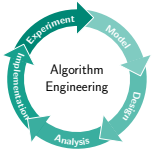
Karlsruher Institut für Technologie



Algorithmen I – Wiederholung

Sommersemester 2025

Peter Sanders | Stand: 31. Juli 2025

Organisatorisches

Klausur

- Montag, **25.08.2025**, 8:00 Uhr
- **Anmeldung** im Campus-System bis 13.08.2025
- Abmeldung bis zur Klausur möglich
- Bearbeitungszeit: 120 Minuten
- doppelseitiges, handbeschriebenes A4 Blatt darf mitgenommen werden

Organisatorisches

Klausur

- Montag, **25.08.2025**, 8:00 Uhr
- **Anmeldung** im Campus-System bis 13.08.2025
- Abmeldung bis zur Klausur möglich
- Bearbeitungszeit: 120 Minuten
- doppelseitiges, handbeschriebenes A4 Blatt darf mitgenommen werden

Wiederholungstutorien

11:30 – 13:00 Uhr, R-102 im Infobau

- Mo, 04.08., Mi, 06.08.
- Mo, 11.08., Mi, 13.08.
- Mo, 18.08., Mi, 20.08.

Organisatorisches

Klausur

- Montag, **25.08.2025**, 8:00 Uhr
- **Anmeldung** im Campus-System bis 13.08.2025
- Abmeldung bis zur Klausur möglich
- Bearbeitungszeit: 120 Minuten
- doppelseitiges, handbeschriebenes A4 Blatt darf mitgenommen werden

Wiederholungstutorien

11:30 – 13:00 Uhr, R-102 im Infobau

- Mo, 04.08., Mi, 06.08.
- Mo, 11.08., Mi, 13.08.
- Mo, 18.08., Mi, 20.08.

Bei Fragen: ILIAS-Forum, Mail an
algo1@mail.informatik.kit.edu

Klausurvorbereitung

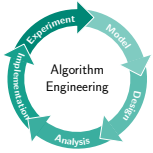
Rechnen Sie Altklausuren!!!

gibt es bei der Fachschaft

- Verständnis $\gg$ Wissen
- Typische Aufgaben
  - **Beweisen Sie.** Eigenschaften optimaler Lösungen & Datenstrukturinvarianten kennen
  - **Wenden Sie Algorithmus x auf ein gegebenes Problem an.** Algorithmen und Datenstruktur-Operationen an Beispielen durchspielen
  - **Entwurfsaufgaben** Welche Operationen brauche ich? Welche Datenstruktur unterstützt diese effizient? Welchen Algorithmus kann ich abwandeln? Leetcode, Hackerrank, ...
  - **Nennen Sie Vor- und Nachteile von x gegenüber y** Entwurfsraum kennen
- Alle Kapitel und Inhalte sind relevant!

Index

Karlsruher Institut für Technologie

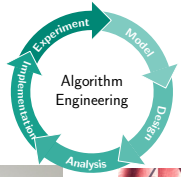


Algorithmen I – 0. Einführung

Sommersemester 2025

Peter Sanders | Stand: 30. Juli 2025

Das Team



Prof. Dr. Peter Sanders

Hans-Peter Lehmann

Tim Niklas Uhl

Matthias Schimek

Moritz Laupichler

algo1@mail.informatik.kit.edu

Organisatorisches

Alle Infos, Materialien, Vorlesungsaufzeichnung etc. im **ILIAS-Kurs**

Vorlesungen + Übung

Mo: 15:45–17:15
Mi: 14:00–14:45

Übung i.d.R. wöchentlich
(meist mittwochs, 2. Hälfte der VL)
1. Übung: 30.04.

Übungsblätter

erscheinen wöchentlich (Mittwoch)
Abgabe Freitags in der Folgewoche
Klausurbonus [+ 1 Notenschritt] (ab $\geq$ 50% der Übungspunkte)
mehr Details in der 1. Übung
1. Blatt: 30.04. – 09.05.

Tutorium

Wöchentlich
Einteilung bis Do., 24.04.25, 18 Uhr
`informatik.kit.edu/tutorieneinteilung/`

Sprechstunde

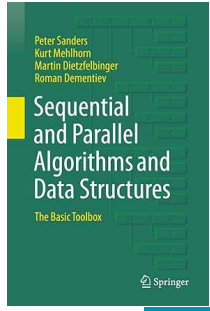
Dienstag 13:45–14:45 Uhr
(jederzeit bei offener Tür oder nach Vereinbarung)
Peter Sanders, Raum 217

Klausur

25.08.2025, 8:00 bis 10:00 Uhr
Nächste Versuchsmöglichkeit:
Nach dem WS 2025/2026

Materialien

- Folien, Übungsblätter
- Diskussionsforum:
  siehe ILIAS
- Buch:
  Sanders/Mehlhorn/Dietzf./Dem.
  Sequential and Parallel Algorithms and Data Structures — The Basic Toolbox
  oder Algorithmen und Datenstrukturen: Die Grundwerkzeuge.
  Inhalte∩Algo I sehr ähnlich.
- Taschenbuch der Algorithmen
  Springer 2008 (Unterhaltung / Motivation)



Weitere Bücher

- Algorithmen - Eine Einführung
  von Thomas H. Cormen, Charles E. Leiserson,
  Ronald L. Rivest, und Clifford Stein von Oldenbourg
- Algorithmen und Datenstrukturen
  von Thomas Ottmann und Peter Widmayer von Spektrum Akademischer Verlag
- Algorithmen kurz gefasst
  von Uwe Schöning von Spektrum Akad. Vlg., Hdg.

Algorithmus? Kann man das essen?

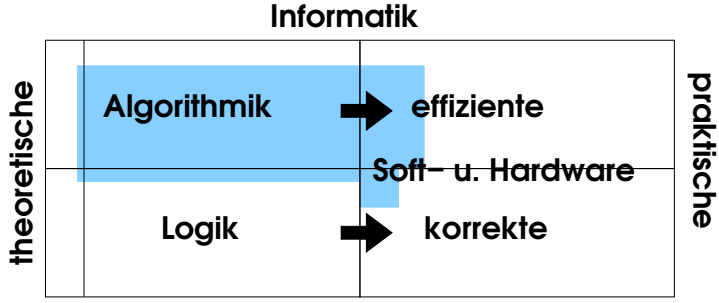
- Pseudogriechische Verballhornung eines Namens, der sich aus einer Landschaftsbezeichnung ableitet
- Al-Khwarizmi war persischer/usbekischer Wissenschaftler (aus Khorasan) aber lebte in Bagdad $\approx$ 780..840.
  Machtzentrum des arabischen Kalifats auf seinem Höhepunkt.
  Er hat ein Rechenlehrbuch geschrieben.
- $\rightsquigarrow$Algorithmus wurde zum Synonym für Rechenvorschrift.

Moderne Definition (Wikipedia)

Unter einem Algorithmus versteht man eine genau definierte Handlungsvorschrift zur Lösung eines Problems oder einer bestimmten Art von Problemen in endlich vielen Schritten

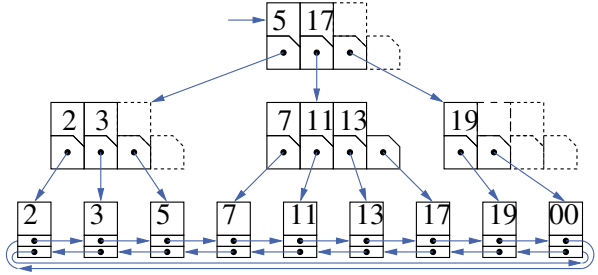
Algorithmik

Kerngebiet der (theoretischen) Informatik mit direktem Anwendungsbezug



Datenstruktur

- Ein Algorithmus bearbeitet Daten.
- Wenn ein Teil dieser Daten eine (interessante) Struktur haben, nennen wir das Datenstruktur.
- Immer wiederkehrende Datenstrukturen und dazugehörige Algorithmenteile
- $\rightsquigarrow$ wichtige Grundwerkzeuge (Basic Toolbox)



Themenauswahl: Werkzeugkasten

Immer wieder benötigte

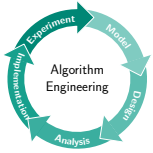
- Datenstrukturen
- Algorithmen
- Entwurfstechniken $\rightsquigarrow$ neue Algorithmen
- Analysetechniken $\rightsquigarrow$ Leistungsgarantien, objektiver Algorithmenvergleich

Jeder Informatiker braucht das $\rightsquigarrow$ Pflichtvorlesung

Inhaltsübersicht

Index

Karlsruher Institut für Technologie



Algorithmen I – 1. Amuse Geule

Sommersemester 2025

Peter Sanders | Stand: 30. Juli 2025

Technik-Check: Interaktive Fragen mit Klicker

Im Laufe der Vorlesung verwenden wir Klicker für interaktive Fragerunden.

Das sieht so aus...

pwa.klicker.uzh.ch/join/algoeng

Technik-Check: Interaktive Fragen mit Klicker

Im Laufe der Vorlesung verwenden wir Klicker für interaktive Fragerunden.

Das sieht so aus...

Frage

In welchem Studiengang studieren Sie?

Frage

Haben Sie schon mal etwas über Algorithmen gelernt? Falls ja, wo?

1. Amuse Geule

Beispiel: Langzahl-Multiplikation

Einordnung: Wir starten auf Grundschulniveau und kommen bis zu Fragestellungen die für Algorithmen I recht fortgeschritten sind.

Schreibe Zahlen als Ziffernfolgen $a = (a_{n-1} \dots a_0)$, $a_i \in 0..B-1$.

Wir zählen

Volladditionen: $(c', s) := a_i + b_j + c$

Beispiel ($B = 10$): $9 + 9 + 1 = (1, 8)$

Ziffernmultiplikationen: $(p', p) := a_i \cdot b_j$

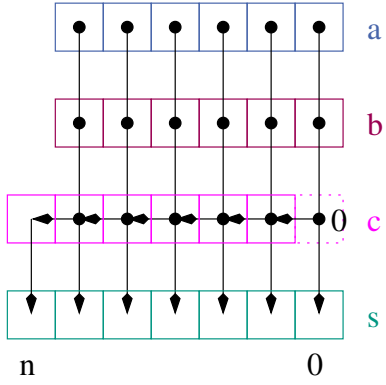
Beispiel ($B = 10$): $9 \cdot 9 = (8, 1)$

Addition

c=0 : Digit // carry / Überlauf

for i := 0 **to** $n-1$ **do** $(c, s_i) := a_i + b_i + c$

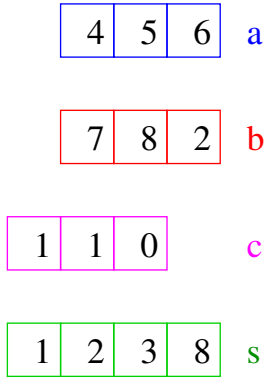
$s_n := c$



Addition

c=0 : Digit // carry / Überlauf
for i := 0 **to** $n-1$ **do** $(c, s_i) := a_i + b_i + c$
$s_n := c$

Beispiel:



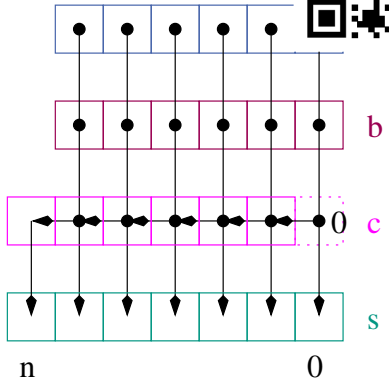
pwa.klicker.uzh.ch/join/algoeng

Addition

c=0 : Digit // carry / Überlauf

for $i := 0$ **to** $n-1$ **do** $(c, s_i) := a_i + b_i + c$

$s_n := c$



Frage

Wie viele Volladditionen (Ziffern-Additionen) benötigt die Addition zweier Zahlen mit je n Ziffern?

Exkurs: Pseudocode

- Kein C/C++/Java — Menschenlesbarkeit vor Maschinenlesbarkeit
- Eher Pascal + Mathe − begin/end — Einrückung trägt Bedeutung

Zuweisung: :=

Kommentar: //

Ausdrücke: volle Mathepower — $\{i \geq 2 : \neg\exists a, b \geq 2 : i = ab\}$

Deklarationen: c=0 : Digit

Tupel: $(c, s_i) := a_i + b_i + c$

Schleifen: **for** , **while** , **repeat** ... **until** ,...

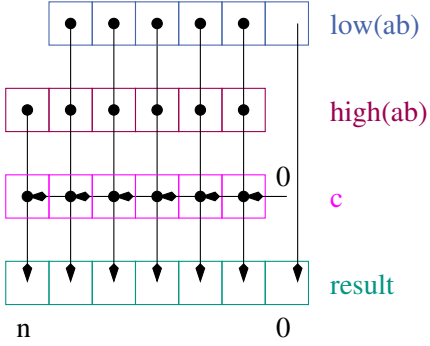
uvam: Buch Abschnitt 2.3, hier: just in time und on demand
if , Datentypen, Klassen, Speicherverwaltung

Exkurs vom Exkurs: Wieso nicht C++/Java-like ?

- Klare Unterscheidung von Programmcode
- viele redundante `()[]{};`
- C `for` ist sehr low level
- == ist unschön während := für Zuweisung klarer ist
- C Logik/Bitoperatoren sind kryptischer als $\wedge$ etc.
- Wir verwenden C++/Java wo dies sinnvoll ist
  `// ++ - += -=`
- Mathenotation ist oft mächtiger

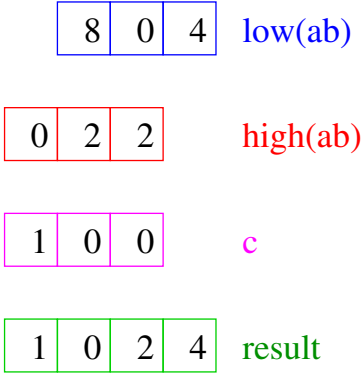
Ziffernmultiplikation

Function numberTimesDigit(a : **Array** $[0..n-1]$**of** Digit, b : Digit)



Beispiel

numberTimesDigit(256, 4)



Ziffernmultiplikation

Function numberTimesDigit(a : **Array** $[0..n-1]$**of** Digit, b : Digit)
    result : **Array** $[0..n]$**of** Digit
    c=0 : Digit // carry / Überlauf
    (h', ℓ):= $a[0] \cdot b$ // Ziffernmultiplikation
    result[0]:= ℓ
    for i := 1 **to** $n-1$ **do** // $n-1$ Iterationen
        (h, ℓ):= $a[i] \cdot b$ // Ziffernmultiplikation
        $(c, \text{result}[i])$:= $c + h' + \ell$ // Ziffernaddition
        h':= h
    result[n]:= $c + h'$ // Ziffernaddition, kein Überlauf?!
    return result

Analyse: $1 + (n-1) = n$ Multiplikationen, $(n-1) + 1 = n$ Additionen

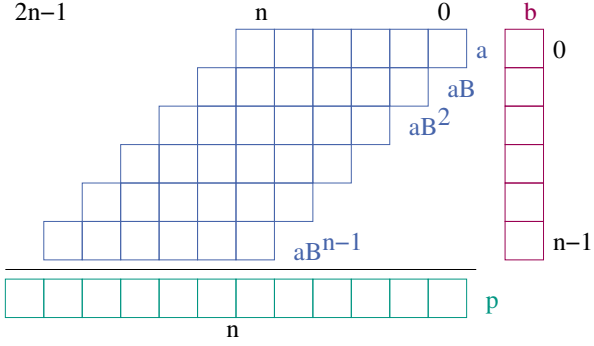
Schulmultiplikation

p=0 : $\mathbb{N}$ // Langzahl

for j := 0 **to** $n-1$ **do**

// Langzahladdition, Langzahl mal Ziffer, Schieben:

p:= $p + a \cdot b[j] \cdot B^j$



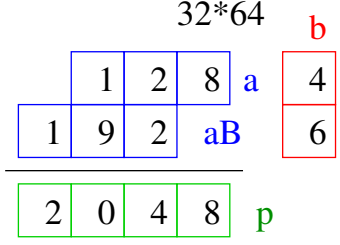
Schulmultiplikation

p=0 : $\mathbb{N}$ // Langzahl

for j := 0 **to** $n-1$ **do**

// Langzahladdition, Langzahl mal Ziffer, Schieben:

p:= $p + a \cdot b[j] \cdot B^j$



Schulmultiplikation Analyse

$p{=}0 \ : \mathbb{N}$
for $j := 0$ **to** $n-1$ **do**
  $p{:=}\ p$ // $n+j$ Ziffern (außer bei $j=0$)
  $+$ // $n+1$ Ziffernadditionen (optimiert)
  $a \cdot b[j]$ // je n Additionen/Multiplikationen
  $\cdot B^j$ // schieben (keine Ziffernarithmetik)

Insgesamt:
n^2 Multiplikationen
$n^2 + (n-1)(n+1) = 2n^2 - 1$ Additionen
$3n^2 - 1 \leq 3n^2$ Ziffernoperationen

Exkurs O-Kalkül, die Erste

$$\mathrm{O}(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

Idee: Konstante Faktoren (und Anfangsstück) ausblenden

- $+$ Operationen zählen $\rightsquigarrow$ Laufzeit
- $+$ Rechnungen vereinfachen
- $+$ Interpretation vereinfachen
- $?$ Werfen wir zuviel Information weg ?

Beispiel: Schulmultiplikation braucht Zeit $\mathrm{O}(n^2)$

pwa.klicker.uzh.ch/join/algoeng

Exkurs O-Kalkül, die Erste

$$\mathrm{O}(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

Idee: Konstante Faktoren (und Anfangsstück) ausblenden

- $+$ Operationen zählen $\rightsquigarrow$ Laufzeit
- $+$ Rechnungen vereinfachen
- $+$ Interpretation vereinfachen
- $?$ Werfen wir zuviel Information weg

Beispiel: Schulmultiplikation braucht Zeit $\mathrm{O}(n^2)$

Frage

Welche Operationen zählen?

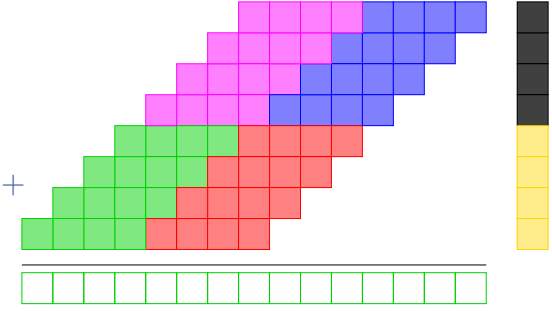
?

Ergebnisüberprüfung

Später an Beispielen

Ein rekursiver Algorithmus

Function recMult(a, b)
  assert a und b haben $n = 2k$ Ziffern, n ist Zweierpotenz
  if $n = 1$ **then return** $a \cdot b$
  Schreibe a als $a_1 \cdot B^k + a_0$ // zerlegen
  Schreibe b als $b_1 \cdot B^k + b_0$ // zerlegen
  return
    $\text{recMult}(a_1, b_1) \cdot B^{2k} +$
    $(\text{recMult}(a_0, b_1) + \text{recMult}(a_1, b_0)) \cdot B^k +$
    $\text{recMult}(a_0, b_0)$



Ein rekursiver Algorithmus
Beispiel

$$\boxed{10}\boxed{01}\cdot\boxed{19}\boxed{84}=$$
$$10\cdot 19\cdot 10000+$$
$$(10\cdot 84+1\cdot 19)\cdot 100+$$
$$1\cdot 84=$$
$$1985984$$

Ein rekursiver Algorithmus
Analyse

Function recMult(a, b) // $T(n)$ Ops

  assert a und b haben $n = 2k$ Ziffern, n ist Zweierpotenz

  if $n = 1$ **then return** $a \cdot b$ // 1 Op

  Schreibe a als $a_1 \cdot B^k + a_0$ // 0 Ops

  Schreibe b als $b_1 \cdot B^k + b_0$ // 0 Ops

  return

    $\text{recMult}(a_1, b_1) \cdot B^{2k} +$ // $T(n/2) + 2n$ Ops

    $(\text{recMult}(a_0, b_1) + \text{recMult}(a_1, b_0))B^k +$ // $2T(n/2) + 2n$ Ops

    $\text{recMult}(a_0, b_0)$ // $T(n/2) + 2n$ Ops

Also $T(n) \leq 4T(n/2) + 6n$

Übung: Wo kann man hier $\approx 2n$ Ops sparen?

Ein rekursiver Algorithmus
Analyse

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ 4 \cdot T(\lceil n/2 \rceil) + 6 \cdot n & \text{if } n \geq 2. \end{cases}$$

$\longrightarrow$ (Master-Theorem, stay tuned)

$T(n) = \Theta\left(n^{\log_2 4}\right) = \mathrm{O}(n^2)$

Satz
$\rightsquigarrow T(n) \leq 7n^2 - 6n$, falls n eine Zweierpotenz ist

Beweis durch Vollständige Induktion

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ 4 \cdot T(\lceil n/2 \rceil) + 6 \cdot n & \text{if } n \geq 2. \end{cases}$$

Behauptung und IV: $T(n) \leq 7n^2 - 6n$.
IA: $T(1) = 1 = 7 \cdot 1^2 - 6$✓
IS:

$$\begin{aligned} T(2n) &\leq 4T(n) + 6 \cdot 2n && \text{nutze IV} \\ &\leq 4(7n^2 - 6n) + 12n \\ &= 7(2n)^2 - 24n + 12n = 7(2n)^2 - 6(2n)\checkmark \end{aligned}$$

□

Exkurs: Algorithmen-Entwurfsmuster

Im Buch: siehe auch Index!

Schleife: z. B. Addition

Unterprogramm: z. B. Ziffernmultiplikation, Addition

Teile und Herrsche: (lat. divide et impera, engl. divide and conquer)
Aufteilen in eins oder mehrere, kleinere Teilprobleme,
oft rekursiv

Es kommen noch mehr: greedy, dynamische Programmierung, Metaheuristiken, Randomisierung,. . .

Karatsuba-Ofman Multiplikation [1962]

Beobachtung: $(a_1 + a_0)(b_1 + b_0) = a_1 b_1 + a_0 b_0 + a_1 b_0 + a_0 b_1$

Function recMult(a, b)
  assert a und b haben $n = 2k$ Ziffern, n ist Zweierpotenz
  if $n = 1$ **then return** $a \cdot b$
  Schreibe a als $a_1 \cdot B^k + a_0$
  Schreibe b als $b_1 \cdot B^k + b_0$
  c_{11}:= recMult(a_1, b_1)
  c_{00}:= recMult(a_0, b_0)
  return
    $c_{11} \cdot B^{2k} +$
    $(\text{recMult}((a_1 + a_0), (b_1 + b_0)) - c_{11} - c_{00})B^k$
    $+ c_{00}$

Karatsuba-Ofman Multiplikation [1962]

Beispiel

$$\boxed{10}\boxed{01}\cdot\boxed{19}\boxed{84}=$$
$$10\cdot 19\cdot 10000+$$
$$((10+1)\cdot(19+84)-10\cdot 19-1\cdot 84)\cdot 100+$$
$$1\cdot 84=$$
$$1985984$$

Karatsuba-Ofman Multiplikation [1962]

Analyse

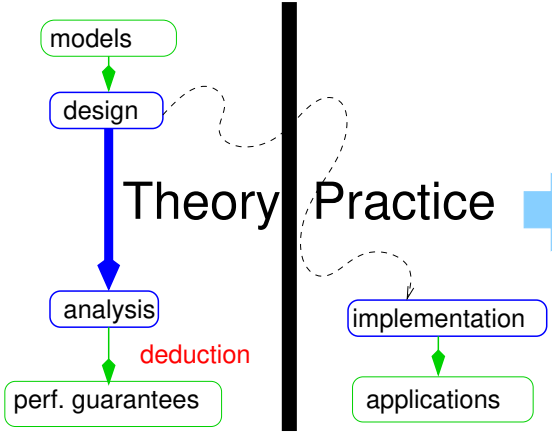
$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ 3 \cdot T(\lceil n/2 \rceil) + 10 \cdot n & \text{if } n \geq 2. \end{cases}$$

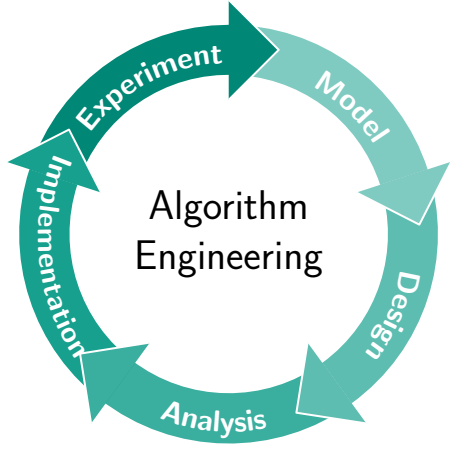
$\longrightarrow$ (Master-Theorem)

$$T(n) = \Theta\left(n^{\log_2 3}\right) \approx \Theta\left(n^{1.58}\right)$$

Algorithm Engineering

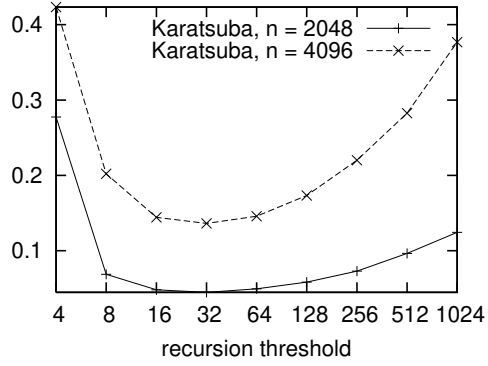
Was hat das mit der Praxis zu tun?





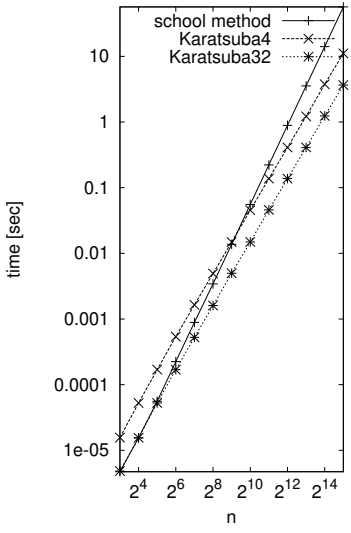
Zurück zur Langzahlmultiplikation

- Zifferngröße $\leftrightarrow$ Hardware-Fähigkeiten
  z. B. 32 Bit
- Schulmultiplikation für kleine Eingaben
- Assembler, SIMD,...



Skalierung

- Asymptotik setzt sich durch
- Konstante Faktoren oft Implementierungsdetail

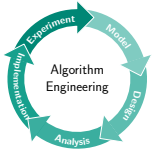


Blick über den Tellerrand

- Bessere Potenzen durch Aufspalten in mehr Teile
- Schnelle Fourier Transformation
  $\rightsquigarrow \mathrm{O}(n)$ Multiplikationen von $\mathrm{O}(\log n)$-Bit Zahlen
- [Harvey / van der Hoeven 2021]: $\mathrm{O}(n \log n)$ Bitkomplexität / Mehrband-Turingmaschine

Index

Karlsruher Institut für Technologie



Algorithmen I – 2. Einführendes

Sommersemester 2025

Peter Sanders | Stand: 30. Juli 2025

Überblick

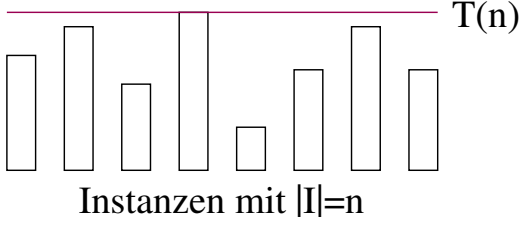
- Algorithmenanalyse
- Maschinenmodell
- Pseudocode
- Codeannotationen
- Mehr Algorithmenanalyse
- Graphen

(Asymptotische) Algorithmenanalyse

Gegeben: Ein Programm

Gesucht: Laufzeit $T(I)$ (# Takte), eigentlich für alle Eingaben I (!)
(oder auch Speicherverbrauch, Energieverbrauch,...)

Erste Vereinfachung: Worst case: $T(n) = \max_{|I|=n} T(I)$
(Später mehr: average case, best case, die Rolle des Zufalls, mehr Parameter)

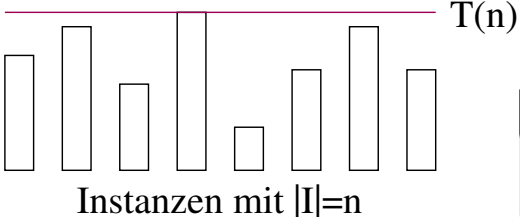


(Asymptotische) Algorithmenanalyse

Gegeben: Ein Programm

Gesucht: Laufzeit $T(I)$ (# Takte), eigentlich für alle Eingaben I (!)
(oder auch Speicherverbrauch, Energieverbrauch,...)

Erste Vereinfachung: Worst case: $T(n) = \max_{|I|=n} T(I)$
(Später mehr: average case, best case, die Rolle des Zufalls, mehr Parameter)



Frage

Wie sieht die Formel für average case aus?

Zweite Vereinfachung: Asymptotik

$$\mathrm{O}(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

„höchstens“

$$\Omega(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

„mindestens“

$$\Theta(f(n)) = \mathrm{O}(f(n)) \cap \Omega(f(n))$$

„genau“

$$o(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

„weniger“

$$\omega(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

„mehr“

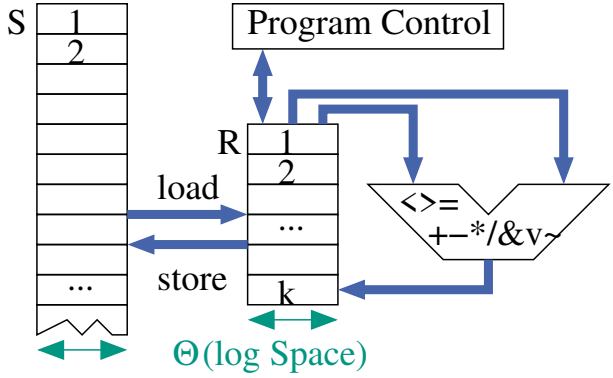
O-Kalkül Rechenregeln

Schludrigkeit: implizite Mengenklammern.
Lese '$f(n) = E$' als '$\{f(n)\} \subseteq E$'

$$cf(n) = \Theta(f(n)) \text{ für jede positive Konstante } c$$
$$\sum_{i=0}^{k} a_i n^i = O(n^k)$$
$$f(n) + g(n) = \Omega\left(f(n)\right),$$
$$f(n) + g(n) = \mathrm{O}(f(n)) \text{ falls } g(n) = \mathrm{O}(f(n)),$$
$$\mathrm{O}(f(n)) \cdot \mathrm{O}(g(n)) = \mathrm{O}(f(n) \cdot g(n)).$$

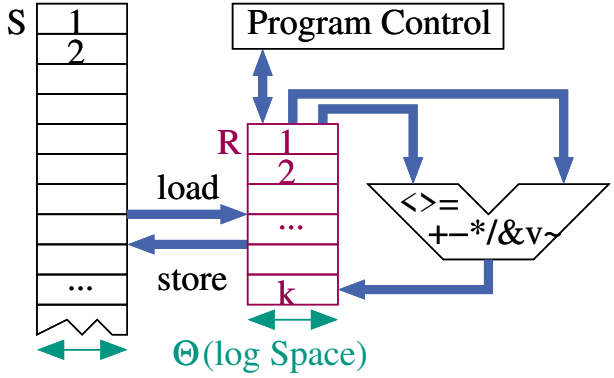
u. s. w.

Maschinenmodell
RAM (Random Access Machine)



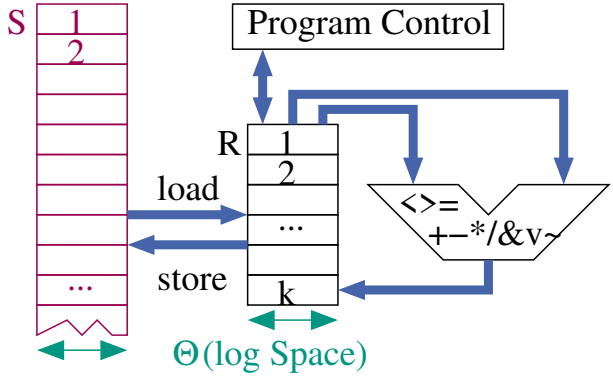
Moderne (RISC) Adaption des von Neumann-Modells [von Neumann 1945]

Maschinenmodell
Register



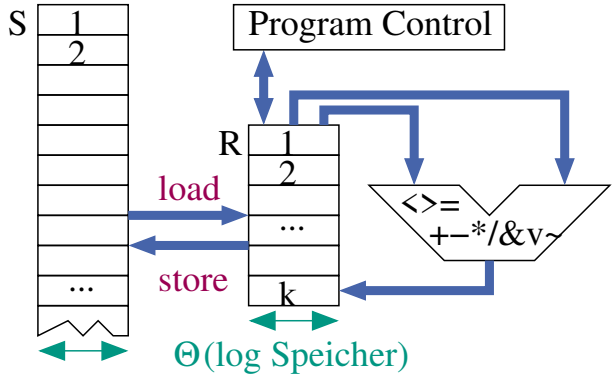
k (irgendeine Konstante) Speicher $R_1,\ldots,R_k$ für (kleine) ganze Zahlen

Maschinenmodell
Hauptspeicher



Unbegrenzter Vorrat an Speicherzellen $S[1]$, $S[2]\ldots$ für (kleine) ganze Zahlen

Maschinenmodell
Speicherzugriff



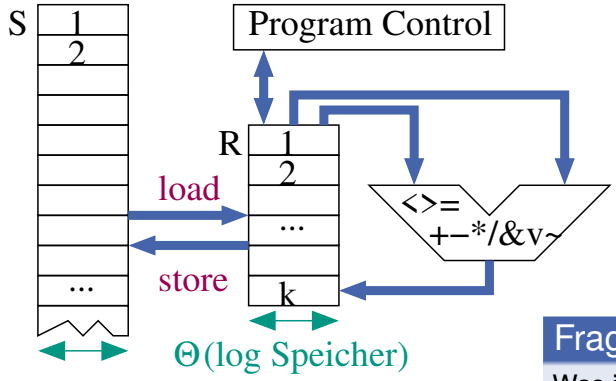
$R_i := S[R_j]$ lädt Inhalt von Speicherzelle $S[R_j]$ in Register R_i.

$S[R_j] := R_i$ speichert Register R_i in Speicherzelle $S[R_j]$.

Maschinenmodell

Speicherzugriff

pwa.klicker.uzh.ch/join/algoeng



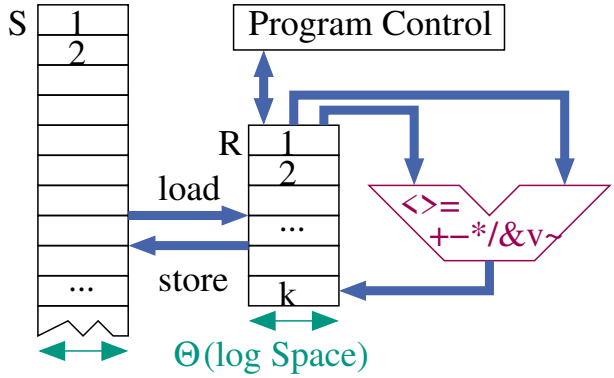
$R_i := S[R_j]$ lädt Inhalt von Speicherzelle $S[R_j]$ in Register R_i.

$S[R_j] := R_i$ speichert Register R_i in Speicherzelle $S[R_j]$.

Frage

Was ist die worst-case Laufzeit eines Speicherzugriffs auf einer physikalisch realistischen Maschine?

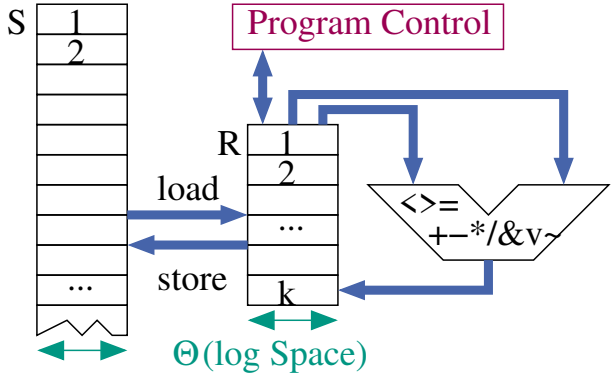
Maschinenmodell
Rechnen



$R_i := R_j \odot R_\ell$ Registerarithmetik.

'$\odot$' ist Platzhalter für eine Vielzahl von Operationen: Arithmetik, Vergleich, Logik

Maschinenmodell
Bedingte Sprünge



`JZ` j, R_i Setze Programmausführung an Stelle j fort falls $R_i = 0$

„Kleine“ ganze Zahlen?

Alternativen:

Konstant viele Bits (64?): theoretisch unbefriedigend, weil nur endlich viel Speicher adressierbar $\rightsquigarrow$ endlicher Automat

Beliebige Genauigkeit: viel zu optimistisch für vernünftige Komplexitätstheorie. Beispiel: n-maliges Quadrieren führt zu einer Zahl mit $\approx 2^n$ Bits.
OK für Berechenbarkeit

Genug um alle benutzten Speicherstellen zu adressieren: bester Kompromiss.

Algorithmenanalyse im RAM-Modell

Zeit: Ausgeführte Befehle zählen, d. h. Annahme 1 Takt pro Befehl.
Nur durch späteres $O(\cdot)$ gerechtfertigt!
Ignoriert Cache, Pipeline, Parallelismus...

Platz: Etwas unklar:

- letzte belegte Speicherzelle?
- Anzahl benutzter Speicherzellen?
- Abhängigkeit von Speicherverwaltungsalgorithmen?

Hier: Es kommt eigentlich nie drauf an.

Mehr Maschinenmodell

Cache

Schneller Zwischenspeicher

- begrenzte Größe $\rightsquigarrow$ kürzlich/häufig zugegriffene Daten sind eher im Cache
- blockweiser Zugriff $\rightsquigarrow$ Zugriff auf konsekutive Speicherbereiche sind schnell

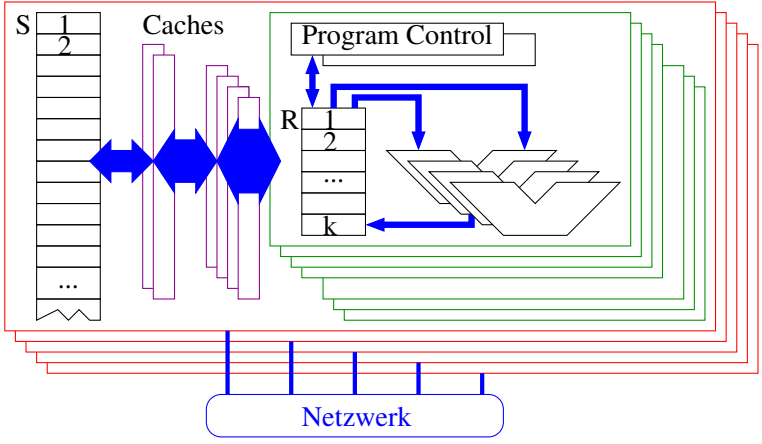
Parallelverarbeitung

Mehrere Prozessoren $\rightsquigarrow$ unabhängige Aufgaben identifizieren

...

Mehr in TI, Algorithmen II, Programmierparadigmen,...

Mehr Maschinenmodell



Pseudocode

Just in time. Beispiel:

Class Complex(x, y : Element) **of** Number
    Number r:= x
    Number i:= y
    Function abs : Number **return** $\sqrt{r^2 + i^2}$
    Function add(c' : Complex) : Complex
        return Complex($r + c'.r, i + c'.i$)

Design by Contract / Schleifeninvarianten

Karlsruher Institut für Technologie

assert: Aussage über Zustand der Programmausführung

Vorbedingung: Bedingung für korrektes Funktionieren einer Prozedur

Nachbedingung: Leistungsgarantie einer Prozedur, falls Vorbedingung erfüllt

Invariante: Aussage, die an „vielen“ Stellen im Programm gilt

Schleifeninvariante: gilt vor / nach jeder Ausführung des Schleifenkörpers

Datenstrukturinvariante: gilt vor / nach jedem Aufruf einer Operation auf abstraktem Datentyp

Hier: Invarianten als zentrales Werkzeug für Algorithmenentwurf und Korrektheitsbeweis.

Beispiel
(Ein anderes als im Buch)

Function power($a : \mathbb{R}; n_0 : \mathbb{N}) : \mathbb{R}$
- **assert** $n_0 \geq 0$ and $\neg(a = 0 \wedge n_0 = 0)$ // Vorbedingung
- $p{=}a : \mathbb{R}; \quad r{=}1 : \mathbb{R}; \quad n{=}n_0 : \mathbb{N}$ // $p^n r = a^{n_0}$
- **while** $n > 0$ **do**
  - **invariant** $p^n r = a^{n_0}$ // Schleifeninvariante **(*)**
  - **if** n is odd **then** $n{-}{-}$; $r{:=}\, r \cdot p$
  - **else** $(n, p){:=}\, (n/2, p \cdot p)$
- **assert** $r = a^{n_0}$ // **(*)**$\wedge n = 0 \longrightarrow$Nachbedingung
- **return** r

Rechenbeispiel: 2^5

$p{=}a = 2 : \mathbb{R}; \quad r{=}1 : \mathbb{R}; \quad n{=}n_0 = 5 : \mathbb{N}$ // $2^5 \cdot 1 = 2^5$

while $n > 0$ **do**

  invariant $p^n r = a^{n_0}$ // Schleifeninvariante

  if n is odd **then** $n{-}{-}$; $r{:=}\, r \cdot p$

  else $(n, p){:=}\, (n/2, p \cdot p)$

Iteration	p	r	n	$p^n r$
0	2	1	5	32
1	2	2	4	32
2	4	2	2	32
3	16	2	1	32
4	32	32	0	**32**

Beispiel

Function power($a : \mathbb{R}; n_0 : \mathbb{N}) : \mathbb{R}$
  assert $n_0 \geq 0$ and $\neg(a = 0 \wedge n_0 = 0)$ // Vorbedingung
  $p{=}a : \mathbb{R}; \quad r{=}1 : \mathbb{R}; \quad n{=}n_0 : \mathbb{N}$ // $p^n r = a^{n_0}$
  while $n > 0$ **do**
    invariant $p^n r = a^{n_0}$ // Schleifeninvariante **(*)**
    if n is odd **then** $n{-}{-}$; $r{:=}\, r \cdot p$
    else $(n, p){:=}\, (n/2, p \cdot p)$
  assert $r = a^{n_0}$ // **(*)**$\wedge n = 0 \longrightarrow$Nachbedingung
  return r

Fall n ungerade: Invariante erhalten wegen $p^n r = p^{\overbrace{n-1}^{\text{neues } n}} \underbrace{pr}_{\text{neues } r}$

Beispiel

Function power($a : \mathbb{R}; n_0 : \mathbb{N}) : \mathbb{R}$
- **assert** $n_0 \geq 0$ and $\neg(a = 0 \wedge n_0 = 0)$ // Vorbedingung
- $p{=}a : \mathbb{R}; \quad r{=}1 : \mathbb{R}; \quad n{=}n_0 : \mathbb{N}$ // $p^n r = a^{n_0}$
- **while** $n > 0$ **do**
  - **invariant** $p^n r = a^{n_0}$ // Schleifeninvariante **(*)**
  - **if** n is odd **then** $n{-}{-}$; $r{:=}\, r \cdot p$
  - **else** $(n, p){:=}\, (n/2, p \cdot p)$
- **assert** $r = a^{n_0}$ // **(*)**$\wedge n = 0 \longrightarrow$ Nachbedingung
- **return** r

Fall n gerade: Invariante erhalten wegen $p^n = \underbrace{(p \cdot p)}_{\text{neues } p}^{\overbrace{n/2}^{\text{neues } n}}$

Programmanalyse

Die fundamentalistische Sicht: Ausgeführte RAM-Befehle zählen

Pseudo-Code $\xrightarrow{\text{einfache Übersetzungsregeln}}$ Maschinenbefehle

Idee: $\mathrm{O}(\cdot)$-Notation vereinfacht die direkte Analyse des Pseudocodes.

- $T(I; I') = T(I) + T(I')$.
- $T(\textbf{if } C \textbf{ then } I \textbf{ else } I') = \mathrm{O}(T(C) + \max(T(I), T(I')))$.
- $T(\textbf{repeat } I \textbf{ until } C) = \mathrm{O}(\sum_i T(i\text{-te Iteration}))$

Rekursion $\rightsquigarrow$ Rekurrenzrelationen

Schleifenanalyse $\rightsquigarrow$ Summen ausrechnen

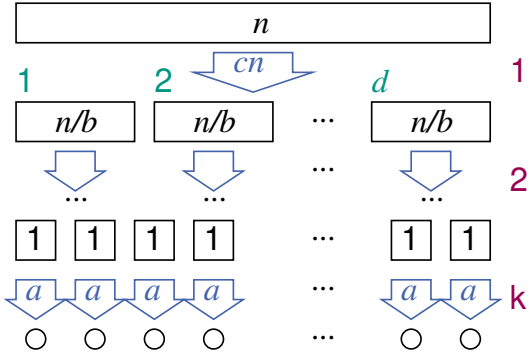
Das lernen Sie in Mathe
Beispiel: Schulmultiplikation

Master Theorem

Eine Rekurrenz für Teile und Herrsche / Rekursion

Für positive Konstanten a, b, c, d, sei $n = b^k$ für ein $k \in \mathbb{N}$.

$$r(n) = \begin{cases} a & \text{falls } n = 1 \text{ Basisfall} \\ cn + dr(n/b) & \text{falls } n > 1 \text{ teile und herrsche.} \end{cases}$$



Master Theorem

Eine Rekurrenz für Teile und Herrsche / Rekursion

Für positive Konstanten a, b, c, d, sei $n = b^k$ für ein $k \in \mathbb{N}$.

$$r(n) = \begin{cases} a & \text{falls } n = 1 \text{ Basisfall} \\ cn + dr(n/b) & \text{falls } n > 1 \text{ teile und herrsche.} \end{cases}$$

Es gilt (einfache Form)

$$r(n) = \begin{cases} \Theta(n) & \text{falls } d < b \\ \Theta(n \log n) & \text{falls } d = b \\ \Theta\left(n^{\log_b d}\right) & \text{falls } d > b. \end{cases}$$

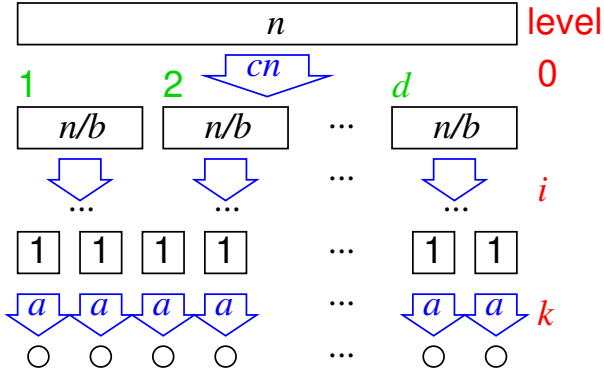
Master Theorem
Beweisskizze

Auf Ebene i, haben wir d^i Probleme der Größe $n/b^i = b^{k-i}$

cost
cn

$$d^i \cdot c \cdot \frac{n}{b^i} = cn\left(\frac{d}{b}\right)^i$$

ad^k

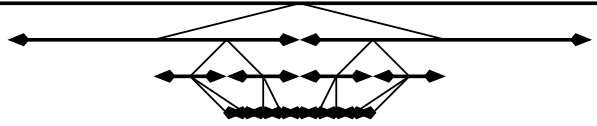


Master Theorem
Beweisskizze Fall $d < b$

geometrisch schrumpfende Reihe
$\longrightarrow$ erste Rekursionsebene kostet konstanten Teil der Arbeit

$$r(n) = \underbrace{a \cdot d^k}_{o(n)} + cn \cdot \underbrace{\sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i}_{O(1)} = \Theta(n)$$

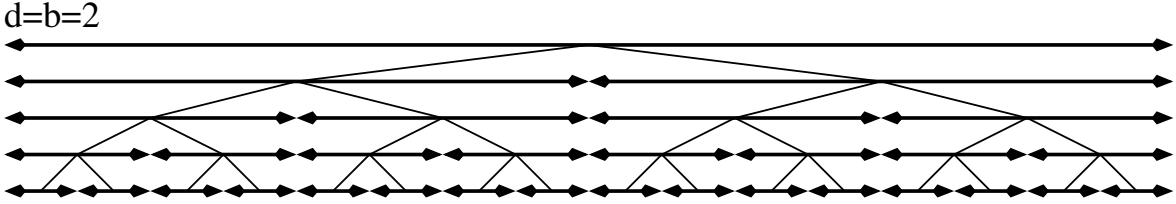
d=2, b=4



Master Theorem
Beweisskizze Fall $d = b$

gleich viel Arbeit auf allen $k = \log_b(n)$ Ebenen.

$$r(n) = an + cn \log_b n = \Theta(n \log n)$$

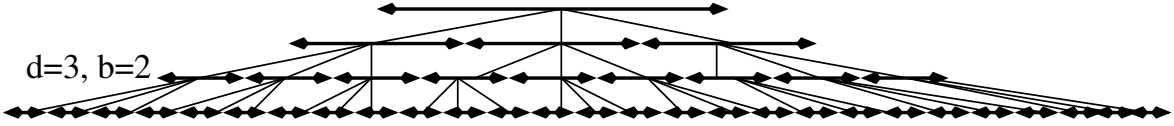


Master Theorem
Beweisskizze Fall $d > b$

geometrisch wachsende Reihe
$\longrightarrow$ letzte Rekursionsebene kostet konstanten Teil der Arbeit

$$r(n) = ad^k + cn \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i = \Theta\left(n^{\log_b d}\right)$$

beachte: $d^k = 2^{k \log d} = 2^{k \frac{\log b}{\log b} \log d} = b^{k \frac{\log d}{\log b}} = b^{k \log_b d} = n^{\log_b d}$



Master Theorem
Beispiele

Für positive Konstanten a, b, c, d, sei $n = b^k$ für ein $k \in \mathbb{N}$.

$$r(n) = \begin{cases} a & \text{falls } n = 1 \text{ Basisfall} \\ cn + dr(n/b) & \text{falls } n > 1 \text{ teile und herrsche.} \end{cases}$$

schon gesehen, kommt noch, allgemeinerer Fall

- $d < b$: Median bestimmen
- $d = b$: mergesort, quicksort
- $d > b$: Schulmultiplikation, Karatsuba-Ofman-Multiplikation

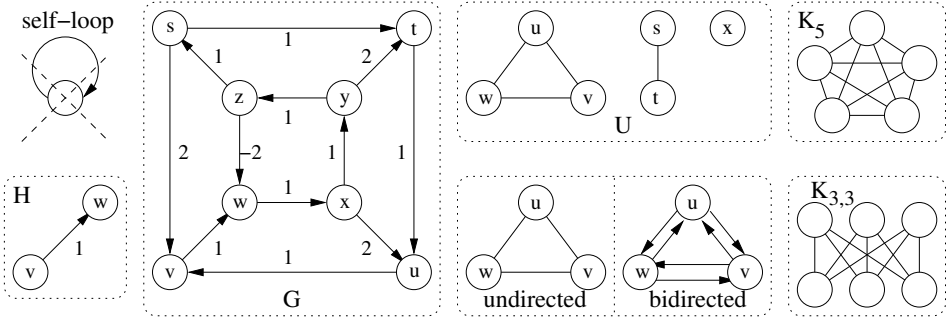
Weitere Themen

Analyse im Mittel: Später an Beispielen

Randomisierte Algorithmen: Später an Beispielen

Graphen

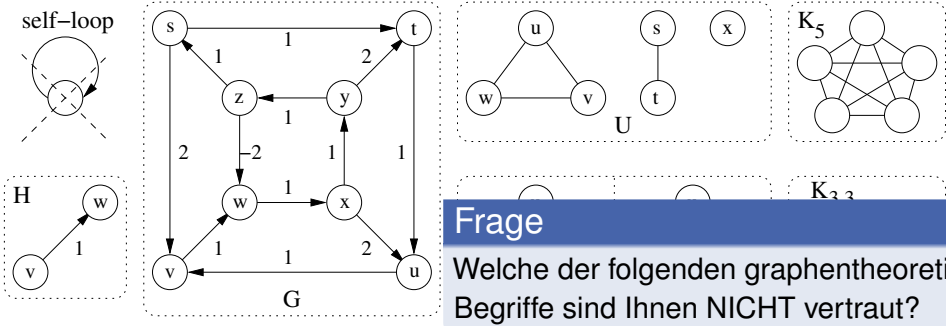
Sie kennen schon (?): Relationen, Knoten, Kanten, (un)gerichtete Graphen, Kantengewichte, Knotengrade, Kantengewichte, knoteninduzierte Teilgraphen.
Pfade (einfach, Hamilton-), Kreise, DAGs



pwa.klicker.uzh.ch/join/algoeng

Graphen

Sie kennen schon (?): Relationen, Knoten, Kanten, (un)gerichtete Graphen, Kar
Knotengrade, Kantengewichte, knoteninduzierte Teilgraphen.
Pfade (einfach, Hamilton-), Kreise, DAGs

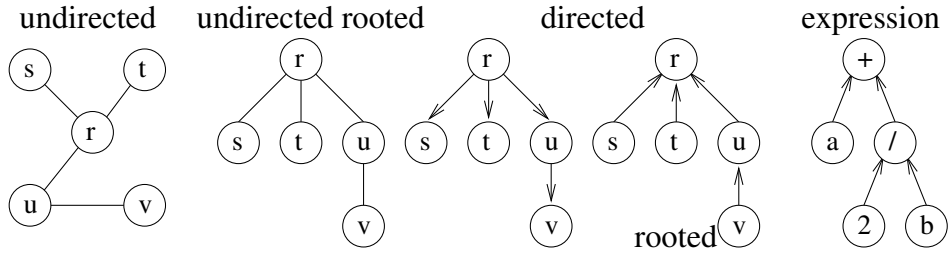


Frage

Welche der folgenden graphentheoretischen Begriffe sind Ihnen NICHT vertraut?

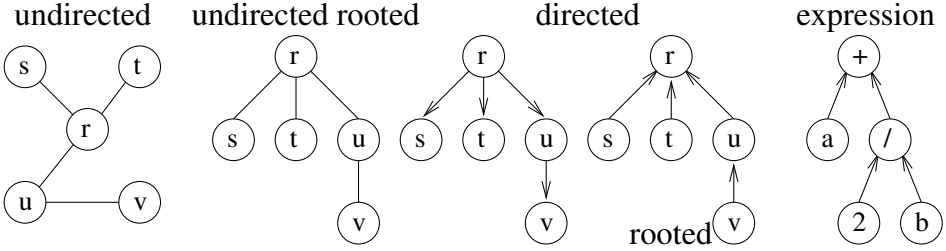
Bäume

Zusammenhang, Bäume, Wurzeln, Wälder, Kinder, Eltern, ...



Bäume

Zusammenhang, Bäume, Wurzeln, Wälder, Kinder, Eltern, ...



Frage

Gibt es Unterschiede zwischen einem gerichteten Wurzelbaum und einem Baum für einen arithmetischen Ausdruck?

Ein erster Graphalgorithmus

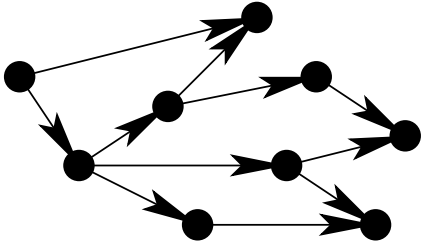
Ein DAG (directed acyclic graph, gerichteter azyklischer Graph) ist ein gerichteter Graph, der keine Kreise enthält.

Function isDAG($G = (V, E)$)
  while $\exists v \in V$: outdegree$(v) = 0$ **do**
    invariant G is a DAG iff the input graph is a DAG
    $V := V \setminus \{v\}$
    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$
  return |V|=0

Analyse: kommt auf Repräsentation an (Kapitel 8), geht aber in $\mathrm{O}(|V| + |E|)$.

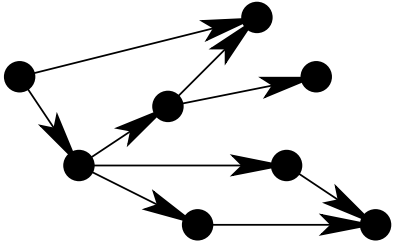
Ein erster Graphalgorithmus
Beispiel

Function isDAG($G = (V, E)$)
  while $\exists v \in V$: outdegree$(v) = 0$ **do**
    V:= $V \setminus \{v\}$
    E:= $E \setminus (\{v\} \times V \cup V \times \{v\})$
  return |V|=0



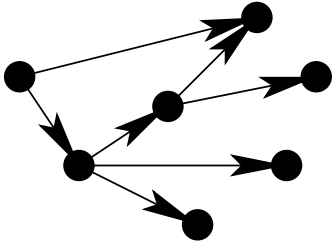
Ein erster Graphalgorithmus
Beispiel

Function isDAG($G = (V, E)$)
  while $\exists v \in V$: outdegree$(v) = 0$ **do**
    V:= $V \setminus \{v\}$
    E:= $E \setminus (\{v\} \times V \cup V \times \{v\})$
  return |V|=0



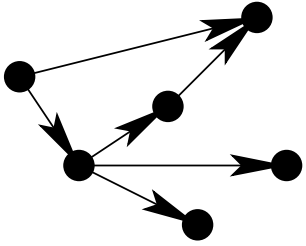
Ein erster Graphalgorithmus
Beispiel

Function isDAG($G = (V, E)$)
  while $\exists v \in V$: outdegree$(v) = 0$ **do**
    $V := V \setminus \{v\}$
    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$
  return |V|=0



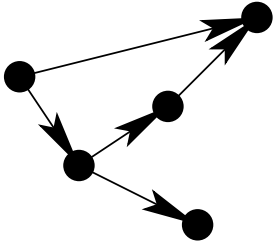
Ein erster Graphalgorithmus
Beispiel

Function isDAG($G = (V, E)$)
  while $\exists v \in V$: outdegree$(v) = 0$ **do**
    V:= $V \setminus \{v\}$
    E:= $E \setminus (\{v\} \times V \cup V \times \{v\})$
  return |V|=0



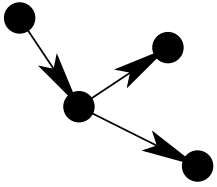
Ein erster Graphalgorithmus
Beispiel

Function isDAG($G = (V, E)$)
  while $\exists v \in V$: outdegree$(v) = 0$ **do**
    $V := V \setminus \{v\}$
    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$
  return |V|=0



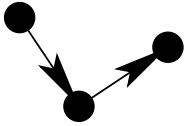
Ein erster Graphalgorithmus
Beispiel

Function isDAG($G = (V, E)$)
  while $\exists v \in V$: outdegree$(v) = 0$ **do**
    $V := V \setminus \{v\}$
    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$
  return |V|=0



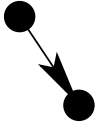
Ein erster Graphalgorithmus
Beispiel

Function isDAG($G = (V, E)$)
  while $\exists v \in V$: outdegree$(v) = 0$ **do**
    V:= $V \setminus \{v\}$
    E:= $E \setminus (\{v\} \times V \cup V \times \{v\})$
  return |V|=0



Ein erster Graphalgorithmus
Beispiel

Function isDAG($G = (V, E)$)
  while $\exists v \in V$: outdegree$(v) = 0$ **do**
    $V := V \setminus \{v\}$
    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$
  return |V|=0



Ein erster Graphalgorithmus
Beispiel

Function isDAG($G = (V, E)$)
  while $\exists v \in V$: outdegree$(v) = 0$ **do**
    $V := V \setminus \{v\}$
    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$
  return |V|=0

Ein erster Graphalgorithmus

Beispiel

Function isDAG($G = (V, E)$)
  while $\exists v \in V$: outdegree$(v) = 0$ **do**
    $V := V \setminus \{v\}$
    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$
  return |V|=0

Leerer Graph.

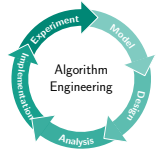
P und NP

Das kommt in "Grundlagen der theoretischen Informatik"
Ganz kurz:

- Es gibt einigermaßen gute Gründe, „effizient“ mit „polynomiell“ gleichzusetzen (d. h. Laufzeit $n^{O(1)}$).
- Es gibt viele algorithmische Probleme (NP-vollständig/hart), bei denen es SEHR überraschend wäre, wenn sie in Polynomialzeit lösbar wären.

