VL-Notizen

Algorithmus Definition

> **Algorithmus**
>
> - eine endliche Folge von Regeln (Finitheit)
> - nach denen sich nach endlich vielen (Terminiertheit)
> - eindeutig festgelegten Schritten (Determiniertheit)
> - die Lösung einer Klasse von Problemen ergibt (Allgemeingültigkeit)

Mögliche Darstellungsformen:

- UML - Struktogramm - Pseudocode

Korrektheit = Richtigkeit der Lösung
Effizienz = vergleichsweise wenig Schritte. Betrachtet werden lesen, schreiben, vergleich und Sprünge)

Effizienz von Algorithmen

Beispiel: Maximale Abschnittssumme:
Summe über Zahlen in bestimmtem Bereich eines Arrays/Vektor

Variante 1: alle Teilfolgen betrachten, berechnen und vergleichen ➔ O(n^3)
Variante 2: von jedem Startpunkt aus aufsummieren ➔ O(n^2)
Variante 3: "Teile und Herrsche:" O(ld(n))

- Zeile in 2 Hälften
- suche von Mitte nach links die maximale Asm
- suche von Mitte nach rechts die maximale Asm
- prüfe die maximale ASM, die über die Mitte geht

Variante 4: nur 1 Durchlauf. Aufsummierung der Werte. Falls `aktSum` kleiner 0 wird ➔ Folge neu beginnen ➔ O(n)

> **Definition Stabilität von Sortierverfahren**
>
> Ein Sort-Alg. ist *stabil*, wenn er bie Elementen die nach dem Sortierkritekum gleich sind, die zuvor vorliegende Reihenfolge der Elemente relativ zueinander behät.
> (Beispiel: Sortierung einer Liste erst nach einem Kriterium, dann nach einem anderen, sodass Elemente die nach einem kriterium gleich sind nach dem anderen Kriterium sortiert sind)
> AKA: "gleiche Elemente werden nicht vertauscht"

Landau Symbole:

Symbole

$f \in O(g)$: f wächst höchstens so schnell wie g*

$$0 \leq f(n) \leq n \cdot g(n)$$

$f \in \Omega(g)$: f wächst mindestens so schnell wie g*

$$0 \leq c \cdot g(n) \leq f(n)$$

$f \in \Theta(g)$: f wächst genauso schnell wie g*

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

* jeweils ab einem bestimmten n_0

Beziehungen:
$\Theta(f) \subseteq O(f)$
$\Theta(f) \subseteq \Omega(f)$
$\Theta(f) = O(f) \cap \Omega(f)$

Sortierverfahren

Bubble-Sort (Swap-mania)

- for n..1, for n..2, for n..3, ... , for n..n-1:
  - vergleiche und tausche jeweils 2 benachbarte Zahlen
  Standardverbesserung: zusätzliche Abfrage: falls nicht getauscht wurde ➔ fertig
  - best: Θ, avg: $\Theta(n^2)$, worst: $\Theta(n^2)$

BUBBLE-SORT2(Z)

```
n ← length(Z)
repeat
    sortiert ← true
    for i ← 0 to n − 2 do
        if Z[i] > Z[i + 1] then
            sortiert ← false
            temp ← Z[i]
            Z[i] ← Z[i + 1]
            Z[i + 1] ← temp
        end if
    end for
    n ← n − 1
until sortiert = true or n ≤ 1
```

Select-Sort (immer auf die kleinen)

- suche nach kleinstem Element
- tausche das gefundene Element mit dem ersten Element
- wdh ab Pos 2...
- best Θ, avg $\Theta(n^2)$, worst $\Theta(n^2)$, NICHT STABIL

```
SELECT-SORT(Z)
n ← length(Z)
for i ← 0 to n − 1 do
    min ← i
    for j ← i + 1 to n − 1 do
        if Z[j] < Z[min] then
            min ← j
        end if
    end for
    temp ← Z[i]
    Z[i] ← Z[min]
    Z[min] ← temp
end for
```

Insert-Sort (Spielkarten)

- für alle Elemente:
  - nimm ein Element
  - durchlaufe Ziel-Array vergleiche mit jedem Element dort
  - füge das Element an der passenden Stelle ein

alternativ mit nur einem Array:
"sortierter und unsortierter Teil"

