte Hash-Funktion $h$ bildet Elemente der Menge $M$ eindeutig auf die Einträge von t ab. Perfekte Hash-Funktionen sind im Allgemeinen aber nicht realistisch, da zwei verschiedene Elemente sehr schnell auf den gleichen Eintrag abgebildet werden (vgl. Geburtstagsparadox).

- Eine Hashfunktion ist nur dann mit vernünftiger Wahrscheinlichkeit perfekt, wenn die Größe der Hashtabelle $m$ quadratisch zur Anzahl der Elemente ist ($m \in \Omega(n^2)$).
- Eine Teilmenge von Hashfunktionen $\mathcal{H} \subset \{h\colon S \to \{0, \ldots, m-1\}\}$ ist universell, falls für alle $x, y \in Keys, x \neq y$ und zufälligem $h \in \mathcal{H}\colon \mathbb{P}[h(x) = h(y)] = \frac{1}{m}$.
- Die Erwartete Anzahl kollidierender Elemente ist $O(1)$, falls $|M| \in O(m)$.

### 3.2 Hashing mit verketteten Listen

Interpretiere die Tabelleneinträge als einfach verkettete Listen und führe bei Kollisionen ein neues Element in die Liste ein. insert($e$), remove($e$) und find($e$) werden wie erwartet implementiert. Vorteile sind hier:

- Weniger anfällig gegen Volllaufen
- Es kann weiterhin von Außen auf Elemente Zugegriffen werden.
- im Gegensatz zur Linearen Suche gibt es Leistungsgarantien

### 3.3 Hashing mit Linearer Suche

Elemente werden direkt in die Tabelle eingespeichert. Kollisionen werden gelöst, indem der nächst freie Platz gesucht wird. Dadurch entsteht zusätzlicher Aufwand beim Löschen. Vorteile sind vor Allem:

- Platz-effizient
- Cache-effizient

## 4 Sortieren

Sortieralgorithmen finden überall in der Algorithmik Anwendung. Sie werden in der Vorverarbeitung, dem Aufbau von Such- oder Spannbäumen, greedy Algorithmen und vielen anderen Problemen verwendet.

### 4.1 Untere Schranke

Deterministische vergeliechsbasierte Sortieralgorithmen brauchen im schlechtesten Fall mindestens $\Omega(n \log(n))$ Vergleiche.

**Beweisskizze** Sei eine Folge mit $n$ Elementen zu sortieren. Nun gibt es $n!$ mögliche Permutationen der richtigen Reihenfolge dieser Elemente. Eine Baum-basierte Sortierer-Darstellung muss also mindestens $n!$ Blätter haben, die Ausführungszeit entspricht der Tiefe $T$ des Baumes.
Da ein Baum der Höhe $T$ höchstens $2^T$ Blätter haben kann folgt (über Umwege) die Abschätzung:

$$2^T \geq n! \implies T \geq n \log(n) - O(n)$$

Es gibt auch Beweise für randomisierte Algorithmen.

### 4.2 Insertionsort

**Algorithm 1:** insertionSort

```
   Data: a: Array[1...n] of Element
1  for i := 2 to n do
2  |  invariant a[1] ≤ ··· ≤ a[i − 1];
3  |  /* move a[i] to the right place                                  */
```

**Laufzeit** Liegt im worst case in $O(n^2)$ und im best case in $O(n)$.

**Sonstiges**

- inplace möglich
- Ein Beispiel für das mögliche Rausziehen einer Randbedingung aus der Schleife und Verwenden eines Sentinels.

## 4.3 Mergesort

**Algorithm 2:** mergeSort

**Data:** $\langle e_1, \ldots, e_n \rangle$: of Element
**if** $n = 1$ **then**
    `/* Basisfall */`
    return $e_1$
**else**
    `/* Zusammenfügen beider Teillisten */`
    return merge(mergeSort($\langle e_1, \ldots, e_{\lfloor n/2 \rfloor} \rangle$),mergeSort($\langle e_{\lfloor n/2 \rfloor + 1} \ldots, e_n \rangle$))

**Laufzeit** Liegt in deterministisch $\Theta(n \log(n))$.

**Sonstiges**

- Der Algorithmus ist stabil (gleiche Elemente behalten gleiche Reihenfolge).