Index

Karlsruher Institut für Technologie



Algorithmen I – 3. Folgen als Felder und Listen

Sommersemester 2025

Peter Sanders | Stand: 30. Juli 2025

Folgen

Spielen in der Informatik eine überragende Rolle.
Das sieht man schon an der Vielzahl von Begriffen:
Folge, Feld, Schlange, Liste, Datei, Stapel, Zeichenkette, Log...
(sequence, array, queue, list, file, stack, string, log...).
Wir unterscheiden:

- abstrakter Begriff $\langle 2, 3, 5, 7, 9, 11, \ldots \rangle$ — Mathe
- Funktionalität (stack, ...) — Softwaretechnik
- Repräsentation — Algorithmik

Anwendungen

- Ablegen und Bearbeiten von Daten aller Art
- Konkrete Repräsentation abstrakterer Konzepte wie Menge, Graph (Kapitel 8),...

Form Follows Function

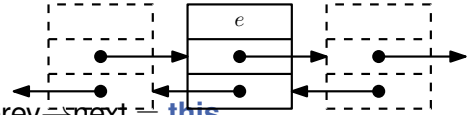
Operation	List	SList	UArray	CArray	explanation '*'
$[\cdot]$	n	n	1	1	
$\lvert\cdot\rvert$	1^*	1^*	1	1	not with inter-list splice
first	1	1	1	1	
last	1	1	1	1	
insert	1	1^*	n	n	insertAfter only
remove	1	1^*	n	n	removeAfter only
pushBack	1	1	1^*	1^*	amortized
pushFront	1	1	n	1^*	amortized
popBack	1	n	1^*	1^*	amortized
popFront	1	1	n	1^*	amortized
concat	1	1	n	n	
splice	1	1	n	n	
findNext,...	n	n	n^*	n^*	cache-efficient

Doppelt verkettete Listen

Doppelt verkettete Listen
Listenglieder (Items)

Class Handle = **Pointer to** Item

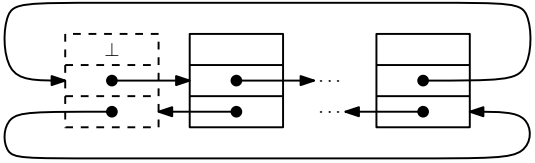
Class Item **of** Element // one link in a doubly linked list
    e : Element
    next : Handle
    prev : Handle
    invariant next→prev = prev→next = **this**



Problem:
Vorgänger des ersten Listenelements?
Nachfolger des letzten Listenelements?

Doppelt verkettete Listen

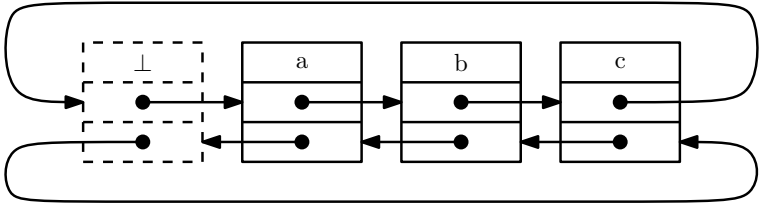
Trick: dummy header



+ Invariante immer erfüllt
+ Vermeidung vieler Sonderfälle
  - ⇝ einfach
  - ⇝ lesbar
  - ⇝ schnell
  - ⇝ testbar
  - ⇝ elegant
- Speicherplatz (irrelevant bei langen Listen)

Doppelt verkettete Listen

Dummy header – Beispiel $\langle a, b, c \rangle$

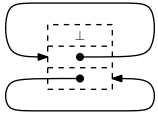


Doppelt verkettete Listen
Die Listenklasse

Class List **of** Element
  // Item h is the predecessor of the first element
  // and the successor of the last element.
  Function head : Handle; **return address of** h

// Pos. before any proper element

$h = \begin{pmatrix} \bot \\ head \\ head \end{pmatrix}$: Item



// init to empty sequence

  // Simple access functions
  Function isEmpty : $\{0,1\}$; **return** h.next = head // $\langle\rangle$?
  Function first : Handle; **assert** ¬isEmpty; **return** h.next
  Function last : Handle; **assert** ¬isEmpty; **return** h.prev
  ⋮

Doppelt verkettete Listen
Splice

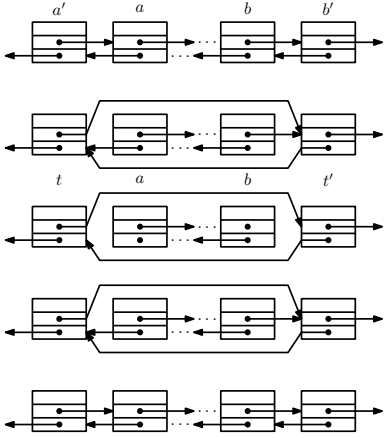
Procedure splice(a, b, t : Handle) //Cut out $\langle a, \ldots, b \rangle$ and insert after t
assert b is not before $a \wedge t \notin \langle a, \ldots, b \rangle$
//Cut out $\langle a, \ldots, b \rangle$
$a' :=$ $a\rightarrow$prev
$b' :=$ $b\rightarrow$next

$a'\rightarrow$next $:= b'$
$b'\rightarrow$prev $:= a'$

//insert $\langle a, \ldots, b \rangle$ after t
$t' :=$ $t\rightarrow$next

$b\rightarrow$next $:= t'$
$a\rightarrow$prev $:= t$

$t\rightarrow$next $:= a$
$t'\rightarrow$prev $:= b$



Doppelt verkettete Listen
Splice Beispiel

$$\langle 1, 2, 3, 4, \overbrace{5}^{a}, 6, 7, \overbrace{8}^{b}, 9, 10\rangle, \langle u, v, \overbrace{w}^{t}, x, y, z\rangle$$

$$\Downarrow$$

$$\langle 1, 2, 3, 4, 9, 10\rangle, \langle u, v, w, 5, 6, 7, 8, x, y, z\rangle$$

Doppelt verkettete Listen

Der Rest sind Einzeiler (?)

// Moving elements around within a sequence.

// $\langle \ldots, a, b, c \ldots, a', c', \ldots \rangle \mapsto \langle \ldots, a, c \ldots, a', b, c', \ldots \rangle$

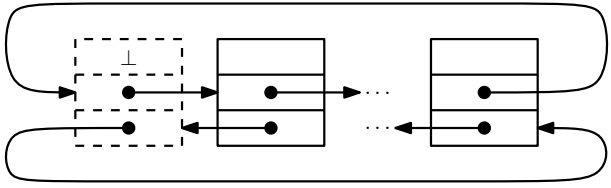
Procedure moveAfter(b, a' : Handle) splice(b, b, a')

// $\langle x, \ldots, a, b, c, \ldots \rangle \mapsto \langle b, x, \ldots, a, c, \ldots \rangle$

Procedure moveToFront(b : Handle) moveAfter(b, head)

// $\langle \ldots, a, b, c, \ldots, z \rangle \mapsto \langle \ldots, a, c, \ldots, z, b \rangle$

Procedure moveToBack(b : Handle) moveAfter(b, last)



Doppelt verkettete Listen
Oder doch nicht? Speicherverwaltung!

naiv / blauäugig / optimistisch:
Speicherverwaltung der Programmiersprache $\rightsquigarrow$ potentiell sehr langsam
Hier: einmal existierende Variable (z. B. `static` member in Java) freeList enthält ungenutzte Items. checkFreeList stellt sicher, dass die nicht leer ist.
Reale Implementierungen:

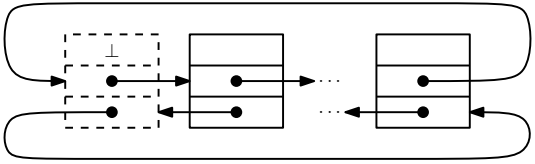
- naiv aber mit guter Speicherverwaltung
- verfeinerte Freelistkonzepte (klassenübergreifend, Freigabe,. . .)
- anwendungsspezifisch, z. B. wenn man weiß wieviele Items man insgesamt braucht

Doppelt verkettete Listen
Items löschen

// $\langle \ldots, a, b, c, \ldots \rangle \mapsto \langle \ldots, a, c, \ldots \rangle$
Procedure remove(b : Handle) moveAfter(b, freeList.head)

// $\langle a, b, c, \ldots \rangle \mapsto \langle b, c, \ldots \rangle$
Procedure popFront remove(first)

// $\langle \ldots, a, b, c \rangle \mapsto \langle \ldots, a, b \rangle$
Procedure popBack remove(last)



Doppelt verkettete Listen
Elemente einfügen

Karlsruher Institut für Technologie

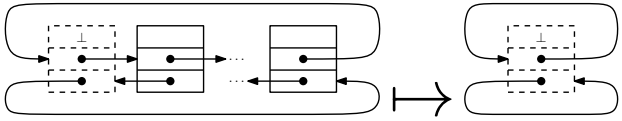
```
// ⟨..., a, b, ...⟩ ↦ ⟨..., a, e, b, ...⟩
Function insertAfter(x : Element; a : Handle) : Handle
    checkFreeList                          // make sure freeList is nonempty.
    a' := freeList.first                   // Obtain an item a' to hold x,
    moveAfter(a', a)                       // put it at the right place.
    a' → e := x                            // and fill it with the right content.
    return a'

Function insertBefore(x : Element; b : Handle) : Handle
    return insertAfter(e, b →prev)
Procedure pushFront(x : Element) insertAfter(x, head)
Procedure pushBack(x : Element) insertAfter(x, last)
```

Doppelt verkettete Listen
Ganze (Teil)Listen Manipulieren

$// (\langle a, \ldots, b\rangle, \langle c, \ldots, d\rangle) \mapsto (\langle a, \ldots, b, c, \ldots, d\rangle, \langle\rangle)$
Procedure concat(L' : List)
    splice(L'.first, L'.last, last)

$// \langle a, \ldots, b\rangle \mapsto \langle\rangle$
Procedure makeEmpty
    freeList.concat(**this**)

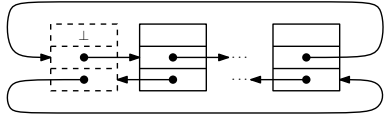


Das geht in konstanter Zeit – unabhängig von der Listenlänge!

Doppelt verkettete Listen
Suchen

Trick: gesuchtes Element in Dummy-Item schreiben:

Function findNext(x : Element; from : Handle) : Handle
    $h.e = x$ // Sentinel
    while from $\rightarrow e \neq x$ **do**
        from:= from $\rightarrow$ next
    return from

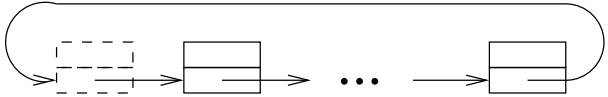


Spart Sonderfallbehandlung.
Allgemein: ein Wächter-Element (engl. Sentinel) fängt Sonderfälle ab.
$\rightsquigarrow$ einfacher, schneller,...

Beispiel: Listenlängen

Verwalte zusätzliches Member size.
Problem: inter-list splice geht nicht mehr in konstanter Zeit
Die Moral von der Geschicht:
Es gibt nicht DIE Listenimplementierung.

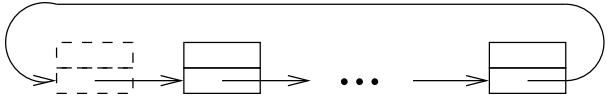
Einfach verkettete Listen



Vergleich mit doppelt verketteten Listen

- weniger Speicherplatz
- Platz ist oft auch Zeit
- eingeschränkter, z. B. kein remove
- merkwürdige Benutzerschnittstelle, z. B. removeAfter

Einfach verkettete Listen
Invariante?



Betrachte den Graphen $G = (\text{Item}, E)$ mit $E = \{(u, v) : u \in \text{Item}, v = u.\text{next}\}$

- u.next zeigt immer auf ein Item
- $\forall u \in \mathit{Item} : \text{indegree}_G(u) = 1$.
  Wohl definiert obwohl nicht unbedingt leicht zu testen.

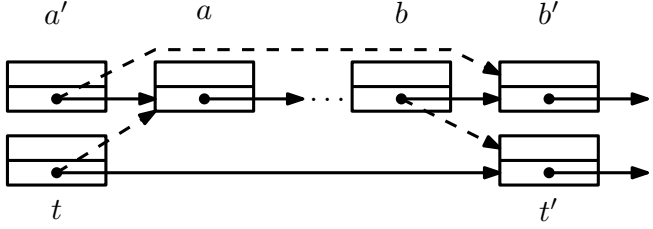
Folge: Items bilden Kollektion von Kreisen

Einfach verkettete Listen
splice

$//\left(\langle \ldots, a', a, \ldots, b, b' \ldots \rangle, \langle \ldots, t, t', \ldots \rangle\right) \mapsto$
$//\left(\langle \ldots, a', b' \ldots \rangle, \langle \ldots, t, a, \ldots, b, t', \ldots \rangle\right)$

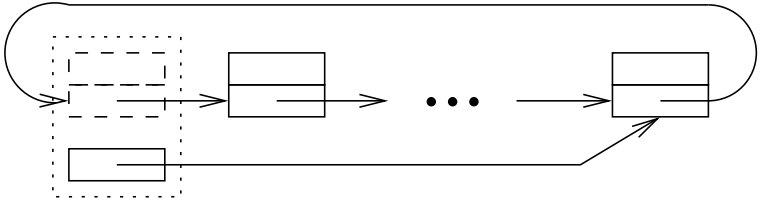
Procedure splice(a', b, t : SHandle)

$$\begin{pmatrix} a' \to \text{next} \\ t \to \text{next} \\ b \to \text{next} \end{pmatrix} := \begin{pmatrix} b \to \text{next} \\ a' \to \text{next} \\ t \to \text{next} \end{pmatrix}$$



Einfach verkettete Listen
pushBack

Zeiger auf letztes Item erlaubt Operation pushBack



Listen:
Zusammenfassung, Verallgemeinerungen

- Zeiger zwischen Items ermöglichen flexible, dynamische Datenstrukturen
  später: Bäume, Prioritätslisten
- (einfache) Datenstrukturinvarianten sind Schlüssel zu einfachen, effizienten Datenstrukturen
- Dummy-Elemente, Wächter,... erlauben Einsparung von Sonderfällen
- Einsparung von Sonderfällen machen Programme, einfacher, lesbarer, testbarer und schneller
- Datenstrukturaugmentierung beschleunigt bestimmte Operationen (Nebenwirkungen beachten). Beispiel: Listenlänge
- Speicherverwaltung ist ein komplexes Thema. Freelists sind eine interessante Option.

Felder (Arrays)

$A[i] = a_i$ falls $A = \langle a_0, \ldots, a_{n-1} \rangle$

Beschränkte Felder (Bounded Arrays)

Eingebaute Datenstruktur: Ein Stück Hauptspeicher + Adressrechnung
Größe muss von Anfang an bekannt sein

Unbeschränkte Felder (Unbounded Arrays)

$$\langle e_0, \ldots, e_n \rangle.\text{pushBack}(\text{e}) \rightsquigarrow \langle e_0, \ldots, e_n, e \rangle,$$
$$\langle e_0, \ldots, e_n \rangle.\text{popBack} \rightsquigarrow \langle e_0, \ldots, e_{n-1} \rangle,$$
$$\text{size}(\langle e_0, \ldots, e_{n-1} \rangle) = n \ .$$

Unbeschränkte Felder
Anwendungen

wenn man nicht weiß, wie lang das Feld wird.
Beispiele:

- Datei zeilenweise einlesen
- später: Stacks, Queues, Prioritätslisten, ...

Unbeschränkte Felder
Grundidee

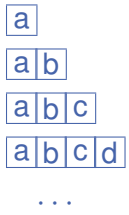
wie beschränkte Felder: Ein Stück Hauptspeicher

pushBack: Element anhängen, size$++$
Kein Platz?: umkopieren und (größer) neu anlegen

popBack: size$--$
Zuviel Platz?: umkopieren und (kleiner) neu anlegen

Immer passender Platzverbrauch?

n pushBack Operationen brauchen Zeit
$O\left(\sum_{i=1}^{n} i\right) = O(n^2)$
Geht es schneller?



Unbeschränkte Felder
mit teilweise ungenutztem Speicher

Class UArray **of** Element
- w=1 : $\mathbb{N}$ // allocated size
- n=0 : $\mathbb{N}$ // current size.
- **invariant** $n \leq w < \alpha n$ or $n = 0$ and $w \leq 2$ // e.g., $\alpha = 4$
- b : **Array** $[0..w-1]$ **of** Element

- **Operator** $[i : \mathbb{N}]$: Element
  - **assert** $0 \leq i < n$
  - **return** $b[i]$
- **Function** size : $\mathbb{N}$ **return** n

Unbeschränkte Felder
mit teilweise ungenutztem Speicher

Procedure pushBack(e : Element)
    if $n = w$ **then**
        reallocate($2n$)
    $b[n]$:= e
    n++

// Example for $n = w = 4$:
// $b \rightarrow$ | 0 | 1 | 2 | 3 |
// $b \rightarrow$ | 0 | 1 | 2 | 3 | | | | |
// $b \rightarrow$ | 0 | 1 | 2 | 3 | e | | | |
// $b \rightarrow$ | 0 | 1 | 2 | 3 | e | | | |

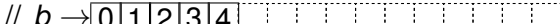
Procedure reallocate(w' : $\mathbb{N}$)
    w:= w'
    b':= **allocate**
        Array $[0..w'-1]$ **of** Element
    $(b'[0], \ldots, b'[n-1])$:=
        $(b[0], \ldots, b[n-1])$
    dispose b
    b:= b'

// Example for $w = 4$, $w' = 8$:
// $b \rightarrow$ | 0 | 1 | 2 | 3 |
// $b' \rightarrow$ | | | | | | | | |
// $b' \rightarrow$ | 0 | 1 | 2 | 3 | | | | |
// $b \rightarrow$ ~~| 0 | 1 | 2 | 3 |~~
// pointer assignment $b \rightarrow$ | 0 | 1 | 2 | 3 | | | | |

Unbeschränkte Felder
Kürzen

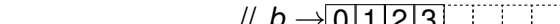
Procedure popBack
    assert $n > 0$
    $n{-}{-}$
    if $4n \leq w \wedge n > 0$ **then**
        reallocate($2n$)

// Example for $n = 5$, $w = 16$:

// $b \rightarrow$ 

// $b \rightarrow$ 

// reduce waste of space

// $b \rightarrow$ 

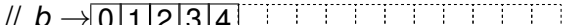
Unbeschränkte Felder

Kürzen

pwa.klicker.uzh.ch/join/algoeng

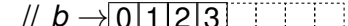
Procedure popBack
    assert $n > 0$
    $n{-}{-}$
    if $4n \leq w \wedge n > 0$ **then**
        reallocate($2n$)

// Example for n :

// $b \rightarrow$ 

// $b \rightarrow$ 

// reduce waste of space

// $b \rightarrow$ 

Frage

Was geht schief, wenn man ein unbeschränktes Feld auf die passende Größe n kürzt, anstatt auf $2n$?

Unbeschränkte Felder
Amortisierte Komplexität

Sei u ein anfangs leeres, unbeschränktes Feld.
Jede Operationenfolge $\sigma = \langle \sigma_1, \ldots, \sigma_m \rangle$
von pushBack oder popBack Operationen auf u
wird in Zeit $O(m)$ ausgeführt.
Sprechweise:
pushBack und popBack haben amortisiert konstante Ausführungszeit —

$$O\left(\overbrace{m}^{\text{Gesamtzeit}} \Big/ \underbrace{m}_{\#\text{Ops}}\right) = O(1) \quad .$$

Unbeschränkte Felder

Beweis: Konto-Methode (oder Versicherung)

Operation	Kosten		Typ
pushBack	$\circ\circ$	(2 Token)	einzahlen
popBack	$\circ$	(1 Token)	einzahlen
reallocate($2n$)	$n \times \circ$	(n Token)	abheben

(Ein Token bezahlt eine Elementverschiebung in reallocate)

Zu zeigen: keine Überziehungen

Erster Aufruf von reallocate: kein Problem

($n = 2$, $\geq$ 2tes pushBack)

Unbeschränkte Felder

Beweis: Konto-Methode (oder Versicherung)

Operation	Kosten		Typ
pushBack	$\circ\circ$	(2 Token)	einzahlen
popBack	$\circ$	(1 Token)	einzahlen
reallocate($2n$)	$n \times \circ$	(n Token)	abheben

(Ein Token bezahlt eine Elementverschiebung in reallocate)

Weitere Aufrufe von reallocate:

rauf: $\text{reallocate}(2n) \underbrace{\overset{\geq n \times \text{pushBack}}{\rightsquigarrow}}_{\geq n \times \circ\circ} \text{reallocate}(4n)$

runter: $\text{reallocate}(2n) \underbrace{\overset{\geq n/2 \times \text{popBack}}{\rightsquigarrow}}_{\geq n/2 \times \circ} \text{reallocate}(n)$

□

Unbeschränkte Felder

Amortisierte Analyse – allgemeiner

- Z: Menge von Operationen, z. B. $\{\text{pushBack}, \text{popBack}\}$
- s: Zustand der Datenstruktur
- $A_X(s)$: amortisierte Kosten von Operation $X \in Z$ in Zustand s
- $T_X(s)$: tatsächliche Kosten von Operation $X \in Z$ in Zustand s
- Berechnung: $s_0 \xrightarrow{Op_1} s_1 \xrightarrow{Op_2} s_2 \xrightarrow{Op_3} \cdots \xrightarrow{Op_n} s_n$

Die angenommenen amortisierten Kosten sind korrekt, wenn

$$\underbrace{\sum_{1 \le i \le n} T_{Op_i}(s_{i-1})}_{\text{tatsächliche Gesamtkosten}} \le c + \underbrace{\sum_{1 \le i \le n} A_{Op_i}(s_{i-1})}_{\text{amortisierte Gesamtkosten}}$$

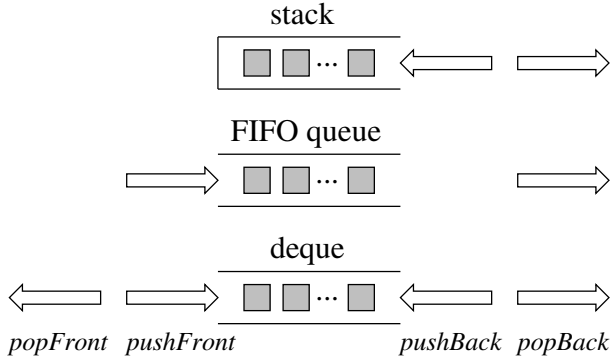
für eine Konstante c

Amortisierte Analyse – Diskussion

- Amortisierte Laufzeiten sind leichter zu garantieren als tatächliche.
- Der Gesamtlaufzeit tut das keinen Abbruch.
- Deamortisierung oft möglich, aber kompliziert und teuer
  - Wie geht das mit unbeschränkten Feldern?
  - Anwendung: Echtzeitsysteme
  - Anwendung: Parallelverarbeitung

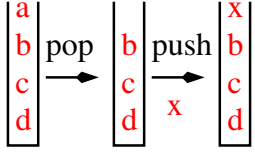
Stapel und Schlangen

- einfache Schnittstellen
- vielseitig einsetzbar
- austauschbare, effiziente Implementierungen
- wenig fehleranfällig



Stapel
Operationen:

push/pop, entsprechen
pushFront/popFront oder pushBack/popBack für Folgen



Stapel

Implementierungsvarianten

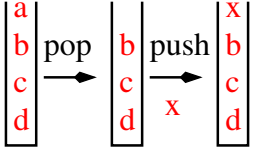
Operationen: push/pop, entsprechen pushFront/popFront oder pushBack/popBack für Folgen

List: OK aber doppelte Verkettung ist overkill

SList: mittels pushFront/popFront.
Endezeiger unnötig, dummy item unnötig

UArray: mittels pushBack/popBack. Cache-effizient aber nur amortisiert konstante Laufzeit pro Operation

In der Vorlesung Algorithm Engineering lernen wir bessere Implementierungen kennen.



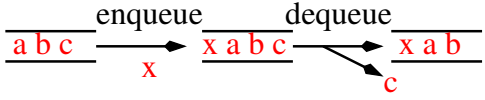
Stapel
Anwendungen

- Rekursion
- Klammerstrukturen,. . . , Parser
- Daten “irgendwie” ablegen und wieder herausholen

Warteschlangen / First-In-First-Out / FIFO

Operationen: enqueue/dequeue, entsprechen pushFront/popBack oder pushBack/popFront für Folgen

Beispiel:



FIFO – Implementierungsvarianten

Operationen: enqueue/dequeue, entsprechen
pushFront/popBack oder pushBack/popFront für Folgen

List: OK aber doppelte Verkettung ist overkill

SList: mittels pushBack/popFront. Endezeiger wichtig, dummy item unnötig

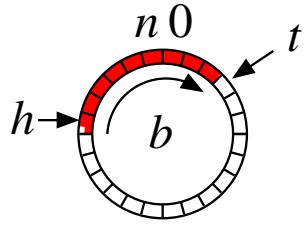
Array,UArray: scheinbar nicht effizient möglich

CArray: “zyklisches” Array
Übung: unbounded cyclic array

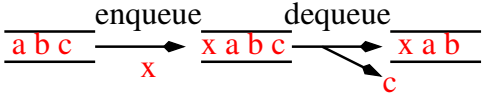
In der Vorlesung Algorithm Engineering lernen wir bessere Implementierungen kennen.

FIFO Warteschlagen (queue) mittels zyklischer Felder

Class BoundedFIFO($n : \mathbb{N}$) **of** Element
- b : **Array** $[0..n]$ **of** Element // CArray
- h=0 : $\mathbb{N}$
- t=0 : $\mathbb{N}$
- **Function** isEmpty : $\{0,1\}$; **return** $h = t$
- **Function** first : Element; **assert** ¬isEmpty; **return** $b[h]$
- **Function** size : $\mathbb{N}$; **return** $(t - h + n + 1)$ **mod** $(n + 1)$
- **Procedure** pushBack(x : Element)
  - **assert** size$< n$
  - $b[t] := x$
  - $t := (t + 1)$ **mod** $(n + 1)$
- **Procedure** popFront **assert** ¬isEmpty; $h := (h + 1)$ **mod** $(n + 1)$

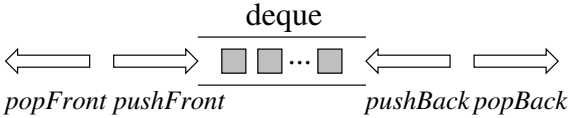


Warteschlangen
Anwendungen



- Datenpuffer für
  - Netzwerke
  - Pipeline-Verarbeitung
- Job-Queues (Fairness...)
- Breitensuche in Graphen (siehe Kapitel 9.1)

Deque
Double-Ended Queues



Aussprache wie “dek”.

List: OK

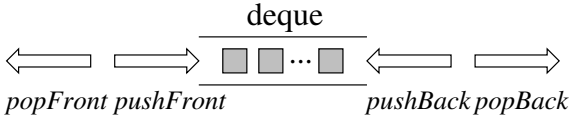
SList: Nein (aber push/pop-Front und pushBack OK)

Array,UArray: Nein

CArray: Ja

Übung: Pseudocode für Deque mittels CArray

Deque
Anwendungen



relativ selten. Oft werden nur 3 der vier Operationen benötigt.

- Work Stealing Load Balancing
- Undo/Redo Operationspuffer

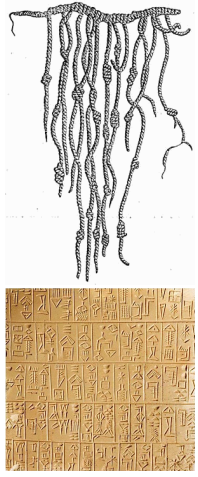
Vergleich: Listen – Felder

Vorteile von Listen

- flexibel
- remove, splice,. . .
- kein Verschnitt

Vorteile von Feldern

- beliebiger Zugriff
- einfach
- kein Overhead für Zeiger
- Cache-effizientes scanning



Vergleich: Listen – Felder

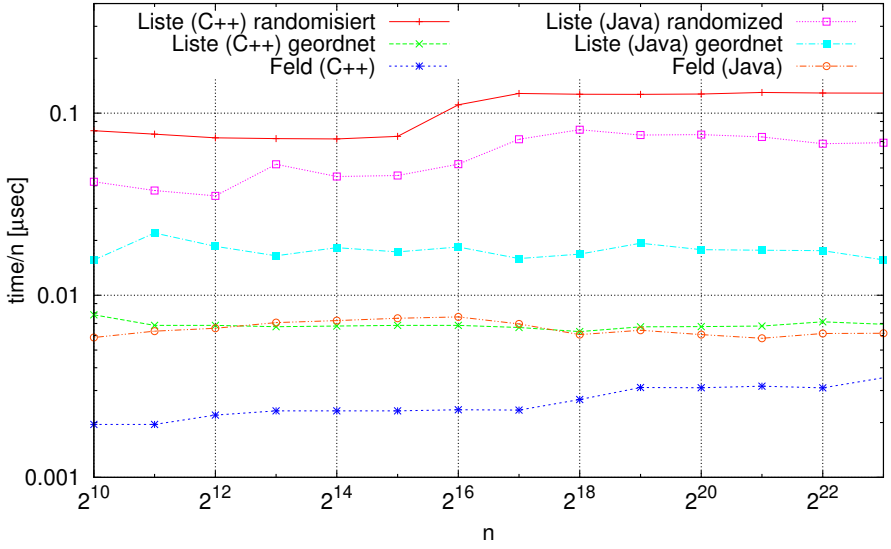
Operation	List	SList	UArray	CArray	explanation '*'
$[\cdot]$	n	n	1	1	
$\lvert\cdot\rvert$	1^*	1^*	1	1	not with inter-list splice
first	1	1	1	1	
last	1	1	1	1	
insert	1	1^*	n	n	insertAfter only
remove	1	1^*	n	n	removeAfter only
pushBack	1	1	1^*	1^*	amortized
pushFront	1	1	n	1^*	amortized
popBack	1	n	1^*	1^*	amortized
popFront	1	1	n	1^*	amortized
concat	1	1	n	n	
splice	1	1	n	n	
findNext,...	n	n	n^*	n^*	cache-efficient

Listen und Arrays in realen Programmiersprachen

Datenstruktur	Sprache	Name
unbounded array	C++	vector
(meist ohne	Java	ArrayList/Vector
schrumpfen)	Rust	Vec
doppelt verkettete	C++	list
Liste	Java	LinkedList
	Rust	LinkedList
einfach verkettete	C++	forward_list
Liste	Java	–
	Rust	–
Stack, Queue, Deque	C++	stack, queue, deque
	Java	Stack, Queue, Deque
	Rust	Vec, Queue/Buffer, VecDeque

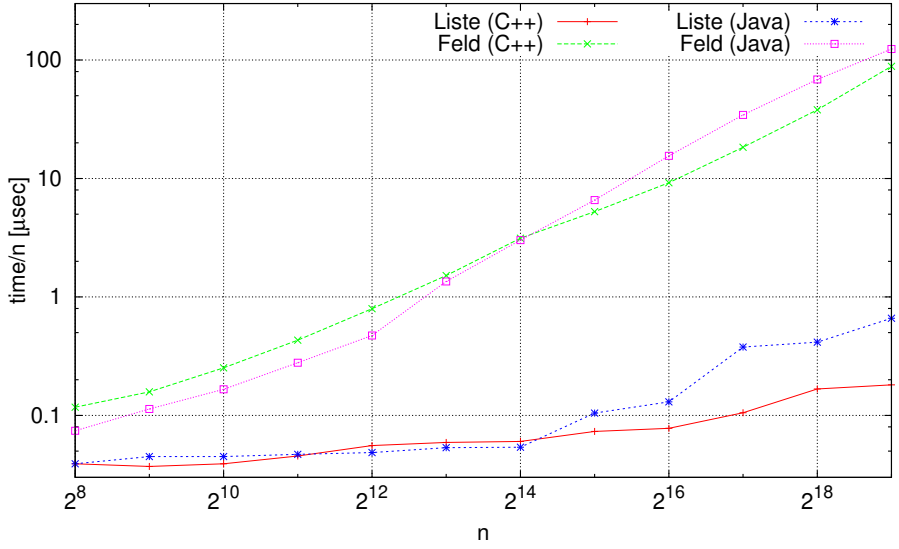
uvam

Messungen
Iterieren



Messungen

Einfügen an zufälliger Position

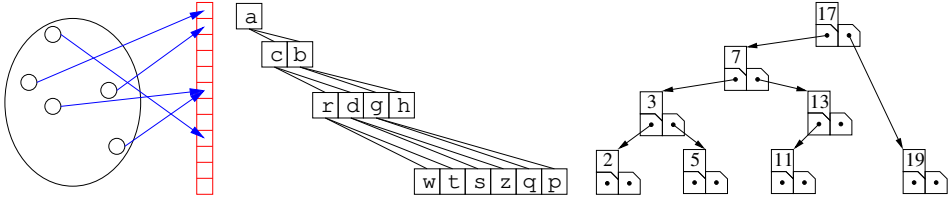


Ausblick: Weitere Repräsentationen von Folgen

Hashtabellen: schnelles Einfügen, Löschen und Suchen — Kapitel 4

Prioritätslisten: schnelles Einfügen, Minimum Entfernen — Kapitel 6

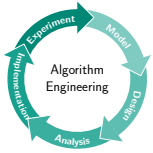
Suchbäume,. . . : sortierte Folgen – einfügen, löschen, suchen, Bereichsanfragen,. . . — Kapitel 7



In der Vorlesung “Algorithm Engineering”:
schlauere Repräsentationen, z.B. list/array-of-blocks

Index

Karlsruher Institut für Technologie

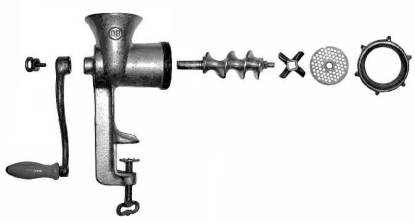


Algorithmen I – 4. Hashing (Streuspeicherung)

Sommersemester 2025

Peter Sanders | Stand: 30. Juli 2025

Hashing (Streuspeicherung)



“to hash” $\approx$ “völlig durcheinander bringen”.
Paradoxerweise hilft das, Dinge wiederzufinden

Hashing
Hashtabellen

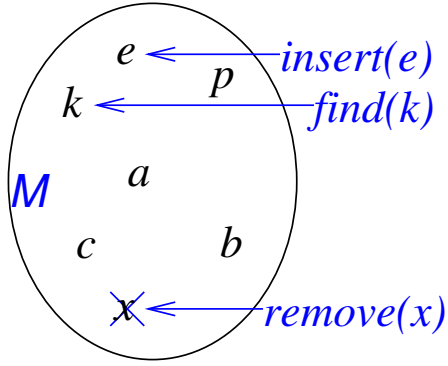
speichere Menge $M \subseteq$ Element.
key(e) ist eindeutig für $e \in M$.
unterstütze Wörterbuch-Operationen in Zeit $O(1)$.

M.insert(e : Element): $M := M \cup \{e\}$

M.remove(k : Key): $M := M \setminus \{e\}$, $e = k$

M.find(k : Key): return $e \in M$ with $e = k$;
$\perp$ falls nichts gefunden

Anderes Interface: map/partielle Funktion
Key$\rightarrow$Element
$M[k] = M$.find(k)



Hashing
Exkurs: Konventionen für Elemente

Viele Datenstrukturen repräsentieren Mengen (engl. auch collection classes).
Die Mengenelemente e haben Schlüssel $\text{key}(e)$.
Elementvergleich hier gleichbedeutend mit Schlüsselvergleich.
$e < / > / = e'$ gdw. $\text{key}(e) < / > / = \text{key}(e')$.
(Fast) äquivalente Sichtweise: Schlüssel-Wert-Paare.

Hashing
Anwendungen

- Entfernen exakter Duplikate
- Schach (oder andere kombinatorische Suchprogramme):
  welche Stellungen wurden bereits durchsucht?
- Symboltabelle bei Compilern
- Assoziative Felder bei Script-Sprachen wie javascript, perl oder awk
- Datenbank-Gleichheits-Join (wenn eine Tabelle in den Speicher passt)
- Unsere Routenplaner: Teilmengen von Knoten, z. B. Suchraum
- ...

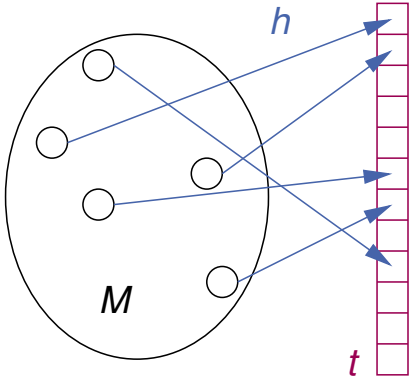
Hashing
Überblick

- Grundidee
- Hashing mit verketteten Listen
- Analyse
- Hashing mit Arrays

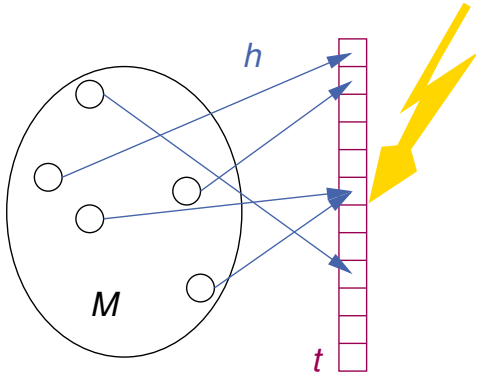
Hashtabellen

Ein (über)optimistischer Ansatz

Eine perfekte Hash-Funktion h bildet Elemente von M injektiv
auf eindeutige Einträge der Tabelle $t[0..m-1]$ ab, d. h., $t[h(\text{key}(e))] = e$

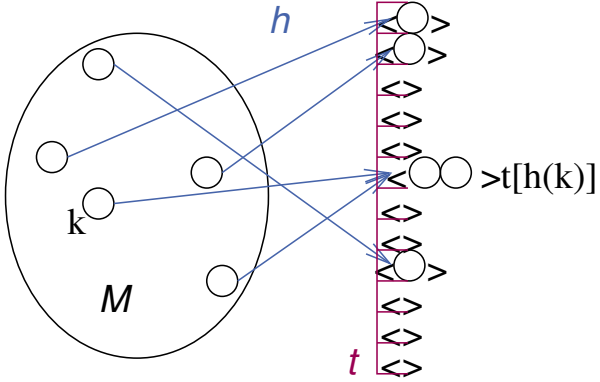


Perfekte Hash-Funktionen sind schwer zu finden. Beispiel: Geburtstagsparadox (stay tuned)



Hashtabellen
Kollisionsauflösung

Eine Möglichkeit: Tabelleneinträge: Elemente $\rightsquigarrow$ Folgen von Elementen



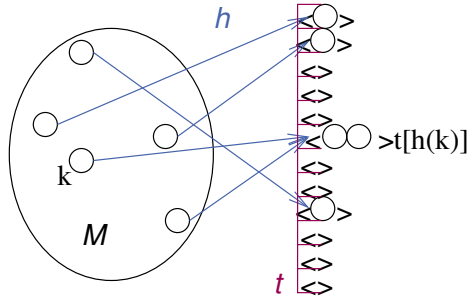
Hashing mit verketteten Listen

Implementiere die Folgen in den Tabelleneinträgen durch einfach verkettete Listen

insert(e): Füge e am Anfang (?) von $t[h(e)]$ ein.

remove(k): Durchlaufe $t[h(k)]$.
Element e mit key(e) $= k$ gefunden?
$\rightsquigarrow$ löschen und zurückliefern.

find(k) : Durchlaufe $t[h(k)]$.
Element e mit key(e) $= k$ gefunden?
$\rightsquigarrow$ zurückliefern.
Sonst: $\perp$ zurückgeben.



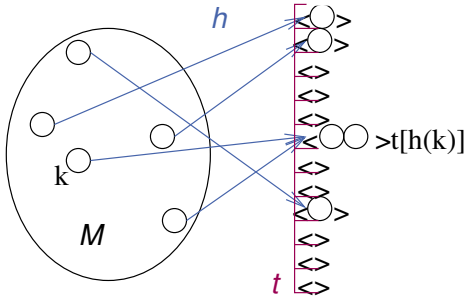
Hashing mit verketteten Listen

Implementiere die Folgen in den Tabelleneinträgen durch einfach verkettete Li

insert(e): Füge e am Anfang (?) von $t[h(e)]$ ein.

remove(k): Durchlaufe $t[h(k)]$.
Element e mit key(e) $= k$ gefunden?
$\rightsquigarrow$ löschen und zurückliefern.

find(k) : Durchlaufe $t[h(k)]$.
Element e mit key(e) $= k$ gefunden?
$\rightsquigarrow$ zurückliefern.
Sonst: $\perp$ zurückgeben.

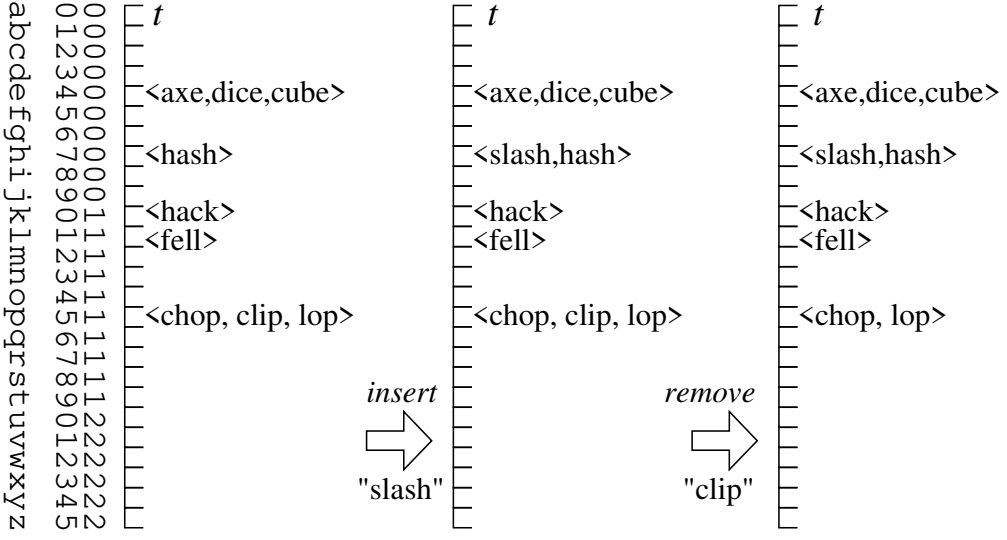


Frage

Was sollte passieren, wenn ein Element eingefügt wird, das bereits enthalten ist?

Hashing mit verketteten Listen

Beispiel (Hashfunktion ist letzter Buchstabe)



Hashing mit verketteten Listen
Analyse

insert(e): konstante Zeit

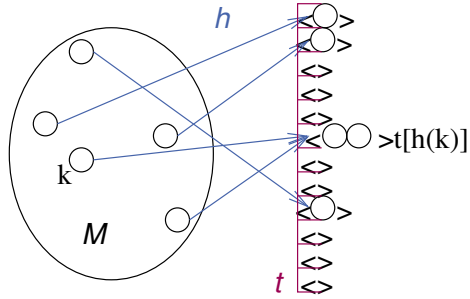
remove(k): $O($Listenlänge$)$

find(k) : $O($Listenlänge$)$

Aber wie lang werden die Listen?

Schlechtester Fall: $O(|M|)$

Besser wenn wir genug Chaos anrichten?



Analyse

Etwas Wahrscheinlichkeitstheorie für den Hausgebrauch

Elementarereignisse Ω

Ereignisse: Teilmengen von Ω

p_x =Wahrscheinlichkeit von $x \in \Omega$. $\sum_x p_x = 1$!

Gleichverteilung: $p_x = \frac{1}{|\Omega|}$

$\mathbb{P}[\mathcal{E}] = \sum_{x \in \mathcal{E}} p_x$

Zufallsvariable (ZV) $X_0 : \Omega \to \mathbb{R}$

0-1-Zufallsvariable (Indikator-ZV) $I : \Omega \to \{0, 1\}$

Erwartungswert $\mathrm{E}[X] = \sum_{y \in \Omega} p_y X(y)$

Linearität des Erwartungswerts: $\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y]$

Hash-Beispiel

Hash-Funktionen $\{0..m-1\}^{\mathrm{Key}}$

$\mathcal{E}_{42} = \{h \in \Omega : h(4) = h(2)\}$

$p_h = m^{-|\mathrm{Key}|}$

$\mathbb{P}[\mathcal{E}_{42}] = \frac{1}{m}$

$X = |\{e \in M : h(e) = 0\}|.$

$\mathrm{E}[X] = \frac{|M|}{m}$

Analyse

Beispiel: Variante des Geburtstagsparadoxon

Wieviele Gäste muss eine Geburtstagsparty “im Mittel” haben, damit mindestens zwei Gäste den gleichen Geburtstag haben?

Gäste $1..n$.

Elementarereignisse: $h \in \Omega = \{0..364\}^{\{1..n\}}$.

Definiere Indikator-ZV $I_{ij} = 1$ gdw $h(i) = h(j)$.

Anzahl Paare mit gleichem Geburtstag: $X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} I_{ij}$.

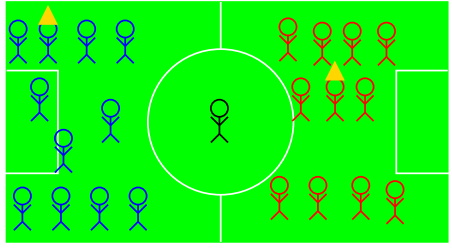
$$\mathrm{E}[X] = \mathrm{E}[\sum_{i=1}^{n} \sum_{j=i+1}^{n} I_{ij}] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathrm{E}[I_{ij}]$$

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{P}[I_{ij} = 1] = \frac{n(n-1)}{2} \cdot \frac{1}{365}$$

$$\stackrel{!}{=} 1 \Leftrightarrow n = -\frac{1}{2} + \sqrt{\frac{1}{2^2} + 730} \approx 26.52$$

Analyse
Mehr zum Geburtstagsparadoxon

Standardfomulierung: Ab wann lohnt es sich zu wetten, dass es zwei Gäste mit gleichem Geburtstag gibt? Etwas komplizierter. Antwort: $n \geq 23$
Verallgemeinerung: Jahreslänge $m =$ Hashtabelle der Größe m:
eine zufällige Hashfunktion $h : 1..n \to 0..m-1$ ist nur dann mit "vernünftiger" Wahrscheinlichkeit perfekt wenn $m = \Omega(n^2)$.
Riesige Platzverschwendung.



Analyse

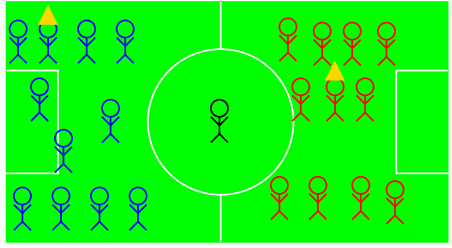
Mehr zum Geburtstagsparadoxon

pwa.klicker.uzh.ch/join/algoeng

Standardfomulierung: Ab wann lohnt es sich zu wetten, dass es zwei Gäste mi
Geburtstag gibt? Etwas komplizierter. Antwort: $n \geq 23$
Verallgemeinerung: Jahreslänge $m =$ Hashtabelle der Größe m:
eine zufällige Hashfunktion $h : 1..n \to 0..m-1$ ist nur dann mit "vernünftiger"
Wahrscheinlichkeit perfekt wenn $m = \Omega(n^2)$.
Riesige Platzverschwendung.

Frage

Reale Geburtstagstermine sind nicht gleichverteilt. Wie wirkt sich das auf die Wettchancen aus?



Analyse von Hashtabellen

Für zufällige Hash-Funktionen

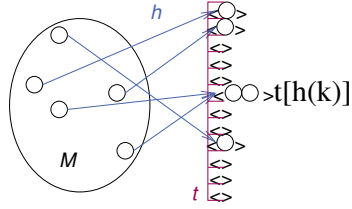
Theorem 1. $\forall k$: *die erwartete Anzahl kollidierender Elemente ist* $O(1)$ *falls* $|M| = O(m)$.

Beweis. Für festen Schlüssel k definiere Kollisionslänge X

$X := |t[h(k)]| = |\{e \in M' : h(e) = h(k)\}|$ mit

$M' = \{e \in M : \text{key}(e) \neq k\}$.

Betrachte die 0-1 ZV $X_e = 1$ für $h(e) = h(k)$, $e \in M'$ und $X_e = 0$ sonst.



$$\mathrm{E}[X] = \mathrm{E}[\sum_{e \in M'} X_e] = \sum_{e \in M'} \mathrm{E}[X_e] = \sum_{e \in M'} \mathbb{P}[X_e = 1] = \frac{|M'|}{m}$$

$$= \frac{O(m)}{m} = O(1)$$

Das gilt unabhängig von der Eingabe M. □

Analyse von Hashtabellen
Für zufällige Hash-Funktionen

Theorem: $\forall k$: die erwartete Anzahl kollidierender Elemente ist $O(1)$ falls $|M| = O(m)$.
Korollar: Alle Operationen laufen in Zeit $O(1)$

Analyse von Hashtabellen

Zufällige Hash-Funktionen?

Naive Implementierung: ein Tabelleneintrag pro Schlüssel. $\rightsquigarrow$ meist zu teuer
Weniger naive Lösungen: kompliziert, immer noch viel Platz. $\rightsquigarrow$ meist unsinnig

Zufällige Schlüssel?

$\rightsquigarrow$ unrealistisch

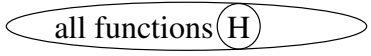
Universelles Hashing

Idee: nutze nur bestimmte “einfache” Hash-Funktionen

Definition 2. $\mathcal{H} \subseteq \{0..m-1\}^{\text{Key}}$ ist *universell*
falls für alle x, y in Key mit $x \neq y$ und zufälligem $h \in \mathcal{H}$,

$$\mathbb{P}[h(x) = h(y)] = \frac{1}{m} \ .$$

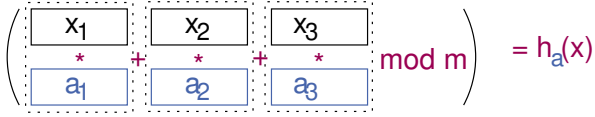
Theorem 3. *Theorem 1 gilt auch für universelle Familien von Hash-Funktionen.*
Beweis. Für $\Omega = \mathcal{H}$ haben wir immer noch $\mathbb{P}[X_e = 1] = \frac{1}{m}$. Der Rest geht wie vorher. □



Universelles Hashing
Eine einfache universelle Familie

m sei eine Primzahl, Key $\subseteq \{0,\ldots,m-1\}^k$

Theorem 4. *Für* $\mathbf{a} = (a_1,\ldots,a_k) \in \{0,\ldots,m-1\}^k$ *definiere*
$h_{\mathbf{a}}(\mathbf{x}) = \mathbf{a}\cdot\mathbf{x} \bmod m$, $H^\bullet = \left\{h_{\mathbf{a}} : \mathbf{a} \in \{0..m-1\}^k\right\}$.
$H^\bullet$ *ist eine universelle Familie von Hash-Funktionen*



Universelles Hashing
Beispiel für $H^\bullet$

Für $\mathbf{a} = (a_1, \ldots, a_k) \in \{0, \ldots, m-1\}^k$ definiere
$h_{\mathbf{a}}(\mathbf{x}) = \mathbf{a} \cdot \mathbf{x} \bmod m$, $H^\bullet = \left\{ h_{\mathbf{a}} : \mathbf{a} \in \{0..m-1\}^k \right\}$.
$k = 3$, $m = 11$
wähle $\mathbf{a} = (\mathbf{8}, \mathbf{1}, \mathbf{5})$.
$h_{\mathbf{a}}((1,1,2)) = (8,1,5) \cdot (1,1,2) = 8 \cdot 1 + 1 \cdot 1 + 5 \cdot 2 = 19 \equiv 8 \bmod 11$

Universelles Hashing

Universalität von $H^\bullet$

Beweis. Betrachte $\mathbf{x} = (x_1, \ldots, x_k)$, $\mathbf{y} = (y_1, \ldots, y_k)$ mit $x_j \neq y_j$
zähle **a**-s mit $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$.
Für jede Wahl von a_is, $i \neq j$, $\exists$ genau ein a_j mit $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$:

$$\sum_{1 \leq i \leq k} a_i x_i \equiv \sum_{1 \leq i \leq k} a_i y_i \; (\mathbf{mod}\ m)$$

$$\Leftrightarrow a_j(x_j - y_j) \equiv \sum_{i \neq j, 1 \leq i \leq k} a_i(y_i - x_i) \; (\mathbf{mod}\ m)$$

$$\Leftrightarrow a_j \equiv (x_j - y_j)^{-1} \sum_{i \neq j, 1 \leq i \leq k} a_i(y_i - x_i) \; (\mathbf{mod}\ m)$$

m^{k-1} Möglichkeiten a_i auszuwählen (mit $i \neq j$).
m^k ist die Gesamtzahl **a**s, d. h., $\mathbb{P}[h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})] = \frac{m^{k-1}}{m^k} = \frac{1}{m}$. □

Universelles Hashing

Universalität von $H^\bullet$

Beweis. Betrachte $\mathbf{x} = (x_1, \ldots, x_k)$, $\mathbf{y} = (y_1, \ldots, y_k)$ mit $x_j \neq y_j$
zähle $\mathbf{a}$-s mit $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$.
Für jede Wahl von a_is, $i \neq j$, $\exists$ genau ein a_j mit $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$:

$$\sum_{1 \leq i \leq k} a_i x_i \equiv \sum_{1 \leq i \leq k} a_i y_i \; (\mathbf{mod}\ m)$$

$$\Leftrightarrow a_j(x_j - y_j) \equiv \sum_{i \neq j, 1 \leq i \leq k} a_i(y_i - x_i) \; (\mathbf{mod}\ m)$$

$$\Leftrightarrow a_j \equiv (x_j - y_j)^{-1} \sum_{i \neq j, 1 \leq i \leq k} a_i(y_i - x_i) \; (\mathbf{mod}\ m)$$

Frage

Wieso muss m eine Primzahl sein?

m^{k-1} Möglichkeiten a_i auszuwählen (mit $i \neq j$).
m^k ist die Gesamtzahl $\mathbf{a}$s, d. h., $\mathbb{P}[h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})] = \frac{m^{k-1}}{m^k} = \frac{1}{m}$. □

Universelles Hashing

Bit-basierte Universelle Familien

Sei $m = 2^w$, Key $= \{0,1\}^k$

Bit-Matrix Multiplikation:

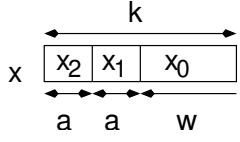
$H^{\oplus} = \left\{ h_{\mathbf{M}} : \mathbf{M} \in \{0,1\}^{w \times k} \right\}$

wobei $h_{\mathbf{M}}(\mathbf{x}) = \mathbf{M}x$ (Arithmetik mod2, d. h., xor, and)

Tabellenzugriff:

$H^{\oplus[]} = \left\{ h^{\oplus[]}_{(t_1,\ldots,t_b)} : t_i \in \{0..m-1\}^{\{0..2^a-1\}} \right\}$

wobei $h^{\oplus[]}_{(t_1,\ldots,t_b)}((x_0, x_1, \ldots, x_b)) = x_0 \oplus \bigoplus_{i=1}^{b} t_i[x_i]$

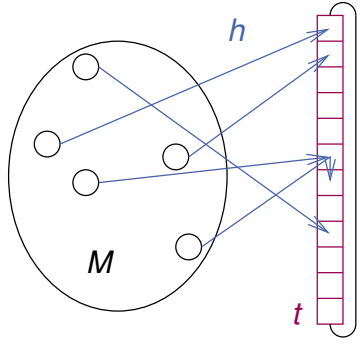


Hashing mit Linearer Suche (Linear Probing)

Zurück zur Ursprungsidee.
Elemente werden direkt in der Tabelle gespeichert.
Kollisionen werden durch Finden anderer Stellen aufgelöst.

Linear probing: Suche nächsten freien Platz.
Am Ende fange von vorn an.

- einfach
- platz-effizient
- cache-effizient



Hashing mit Linearer Suche

Der einfache Teil

Class BoundedLinearProbing($m, m' : \mathbb{N}$; $h : \text{Key} \to 0..m-1$)

t=[$\bot, \ldots, \bot$] : **Array** [$0..m+m'-1$] **of** Element

invariant $\forall i : t[i] \neq \bot \Rightarrow \forall j \in \{h(t[i])..i-1\} : t[j] \neq \bot$

Function findPos(k : Key) : $0..m+m'-1$

  for ($i := h(k)$; $t[i] \neq k \wedge t[i] \neq \bot$; $i{+}{+}$) ;

  return i

Function find(k : Key) : Element

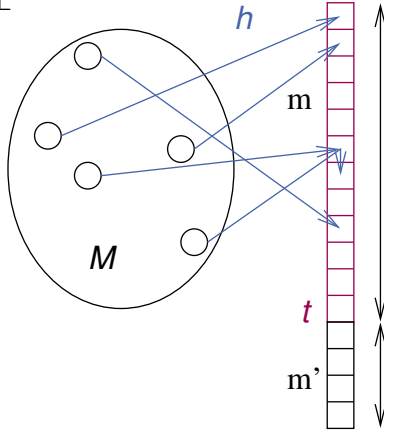
  return t[findPos(k)]

Procedure insert(e : Element)

  i:= findPos(key(e))

  assert $i < m + m' - 1$

  if $t[i] = \bot$ **then** $t[i] := e$



Hashing mit Linearer Suche

Remove

Beispiel: $t = [\ldots, \underset{h(z)}{x}, y, z, \ldots]$, remove($x$)

invariant $\forall i : t[i] \neq \bot \Rightarrow \forall j \in \{h(t[i])..i-1\} : t[j] \neq \bot$

Procedure remove(k : Key)

```
i:= findPos(key(e))                                   // where is e?
if t[i] = ⊥ then return                               // nothing to do
//we plan for a hole at i.
for (j := i + 1; t[j] ≠ ⊥; j++ )
    //Establish invariant for t[j].
    if h(t[j]) ≤ i then
        t[i] := t[j]                                  // overwrite removed element
        i := j                                        // move planned hole
t[i] := ⊥                                             // erase freed entry
```

Hashing mit Linearer Suche

Beispiel

insert : axe, chop, clip, cube, dice, fell, hack, hash, lop, slash

	an	bo	cp	dq	er	fs	gt	hu	iv	jw	kx	ly	mz
tt	0	1	2	3	4	5	6	7	8	9	10	11	12
	⊥	⊥	⊥	⊥	axe	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	⊥	axe	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	cube	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	cube	dice	⊥	⊥	⊥	⊥	⊥	⊥
	⊥	⊥	chop	clip	axe	cube	dice	⊥	⊥	⊥	⊥	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	⊥	⊥	⊥	hack	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	hash	⊥	⊥	⊥	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	hash	lop	⊥	hack	fell	⊥
	⊥	⊥	chop	clip	axe	cube	dice	hash	lop	slash	hack	fell	⊥

remove ⇩ clip

	⊥	⊥	chop	~~clip~~	axe	cube	dice	hash	lop	slash	hack	fell	⊥
	⊥	⊥	chop	lop	axe	cube	dice	hash	~~lop~~	slash	hack	fell	⊥
	⊥	⊥	chop	lop	axe	cube	dice	hash	slash	~~slash~~	hack	fell	⊥
	⊥	⊥	chop	lop	axe	cube	dice	hash	slash	⊥	hack	fell	⊥

Hashing mit Linearer Suche

Vereinfachtes Remove (?)

pwa.klicker.uzh.ch/join/algoeng

Vorschlag: Verwende spezielles “Grabstein”-Element $\underline{\cap} \neq \bot$

Überschreibe gelöschte Elemente mit $\underline{\cap}$, kein Verschieben des Loches
Suche scannt über $\underline{\cap}$ hinweg
Einfügungen dürfen $\underline{\cap}$ überschreiben

Frage

Wie schätzen Sie diese Variante des “vereinfachten remove” ein?

Verketten $\leftrightarrow$ Lineare Suche

Volllaufen: Verketten weniger empfindlich.
Unbeschränktes Hashing mit lin. Suche hat nur amortisiert konst. Einfügezeit

Cache: Lineare Suche besser. Vor allem für doall

Platz/Zeit Abwägung: Kompliziert! Abhängig von n, Füllgrad, Elementgröße,
Implementierungsdetails bei Verketten
(shared dummy!, t speichert Zeiger oder item),
Speicherverwaltung bei Verketten, beschränkt oder nicht,...

Referentielle Integrität: Nur bei Verketten!