- erstes Element des unsortierten Teils nehmen,
  - vergleichen und tauschen, bis Element links davon kleiner ist
- best: $\Theta(n)$, avg $\Theta(n^2)$, worst $\Theta(n^2)$

```
INSERT-SORT(Z)
n ← length(Z)
for i ← 1 to n − 1 do
    elem ← Z[i]
    j ← i
    while j > 0 and Z[j − 1] > elem do
        Z[j] ← Z[j − 1]
        j ← j − 1
    end while
    Z[j] ← elem
end for
```

Merge-Sort (Arrays halbieren)

Schritt 1:

- halbiere Arrays in Teilarrays sooft, bis Array-länge = 1
- führe jeweils 2 Arrays, sodass in beiden Arrays von beiden ersten Zellen der kleinere genommen wird. Der Index des arrays aus dem er kommt wird inkrementiert,
  - anschließend wieder die Einträge der Pointer vergleichen.

Quicksort (Pivot)

- nimm rechtestes Element als Pivot
- suche von links die erste Zahl, die größer ist als Pivot
- suche von rechts die erste Zahl, die kleiner ist als Pivot
- tausche beide Elemente, solange bis sich beide Pointer treffen

best: $\Theta(n \log_2(n))$, avg $\Theta(n \cdot \log_2(n))$worst: $\Theta(n^2)$

Heap-Sort (sortierter Baum)

Anordnung eines Arrays als Baum

- Elternknoten i hat Index i/2 abgerundet
- linker Knoten hat Index 2i
- rechter Knoten hat Index 2i+1

Max-Heap: Knoten ist immer größer als die Kinder

Sortierung: jeweils das oberste Element ans Ende stellen und für die Kind-Bäume die Max-Heap Eigenschaft herstellen

- Elemente einfügen: von oben nach unten durchreichen, ggf tauschen
- Best: $\Theta(n \log(n))$, avg: $\Theta(n \log(n))$, worst $\Theta(n \log(n))$

bestmögliche Laufzeitkomplexität für Sortieralgorithmen: $\Omega(n \log(n))$-hart

Brute-Force Algorithmen

➔ Uninformierte Suche
pro: kein Wissen über Lösungsweg nötig
pro: simpel
contra: teuer (Zeit und Rechenaufwand)
contra: bei "iterativen" Problemen (zB Schach) ➔ exponentielles Wachstum

Nutzen: öffentliche Blockchain machen sich zu nutzen, dass Brute-Force langsam ist
Beispiel: Passwörter

Greedy Algorithmen

- bei jeder Entscheidung: Variante mit größter Wsk gewählt
- Ergebnis stark vom Problem abhängig
- keine Rücknahme von Entscheidungen
- keine Garantie für Erfolg

Beispiel: 3 Größen an Objekten kann gekauft werden. solange den größten nehmen, bis Maximalgewicht überschritten würde. Dann auf nächst kleinere Option zurückgehen
Problem: max = 60; set = {1, 5, 40} findet optimles ergebnis.
Problem: max = 60; set = {1, 20, 50} findet nicht optimles ergebnis.

Bewertung: wenn mehrfaches einer Einheit ohne Rest größere Einheit ergibt ➔ gut. Wenn Rest bleibt ➔ Indiz für nicht optimale Lösung

Beispiel: möglichst viele Konzerte eines Festivals: immer das Konzert welches den frühesten Schluss hat (ermöglicht die meisten anschlussmöglichkeiten), entferne alle ungünstigen Überlappungen, wähle nächstes, "kürzestes" Konzert
➔Problem zB am Ende wird nicht die Konzertlänge maximiert

➔ Greedy findet tatsächlich optimale Lösung (wiederspruch in Gegenbeweis)
Falls Erfolg des Alg nachweisbar ➔ Greedy = gute Lösung

Backtracking

Zurückgehen im Entscheidungsbaum.
pro: bei Entscheidungen kann Vorwissen mit einfließen
pro: Entscheidungspfade kann frühzeitig abgebrochen werden

```
Löse_Problem(bisherige Entscheidungen){
- problem noch lösbar mit bisherigen entscheidungen?
        - J: ist das Problem gelöst?
                - J: Gibt bisherige Entscheidung als Lösung aus //(ende falls nur eine lsg gesucht wird)
                - N: Für alle möglichen Entscheidungen e
                        - Löse_Problem(bisherige Entscheidung + e) //hier ggf Heuristik
        - N: Pruning (Baum beschneiden)
        }
```