## 4.4 Quicksort

**Algorithm 3:** quickSort

**Data:** s: Sequence of Element
**if** $|s| \leq 1$ **then**
    `/* Basisfall */`
    return s
**else**
    `/* wähle Pivotelement */`
    wähle ein gewisses $p \in s$
    $a \coloneqq \langle e \in s \colon e < p \rangle$
    $a \coloneqq \langle e \in s \colon e = p \rangle$
    $a \coloneqq \langle e \in s \colon e > p \rangle$
    return Konkatenation von quickSort(a), b und quickSort(c)

**Laufzeit** Kann im worst case in $O(n^2)$ liegen. Im best case würde immer der Median ausgewählt und die Laufzeit läge in $O(n \log(n))$. Bei zufälliger Wahl des Pivotelements liegt der Algorithmus erwartet in $O(n \log(n))$.

**Beweisskizze für erwartete Laufzeit**

- Bei zufälliger Pivot-Wahl ist das Aufspaltverhältnis mit einer Wahrscheinlichkeit von $\frac{1}{2}$ nicht schlechter als $\frac{1}{4} : \frac{3}{4}$.
- Die Wahrscheinlichkeit, dass $e_i$ und $e_j$ miteinander verglichen werden ist $\frac{2}{j-i+1}$.
- Die Anzahl aller Vergleiche ist $\sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \leq 2n \sum_{k=2}^{n} \frac{1}{k} \leq 2n \ln(n)$.
- Die letzte Abschätzung ist eine Abschätzung der harmonischen Summe: $\ln(n) \leq \sum_{i=1}^{n} \frac{1}{i} \leq 1 + \ln(n)$.

**Sonstiges**

- Es gibt eine Effiziente inplace Implementierung. Im Methodenkopf werden die Grenzen des zu sortierenden Array-Teils übergeben. Hier wurde noch nicht der Rekursionsstapel beachtet, welcher $O(\log(n))$ Speicher benötigt.
- Quickselect ist eine Variante, welche das k-te Element einer Folge in $O(n)$ raussucht.

## 4.5 Bucketsort

**Algorithm 4:** KSort

```
Data: s: Sequence of Element
b = ⟨⟨⟩, ..., ⟨⟩⟩ : Array[0, ..., K − 1] of Sequence of Element
for each e ∈ s do
    b[key(e)].pushBack(e)
s := Konkatenation von b[0], ..., b[K − 1]
```

**Laufzeit** Laufzeit liegt in $O(n + K)$.

**Algorithm 5:** LSDRadixSort

```
Data: s: Sequence of Element
for i := 0 to d − 1 do
    /* wählt i-te Stelle der zahl als key                    */
    definiere key(x) als (x div K^i) mod K
    KSort(s)
```

**Laufzeit** Laufzeit liegt in $O(d(n + K))$ wobei $d$ die Anzahl der Stellen der größten Zahl ist und $K$ die Basis.

**Sonstiges**

- Ist nicht inplace.
- Ganzzahlige Sortieralgorithmen sind asymptotisch schnelle machen aber mehr Annahmen als vergleichsbasierte über die zu sortierenden Daten.

# 5 Prioritätslisten

Verwaltet eine Menge $M$ von Elementen mit Schlüssel und stllt im wesentlichen die Operationen insert(e) und deleteMin bereit. Die zentrale Eigenschaft eines Heaps ist:

$$\text{Baum mit } \forall v \in M : \text{parent}(v) \leq v$$

## 5.1 Heap

Ein binärer Heap mit n Elementen ist ein Binärbaum der Höhe $\lfloor \log(n) \rfloor$. Fehlende Blätter stehen unten rechts. Das Minimum Des Baumes steht immer an seiner Wurzel.

**Implizite Baum-Repräsentation** Der Heap wird in einem Array gespeichert. Hierbei wird jede Schicht des Baumes nacheinander im Array abgelegt. Dabei ergibt sich: parent(j) = $\lfloor j/2 \rfloor$, linkesKind(j) = $2j$ und rechtesKind(j) = $2j + 1$.

Eine Wichtige Hilfsoperation, die beim Einfügen neuer Elemente in den Heap gebraucht wird ist siftUp. Hingegen wird siftDown beim Entfernen des kleinsten Elements gebraucht.