Leistungsgarantien: Universelles Hashing funktioniert so nur mit Verketten

Hashtabellen in realen Programmiersprachen

C++: `std::unordered_set`, `std::unordered_multiset`, `std::unordered_map` verschiedene interfaces. Kaputte API erzwingt mehr oder weniger hashing mit chaining. Lieber effizientere open source Implementierungen mit anderen Interfaces verwenden, z.B: Google SparseHash oder `absl::flat_hash_map`

Rust: `HashSet`, `HashMap`

Java: `java.util.HashMap` (OpenJDK: Suchbaum statt LinkedList verhindert worst-case bei schlechter Hashfunktion), `java.util.LinkedHashMap` (Hashtabelle plus LinkedList aller Elemente)

Python: `dict`

Meistens kann man zusätzlich eine Hashfunktion wählen. Defaults sind i.allg. nicht zuverlässig. Breites Spektrum:
`std::hash<int>` ist Identität $\Leftrightarrow$ Rust DefaultHasher ist SipHash (kryptographisch)

Perfektes Hashing

hier nicht

Mehr Hashing

- Tradeoff Platz versus Zeit
- Fortgeschrittene Implementierung, Cache, SIMD,...
- Hohe Wahrscheinlichkeit, Garantien für den schlechtesten Fall, Garantien für linear probing $\rightsquigarrow$ höhere Anforderungen an die Hash-Funktionen
- $O(1)$ find im schlechtesten Fall, z.B. cuckoo hashing

Siehe auch Vorlesungen Algorithmen II und Algorithm Engineering

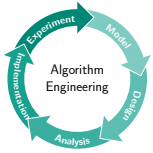
Noch Mehr Hashing

- Hashing als Mittel zur Lastverteilung
  z. B., storage servers,
  (peer to peer Netze,. . .)
- Perfekte Hashfunktionen benötigen Platz ≈ 1.44 bit pro Element.
  $z.B.$ für schnelle statische Hashtabellen
- static function retrieval: Undefiniertes Ergebnis wenn Schllüssel unbekannt.
  Platzverbrauch (fast) nur für die Werte (nicht für Schlüssel)!
- approximate membership query (AMQ) filter (aka Bloom filter):
  Verwalte Menge mit approximativer Operation “contains”.
  c bits pro Element $\rightsquigarrow$ Wahrscheinlichkeit für falsch negatives Ergebnis ist $\geq 2^{-c}$

Siehe auch Vorlesungen Algorithmen II und Algorithm Engineering

Index

Karlsruher Institut für Technologie



Algorithmen I – 5. Sortieren

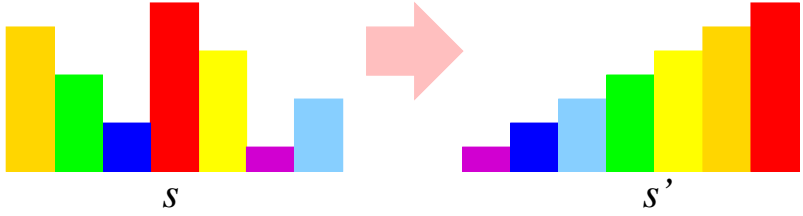
Sommersemester 2025

Peter Sanders | Stand: 30. Juli 2025

Das Sortierproblem formal

Gegeben: Elementfolge $s = \langle e_1, \ldots, e_n \rangle$
Gesucht: $s' = \langle e'_1, \ldots, e'_n \rangle$ mit

- s' ist Permutation von s
- $e'_1 \leq \cdots \leq e'_n$ für eine lineare Ordnung '$\leq$'



Anwendungsbeispiele

- Allgemein: Vorverarbeitung
- Suche: Telefonbuch $\leftrightarrow$ unsortierte Liste
- Gruppieren (Alternative Hashing?)

Beispiele aus Kurs/Buch

- Aufbau von Suchbäumen
- Kruskals MST-Algorithmus
- Verarbeitung von Intervallgraphen (z. B. Hotelbuchungen)
- Rucksackproblem
- Scheduling, die schwersten Probleme zuerst
- Sekundärspeicheralgorithmen, z. B. Datenbank-Join

Viele verwandte Probleme. Zum Beispiel Transposition dünner Matrizen, invertierten Index aufbauen, Konversion zwischen Graphrepräsentationen.

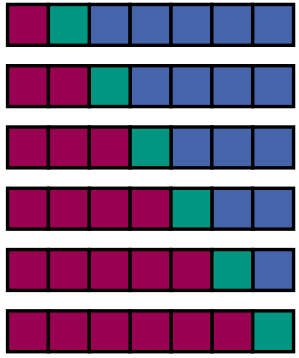
Überblick

- Einfache Algorithmen / kleine Datenmengen
- Mergesort – ein erster effizienter Algorithmus
- Eine passende untere Schranke
- Quicksort
- das Auswahlproblem
- ganzzahlige Schlüssel – jenseits der unteren Schranke

Sortieren durch Einfügen

Insertion Sort – ein einfacher Sortieralgorithmus

Procedure insertionSort(a : **Array** [1..n] **of** Element)
  for $i := 2$ **to** n **do**
    invariant $a[1] \leq \cdots \leq a[i-1]$
    move $a[i]$ to the right place



Beispiel:

$$\langle 4 \rangle, \langle 7, 1, 1 \rangle \rightsquigarrow$$
$$\langle 4, 7 \rangle, \langle 1, 1 \rangle \rightsquigarrow$$
$$\langle 1, 4, 7 \rangle, \langle 1 \rangle \rightsquigarrow$$
$$\langle 1, 1, 4, 7 \rangle, \langle 9 \rangle$$

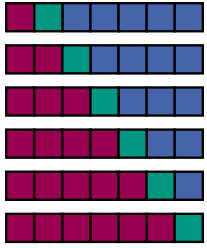
Sortieren durch Einfügen
Sentinels

```
Procedure insertionSort(a : Array [1..n] of Element)
    for i := 2 to n do
        invariant a[1] ≤ ··· ≤ a[i − 1]
        // move a[i] to the right place
        e:= a[i]
        if e < a[1] then                                    // new minimum
            for j := i downto 2 do  a[j]:= a[j − 1]
            a[1]:= e
        else                                                // use a[1] as a sentinel
            for (j := i;  a[j − 1] > e;  j−− )  a[j]:= a[j − 1]
            a[j]:= e
```

Sortieren durch Einfügen
Analyse

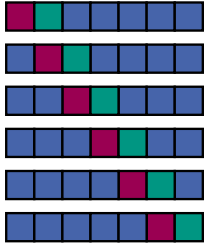
Schlechtester Fall
Die i-te Iteration braucht Zeit $\mathrm{O}(i)$.

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



Bester Fall
Die i-te Iteration braucht Zeit $\mathrm{O}(1)$ z. B. (beinahe) sortiert.

$$\sum_{i=2}^{n} \mathrm{O}(1) = \mathrm{O}(n)$$



Sortieren durch Mischen

Idee: Teile und Herrsche

Function mergeSort($\langle e_1, \ldots, e_n \rangle$) : Sequence **of** Element
  if $n = 1$ **then return** $\langle e_1 \rangle$ // base case
  else return merge(mergeSort($\langle e_1, \ldots, e_{\lfloor n/2 \rfloor} \rangle$),
    mergeSort($\langle e_{\lfloor n/2 \rfloor + 1}, \ldots, e_n \rangle$))

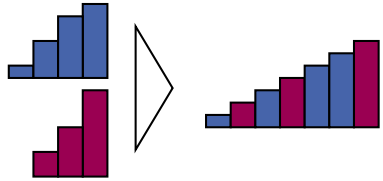
Mischen (merge)

Gegeben:

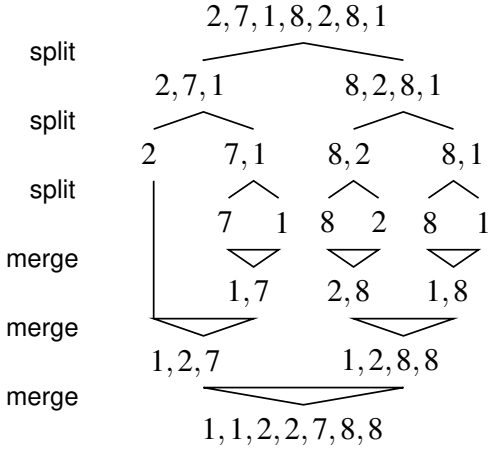
zwei sortierte Folge a und b

Berechne:

sortierte Folge der Elemente aus a und b



Beispiel

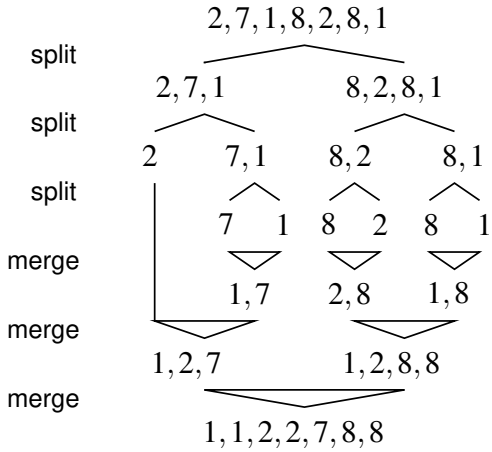

2,7,1,8,2,8,1
split
2,7,1
8,2,8,1
split
2
7,1
8,2
8,1
split
7
1
8
2
8
1
merge
1,7
2,8
1,8
merge
1,2,7
1,2,8,8
merge
1,1,2,2,7,8,8

Mischen

Jeweils $\min(a, b)$ in die Ausgabe schieben. Zeit $O(n)$

a	b	c	operation
$\langle 1,2,7\rangle$	$\langle 1,2,8,8\rangle$	$\langle\rangle$	move a
$\langle 2,7\rangle$	$\langle 1,2,8,8\rangle$	$\langle 1\rangle$	move b
$\langle 2,7\rangle$	$\langle 2,8,8\rangle$	$\langle 1,1\rangle$	move a
$\langle 7\rangle$	$\langle 2,8,8\rangle$	$\langle 1,1,2\rangle$	move b
$\langle 7\rangle$	$\langle 8,8\rangle$	$\langle 1,1,2,2\rangle$	move a
$\langle\rangle$	$\langle 8,8\rangle$	$\langle 1,1,2,2,7\rangle$	concat b
$\langle\rangle$	$\langle\rangle$	$\langle 1,1,2,2,7,8,8\rangle$	

Analyse



Analyse: $T(n) = O(n) + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) = O(n \log n)$.

Sortieren durch Mischen
Analyse

$$T(n) = \mathrm{O}(n) + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$$

Problem: Runderei
Ausweg: genauer rechnen (siehe Buch)
Dirty trick:
Eingabe auf Zweierpotenz aufblasen
(z. B. $(2^{\lceil \log n \rceil} - n) \times \infty$ anhängen)
$\rightsquigarrow$
normales Master-Theorem anwendbar
Zeit $\mathrm{O}(n \log n)$

Untere Schranken

Geht es schneller als $\Theta(n \log n)$?

Unmöglichkeit einer Verbesserung i.allg. schwer zu beweisen –
sie erfordert eine Aussage über alle denkbaren Algorithmen.

$\rightsquigarrow$
einschränkende Annahmen

Eine vergleichsbasierte untere Schranke

Vergleichsbasiertes Sortieren:
Informationen über Elemente nur durch Zwei-Wege-Vergleich $e_i \leq e_j$?.

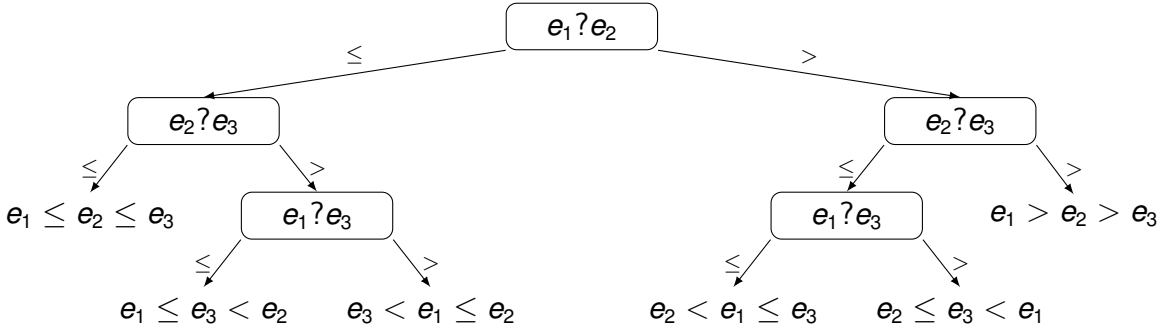
Satz. *Deterministische vergleichsbasierte Sortieralgorithmen brauchen*

$$n \log n - \mathrm{O}(n)$$

Vergleiche im schlechtesten Fall.

Beweis. Betrachte Eingaben, die Permutationen von $1..n$ sind.
Es gibt genau $n!$ solche Permutationen. □

Baumbasierte Sortierer-Darstellung



Mindestens ein Blatt pro Permutation von $e_1, \ldots, e_n$
Ausführungszeit entspricht Tiefe T

Beweis

Baum der Tiefe T hat höchstens 2^T Blätter.

$\Rightarrow 2^T \geq n!$

$\Leftrightarrow T \geq \log \underbrace{n!}_{\geq \left(\frac{n}{e}\right)^n} \geq \log \left(\frac{n}{e}\right)^n = n \log n - n \log e = n \log n - \mathrm{O}(n)$

Einfache Approximation der Fakultät: $\left(\frac{n}{e}\right)^n \leq n! \leq n^n$

Beweis für linken Teil:

$$\ln n! = \sum_{2 \leq i \leq n} \ln i \geq \int_1^n \ln x \, dx = \Big[x(\ln x - 1)\Big]_{x=1}^{x=n} \geq n(\ln n - 1) \ .$$

$$\Rightarrow n! \geq e^{n(\ln n - 1)} = \frac{e^{n \ln n}}{e^n} = \frac{n^n}{e^n} = \left(\frac{n}{e}\right)^n$$

Randomisierung, Mittlere Ausführungszeit

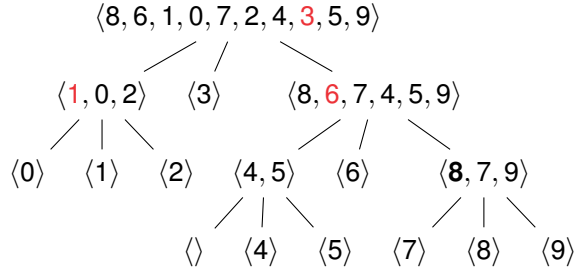
Satz. *Immer noch* $n \log n - \mathrm{O}(n)$ *Vergleiche.*

Beweis. nicht hier. □

Quicksort
Erster Versuch

Idee: Teile-und-Herrsche aber verglichen mit mergesort „andersrum“.
Leiste Arbeit vor rekursivem Aufruf

Function quickSort(s : Sequence **of** Element) : Sequence **of** Element
  if $|s| \leq 1$ **then return** s
  pick “some” $p \in s$
  $a := \langle e \in s : e < p \rangle$
  $b := \langle e \in s : e = p \rangle$
  $c := \langle e \in s : e > p \rangle$
  return concatenation of
    quickSort(a), b, and
    quickSort(c)



Quicksort
Analyse im schlechtesten Fall

Annahme: Pivot ist immer Minimum (oder Max.) der Eingabe

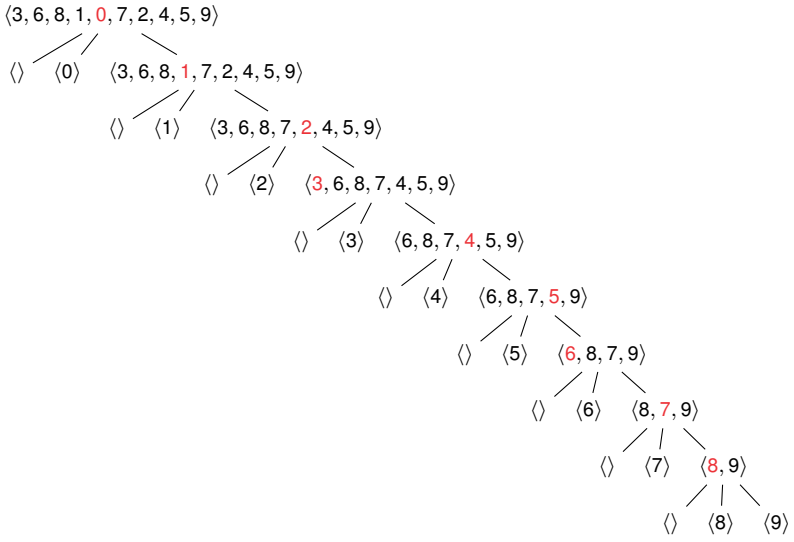
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ \Theta(n) + T(n-1) & \text{if } n \geq 2. \end{cases}$$

$\Rightarrow$

$T(n) = \Theta(n + (n-1) + \cdots + 1) = \Theta(n^2)$

Quicksort

Beispiel: Schlechtester Fall

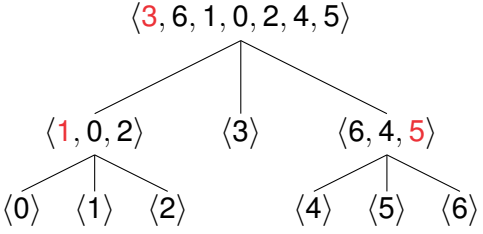


Quicksort
Analyse im besten Fall

Annahme: Pivot ist immer Median der Eingabe

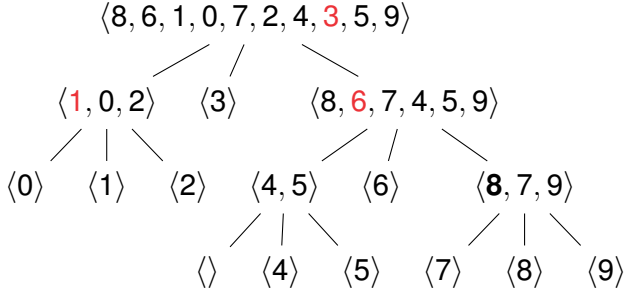
$$T(n) \leq \begin{cases} O(1) & \text{if } n = 1, \\ O(n) + 2T(\lfloor n/2 \rfloor) & \text{if } n \geq 2. \end{cases}$$

$\Rightarrow$ (Master-Theorem)
$T(n) = O(n \log n)$
Problem: Median bestimmen ist nicht so einfach



Quicksort
Zufälliger Pivot

Function quickSort(s : Sequence **of** Element) : Sequence **of** Element
  if $|s| \leq 1$ **then return** s
  pick $p \in s$ uniformly at random
  $a := \langle e \in s : e < p \rangle$
  $b := \langle e \in s : e = p \rangle$
  $c := \langle e \in s : e > p \rangle$
  return concatenation of
    quickSort(a), b, and quickSort(c)



Quicksort
Erwartete Laufzeit

Satz. *Quicksort hat erwartete Laufzeit* $O(n \log n)$

Annahme: alle Elemente verschieden — Warum 'OBdA'?
Es genügt, die 3-Wege Vergleiche $(<, =, >)$ $C(n)$ zu zählen.
Genauer: wir bestimmen $\bar{C}(n) = \mathrm{E}[C(n)]$

```
Function quickSort(s : Sequence of Element) : Sequence of Element
    if |s| ≤ 1 then return s
    pick p ∈ s uniformly at random
    a:= ⟨e ∈ s : e < p⟩                    // |s|
    b:= ⟨e ∈ s : e = p⟩                    // 3-Wege
    c:= ⟨e ∈ s : e > p⟩                    // Vergleiche
    return concatenation of quickSort(a), b, and quickSort(c)
```

Beweisansatz 1: Rekurrenzen

Idee: Im Buch wird bewiesen, dass mit Wahrscheinlichkeit $1/2$ das Aufspaltverhältnis nicht schlechter als $\frac{1}{4} : \frac{3}{4}$ ist.

Das genügt um $\bar{C}(n) = \mathrm{O}(n \log n)$ zu zeigen.

Beweisansatz 2: Genauere, elegantere Analyse

Satz. $\bar{C}(n) \leq 2n \ln n \leq 1.45 n \log n$

Satz: $\bar{C}(n) \leq 2n \ln n \leq 1.45 n \log n$

Sei $s' = \langle e'_1, \ldots, e'_n \rangle$ sortierte Eingabefolge.

Indikatorzufallsvariable: $X_{ij} := 1$ gdw. e'_i wird mit e'_j verglichen.

$$\bar{C}(n) = \mathrm{E}\left[\sum_{i=1}^{n}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n}\sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] = \sum_{i=1}^{n}\sum_{j=i+1}^{n} \mathbb{P}[X_{ij} = 1] \ .$$

Lemma: $\mathbb{P}[X_{ij} = 1] = \frac{2}{j-i+1}$

Sortierte Eingabefolge:

$s' = \langle e'_1, \ldots, e'_{i-1}, \underbrace{e'_i, e'_{i+1}, \ldots, e'_{j-1}, e'_j}_{j-i+1 \text{ Elemente}}, e'_{j+1}, \ldots, e'_n \rangle$

$X_{ij} = 1$

$\Leftrightarrow$

e'_i wird mit e'_j verglichen

$\Leftrightarrow$

e'_i oder e'_j wird Pivot bevor ein Pivot aus $\langle e'_{i+1}, \ldots, e'_{j-1} \rangle$ gewählt wird.

$\Rightarrow$

$\mathbb{P}[X_{ij} = 1] = \frac{2}{j-i+1}$ $\square$

Satz: $\bar{C}(n) \leq 2n \ln n \leq 1.45 n \log n$

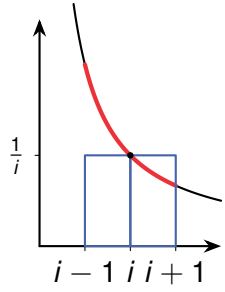
$$\begin{aligned}
\bar{C}(n) &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \\
&\leq \sum_{i=1}^{n} \sum_{k=2}^{n} \frac{2}{k} \\
&= 2n \sum_{k=2}^{n} \frac{1}{k} \text{(harmonische Summe)} \\
&= 2n(H_n - 1) \leq 2n(1 + \ln n - 1) = 2n \ln n \ .
\end{aligned}$$

i	j	$\overbrace{j-i+1}^{=:k}$
1	$2..n$	$2..n$
2	$3..n$	$2..n-1$
3	$4..n$	$2..n-2$
$\vdots$	$\vdots$	$\vdots$
$n-1$	$n..n$	$2..2$
n	$\emptyset$	$\emptyset$

□

Exkurs: Harmonische Summe

$$\int_i^{i+1} \frac{1}{x}\,dx \le \frac{1}{i} \le \int_{i-1}^{i} \frac{1}{x}\,dx$$



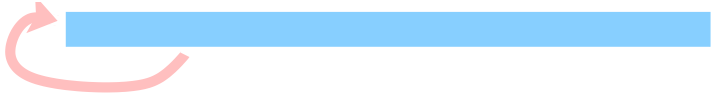
Also

$$\ln n = \int_1^n \frac{1}{x}\,dx = \sum_{i=1}^{n-1} \int_i^{i+1} \frac{1}{x}\,dx \le \sum_{i=1}^{n-1} \frac{1}{i} \le \sum_{i=1}^{n} \frac{1}{i} = 1 + \sum_{i=2}^{n} \frac{1}{i}$$

$$\le 1 + \sum_{i=2}^{n} \int_{i-1}^{i} \frac{1}{x} = 1 + \int_1^n \frac{1}{x}\,dx = 1 + \ln n$$

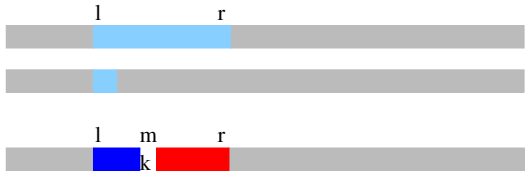
Quicksort
Effiziente Implementierung

- Array-Implementierung
- „in-place“
- 2-Wegevergleiche



Quicksort
Effiziente Implementierung

Procedure qSort(a : **Array of** Element; $\ell, r : \mathbb{N}$)
  if $\ell \geq r$ **then return**
  k:= pickPivotPos(a, ℓ, r)
  m:= partition(a, ℓ, r, k)
  qSort($a, \ell, m-1$)
  qSort($a, m+1, r$)



Quicksort

Effiziente Implementierung

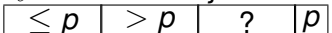
Function partition(a : **Array of** Element; $\ell, r, k : \mathbb{N}$)

$p := a[k]$ // pivot

swap($a[k], a[r]$)

$i := \ell$

invariant



for $j := \ell$ **to** $r - 1$ **do**

  if $a[j] \leq p$ **then**

    swap($a[i], a[j]$)

    i++

assert

swap($a[i], a[r]$)

assert

return i

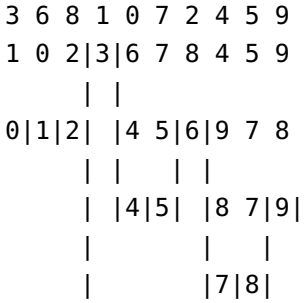
Quicksort

Beispiel: Partitionierung, $k = 1$

Notation: p, $\overline{i}$, $\underline{j}$

3	6	8	1	0	7	2	4	5	9
$\overline{\underline{9}}$	6	8	1	0	7	2	4	5	3
$\overline{9}$	$\underline{6}$	8	1	0	7	2	4	5	3
$\overline{9}$	6	$\underline{8}$	1	0	7	2	4	5	3
$\overline{9}$	6	8	$\underline{1}$	0	7	2	4	5	3
1	$\overline{6}$	8	9	$\underline{0}$	7	2	4	5	3
1	0	$\overline{8}$	9	6	$\underline{7}$	2	4	5	3
1	0	$\overline{8}$	9	6	7	$\underline{2}$	4	5	3
1	0	2	$\overline{9}$	6	7	8	$\underline{4}$	5	3
1	0	2	$\overline{9}$	6	7	8	4	$\underline{5}$	3
1	0	2	$\overline{9}$	6	7	8	4	5	3
1	0	2	$\overline{3}$	6	7	8	4	5	9

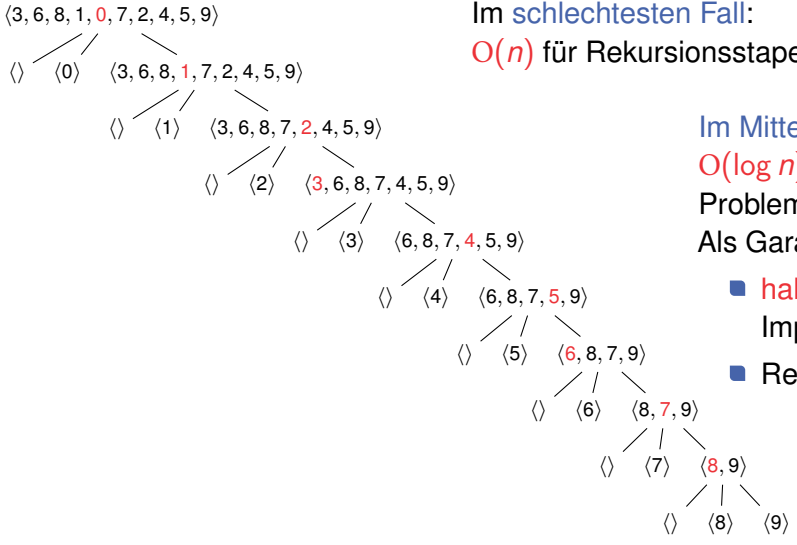
Quicksort
Beispiel: Rekursion



Quicksort
Größerer Basisfall

Procedure qSort(a : **Array of** Element; $\ell, r : \mathbb{N}$)
  if $r - \ell + 1 \leq n_0$ **then** insertionSort($a[\ell..r]$)
  k:= pickPivotPos(a, ℓ, r)
  m:= partition(a, ℓ, r, k)
  qSort($a, \ell, m-1$)
  qSort($a, m+1, r$)

Wirklich Inplace?



Im schlechtesten Fall:
$O(n)$ für Rekursionsstapel.

Im Mittel:
$O(\log n)$ zusätzlicher Platz – kein Problem.
Als Garantie für schlechtesten Fall:

- halbrekursive Implementierung
- Rekursion auf kleinere Hälfte

Quicksort
Halbrekursive Implementierung

Procedure qSort(a : **Array of** Element; $\ell, r : \mathbb{N}$)
  while $r - \ell + 1 > n_0$ **do**
    k:= pickPivotPos(a, ℓ, r)
    m:= partition(a, ℓ, r, k)
    if $m < (\ell + r)/2$ **then** qSort($a, \ell, m - 1$); ℓ:= $m + 1$
    else qSort($a, m + 1, r$); r:= $m - 1$
  insertionSort($a[\ell..r]$)

Procedure qSort(a : **Array of** Element; $\ell, r : \mathbb{N}$)
  while $r - \ell + 1 > n_0$ **do**
    k:= pickPivotPos(a, ℓ, r)
    m:= partition(a, ℓ, r, k)
    if $m < (\ell + r)/2$ **then** qSort($a, \ell, m - 1$); ℓ:= $m + 1$
    else qSort($a, m + 1, r$); r:= $m - 1$
  insertionSort($a[\ell..r]$)

Satz. *Rekursionstiefe* $\leq \left\lceil \log \frac{n}{n_0} \right\rceil$

Beweis. Induktion. Teilproblemgröße halbiert sich (mindestens) mit jedem rekursiven Aufruf □

Quicksort

Quadratische Komplexität bei gleichen Elementen?

- Variante aus dem Buch verwenden
- oder doch Drei-Wege-Partitionierung

Quicksort
Drei-Wege-Partitionierung

Procedure qSortTernary(a : **Array of** Element; $\ell, r : \mathbb{N}$)
    if $\ell \geq r$ **then return**
    p:= *key*(a[pickPivotPos(a, ℓ, r)])
    (m, m'):= partitionTernary(a, ℓ, r, p)
    qSortTernary($a, \ell, m-1$)
    qSortTernary($a, m'+1, r$)

Quicksort

Drei-Wege-Partitionierung

Function partitionTernary(a : **Array of** Element; $\ell, r : \mathbb{N}$; p : *Key*)

$i := \ell, \quad j := \ell, \quad k := r$

invariant

ℓ	i	j	k	r
$< p$	$> p$	?		$= p$

while $(j \leq k)$

if $a[j] = p$ **then** swap$(a[j], a[k])$, $k{-}{-}$;

else if $a[j] < p$ **then** swap$(a[j], a[i])$, $i{+}{+}$, $j{+}{+}$;

else $j{+}{+}$;

assert

ℓ	i	k	r
$< p$	$> p$		$= p$

$i' := i + r - k + 1$

swap$(a[i..i'], a[k+1..r])$

assert

ℓ	i	i'	r
$< p$	$= p$	$> p$	

return (i, i')

Vergleich Quicksort $\leftrightarrow$ Mergesort

Pro Mergesort

- $O(n \log n)$ Zeit (deterministisch)

  qsort: $\exists$ det. Varianten
- $n \log n + O(n)$ Elementvergleiche ($\approx$ untere Schranke)

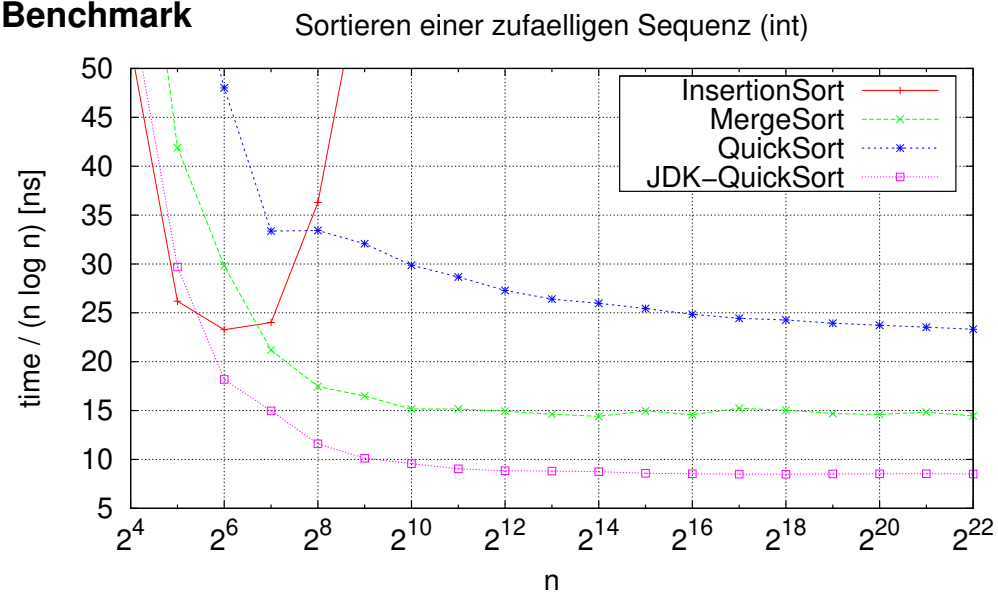
  qsort: möglich bei sorgfältiger Pivotwahl
- Stabil (gleiche Elemente behalten Reihenfolge bei)

  qsort: leicht bei Aufgabe der inplace-Eigenschaft

Pro Quicksort

- inplace
- Etwas schneller?

Benchmark


Sortieren einer zufaelligen Sequenz (int)
time / (n log n) [ns]
n
InsertionSort
MergeSort
QuickSort
JDK−QuickSort
50
45
40
35
30
25
20
15
10
5
2^4
2^6
2^8
2^{10}
2^{12}
2^{14}
2^{16}
2^{18}
2^{20}
2^{22}

Auswahl (Selection)

Definition: Rang eines Elements e einer Folge $s =$ Position von e in sort(s) (angefangen bei 1).

Frage: warum ist r nicht notwendig eindeutig?

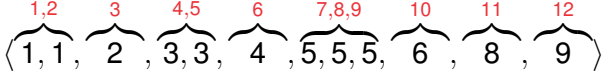
// return an element of s with rank k
Function select(s : Sequence **of** Element; k : $\mathbb{N}$) : Element
    assert $|s| \geq k$

Vorsicht: Es gibt verschiedene Definitionen von “Rang”

Auswahl
Beispiel

$\langle 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8 \rangle$
$\rightsquigarrow$ sortieren
$\langle 1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 8, 9 \rangle$

mögliche Ränge:



Auswahl
Anwendungen

Karlsruher Institut für Technologie

Statistik

- Spezialfall Medianauswahl: $k = \lceil |s|/2 \rceil$
- allgemeinere Quantile (10 % ,...)

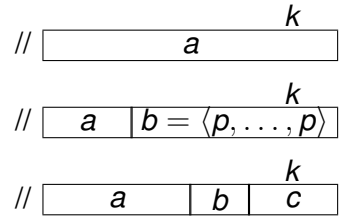
Unterprogramm

Eingabe eingrenzen auf vielversprechendste Elemente,
z. B. top-k Attention in Transformer-Netzwerken.

Quickselect

$\approx$ quicksort mit einseitiger Rekursion

Function select(s : Sequence **of** Element; k : $\mathbb{N}$) : Element
- **assert** $|s| \geq k$
- pick $p \in s$ uniformly at random // pivot key
- a := $\langle e \in s : e < p \rangle$
- **if** $|a| \geq k$ **then return** select(a, k)
- b := $\langle e \in s : e = p \rangle$
- **if** $|a| + |b| \geq k$ **then return** p
- c := $\langle e \in s : e > p \rangle$
- **return** select$(c, k - |a| - |b|)$

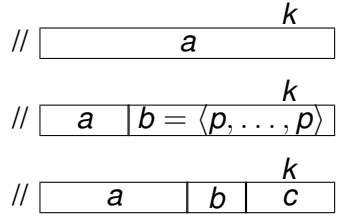


Quickselect
Beispiel

s	k	p	a	b	c
$\langle 3,1,4,5,9,\mathbf{2},6,5,3,5,8\rangle$	6	2	$\langle 1\rangle$	$\langle 2\rangle$	$\langle 3,4,5,9,6,5,3,5,8\rangle$
$\langle 3,4,5,9,\mathbf{6},5,3,5,8\rangle$	4	6	$\langle 3,4,5,5,3,5\rangle$	$\langle 6\rangle$	$\langle 9,8\rangle$
$\langle 3,4,\mathbf{5},5,3,5\rangle$	4	5	$\langle 3,4,3\rangle$	$\langle 5,5,5\rangle$	$\langle\rangle$

Quickselect
Analyse

Function select(s : Sequence **of** Element; k : $\mathbb{N}$) : Element
- **assert** $|s| \geq k$
- pick $p \in s$ uniformly at random // pivot key
- a := $\langle e \in s : e < p \rangle$
- **if** $|a| \geq k$ **then return** select(a, k)
- b := $\langle e \in s : e = p \rangle$
- **if** $|a| + |b| \geq k$ **then return** p
- c := $\langle e \in s : e > p \rangle$
- **return** select$(c, k - |a| - |b|)$

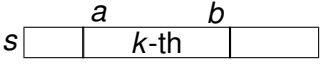


Satz. *Quickselect hat erwartete Ausführungszeit* $O(|s|)$

Beweis. hier nicht □

Mehr zum Auswahlproblem

- Tuning (array, inplace, 2-Wege-Vergleiche, iterativ)
  analog quicksort
- Deterministische Auswahl: quickselect mit spezieller det. Pivotwahl
- k Elemente mit Rang $\leq k$
  ggf. sortiert.
- Weitere Verallgemeinerungen:
  mehrere Ränge, teilweise sortierte Eingaben,...
  Beispiel: Optimale Range Median Berechnung
  [B. Gfeller, P. Sanders, ICALP 2009].
  Vorberechnungszeit $O(n \log n)$, Zeit $O(\log n)$ für
  select$(\langle s[a], \ldots, s[b] \rangle, k)$

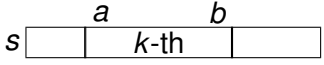


Mehr zum Auswahlproblem

- Tuning (array, inplace, 2-Wege-Vergleiche, iterativ)
  analog quicksort
- Deterministische Auswahl: quickselect mit spezieller det. Pivotwahl
- k Elemente mit Rang $\leq k$
  ggf. sortiert.
- Weitere Verallgemeinerungen:
  mehrere Ränge, teilweise sortierte Eingaben,...
  Beispiel: Optimale Range Median Berechnung
  [B. Gfeller, P. Sanders, ICALP 2009].
  Vorberechnungszeit $O(n \log n)$, Zeit $O(\log n)$ für
  select$(\langle s[a], \ldots, s[b] \rangle, k)$

Frage

Gegeben eine Folge s, wie findet man effizient die k kleinsten Elemente?



Durchbrechen der unteren Schranke – Ganzzahliges Sortieren

Untere Schranke $=$ schlechte Nachricht?
Nein: u.U. Hinweis, welche Annahmen man in Frage stellen muss.

Beim Sortieren:
Mehr mit den Schlüsseln machen als nur Vergleichen.

K Schlüssel – Eimer-Sortieren (bucket sort)

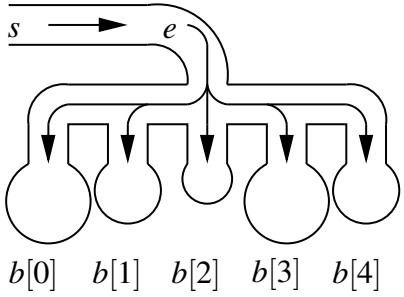
Procedure KSort(s : Sequence **of** Element) // n elements

b=$\langle\langle\rangle, \ldots, \langle\rangle\rangle$: **Array** $[0..K-1]$ **of** Sequence **of** Element

foreach $e \in s$ **do** $b[\text{key}(e)]$.pushBack(e)

$s :=$ concatenation of $b[0], \ldots, b[K-1]$

Zeit: $O(n+K)$



Beispiel: $K = 4$

Procedure KSort(s : Sequence **of** Element)
b=$\langle\langle\rangle, \ldots, \langle\rangle\rangle$: **Array** $[0..K-1]$ **of** Sequence **of** Element
foreach $e \in s$ **do** $b[\text{key}(e)]$.pushBack(e)
$s :=$ concatenation of $b[0], \ldots, b[K-1]$

$s = \langle(3,a),(1,b),(2,c),(3,d),(0,e),(0,f),(3,g),(2,h),(1,i)\rangle$

verteilen$\rightsquigarrow$
$b =$ | $\langle(0,e),(0,f)\rangle$ | $\langle(1,b),(1,i)\rangle$ | $\langle(2,c),(2,h)\rangle$ | $\langle(3,a),(3,d),(3,g)\rangle$ |

aneinanderhängen$\rightsquigarrow$
$s = \langle(0,e),(0,f),(1,b),(1,i),(2,c),(2,h),(3,a),(3,d),(3,g)\rangle$.

Array-Implementierung

Procedure KSortArray(a,b : **Array** $[1..n]$ **of** Element)

$c = \langle 0, \ldots, 0 \rangle$: **Array** $[0..K-1]$ **of** $\mathbb{N}$

for $i := 1$ **to** n **do** $c[key(a[i])]{+}{+}$

$C := 1$

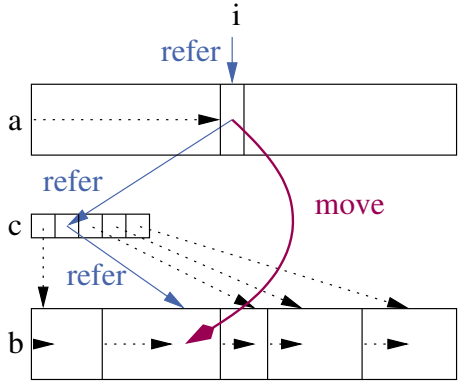
for $k := 0$ **to** $K-1$ **do**

$$\begin{pmatrix} C \\ c[k] \end{pmatrix} := \begin{pmatrix} C + c[k] \\ C \end{pmatrix}$$

for $i := 1$ **to** n **do**

$b[c[key(a[i])]] := a[i]$

$c[key(a[i])]{+}{+}$



Beispiel: $a = [3,1,2,3,0,0,3,2,1]$, $K = 4$

Procedure KSortArray(a,b : **Array** $[1..n]$ **of** Element)

$c = \langle 0, \ldots, 0 \rangle$: **Array** $[0..K-1]$ **of** $\mathbb{N}$

for $i := 1$ **to** n **do** $c[key(a[i])]{+}{+}$ // $c := [2,2,2,3]$

$C := 1$

for $k := 0$ **to** $K-1$ **do**

$$\begin{pmatrix} C \\ c[k] \end{pmatrix} := \begin{pmatrix} C + c[k] \\ C \end{pmatrix}$$ // $c := [1,3,5,7]$

for $i := 1$ **to** n **do**

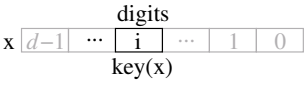
$b[c[key(a[i])]] := a[i]$ // $b := [0,0,1,1,2,2,3,3,3]$

$c[key(a[i])]{+}{+}$ // bei $i = [5,6,2,9,3,8,1,4,7]$

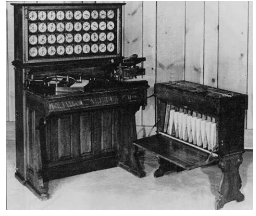
K^d Schlüssel – Least-Significant-Digit Radix-Sortieren

Beobachtung: KSort ist stabil, d. h.,
Elemente mit gleichem Schlüssel behalten ihre relative Reihenfolge.

Procedure LSDRadixSort(s : Sequence **of** Element)
for $i := 0$ **to** $d - 1$ **do**
redefine key(x) as $(x$ **div** $K^i)$ **mod** K
KSort(s)
invariant
s is sorted with respect to digits $i..0$

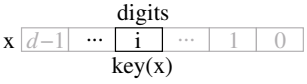


Zeit: $O(d(n + K))$



LSD Radix-Sortieren Beispiel

Procedure LSDRadixSort(s : Sequence **of** Element)
for $i := 0$ **to** $d - 1$ **do**
redefine key(x) as $(x$ **div** $K^i)$ **mod** K
KSort(s)
invariant
s is sorted with respect to digits $i..0$



$K = 10$, $d = 2$
$s = \langle 42, 17, 88, 12, 96, 47, 89 \rangle$
sortiere nach 0-ter Ziffer:
$s = \langle 42, 12, 96, 17, 47, 88, 89 \rangle$
sortiere stabil nach 1-ter Ziffer:
$s = \langle 12, 17, 42, 47, 88, 89, 96 \rangle$

Mehr zu ganzzahligem Sortieren

- MSD-Radix-Sort: Wichtigste Ziffer zuerst.
  im Mittel Cache-effizienter aber Probleme mit schlechtestem Fall
- Kleineres K kann besser sein. (Cache-Misses, TLB-Misses)
- Inplace: z.B. *Engineering in-place (shared-memory) sorting algorithms*. M. Axtmann, S. Witt, D. Ferizovic, P. Sanders. ACM TOPC 9 (1), 1–62
- “Machine Learning” Ansatz:
  Mittelding zwischen Sortieren durch Mehrwegeverteilen (siehe später) und MSD-Radix-Sort.
  Lerne Vertelung der Schlüssel (z.B. stückweise linear)

Sortieren: vergleichsbasiert $\leftrightarrow$ ganzzahlig

pro ganzzahlig:

- asymptotisch schneller

pro vergleichsbasiert

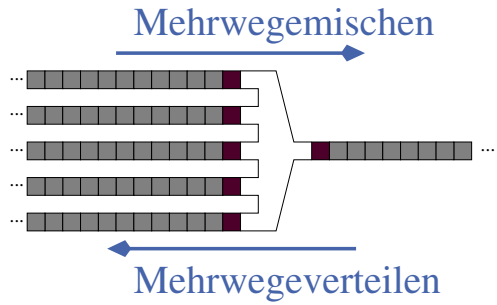
- weniger Annahmen
  (z. B. wichtig für Algorithmenbibliotheken)
- robust gegen beliebige Eingabeverteilungen
- Cache-Effizienz weniger schwierig

Mehr zum Sortieren

- Verfügbar in Algorithmenbibliotheken
- (binary) mergesort $\rightsquigarrow$ Mehrwegemischen
- quicksort $\rightsquigarrow$ Sortieren durch Mehrwegeverteilen

$\rightsquigarrow$ Parallel

$\rightsquigarrow$ Extern: oft noch wichtiger als intern

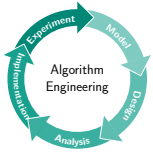


Verallgemeinerungen:

- Prioritätslisten (kommen als nächstes)
- Dynamische sortierte Listen (als übernächstes)

Index

Karlsruher Institut für Technologie



Algorithmen I – 6. Prioritätslisten

Sommersemester 2025

Peter Sanders | Stand: 30. Juli 2025

Prioritätslisten
(priority queues)

Verwalte Menge M von Elementen mit Schlüsseln

Insert(e): $M := M \cup e$

DeleteMin: return and remove min M

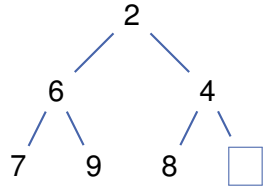
Prioritätslisten
Anwendungen

(ohne zusätzliche Operationen)

- Mehrwegemischen (klein)
- Greedy Algorithmen (z. B., Scheduling) (klein–mittel)
- Simulation diskreter Ereignisse (mittel–groß)
- Branch-and-Bound Suche (groß)
- run formation für externes Sortieren (groß)
- Time forward processing (riesig)

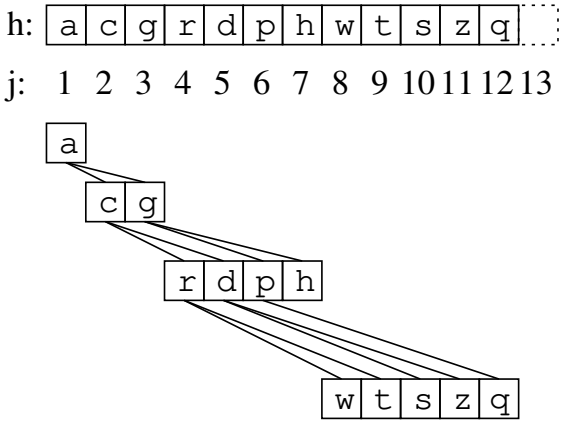
Binäre Heaps

Heap-Eigenschaft: Bäume (oder Wälder) mit $\forall v$: parent$(v) \leq v$
Binärer Heap: Binärbaum, Höhe $\lfloor \log n \rfloor$, fehlende Blätter rechts unten.



Beobachtung: Minimum $=$ Wurzel
Idee: Änderungen nur entlang eines Pfades Wurzel–Blatt
$\rightsquigarrow$
insert, deleteMin brauchen Zeit $O(\log n)$

Binäre Heaps
Implizite Baum-Repräsentation

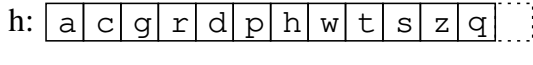


- Array $h[1..n]$
- Schicht für Schicht
- parent(j) $= \lfloor j/2 \rfloor$
- linkes Kind(j): $2j$
- rechtes Kind(j): $2j + 1$

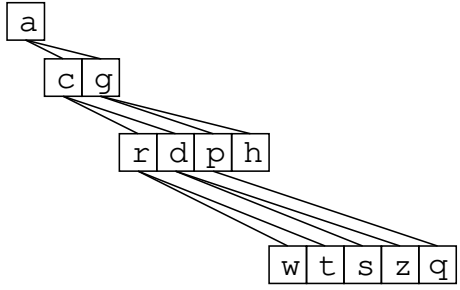
Nicht nur nützlich für heaps:
z. B. Turnierbäume, statische Suchbäume

Binäre Heaps
Pseudocode

(beschränkte PQ)



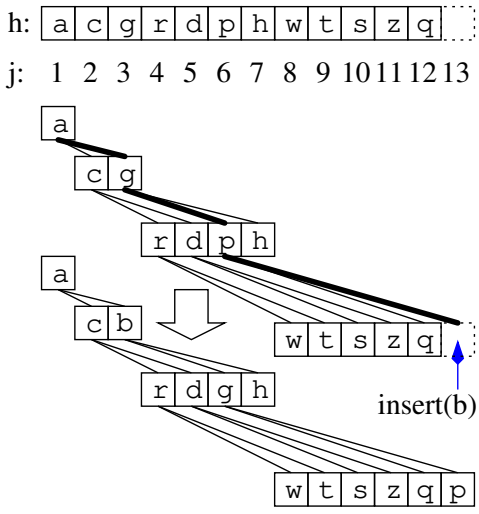
Class BinaryHeapPQ($w : \mathbb{N}$) **of** Element
  h : **Array** $[1..w]$ **of** Element
  n=0 : $\mathbb{N}$
  invariant $\forall j \in 2..n : h[\lfloor j/2 \rfloor] \leq h[j]$
  Function min **assert** $n > 0$; **return** $h[1]$



Binäre Heaps
Pseudocode – Einfügen

Procedure insert(e : *Element*)
  assert $n < w$
  $n{+}{+}$; $h[n] := e$
  siftUp(n)

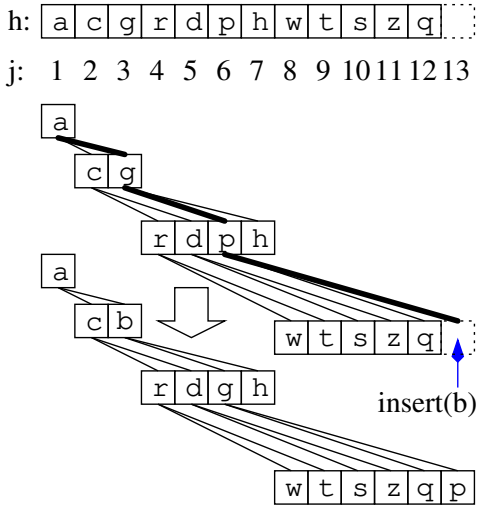
Procedure siftUp($i : \mathbb{N}$)
  assert the heap property holds
    except maybe at position i
  if $i = 1 \vee h[\lfloor i/2 \rfloor] \leq h[i]$ **then return**
  swap($h[i], h[\lfloor i/2 \rfloor]$)
  siftUp($\lfloor i/2 \rfloor$)



Binäre Heaps
Nichtrekursives Einfügen

Procedure insert(e : *Element*)
  assert $n < w$
  $n{+}{+}$
  $i := n$
  while $i > 1 \wedge e < h[\lfloor i/2 \rfloor]$ **do**
    $h[i] := h[\lfloor i/2 \rfloor]$
    $i := \lfloor i/2 \rfloor$
  $h[i] := e$

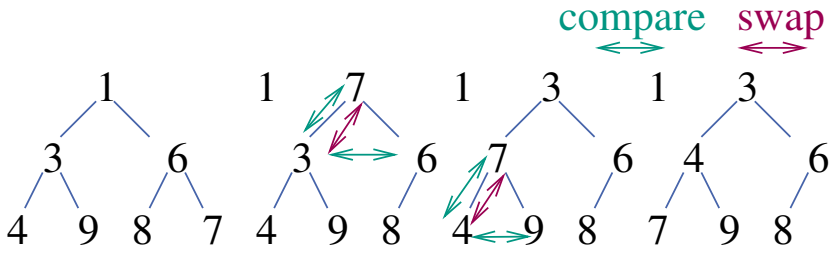
Schneller, weil weniger Speicherzugriffe



Binäre Heaps

Pseudocode – deleteMin

```
Function deleteMin : Element
    result=h[1] : Element
    h[1]:= h[n];   n−−
    siftDown(1)
    return result
```

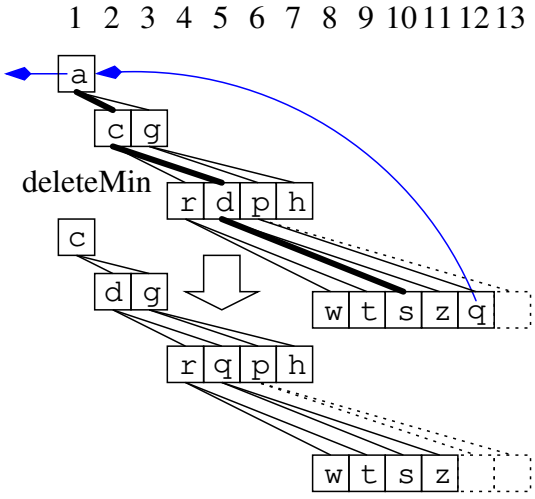


Binäre Heaps

Pseudocode – deleteMin

```
Function deleteMin : Element
    result=h[1] : Element
    h[1]:= h[n]; n−−
    siftDown(1)
    return result
Procedure siftDown(i : ℕ)
    assert heap property except, possibly at j = 2i and j = 2i + 1
    if 2i ≤ n then                                          // i is not a leaf
        if 2i + 1 > n ∨ h[2i] ≤ h[2i + 1] then m:= 2i else m:= 2i + 1
        assert ∄sibling(m) ∨ h[sibling(m)] ≥ h[m]
        if h[i] > h[m] then                                 // heap property violated
            swap(h[i], h[m])
            siftDown(m)
    assert the heap property holds for the subtree rooted at i
```

Binäre Heaps
deleteMin: Beispiel



Binäre Heaps
Analyse

Satz. min *dauert* $O(1)$. □

Lemma. *Höhe ist* $\lfloor \log n \rfloor$ □

Satz. insert *dauert* $O(\log n)$.

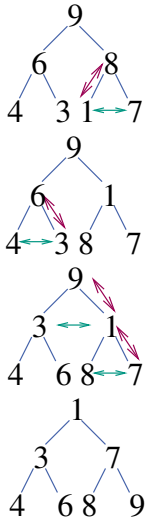
Satz. deleteMin *dauert* $O(\log n)$.

Beweis. Zeit $O(1)$ pro Schicht. □

Binäre Heaps
Konstruktion

Procedure buildHeapBackwards
  for $i := \lfloor n/2 \rfloor$ **downto** 1 **do** siftDown(i)

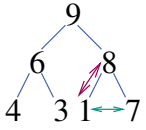
compare swap



Binäre Heaps
Konstruktion

compare swap

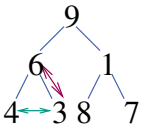
Procedure buildHeapBackwards
  for $i := \lfloor n/2 \rfloor$ **downto** 1 **do** siftDown(i)



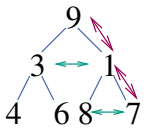
Satz. buildHeap *läuft in Zeit* $O(n)$

Beweis. Sei $k = \lfloor \log n \rfloor$. In Tiefe $\ell \in 0..\lfloor \log n \rfloor$:

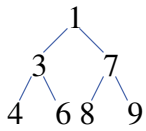
- 2^ℓ Aufrufe von siftDown
- Kosten je $O(k-\ell)$. Insgesamt:



$$O\left(\sum_{0\le\ell<k} 2^\ell (k-\ell)\right) = O\left(2^k \sum_{0\le\ell<k} \frac{k-\ell}{2^{k-\ell}}\right) = O\left(2^k \underbrace{\sum_{j\ge 1} \frac{j}{2^j}}_{O(1)!}\right)$$



$$= O(2^k) = O(n)$$ □



Ein nützlicher Rechentrick

$$\sum_{j\geq 1} j\cdot 2^{-j} = \sum_{j\geq 1} 2^{-j} + \sum_{j\geq 2} 2^{-j} + \sum_{j\geq 3} 2^{-j} + \ldots$$
$$= (1 + 1/2 + 1/4 + 1/8 + \ldots)\cdot \sum_{j\geq 1} 2^{-j}$$
$$= 2\cdot 1 = 2$$

```
 1/2  +  1/4  +  1/8  +  1/16  +  ... = 1
         1/4  +  1/8  +  1/16  +  ... = 1/2
                 1/8  +  1/16  +  ... = 1/4
                         1/16  +  ... = 1/8
                          ...         = ...
-----------------------------------------------
1*1/2 + 2*1/4 + 3*1/8 + 4*1/16  +  ... =  2
```

Heapsort

Procedure heapSortDecreasing($a[1..n]$)
    buildHeap(a)
    for $i := n$ **downto** 2 **do**
        $h[i] :=$ deleteMin

- Laufzeit: $O(n \log n)$
- Andere Sichtweise: effiziente Implementierung von Sortieren durch Auswahl

pwa.klicker.uzh.ch/join/algoeng

Heapsort

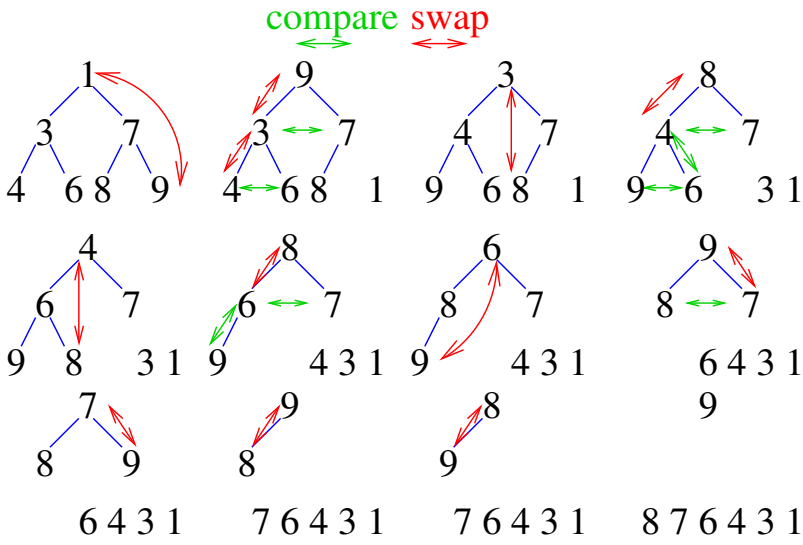
Procedure heapSortDecreasing($a[1..n]$)
    buildHeap(a)
    for $i := n$ **downto** 2 **do**
        $h[i] :=$ deleteMin

- Laufzeit: $O(n \log n)$
- Andere Sichtweise: effiziente Implementierung von Sortieren durch Auswahl

Frage

Wie kann HeapSort am elegantesten/effizientesten verwendet werden, um aufsteigend zu sortieren?

Heapsort
Beispiel



Heapsort ↔ Quicksort ↔ Mergesort

	Heapsort	Quicksort	Mergesort
Vergleiche	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
E[Vergleiche]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
zusätzl. Platz	$O(1)$	$O(\log n)$	$O(n)$
Cachezugriffe	$O(n \log n)$	$O(\frac{n}{B} \log n)$	$O(\frac{n}{B} \log n)$
(B =Blockgröße)			

Kompromiss: z. B.
introspektives Quicksort der C++ Standardbibliothek:
Quicksort starten. Zu wenig Fortschritt? Umschalten auf Heapsort.

Adressierbare Prioritätslisten

Procedure build($\{e_1, \ldots, e_n\}$) $M := \{e_1, \ldots, e_n\}$
Function size **return** $|M|$
Procedure insert(e) $M := M \cup \{e\}$
Function min **return** $\min M$
Function deleteMin $e := \min M$; $M := M \setminus \{e\}$; **return** e
Function remove(h : Handle) $e := h$; $M := M \setminus \{e\}$; **return** e
Procedure decreaseKey(h : Handle, k : Key) **assert** key(h) $\geq k$; key(h) $:= k$
Procedure merge(M') $M := M \cup M'$

Adressierbare Prioritätslisten
Anwendungen

Greedy-Algorithmus:

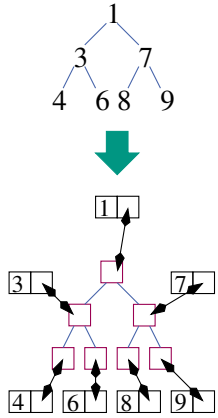
while solution not complete **do**
    add the best available “piece” to the solution
    update piece priorities // e.g., using addressable priority queue

Beispiele:

- Dijkstras Algorithmus für kürzeste Wege
- Jarník-Prim Algorithmus für minimale Spannbäume
- Scheduling: Jobs $\rightarrow$ am wenigsten belastete Maschine
- Hierarchiekonstruktion für Routenplanung
- Suche nach erfüllenden Belegungen aussagenlog. Formeln?