Beispiel: 8 Damen Problem: lassen sich n Damen auf $n \cdot n$ Plätzen platzieren, ohne dass sie sich gegenseitig bedrohen.
➔ Lösung Damen konsekutiv platzieren und jeweils überprüfen

Binary Search Tree

Anwesenheit eines Werts prüfen
in Suchbaum nur log(n) aufrufe

Binary Search:

- Suchbereich halbieren
- Vergleich mit Element in der Mitte
- rekursive Suche in linkem oder rechtem Bereich

binärer Suchbaum

- jeder Knoten maximal 2 Nachfolger
- kein einfaches Abbild auf Array, da frühzeitig Enden erreicht werden können
- linkes Kind $\leq$ Key des Knotens
- rechtes Kind $\geq$ Key des Knotens

Traversieren: $O(n)$

- sofern nicht Endknoten:
    - laufe links
    - print key
    - laufe rechts

Suchen: $O(h)$

- falls key=gesucht return true
- falls key > gesucht suche links
- falls key < gesucht suche rechts
- falls ende: return false

Einfügen: $O(h)$

- falls Frei: einfügen
- falls neu < key: füge rekursiv links ein
- falls nue > key: füge rekursiv rechts ein

Löschen $O(h)$

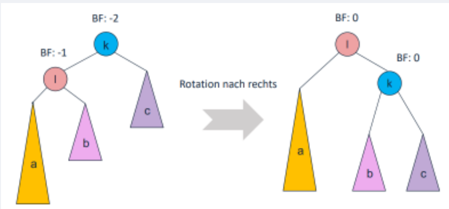
- enden einfach löschen
- ein Kind: Nachfolger nimmt Position des gelöschten ein
- zwei Kinder:
    - Stelle durch kleinstes Element des rechten Teilbaums (ein Schritt nach rechts, dann, solange links bis ende)

AVL-Bäume

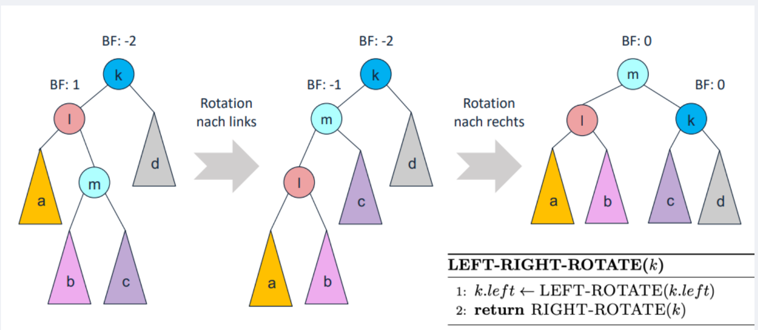
Idee: jeder Knoten darf einen maximalen Balance-Faktor (BF(k)) von $\{-1; 0; 1\}$ haben

Da AVL Bäume nicht "sehr" tief sind, können Algorithmen mit $O(\log(n))$ erstellt werden!

Bei einem Ungleichgewicht "Rotation".
Falls beide Knoten der Rotation gleiches VZ haben:



Falls die Knoten unterschiedlich sind:



Nach Einfügen rekursiv von eingefügtem Konten beginnend prüfen ob Parent-Knoten balanciert ist, ggf parent drehen.

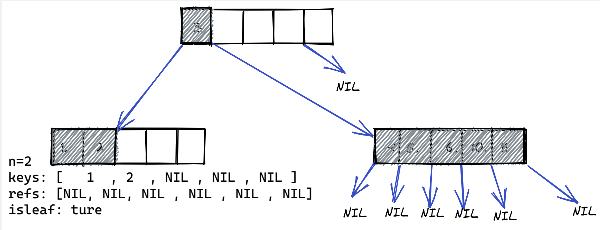
B-Bäume

Idee: mehr Nachfolger möglich
Idee: jeder Knoten entspricht z. B. 1 Seite eines externen Mediums

- Ordnung m bestimmt, wie groß der Knoten werden kann
  - maximal 2m-1 Schlüsselwerte
  - besitzt (Ausnahme der Wurzel) m-1 Schlüsselwerte
- Größe von m meist Seitengröße vom Speicher

für einen B-Baum der Ordnung m gilt:
$h \leq \log_m(\frac{n+1}{2})$

Baum wächst von unten nach oben. Falls ein Knoten zu voll wird "platzt dieser" nach oben.