**Algorithm 6:** siftUp

```
Data: i: ℕ
if i = 1 oder h[parent(i)] ≤ h[i] then
    return
else
    tausche(h[parent(i)], h[i])
    siftUp(parent(i))
```

**Algorithm 7:** siftDown

```
Data: i: ℕ
/* Überprüfe ob i mindestens ein Kind hat                    */
if linkesKind(i) ≤ n then
    /* wähle kleineres Kind aus                              */
    if rechtesKind(i) > n oder h[linkesKind(i)] ≤ h[rechtesKind(i)] then
        m := linkesKind(i)
    else
        m := rechtesKind(i)
    if h[i] > h[m] then
        tausche(h[i], h[m])
        siftDown(m)
```

**Laufzeit und Heapsort** min liegt in $O(1)$. insert und deleteMin liegen beiden in $O(\log(n))$. Um einen Heap zu bauen wird $n/2$ mal siftDown ausgeführt, dies liegt in $O(n)$. Somit ist auch ein Heapsort möglich. Dieser liegt in $O(n \log(n))$ und ist inplace.

### 5.2 Adressierbare Prioritätslisten

Eine adressierbare Prioritätsliste biete im Wesentlichen die gleichen Funktionalitäten wie ein Heap, besitzt aber noch die Möglichkeit den Key eines bereits eingefügten Elements zu verringern und beliebige Elemente aus der Liste zu entfernen.
Solch adressierbare Prioritätslisten finden vor Allem bei Greedy-Algorithmen Anwendung. Die Implementierung unterscheidet sich nicht gtoß von einem normalen Heap, grob gesagt, werden jetzt Proxy-Elemente vorgeschoben.

**Fibonacci-Heap** besonders beliebt unter Algorithmikern ist der Fibonacci-Heap, der sich durch insert, remove und updateKey amortisiert in $O(1)$ und deleteMin in $O(\log(n))$ auszeichnet. In der Praxis hat dieser jedoch zu hohe konstante Faktoren.

## 6 Sortierte Liste

Wird gelegentlich als Synonym für Suchbaum verwendet.

### 6.1 Funktionen

- **locate(k: Element)** Ist die kennzeichnende Funktion einer sortierten Liste. Mit Hilfe einer binären Suche wird in $O(\log(n))$ das Element gefunden, welches als erstes größer als k ist ($M.\text{locate}(k) := \min\{e \in M : e \geq k\}$).
- **min(): Element, max(): Element** in $O(1)$.
- **insert(e: Element), remove(e: Element)** in $O(\log(n))$ für $a, b \in O(1)$.
- **rangeSearch(a,b: Element): List of Element** gibt alle Elemente zurück, die größer gleich a und kleiner gleich b sind. Liegt in $O(\log(n) + |\text{result}|)$.
- **build()** wird auf einer sortierten Liste aufgerufen. In $O(n)$
- **concate(s: Sortierte Folge), split(i: $\mathbb{N}$)** in $O(\log(n))$
- **merge(N: Sortierte Folge, M: Sortierte Folge)** wenn $n = |N| \leq m = |M|$ in $O(n \log(\frac{m}{n}))$.

### 6.2 Vergleich zu anderen Datenstrukturen

- **Hash-Tabelle** hat insert, remove und find aber kein locate
- **Prioritätslisten** hat insert, delteMin und schnelleres mergen aber **min()** nicht in $O(1)$.

### 6.3 [a,b]-Bäume

Alle Knoten (abgesehen von den Blättern und der Wurzel) haben einen Grad zwischen a und b. Es muss gelten: $a \geq 2$ und $b \geq 2a - 1$. Die Höhe des Baumes $h$ ist: $h \leq 1 + \lfloor \log_a \frac{n+1}{2} \rfloor$.

**Insert und Remove** unsere (a,b)-Bäume verwenden zwei etwas kompliziertere Funktionen an, die im Folgenden kurz angerissen werden. Insert spaltet einen Knoten in zwei kleinere auf, wenn der neu entstehende Knoten zu groß wäre. Remove fügt zwei Knoten zusammen oder balanciert diese, wenn der entstehende Knoten zu klein wäre.