Adressierbare Binäre Heaps

Problem: Elemente bewegen sich.
Dadurch werden Elementverweise ungültig.
(Ein) **Ausweg:** Unbewegliche Vermittler-Objekte.
Invariante: proxy(e) verweist auf Position von e.
$\rightsquigarrow$ Vermittler bei jeder Vertauschung aktualisieren.
$\rightsquigarrow$ Rückverweis Element $\rightarrow$ Vermittler



Laufzeit:
$O(\log n)$ für alle Operationen ausser merge und buildHeap, die $O(n)$ brauchen.

Adressierbare Prioritätslisten

Laufzeiten

Operation	Binary Heap	Fibonacchi Heap (Buch)
build	$O(n)$	$O(n)$
size	$O(1)$	$O(1)$
min	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(1)$
deleteMin	$O(\log n)$	$O(\log n)$
remove	$O(\log n)$	$O(\log n)$
decreaseKey	$O(\log n)$	$O(1)$ am.
merge	$O(n)$	$O(1)$

Prioritätslisten: Mehr

- Untere Schranke $\Omega(\log n)$ für deleteMin, vergleichsbasiert. Beweis: Übung
- ganzzahlige Schlüssel (stay tuned)
- extern: Geht gut (nichtaddressierbar)
- parallel: Semantik?

Prioritätslisten: Zusammenfassung

- Häufig benötigte Datenstruktur
- Addressierbarkeit ist nicht selbstverständlich
- Binäre Heaps sind einfache, relativ effiziente Implementierung

Index

Karlsruher Institut für Technologie

Algorithm Engineering

Algorithmen I – 7. Sortierte Folgen

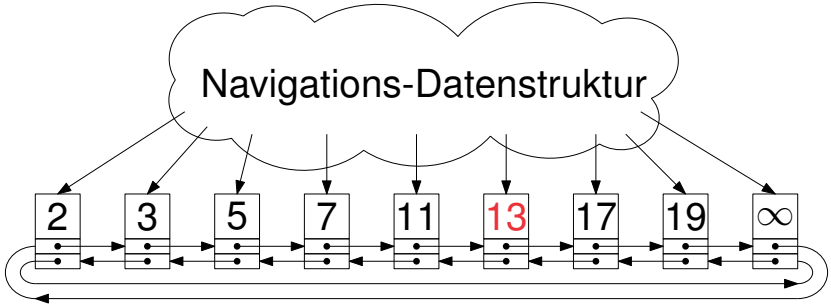
Sommersemester 2025

Peter Sanders | Stand: 30. Juli 2025

Sortierte Folgen

$\langle e_1, \ldots, e_n \rangle$ mit $e_1 \leq \cdots \leq e_n$
„kennzeichnende“ Funktion:

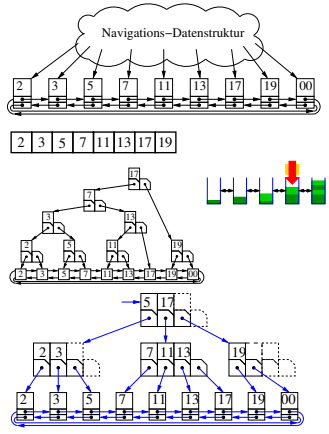
$$M.\text{locate}(k) := \textbf{addressof } \min \{e \in M : e \geq k\}$$



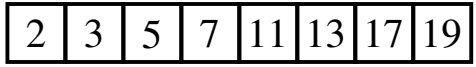
Annahme: Dummy-Element mit Schlüssel ∞

Überblick

- Statisch: sortiertes Array
- Untere Schranke für die Suche
- Dynamisch: Operationen
- Anwendungen/Abgrenzungen
- Binäre Suchbäume
- Allgemeine Knotengrade: (a, b)**-Bäume**
- Augmentierung
  $\rightsquigarrow$ mehr Operationen
- Zusammenfassung,
  Messungen,
  Blick über Tellerrand



Statisch: Sortiertes Feld mit binärer Suche




```
// Find min {i ∈ 1..n+1 : a[i] ≥ k}
Function locate(a[1..n], k : Element)
    (ℓ, r) := (0, n+1)                          // Assume a[0] = −∞, a[n+1] = ∞
    while ℓ + 1 < r do
        invariant 0 ≤ ℓ < r ≤ n+1  and  a[ℓ] < k ≤ a[r]
        m := ⌊(r + ℓ)/2⌋                        // ℓ < m < r
        if k ≤ a[m] then r := m else ℓ := m
    return r
```

Übung: Müssen die Sentinels ∞ / $-\infty$ tatsächlich vorhanden sein?

Übung: Variante von binärer Suche:
bestimme ℓ, r so dass $a[\ell..r-1] = [k, \ldots, k]$, $a[\ell-1] < k$ und $a[r] > k$

Statisch: Sortiertes Feld mit binärer Suche

2	3	5	7	11	13	17	19

```
// Find min {i ∈ 1..n+1 : a[i] ≥ k}
Function locate(a[1..n], k : Element)
    (ℓ, r) := (0, n+1)                          // Assume a[0] = −∞, a[n+1] = ∞
    while ℓ + 1 < r do
        invariant 0 ≤ ℓ < r ≤ n+1  and  a[ℓ] < k ≤ a[r]
        m := ⌊(r + ℓ)/2⌋                        // ℓ < m < r
        if k ≤ a[m] then r := m else ℓ := m
    return r
```

Zeit: $O(\log n)$

Beweisidee: $r - \ell$ „halbiert“ sich in jedem Schritt

Statisch: Sortiertes Feld mit binärer Suche

```
// Find min {i ∈ 1..n+1 : a[i] ≥ k}
Function locate(a[1..n], k : Element)
    (ℓ, r) := (0, n+1)                                   // Assume a[0] = −∞, a[n+1] = ∞
    while ℓ + 1 < r do
        invariant 0 ≤ ℓ < r ≤ n+1   and   a[ℓ] < k ≤ a[r]
        m := ⌊(r + ℓ)/2⌋                                 // ℓ < m < r
        if k ≤ a[m] then r := m else ℓ := m
    return r
```

index:	0	1	2	3	4	5	6	7	8	9	k=16
a:	$-\infty$	2	3	5	7	11	13	17	19	∞	
	$-\infty$	2	3	5	7	11	13	17	19	∞	
	$-\infty$	2	3	5	7	11	13	17	19	∞	
	$-\infty$	2	3	5	7	11	13	17	19	∞	

Überblick

- Statisch: sortiertes Array
- Untere Schranke für die Suche
- Dynamisch: Operationen
- Anwendungen/Abgrenzungen
- Binäre Suchbäume
- Allgemeine Knotengrade: (a, b)**-Bäume**
- Augmentierung $\rightsquigarrow$ mehr Operationen
- Zusammenfassung, Messungen, Blick über Tellerrand

Logarithmische Untere Schranke für Suche

Satz: Wenn nur Elementvergleiche möglich sind benötigt Suche in einer Menge von n Elementen Zeit $\Omega(\log n)$.

Beweisskizze:
n verschiedene Ergebnisse.
Nur ein Bit Information pro Vergleich.
$\lceil \log n \rceil$ bits nötig um n verschiedene Ergebnisse zu ermöglichen.

Überblick

- Statisch: sortiertes Array
- Untere Schranke für die Suche
- Dynamisch: Operationen
- Anwendungen/Abgrenzungen
- Binäre Suchbäume
- Allgemeine Knotengrade: (a, b)**-Bäume**
- Augmentierung $\rightsquigarrow$ mehr Operationen
- Zusammenfassung, Messungen, Blick über Tellerrand

Dynamische Sortierte Folgen – Grundoperationen

insert, remove, update, locate $O(\log n)$

(M.locate(k):= $\min \{e \in M : e \geq k\}$)

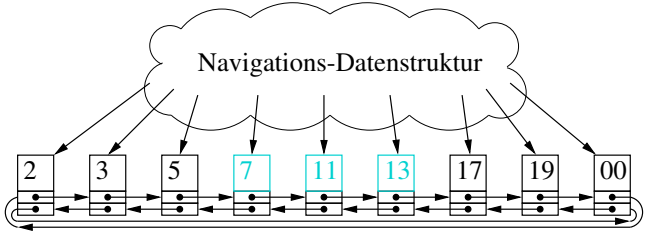
Mehr Operationen

$\langle \text{min}, \ldots, a, \ldots, b, \ldots, \text{max} \rangle$

min: Erstes Listenelement — Zeit $O(1)$

max: Letztes Listenelement — Zeit $O(1)$

```
rangeSearch(a, b)                                  // O(log n + |result|)
    result:= ⟨⟩
    h:= locate(a)
    while h → e ≤ b do
        result.pushBack(h → e)
        h:= h →next
    return result
```

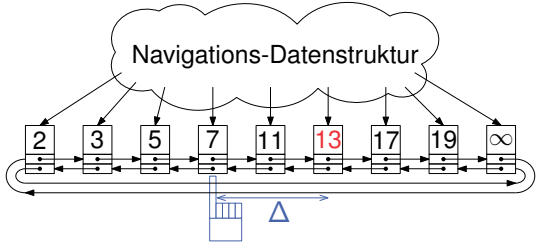


Noch mehr Operationen

- (re)**build**: Navigationstruktur für sortierte Liste aufbauen — $O(n)$
- $\langle w, \ldots, x \rangle$.**concat**$(\langle y, \ldots, z \rangle) = \langle w, \ldots, x, y, \ldots, z \rangle$ — $O(\log n)$
- $\langle w, \ldots, x, y, \ldots, z \rangle$.**split**$(y) = (\langle w, \ldots, x \rangle, \langle y, \ldots, z \rangle)$ — $O(\log n)$

Zählen: rank, select, rangeSize — $O(\log n)$

Fingersuche: Δ = Abstand zu Fingerinfo
zusätzlicher Parameter für insert, remove, locate,... — $O(\log n) \to \log \Delta$



Überblick

- Statisch: sortiertes Array
- Untere Schranke für die Suche
- Dynamisch: Operationen
- Anwendungen/Abgrenzungen
- Binäre Suchbäume
- Allgemeine Knotengrade: (a, b)**-Bäume**
- Augmentierung $\rightsquigarrow$ mehr Operationen
- Zusammenfassung, Messungen, Blick über Tellerrand

Abgrenzung: Sortierte Folgen $\leftrightarrow$ andere Datenstrukturen

Hash-Tabelle: nur insert, remove, find. Kein locate, rangeQuery

Sortiertes Feld: nur bulk-Updates. Aber:
Hybrid-Datenstruktur oder $\log \frac{n}{M}$ geometrisch wachsende statische Datenstrukturen

Prioritätsliste: nur insert, deleteMin, (decreaseKey, remove). Dafür: schnelles merge

Insgesamt: die eierlegende Wollmilchdatenstruktur.
„Etwas“ langsamer als speziellere Datenstrukturen

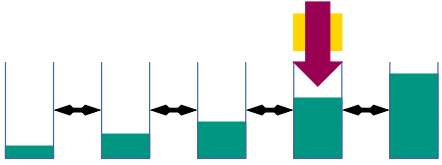
Sortierte Folgen – Anwendungen

- Best-First Heuristiken
- Alg. Geometrie: Sweepline-Datenstrukturen
- Datenbankindex
- ...

Anwendungsbeispiel: Best Fit Bin Packing

Procedure binPacking(s)
    B : SortedSequence // used bins sorted by free capacity
    foreach $e \in s$ by decreasing element size // sort
        if $\neg\exists b \in B$: free(b) $> e$ **then** B.insert(new bin)
        locate $b \in B$ with smallest free(b) $\geq e$
        insert e into bin b

Zeit: $O(|s| \log |s|)$
Qualität: „gut“. Details: nicht hier

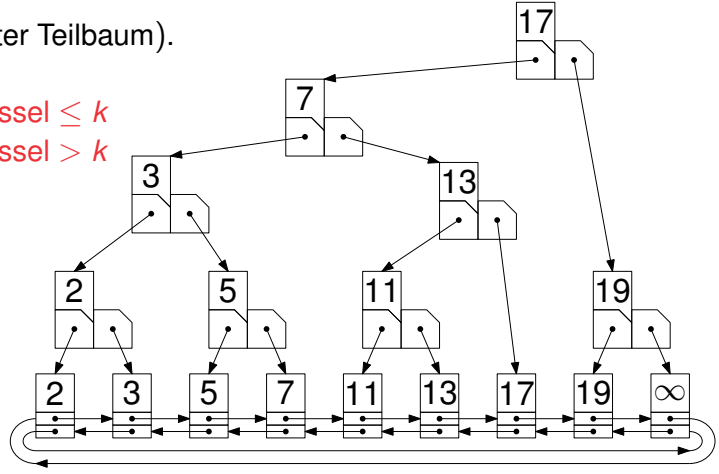


Überblick

- Statisch: sortiertes Array
- Untere Schranke für die Suche
- Dynamisch: Operationen
- Anwendungen/Abgrenzungen
- Binäre Suchbäume
- Allgemeine Knotengrade:
  (a, b)**-Bäume**
- Augmentierung
  $\rightsquigarrow$ mehr Operationen
- Zusammenfassung, Messungen,
  Blick über Tellerrand

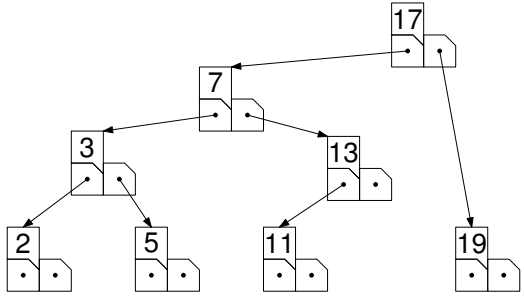
Binäre Suchbäume

Blätter: Elemente einer sortierten Folge.
Innere Knoten $v = (k, \ell, r)$,
(Spalt-Schlüssel, linker Teilbaum, rechter Teilbaum).
Invariante:
über ℓ erreichbare Blätter haben Schlüssel $\leq k$
über r erreichbare Blätter haben Schlüssel $> k$



Varianten, Bemerkungen

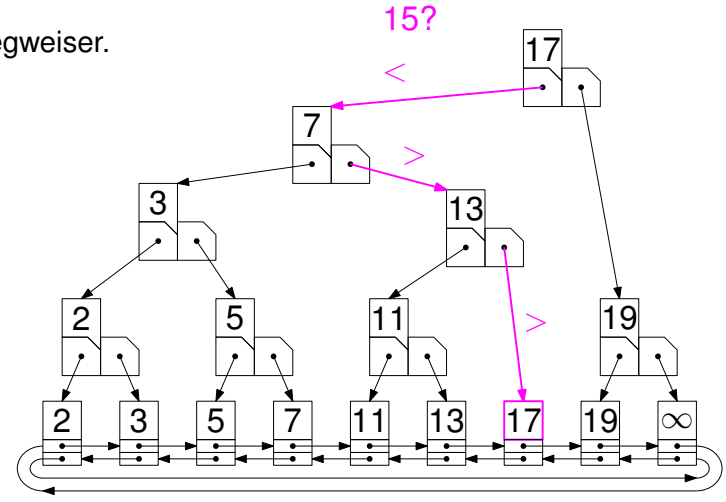
- Dummy Element im Prinzip verzichtbar
- Oft speichern auch innere Knoten Elemente
- „Suchbaum“ wird oft als Synomym für „sortierte Folge“ verwendet. (Aber das vermischt (eine) Implementierung mit der Schnittstelle)



locate(k) – Beispiel

Idee: Benutze Spaltschlüssel x als Wegweiser.

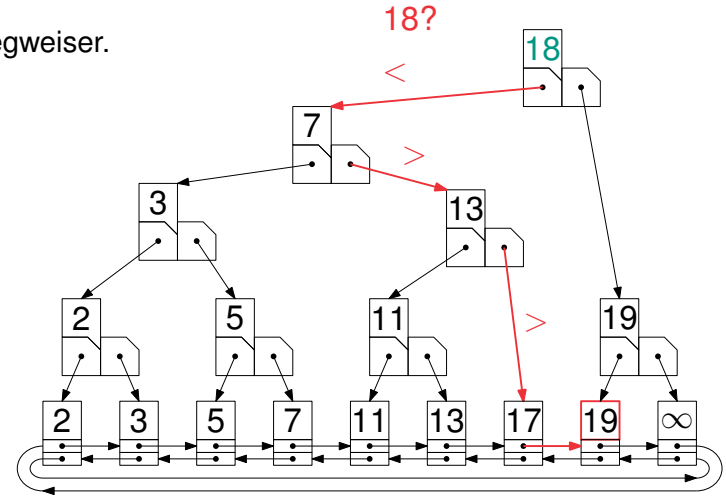
Function locate(k, x)
  if x is a leaf **then**
    if $k \leq x$ **then return** x
    else return $x \rightarrow$next
  if $k \leq x$ **then**
    return locate$(k, x \rightarrow$left$)$
  else
    return locate$(k, x \rightarrow$right$)$



locate(k) – Anderes Beispiel

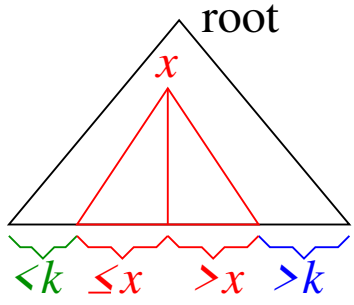
Idee: Benutze Spaltschlüssel x als Wegweiser.

Function locate(k, x)
  if x is a leaf **then**
    if $k \leq x$ **then return** x
    else return $x \rightarrow$next
  if $k \leq x$ **then**
    return locate$(k, x \rightarrow$left$)$
  else
    return locate$(k, x \rightarrow$right$)$



Invariante von locate(k)

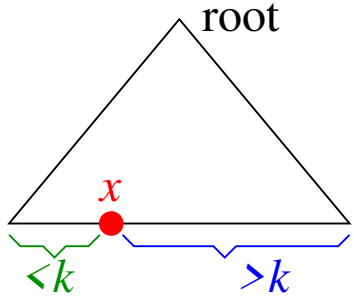
Function locate(k, x)
  if x is a leaf **then**
    if $k \leq x$ **then return** x
    else return $x \rightarrow$next
  if $k \leq x$ **then**
    return locate$(k, x \rightarrow$left$)$
  else
    return locate$(k, x \rightarrow$right$)$



Invariante: Sei X die Menge aller von x erreichbaren Listenelemente.
Listenelemente links von X sind $< k$
Listenelemente rechts von X sind $> k$

Ergebnisberechnung von locate(k)

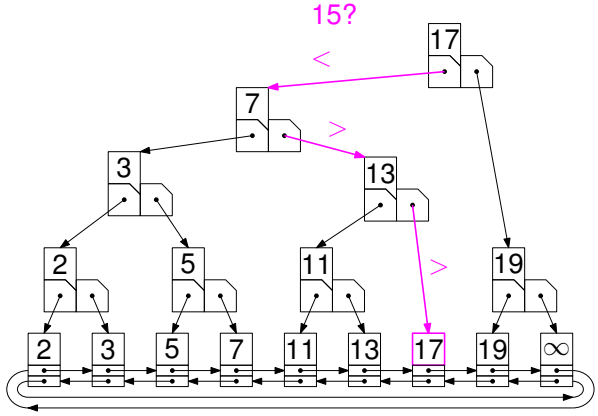
Function locate(k, x)
  if x is a leaf **then**
    if $k \leq x$ **then return** x
    else return $x \rightarrow$next
  if $k \leq x$ **then**
    return locate$(k, x \rightarrow$left$)$
  else
    return locate$(k, x \rightarrow$right$)$



Fall $k = x$**:** return x — Bingo!
Fall $k < x$**:** return x — links isses auch net.
Fall $k > x$**:** return $x \rightarrow$next — das ist $> k$ und k gibts nich

Laufzeit von locate(k)

Function locate(k, x)
  if x is a leaf **then**
    if $k \leq x$ **then return** x
    else return $x \rightarrow$next
  if $k \leq x$ **then**
    return locate$(k, x \rightarrow$left$)$
  else
    return locate$(k, x \rightarrow$right$)$

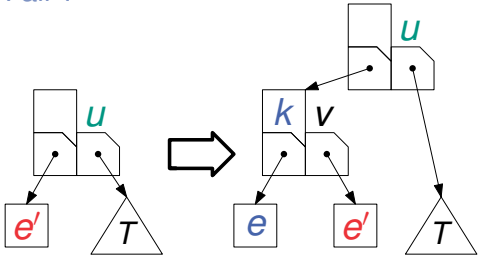


Laufzeit: $O($Höhe$)$.
Bester Fall: perfekt balanciert, d. h. Tiefe $= \lfloor \log n \rfloor$
Schlechtester Fall: Höhe n

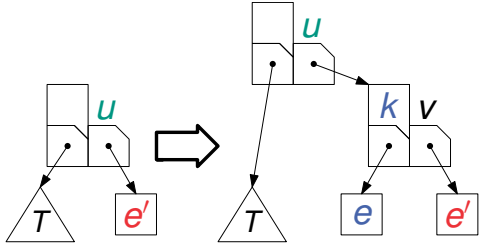
Naives Einfügen

Zunächst wie locate(e). Sei e' gefundenes Element, u der Elterknoten

Fall 1

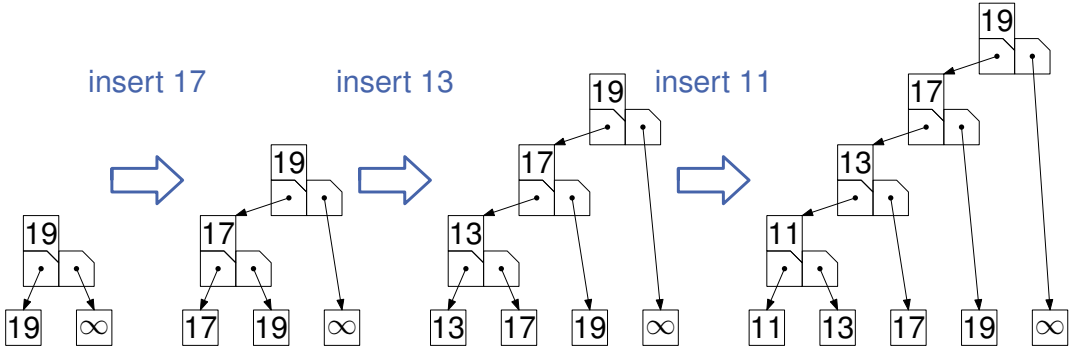


Fall 2



$$k = \text{key}(e)$$

Beispiel



Problem: Der Baum wird beliebig unbalanciert $\rightsquigarrow$ langsam

Suchbäume balancieren

perfekte Balance: schwer aufrechtzuerhalten

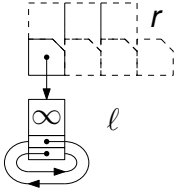
flexible Höhe $O(\log n)$: balancierte binäre Suchbäume.
nicht hier (Variantenzoo).

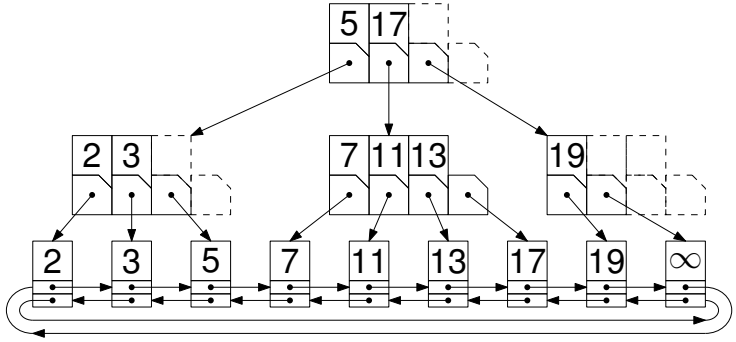
flexibler Knotengrad: (a, b)-Bäume.
$\approx$ Grad zwischen a und b.
Höhe $\approx \log_a n$

Überblick

- Statisch: sortiertes Array
- Untere Schranke für die Suche
- Dynamisch: Operationen
- Anwendungen/Abgrenzungen
- Binäre Suchbäume
- Allgemeine Knotengrade:
  (a, b)**-Bäume**
- Augmentierung
  $\rightsquigarrow$ mehr Operationen
- Zusammenfassung, Messungen,
  Blick über Tellerrand

(a, b)-Bäume





Blätter: Listenelemente (wie gehabt). Alle mit gleicher Tiefe!

Innere Knoten: Grad $a..b$

Wurzel: Grad $2..b$, (Grad 1 für $\langle\rangle$)

Items

Class ABHandle : **Pointer** to ABItem or Item

Class ABItem(splitters : Sequence **of** Key, children : Sequence **of** ABHandle)

d=|children| : $1..b$ // outdegree

s=splitters : **Array** $[1..b-1]$ **of** Key

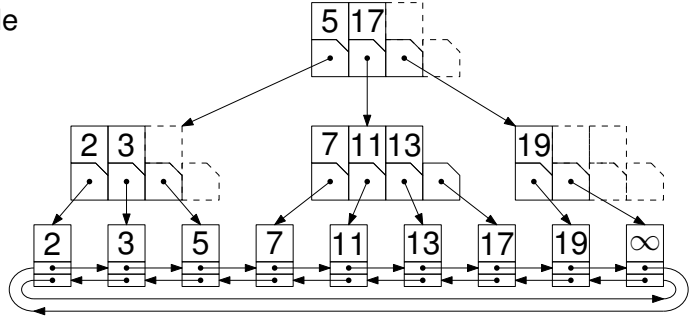
c=children : **Array** $[1..b]$ **of** Handle

Invariante:

e über $c[i]$ erreichbar

$\Rightarrow s[i-1] < e \leq s[i]$ mit

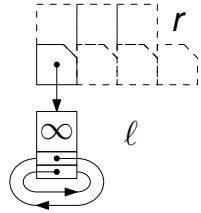
$s[0] = -\infty$, $s[d] = s[d+1] = \infty$



Initialisierung

Class ABTree($a \geq 2 : \mathbb{N}$, $b \geq 2a-1 : \mathbb{N}$) **of** Element
$\ell = \langle\rangle$: List **of** Element
r : ABItem($\langle\rangle$, $\langle \ell.\text{head} \rangle$)
height$=1$: $\mathbb{N}$

// Locate the smallest Item with key $k' \geq k$
Function locate(k : Key) : Handle **return** r.locateRec(k, height)

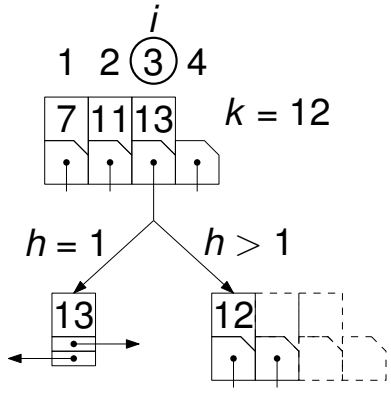


Locate

Function ABItem::locateLocally(k : Key) : $\mathbb{N}$
    return min $\{i \in 1..d : k \leq s[i]\}$

Function ABItem::locateRec(k : Key, $h : \mathbb{N}$) : Handle
    i:= locateLocally(k)
    if $h = 1$ **then**
        if $c[i] \rightarrow e \geq k$ Then **return** $c[i]$
        else return $c[i] \rightarrow$ next
    else
        return $c[i] \rightarrow$locateRec(k, $h - 1$)

Invariante: analog binäre Suchbäume

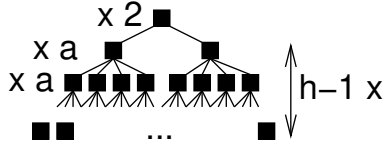


Locate – Laufzeit

$O(b \cdot \text{height})$

Übung: $b \to \log b$?

Lemma: $\text{height} = h \leq 1 + \left\lfloor \log_a \frac{n+1}{2} \right\rfloor$



Beweis:

Fall $n = 1$**:** $\text{height} = 1$.

Fall $n > 1$**:**

Wurzel hat Grad ≥ 2 und innere Knoten haben Grad $\geq a$.

$\Rightarrow \geq 2a^{h-1}$ Blätter.

Es gibt $n+1$ Blätter.

$\Rightarrow n+1 \geq 2a^{h-1}$

$\Rightarrow h \leq 1 + \log_a \frac{n+1}{2}$

Rundung folgt weil h eine ganze Zahl ist. □

Einfügen – Algorithmenskizze

Procedure insert(e)

Finde Pfad Wurzel–nächstes Element e'

ℓ.insertBefore(e, e')

füge key(e) als neuen Splitter in Vorgänger u

if $u.d = b + 1$ **then**

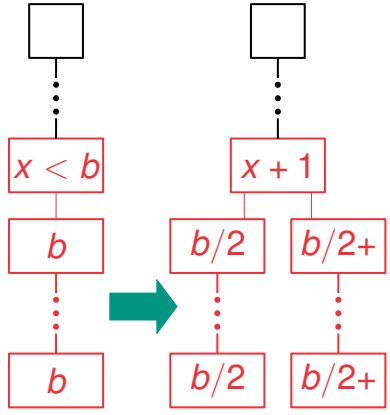
spalte u in 2 Knoten mit Graden

$\lfloor (b+1)/2 \rfloor, \lceil (b+1)/2 \rceil$

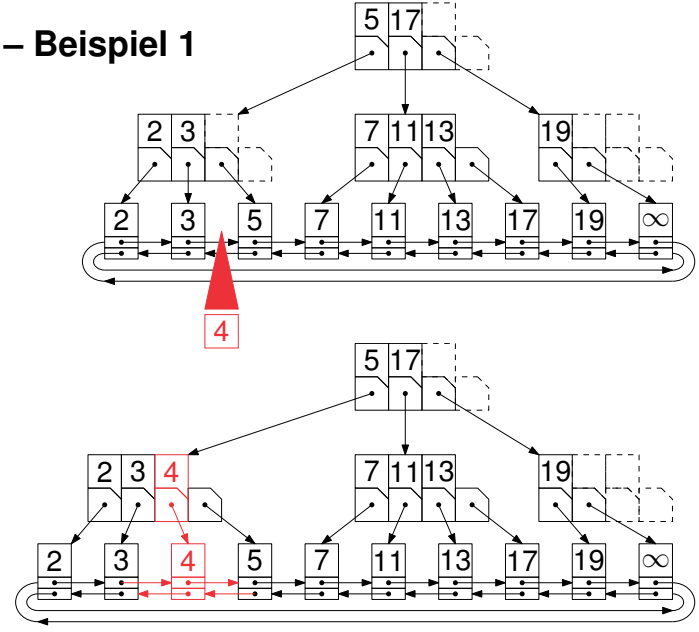
Weiter oben einfügen, spalten

...

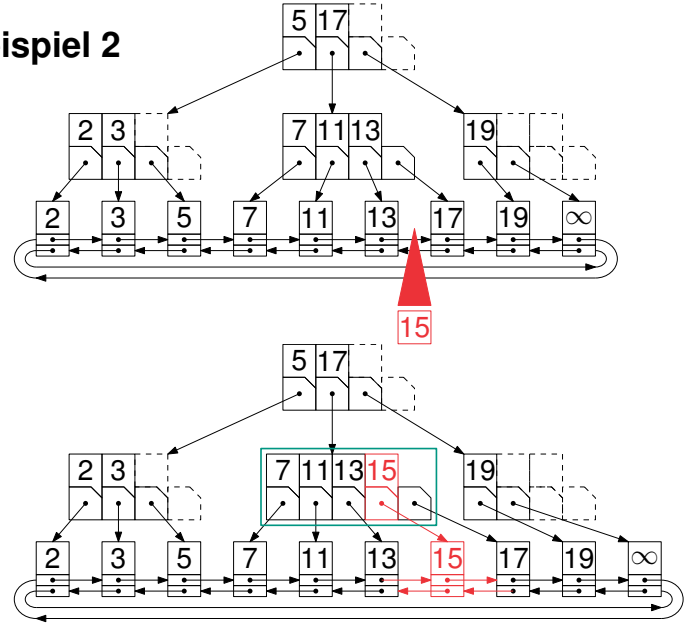
ggf. neue Wurzel



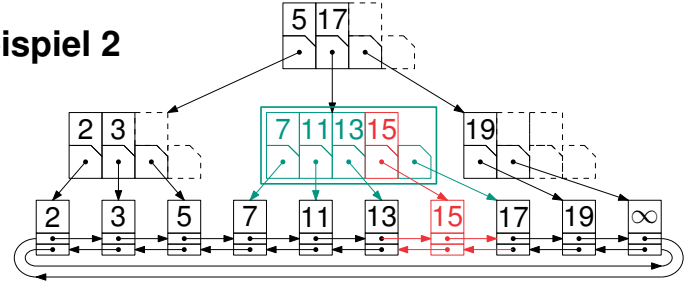
Einfügen – Beispiel 1

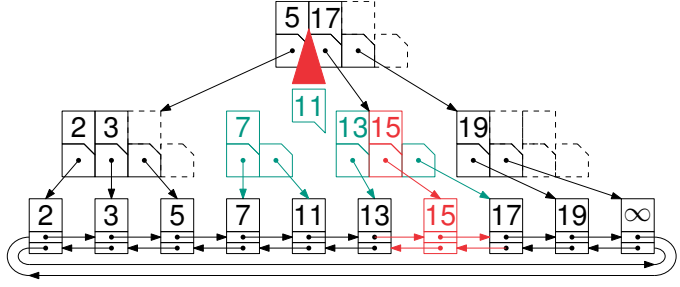


Einfügen – Beispiel 2

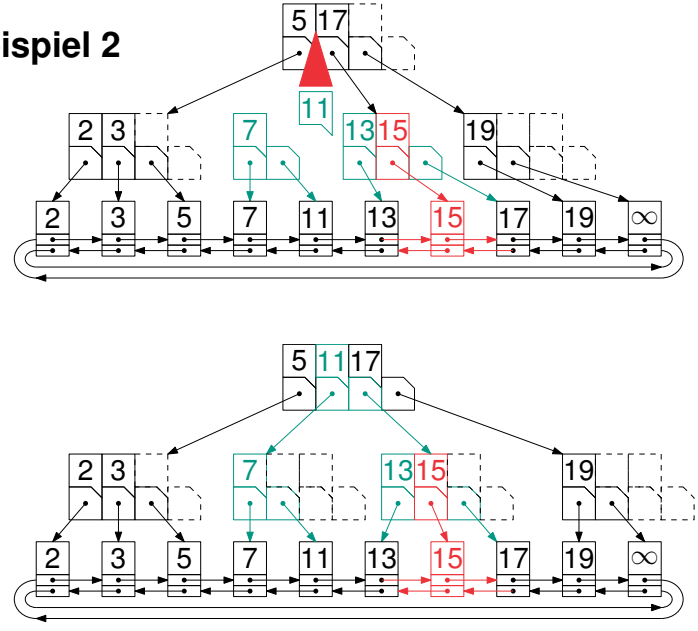


Einfügen – Beispiel 2

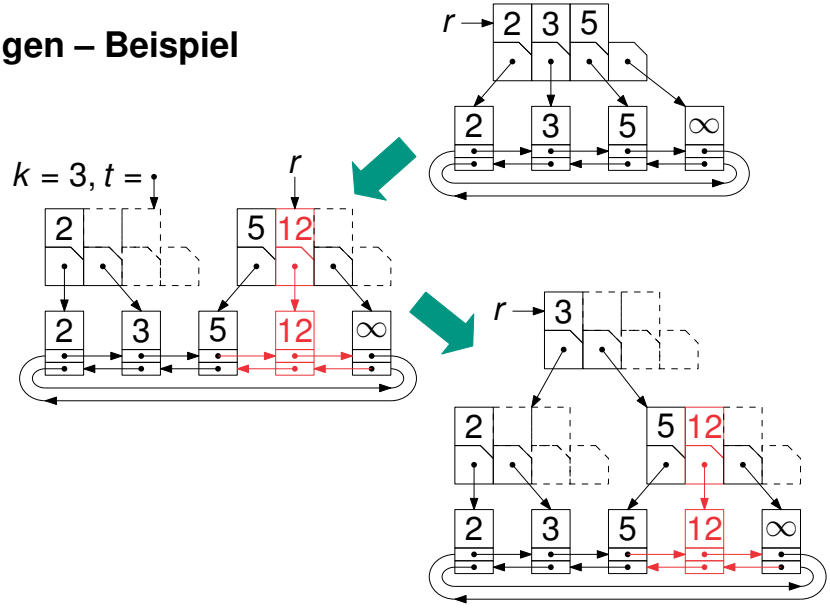




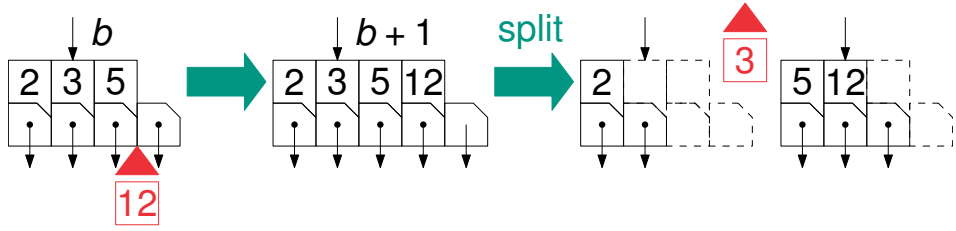
Einfügen – Beispiel 2



Einfügen – Beispiel



Einfügen – Korrektheit



Nach dem Spalten müssen zulässige Items entstehen:

$$\left\lfloor \frac{b+1}{2} \right\rfloor \overset{!}{\geq} a \Leftrightarrow b \geq 2a-1$$

Weil $\left\lfloor \frac{(2a-1)+1}{2} \right\rfloor = \left\lfloor \frac{2a}{2} \right\rfloor = a$

Einfügen – Implementierungsdetails

- Spalten pflanzt sich von unten nach oben fort. Aber wir speichern nur Zeiger nach unten.
  Lösung: Rekursionsstapel speichert Pfad.
- Einheitlicher Itemdatentyp mit Kapazität für b Nachfolger.
  einfacher, schneller, Speicherverwaltung!
- Baue nie explizit temporäre Knoten mit $b + 1$ Nachfolgern.

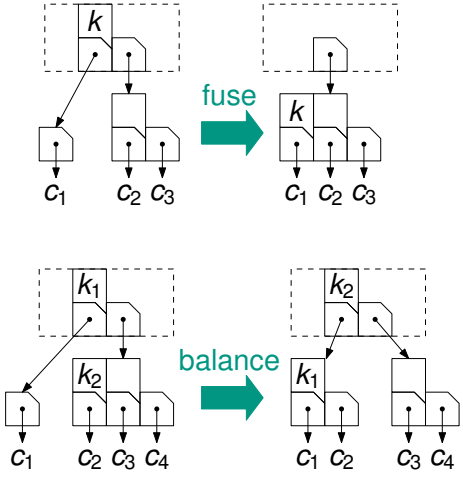
Einfügen – Pseudocode

```
// ℓ: "the list"
// r: root
// height (of tree)
Procedure ABTree::insert(e : Element)
    (k, t):= r.insertRec(e, height, ℓ)
    if t ≠ null then
        r:= allocate ABItem(⟨k⟩, ⟨r, t⟩)
        height++
```

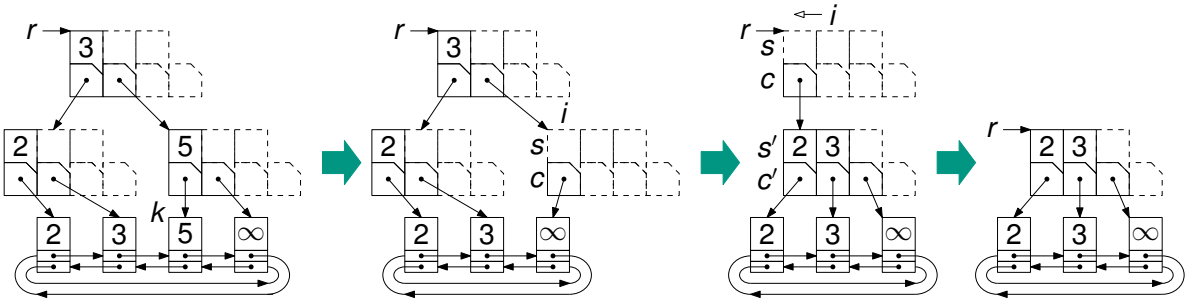
```
Function ABItem::insertRec(e : Element, h : ℕ, ℓ : List of Element) : Key×ABHandle
    i:= locateLocally(e)
    if h = 1 then
        (k, t):= (key(e), ℓ.insertBefore(e, c[i]))                    // base
    else
        (k, t):= c[i] → insertRec(e, h − 1, ℓ)                        // recurse
        if t = null then return (⊥, null )
    s′:= ⟨s[1], ..., s[i − 1], k, s[i], ..., s[d − 1]⟩                // new splitter
    c′:= ⟨c[1], ..., c[i − 1], t, c[i], ..., c[d]⟩                    // new child
    if d < b then
        (s, c, d):= (s′, c′, d + 1)
        return (⊥, null )
    else                                                              // split this node
        d:= ⌊(b + 1)/2⌋
        s:= s′[b + 2 − d..b]
        c:= c′[b + 2 − d..b + 1]
        return (s′[b + 1 − d], allocate ABItem(s′[1..b − d], c′[1..b + 1 − d]))
```

Entfernen – Algorithmenskizze

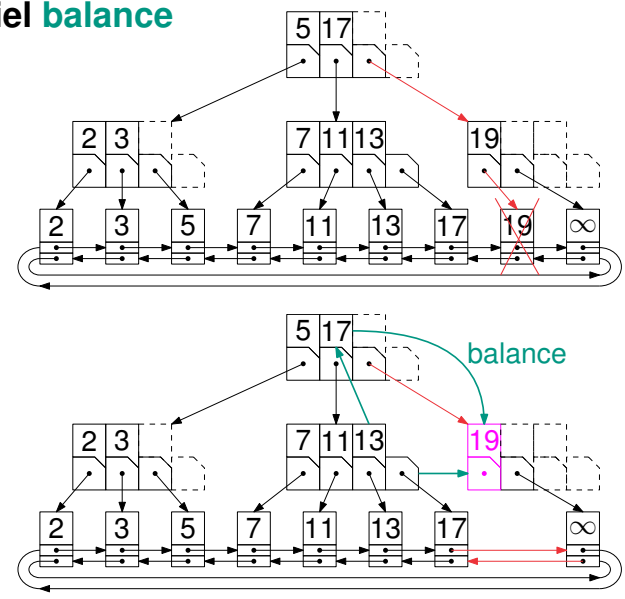
Procedure remove(e)
    Finde Pfad Wurzel–e
    ℓ.remove(e)
    entferne key(e) in Vorgänger u
    if $u.d = a - 1$ **then**
        finde Nachbarn u'
        if $u'.d + a - 1 \leq b$ **then**
            fuse(u', u)
            Weiter oben splitter entfernen
            ...
            ggf. Wurzel entfernen
        else balance(u', u)



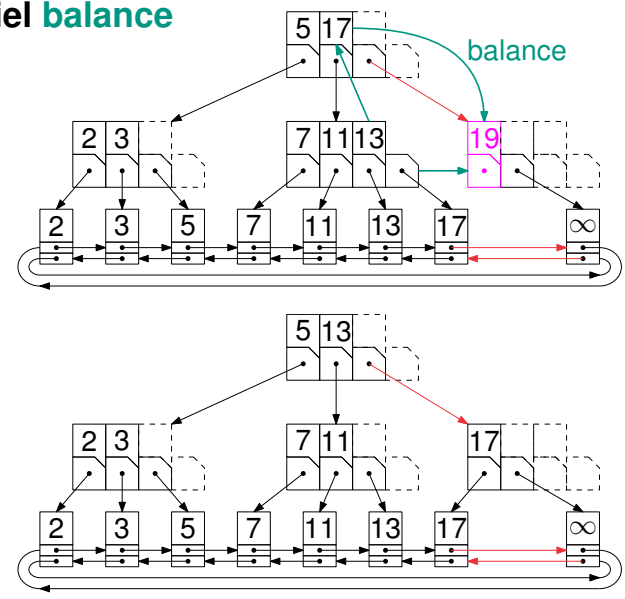
Entfernen – Beispiel fuse



Entfernen – Beispiel balance



Entfernen – Beispiel balance

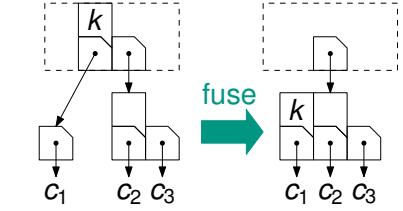


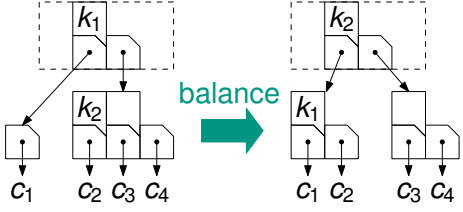
Entfernen – Korrektheit

Nach fuse
müssen zulässige Items entstehen:

$a + (a-1) \overset{!}{\leq} b \Leftrightarrow b \geq 2a-1$

hatten wir schon!





Einfügen und Entfernen – Laufzeit

$$\begin{aligned} O(b \cdot \text{Höhe}) &= O(b \log_a n) \\ &= O(\log n) \text{ für } \{a, b\} \subseteq O(1) \end{aligned}$$

(a, b)-Bäume
Implementierungsdetails

Etwas kompliziert...

Wie merkt man sich das? $\rightarrow$ Gar nicht!

Man merkt sich:

- Invarianten
  Höhe, Knotengrade
- Grundideen
  split, balance, fuse

Den Rest leitet man sich nach Bedarf neu her.

```
Procedure ABTree::remove(k : Key)
    r.removeRec(k, height, ℓ)
    if r.d = 1 ∧ height > 1 then r' := r; r := r'.c[1]; dispose r'
Procedure ABItem::removeRec(k : Key, h : ℕ, ℓ : List of Element)
    i := locateLocally(k)
    if h = 1 then
        if key(c[i] → e) = k then
            ℓ.remove(c[i])
            removeLocally(i)
    else
        c[i] → removeRec(e, h − 1, ℓ)
        if c[i] → d < a then
            if i = d then i−−
            s' := concatenate(c[i] → s, ⟨s[i]⟩, c[i + 1] → s))
            c' := concatenate(c[i] → c, c[i + 1] → c)
            d' := |c'|
            if d' ≤ b then    // fuse
                (c[i + 1] → s, c[i + 1] → c, c[i + 1] → d) := (s', c', d')
                dispose c[i]; removeLocally(i)
            else              // balance
                m := ⌈d'/2⌉
                (c[i] → s, c[i] → c, c[i] → d) := (s'[1..m − 1], c'[1..m], m)
                (c[i + 1] → s, c[i + 1] → c,  c[i + 1] → d) :=
                    (s'[m + 1..d' − 1], c'[m + 1..d'],  d' − m)
                s[i] := s'[m]
Procedure ABItem::removeLocally(i : ℕ)
    c[i..d − 1] := c[i + 1..d]
    s[i..d − 2] := s[i + 1..d − 1]
    d−−
```

Mehr Operationen

min, max, rangeSearch(a, b): $\langle \text{min}, \ldots, a, \ldots, b, \ldots, \text{max} \rangle$
hatten wir schon

build: Übung! Laufzeit $O(n)$!
(Navigationstruktur für sortierte Liste aufbauen)

concat, **split**: nicht hier. Zeit $O(\log n)$
Idee: Ganze Teilbäume umhängen

merge(N, M): sei $n = |N| \leq m = |M|$ Zeit $O\left(n(1 + \log \frac{n}{m})\right)$
nicht hier. Idee: z. B. Fingersuche

Amortisierte Analyse von insert **und** remove

nicht hier.

Grob gesagt: Abgesehen von der Suche fällt nur konstant viel Arbeit an (summiert über alle Operationsausführungen).

Überblick

- Statisch: sortiertes Array
- Untere Schranke für die Suche
- Dynamisch: Operationen
- Anwendungen/Abgrenzungen
- Binäre Suchbäume
- Allgemeine Knotengrade: (a, b)**-Bäume**
- Augmentierung
  $\rightsquigarrow$ mehr Operationen
- Zusammenfassung, Messungen, Blick über Tellerrand

Erweiterte (augmentierte) Suchbäume

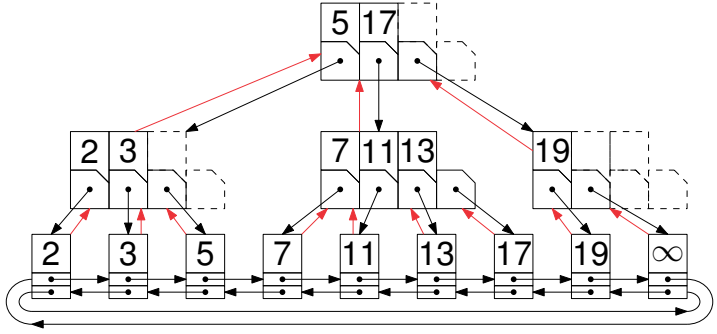
Idee: zusätzliche Infos verwalten $\rightsquigarrow$ mehr (schnelle) Operationen.

Nachteil: Zeit- und Platzverschwendung,
wenn diese Operationen nicht wichtig sind.

gold plating

Elternzeiger

Idee (Binärbaum): Knoten speichern Zeiger auf Elternknoten



Anwendungen: schnelleres remove, insertBefore, insertAfter,
wenn man ein handle des Elements kennt.
Man spart die Suche.
Frage: was speichert man bei (a, b)-Bäumen?

Teilbaumgrößen

Idee (Binärbaum): speichere wieviele Blätter von links erreichbar.
(Etwas anders als im Buch!)

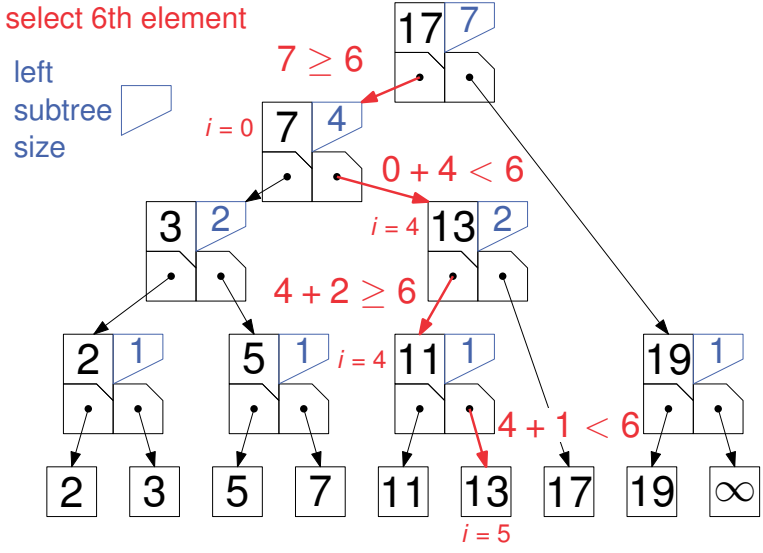
```
// return k-th Element in subtree rooted at h
Function selectRec(h, k)
    if h → leftSize > k then return select(ℓ, k)
    else return select(r, k − leftSize)
```

Zeit: $O(\log n)$

Übung: Was ist anders bei (a, b)-Bäumen?
Übung: Rang eines Elements e bestimmen.
Übung: Größe einer Range $x..y$ bestimmen.

Teilbaumgrößen

Beispiel



Überblick

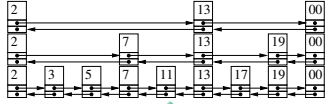
- Statisch: sortiertes Array
- Untere Schranke für die Suche
- Dynamisch: Operationen
- Anwendungen/Abgrenzungen
- Binäre Suchbäume
- Allgemeine Knotengrade: (a, b)**-Bäume**
- Augmentierung $\rightsquigarrow$ mehr Operationen
- Zusammenfassung, Messungen, Blick über Tellerrand

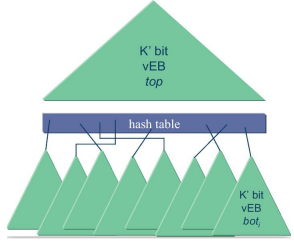
Zusammenfassung

- Suchbäume erlauben viele effiziente Operationen auf sortierten Folgen.
- Oft logarithmische Ausführungszeit
- Der schwierige Teil: logarithmische Höhe erzwingen.
- Augmentierungen $\rightsquigarrow$ zusätzliche Operationen
- Wiedermal: Invarianten leiten die Algorithmenentwicklung

Mehr zu sortierten Folgen

- Karteikasten $\rightsquigarrow$ Array mit Löchern
- “Trees outside”: Skip-Listen
- (a, b)-Bäume sind wichtig für externe Datenstrukturen
- Ganzzahlige Schlüssel aus $1..U$ $\rightsquigarrow$ Grundoperationen in Zeit $O(\log \log U)$
- Verallgemeinerungen: Zeichenketten, mehrdimensionale Daten
- Parallelisierung: Bulk-Zugriffe





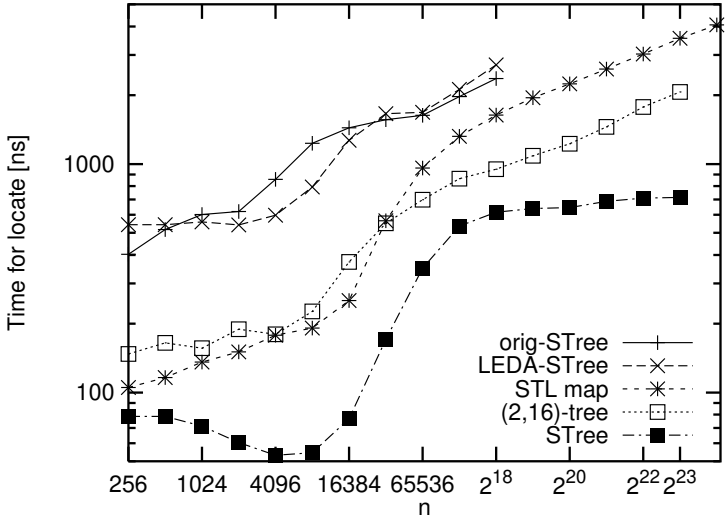
Sortierte Folgen in realen Programmiersprachen

C++ map, multimap, set, multiset
Typischerweise implementiert als balancierter binärer Suchbaum, z.B. red-black-tree, der seinerseits eine Repräsentation eine $(2, 4)$-Baums ist.
Eine effiziente Implementierung allgemeiner (a, b)-Trees ist z.B. hier:
`github.com/tlx/tlx`, `tlx.github.io/group__tlx__container__btree.html`

Java NavigableMap, TreeSet, TreeMap
ebenfalls red-black-tree

Rust BTreeMap, BTreeSet
(a, b)-Bäume

Ein paar Zahlen



Index

Karlsruher Institut für Technologie

Algorithm Engineering

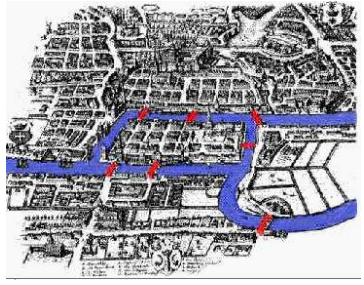
Algorithmen I – 8. Graphrepräsentation

Sommersemester 2025

Peter Sanders | Stand: 30. Juli 2025

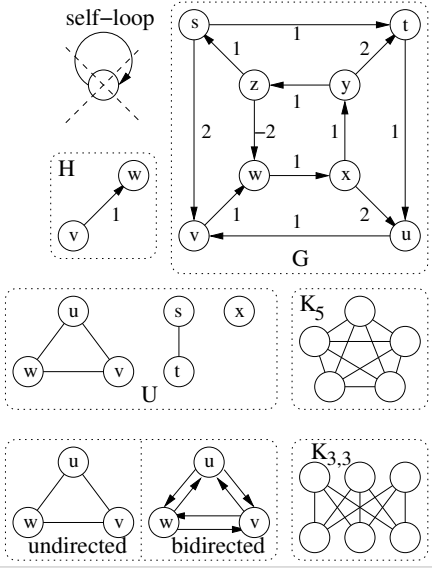
Graphrepräsentation

- 1736 fragt L. Euler die folgende “touristische” Frage: Gibt es eine Stadtspaziergang, der jede Brücke genau einmal überquert? (das Königsberger Brückenproblem)
- Straßen- oder Computernetzwerke
- Schaltkreise, natürliche und reale neuronale Netzwerke
- Zugverbindungen (Raum und Zeit)
- Soziale Netzwerke (Freundschafts-, Zitier-, Empfehlungs-,...)
- Aufgabenabhängigkeiten $\rightsquigarrow$ Scheduling Probleme
- Werte und arithmetische Operationen $\rightsquigarrow$ Compilerbau
- Diskretisierung kontinuierlicher Fragestellungen (Robotik, numerische Simulationen,...)
- ...



Graphrepräsentation

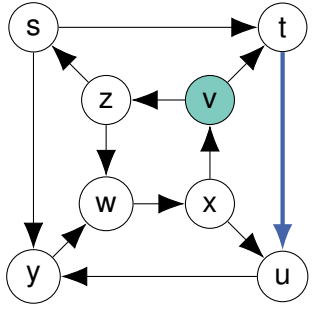
- Was zählt sind die Operationen
- Eine triviale Repräsentation
- Felder
- Verkettete Listen
- Matrizen
- Implizit
- Diskussion



Notation und Konventionen

Graph $G = (\underbrace{V}_{\text{Knoten}}, \underbrace{E}_{\text{Kanten}})$:

- $n = |V|$
- $m = |E|$
- Knoten: s, t, u, v, w, x, y, z (engl. nodes, vertices)
- Kanten $e \in E$. Knotenpaare (manchmal Knotenmengen der Größe 2) (engl. edges)



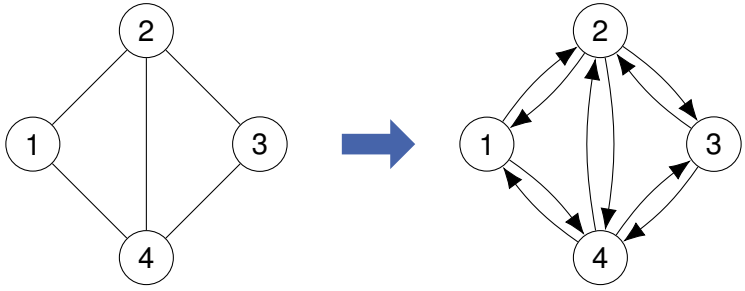
WICHTIG: Buchstabenzuordnungen = unverbindliche Konvention

- Manchmal werden ganz andere Buchstaben verwendet.
- Im Zweifel immer genau sagen was was ist.