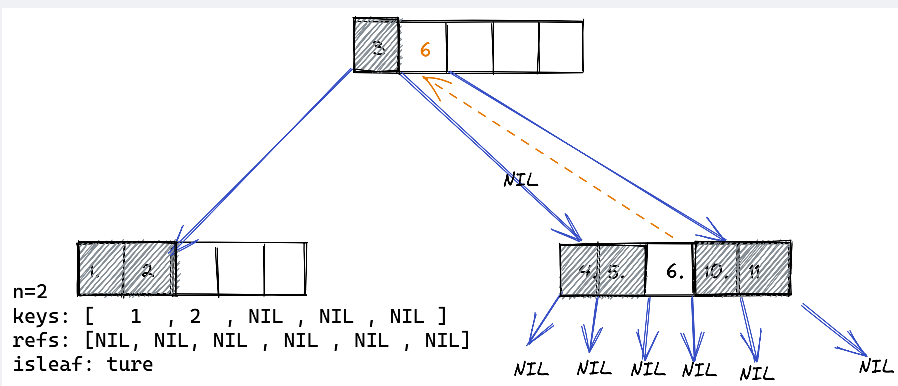


Suchen:

- laufe so lange in der Wurzel nach rechts, bis gesucht>key
- folge dem Verweis nach unten
- falls NIL ende
- sonst wiederhole alles

Einfügen:

- laufe nach unten und füge ein
- falls Blatt bereits voll: halbieren, mittleres Element 1 Schicht nach oben schieben
  - linker Verweis auf linke Hälfte,
  - rechter Verweis auf rechte Hälfte
  -



Damit ein Knoten nicht übervoll wird:

- Präventiv beim Absteigen zum Blatt jeden Knoten "platzen lassen" der bereits voll ist,
- so kann garantiert werden, dass beim Einfügen keine Kettenreaktion durch Platzen entsteht.
- Sonderfall: Root platzt: Ausnahmeregel tritt ein und ein neuer Root-Knoten entsteht mit nur einem Element[0] das Splitt-element ist

Löschen:

- im Blattknoten: kein Problem
- im nicht-Blattknoten: Vorgänger im vorherigen Teilknoten das letzte Element nach oben ziehen
- falls nicht möglich, weil minimale Größe unterschritten: 2 Knoten verschmelzen
- falls Knoten bereits Mindestgröße: verschmelzen mit "Geschwister-knoten" (auf gleicher Ebene) inklusive Elternknoten

Hashtabellen

$h : \mathbb{D} \to \mathbb{W}$
$|\mathbb{D}| >> |\mathbb{W}|$

- schnell berechenbar (idealerweise $O(1)$)
- alle $e \in \mathbb{W}$ werden erreicht
- möglichst gleich Wahrscheinlich
- Unstetigkeit: ähnliche Werte $\nrightarrow$ ähnliche Hashes

Anwendung von Hashtabelle:

- einfügen, suchen und entfernen von Daten in Liste
  Idee:
- Objekte werden an der Position ihres Hashes gespeichert
  Problem:
- Kollisionen (2 Werte aus Datenmenge haben gleichen Hashwert)

Beispiel Modulo als Hash
$h(x) = x \mod m$
m: idealerweise Primzahl

bei Strings: summe über ASCII Repräsentation

andere Möglichkeit:
$h(x) = \lfloor (x \cdot A - \lfloor x \cdot A \rfloor) \cdot m \rfloor$ (innere Klammer: betrachtet nur Nachkommastellen)
$A \in \mathbb{R}; A \notin \mathbb{Q}$ (irrationale Zahl)

Lösung für Kollisionen

- Verkettung: verkettete Liste, falls Kollision
- offene Adressierung: Hashfunktion liefert Reihe an Adressen, (z. B. wenn voll, einfach zur nächsten Position)
  - Problem: Klumpenbildung für $h(x,i) = (x+i) \mod 10$
  - etwas besser: $h(x,i) = (h'(x) + c_1 i + c_2(i^2)) \mod m$
  - weiteres Problem: beim Löschen muss sich gemerkt werden, dass die Zelle mal gelegt war, sonst können weitere Zahlen nicht gefunden werden.
  - ➔ Zellen haben 3 Zustände: belegt, frei, und wieder-frei

Effizient:

- bei Verkettung:
  - Einfügen/Löschen $O(1)$
  - Suchen $O(1 + n/m)$ (n/m = Belegungsfaktor)
- bei offener Adressierung und optimalem $h(x)$
  - $\frac{1}{1-\alpha}$ (alpha = Belegungsfaktor)

Grundlagen zu Graphen

Gerichtete Graphen:
Menge aus Knoten (V) und Kanten (E)
$G = (V, E)$ $V = \{1, 2, 3\}$ $E = \{(1, 2), (2, 3), (3, 1)\}$

Ungerichtete Graphen: wie gerichteter Graph, aber ohne Richtungen

Unterschiede Baum ↔ Graph

- Baum hat keine Zyklen
- Graphen können disjunkte Teile haben, Bäume haben nur zusammenhängende Knoten
- Zwei verbundene Knoten nennt man *adjazent*, bei ungerichteten Graphen sind beide Knoten wechselseitig *adjazent*
- *vollständig* alle mit allen verbunden
- *Dicht* = nahezu vollst
- *Licht / Sparse* = Gegenteil von Dicht

Darstellung im Rechner:

- Adjazenz liste (verkettete Liste)
  - für jeden Knoten jeden Nachbarknoten merken
  - bei Richtungen halbieren die Einträge
- Adjazenz Matrix
  - Matrix, die beschreibt, von welchem Knoten welcher Knoten erreichbar ist.
  - Es ist möglich z. B. Kosten oder Entfernungen in der Matrix abzuspeichern

Suchen in Graphen

Breitensuche

- alle erreichbaren Knoten identifizieren
- kürzester Pfad finden

Frage: Welche Knoten sind erreichbar, wie ist der kürzeste Pfad dorthin
Vorgehen:

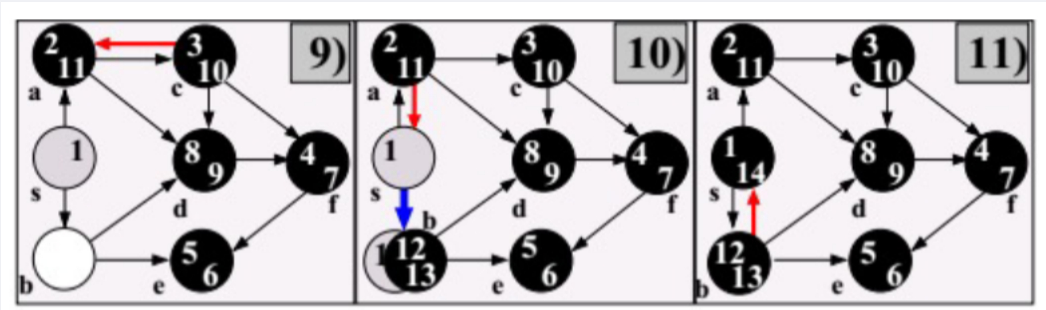
- alle Knoten weiß
- Startknoten Grau einfärben (in Behandlung)
- Alle nachfolgenden weißen Knoten in Queue einfügen und Grau färben
- wenn alle Nachfolger in Queue enthalten sind Schwarz(als fertig) markieren.
- Counter in jedem Knoten zählt, wie viele Schritte benötigt wurden.
  - Neue Knoten bekommen alten Knotenwert +1

Sofern alle Wege gleiches Gewicht haben, wird immer der kürzeste Weg gefunden.

Komplexität: $O(|E| + |V|)$

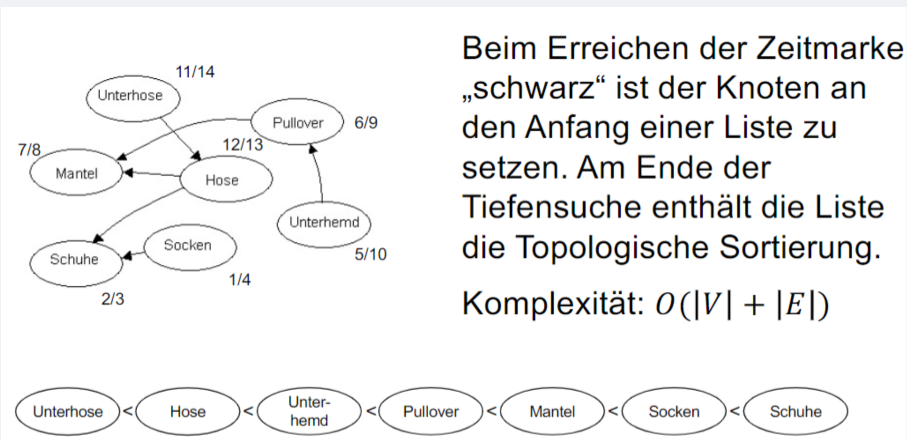
Tiefensuche

rekursiver Abstieg mit Abbruch, wenn keine unbesuchten Knoten mehr zu finden sind
dabei merken, wann (Zeit) der Knoten grau/schwarz wurde.
Zeit-Counter zählt bei jedem Traversieren und Schwarzfärben hochgezählt