**Algorithm 8:** insert

```
   Data: e: Element
1  füge e an die richtige Stelle in der Liste ein
2  füge key(e) als neuen Splitter in den Vorgänger u ein
3  /* Grad zu groß                                              */
4  if u.d = b + 1 then
5  |  splitte u in 2 Knoten mit Graden ⌊(b+1)/2⌋, ⌈(b+1)/2⌉
6  |  weiter oben einfügen, ggf. spalten, ggf. neue Wurzel
```

**Algorithm 9:** remove

```
   Data: e: Element
1  finde Pfad zum Element e
2  entferne e aus der List
3  entferne key(e) in Vorgänger u
4  /* Grad zu klein                                             */
5  if u.d = a − 1 then
6  |  finde Nachbarn u'
7  |  /* wenn zusammenfügen nicht zu groß                       */
8  |  if u'.d + a − 1 ≤ b then
9  |  |  fuse(u', u)
10 |  |  Weiter oben splitter entfernen
11 |  |  ggf. Wurzel entfernen
12 |  else
13 |  |  balane(u', u)
```

# 7 Graphrepräsentationen

Wir repräsentierten ungerichtete Graphen im Rechner meist als doppelt gerichtete Graphen.

### Begriffe

- **DAG** $G = (V, E)$ DAG $\iff$ $G$ gerichtet und $G$ enthält keine Zyklen. $\implies$ $G$ lässt sich als obere Dreieckmatrix repräsentieren.
- **verbindbar** zwei Knoten sind genau dann verbindbar, wenn ein Pfad zwischen ihnen existiert.
- **zusammenhängend** ein Graph ist genau dann zusammenhängend, wenn alle Knoten verbindbar sind

### 7.1 Triviale Darstellung

Der Graph wird als Folge von Knotenpaaren gespeichert. Diese Variante ist kompakt und gut für die Ein- und Ausgabe aber bietet fast keine nützlichen Operationen in angemessener Zeit.

### 7.2 Adjazenzfelder

Knoten werden den Zahlen 1,...,n zugeordnet. Knotenfeld $V$ hat $n+1$ Felder und speichert den Index der ersten ausgehenden Kante in $E$. Kantenfeld $E$ speichert Ziele der Kante gruppiert nach Startknoten. Dummy-Eintrag $V[n+1]$ speichert $m+1$.
Ist in der Regel besser für statische Graphen geeignet.

### 7.3 Adjazenzliste

Ähnlich wie Adjazenzfeld aber speichert das Kantendeld als einzelne Listen. Dadurch werden Einfügen und Löschen vereinfacht aber auch mehr Platz verbraucht und mehr Cache-Misses verursacht.

### 7.4 Adjazenz-Matrix

Speichert die Kanten in einer $n \times n$-Matrix C wobei eine 1 für $c_{ij}$ eine Kante von $i$ nach $j$ repräsentiert. Diese Darstellung ist plazeffizient für dichte Graphen aber ineffizient für dünne Graphen.

### 7.5 Algorithmen

**Algorithm 10:** isDAG

**Data:** $G = (V, E)$: Graph

```
1 /* Graph behält die Eigenschaft von vorhandenen bzw. nicht vorhandenen
     Zyklen                                                          */
```

2 **while** $\exists v \in V : \mathit{indegree}(v) = 0$ **do**

3 ⌊ entferne $v$ und alle eingehenden Kanten

4 return $|V| == 0$

**Laufzeit** Für $n = |V|, m = |E|$ liegt die Laufzeit in $O(m+n)$. Der Algorithmus kann gut mit einem Stack für alle Knoten mit Eingangsgrad 0 implementiert werden.

## 8 Graphtraversierung

### 8.1 Kantenklassifizierung

Graphtraversierung hilft uns auch immer dabei, die Kanten eines Baumes zu klassifizieren. Wir unterscheiden dabei zwischen folgenden Kanten:

- **tree** Kante ist ein Element des Waldes, das bei der Suche gebaut wird
- **forward** Kanten verlaufen parallel zu Wegen aus Baumkanten
- **backward** Kanten verlaufen antiparallel zu Wegen aus Baumkanten

- **cross** Kanten verlaufen zwischen zwei Ästen eines Baumes.

## 8.2 Breitensuche

Die Breitensuche baut einen Baum auf Grundlage eines Graphen (ungewichtet), der von einem Startknoten $s$ alle von $s$ erreichbaren Knoten mit möglichst kurzen Pfaden erreicht.