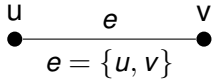
Das gilt für die ganze theoretische Informatik

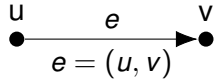
Ungerichtete $\rightarrow$ gerichtete Graphen

Meist repräsentieren wir ungerichtete Graphen durch bigerichtete Graphen
$\rightsquigarrow$ wir konzentrieren uns auf gerichtete Graphen



Kante als Menge





Kante als Tupel

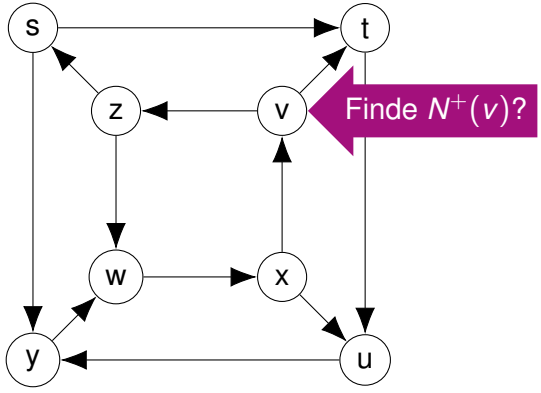
Übersicht

Operationen
Operationen

Ziel: $\mathrm{O}(\text{Ausgabegröße})$ für alle Operationen

Grundoperationen

- Statische Graphen
  - Konstruktion, Konversion und Ausgabe: ($\mathrm{O}(m+n)$ Zeit)
  - Navigation: Gegeben v, finde ausgehende Kanten.
- Dynamische Graphen
  - Knoten/Kanten einfügen/löschen

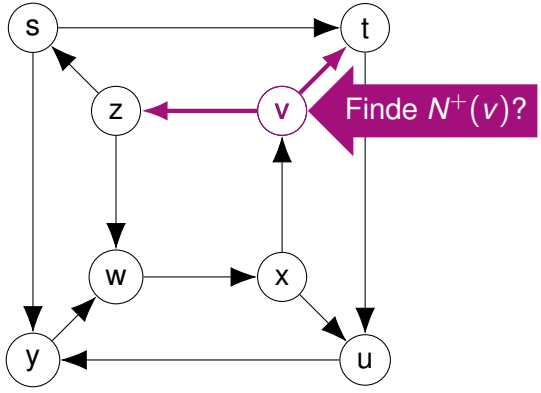


Operationen
Operationen

Ziel: $\mathrm{O}(\text{Ausgabegröße})$ für alle Operationen

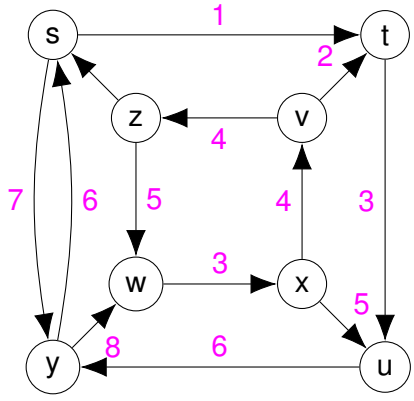
Grundoperationen

- Statische Graphen
  - Konstruktion, Konversion und Ausgabe: ($\mathrm{O}(m+n)$ Zeit)
  - Navigation: Gegeben v, finde ausgehende Kanten.
- Dynamische Graphen
  - Knoten/Kanten einfügen/löschen



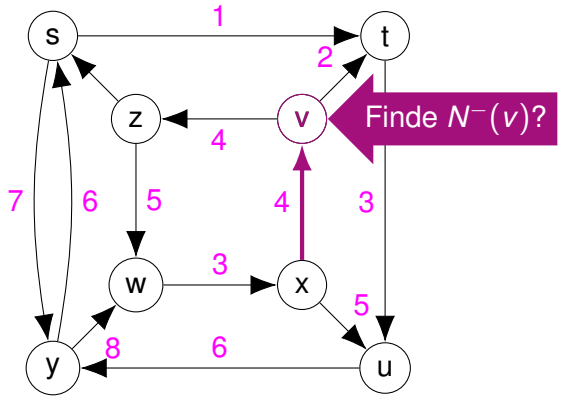
Operationen
Weitere Operationen

- Zugriff auf assoziierte Information
- mehr Navigation: finde eingehende Kanten
- Kantenanfragen: $(z, x) \in E$?
- Zugriff auf Rückkante



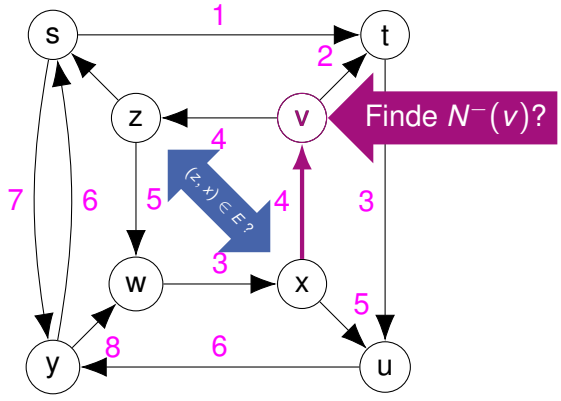
Operationen
Weitere Operationen

- Zugriff auf assoziierte Information
- mehr Navigation: finde eingehende Kanten
- Kantenanfragen: $(z, x) \in E$?
- Zugriff auf Rückkante



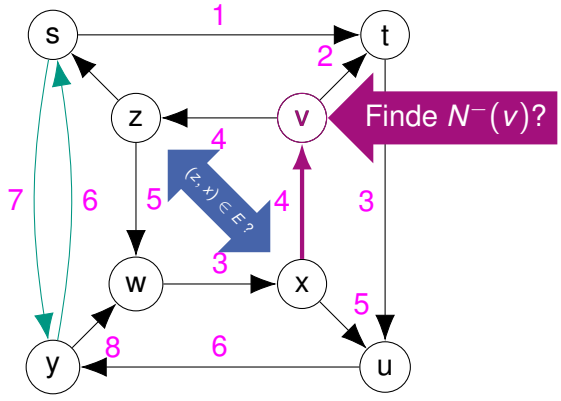
Operationen
Weitere Operationen

- Zugriff auf assoziierte Information
- mehr Navigation: finde eingehende Kanten
- Kantenanfragen: $(z, x) \in E$?
- Zugriff auf Rückkante



Operationen
Weitere Operationen

- Zugriff auf assoziierte Information
- mehr Navigation: finde eingehende Kanten
- Kantenanfragen: $(z, x) \in E$?
- Zugriff auf Rückkante



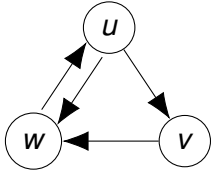
Übersicht

Kantenfolgenrepräsentation

Folge von Knotenpaaren (oder Tripel mit Kantengewicht)

- \+ kompakt
- \+ gut für I/O
- – Fast keine nützlichen Operationen außer alle Kanten durchlaufen

Beispiele: Übung: isolierte Knoten suchen, Kruskals MST-Algorithmus (später), Konvertierung.

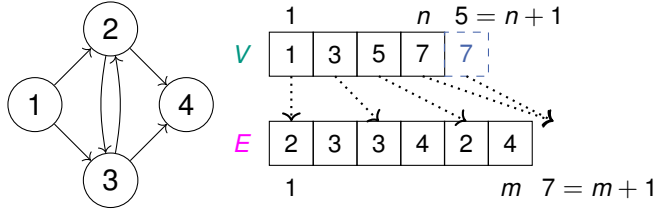


$\Leftrightarrow \langle (u,v),(v,w),(w,u),(u,w) \rangle$

Übersicht

Adjazenzfelder aka Adjazenzarray, Compressed-Sparse-Row (CSR), ...

- $V = 1..n$ oder $0..n-1$
- Kantenfeld E speichert Ziele
- gruppiert nach Startknoten
- V speichert Index der ersten ausgehenden Kante
- Dummy-Eintrag $V[n+1]$ speichert $m+1$



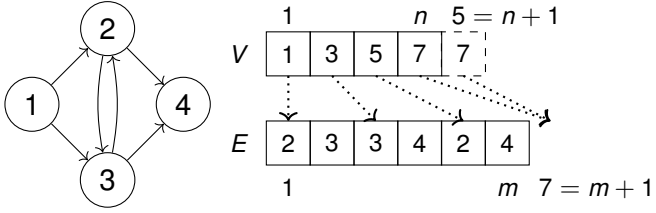
Beispiel: Ausgangsgrad$(v) = V[v+1] - V[v]$

Adjazenzfelder

Kantenfolge $\rightarrow$ Adjazenzfeld

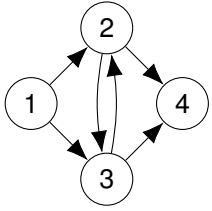
Zur Erinnerung: KSort

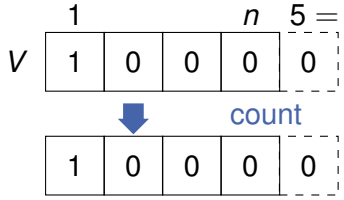
Function adjacencyArray(EdgeList)
$V=\langle 1,0,\ldots,0\rangle$: **Array** $[1..n+1]$ **of** $\mathbb{N}$
foreach $(u,v) \in$ EdgeList **do** $V[u]{+}{+}$ // count
for $v := 2$ **to** $n+1$ **do** $V[v]{+}{=}V[v-1]$ // prefix sums
foreach $(u,v) \in$ EdgeList **do** $E[--V[u]] = v$ // place
return (V,E)



Kantenliste $\rightarrow$ Adjazenzfeld

Beispiel

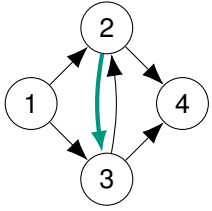


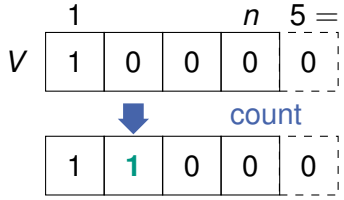


Function adjacencyArray(EdgeList)
  V=⟨1, 0, . . . , 0⟩ : **Array** [1..$n+1$] **of** $\mathbb{N}$
  foreach $(u, v) \in$ EdgeList **do** $V[u]{+}{+}$ // count
  for $v := 2$ **to** $n+1$ **do** $V[v]$+=$V[v-1]$ // prefix sums
  foreach $(u, v) \in$ EdgeList **do** $E[--V[u]] = v$ // place
  return (V, E)

Kantenliste $\rightarrow$ Adjazenzfeld

Beispiel

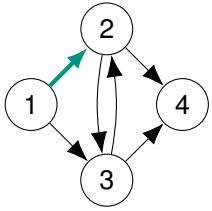


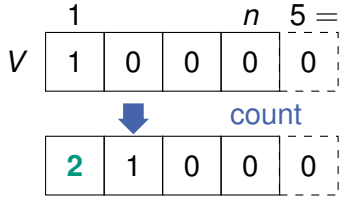


Function adjacencyArray(EdgeList)
  V=⟨1, 0, . . . , 0⟩ : **Array** [1..$n+1$] **of** $\mathbb{N}$
  foreach $(u, v) \in$ EdgeList **do** $V[u]{+}{+}$ // count
  for $v := 2$ **to** $n+1$ **do** $V[v]$+=$V[v-1]$ // prefix sums
  foreach $(u, v) \in$ EdgeList **do** $E[--V[u]] = v$ // place
  return (V, E)

Kantenliste $\rightarrow$ Adjazenzfeld

Beispiel

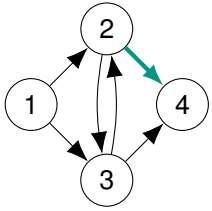


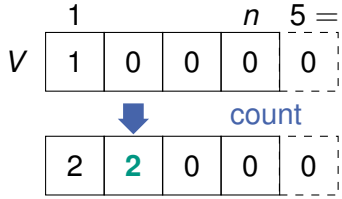


Function adjacencyArray(EdgeList)
  $V=\langle 1,0,\ldots,0\rangle$: **Array** $[1..n+1]$ **of** $\mathbb{N}$
  foreach $(u,v) \in$ EdgeList **do** $V[u]{+}{+}$ // count
  for $v := 2$ **to** $n+1$ **do** $V[v] += V[v-1]$ // prefix sums
  foreach $(u,v) \in$ EdgeList **do** $E[--V[u]] = v$ // place
  return (V,E)

Kantenliste $\rightarrow$ Adjazenzfeld

Beispiel

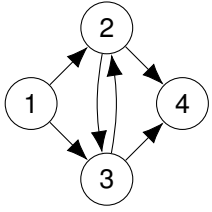


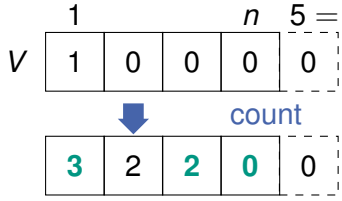



```
Function adjacencyArray(EdgeList)
    V=⟨1, 0, ..., 0⟩ : Array [1..n + 1] of ℕ
    foreach (u, v) ∈ EdgeList do V[u]++                    // count
    for v := 2 to n + 1 do  V[v]+=V[v − 1]                 // prefix sums
    foreach (u, v) ∈ EdgeList do E[−− V[u]] = v            // place
    return (V, E)
```

Kantenliste $\rightarrow$ Adjazenzfeld

Beispiel

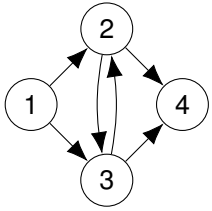


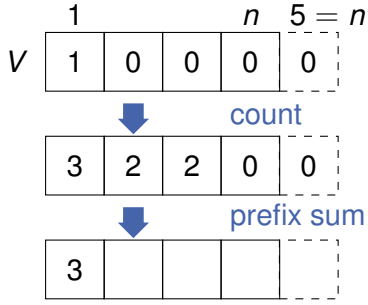


Function adjacencyArray(EdgeList)
  $V=\langle 1,0,\ldots,0\rangle$: **Array** $[1..n+1]$ **of** $\mathbb{N}$
  foreach $(u,v) \in$ EdgeList **do** $V[u]{+}{+}$ // count
  for $v := 2$ **to** $n+1$ **do** $V[v]{+}{=}V[v-1]$ // prefix sums
  foreach $(u,v) \in$ EdgeList **do** $E[--V[u]] = v$ // place
  return (V,E)

Kantenliste $\rightarrow$ Adjazenzfeld

Beispiel

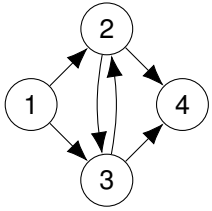


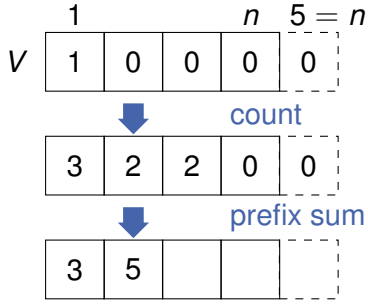


Function adjacencyArray(EdgeList)
V=⟨1, 0, . . . , 0⟩ : **Array** [1..$n+1$] **of** $\mathbb{N}$
foreach $(u, v) \in$ EdgeList **do** $V[u]$++ // count
for $v := 2$ **to** $n+1$ **do** $V[v]$+=$V[v-1]$ // prefix sums
foreach $(u, v) \in$ EdgeList **do** $E[--V[u]] = v$ // place
return (V, E)

Kantenliste $\rightarrow$ Adjazenzfeld

Beispiel

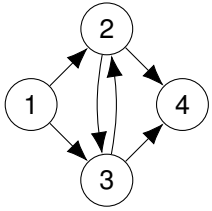


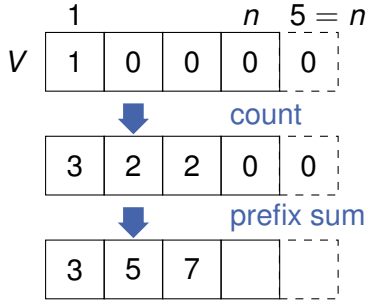


Function adjacencyArray(EdgeList)
V=⟨1, 0, . . . , 0⟩ : **Array** [1..$n+1$] **of** $\mathbb{N}$
foreach $(u, v) \in$ EdgeList **do** $V[u]$++ // count
for $v := 2$ **to** $n+1$ **do** $V[v]$+=$V[v-1]$ // prefix sums
foreach $(u, v) \in$ EdgeList **do** $E[--V[u]] = v$ // place
return (V, E)

Kantenliste $\rightarrow$ Adjazenzfeld

Beispiel

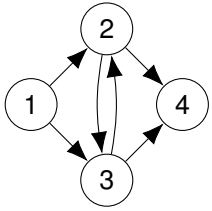


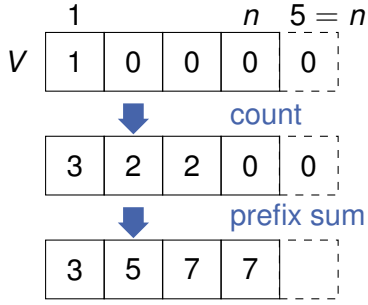


Function adjacencyArray(EdgeList)
V=⟨1, 0, . . . , 0⟩ : **Array** [1..$n+1$] **of** $\mathbb{N}$
foreach $(u, v) \in$ EdgeList **do** $V[u]$++ // count
for $v := 2$ **to** $n+1$ **do** $V[v]$+=$V[v-1]$ // prefix sums
foreach $(u, v) \in$ EdgeList **do** $E[--V[u]] = v$ // place
return (V, E)

Kantenliste $\rightarrow$ Adjazenzfeld

Beispiel

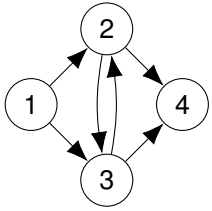


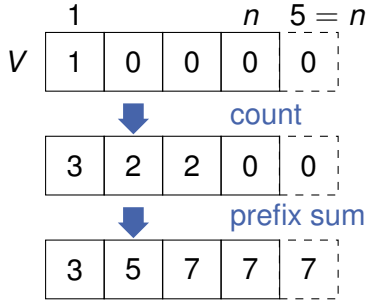


Function adjacencyArray(EdgeList)
V=⟨1, 0, . . . , 0⟩ : **Array** [1..$n+1$] **of** $\mathbb{N}$
foreach $(u, v) \in$ EdgeList **do** $V[u]$++ // count
for $v := 2$ **to** $n+1$ **do** $V[v]$+=$V[v-1]$ // prefix sums
foreach $(u, v) \in$ EdgeList **do** $E[--V[u]] = v$ // place
return (V, E)

Kantenliste $\rightarrow$ Adjazenzfeld

Beispiel

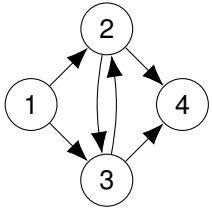




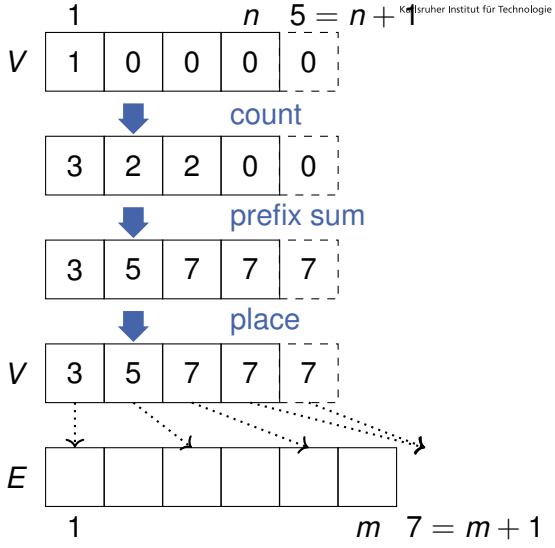
Function adjacencyArray(EdgeList)
V=⟨1, 0, . . . , 0⟩ : **Array** [1..$n+1$] **of** $\mathbb{N}$
foreach $(u, v) \in$ EdgeList **do** $V[u]$++ // count
for $v := 2$ **to** $n+1$ **do** $V[v]$+=$V[v-1]$ // prefix sums
foreach $(u, v) \in$ EdgeList **do** $E[--V[u]] = v$ // place
return (V, E)

Kantenliste $\rightarrow$ Adjazenzfeld

Beispiel

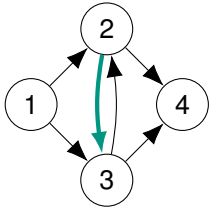


Function adjacencyArray(EdgeList)
V=$\langle 1, 0, \ldots, 0 \rangle$: **Array** $[1..n+1]$ **of** $\mathbb{N}$
foreach $(u, v) \in$ EdgeList **do** $V[u]$++ // count
for $v := 2$ **to** $n+1$ **do** $V[v]$+=$V[v-1]$ // prefix sums
foreach $(u, v) \in$ EdgeList **do** $E[--V[u]] = v$ // place
return (V, E)

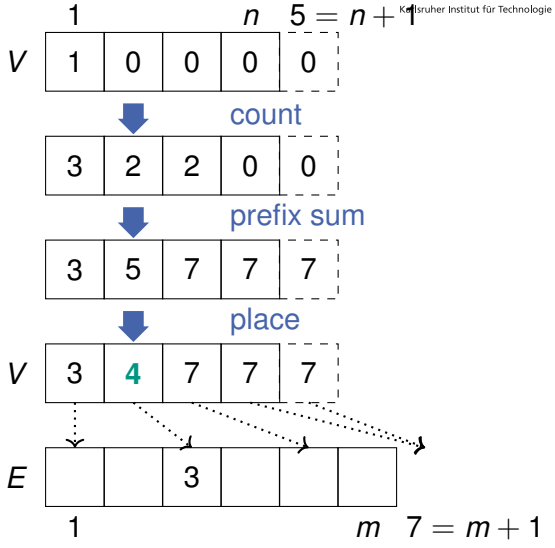


Kantenliste $\rightarrow$ Adjazenzfeld

Beispiel

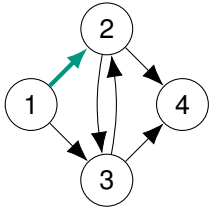


Function adjacencyArray(EdgeList)
V=$\langle 1, 0, \ldots, 0 \rangle$: **Array** $[1..n+1]$ **of** $\mathbb{N}$
foreach $(u, v) \in$ EdgeList **do** $V[u]$++ // count
for $v := 2$ **to** $n+1$ **do** $V[v]$+=$V[v-1]$ // prefix sums
foreach $(u, v) \in$ EdgeList **do** $E[--V[u]] = v$ // place
return (V, E)

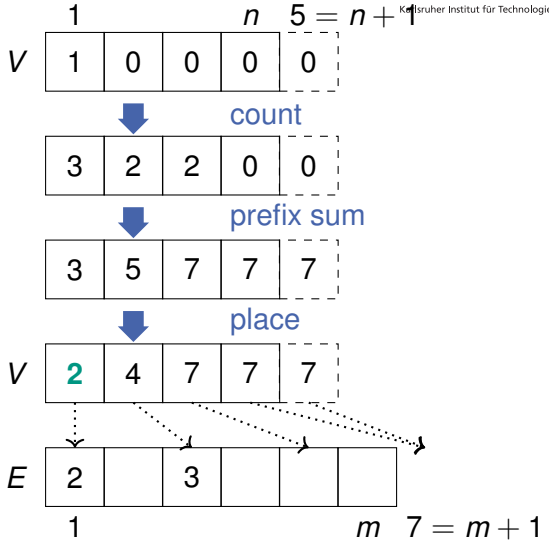


Kantenliste $\rightarrow$ Adjazenzfeld

Beispiel

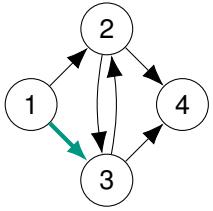


Function adjacencyArray(EdgeList)
V=$\langle 1, 0, \ldots, 0 \rangle$: **Array** $[1..n+1]$ **of** $\mathbb{N}$
foreach $(u, v) \in$ EdgeList **do** $V[u]$++ // count
for $v := 2$ **to** $n+1$ **do** $V[v]$+=$V[v-1]$ // prefix sums
foreach $(u, v) \in$ EdgeList **do** $E[--V[u]] = v$ // place
return (V, E)

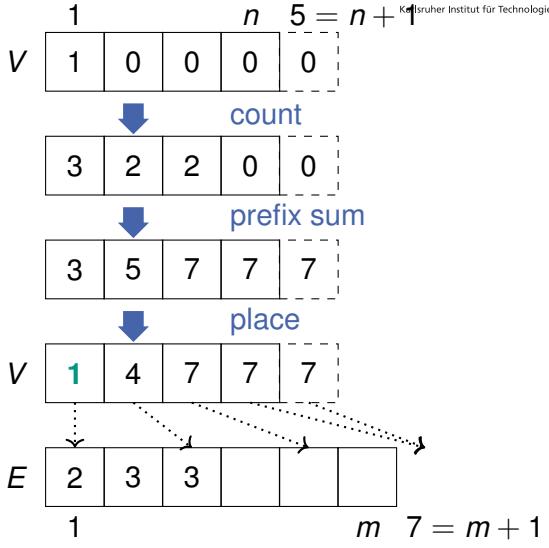


Kantenliste $\rightarrow$ Adjazenzfeld

Beispiel

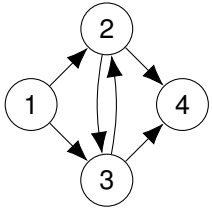


Function adjacencyArray(EdgeList)
V=$\langle 1, 0, \ldots, 0 \rangle$: **Array** $[1..n+1]$ **of** $\mathbb{N}$
foreach $(u, v) \in$ EdgeList **do** $V[u]$++ // count
for $v := 2$ **to** $n+1$ **do** $V[v]$+=$V[v-1]$ // prefix sums
foreach $(u, v) \in$ EdgeList **do** $E[--V[u]] = v$ // place
return (V, E)

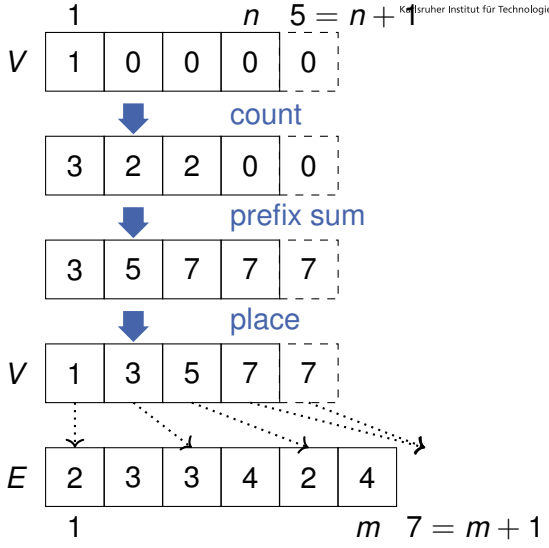


Kantenliste $\rightarrow$ Adjazenzfeld

Beispiel

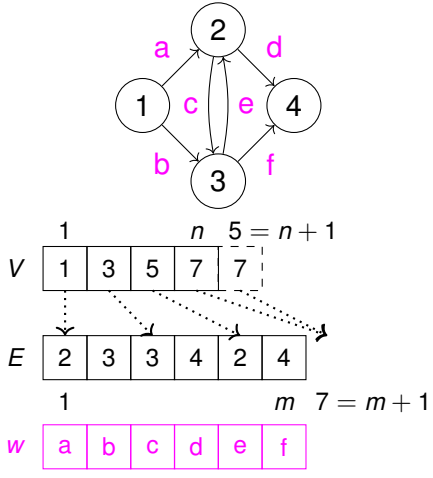



```
Function adjacencyArray(EdgeList)
    V=⟨1, 0, ..., 0⟩ : Array [1..n + 1] of ℕ
    foreach (u, v) ∈ EdgeList do V[u]++                    // count
    for v := 2 to n + 1 do  V[v]+=V[v − 1]                 // prefix sums
    foreach (u, v) ∈ EdgeList do E[−− V[u]] = v            // place
    return (V, E)
```



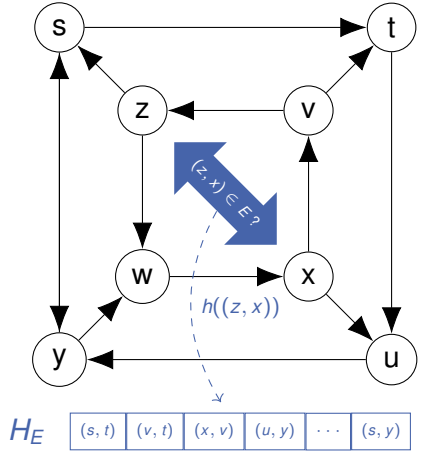
Adjanzenzfelder
Operationen

- Navigation: einfach
- Kantengewichte: E wird Feld von Records (oder mehrere Felder)
- Knoteninfos: V wird Feld von Records (oder mehrere Felder)
- Eingehende Kanten: umgedrehten Graphen speichern
- Kanten löschen: explizite Endindizes
- Batched Updates: neu aufbauen



Adjazenzfelder
Operationen – Kantenanfragen

Hashtabelle H_E speichert (ggf. zusätzlich) alle Kanten.
Unabhängig von der sonstigen Graphrepräsentation

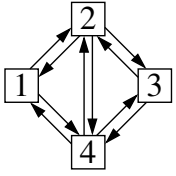


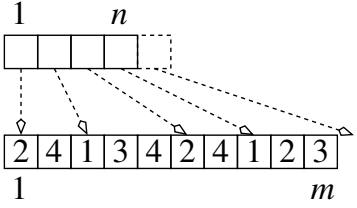
Übersicht

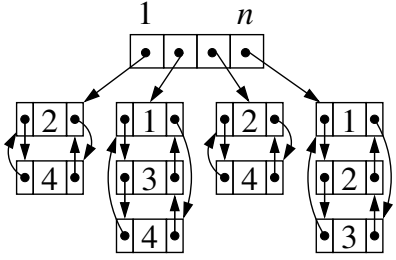
Adjazenzlisten

speichere (doppelt) verkettete Liste adjazenter Kanten für jeden Knoten.

- \+ einfaches Einfügen von Kanten
- \+ einfaches Löschen von Kanten (ordnungserhaltend)
- – mehr Platz (bis zu Faktor 3) als Adjazenzfelder
- – mehr Cache-Misses

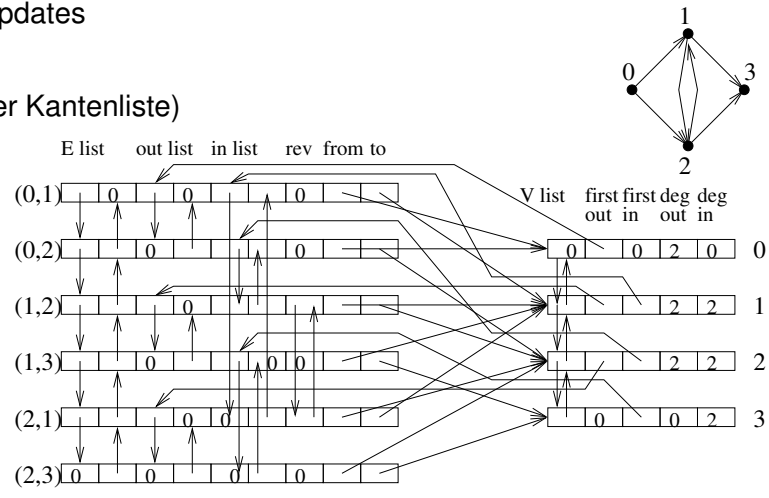






Adjazenzlisten
Aufgerüstet

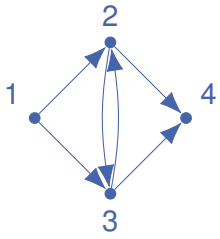
- Knotenlisten für Knotenupdates
- Eingehende Kanten
- Kantenobjekte (in globaler Kantenliste)
- Zeiger auf Umkehrkante



Übersicht

Adjazenz-Matrix

$A \in \{0,1\}^{n \times n}$ mit $A(i,j) = [(i,j) \in E]$



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

Adjazenz-Matrix

$A \in \{0,1\}^{n \times n}$ mit $A(i,j) = [(i,j) \in E]$

$+$ platzeffizient für sehr dichte Graphen

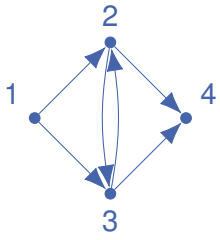
$--$ platzineffizient sonst.

Übung: was bedeutet “sehr dicht” hier?

$+$ einfache Kantenanfragen

$-$ langsame Navigation

$++$ verbindet lineare Algebra und Graphentheorie



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

Adjazenz-Matrix

$A \in \{0,1\}^{n \times n}$ mit $A(i,j) = [(i,j) \in E]$

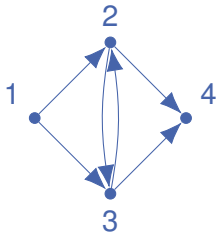
$+$ platzeffizient für sehr dichte Graphen

$--$ platzineffizient sonst. Übung: was bedeutet “sehr dicht” hier?

$+$ einfache Kantenanfragen

$-$ langsame Navigation

$++$ verbindet lineare Algebra und Graphentheorie



Beispiel: $\mathbf{C} = \mathbf{A}^k$. $\mathbf{C}_{ij} = \#\ k$-Kanten-Pfade von i nach j

Übung: zähle Pfade der Länge $\leq k$

Wichtige Beschleunigungstechniken:

$O(\log k)$ Matrixmult. für Potenzberechnung

Matrixmultiplikation in subkubischer Zeit, z. B., Strassens Algorithmus

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

Adjazenz-Matrix

Beispiel wo Graphentheorie bei LA hilft

Problem: löse $\mathbf{Bx} = \mathbf{c}$
Sei $G = (1..n, E = \{(i,j) : B_{ij} \neq 0\})$
Nehmen wir an, G habe zwei Zusammenhangskomponenten $\Rightarrow$
tausche Zeilen und Spalten so dass

$$\begin{pmatrix} \mathbf{B}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2 \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{pmatrix} .$$

Übung: Was wenn G ein DAG ist?

Übersicht

Customization (Zuschneiden)

Anpassen der (Graph)datenstruktur an die Anwendung.

- Ziel: schnell, kompakt.
- benutze Entwurfsprinzip: Make the common case fast
- Listen vermeiden

Software Engineering Alptraum

Möglicher Ausweg: Trennung von Algorithmus und Repräsentation

Customization
Beispiel: DAG-Erkennung – Abstrakter Algorithmus

Haben wir schon mal gesehen:

Ein DAG (directed acyclic graph, gerichteter azyklischer Graph) ist ein gerichteter Graph, der keine Kreise enthält.

Function isDAG($G = (V, E)$)
  while $\exists v \in V$: outdegree$(v) = 0$ **do**
    invariant G is a DAG iff the input graph is a DAG
    $V := V \setminus \{v\}$
    $E := E \setminus (\{v\} \times V \cup V \times \{v\})$
  return |V|=0

Customization

Beispiel: DAG-Erkennung – Implementierung mit Adjazenzfeldern

Function isDAG($G = (V, E)$) // Adjazenzfeld!

dropped:= 0

compute array inDegree of indegrees of all nodes // Zeit $O(m)$!

droppable=$\{v \in V : \text{inDegree}[v] = 0\}$: Stack

while droppable $\neq \emptyset$ **do**

invariant G is a DAG iff the input graph is a DAG

v:= droppable.pop

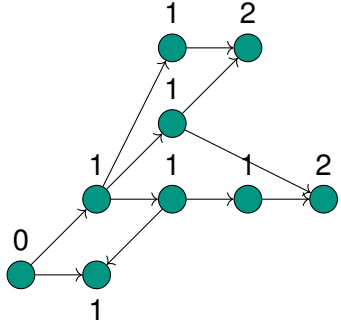
dropped++

foreach edge $(v, w) \in E$ **do**

inDegree[w]−−

if inDegree[w] $= 0$ **then** droppable.push(w)

return $|V| =$ dropped



Customization

Beispiel: DAG-Erkennung – Implementierung mit Adjazenzfeldern

Function isDAG($G = (V, E)$) // Adjazenzfeld!

dropped:= 0

compute array inDegree of indegrees of all nodes // Zeit $O(m)$!

droppable=$\{v \in V : \text{inDegree}[v] = 0\}$: Stack

while droppable $\neq \emptyset$ **do**

invariant G is a DAG iff the input graph is a DAG

v:= droppable.pop

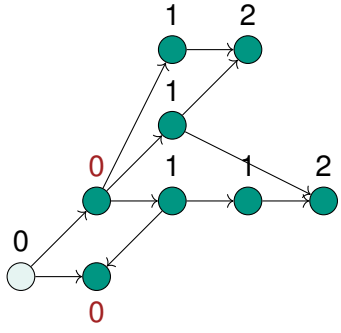
dropped++

foreach edge $(v, w) \in E$ **do**

inDegree$[w]$−−

if inDegree$[w] = 0$ **then** droppable.push(w)

return $|V| =$ dropped



Customization

Beispiel: DAG-Erkennung – Implementierung mit Adjazenzfeldern

Function isDAG($G = (V, E)$) // Adjazenzfeld!

dropped:= 0

compute array inDegree of indegrees of all nodes // Zeit $O(m)$!

droppable=$\{v \in V : \text{inDegree}[v] = 0\}$: Stack

while droppable $\neq \emptyset$ **do**

invariant G is a DAG iff the input graph is a DAG

v:= droppable.pop

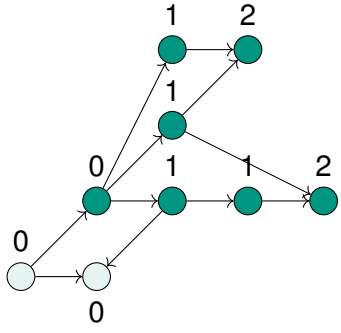
dropped++

foreach edge $(v, w) \in E$ **do**

inDegree[w]−−

if inDegree[w] $= 0$ **then** droppable.push(w)

return $|V| =$ dropped



Customization

Beispiel: DAG-Erkennung – Implementierung mit Adjazenzfeldern

Function isDAG($G = (V, E)$) // Adjazenzfeld!

dropped:= 0

compute array inDegree of indegrees of all nodes // Zeit $O(m)$!

droppable=$\{v \in V : \text{inDegree}[v] = 0\}$: Stack

while droppable $\neq \emptyset$ **do**

invariant G is a DAG iff the input graph is a DAG

v:= droppable.pop

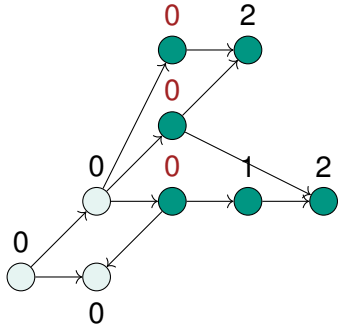
dropped++

foreach edge $(v, w) \in E$ **do**

inDegree[w]−−

if inDegree[w] $= 0$ **then** droppable.push(w)

return $|V|$ = dropped



Customization

Beispiel: DAG-Erkennung – Implementierung mit Adjazenzfeldern

Function isDAG($G = (V, E)$) // Adjazenzfeld!

dropped:= 0

compute array inDegree of indegrees of all nodes // Zeit $O(m)$!

droppable=$\{v \in V : \text{inDegree}[v] = 0\}$: Stack

while droppable $\neq \emptyset$ **do**

invariant G is a DAG iff the input graph is a DAG

v:= droppable.pop

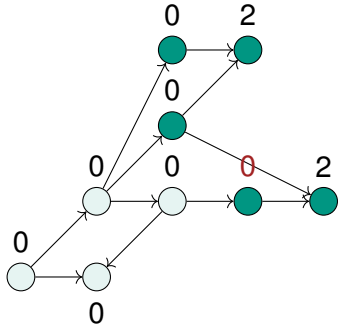
dropped++

foreach edge $(v, w) \in E$ **do**

inDegree[w]−−

if inDegree[w] $= 0$ **then** droppable.push(w)

return $|V|$ = dropped



Customization

Beispiel: DAG-Erkennung – Implementierung mit Adjazenzfeldern

Function isDAG($G = (V, E)$) // Adjazenzfeld!

dropped:= 0

compute array inDegree of indegrees of all nodes // Zeit $O(m)$!

droppable=$\{v \in V : \text{inDegree}[v] = 0\}$: Stack

while droppable $\neq \emptyset$ **do**

invariant G is a DAG iff the input graph is a DAG

v:= droppable.pop

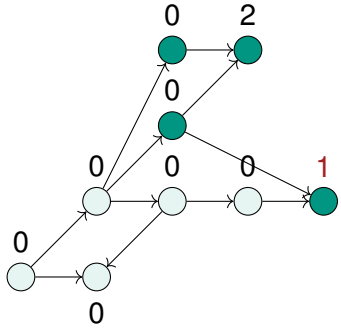
dropped++

foreach edge $(v, w) \in E$ **do**

inDegree[w]−−

if inDegree[w] $= 0$ **then** droppable.push(w)

return $|V|$ = dropped



Customization

Beispiel: DAG-Erkennung – Implementierung mit Adjazenzfeldern

Function isDAG($G = (V, E)$) // Adjazenzfeld!

dropped:= 0

compute array inDegree of indegrees of all nodes // Zeit $O(m)$!

droppable=$\{v \in V : \text{inDegree}[v] = 0\}$: Stack

while droppable $\neq \emptyset$ **do**

invariant G is a DAG iff the input graph is a DAG

v:= droppable.pop

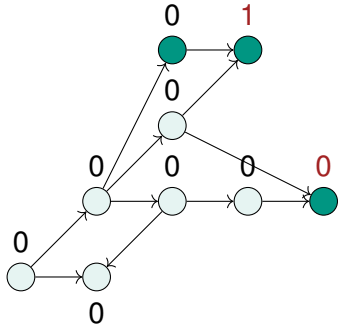
dropped++

foreach edge $(v, w) \in E$ **do**

inDegree[w]−−

if inDegree[w] $= 0$ **then** droppable.push(w)

return $|V|$ = dropped



Customization

Beispiel: DAG-Erkennung – Implementierung mit Adjazenzfeldern

Function isDAG($G = (V, E)$) // Adjazenzfeld!

dropped:= 0

compute array inDegree of indegrees of all nodes // Zeit $O(m)$!

droppable=$\{v \in V : \text{inDegree}[v] = 0\}$: Stack

while droppable $\neq \emptyset$ **do**

invariant G is a DAG iff the input graph is a DAG

v:= droppable.pop

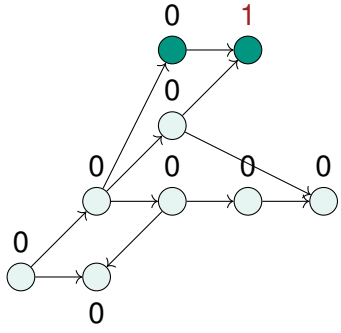
dropped++

foreach edge $(v, w) \in E$ **do**

inDegree[w]−−

if inDegree[w] $= 0$ **then** droppable.push(w)

return $|V|$ = dropped



Customization

Beispiel: DAG-Erkennung – Implementierung mit Adjazenzfeldern

Function isDAG($G = (V, E)$) // Adjazenzfeld!

dropped:= 0

compute array inDegree of indegrees of all nodes // Zeit $O(m)$!

droppable=$\{v \in V : \text{inDegree}[v] = 0\}$: Stack

while droppable $\neq \emptyset$ **do**

invariant G is a DAG iff the input graph is a DAG

v:= droppable.pop

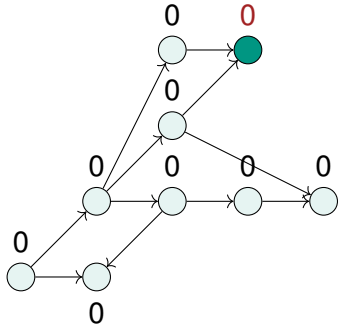
dropped++

foreach edge $(v, w) \in E$ **do**

inDegree[w]−−

if inDegree[w] $= 0$ **then** droppable.push(w)

return $|V|$ = dropped



Customization

Beispiel: DAG-Erkennung – Implementierung mit Adjazenzfeldern

Function isDAG($G = (V, E)$) // Adjazenzfeld!

dropped:= 0

compute array inDegree of indegrees of all nodes // Zeit $O(m)$!

droppable=$\{v \in V : \text{inDegree}[v] = 0\}$: Stack

while droppable $\neq \emptyset$ **do**

invariant G is a DAG iff the input graph is a DAG

v:= droppable.pop

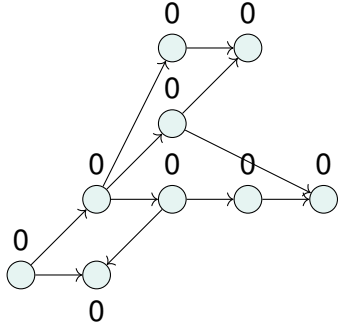
dropped++

foreach edge $(v, w) \in E$ **do**

inDegree$[w]$−−

if inDegree$[w] = 0$ **then** droppable.push(w)

return $|V|$ = dropped



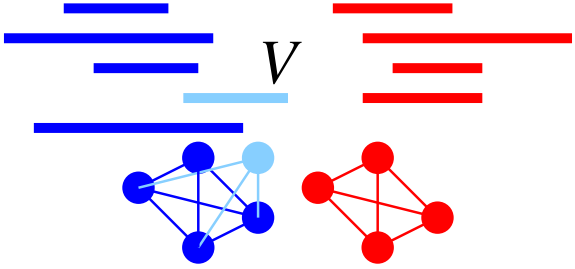
Übersicht

Implizite Repräsentation

Kompakte Repräsentation möglicherweise sehr dichter Graphen
Implementiere Algorithmen direkt mittels dieser Repräsentation

Beispiel: Intervall-Graphen

- Knoten: Intervalle $[a, b] \subseteq \mathbb{R}$
- Kanten: zwischen überlappenden Intervallen



Implizite Repräsentation

Beispiel: Zusammenhangstest für Intervallgraphen

$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$

$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$

Idee: durchlaufe Intervalle von links nach rechts.

Die Anzahl überlappender Intervalle darf nie auf Null sinken.

Implizite Repräsentation

Beispiel: Zusammenhangstest für Intervallgraphen

$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$
$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$
Idee: durchlaufe Intervalle von links nach rechts.
Die Anzahl überlappender Intervalle darf nie auf Null sinken.

```
Function isConnected(L : SortedListOfIntervalEndPoints) : {0, 1}
    remove first element of L;   overlap := 1
    foreach p ∈ L do
        if overlap= 0 return 0
        if p is a start point then overlap++
        else overlap−−                          // end point
    return 1
```

Implizite Repräsentation

Beispiel: Zusammenhangstest für Intervallgraphen

$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$

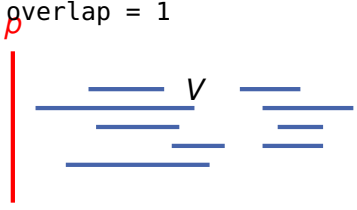
$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$

Idee: durchlaufe Intervalle von links nach rechts.

Die Anzahl überlappender Intervalle darf nie auf Null sinken.

Function isConnected(L : SortedListOfIntervalEndPoints) : $\{0, 1\}$

    remove first element of L; overlap := 1
    foreach $p \in L$ **do**
        if overlap$= 0$ **return** 0
        if p is a start point **then** overlap$++$
        else overlap$--$ // end point
    return 1



Implizite Repräsentation

Beispiel: Zusammenhangstest für Intervallgraphen

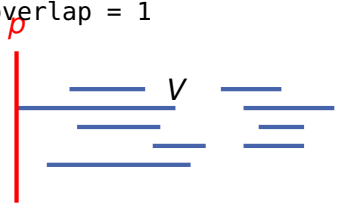
$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$

$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$

Idee: durchlaufe Intervalle von links nach rechts.

Die Anzahl überlappender Intervalle darf nie auf Null sinken.

```
Function isConnected(L : SortedListOfIntervalEndPoints) : {0, 1}
    remove first element of L;   overlap := 1
    foreach p ∈ L do
        if overlap= 0 return 0
        if p is a start point then overlap++
        else overlap−−                          // end point
    return 1
```



Implizite Repräsentation

Beispiel: Zusammenhangstest für Intervallgraphen

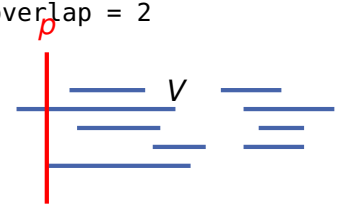
$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$

$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$

Idee: durchlaufe Intervalle von links nach rechts.

Die Anzahl überlappender Intervalle darf nie auf Null sinken.

Function isConnected(L : SortedListOfIntervalEndPoints) : $\{0, 1\}$
    remove first element of L; overlap := 1
    foreach $p \in L$ **do**
        if overlap$=0$ **return** 0
        if p is a start point **then** overlap$++$
        else overlap$--$ // end point
    return 1



Implizite Repräsentation

Beispiel: Zusammenhangstest für Intervallgraphen

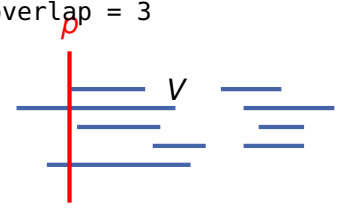
$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$

$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$

Idee: durchlaufe Intervalle von links nach rechts.

Die Anzahl überlappender Intervalle darf nie auf Null sinken.

```
Function isConnected(L : SortedListOfIntervalEndPoints) : {0, 1}
    remove first element of L;   overlap := 1
    foreach p ∈ L do
        if overlap= 0 return 0
        if p is a start point then overlap++
        else overlap−−                          // end point
    return 1
```



Implizite Repräsentation

Beispiel: Zusammenhangstest für Intervallgraphen

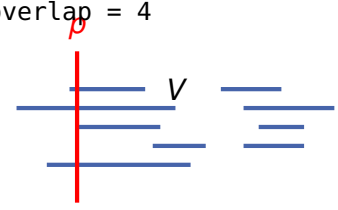
$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$

$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$

Idee: durchlaufe Intervalle von links nach rechts.

Die Anzahl überlappender Intervalle darf nie auf Null sinken.

```
Function isConnected(L : SortedListOfIntervalEndPoints) : {0, 1}
    remove first element of L;   overlap := 1
    foreach p ∈ L do
        if overlap= 0 return 0
        if p is a start point then overlap++
        else overlap−−                          // end point
    return 1
```



Implizite Repräsentation

Beispiel: Zusammenhangstest für Intervallgraphen

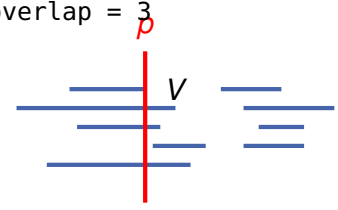
$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$

$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$

Idee: durchlaufe Intervalle von links nach rechts.

Die Anzahl überlappender Intervalle darf nie auf Null sinken.

```
Function isConnected(L : SortedListOfIntervalEndPoints) : {0, 1}
    remove first element of L;   overlap := 1
    foreach p ∈ L do
        if overlap= 0 return 0
        if p is a start point then overlap++
        else overlap−−                          // end point
    return 1
```



Implizite Repräsentation

Beispiel: Zusammenhangstest für Intervallgraphen

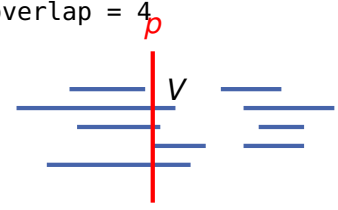
$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$

$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$

Idee: durchlaufe Intervalle von links nach rechts.

Die Anzahl überlappender Intervalle darf nie auf Null sinken.

Function isConnected(L : SortedListOfIntervalEndPoints) : $\{0, 1\}$
    remove first element of L; overlap := 1
    foreach $p \in L$ **do**
        if overlap$=0$ **return** 0
        if p is a start point **then** overlap$++$
        else overlap$--$ // end point
    return 1



Implizite Repräsentation

Beispiel: Zusammenhangstest für Intervallgraphen

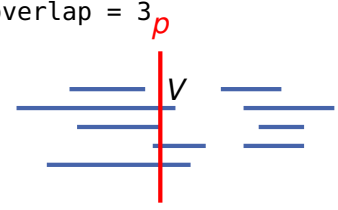
$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$

$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$

Idee: durchlaufe Intervalle von links nach rechts.

Die Anzahl überlappender Intervalle darf nie auf Null sinken.

```
Function isConnected(L : SortedListOfIntervalEndPoints) : {0, 1}
    remove first element of L;   overlap := 1
    foreach p ∈ L do
        if overlap= 0 return 0
        if p is a start point then overlap++
        else overlap−−                          // end point
    return 1
```



Implizite Repräsentation

Beispiel: Zusammenhangstest für Intervallgraphen

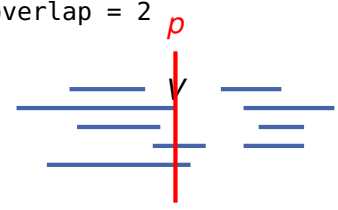
$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$

$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$

Idee: durchlaufe Intervalle von links nach rechts.

Die Anzahl überlappender Intervalle darf nie auf Null sinken.

```
Function isConnected(L : SortedListOfIntervalEndPoints) : {0, 1}
    remove first element of L;   overlap := 1
    foreach p ∈ L do
        if overlap= 0 return 0
        if p is a start point then overlap++
        else overlap−−                          // end point
    return 1
```



Implizite Repräsentation

Beispiel: Zusammenhangstest für Intervallgraphen

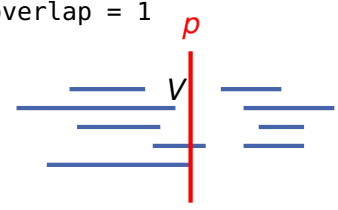
$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$

$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$

Idee: durchlaufe Intervalle von links nach rechts.

Die Anzahl überlappender Intervalle darf nie auf Null sinken.

```
Function isConnected(L : SortedListOfIntervalEndPoints) : {0, 1}
    remove first element of L;   overlap := 1
    foreach p ∈ L do
        if overlap= 0 return 0
        if p is a start point then overlap++
        else overlap−−                          // end point
    return 1
```



Implizite Repräsentation

Beispiel: Zusammenhangstest für Intervallgraphen

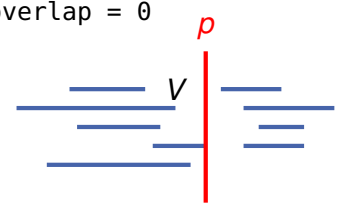
$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$

$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$

Idee: durchlaufe Intervalle von links nach rechts.

Die Anzahl überlappender Intervalle darf nie auf Null sinken.

```
Function isConnected(L : SortedListOfIntervalEndPoints) : {0, 1}
    remove first element of L;   overlap := 1
    foreach p ∈ L do
        if overlap= 0 return 0
        if p is a start point then overlap++
        else overlap−−                          // end point
    return 1
```



Implizite Repräsentation

Beispiel: Zusammenhangstest für Intervallgraphen

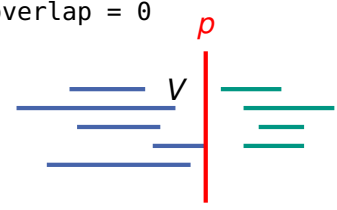
$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$

$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$

Idee: durchlaufe Intervalle von links nach rechts.

Die Anzahl überlappender Intervalle darf nie auf Null sinken.

```
Function isConnected(L : SortedListOfIntervalEndPoints) : {0, 1}
    remove first element of L;   overlap := 1
    foreach p ∈ L do
        if overlap= 0 return 0
        if p is a start point then overlap++
        else overlap−−                          // end point
    return 1
```

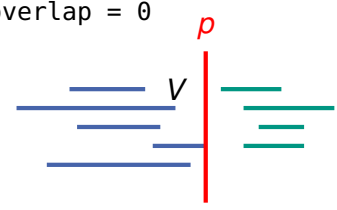


Implizite Repräsentation

Beispiel: Zusammenhangstest für Intervallgraphen

$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$
$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$
Idee: durchlaufe Intervalle von links nach rechts.
Die Anzahl überlappender Intervalle darf nie auf Null sinken.

Function isConnected(L : SortedListOfIntervalEndPoints) : $\{0, 1\}$
    remove first element of L; overlap := 1
    foreach $p \in L$ **do**
        if overlap$= 0$ **return** 0
        if p is a start point **then** overlap$++$
        else overlap$--$ // end point
    return 1



$O(n \log n)$ Algorithmus für bis zu $O(n^2)$ Kanten!
Übung: Zusammenhangskomponenten finden

Übersicht

Graphrepräsentation: Zusammenfassung

- Welche Operationen werden gebraucht?
- Wie oft?
- Adjazenzarrays gut für statische Graphen
- Trenne Repräsentation und Algorithmus
- Pointer $\rightsquigarrow$ flexibler aber auch teurer
- Matrizen eher konzeptionell interessant

Implementierungen von Graphrepräsentationen

Oft macht jeder sein eigenes “Ding”

C++:

BGL: Boost Graph Library – mächtig und flexibel, schwer richtig einzusetzen

LEDA: Elegant und mächtig aber langsam (eierlegende Wollmilchsau)

Rust:

petrgraph: Etwas weniger flexibel als die BGL. Rust’s Trait-System erlaubt eine elegante Trennung von Interface und Implementierung

Java:

JGraphT: verschiedene Repräsentation, anpassbar, Algorithmen

Neo4J: graph database (Query-Sprache, Persistenz,...)