Komplexität: $O(|E| + |V|)$

Topologisches Sortieren: zB in welcher Reihenfolge bestimmte Dinge passieren müssen
➔mögliche *topologische Sortierung* wäre nach "schwarz-Zeitmarken" absteigend zu sortieren



Vorteil der Tiefensuche: Reihenfolge der Children in der Rekursion kann durch Vorwissen beeinflusst werden (Heuristik möglich)

kürzeste Pfade

$\exists \quad w : E \to \mathbb{R}$ Gewichtungsfunktion die allen Kanten ein Gewicht zuordnet

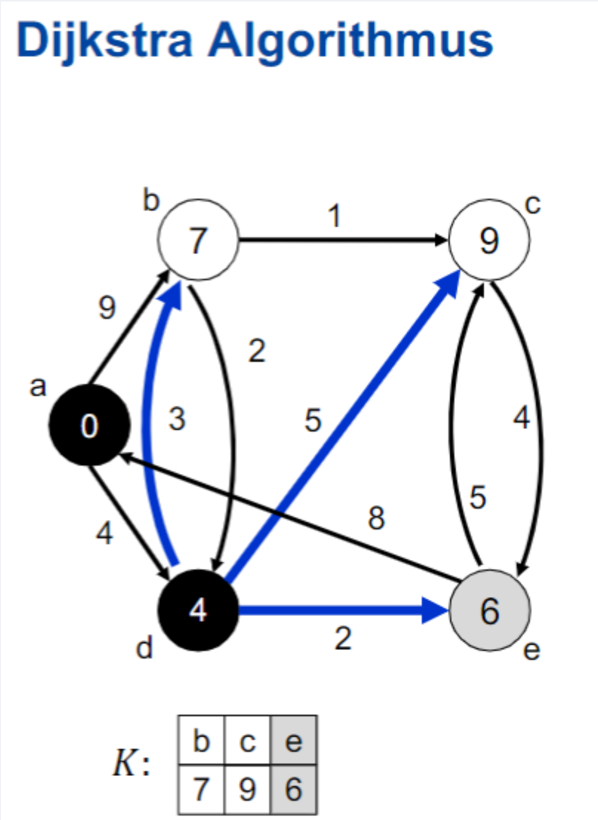
Gewicht eines Pfades = Summe der Gewichte aller passierten Kanten

falls zwei Knoten nicht verbunden sind gibts: Gewicht ∞
Negative Zyklen,

- falls ein Zyklus mit negativem aufsummiertem Kantengewicht: $-\infty$
  obere Schranke f. kürzesten Pfad: $d(v) = \begin{cases} 0 & v = s \\ \infty & v \neq s \end{cases}$
  Relaxation: wie bei Breitensuche, aber ggf Wert von Knoten anpassen, falls kürzerer Weg gefunden wurde (Wert und Vorgänger ersetzten)

Dijkstra Alg

Idee: Wie bei Breitensuche, aber anstelle FIFO immer vom Knoten mit dem kleinsten Wert ausgehen



Bellman-Ford Alg

(kommt auch mit negativen Zyklen zurecht)
Idee: kürzester Pfad darf maximal $|V| - 1$ lang sein
zB |V|-1 = 4:
4x alle Kanten durchlaufen und Relaxation nur bei nicht-Unendlichen Knoten

Falls nach $|V| - 1$ immer noch Verbesserungen möglich ➔ negativer Zyklus enthalten!

A$\star$ Alg

findet kürzesten Weg von s nach z

- nur positive Kantengewichte
- Heuristik (monoton) benötigt, die schätzt, wie weit z von s entfernt ist
- Schätzwert darf wahre Knotenkosten nicht übersteigen (zB Luftlinienkosten)
  Idee: wie Dijkstra, aber zu jedem Knoten werden die Kosten der Heuristik dazugezählt.