**Algorithm 11:** BFS

```
1  /* BFS steht für breadth-first search                    */
   Data: s: Knoten
2  Q := ⟨s⟩: FIFO queue oder Stack
3  while Q ≠ ⟨⟩ do
4  |  exploriere Knoten in Q
5  |  merke Knoten der nächsten Schicht in Q'
6  |  Q := Q'
```

### Vorteile/Nachteile

- nicht rekursiv
- keine Vorwärtskanten
- findet kürzeste Wege, womit Umgebung eines Knoten definiert werden kann
- Bei DAGs sollten zuerst die Wurzeln gesucht werden

## 8.3 Tiefensuche

**Algorithm 12:** DFS

```
1  /* DFS steht für depth-first search                     */
2  /* DFS wird mit DFS(s,s) initialisiert                  */
   Data: u, v: Knoten
3  /* Erkunde v von u kommend                              */
4  for each (v, w) ∈ E do
5  |  if w is marked then
6  |  |  traverseNonTreeEdge(v, w)
7  |  else
8  |  |  traverseTreeEdge(v, w);
9  |  |  markiere w;
10 |  |  DFS(v, w)
11 backTrack(u, v)
```

### Vorteile

- benötigt keine extra Datenstruktur, da er eine Rekursionsstapel verwendet
- sehr flexibel einsetzbar

**DFS-Nummerierung** wird auf 1 initialisiert, in traverseTreeEdge aufgerufen und bei dem entsprechenden Knoten um 1 erhöht. Somit bekommt der Startknoten die Nummer 1 und der zuletzt besuchte knoten die Nummer m.

**Fertigstellungszeit** wird auf 1 initialisiert, in backTrack aufgerufen und bei dem entsprechenden Knoten um 1 erhöht. Somit bekommt der Startknoten die Nummer m.

**Kantenklassifizierung bei DFS**

| Typ $(v, w)$ | dfsNum$[v]$ < dfsNum$[w]$ | finishTime$[v]$ < finishTime$[w]$ | $w$ ist markiert |
|---|---|---|---|
| tree | yes | yes | no |
| forward | yes | yes | yes |
| backward | no | no | yes |
| cross | no | yes | yes |

**Topologische Sortierung** Eine Sortierung $t$ von Knoten eines Graphen $G = (VE)$ heißt topologische Sortierung, wenn gilt:

$$\forall (u, v) \in E \colon t(u) < t(v)$$

Mithilfe von DFS kann eine topologische Sortierung für einen Graphen bestimmt werden. Dazu gilt folgendes Theorem:
$G$ kreisfrei (DAG) $\iff$ DFS findet keine Rückwärtskante. Dann liefert $t(v) \coloneqq n - \text{finishTime}[v]$ eine topologische Sortierung.

# 9 Kürzeste Wege

Wir betrachten nun gewichtete Graphen. Das heißt, wir haben eine Kostenfunktion $c \colon E \to \mathbb{R}$, die jeder Kante einen Wert zuweist. Unser Ziel ist es nun den Pfad mit den niedrigsten Kantengewichten zwischen zwei Knoten zu finden.

## 9.1 Grundlagen

- Der kürzeste Pfad zwischen zwei Knoten ist nicht eindeutig bestimmbar, wenn es negative Kreise gibt.
- Wennin einem Graphen mit negativen Kantengewichten ein kürzester Pfad existiert, ist dieser Schleifenfrei.
- Teilpfade kürzester Pfade sind selbst kürzeste Pfade

### 9.2 Dijkstras Algorithmus

**Algorithm 13:** dijkstra

**Data:** $G = (V, E)$: Graph, $s$: Knoten
`/* vorläufig kürzeste Distanz */`
$d$: Array$[1 \dots |V|]$ of Digit
`/* Vorgänger auf dem vorläufig kürzesten Weg */`
parent: Array$[1 \dots |V|]$ of Knoten
$d[s] := 0$, parent$[s] := s$
ansonsten setzte $d[v] := \infty$ und parent$[v] :=\perp$
setze alle Knoten auf nicht gescannt
**while** $\exists$ *ein nicht gescannter Knoten* $u$ *mit* $d[u] < \infty$ **do**
  `/* verwende adressierbare Prioritätsliste */`
  $u :=$ Knoten $m$, der von allen ungescannten Knoten das kleinste $d[m]$ hat
  relaxiere alle Kanten $(u, v)$ von u
  markiere $u$ als gescannt

**Laufzeit** Ein wesentlicher Bestandteil von Dijkstras Algorithmus ist eine adressierbare Prioritätsliste. Daher hängt auch die Laufzeit im wesentlichen von dieser ab.

$$T_{\text{Dijkstra}} = O(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n)))$$

Mit unser behandelten Prioritätsliste kommen wir auf $O((m+n)\log(n))$, ein Fibonacci-Heap kommt auf $O(m + n\log(n))$.