Python:

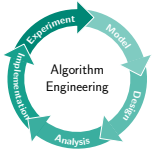
NetworkX: Einfach zu benutzen aber langsam

NetworKit: (AG Meyerhenke) Flexibel. Schnell wegen C++-Backend

(Dank an Niklas Uhl)

Index

Karlsruher Institut für Technologie



Algorithmen I – 9. Graphtraversierung

Sommersemester 2025

Peter Sanders | Stand: 30. Juli 2025

Graphtraversierung

Ausgangspunkt oder Baustein fast jedes nichtrivialen Graphenalgorithmus
$\rightarrow$ laufe systematisch von einem Knoten v den gesamten Graphen ab

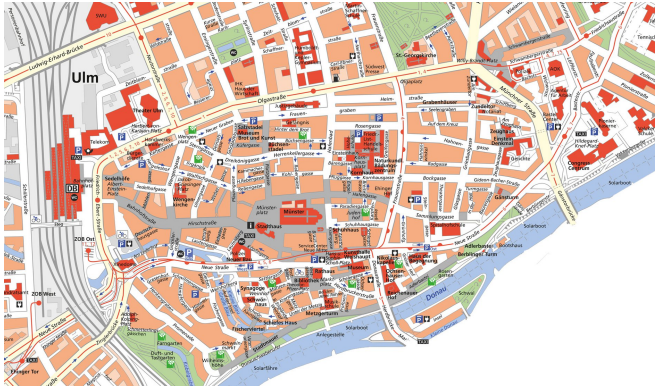
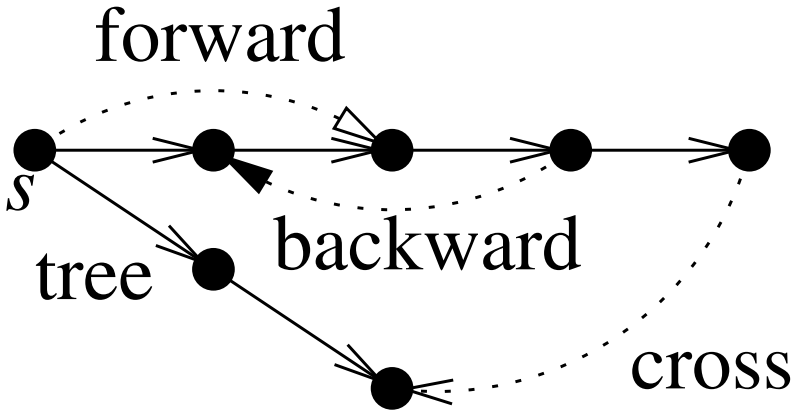


Abbildung: Einbahnstraßen in Ulm

Konkrete Anwendung: Zusammenhangskomponenten finden

Abstrakte Sicht

Graphtraversierung als Kantenklassifizierung

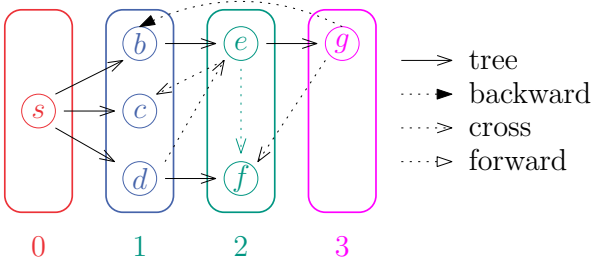


Übersicht

Breitensuche engl. Breadth-First-Search (BFS)

Problemstellung

Baue Baum von Startknoten s der alle von s erreichbaren Knoten mit möglichst kurzen Pfaden erreicht. Berechne Abstände

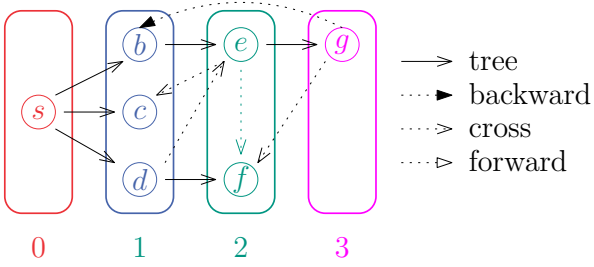


Breitensuche engl. Breadth-First-Search (BFS)

Problemstellung

Baue Baum von Startknoten s der alle von s erreichbaren Knoten mit möglichst kurzen Pfaden erreicht. Berechne Abstände

- Einfachste Form des kürzeste Wege Problems
- Umgebung eines Knotens definieren (ggf. begrenzte Suchtiefe)
- Einfache, effiziente Graphtraversierung (auch wenn Reihenfolge egal)

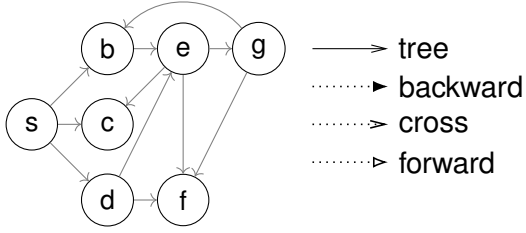


Breitensuche

Algorithmenidee

Idee: Baum Schicht für Schicht aufbauen

Function bfs(s) :
    $Q := \langle s \rangle$ // aktuelle Schicht
    while $Q \neq \langle\rangle$ **do**
        exploriere Knoten in Q
        merke dir Knoten der nächsten Schicht in Q'
        $Q := Q'$

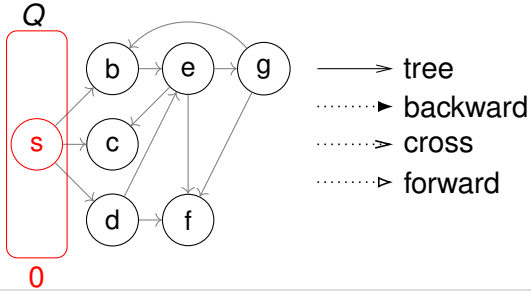


Breitensuche

Algorithmenidee

Idee: Baum Schicht für Schicht aufbauen

Function bfs(s) :
    $Q := \langle s \rangle$ // aktuelle Schicht
    while $Q \neq \langle\rangle$ **do**
        exploriere Knoten in Q
        merke dir Knoten der nächsten Schicht in Q'
        $Q := Q'$



Breitensuche

Algorithmenidee

Idee: Baum Schicht für Schicht aufbauen

Function bfs(s) :

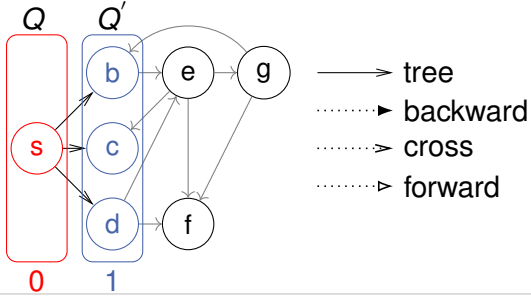
$Q := \langle s \rangle$ // aktuelle Schicht

while $Q \neq \langle\rangle$ **do**

exploriere Knoten in Q

merke dir Knoten der nächsten Schicht in Q'

$Q := Q'$

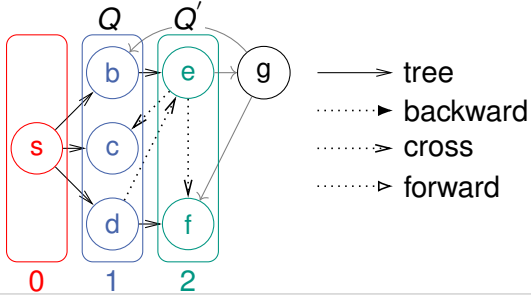


Breitensuche

Algorithmenidee

Idee: Baum Schicht für Schicht aufbauen

Function bfs(s) :
    $Q := \langle s \rangle$ // aktuelle Schicht
    while $Q \neq \langle\rangle$ **do**
        exploriere Knoten in Q
        merke dir Knoten der nächsten Schicht in Q'
        $Q := Q'$

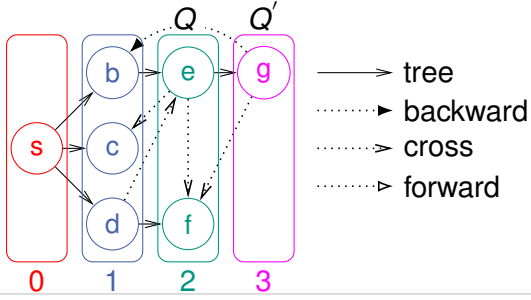


Breitensuche

Algorithmenidee

Idee: Baum Schicht für Schicht aufbauen

Function bfs(s) :
    $Q := \langle s \rangle$ // aktuelle Schicht
    while $Q \neq \langle\rangle$ **do**
        exploriere Knoten in Q
        merke dir Knoten der nächsten Schicht in Q'
        $Q := Q'$

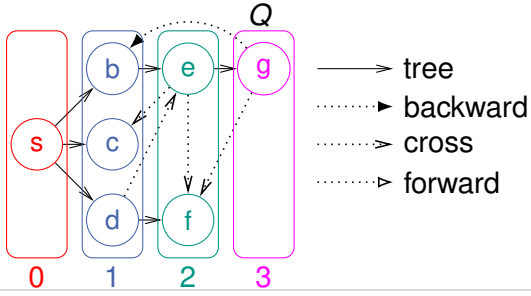


Breitensuche

Algorithmenidee

Idee: Baum Schicht für Schicht aufbauen

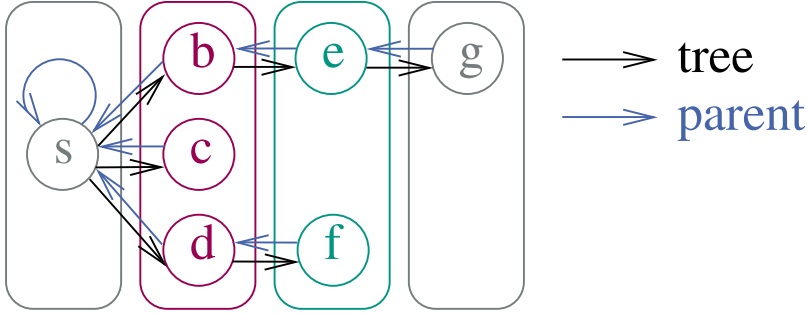
Function bfs(s) :
    $Q := \langle s \rangle$ // aktuelle Schicht
    while $Q \neq \langle\rangle$ **do**
        exploriere Knoten in Q
        merke dir Knoten der nächsten Schicht in Q'
        $Q := Q'$



Breitensuche
Repräsentation des Baums

Feld parent speichert Vorgänger.

- noch nicht erreicht: parent$[v] = \perp$
- Startknoten/Wurzel: parent$[s] = s$



Breitensuche
Der Algorithmus

Function bfs(s : NodeId) : (NodeArray **of** NodeId) $\times$ (NodeArray **of** $\mathbb{N}_0 \cup \{\infty\}$)
  $d = \langle \infty, \ldots, \infty \rangle$: NodeArray **of** $\mathbb{N}_0 \cup \{\infty\}$; $d[s] := 0$
  parent $= \langle \bot, \ldots, \bot \rangle$: NodeArray **of** NodeId; parent$[s] := s$
  $Q = \langle s \rangle, Q' = \langle\rangle$: Set **of** NodeId // current, next layer
  for ($\ell := 0$; $Q \neq \langle\rangle$; $\ell{+}{+}$)
    invariant Q contains all nodes with distance ℓ from s
    foreach $u \in Q$ **do**
      foreach $(u, v) \in E$ **do** // scan u
        if parent$(v) = \bot$ **then** // unexplored
          $Q' := Q' \cup \{v\}$
          $d[v] := \ell + 1$; parent$(v) := u$
    $(Q, Q') := (Q', \langle\rangle)$ // next layer
  return (parent, d) // BFS $= \{(v, w) : w \in V, v = \text{parent}(w)\}$

Breitensuche

Repräsentation von Q und Q'

- Zwei Stapel
- Schleife $1\times$ ausrollen
  loop $Q \longrightarrow Q'$; $Q' \longrightarrow Q$
- Beide Stapel in ein Feld der Größe n

Breitensuche

Repräsentation von Q und Q'

- Zwei Stapel
- Schleife $1\times$ ausrollen
  loop $Q \longrightarrow Q'; Q' \longrightarrow Q$
- Beide Stapel in ein Feld der Größe n

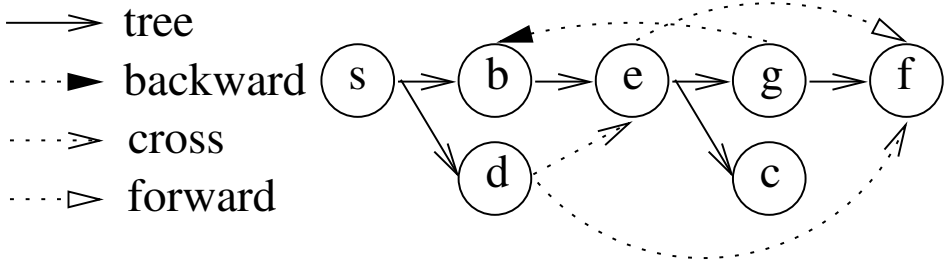
BFS mittels FIFO

$Q, Q' \longrightarrow$ einzelne FIFO Queue

- Standardimplementierung in anderen Büchern
- $+$ „Oberflächlich“ einfacher
- $-$ Korrektheit weniger evident — Übung?
- $-$ Parallelisierung schwieriger
- $=$ Effizient (?) — Übung: ausprobieren?

Übersicht

Tiefensuche engl. Depth-First-Search (DFS)

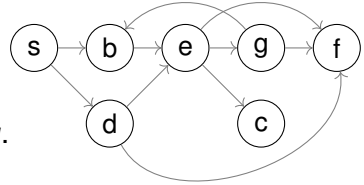


Tiefensuche

Tiefensuchschema für $G = (V, E)$

```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s  // make s a root and grow
        root(s)  // a new DFS-tree rooted at it.
        DFS(s, s)

Procedure DFS(u, v : NodeId) // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else      traverseTreeEdge(v, w)
                  mark w
                  DFS(v, w)
    backtrack(u, v) // return from v along the incoming edge
```

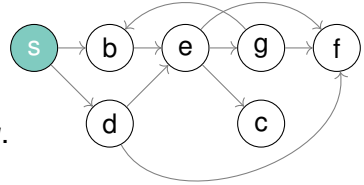


Tiefensuche

Tiefensuchschema für $G = (V, E)$

```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s  // make s a root and grow
        root(s)  // a new DFS-tree rooted at it.
        DFS(s, s)

Procedure DFS(u, v : NodeId) // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else      traverseTreeEdge(v, w)
                  mark w
                  DFS(v, w)
    backtrack(u, v) // return from v along the incoming edge
```

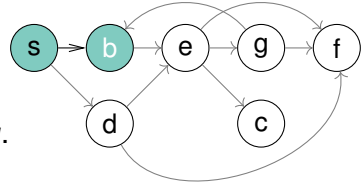


Tiefensuche

Tiefensuchschema für $G = (V, E)$

```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s  // make s a root and grow
        root(s)  // a new DFS-tree rooted at it.
        DFS(s, s)

Procedure DFS(u, v : NodeId) // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else      traverseTreeEdge(v, w)
                  mark w
                  DFS(v, w)
    backtrack(u, v) // return from v along the incoming edge
```

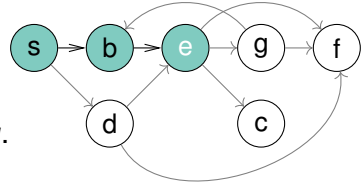


Tiefensuche

Tiefensuchschema für $G = (V, E)$

```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s  // make s a root and grow
        root(s)  // a new DFS-tree rooted at it.
        DFS(s, s)

Procedure DFS(u, v : NodeId) // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else      traverseTreeEdge(v, w)
                  mark w
                  DFS(v, w)
    backtrack(u, v) // return from v along the incoming edge
```

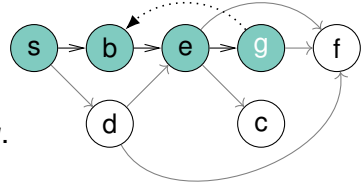


Tiefensuche

Tiefensuchschema für $G = (V, E)$

```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s  // make s a root and grow
        root(s)  // a new DFS-tree rooted at it.
        DFS(s, s)

Procedure DFS(u, v : NodeId) // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else      traverseTreeEdge(v, w)
                  mark w
                  DFS(v, w)
    backtrack(u, v) // return from v along the incoming edge
```

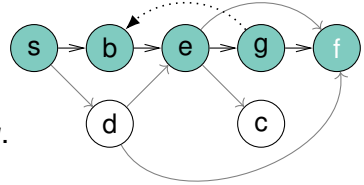


Tiefensuche

Tiefensuchschema für $G = (V, E)$

```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s  // make s a root and grow
        root(s)  // a new DFS-tree rooted at it.
        DFS(s, s)

Procedure DFS(u, v : NodeId) // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else      traverseTreeEdge(v, w)
                  mark w
                  DFS(v, w)
    backtrack(u, v) // return from v along the incoming edge
```

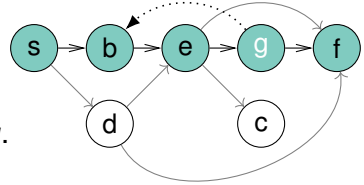


Tiefensuche

Tiefensuchschema für $G = (V, E)$

```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s  // make s a root and grow
        root(s)  // a new DFS-tree rooted at it.
        DFS(s, s)

Procedure DFS(u, v : NodeId) // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else      traverseTreeEdge(v, w)
                  mark w
                  DFS(v, w)
    backtrack(u, v) // return from v along the incoming edge
```

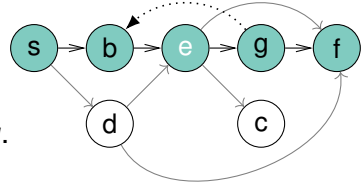


Tiefensuche

Tiefensuchschema für $G = (V, E)$

```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s  // make s a root and grow
        root(s)  // a new DFS-tree rooted at it.
        DFS(s, s)

Procedure DFS(u, v : NodeId) // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else      traverseTreeEdge(v, w)
                  mark w
                  DFS(v, w)
    backtrack(u, v) // return from v along the incoming edge
```

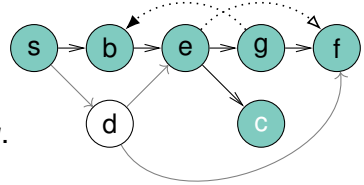


Tiefensuche

Tiefensuchschema für $G = (V, E)$

```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s  // make s a root and grow
        root(s)  // a new DFS-tree rooted at it.
        DFS(s, s)

Procedure DFS(u, v : NodeId) // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else      traverseTreeEdge(v, w)
                  mark w
                  DFS(v, w)
    backtrack(u, v) // return from v along the incoming edge
```

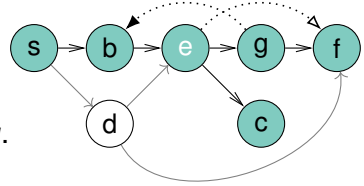


Tiefensuche

Tiefensuchschema für $G = (V, E)$

```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s  // make s a root and grow
        root(s)  // a new DFS-tree rooted at it.
        DFS(s, s)

Procedure DFS(u, v : NodeId) // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else      traverseTreeEdge(v, w)
                  mark w
                  DFS(v, w)
    backtrack(u, v) // return from v along the incoming edge
```

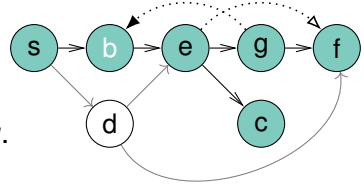


Tiefensuche

Tiefensuchschema für $G = (V, E)$

```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s  // make s a root and grow
        root(s)  // a new DFS-tree rooted at it.
        DFS(s, s)

Procedure DFS(u, v : NodeId) // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else      traverseTreeEdge(v, w)
                  mark w
                  DFS(v, w)
    backtrack(u, v) // return from v along the incoming edge
```

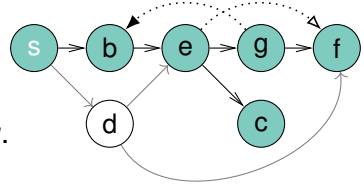


Tiefensuche

Tiefensuchschema für $G = (V, E)$

```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s  // make s a root and grow
        root(s)  // a new DFS-tree rooted at it.
        DFS(s, s)

Procedure DFS(u, v : NodeId) // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else      traverseTreeEdge(v, w)
                  mark w
                  DFS(v, w)
    backtrack(u, v) // return from v along the incoming edge
```

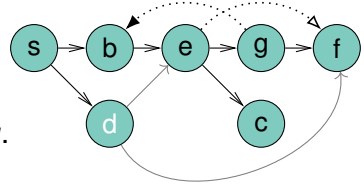


Tiefensuche

Tiefensuchschema für $G = (V, E)$

```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s  // make s a root and grow
        root(s)  // a new DFS-tree rooted at it.
        DFS(s, s)

Procedure DFS(u, v : NodeId) // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else      traverseTreeEdge(v, w)
                  mark w
                  DFS(v, w)
    backtrack(u, v) // return from v along the incoming edge
```

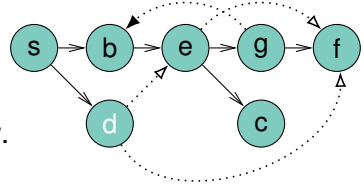


Tiefensuche

Tiefensuchschema für $G = (V, E)$

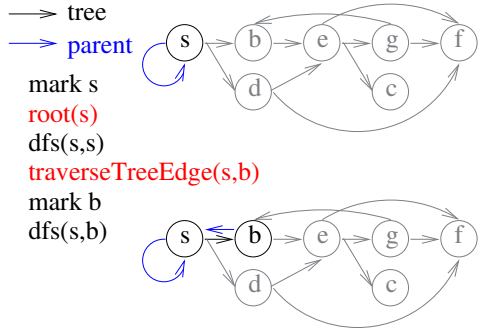
```
unmark all nodes;   init
foreach s ∈ V do
    if s is not marked then
        mark s  // make s a root and grow
        root(s)  // a new DFS-tree rooted at it.
        DFS(s, s)

Procedure DFS(u, v : NodeId) // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else      traverseTreeEdge(v, w)
                  mark w
                  DFS(v, w)
    backtrack(u, v) // return from v along the incoming edge
```



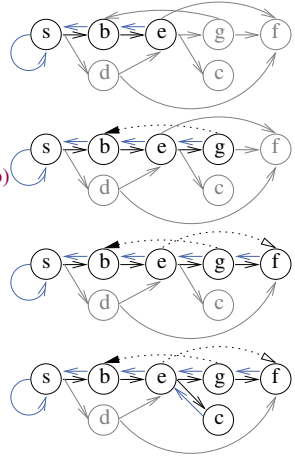
Tiefensuche
DFS Baum

init: parent=$\langle \perp, \ldots, \perp \rangle$: NodeArray **of** NodeId
root(s): parent[s]:= s
traverseTreeEdge(v, w): parent[w]:= v



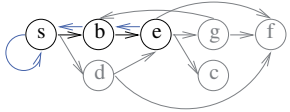
Tiefensuche
Beispiel

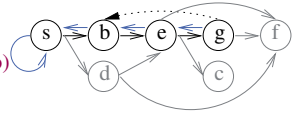
```
dfs(s,b)
  traverseTreeEdge(b,e)
  mark(e)
  dfs(b,e)
    traverseTreeEdge(e,g)
    mark(g)
    dfs(e,g)
      traverseNonTreeEdge(g,b)
      traverseTreeEdge(g,f)
      mark(f)
      dfs(g,f)
        backtrack(g,f)
      backtrack(e,g)
    traverseNonTreeEdge(e,f)
    traverseTreeEdge(e,c)
    mark(c)
    dfs(e,c)
      backtrack(e,c)
    backtrack(b,e)
  backtrack(s,b)
```

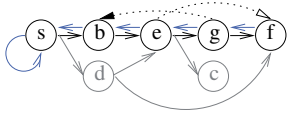


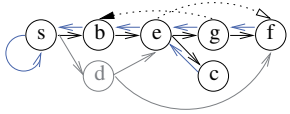
Tiefensuche
Beispiel

dfs(s,b)
  traverseTreeEdge(b,e)
  mark(e)
  dfs(b,e)
    traverseTreeEdge(e,g)
    mark(g)
    dfs(e,g)
      traverseNonTreeEdge(g,b)
      traverseTreeEdge(g,f)
      mark(f)
      dfs(g,f)
        backtrack(g,f)
      backtrack(e,g)
    traverseNonTreeEdge(e,f)
    traverseTreeEdge(e,c)
    mark(c)
    dfs(e,c)
      backtrack(e,c)
    backtrack(b,e)
  backtrack(s,b)

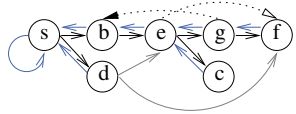


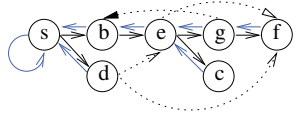






traverseTreeEdge(s,d)
mark(d)
dfs(s,d)
  traverseNonTreeEdge(d,e)
  traverseNonTreeEdge(d,f)
  backtrack(s,d)
backtrack(s,s)





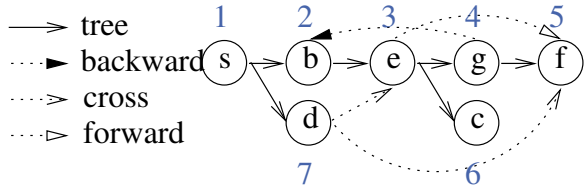
Tiefensuche

DFS Nummerierung

init:	dfsPos=1 : 1..n
root(s):	dfsNum[s]:= dfsPos++
traverseTreeEdge(v, w):	dfsNum[w]:= dfsPos++

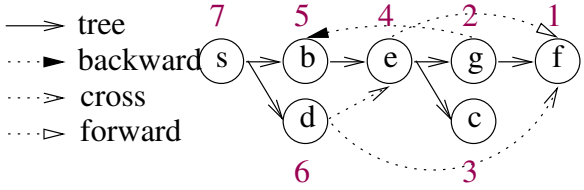
$$u \prec v :\Leftrightarrow \text{dfsNum}[u] < \text{dfsNum}[v] \ .$$

Beobachtung:
Knoten auf dem Rekursionsstapel sind bzgl., $\prec$ sortiert



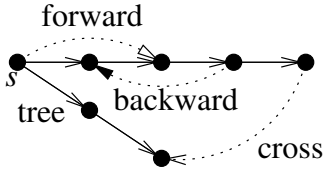
Tiefensuche
Fertigstellungszeit

init: finishingTime=1 : 1..n
backtrack(u, v): finishTime[v]:= finishingTime++



Tiefensuche
Kantenklassifizierung

type (v, w)	dfsNum$[v] <$ dfsNum$[w]$	finishTime$[w] <$ finishTime$[v]$	w is marked
tree	yes	yes	no
forward	yes	yes	yes
backward	no	no	yes
cross	no	yes	yes



Topologisches Sortieren mittels DFS

Satz

G ist kreisfrei (DAG) $\Leftrightarrow$ DFS findet keine Rückwärtskante.
In diesem Fall liefert

$$t(v) := n - \text{finishTime}[v]$$

eine topologische Sortierung,
d. h.

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

Topologisches Sortieren mittels DFS

Satz

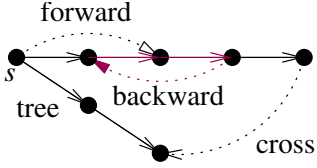
G ist kreisfrei (DAG) $\Leftrightarrow$ DFS findet keine Rückwärtskante.
In diesem Fall liefert

$$t(v) := n - \text{finishTime}[v]$$

eine topologische Sortierung,
d. h.

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

Beweis. "$\Rightarrow$" Annahme $\exists$ Rückwärtskante.
Zusammen mit Baumkanten ergibt sich ein Kreis.
Widerspruch. $\square$



Topologisches Sortieren mittels DFS

Satz

G ist kreisfrei (DAG) $\Leftrightarrow$ DFS findet keine Rückwärtskante.
In diesem Fall liefert

$$t(v) := n - \text{finishTime}[v]$$

eine topologische Sortierung,
d. h.

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

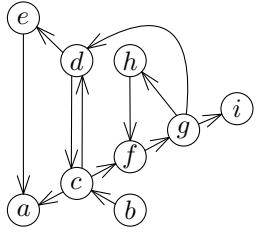
Beweis. "$\Leftarrow$" Keine Rückwärtskante
Kantenklassifizierung
$\overset{\Rightarrow}{\frown}$ $\forall (v,w) \in E : \text{finishTime}[v] > \text{finishTime}[w]$
$\Rightarrow$ finishTime definiert umgekehrte topologische Sortierung. □

Starke Zusammenhangskomponenten

Betrachte die Relation $\overset{*}{\leftrightarrow}$ mit
$u \overset{*}{\leftrightarrow} v$ falls $\exists$ Pfad $\langle u, \ldots, v \rangle$ und $\exists$ Pfad $\langle v, \ldots, u \rangle$.

Beobachtung: $\overset{*}{\leftrightarrow}$ ist Äquivalenzrelation Übung

Die Äquivalenzklassen von $\overset{*}{\leftrightarrow}$ bezeichnet man als starke Zusammenhangskomponenten.



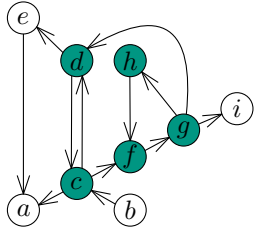
DFS-basierter Linearzeitalgorithmus $\longrightarrow$ Algorithmen II

Starke Zusammenhangskomponenten

Betrachte die Relation $\overset{*}{\leftrightarrow}$ mit
$u \overset{*}{\leftrightarrow} v$ falls $\exists$ Pfad $\langle u, \ldots, v \rangle$ und $\exists$ Pfad $\langle v, \ldots, u \rangle$.

Beobachtung: $\overset{*}{\leftrightarrow}$ ist Äquivalenzrelation Übung

Die Äquivalenzklassen von $\overset{*}{\leftrightarrow}$ bezeichnet man als starke Zusammenhangskomponenten.



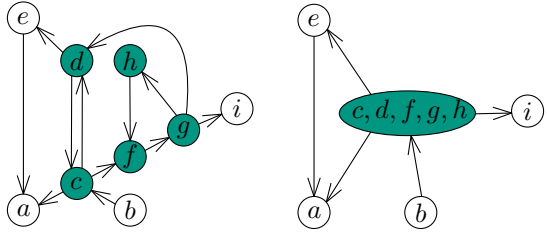
DFS-basierter Linearzeitalgorithmus $\longrightarrow$ Algorithmen II

Starke Zusammenhangskomponenten

Betrachte die Relation $\overset{*}{\leftrightarrow}$ mit
$u \overset{*}{\leftrightarrow} v$ falls $\exists$ Pfad $\langle u, \ldots, v \rangle$ und $\exists$ Pfad $\langle v, \ldots, u \rangle$.

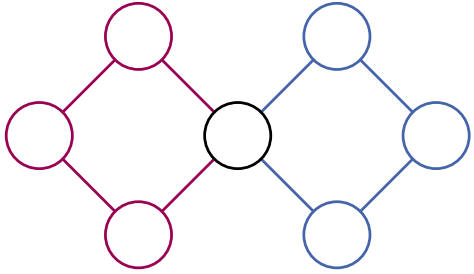
Beobachtung: $\overset{*}{\leftrightarrow}$ ist Äquivalenzrelation Übung

Die Äquivalenzklassen von $\overset{*}{\leftrightarrow}$ bezeichnet man als starke Zusammenhangskomponenten.



DFS-basierter Linearzeitalgorithmus $\longrightarrow$ Algorithmen II

Mehr DFS-basierte Linearzeitalgorithmen



- 2-zusammenhängende Komponenten: bei Entfernen eines Knotens aus einer Komponente bleibt diese zusammenhängend (ungerichtet)
- 3-zusammenhängende Komponenten
- Planaritätstest (läßt sich der Graph kreuzungsfrei zeichnen?)
- Einbettung planarer Graphen

Übersicht

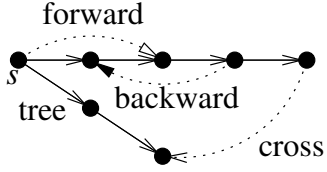
BFS $\longleftrightarrow$ DFS

pro BFS:

- nichtrekursiv
- keine Vorwärtskanten
- kürzeste Wege, „Umgebung“

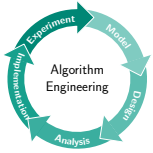
pro DFS:

- keine explizite TODO-Datenstruktur (Rekursionsstapel)
- Grundlage vieler Algorithmen



Index

Karlsruher Institut für Technologie



Algorithmen I – 10. Kürzeste Wege

Sommersemester 2025

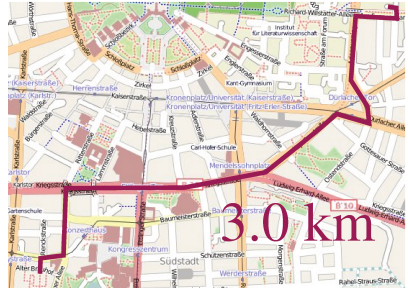
Peter Sanders | Stand: 30. Juli 2025

Übersicht

Kürzeste Wege

Eingabe: Graph $G = (V, E)$
Kostenfunktion/Kantengewicht $c : E \to \mathbb{R}$
Anfangsknoten s.

Ausgabe: für alle $v \in V$
Länge $\mu(v)$ des kürzesten Pfades von s nach v,
$\mu(v) := \min\{c(p) : p \text{ ist Pfad von } s \text{ nach } v\}$
mit $c(\langle e_1, \ldots, e_k \rangle) := \sum_{i=1}^{k} c(e_i)$.



Oft wollen wir auch „geeignete“ Repräsentation der kürzesten Pfade.

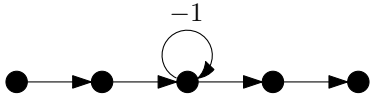
Anwendungen

- Routenplanung
  - Straßennetze
  - (Computer-)Spiele
  - Kommunikationsnetze
  - Roboterbewegungen
- Als Unterprogramm für
  - Flüsse in Netzwerken
  - ...
- Tippfehlerkorrektur
- Disk Scheduling
- ...

Grundlagen

Gibt es immer einen kürzesten Pfad?

Nein, es kann negative Kreise geben!



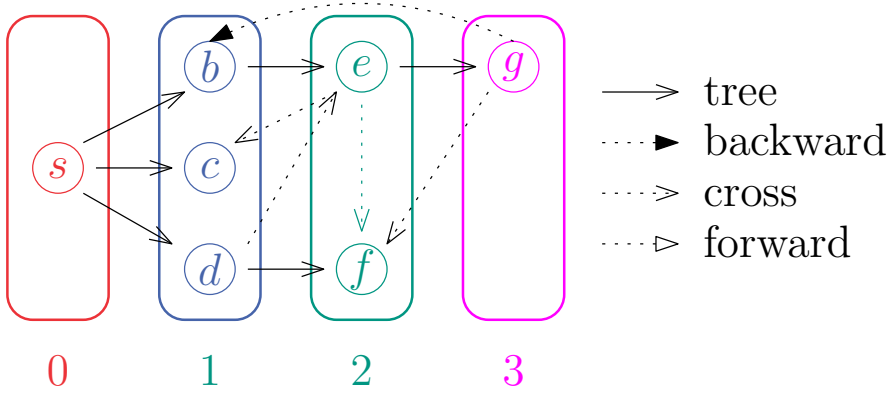
weitere Grundlagen just in time

Azyklische Graphen

später

Kantengewichte ≥ 0

Alle Gewichte gleich: Breitensuche (BFS)!



Übersicht

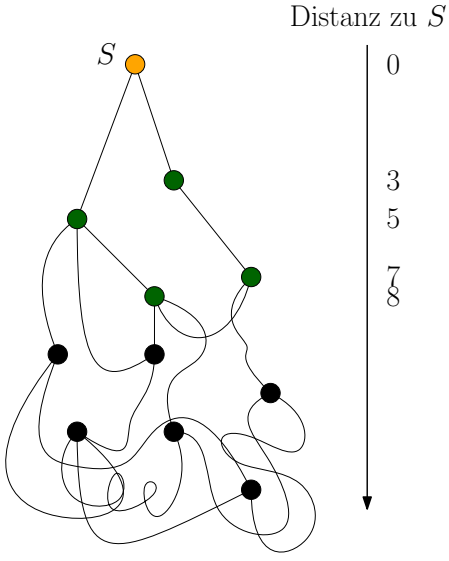
Dijkstras Algorithmus

Für allgemeine, nichtnegative Gewichte

Intuition: Fadenknäul

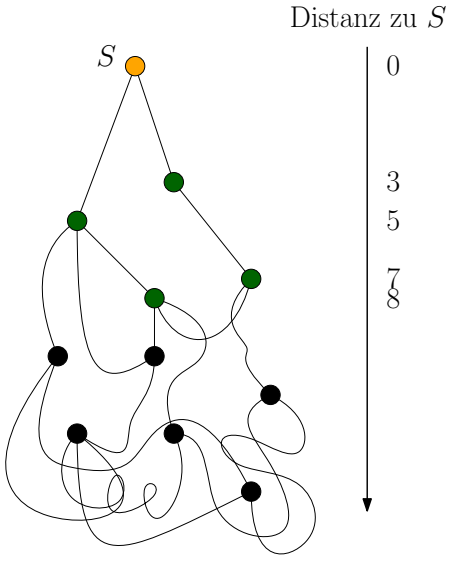
- Graph-Kanten $\rightarrow$ Fäden
- Graph-Knoten $\rightarrow$ Knoten!

Am Startknoten anheben.

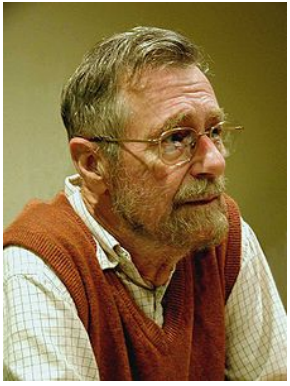


Korrekte Bindfäden

- Betrachte beliebigen Knoten v
  - Hängetiefe $d[v]$
- $\exists$ **Pfad mit Hängetiefe:**
  - verfolge straffe Fäden
- $\neg\exists$ **kürzerer Pfad:**
  - dann wäre einer seiner Fäden zerrissen ↯



Edsger Wybe Dijkstra 1930–2002



- 1972 ACM Turingpreis
- THE: das erste Mulitasking-OS
- Semaphor
- Selbst-stabilisierende Systeme
- GOTO Statement Considered Harmful

Dijkstras Algorithmus: Allgemeine Definitionen

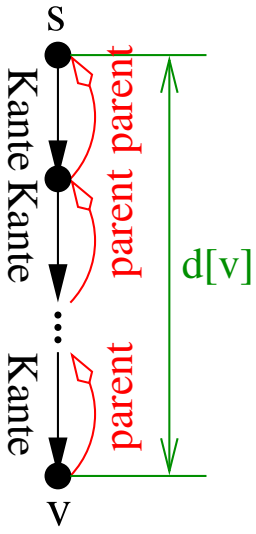
Wie bei BFS benutzen wir zwei Knotenarrays:

- $d[v]$ aktuelle (vorläufige) Distanz von s nach v
  $\mu(v)$ tatsächliche kürzeste Distanz von s nach v
  Invariante: $d[v] \geq \mu(v)$
- parent$[v]$ Vorgänger von v auf dem (vorläufigen) kürzesten Pfad von s nach v
  Invariante: Dieser Pfad bezeugt $d[v]$

Initialisierung:

$d[s] := 0$; parent$[s] := s$

$d[v] := \infty$; parent$[v] := \perp$

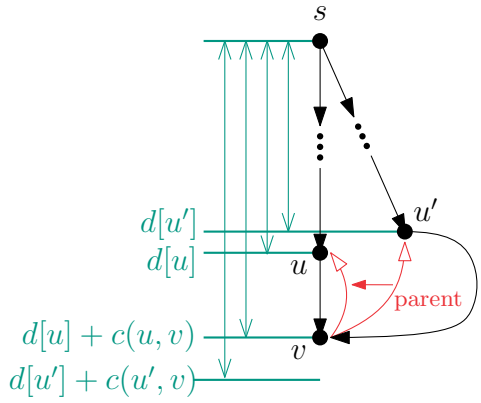


Dijkstras Algorithmus: Kante (u, v) relaxieren

falls $d[u] + c(u, v) < d[v]$
vielleicht $d[v] = \infty$
setze $d[v] := d[u] + c(u, v)$ und parent$[v] := u$

Invarianten bleiben erhalten!

Beobachtung:
$d[v]$ Kann sich mehrmals ändern!

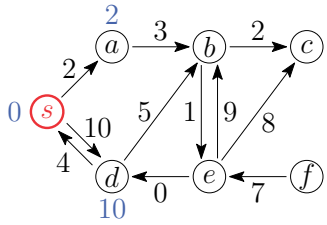


Dijkstras Algorithmus: Pseudocode

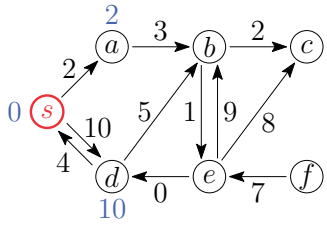
initialize d and parent
consider all nodes non-scanned
while $\exists$ non-scanned node u with $d[u] < \infty$
    $u :=$ non-scanned node v with minimal $d[v]$
    relax all edges (u, v) out of u // i.e., scan u
    consider u scanned

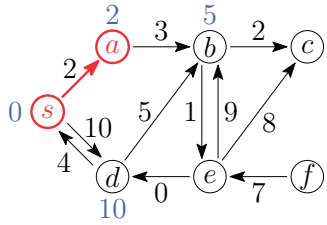
Behauptung:
Am Ende definiert d die optimalen Entfernungen und parent die zugehörigen Wege.

Dijkstras Algorithmus: Beispiel

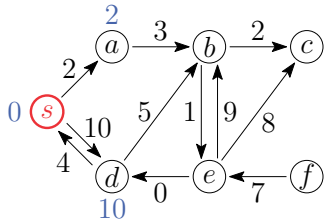


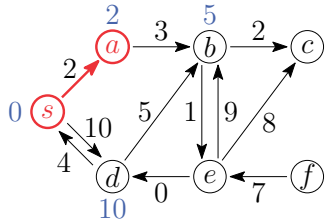
Dijkstras Algorithmus: Beispiel

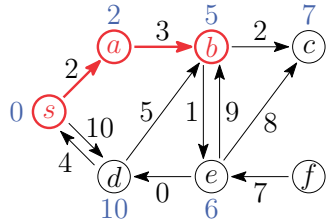




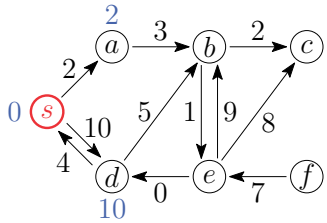
Dijkstras Algorithmus: Beispiel

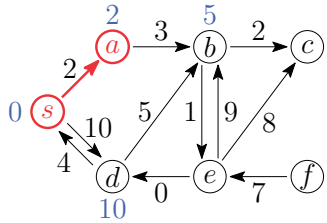


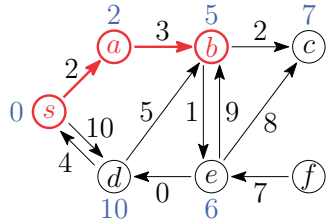


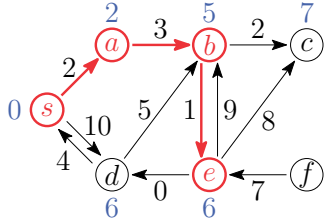


Dijkstras Algorithmus: Beispiel

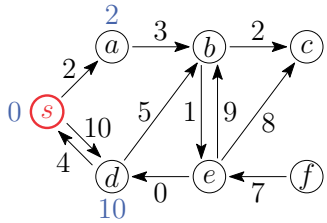


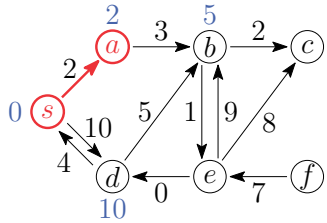


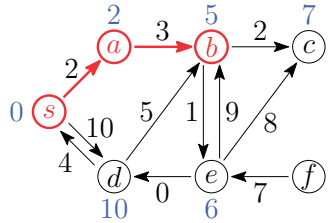


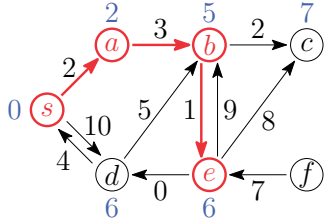


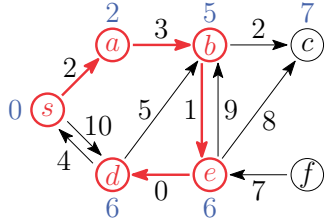
Dijkstras Algorithmus: Beispiel



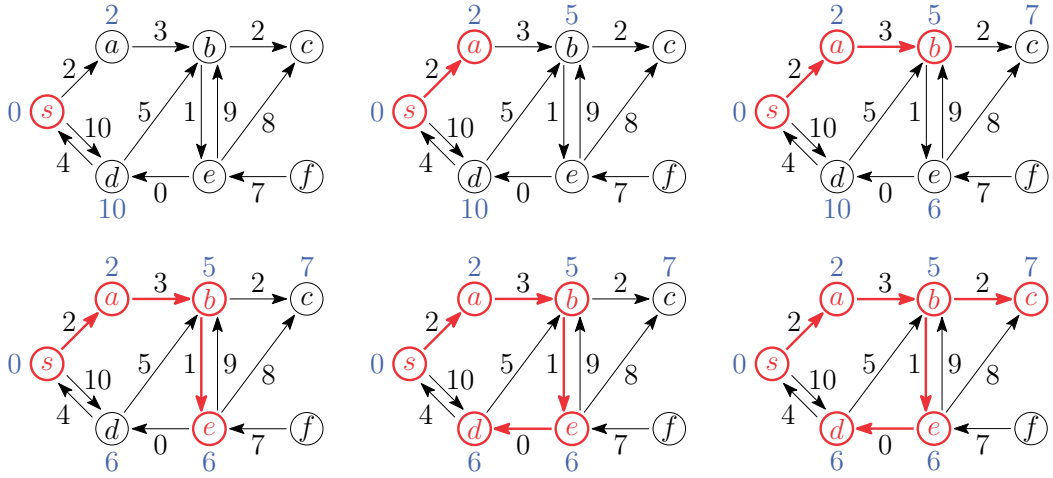








Dijkstras Algorithmus: Beispiel



Dijkstras Algorithmus: Korrektheit

Annahme

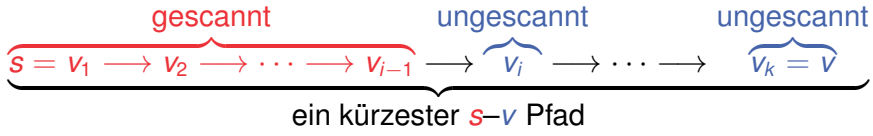
Alle Kosten sind nichtnegativ! $c(u, v) \geq 0$

Wir zeigen: $\forall v \in V$:

- v erreichbar $\Longrightarrow$ v wird irgendwann gescannt
- v gescannt $\Longrightarrow$ $\mu(v) = d[v]$

v **erreichbar** $\Longrightarrow$ v **wird irgendwann gescannt**

Annahme: v ist erreichbar aber wird nicht gescannt

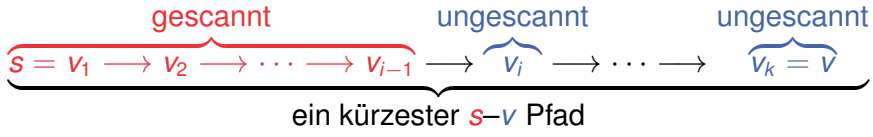


$\Longrightarrow$ v_{i-1} wird gescannt
$\Longrightarrow$ Kante $v_{i-1} \longrightarrow v_i$ wird relaxiert
$\Longrightarrow$ $d[v_i] < \infty$

Widerspruch ↯ Nur Knoten x mit $d[x] = \infty$ werden nie gescannt $\square$?

v erreichbar $\Longrightarrow$ v wird irgendwann gescannt

Annahme: v ist erreichbar aber wird nicht gescannt



$\Longrightarrow$ v_{i-1} wird gescannt
$\Longrightarrow$ Kante $v_{i-1} \longrightarrow v_i$ wird relaxiert
$\Longrightarrow$ $d[v_i] < \infty$

Widerspruch ↯ Nur Knoten x mit $d[x] = \infty$ werden nie gescannt

Oops: Spezialfall $i = 1$?

- Kann auch nicht sein.
- $v_1 = s$ wird bei Initialisierung gescannt.

□

v **gescannt** $\Longrightarrow \mu(v) = d[v]$

Annahme: v gescannt und $\mu(v) \neq d[v]$
Zwei Fälle:

1. $\mu(v) > d[v]$: Kann nicht sein, weil die Invariante sagt, dass $d[v] \geq \mu(v)$
2. $\mu(v) < d[v]$: nächste Folie

v **gescannt** $\Longrightarrow \mu(v) = d[v]$

Annahme: v gescannt und $\mu(v) < d[v]$

OBdA: v ist der erste gescannte Knoten mit $\mu(v) < d[v]$.

$t :=$ Scan-Zeit von v

Also gilt zur Zeit t:

$\mu(v_{i-1}) = d[v_{i-1}]$

$v_{i-1} \rightarrow v_i$ wurde relaxiert

$\Longrightarrow d[v_i] \leq d[v_{i-1}] + c(v_{i-1}, v_i) = \mu(v_i) \leq \mu(v) < d[v]$

$\Longrightarrow v_i$ wird vor v gescannt. **Widerspruch** ↯

Spezialfall $i = 1$ unmöglich. □

Dijkstras Algorithmus: Implementierung?

initialize d, parent
consider all nodes non-scanned
while $\exists$ non-scanned node u with $d[u] < \infty$
    $u :=$ non-scanned node v with minimal $d[v]$
    relax all edges (u, v) out of u
    u is scanned now

Wichtigste Operation: finde u

Prioritätsliste

Addressierbare Prioritätsliste Q

- Speichert ungescannte erreichte Knoten
- Schlüssel ist $d[v]$
- Knoten speichern handles von PQ-Einträgen. (oder gleich items)

Implementierung $\approx$ BFS mit PQ statt FIFO

Function Dijkstra(s : NodeId) : NodeArray$\times$NodeArray // returns (d, parent)

Initialisierung:

$d = \langle \infty, \ldots, \infty \rangle$: NodeArray **of** $\mathbb{R} \cup \{\infty\}$ // tentative distance from root

parent$= \langle \bot, \ldots, \bot \rangle$: NodeArray **of** NodeId

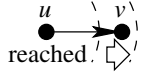
parent$[s] := s$ // self-loop signals root

Q : NodePQ // unscanned reached nodes

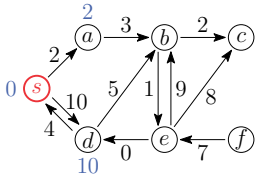
$d[s] := 0$; Q.insert(s)

Implementierung

```
Function Dijkstra(s : NodeId) : NodeArray×NodeArray
    d = ⟨∞, ..., ∞⟩; parent[s]:= s; d[s] := 0;   Q.insert(s)
    while Q ≠ ∅ do
        u := Q.deleteMin
        //scan u
        foreach edge e = (u, v) ∈ E do //use adjacency array
            if d[u] + c(e) < d[v] then                          // relax
                d[v]:= d[u] + c(e)
                parent[v] := u                                  // update tree
                if v ∈ Q then Q.decreaseKey(v)
                else Q.insert(v)
    return (d, parent)
```

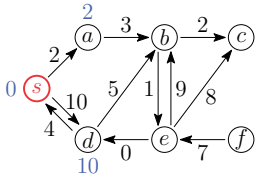


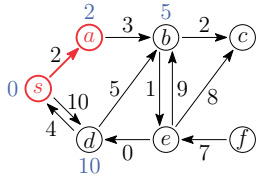
Dijkstras Algorithmus: Beispiel



Operation	Queue
insert(s)	$\langle(s,0)\rangle$
deleteMin$\rightsquigarrow$ $(s,0)$	$\langle\rangle$
$s \xrightarrow{2} a$	$\langle(a,2)\rangle$
$s \xrightarrow{10} d$	$\langle(a,2),(d,10)\rangle$

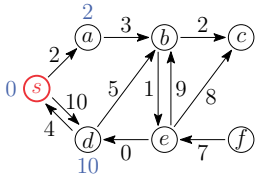
Dijkstras Algorithmus: Beispiel

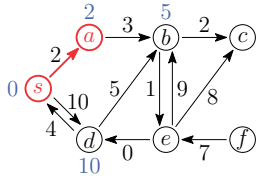


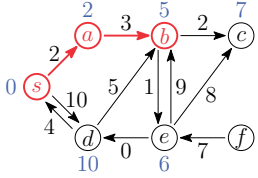


Operation	Queue
insert(s)	$\langle (s,0) \rangle$
deleteMin$\rightsquigarrow$ $(s,0)$	$\langle \rangle$
$s \xrightarrow{2} a$	$\langle (a,2) \rangle$
$s \xrightarrow{10} d$	$\langle (a,2),(d,10) \rangle$
deleteMin$\rightsquigarrow$ $(a,2)$	$\langle (d,10) \rangle$
$a \xrightarrow{3} b$	$\langle (b,5),(d,10) \rangle$

Dijkstras Algorithmus: Beispiel

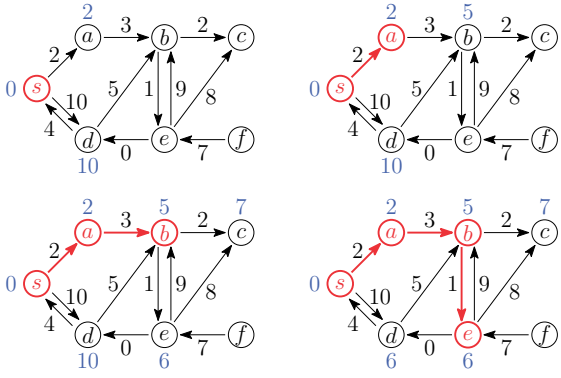






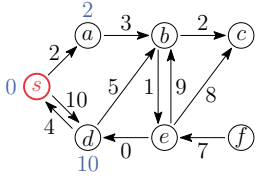
Operation	Queue
insert(s)	$\langle(s,0)\rangle$
deleteMin$\rightsquigarrow$ $(s,0)$	$\langle\rangle$
$s \xrightarrow{2} a$	$\langle(a,2)\rangle$
$s \xrightarrow{10} d$	$\langle(a,2),(d,10)\rangle$
deleteMin$\rightsquigarrow$ $(a,2)$	$\langle(d,10)\rangle$
$a \xrightarrow{3} b$	$\langle(b,5),(d,10)\rangle$
deleteMin$\rightsquigarrow$ $(b,5)$	$\langle(d,10)\rangle$
$b \xrightarrow{2} c$	$\langle(c,7),(d,10)\rangle$
$b \xrightarrow{1} e$	$\langle(e,6),(c,7),(d,10)\rangle$

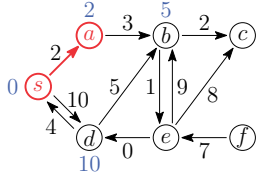
Dijkstras Algorithmus: Beispiel

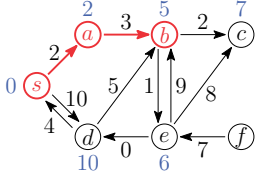


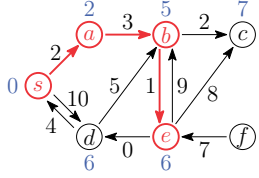
Operation	Queue
insert(s)	$\langle(s,0)\rangle$
deleteMin$\rightsquigarrow$ $(s,0)$	$\langle\rangle$
$s \xrightarrow{2} a$	$\langle(a,2)\rangle$
$s \xrightarrow{10} d$	$\langle(a,2),(d,10)\rangle$
deleteMin$\rightsquigarrow$ $(a,2)$	$\langle(d,10)\rangle$
$a \xrightarrow{3} b$	$\langle(b,5),(d,10)\rangle$
deleteMin$\rightsquigarrow$ $(b,5)$	$\langle(d,10)\rangle$
$b \xrightarrow{2} c$	$\langle(c,7),(d,10)\rangle$
$b \xrightarrow{1} e$	$\langle(e,6),(c,7),(d,10)\rangle$
deleteMin$\rightsquigarrow$ $(e,6)$	$\langle(c,7),(d,10)\rangle$
$e \xrightarrow{9} b$	$\langle(c,7),(d,10)\rangle$
$e \xrightarrow{8} c$	$\langle(c,7),(d,10)\rangle$
$e \xrightarrow{0} d$	$\langle(d,6),(c,7)\rangle$

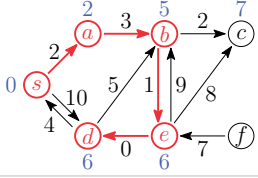
Dijkstras Algorithmus: Beispiel





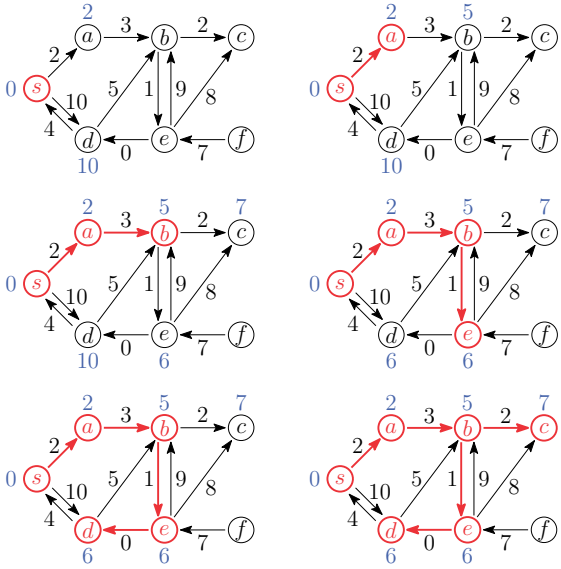






Operation	Queue
insert(s)	$\langle(s,0)\rangle$
deleteMin$\rightsquigarrow$ $(s,0)$	$\langle\rangle$
$s \xrightarrow{2} a$	$\langle(a,2)\rangle$
$s \xrightarrow{10} d$	$\langle(a,2),(d,10)\rangle$
deleteMin$\rightsquigarrow$ $(a,2)$	$\langle(d,10)\rangle$
$a \xrightarrow{3} b$	$\langle(b,5),(d,10)\rangle$
deleteMin$\rightsquigarrow$ $(b,5)$	$\langle(d,10)\rangle$
$b \xrightarrow{2} c$	$\langle(c,7),(d,10)\rangle$
$b \xrightarrow{1} e$	$\langle(e,6),(c,7),(d,10)\rangle$
deleteMin$\rightsquigarrow$ $(e,6)$	$\langle(c,7),(d,10)\rangle$
$e \xrightarrow{9} b$	$\langle(c,7),(d,10)\rangle$
$e \xrightarrow{8} c$	$\langle(c,7),(d,10)\rangle$
$e \xrightarrow{0} d$	$\langle(d,6),(c,7)\rangle$
deleteMin$\rightsquigarrow$ $(d,6)$	$\langle(c,7)\rangle$
$d \xrightarrow{4} s$	$\langle(c,7)\rangle$
$d \xrightarrow{5} b$	$\langle(c,7)\rangle$

Dijkstras Algorithmus: Beispiel



Operation	Queue
insert(s)	$\langle(s,0)\rangle$
deleteMin$\rightsquigarrow$ $(s,0)$	$\langle\rangle$
$s \xrightarrow{2} a$	$\langle(a,2)\rangle$
$s \xrightarrow{10} d$	$\langle(a,2),(d,10)\rangle$
deleteMin$\rightsquigarrow$ $(a,2)$	$\langle(d,10)\rangle$
$a \xrightarrow{3} b$	$\langle(b,5),(d,10)\rangle$
deleteMin$\rightsquigarrow$ $(b,5)$	$\langle(d,10)\rangle$
$b \xrightarrow{2} c$	$\langle(c,7),(d,10)\rangle$
$b \xrightarrow{1} e$	$\langle(e,6),(c,7),(d,10)\rangle$
deleteMin$\rightsquigarrow$ $(e,6)$	$\langle(c,7),(d,10)\rangle$
$e \xrightarrow{9} b$	$\langle(c,7),(d,10)\rangle$
$e \xrightarrow{8} c$	$\langle(c,7),(d,10)\rangle$
$e \xrightarrow{0} d$	$\langle(d,6),(c,7)\rangle$
deleteMin$\rightsquigarrow$ $(d,6)$	$\langle(c,7)\rangle$
$d \xrightarrow{4} s$	$\langle(c,7)\rangle$
$d \xrightarrow{5} b$	$\langle(c,7)\rangle$
deleteMin$\rightsquigarrow$ $(c,7)$	$\langle\rangle$

Dijkstras Algorithmus: Laufzeit

Function Dijkstra(s : NodeId) : NodeArray$\times$NodeArray
Initialisierung:
$d = \langle \infty, \ldots, \infty \rangle$: NodeArray **of** $\mathbb{R} \cup \{\infty\}$ // $O(n)$
parent$= \langle \bot, \ldots, \bot \rangle$: NodeArray **of** NodeId // $O(n)$
parent$[s] := s$
Q : NodePQ // unscanned reached nodes, $O(n)$
$d[s] := 0$; Q.insert(s)

Dijkstras Algorithmus: Laufzeit

```
Function Dijkstra(s : NodeId) : NodeArray×NodeArray
    d = {∞, ..., ∞}; parent[s]:= s; d[s] := 0;   Q.insert(s)          // O(n)
    while Q ≠ ∅ do
        u := Q.deleteMin                                               // ≤ n×
        foreach edge e = (u, v) ∈ E do                                 // ≤ m×
            if d[u] + c(e) < d[v] then                                 // ≤ m×
                d[v]:= d[u] + c(e)                                     // ≤ m×
                parent[v] := u                                         // ≤ m×
                if v ∈ Q then Q.decreaseKey(v)                         // ≤ m×
                else Q.insert(v)                                       // ≤ n×
    return (d, parent)
```

Dijkstras Algorithmus: Laufzeit

```
Function Dijkstra(s : NodeId) : NodeArray×NodeArray
    d = {∞, ..., ∞}; parent[s]:= s; d[s] := 0;   Q.insert(s)          // O(n)
    while Q ≠ ∅ do
        u := Q.deleteMin                                               // ≤ n×
        foreach edge e = (u, v) ∈ E do                                 // ≤ m×
            if d[u] + c(e) < d[v] then                                 // ≤ m×
                d[v]:= d[u] + c(e)                                     // ≤ m×
                parent[v] := u                                         // ≤ m×
                if v ∈ Q then Q.decreaseKey(v)                         // ≤ m×
                else Q.insert(v)                                       // ≤ n×
    return (d, parent)
```

Insgesamt:

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

Dijkstras Algorithmus: Laufzeit

Dijkstra's ursprüngliche Implementierung: „naiv“

- insert $O(1)$ — $d[v] := d[u] + c(u,v)$
- decreaseKey $O(1)$ — $d[v] := d[u] + c(u,v)$
- deleteMin $O(n)$ — d komplett durchsuchen

$$
\begin{aligned}
T_{\text{Dijkstra}} &= O(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))) \\
T_{\text{Dijkstra59}} &= O(m \cdot 1 + n \cdot (n+1)) \\
&= O(m + n^2)
\end{aligned}
$$

Dijkstras Algorithmus: Laufzeit

Bessere Implementierung mit Binary-Heap-Prioritätslisten:

- insert $\mathrm{O}(\log n)$
- decreaseKey $\mathrm{O}(\log n)$
- deleteMin $\mathrm{O}(\log n)$

$$
\begin{aligned}
T_{\text{Dijkstra}} &= \mathrm{O}(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))) \\
T_{\text{DijkstraBHeap}} &= \mathrm{O}(m \cdot \log n + n \cdot (\log n + 1)) \\
&= \mathrm{O}((m+n)\log n)
\end{aligned}
$$

Dijkstras Algorithmus: Laufzeit

(Noch) besser mit Fibonacci-Heapprioritätslisten: (siehe AlgoII)

- insert $O(1)$
- decreaseKey $O(1)$ (amortisiert)
- deleteMin $O(\log n)$ (amortisiert)

$$
\begin{aligned}
T_{\text{Dijkstra}} &= O(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))) \\
T_{\text{DijkstraFib}} &= O(m \cdot 1 + n \cdot (\log n + 1)) \\
&= O(m + n \log n)
\end{aligned}
$$

Aber: konstante Faktoren in $O(\cdot)$ sind hier größer!

Analyse im Mittel

Modell: Kantengewichte sind „zufällig“ auf die Kanten verteilt
Dann gilt

$$\mathrm{E}[T_{\text{DijkstraBHeap}}] = \mathrm{O}\left(m + n \log n \log \frac{m}{n}\right)$$

Beweis: In Algorithmen II

Monotone ganzzahlige Prioritätslisten

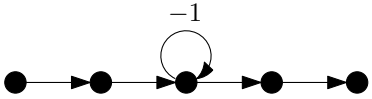
Beobachtung: In Dijkstra's Algorithmus steigt das Minimum in der Prioritätsliste monoton.
Das kann man ausnutzen. $\rightsquigarrow$ schnellere Algorithmen
u.U. bis herunter zu $O(m+n)$.

Details: in Algorithmen II

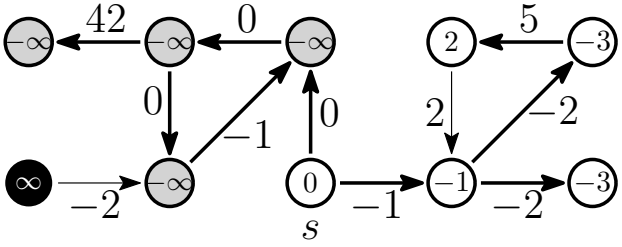
Übersicht

Negative Kosten

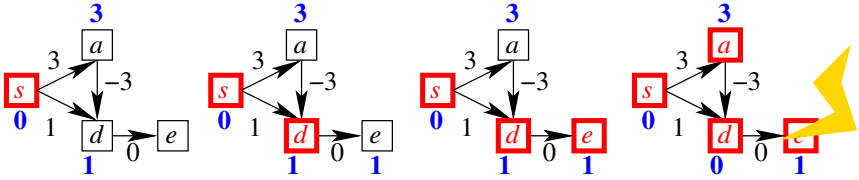
Was machen wir wenn es Kanten mit negativen Kosten gibt?
Es kann Knoten geben mit $d[v] = -\infty$



Wie finden wir heraus, welche das sind?
Endlosschleifen vermeiden!



Dijkstra funkioniert nicht



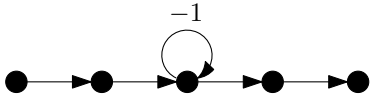
Selbst für kreisfreie Graphen.

Zurück zu Basiskonzepten (Abschnitt 10.1 im Buch)

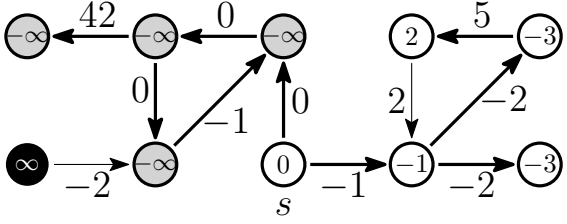
Lemma: $\exists$ kürzesten s–v-Pfad $P \Longrightarrow$ P ist OBdA einfach (eng. simple)

Beweisidee:

Fall: v über negativen Kreis erreichbar?**:**
$\neg\exists$ kürzesten s–v-Pfad
(sondern beliebig viele immer kürzere)



Sonst: betrachte beliebigen nicht-einfachen s–v-Pfad.
Alle Kreise streichen $\rightsquigarrow$ einfacher, nicht längerer Pfad. ■



Mehr Basiskonzepte

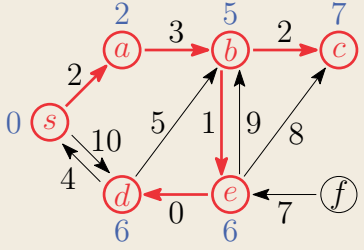
Übung

Zeige: Teilpfade kürzester Pfade sind selbst kürzeste Pfade.

$$a-b-c-d \rightsquigarrow a-b, b-c, c-d, a-b-c, b-c-d$$

Übung

Zeige: Kürzeste Pfade von s aus bilden einen Baum falls es keine negativen Kreise gibt.



Allgemeines Korrektheitskriterium

Sei $R = \langle \cdots \overbrace{\text{relax}(e_1)}^{t_1} \cdots \overbrace{\text{relax}(e_2)}^{t_2} \cdots \overbrace{\text{relax}(e_k)}^{t_k} \cdots \rangle$
eine Folge von Relaxierungsoperationen und
$p = \langle e_1, e_2, \ldots, e_k \rangle = \langle s, v_1, v_2, \ldots, v_k \rangle$ ein kürzester Weg.

Dann gilt anschließend $d[v_k] = \mu(v_k)$

Beweisskizze

(Eigentlich Induktion über k)

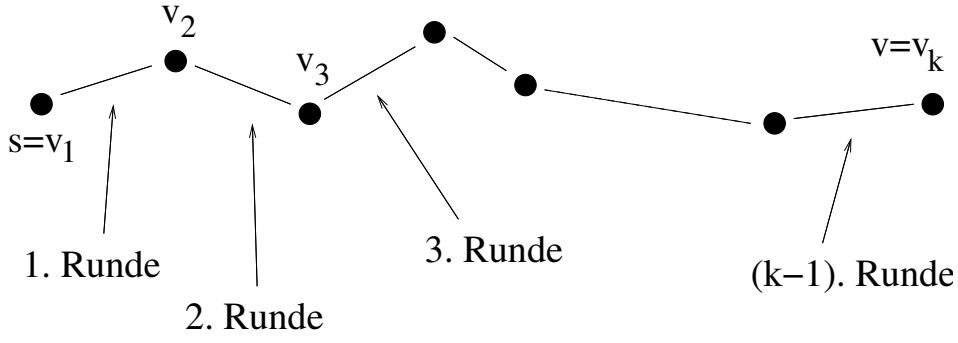
$$
\begin{aligned}
d[s] &= \mu(s) \text{ bei Initialisierung} \\
d[v_1] &= \mu(v_1) \text{ nach Zeitpunkt } t_1 \\
d[v_2] &= \mu(v_2) \text{ nach Zeitpunkt } t_2 \\
&\cdots \\
d[v_k] &= \mu(v_k) \text{ nach Zeitpunkt } t_k
\end{aligned}
$$

■

Algorithmen Brutal – Bellman-Ford-Algorithmus für beliebige Kantengewichte

Wir relaxieren alle Kanten (in irgendeiner Reihenfolge) $n - 1$ mal

Alle kürzeste Pfade in G haben höchstens $n - 1$ Kanten
Jeder kürzeste Pfad ist eine Teilfolge dieser Relaxierungen!



Negative Kreise Finden

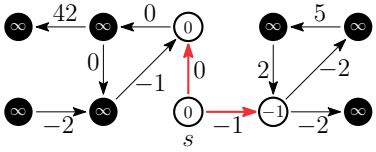
Nach Ausführung von Bellman-Ford:

$\forall$ negativen Kreise C:
$\exists (u, v) \in C$:
$d[u] + c(e) < d[v]$

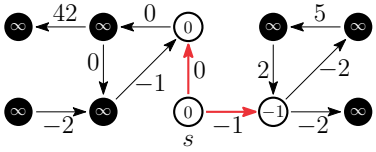
Beweis: Übung

v und alle von v erreichbaren Knoten x erhalten $\mu(x) = -\infty$

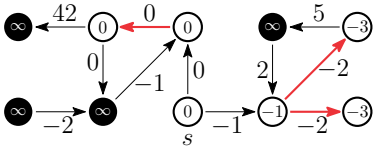
Runde 1



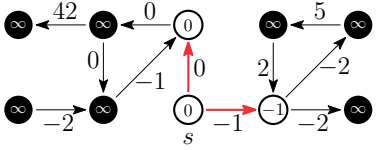
Runde 1



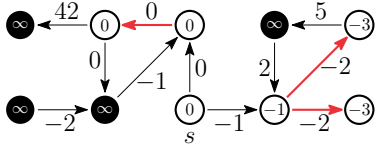
Runde 2



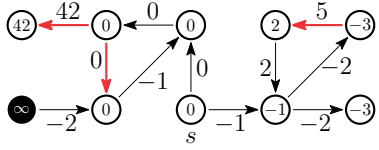
Runde 1



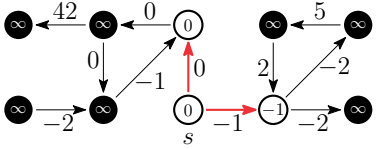
Runde 2



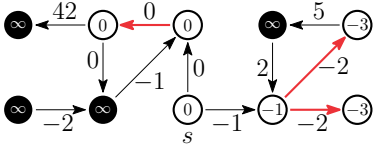
Runde 3



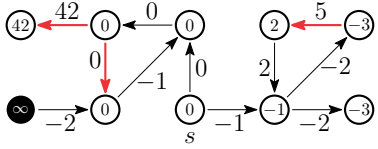
Runde 1



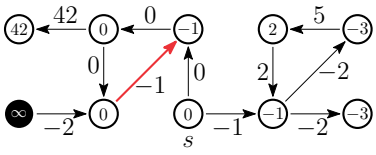
Runde 2



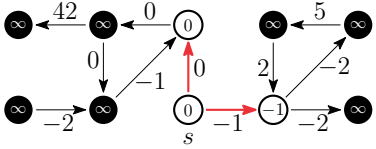
Runde 3



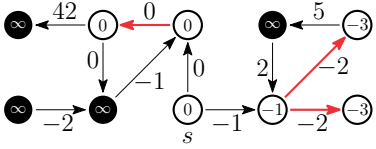
Runde 4



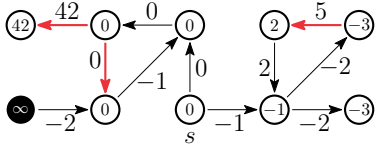
Runde 1



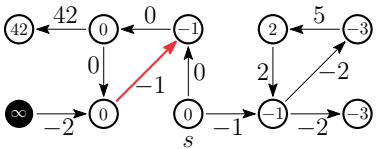
Runde 2



Runde 3

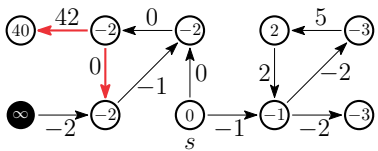


Runde 4

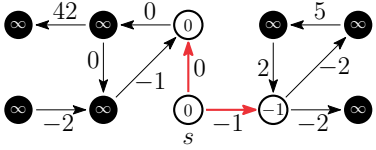


...

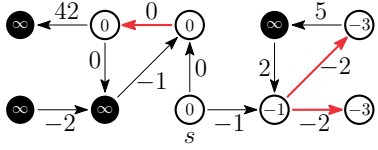
Runde $n-1$



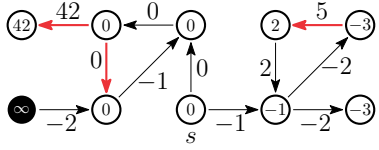
Runde 1



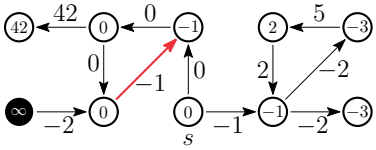
Runde 2



Runde 3

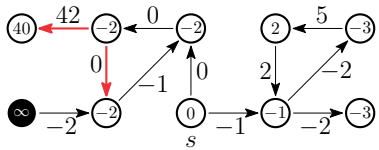


Runde 4

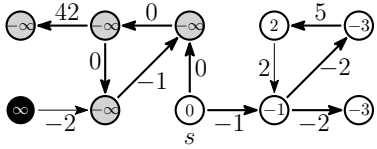


...

Runde $n-1$



Finde negative Kreise



Bellman-Ford: Laufzeit

$O(nm)$ viel langsamer als Dijkstra!

Es gibt Algorithmenvarianten mit viel besserem best case.

Blick über den Tellerrand

Algorithmus von Bernstein und Nanongkai, 2022

$$O\left(m \log^8(n) \log W\right)$$

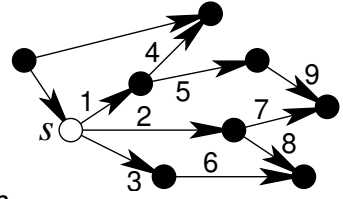
Zeit für ganzzahlige Kantengewichte aus $-W..W$.

Azyklische Graphen (10.2 im Buch)

Beobachtungen

- Keine (gerichteten) Kreise $\Longrightarrow$ keine negativen Kreise!
- Für jeden (kürzesten) Pfad $\langle v_1, \ldots, v_n \rangle$:
  Die Kanten sind aufsteigend bzgl. jeder topologischen Sortierung!

initialize d, parent
foreach $v \in V$ in topological order **do** scan(v)



Laufzeit: $O(m+n)$
Korrektheit: folgt aus dem allgemeinen Korrektheitskriterium

Übersicht

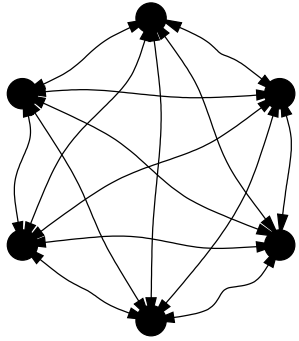
Von Überall nach Überall

Im Prinzip: $n\times$ von s nach Überall

nichtnegative Kantengewichte: Zeit $O(n(m + n \log n))$.
($n\times$ Dijkstra)

beliebige Kantengewichte: Zeit $O(n^2 m)$.
($n\times$ Bellman-Ford)

In Algorithmen II: Zeit $O(n(m + n \log n))$.
($1\times$ Bellman-Ford $+$ $n\times$ Dijkstra)



Übersicht

Kürzeste Wege: Zusammenfassung

- Einfache, effiziente Algorithmen für nichtnegative Kantengewichte und azyklische Graphen
- Optimale Lösungen bei nicht (ganz) trivialen Korrektheitsbeweisen
- Prioritätslisten sind wichtige Datenstruktur

Mehr zu kürzesten Wegen

Viele Arbeiten zu besseren Prioritätslisten $\rightsquigarrow O(m + n \log \log n)$ [Thorup 2004]

Verallgemeinerungen:

- Mehrere Zielfunktionen abwägen
- Mehrere Ziele in beliebiger Reihenfolge anfahren — siehe auch Optimierungskapitel
- Mehrere disjunkte Wege

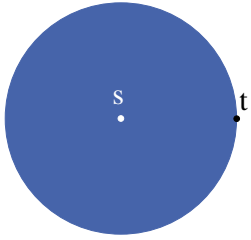
Fast alles schwierig ($\mathcal{NP}$-hart)

Übersicht

Distanz zu einem Zielknoten t

Was machen wir, wenn wir nur die Distanz von s zu einem bestimmten Knoten t wissen wollen?

Trick 0: Dijkstra hört auf, wenn t aus Q entfernt wird
Spart “im Durchschnitt” Hälfte der Scans

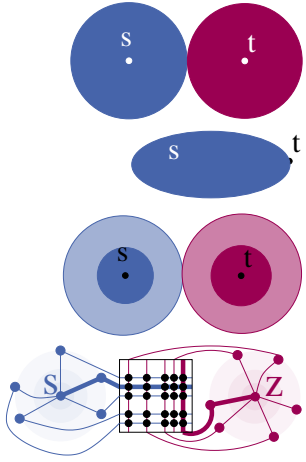


Frage: Wieviel spart es (meist) beim Europa-Navi?

Ideen für Routenplanung

mehr in Algorithmen II, Algorithm Engineering

- Vorwärts + Rückwärtssuche
- Zielgerichtete Suche
- Hierarchien ausnutzen
- Teilabschnitte tabellieren

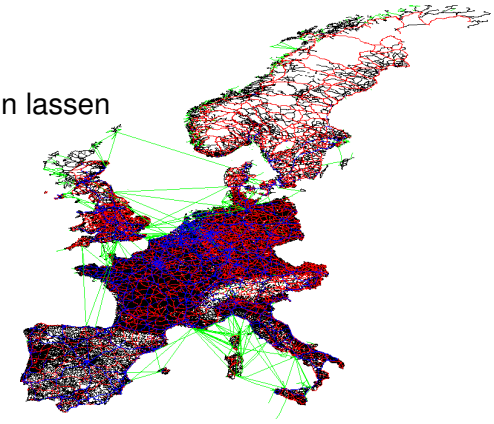


Meist zentrale Idee: **Vorberechnung** amortisiert über viele Anfragen

Straßennetzwerke

Wir konzentrieren uns auf Straßennetzwerke.

- mehrere nützliche Eigenschaften die sich ausnutzen lassen
- viele reale Anwendungen



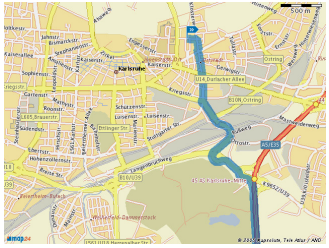
- einige Techniken: anwendbar für öffentliche Verkehrsmittel
- die meisten Techniken: unklar wie nützlich sie für weitere Graphtypen sind

Straßennetzwerke
Eigenschaften

- groß, z.B. $n = 18\,000\,000$ Knoten für Westeuropa
- dünn besetzt, z.B., $m = \Theta(n)$ Kanten
- beinahe planar, d.h., wenige Kanten kreuzen sich (Brücken)
- inhärente Hierarchie, schnellste Pfade benutzen wichtige Straßen

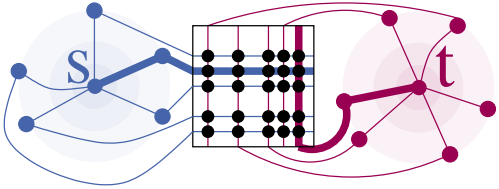
Straßennetzwerke
Anwendungen

- Routenplanungssysteme im Internet
- Fahrzeugnavigationssysteme
- Logistik
- Verkehrssimulation



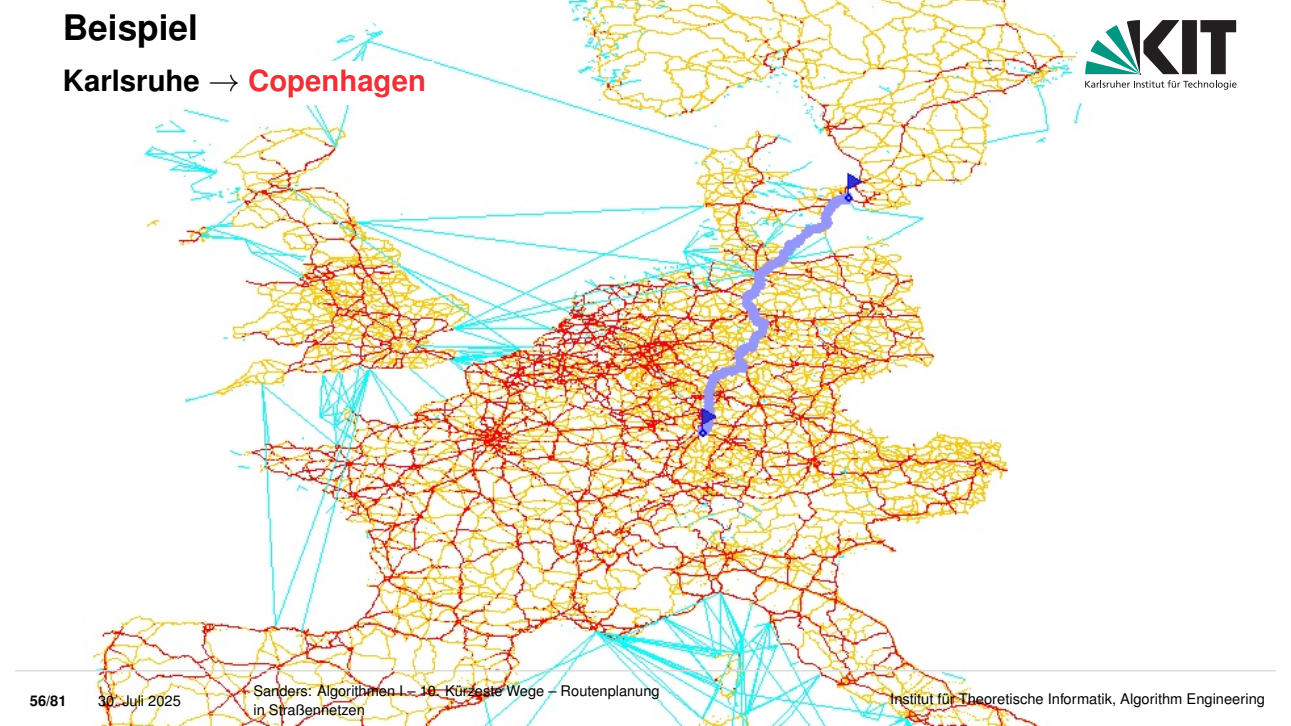
Transit-Node Routing

[with H. Bast and S. Funke]



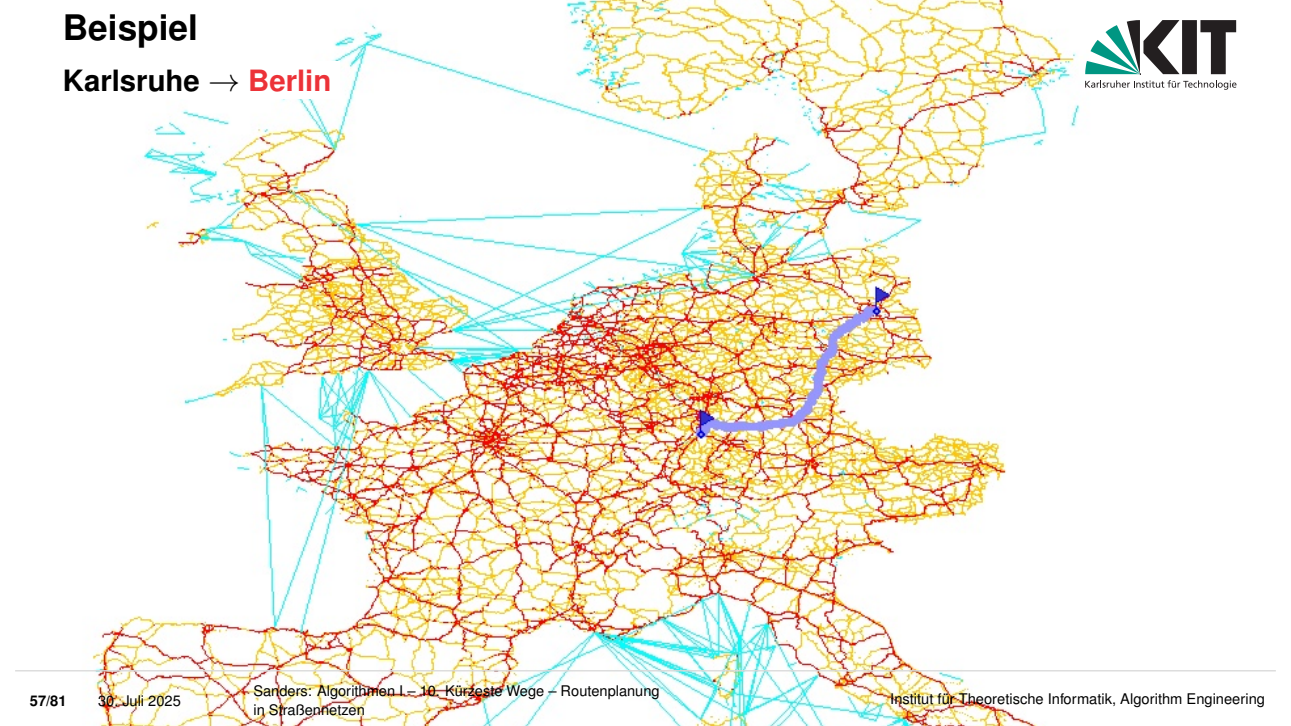
Beispiel

Karlsruhe → Copenhagen



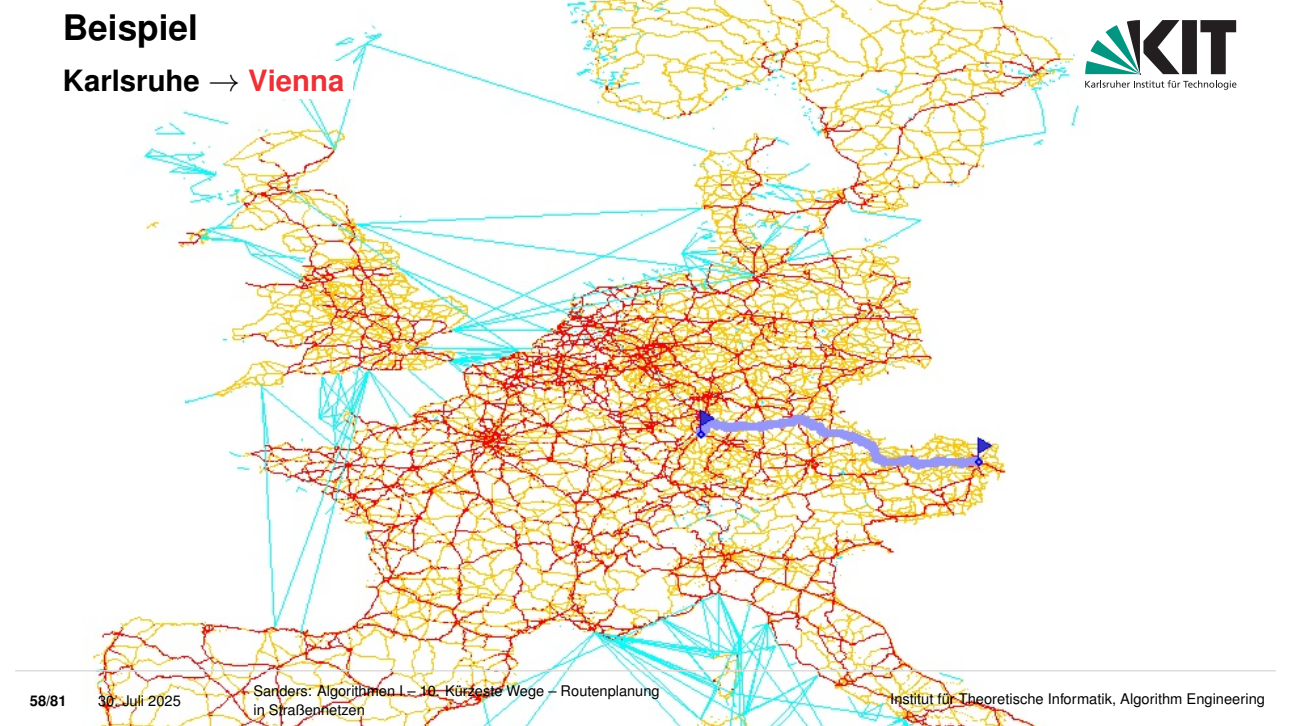
Beispiel

Karlsruhe → Berlin



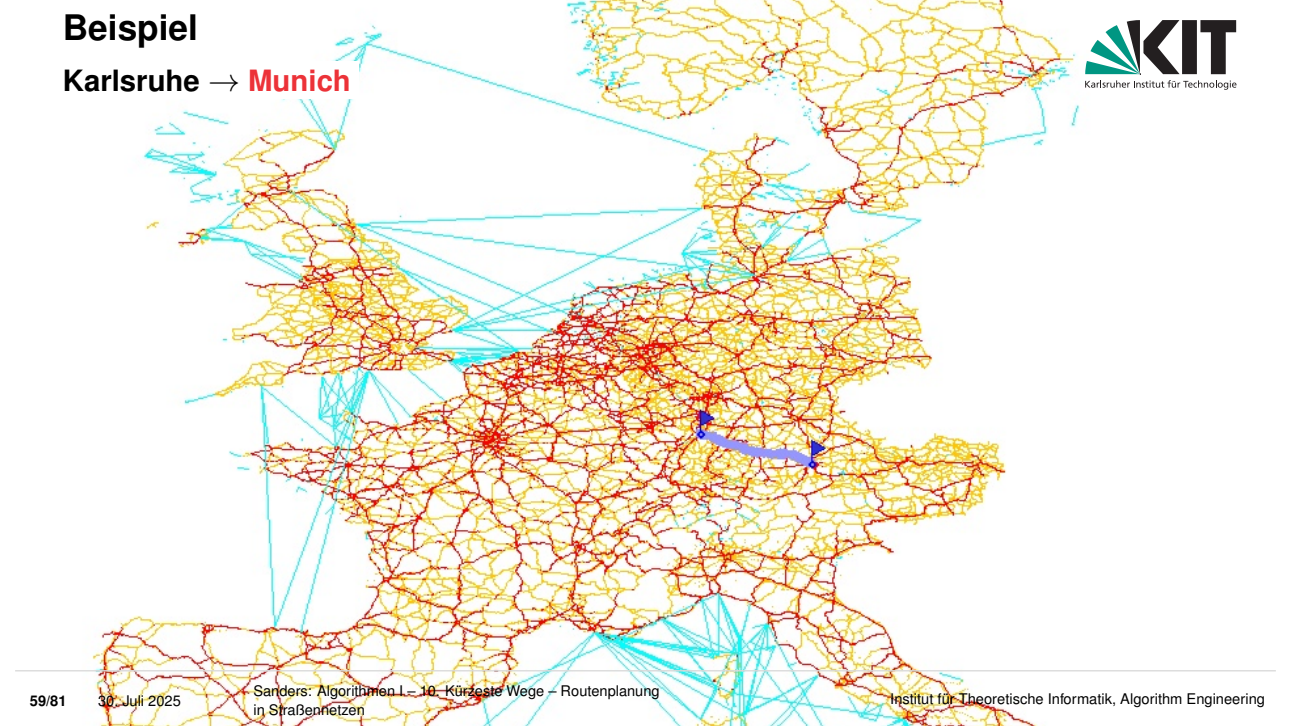
Beispiel

Karlsruhe → Vienna



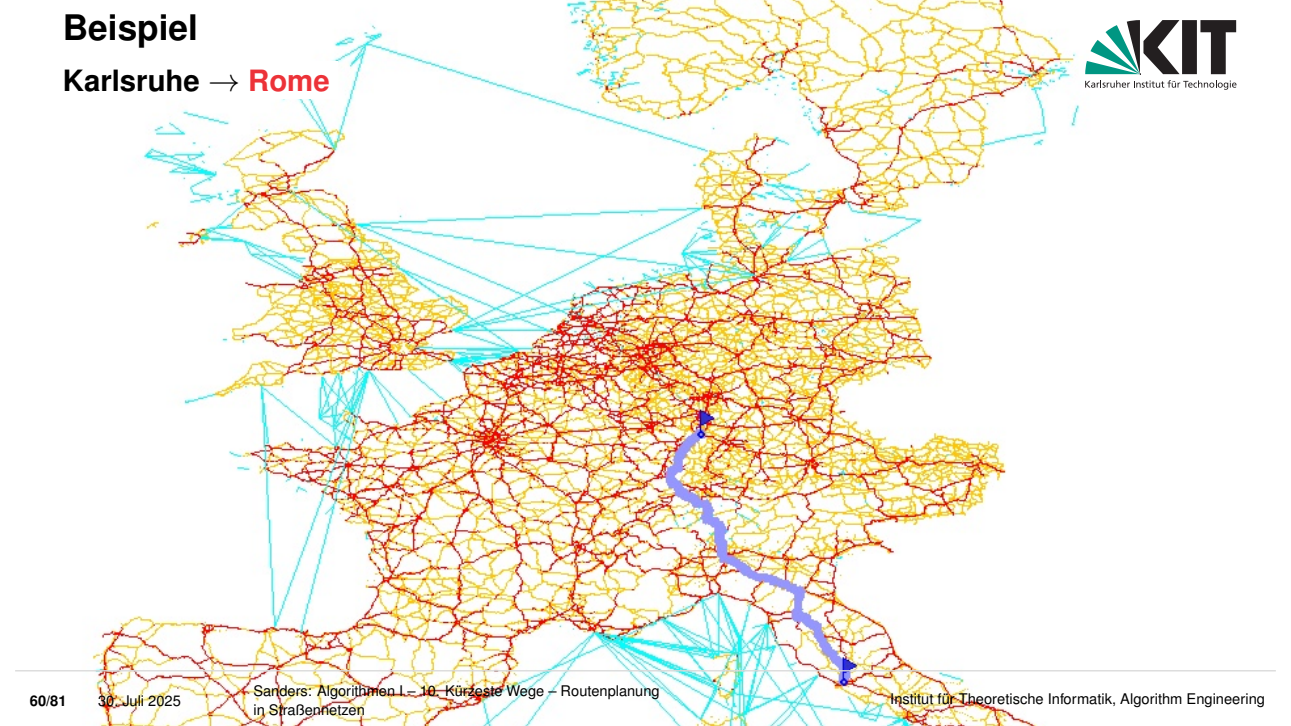
Beispiel

Karlsruhe → Munich



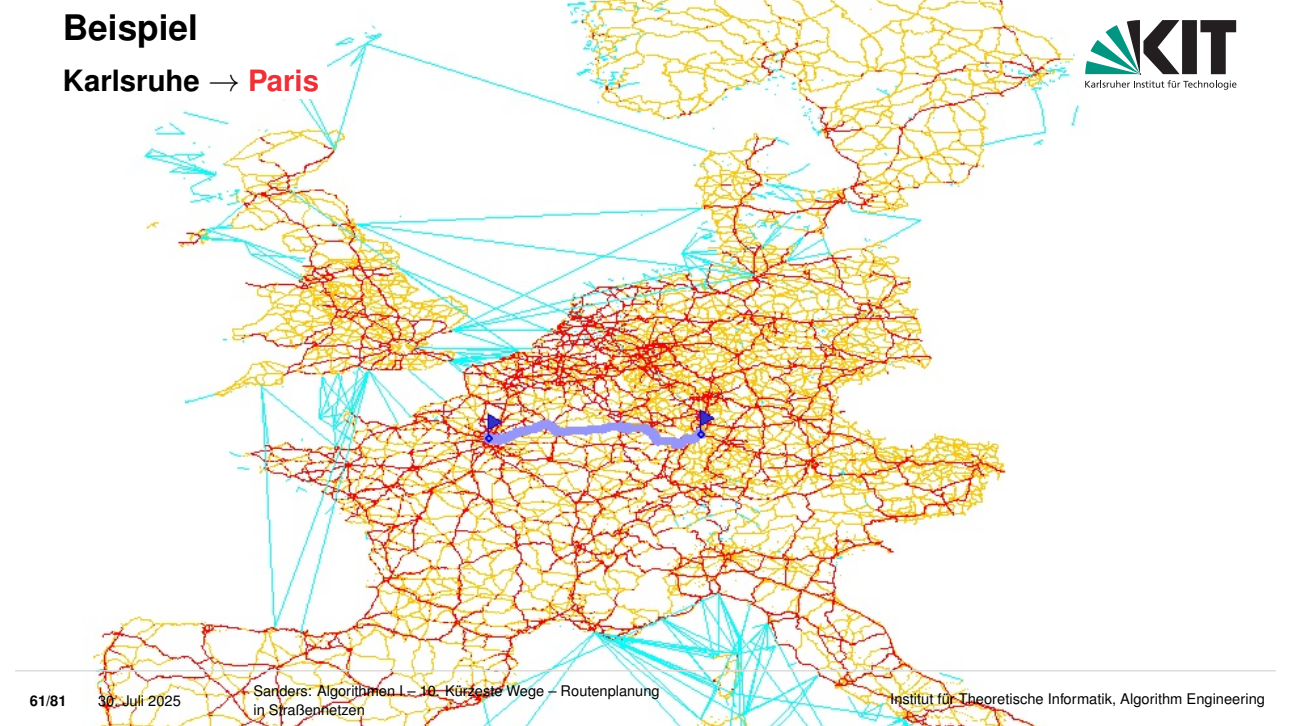
Beispiel

Karlsruhe → Rome



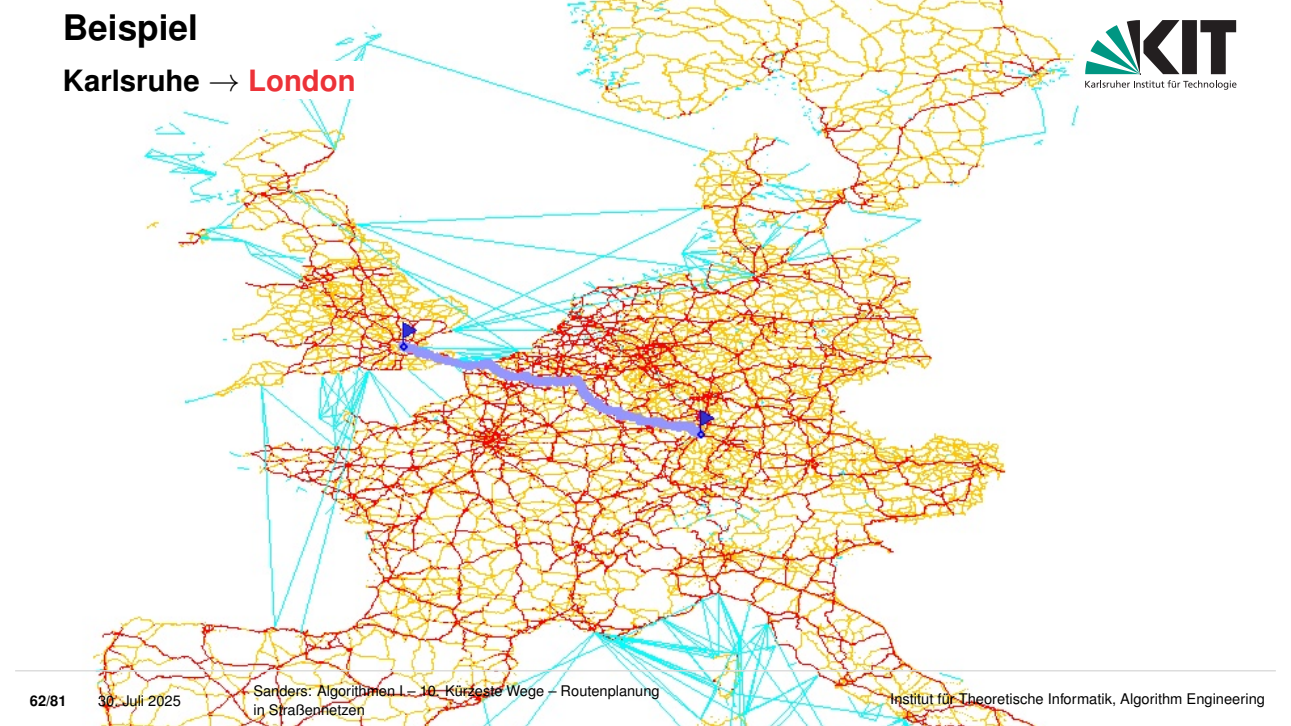
Beispiel

Karlsruhe → Paris



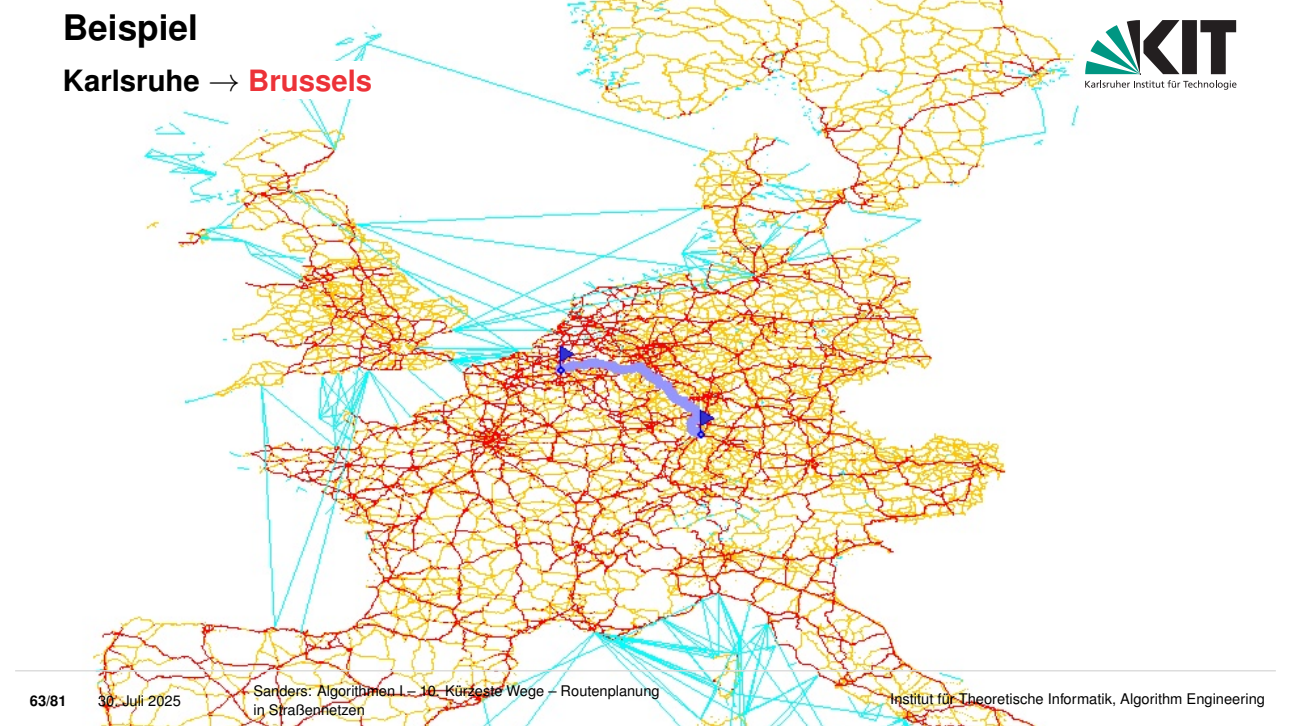
Beispiel

Karlsruhe → London



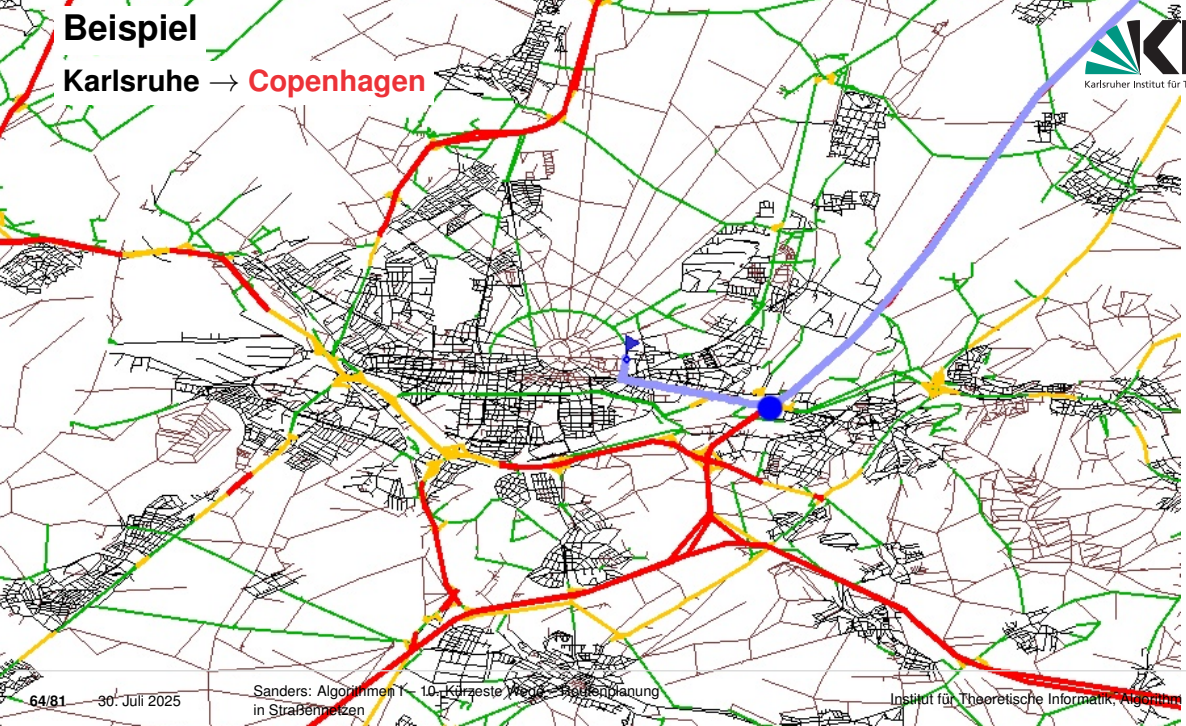
Beispiel

Karlsruhe → Brussels



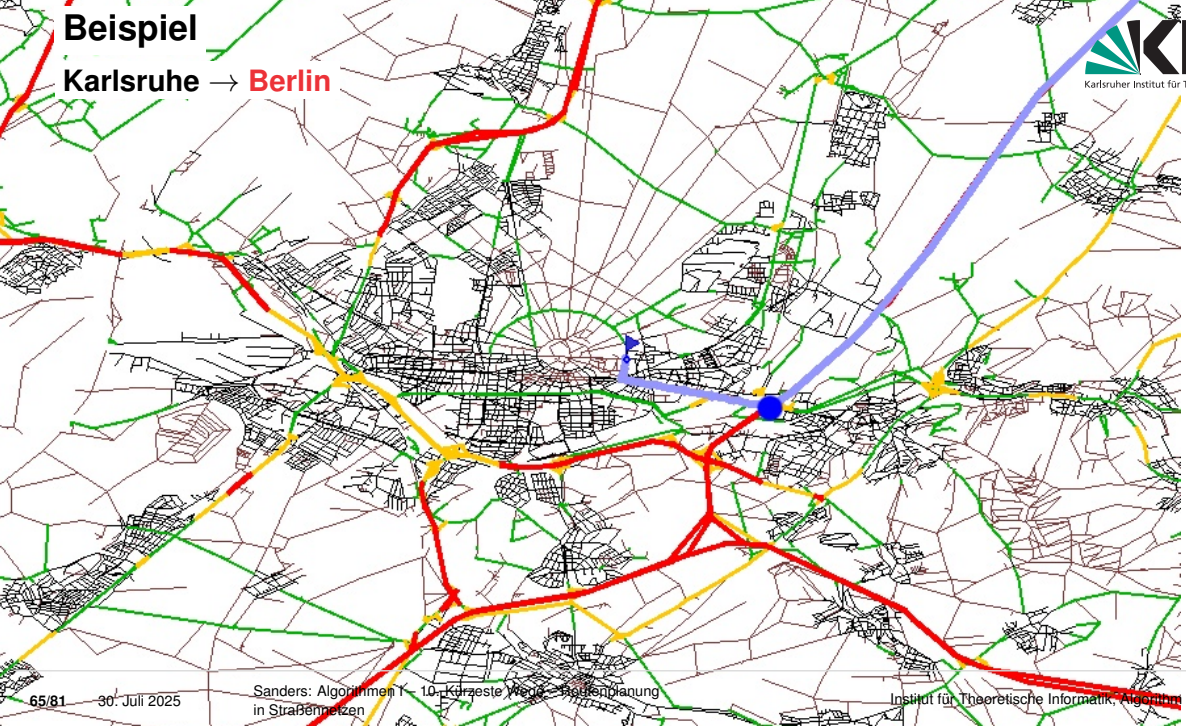
Beispiel

Karlsruhe → Copenhagen



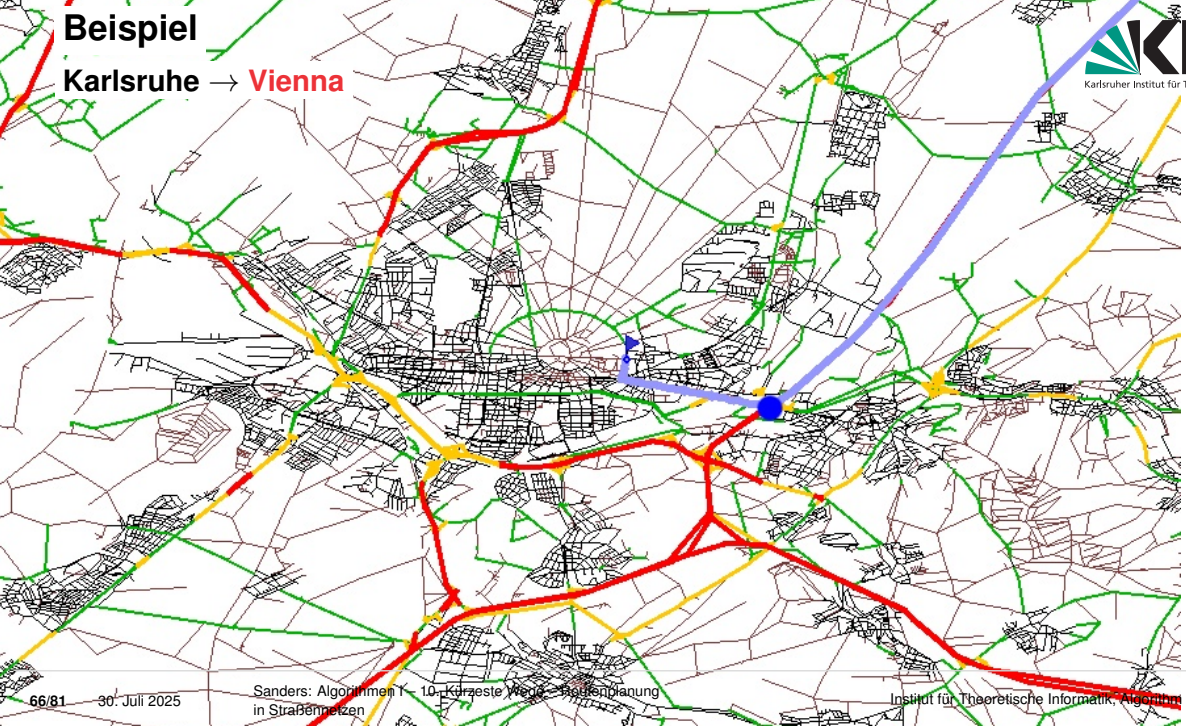
Beispiel

Karlsruhe → Berlin



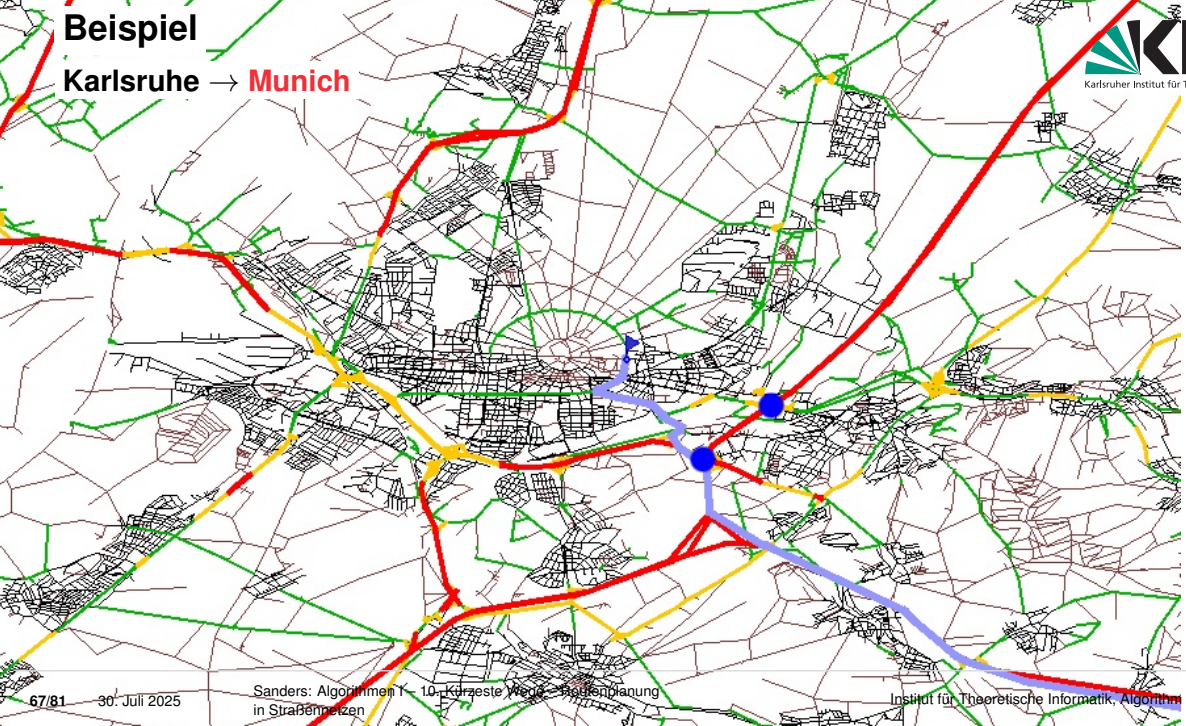
Beispiel

Karlsruhe → Vienna



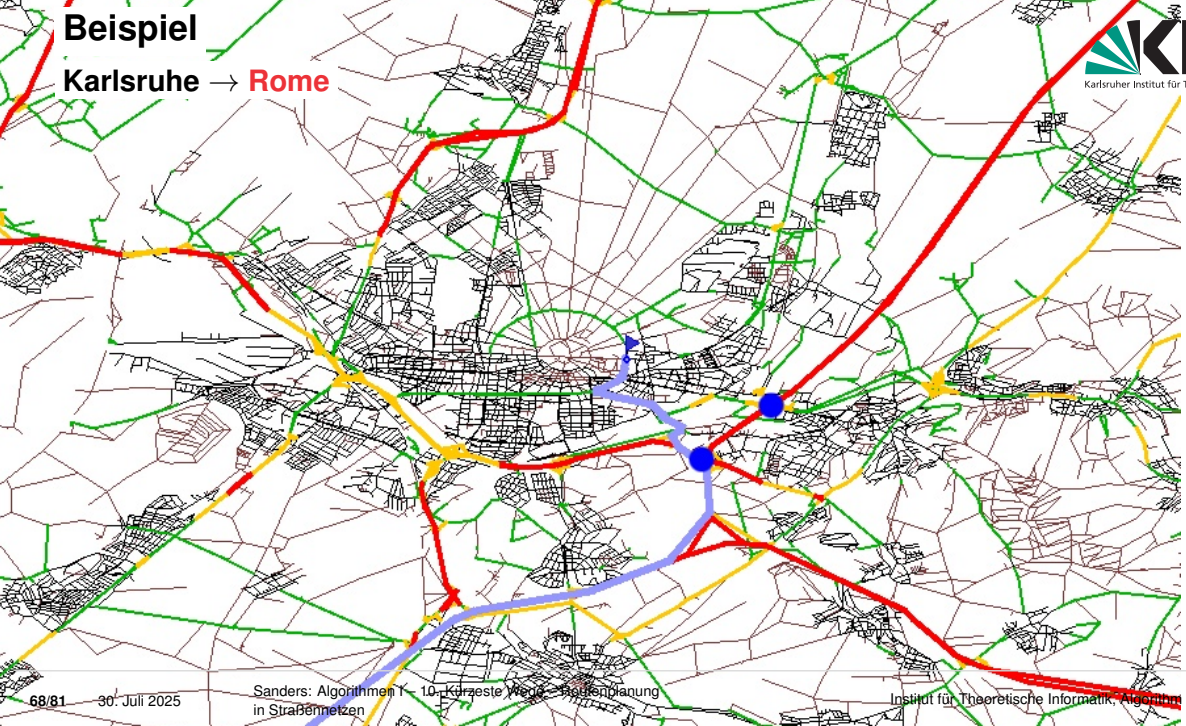
Beispiel

Karlsruhe → Munich



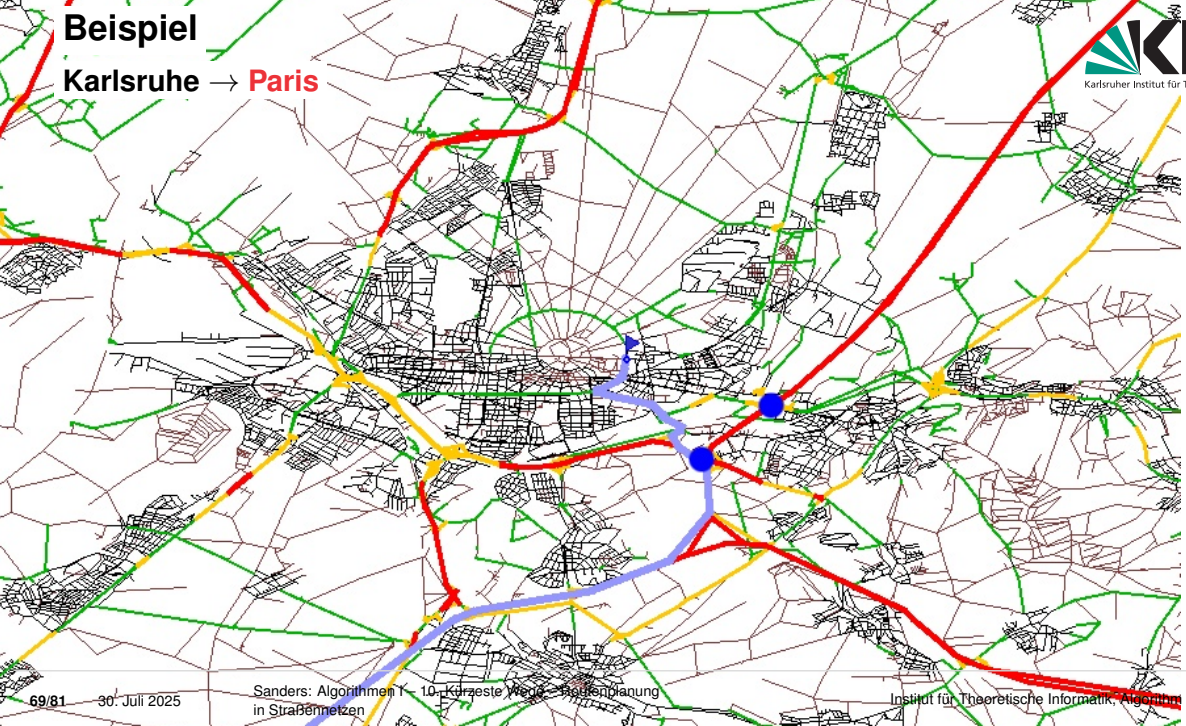
Beispiel

Karlsruhe → Rome



Beispiel

Karlsruhe → Paris

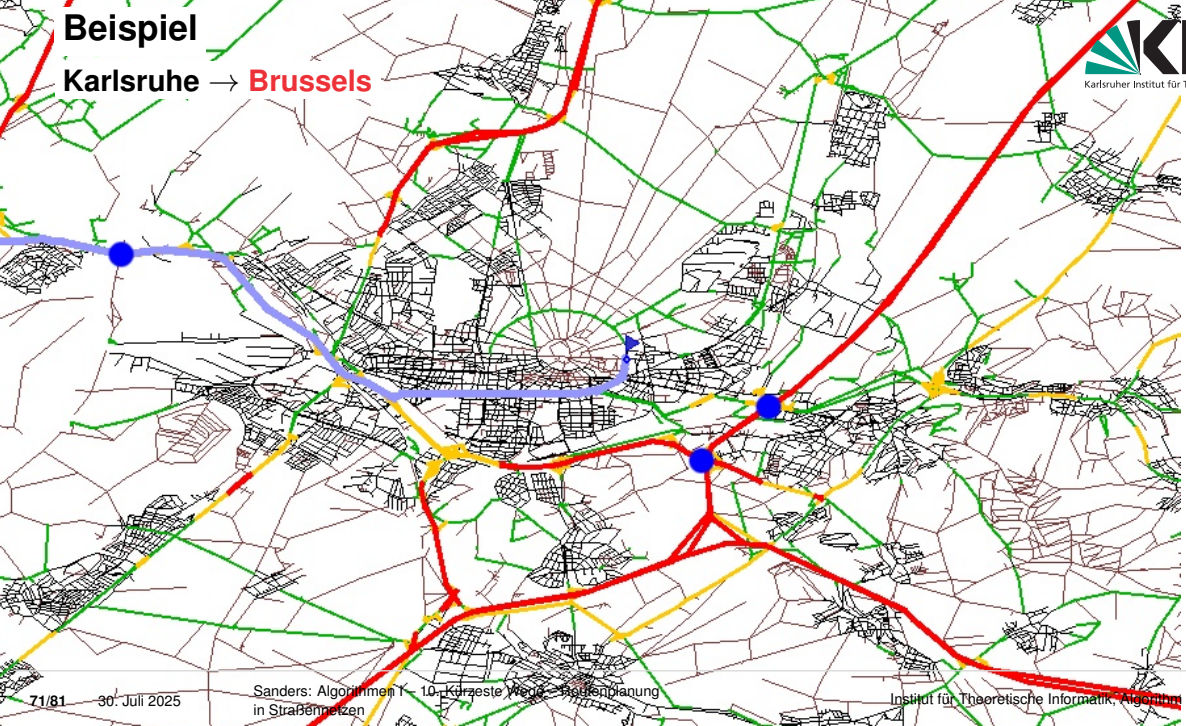


Beispiel

Karlsruhe → London

Beispiel

Karlsruhe → Brussels



Erste Beobachtung

Für *lange* Strecken

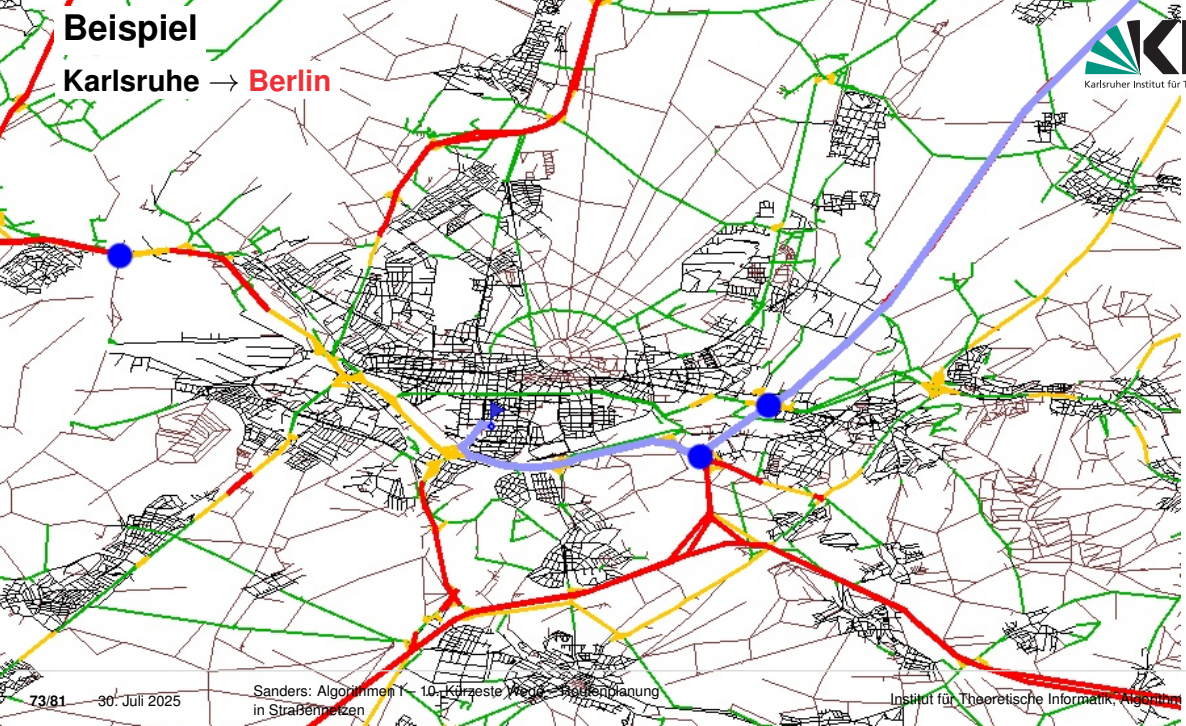
Nur *wenige „wichtige“* Zugänge zum Fernverkehrsnetzwerk (**access points**).