### 9.3 Bellman-Ford-Algorithmus

Dieser Algorithmus relaxiert alle Kanten (in irgendeiner Reihenfolge) $|V| - 1$ mal. Dadurch werden alle kürzesten Pfade (ausgenommen $-\infty$) gefunden. Dieser Algorithmus kommt also auch mit negative Kantengewichten zurecht.
Jeder negative Kreis hat nun mindestens einen Knoten, der sich bei erneuter Relaxion vom Gewicht her verringert. Alle von diesen Knoten aus erreichbaren Kanten haben $-\infty$ als Gewicht.

**Laufzeit** liegt in $O(n \cdot m)$.

## 10 Minimale Spannbäume

Bei minimalen Spannbäumen arbeiten wir mit ungerichteten Graphen, die positiven Kantengewichte besitzen. Unser Ziel ist es, einen minimalen Baum zu finden, der alle Knoten verbindet

### 10.1 Schnitteigenschaft

Für eine beliebige Teilmenge $S \subset V$ betrachte die Menge der Schnittkanten

$$C = \{\{u, v\} \in E \colon u \in S, v \in V \setminus S\}$$

Die leichteste Kante in $C$ kann in einem minimalem Spannbaum verwendet werden.

## 10.2 Kreiseigenschaft

Die schwerste Kante auf einem Kreis wird nicht für den minimalen Spannbaum verwendet.

## 10.3 Jarník-Prim-Algorithmus

**Algorithm 14:** jpMST

**Data:** $G = (V, E)$: ungerichteter Graph(zusammenhängend)

```
1 T := ⟨⟩
2 S := {s} für einen beliebigen Startknoten
3 for i = 1, i ≤ |V| − 1 do
4     /* Hier kann gut eine Prioritätsliste verwendet werden */
5     finde (u, v), welche die Schnitteigenschaft für S erfüllt
6     S := S ∪ {v}
7     T := T ∪ {(u, v)}
8 return (S, T)
```

**Laufzeit** liegt in $O((m+n)\log(n))$ mit unseren Heaps und $O(m + n\log(n))$ mit Fibonacci-Heaps

**Vorteile**

- Asymptotisch gut für $m \gg n$.

## 10.4 Kruskals Algorithmus

**Algorithm 15:** Kruskals Algorithmus

**Data:** $G = (V, E)$: ungerichteter Graph

```
1 /* Hier wird der Union-Find verwendet */
2 Tc := UnionFind(|V|)
3 T := ⟨⟩
4 for each (u, v) ∈ E in absteigender Reihenfolge zum Gewicht do
5     /* Hier wird Schnitt-/ Kreiseigenschaft verwendet */
6     if u und v sind in verschiedenen Teilmengen von Tc then
7         T := T ∪ {(u, v)}
8         Tc.union(u, v)
9 return T
```

**Union-Find** Wir benötigen eine effiziente Datenstruktur, um zu erkennen, ob zwei Knoten in verschiedenen Teilbäumen sind.
Diese Datenstruktur biete folgende Funktionen an:

- **union**$(i, j : \mathbb{N})$ fügt die Blöcke, die $i$ und $j$ enthalten zu einem neuen Block zusammen. Hier wird nach der Größe des Rangs zusammengefügt.

- **find**($i\colon \mathbb{N}$) gibt einen eindeutigen Identifier für den Block der $i$ enthält zurück. Bei den find aufrufen kann Pfadkompression angewendet werden.
- Die Zeitkomplexität von $m\times$**find** und $n\times$**union** brauchen Zeit $O(m\alpha_T(m,n)$ wobei $\alpha_T$ das Inverse der Ackermannfunktion ist.

**Laufzeit** Mit der Union-Find Datenstruktur liegt der Algorithmus von Kurskal in $O(m\log(m))$ (Sortierung ist ausschlaggebend).

**Vorteile**

- gut für $m = O(n)$
- braucht nur Kantenliste.
- Profitiert von schnellem Sortieren

### 10.5 Sonstiges

- Es gibt das Steinerbaumproblem, bei dem nur eine Teilmenge aller Knoten im Spannbaum enthalten sein müssen. Dies ist eine Verallgemeinerung des MST-Problems und ist NP-Vollständig

## 11 Optimierung

### 11.1 Lineare Programmierung

Ein lineares Programm setzt sich aus $n$ Variablen und $m$ Constraints (Einschränkungen) Zusammen. Zusätzlich haben wir eine lineare Kostenfunktion $f(x) = c \cdot x$.
Lineare Programme lassen sich in polynomieller Zeit lösen.

**Ganzzahlige Lineare Programmierung** Lassen nur ganzzahlige Variablen zu. Häufig auch nur die Werte 1 oder 0. Dies Probleme sind NP-Schwer es gibt aber viele Möglichkeiten für Näherungslösungen.

### 11.2 Dynamische Programmierung

Ist Anwendbar, wenn das Optimalitätsprinzip gilt. Das bedeutet, dass optimale Lösungen aus optimalen Lösungen für Teilprobleme bestehen und bei mehrere optimalen Lösungen es egal ist, welche benutzt wird.

**Rucksackproblem** Die zentrale Eigenschaft, die wir beim Rucksackproblem ausnutzen ist, dass für die Profitfunktion $P(i,C)$, die den maximalen Profit bei der Verwendung der ersten i Gegenstände (müssen nicht zwangsweise sortiert sein) mit einer Kapacität von $C$ gilt:

$$\forall 1 \leq i \leq n\colon P(i,C) = \max\{P(i-1,C), P(i-1,C-w_i)+p_i\}$$

Auf Basis dieser Aussage können wir nun eine Tabelle von unten nach oben ausfüllen und so die optimale Lösung für Kapazität $C$ rekonstruieren.

## 11.3 Systematische Suche

Bei diesem Algorithmenschema werden alle (sinnvollen) Möglichkeiten der Problemlösung ausprobiert. Sie finden die optimale Lösung.

**Branch-And-Bound** Beispiel für systematische Suche.

**Algorithm 16:** bbKnapsack

**Data:** $(p_1, \ldots, p_n)$: ProfitVektor, $(w_1, \ldots, w_n)$: GewichtVektor, $C$: Kapazität

```
/* Profit- und KostenVektor sind in absteigender Reihenfolge zum
   Profitverhältnis p_i/w_i sortiert.                              */
/* wahlweise kann x̂ auf den NullVektor gesetzt werden. x̂ repräsentiert die
   derzeitig beste gefundene Lösung                                */
x̂ := heuristicKnapsack((p_1, ..., p_n), (w_1, ..., w_n), C): 𝕃
/* Menge von unvollständigen Lösungen, an denen in recurse gearbeitet wird
   */
x: 𝕃
recurse(1, C, 0)
```

**Algorithm 17:** recurse

**Data:** $i$: $\mathbb{N}$, $C$: Kapazität, $P$: Profit

```
/* Gut geschätzte obere! Schranke für den derzeitigen Lösungsansatz */
u := P + upperBound((p_1, ..., p_n), (w_1, ..., w_n), C)
/* Obere Schranke muss besser sein als derzeitig beste Lösung       */
if u > p · x̂ then
    /* Neue beste Lösung gefunden                                   */
    if i > n then
        x̂ := x
    else
        /* Objekt kann noch rein gepackt werden                     */
        if w_i ≤ C then
            x_i := 1
            recurse(i + 1, C − w_i, P + p_i)
        /* Ist obere Schranke immer noch gut genug?                 */
        if u > p · x̂ then
            x_i := 0
            recurse(i + 1, C, P)
```

**Laufzeit** Im Schlechtesten Fall $O(2^n)$ im Mittel aber besser. Hängt von der Qualität der upperBound Methode ab. Findet die optimale Lösung.

## 11.4 Lokale Suche

Zuerst wird eine Lösung $x \in \mathbb{L}$ (mehr oder weniger zufällig) ausgewählt und anschließend eine Lösungsumgebung $U(x)$ berechnet. In dieser Lösungsumgebung wird lokal optimiert (und $U$ geup-

dated).
Problematisch ist, dass so nur lokale Optima gefunden werden. $U$ muss gut definiert werden

„Da ich den kürzesten Weg durch die Vorlesung gefunden habe, sind wir jetzt schon fertig.“
-Müller Quade

## Index