$\rightsquigarrow$ wir können alle Zugangspunkte vorberechnen

[in Europa: etwa 10 Zugangspunkte pro Knoten im Mittel]

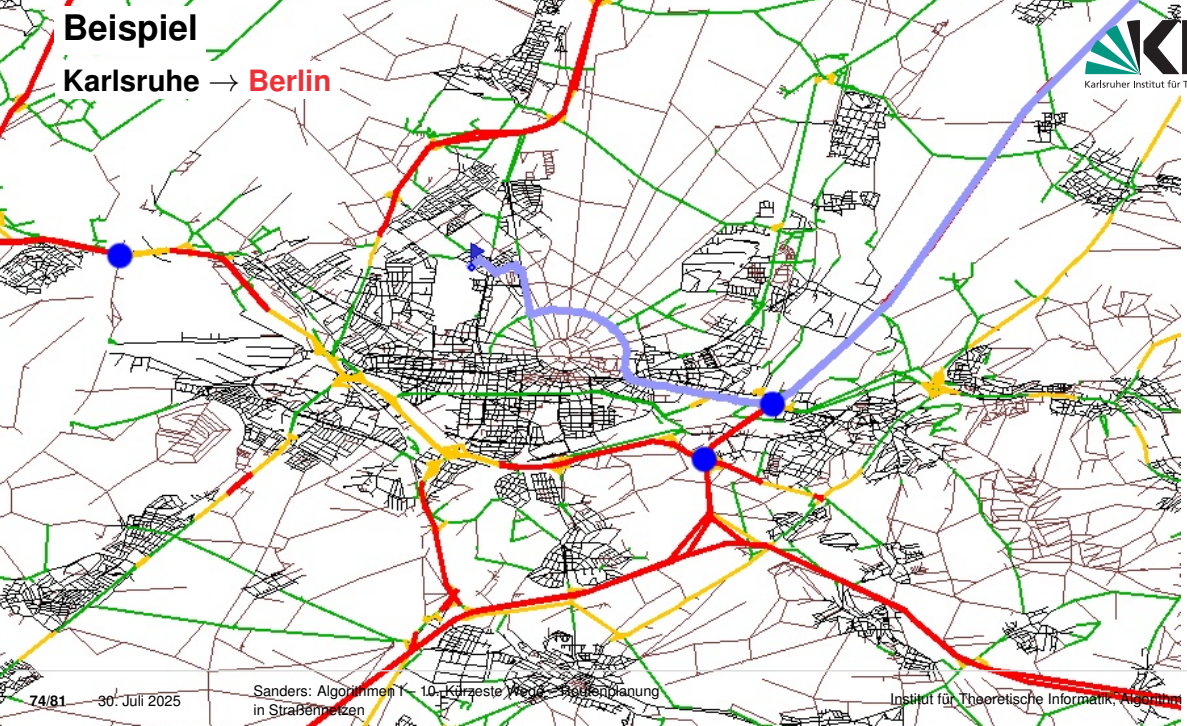
Beispiel

Karlsruhe → Berlin



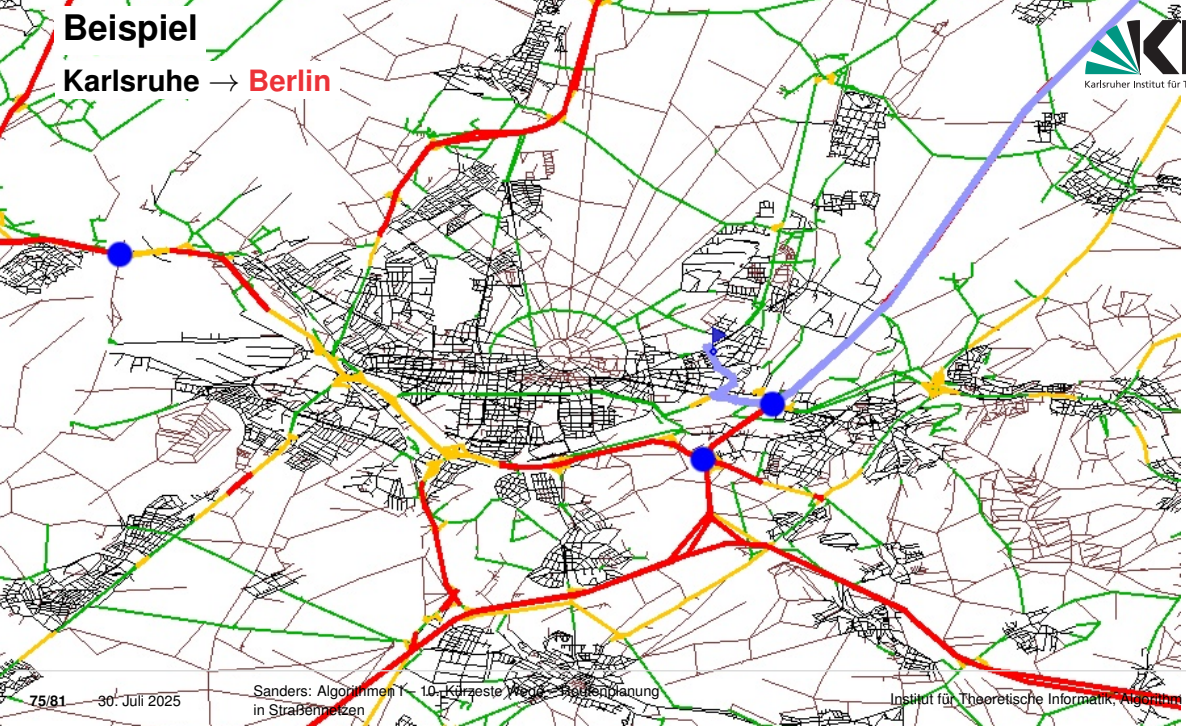
Beispiel

Karlsruhe → Berlin



Beispiel

Karlsruhe → Berlin



Zweite Beobachtung

Jeder Zugangspunkt ist für mehrere Knoten relevant

Gesamtmenge aller Zugangspunkte ist *klein*

Transitknotenmenge

$\rightsquigarrow$ wir können alle Abstände zwischen allen Transitknoten speichern)

[in Europa: $\approx 10\,000$ Transitknoten]

Transit-Node Routing

Preprocessing:

- Identifiziere Transitknoten $\mathcal{T} \subseteq V$
- Berechne $|\mathcal{T}| \times |\mathcal{T}|$ Abstandstabelle
- Für jeden Knoten: identifiziere Zugangsknoten (Abbildung $A : V \to 2^{\mathcal{T}}$)
- Speichere Abstände

Query (geg. Start s und Ziel t):

$$d_{\text{top}}(s,t) := \min \{d(s,u)+d(u,v)+d(v,t) : u \in A(s), v \in A(t)\}$$

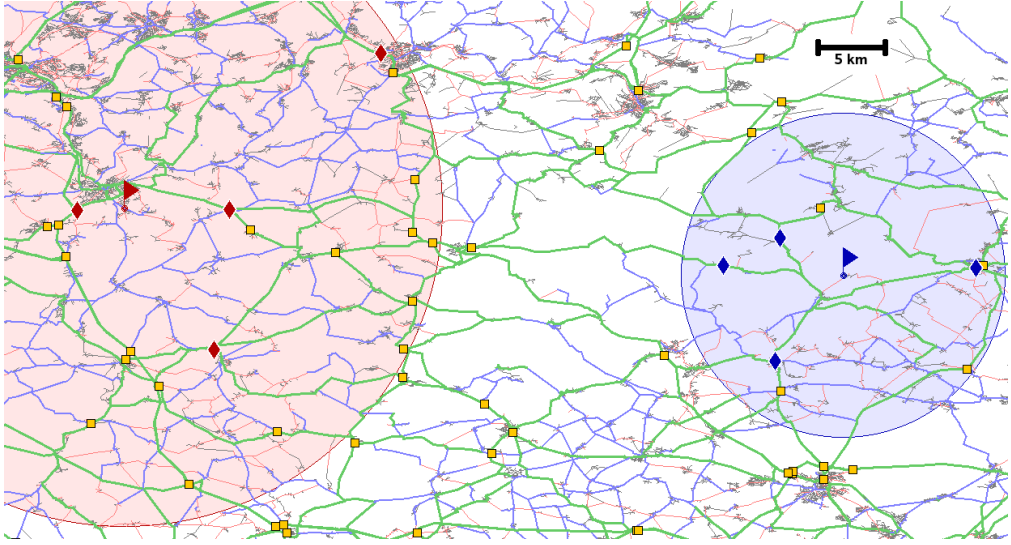
Transit-Node Routing

Lokalitätsfilter:
lokale Fälle ausfiltern ($\rightsquigarrow$ Spezialbehandlung)

$L : V \times V \to \{\text{true}, \text{false}\}$

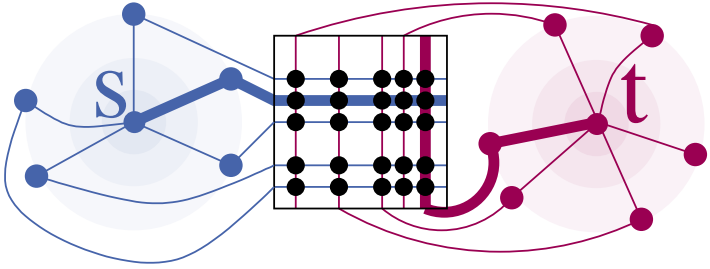
$$\neg L(s,t) \text{ impliziert } d(s,t) = d_{\text{top}}(s,t)$$

Transit-Node Routing: Beispiel



Transit-Node Routing: Experimente

- *sehr* schnelle queries
  (4 μs, > 1 000 000 mal schneller als DIJKSTRA)
- Gewinner der 9. DIMACS Implementation Challenge
- erträgliche Vorberechnungszeiten (1:15 h) und
  Speicherbedarf (247 bytes/Knoten)



Transit-Node Routing: Offene Fragen

- Wie bestimmt man die Transitknoten?
- Wie bestimmt man die Zugangsknoten effizient?
- Wie bestimmt man die Lokalitätsfilter?
- Wie handhabt man lokale Anfragen?

Transit-Node Routing: Offene Fragen

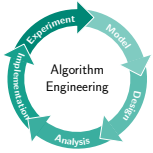
- Wie bestimmt man die Transitknoten?
- Wie bestimmt man die Zugangsknoten effizient?
- Wie bestimmt man die Lokalitätsfilter?
- Wie handhabt man lokale Anfragen?

Antwort:

- Andere Routenplanungstechniken benutzen!
- (Vertiefungsvorlesung “Algorithmen für Routenplanung”)

Index

Karlsruher Institut für Technologie



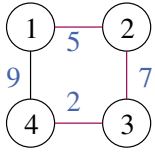
Algorithmen I – 11. Minimale Spannbäume

Sommersemester 2025

Peter Sanders | Stand: 30. Juli 2025

Minimale Spannbäume (MSTs)

ungerichteter (zusammenhängender) Graph $G = (V, E)$.
Knoten V, $n = |V|$, z.B., $V = \{1, \ldots, n\}$
Kanten $e \in E$, $m = |E|$, 2-elementige Teilmengen von V.
Kantengewichte $c(e) \in \mathbb{R}_+$.

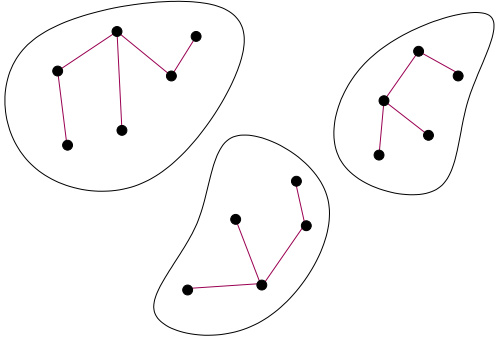


Finde Baum (V, T) mit minimalem Gewicht $\sum_{e \in T} c(e)$ der alle Knoten verbindet.

Minimale spannende Wälder (MSF)

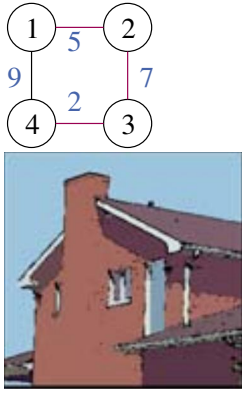
Falls G nicht zusammenhängend, finde minimalen spannenden Wald T der alle Zusammenhangskomponenten von G aufspannt.

MST Algorithmen lassen sich leicht zu MSF Algorithmen verallgemeinern.



Anwendungen

- Netzwerk-Entwurf
- Bottleneck-Shortest-Paths:
  Suche s–t-Pfad,
  dessen max. Kantengewicht minimal ist.
  Dies ist der Pfad im MST!
- Clustering: Lass schwere MST-Kanten weg.
  Teilbäume definieren Cluster. Konkret z. B. Bildsegmentierung
- Näherungslösungen für schwere Probleme, z. B.
  Handlungsreisendenproblem, Steinerbaumproblem.
  Siehe Buch, VL G. theoretischer Informatik, Algorithmen II.



Mehr zur Anwendung Bildsegmentierung

- Pixel $\rightarrow$ Knoten des Graphen
- Kanten zwischen “benachbarten” Pixeln
- “Kontrast” $\rightarrow$ Kantengewichte

Berechne MST
Lass (z.B.) MST Kanten mit Kontrast $\geq x$ weg.

Zusammenhangskomponenten des entstehenden Waldes
$\rightarrow$ Segmente des Bildes

Satz: Segmente haben intern Kontrast $< x$
Es gibt keine Brücken zwischen Segmenten mit Kontrast $< x$.

Explainable (Non)-AI

Heute wird Bildsegementierung oft mit neuronalen Netzwerken gemacht

- Bessere Segmentierung in vielen Benchmarks
- “Erklärbarkeit” wie bei MST und anderen graphtheoretischen Segmentierungsverfahren geht verloren

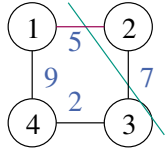
Wichtiges aktuelles Forschungsthema

MST-Kanten auswählen und verwerfen

Die Schnitteigenschaft (Cut Property)

Für beliebige Teilmenge $S \subset V$ betrachte die Schnittkanten
$C = \{\{u, v\} \in E : u \in S, v \in V \setminus S\}$

Die leichteste Kante e in C
kann in einem MST verwendet werden.

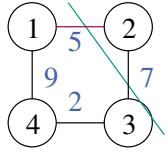


MST-Kanten auswählen und verwerfen

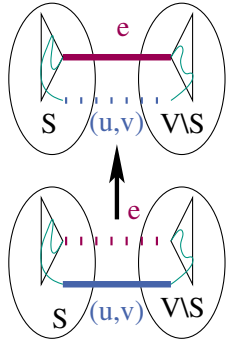
Die Schnitteigenschaft (Cut Property)

Für beliebige Teilmenge $S \subset V$ betrachte die Schnittkanten
$C = \{\{u, v\} \in E : u \in S, v \in V \setminus S\}$

Die leichteste Kante e in C
kann in einem MST verwendet werden.

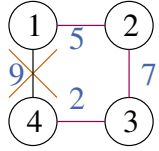


Beweis: Betrachte MST T'.
Fall $e \in T'$**:** Beweis fertig.
Sonst: $T' \cup \{e\}$ enthält Kreis K.
Betrachte eine Kante $\{u, v\} \in C \cap K \neq e$.
Dann ist $T = T' \setminus \{\{u, v\}\} \cup \{e\}$ ein Spannbaum
der nicht schwerer ist.



Die Kreiseigenschaft (Cycle Property)

Die schwerste Kante auf einem Kreis wird nicht für einen MST benötigt

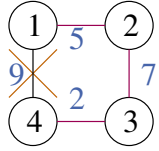


Die Kreiseigenschaft (Cycle Property)

Die schwerste Kante auf einem Kreis wird nicht für einen MST benötigt

Beweis. Angenommen MST T' benutzt die schwerste Kante e' auf Kreis C.
Wähle $e \in C$ mit $e \notin T'$.
Es gilt $c(e) \leq c(e')$.
Dann ist $T = T' \setminus \{e'\} \cup \{e\}$ auch ein MST.

□

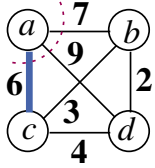


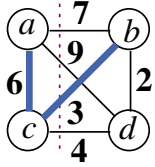
Der Jarník-Prim Algorithmus

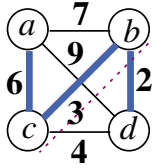
[Jarník 1930, Prim 1957]

Idee: Lasse einen Baum wachsen

$T := \emptyset$
$S := \{s\}$ for arbitrary start node s
repeat $n - 1$ times
    find (u, v) fulfilling the cut property for S
    $S := S \cup \{v\}$
    $T := T \cup \{(u, v)\}$

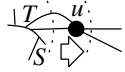


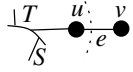




Der Jarník-Prim Algorithmus

```
Function jpMST : Set of Edge                                   // weitgehend analog zu Dijkstra
    pick any s ∈ V
    d = {∞, ..., ∞}; parent[s]:= s; d[s] := 0;    Q.insert(s)
    while Q ≠ ∅ do
        u := Q.deleteMin
        d[u]:= 0
        //scan u
        foreach edge e = (u, v) ∈ E do
            if c(e) < d[v] then                                // relax
                d[v]:= c(e)
                parent[v] := u                                 // update tree
                if v ∈ Q then Q.decreaseKey(v)
                else Q.insert(v)
    return {(v, parent[v]) : v ∈ V \ {s}}
```





Der Jarník-Prim Algorithmus
Analyse

Praktisch identisch zu Dijkstra

- $O(m+n)$ Zeit ausserhalb der PQ
- n deleteMin (Zeit $O(n \log n)$)
- $O(m)$ decreaseKey

$\rightsquigarrow$ $O((m+n)\log n)$ mit binären Heaps

$\rightsquigarrow$ $O(m + n\log n)$ mit Fibonacci Heaps

Wichtigster Unterschied: monotone PQs reichen nicht

Warum?

Kruskals Algorithmus [1956]

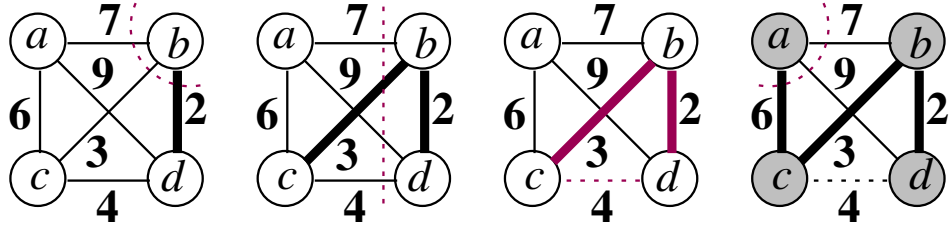
$T := \emptyset$ // subforest of the MST

foreach $(u, v) \in E$ in ascending order of weight **do**

  if u and v are in different subtrees of (V, T) **then**

    $T := T \cup \{(u, v)\}$ // Join two subtrees

return T



Kruskals Algorithmus
Korrektheit

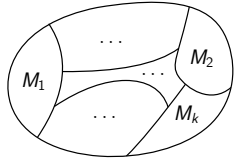
$T := \emptyset$ // subforest of the MST
foreach $(u, v) \in E$ in ascending order of weight **do**
  if u and v are in different subtrees of (V, T) **then**
    $T := T \cup \{(u, v)\}$ // Join two subtrees
return T

Fall u, v in verschiedenen Teilbäumen: benutze Schnitteigenschaft
$\Longrightarrow (u, v)$ ist leichteste Kante im cut (Komponente(u), $V \setminus$ Komponente(u))
$\Longrightarrow (u, v) \in$ MST
Sonst: benutze Kreiseigenschaft
$\Longrightarrow (u, v)$ ist schwerste Kante im Kreis $\langle u, v, v$–u-Pfad in $T\rangle$
$\Longrightarrow (u, v) \notin$ MST □

Union-Find Datenstruktur

Verwalte Partition der Menge $1..n$, d. h., Mengen (Blocks) $M_1,\ldots,M_k$ mit

$$M_1 \cup \cdots \cup M_k = 1..n,$$
$$\forall i \neq j : M_i \cap M_j = \emptyset$$



Class UnionFind($n : \mathbb{N}$)

Procedure union($i, j : 1..n$)

join the blocks containing i and j to a single block.

Function find($i : 1..n$) : $1..n$

return a unique identifier for the block containing i.

Union-Find Datenstruktur

Erste Version

Class UnionFind($n : \mathbb{N}$)

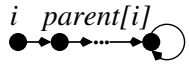
parent$=\langle 1, 2, \ldots, n \rangle$: **Array** $[1..n]$ **of** $1..n$

invariant parent-refs lead to unique Partition-Reps

Function find($i : 1..n$) : $1..n$

if parent$[i] = i$ **then return** i

else return find(parent$[i]$)



Union-Find Datenstruktur

Erste Version

Class UnionFind($n : \mathbb{N}$)

parent$=\langle 1, 2, \ldots, n\rangle$: **Array** $[1..n]$ **of** $1..n$

invariant parent-refs lead to unique Partition-Reps

Function find($i : 1..n$) : $1..n$

if parent$[i] = i$ **then return** i

else return find(parent$[i]$)

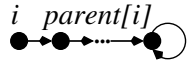
Procedure link($i, j : 1..n$)

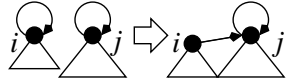
assert i and j are representatives of different blocks

parent$[i] := j$

Procedure union($i, j : 1..n$)

if find$(i) \neq$ find(j) **then** link(find(i), find(j))





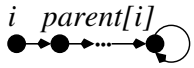
Union-Find Datenstruktur
Erste Version

Analyse:

+: union braucht konstante Zeit

−: find braucht Zeit $\Theta(n)$ im schlechtesten Fall !

zu langsam.



Idee: find-Pfade kurz halten

Union-Find Datenstruktur
Pfadkompression

Class UnionFind($n : \mathbb{N}$)

parent$=\langle 1, 2, \ldots, n \rangle$: **Array** $[1..n]$ **of** $1..n$

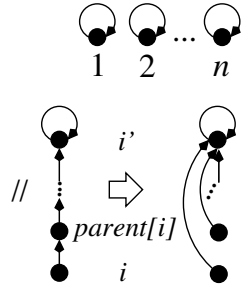
Function find($i : 1..n$) : $1..n$

if parent$[i] = i$ **then return** i

else $i' :=$ find(parent$[i]$)

parent$[i] := i'$

return i'



Union-Find Datenstruktur
Union by Rank

Class UnionFind($n : \mathbb{N}$)

parent$=\langle 1, 2, \ldots, n\rangle$: **Array** $[1..n]$ **of** $1..n$

rank$=\langle 0, \ldots, 0\rangle$: **Array** $[1..n]$ **of** $0..\log n$

Procedure link($i, j : 1..n$)

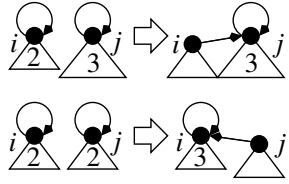
assert i and j are representatives of different blocks

if rank$[i] <$ rank$[j]$ **then** parent$[i] := j$

else

parent$[j] := i$

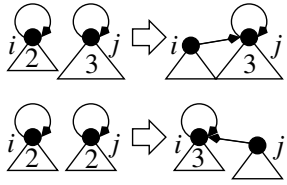
if rank$[i] =$ rank$[j]$ **then** rank$[i]$++



Union-Find Datenstruktur
Analyse – nur Union-by-rank

invariant Der Pfad zum Repr. x hat Länge höchstens $\text{rank}[x]$
invariant x ist Repr. $\Rightarrow$ x's Menge hat Größe mindestens $2^{\text{rank}[x]}$

Korollar. find *braucht Zeit* $\mathrm{O}(\log n)$



Union-Find Datenstruktur
Analyse – nur Pfadkompression

Satz. find *braucht Zeit* $\mathrm{O}(\log n)$ *(amortisiert)*

Beweis. im Buch □

Union-Find Datenstruktur

Analyse – Pfadkompression + Union-by-rank

Satz. *$m\times$ find $+$ $n\times$ link brauchen Zeit $O(m\alpha_T(m,n))$ mit*

$$\alpha_T(m,n) = \min\{i \geq 1 : A(i, \lceil m/n \rceil) \geq \log n\},$$

und

$$\begin{aligned} A(1,j) &= 2^j && \text{for } j \geq 1, \\ A(i,1) &= A(i-1,2) && \text{for } i \geq 2, \\ A(i,j) &= A(i-1, A(i,j-1)) && \text{for } i \geq 2 \text{ and } j \geq 2. \end{aligned}$$

Beweis. [Tarjan 1975, Seidel Sharir 2005] □

A ist die Ackermannfunktion und α_T die inverse Ackermannfunktion.
$\alpha_T(m,n) = \omega(1)$ aber ≤ 4 für alle physikalisch denkbaren n, m.

Ackermannfunktion – Beispiele

$$A(1,j) = 2^j \quad \text{for } j \geq 1,$$
$$A(i,1) = A(i-1,2) \quad \text{for } i \geq 2,$$
$$A(i,j) = A(i-1, A(i,j-1)) \quad \text{for } i \geq 2 \text{ and } j \geq 2.$$

$$A(2,1) = A(1,2) = 2^2$$
$$A(2,2) = A(1, A(2,1)) = 2^{2^2}$$
$$A(2,3) = A(1, A(2,2)) = 2^{2^{2^2}}$$
$$A(2,4) = A(1, A(2,3)) = 2^{2^{2^{2^2}}}$$
$$A(3,1) = A(2,2) = 2^{2^2}$$
$$A(3,2) = A(2, A(3,1) = A(2,16) = ???$$
$$A(4,1) = A(3,2) = ???$$

Kruskal mit Union-Find

Sei $V = 1..n$

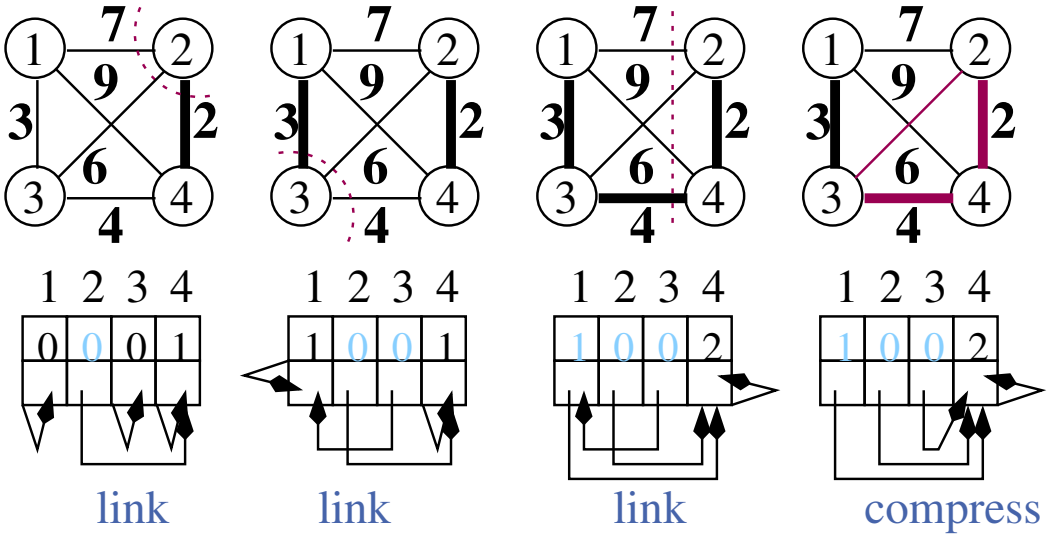
```
Tc : UnionFind(n)                                   // encodes components of forest T
foreach (u, v) ∈ E in ascending order of weight do  // sort
    if Tc.find(u) ≠ Tc.find(v) then
        output {u, v}
        Tc.union(u, v)                              // link reicht auch
```

Zeit $O(m \log m)$. Schneller für ganzzahlige Gewichte.

Graphrepräsentation: Kantenliste

Bäume im MSF $\leftrightarrow$ Blöcke in Partition $\rightarrow$ Wurzelbäume
aber mit anderer Struktur als die Bäume im MSF

Beispiel


1 2 3 4
0 0 0 1
1 2 3 4
1 0 0 1
1 2 3 4
1 0 0 2
1 2 3 4
1 0 0 2
link
link
link
compress

Vergleich Jarník-Prim $\leftrightarrow$ Kruskal

Pro Jarník-Prim

- Asymptotisch gut für alle m, n
- Sehr schnell für $m \gg n$

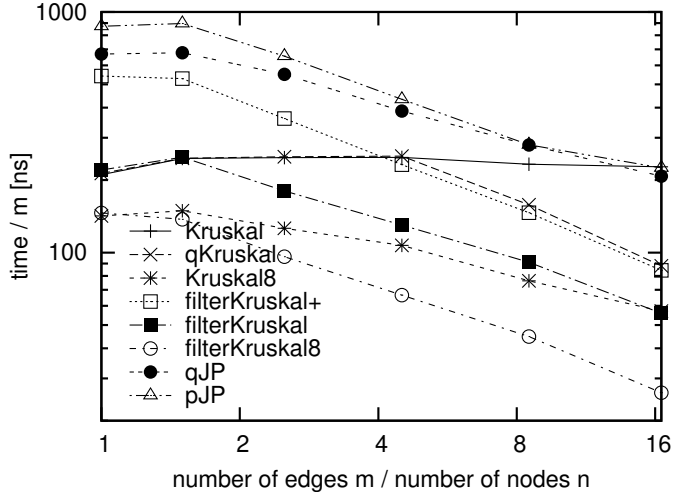
Pro Kruskal

- Gut für $m = \mathrm{O}(n)$
- Braucht nur Kantenliste
- Profitiert von schnellen Sortierern (ganzzahlig, parallel,...)
- Verfeinerungen auch gut für große m/n

Mehr MST-Algorithmen

- Zeit $O(m \log n)$ [Boruvka 1926]
  Zutat vieler fortgeschrittener Algorithmen
- Erwartete Zeit $O(m)$ [Karger Klein Tarjan 1995],
  parallelisierbar, externalisierbar
- Det. Zeit $O(m\alpha_T(m,n))$ [Chazelle 2000]
- “optimaler” det. Algorithmus [Pettie, Ramachandran 2000]
- “Invented here”:
  Praktikabler externer Algorithmus [Sanders Schultes Sibeyn 2004]
  Verbesserung von Kruskal (parallelisierbar, weniger Sortieraufwand). [Osipov Sanders Singler 2009]

Messungen, Zufallsgraph, $n = 2^{22}$

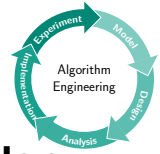


Zusammenfassung

- Schnitt- und Kreiseigenschaft als Basis für abstrakte Algorithmen.
  Entwurfsprinzip: benutze abstrakte Problemeigenschaften.
- Beweise mittels Austauschargumenten
- Implementierung braucht effiziente Datenstrukturen.
  Auch ein Entwurfsprinzip...
- Dijkstra $\approx$ JP.
  Noch ein Entwurfsprinzip:
  Greedy-Algorithmus effizient implementiert mittels Prioritätsliste
- Union-Find: effiziente Verwaltung von Partitionen mittels Pfadkompression und Union-by-rank.
  Beispiel für einfache Algorithmen mit nichttrivialer Analyse

Index

Karlsruher Institut für Technologie



Algorithmen I – 12. Generische Optimierungsmethoden

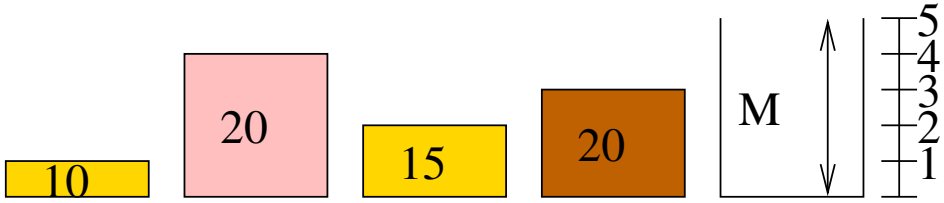
Sommersemester 2025

Peter Sanders | Stand: 30. Juli 2025

Generische Optimierungsmethoden

- Black-Box-Löser
- Greedy
- Dynamische Programmierung
- Systematische Suche
- Lokale Suche
- Evolutionäre Algorithmen

Durchgehendes Beispiel: Rucksackproblem



- n Gegenstände mit Gewicht $w_i \in \mathbb{N}$ und profit p_i
- Wähle eine Teilmenge **x** von Gegenständen
- so dass $\sum_{i \in \mathbf{x}} w_i \leq M$ und
- maximiere den Profit $\sum_{i \in \mathbf{x}} p_i$

Allgemein: Maximierungsproblem $(\mathcal{L}, f)$

- $\mathcal{L} \subseteq \mathcal{U}$: zulässige Lösungen
- $f : \mathcal{L} \to \mathbb{R}$ Zielfunktion
- $\mathbf{x}^* \in \mathcal{L}$ ist optimale Lösung falls $f(\mathbf{x}^*) \geq f(\mathbf{x})$ für alle $\mathbf{x} \in \mathcal{L}$

Minimierungsprobleme: analog

Problem: variantenreich, meist NP-hart

Generische Optimierungsmethoden

- Black-Box-Löser
- Greedy
- Dynamische Programmierung
- Systematische Suche
- Lokale Suche
- Evolutionäre Algorithmen

Black-Box-Löser

- (Ganzzahlige) Lineare Programmierung
- Aussagenlogik
- Constraint-Programming $\approx$ Verallgemeinerung von beidem
- QUBO – Quadratic Unconstrained Binary Optimization, z.B. mittel Quantum-Annealing
- Verschiedene Varianten von convexer Programmierung

Lineare Programmierung

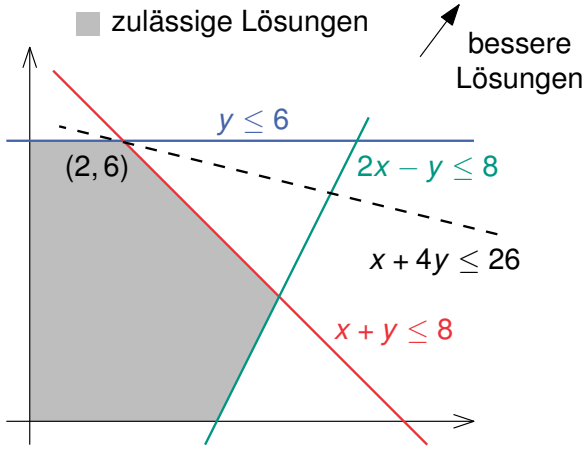
Ein lineares Programm mit n Variablen und m Constraints wird durch das folgende Minimierungs/Maximierungsproblem definiert:

- Kostenfunktion $f(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x}$
  $\mathbf{c}$ ist der Kostenvektor
- m constraints der Form $\mathbf{a}_i \cdot \mathbf{x} \bowtie_i b_i$ mit $\bowtie_i \in \{\leq, \geq, =\}$, $\mathbf{a}_i \in \mathbb{R}^n$ Wir erhalten

$$\mathcal{L} = \{\mathbf{x} \in \mathbb{R}^n : \forall j \in 1..n : x_j \geq 0 \wedge \forall i \in 1..m : \mathbf{a}_i \cdot \mathbf{x} \bowtie_i b_i\} \quad .$$

Sei a_{ij} die j-te Komponente von Vektor $\mathbf{a}_i$.

Ein einfaches Beispiel

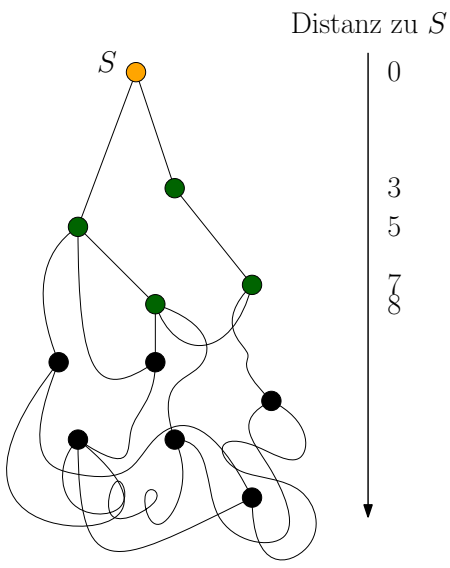


Beispiel: Kürzeste Wege

maximiere $\sum_{v \in V} d_v$

so dass $d_s = 0$

$d_w \leq d_v + c(v, w)$ für alle $(v, w) \in E$



Eine Anwendung – Tierfutter

- n Futtersorten.
  Sorte i kostet c_i Euro/kg.
- m Anforderungen an gesunde Ernährung.
  - (Kalorien, Proteine, Vitamin C,...)
  - Sorte i enthält a_{ji} Prozent des täglichen Bedarfs pro kg bzgl. Anforderung j
- Definiere x_i als zu beschaffende Menge von Sorte i
- LP-Lösung gibt eine kostenoptimale “gesunde” Mischung.

Verfeinerungen

- Obere Schranken (Radioaktivität, Cadmium, Kuhhirn, ...)
- Beschränkte Reserven (z. B. eigenes Heu)
- bestimmte abschnittweise lineare Kostenfunktionen (z. B. mit Abstand wachsende Transportkosten)

Grenzen

- Minimale Abnahmemengen
- die meisten nichtlinearen Kostenfunktionen
- Ganzzahlige Mengen (für wenige Tiere)
- Garbage in Garbage out

Algorithmen und Implementierungen

- LPs lassen sich in polynomieller Zeit lösen [Khachiyan 1979]
- Worst case $\mathrm{O}\left(\max(m,n)^{\frac{7}{2}}\right)$
- In der Praxis geht das viel schneller
- Robuste, effiziente Implementierungen sind sehr aufwändig

$\rightsquigarrow$ Fertige freie und kommerzielle Pakete

Ganzzahlige Lineare Programmierung

- ILP: Integer Linear Program, lineares Programm mit der zusätzlichen Bedingung $x_i \in \mathbb{N}$. oft: 0/1 ILP mit $x_i \in \{0, 1\}$
- MILP: Mixed Integer Linear Program, lineares Programm bei dem einige Variablen ganzzahlig sein müssen.
- Lineare Relaxation: Entferne die Ganzzahligkeitsbedingungen eines (M)ILP

Beispiel: Rucksackproblem

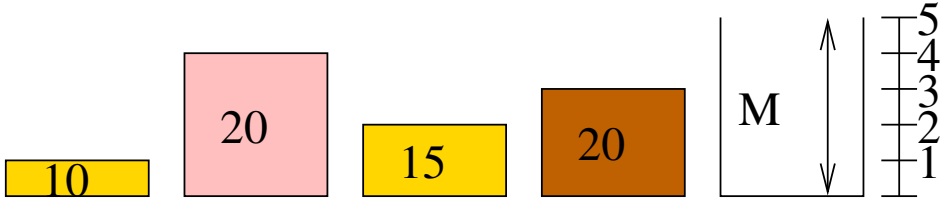
maximiere $\mathbf{p} \cdot \mathbf{x}$

so dass

$$\mathbf{w} \cdot x \leq M, x_i \in \{0, 1\} \text{ for } 1 \leq i \leq n \ .$$

$x_i = 1$ gdw Gegenstand i in den Rucksack kommt.

0/1 Variablen sind typisch für ILPs



Umgang mit (M)ILPs

- NP-hart
+ Ausdrucksstarke Modellierungssprache
+ Es gibt generische Lösungsmethoden, die manchmal gut funktionieren
+ Viele Möglichkeiten für Näherungslösungen
+ Die Lösung der linearen Relaxierung hilft oft, z. B. einfach runden.
+ Ausgefeilte Softwarepakete

Beispiel: Beim Rucksackproblem gibt es nur eine fraktionale Variable in der linearen Relaxierung – Abrunden ergibt zulässige Lösung. Annähernd optimal falls Gewichte und Profite $\ll$ Kapazität

Generische Optimierungsmethoden

- Black-Box-Löser
- Greedy
- Dynamische Programmierung
- Systematische Suche
- Lokale Suche
- Evolutionäre Algorithmen

Nie Zurückschauen – Greedy-Algorithmen
(deutsch: gierige Algorithmen, wenig gebräuchlich)

Idee: treffe jeweils eine lokal optimale Entscheidung

Optimale Greedy-Algorithmen

- Dijkstra's Algorithmus für kürzeste Wege
- Minimale Spannbäume
  - Jarník-Prim
  - Kruskal
- Selection-Sort (wenn man so will)

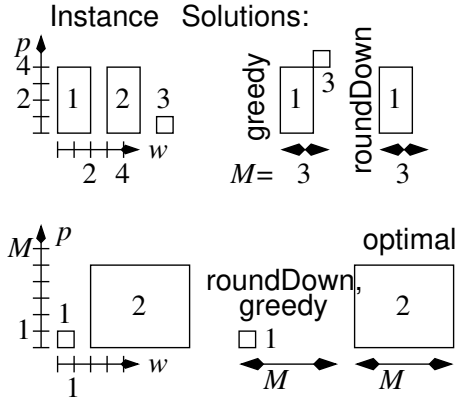
Näherungslösungen mit Greedy-Algorithmen

- Viel häufiger, z.T. mit Qualitätsgarantien.
- Mehr: Algorithmen II

Beispiel: Rucksackproblem

Procedure roundDownKnapsack
sort items by profit density $\frac{p_i}{w_i}$
find max $\left\{ j : \sum_{i=1}^{j} w_j \right\} > M$ // critical item
output items $1..j-1$

Procedure greedyKnapsack
sort items by profit density $\frac{p_i}{w_i}$
for $i := 1$ **to** n **do**
if there is room for item i **then**
insert it into the knapsack



Generische Optimierungsmethoden

- Black-Box-Löser
- Greedy
- Dynamische Programmierung
- Systematische Suche
- Lokale Suche
- Evolutionäre Algorithmen

Dynamische Programmierung – Aufbau aus Bausteinen

Anwendbar wenn, das **Optimalitätsprinzip** gilt:

- Optimale Lösungen bestehen aus optimalen Löungen für Teilprobleme.
- Mehrere optimale Lösungen $\Rightarrow$ es is egal welche benutzt wird.

Beispiel: Rucksackproblem

Annahme: ganzzahlige Gewichte
$P(i, C)$:= optimaler Profit für Gegenstände $1,\ldots,i$ unter Benutzung von Kapazität $\leq C$.
$P(0, C)$:= 0

Lemma:

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

$P(i, C)$:= optimaler Profit für Gegenstände $1,\ldots,i$ bei Kap. C.

Lemma: $P(i, C) = \max(P(i-1, C), P(i-1, C-w_i) + p_i)$

Beweis:

Sei **x** optimale Lösung für Objekte $1..i$, Kapazität C,

d.h. $\mathbf{c} \cdot \mathbf{x} = P(i, C)$.

Fall $x_i = 0$:

$\Rightarrow$ **x** ist auch (opt.) Lösung für Objekte $1..i-1$, Kapazität C.

$\Rightarrow P(i, C) = \mathbf{c} \cdot \mathbf{x} = P(i-1, C)$

$P(i, C)$:= optimaler Profit für Gegenstände $1,\ldots,i$ bei Kap. C.

Lemma: $P(i, C) = \max(P(i-1, C), P(i-1, C-w_i) + p_i)$

Beweis:

Sei **x** optimale Lösung für Objekte $1..i$, Kapazität C,

d.h. $\mathbf{c} \cdot \mathbf{x} = P(i, C)$.

Fall $x_i = 0$: $P(i, C) = \mathbf{c} \cdot \mathbf{x} = P(i-1, C)$

Fall $x_i = 1$:

$\Rightarrow$ **x** ohne i ist auch Lösung für Objekte $1..i-1$, Kapazität $C - w_i$.

Wegen Austauschbarkeit muß **x** ohne i optimal für diesen Fall sein.

$\Rightarrow P(i, C) - p_i = P(i-1, C-w_i)$

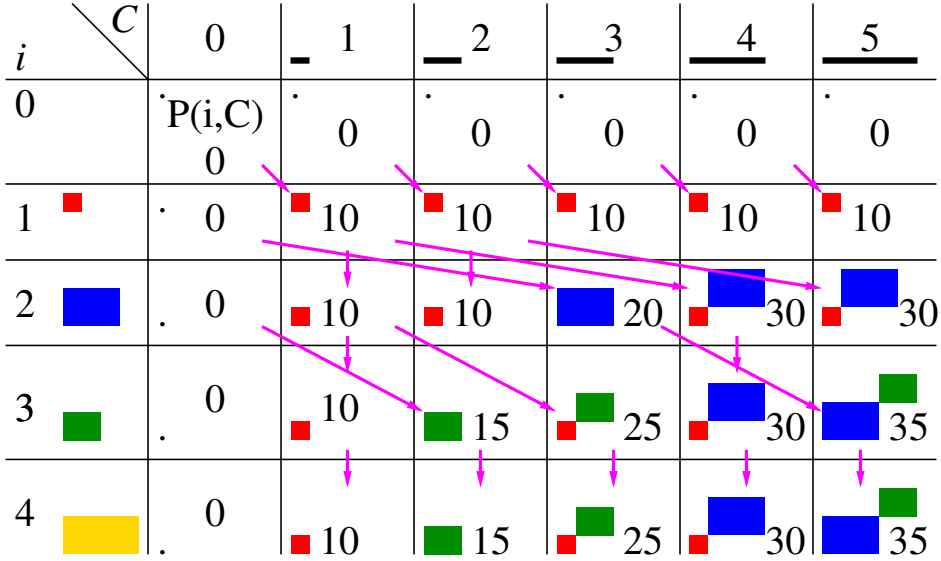
$\Leftrightarrow P(i, C) = P(i-1, C-w_i) + p_i$

$P(i, C)$:= optimaler Profit für Gegenstände $1,\ldots,i$ bei Kap. C.

Lemma: $P(i, C) = \max(P(i-1, C), P(i-1, C-w_i) + p_i)$
Beweis:
Sei **x** optimale Lösung für Objekte $1..i$, Kapazität C,
d.h. $\mathbf{c} \cdot \mathbf{x} = P(i, C)$.
Fall $x_i = 0$: $P(i, C) = \mathbf{c} \cdot \mathbf{x} = P(i-1, C)$
Fall $x_i = 1$:
$\Rightarrow$ **x** ohne i ist auch Lösung für Objekte $1..i-1$, Kapazität $C - w_i$.
Wegen Austauschbarkeit muß **x** ohne i optimal für diesen Fall sein.
$\Rightarrow P(i, C) - p_i = P(i-1, C-w_i)$
$\Leftrightarrow P(i, C) = P(i-1, C-w_i) + p_i$

Insgesamt, wegen Optimalität von **x**,
$P(i, C) = \max(P(i-1, C), P(i-1, C-w_i) + p_i)$ □

Beispiel: maximiere $(10, 20, 15, 20) \cdot \mathbf{x}$ so dass $(1, 3, 2, 4) \cdot \mathbf{x} \leq 5$



Berechung von $P(i, C)$ elementweise:

Procedure knapsack(**p**, **c**, n, M)
    array P$[0 \ldots M] = [0, \ldots, 0]$
    bitarray decision$[1 \ldots n, 0 \ldots M] = [(0, \ldots, 0), \ldots, (0, \ldots, 0)]$
    for i := 1 **to** n **do**
        //invariant: $\forall C \in \{1, \ldots, M\} : \mathrm{P}[C] = P(i-1, C)$
        for $C := M$ **downto** w_i **do**
            if $\mathrm{P}[C - w_i] + p_i > \mathrm{P}[C]$ **then**
                $\mathrm{P}[C] := \mathrm{P}[C - w_i] + p_i$
                decision$[i, C]$:= 1

Beispiel

maximiere $(10, 20, 15, 20) \cdot \mathbf{x}$

so dass $(1, 3, 2, 4) \cdot \mathbf{x} \leq 5$

$P(i, C), (\text{decision}[i, C])$

$i \setminus C$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0, (0)	10, (1)	10, (1)	10, (1)	10, (1)	10, (1)
2						
3						
4						

Beispiel

maximiere $(10, 20, 15, 20) \cdot \mathbf{x}$

so dass $(1, 3, 2, 4) \cdot \mathbf{x} \leq 5$

$P(i, C), (\text{decision}[i, C])$

$i \setminus C$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0, (0)	10, (1)	10, (1)	10, (1)	10, (1)	10, (1)
2	0, (0)	10, (0)	10, (0)	20, (1)	30, (1)	30, (1)
3						
4						

Beispiel

maximiere $(10, 20, 15, 20) \cdot \mathbf{x}$

so dass $(1, 3, 2, 4) \cdot \mathbf{x} \leq 5$

$P(i, C), (\text{decision}[i, C])$

$i \setminus C$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0, (0)	10, (1)	10, (1)	10, (1)	10, (1)	10, (1)
2	0, (0)	10, (0)	10, (0)	20, (1)	30, (1)	30, (1)
3	0, (0)	10, (0)	15, (1)	25, (1)	30, (0)	35, (1)
4						

Beispiel

maximiere $(10, 20, 15, 20) \cdot \mathbf{x}$

so dass $(1, 3, 2, 4) \cdot \mathbf{x} \leq 5$

$P(i, C), (\text{decision}[i, C])$

$i \setminus C$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0, (0)	10, (1)	10, (1)	10, (1)	10, (1)	10, (1)
2	0, (0)	10, (0)	10, (0)	20, (1)	30, (1)	30, (1)
3	0, (0)	10, (0)	15, (1)	25, (1)	30, (0)	35, (1)
4	0, (0)	10, (0)	15, (0)	25, (0)	30, (0)	35, (0)

Beispiel

maximiere $(10, 20, 15, 20) \cdot \mathbf{x}$

so dass $(1, 3, 2, 4) \cdot \mathbf{x} \leq 5$

$P(i, C), (\text{decision}[i, C])$

$i \setminus C$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0, (0)	10, (1)	10, (1)	10, (1)	10, (1)	10, (1)
2	0, (0)	10, (0)	10, (0)	20, (1)	30, (1)	30, (1)
3	0, (0)	10, (0)	15, (1)	25, (1)	30, (0)	35, (1)
4	0, (0)	10, (0)	15, (0)	25, (0)	30, (0)	35, (0)

Rekonstruktion der Lösung

$C := M$
array $\mathbf{x}[1 \ldots n]$
for $i := n$ **downto** 1 **do**
    $\mathbf{x}[i] :=$ decision$[i, C]$
    if $\mathbf{x}[i] = 1$ **then** $C := C - w_i$
endfor
return $\mathbf{x}$

Analyse:

Zeit: $O(nM)$ pseudopolynomiell

Platz: $M + O(n)$ Maschinenwörter plus Mn bits.

Algorithmenentwurf mittels dynamischer Programmierung

Karlsruher Institut für Technologie

1. Was sind die Teilprobleme? — Kreativität!
2. Wie setzen sich optimale Lösungen aus Teilproblemlösungen zusammen? — Beweisnot
3. Bottom-up Aufbau der Lösungstabelle — einfach
4. Rekonstruktion der Lösung — einfach
5. Verfeinerungen:
   Platz sparen, Cache-effizient, Parallelisierung — Standard-Trickkiste

Anwendungen dynamischer Programmierung

Bellman-Ford Alg. für kürzeste Wege	Teilpfade
Edit distance/approx. string matching	Algorithmen II?
Verkettete Matrixmultiplikation	Übung?
Rucksackproblem	Gegenstände $1..i$ füllen Teil des Rucksacks
Geld wechseln	Übung?

Gegenbeispiel: Teilproblemeigenschaft

Angenommen, die schnellste Strategie für 20 Runden auf dem Hockenheimring verbraucht den Treibstoff vollständig.

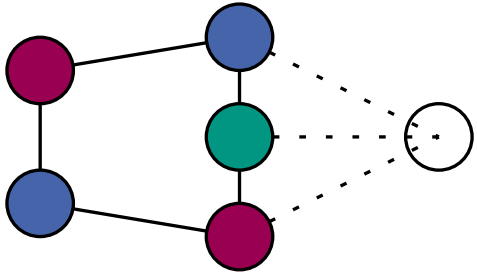
$\Rightarrow$

Keine gute Teilstrategie für 21 Runden.

Frage: Wie kann man “constrained shortest path” trotzdem mittels dynamischer Programmierung modellieren?

Gegenbeispiel: Austauschbarkeit

Optimale Graphfärbungen sind nicht austauschbar.



Generische Optimierungsmethoden

- Black-Box-Löser
- Greedy
- Dynamische Programmierung
- Systematische Suche
- Lokale Suche
- Evolutionäre Algorithmen

Systematische Suche

Idee: Alle (sinnvollen) Möglichkeiten ausprobieren.

Anwendungen:

- Integer Linear Programming (ILP)
- Constraint Satisfaction
- SAT (Aussagenlogik)
- Theorembeweiser (Prädikatenlogik,...)
- konkrete NP-harte Probleme
- Strategiespiele
- Puzzles

Beispiel: Branch-and-Bound für das Rucksackproblem

Function bbKnapsack$((p_1,\ldots,p_n),(w_1,\ldots,w_n),M) : \mathcal{L}$
  assert $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$
  $\hat{\mathbf{x}}$=heuristicKnapsack$((p_1,\ldots,p_n),(w_1,\ldots,w_n),M) : \mathcal{L}$
  $\mathbf{x} : \mathcal{L}$
  recurse$(0, M, 0)$
  return $\hat{\mathbf{x}}$
  // Find solutions assuming $x_1,\ldots,x_{i-1}$ are fixed,
  // $M' = M - \sum_{k<i} x_i w_i$, $P = \sum_{k<i} x_i p_i$.

$\mathbf{x}$: current Solution

$\hat{\mathbf{x}}$: best solution so far

Procedure recurse$(i, M', P : \mathbb{N})$

u:= P + upperBound$((p_i, \ldots, p_n), (w_i, \ldots, w_n), M')$

if $u > \mathbf{p} \cdot \hat{\mathbf{x}}$ **then**

if $i > n$ **then** $\hat{\mathbf{x}} := \mathbf{x}$

else // Branch on variable x_i

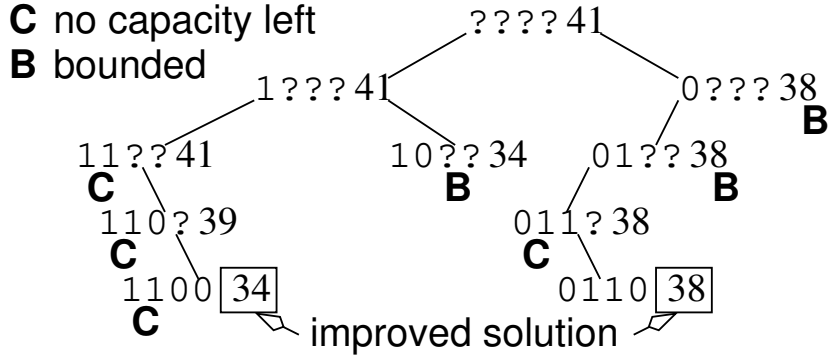
if $w_i \leq M'$ **then** x_i:= 1; recurse$(i + 1, M' - w_i, P + p_i)$

if $u > \mathbf{p} \cdot \hat{\mathbf{x}}$ **then** x_i:= 0; recurse$(i + 1, M', P)$

Schlechtester Fall: 2^n rekursive Aufrufe

Im Mittel: Linearzeit?

Beispielrechnung



Branch-and-Bound – allgemein

Branching (Verzweigen): Systematische Fallunterscheidung, z. B. rekursiv (Alternative, z. B. Prioritätsliste)

Verweigungsauswahl: Wonach soll die Fallunterscheidung stattfinden? (z. B. welche Variable bei ILP)

Reihenfolge der Fallunterscheidung: Zuerst vielversprechende Fälle (lokal oder global)

Bounding: Nicht weitersuchen, wenn optimistische Abschätzung der erreichbaren Lösungen schlechter als beste (woanders) gefundene Lösung.

Duplikatelimination: Einmal suchen reicht

Anwendungsspez. Suchraumbeschränkungen: Schnittebenen (ILP), Lemma-Generierung (Logik),. . .

Generische Optimierungsmethoden

- Black-Box-Löser
- Greedy
- Dynamische Programmierung
- Systematische Suche
- Lokale Suche
- Evolutionäre Algorithmen

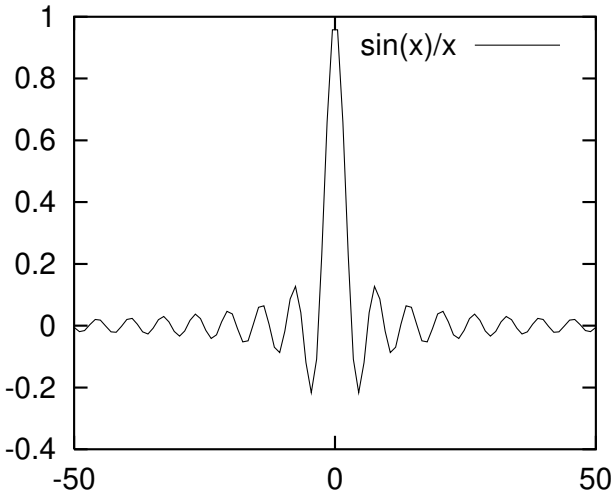
Lokale Suche – global denken, lokal handeln

find some feasible solution $\mathbf{x} \in \mathcal{L}$
$\hat{\mathbf{x}} := \mathbf{x}$ // $\hat{\mathbf{x}}$ is best solution found so far
while not satisfied with $\hat{\mathbf{x}}$ **do**
    $\mathbf{x}$:= some heuristically chosen element from $\mathcal{N}(\mathbf{x}) \cap \mathcal{L}$
    if $f(\mathbf{x}) < f(\hat{\mathbf{x}})$ **then** $\hat{\mathbf{x}} := \mathbf{x}$

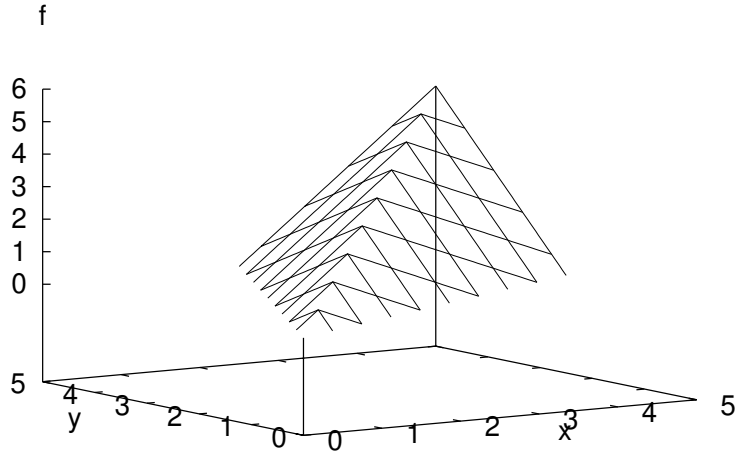
Hill Climbing

Find some feasible solution $\mathbf{x} \in \mathcal{L}$
$\hat{\mathbf{x}} := \mathbf{x}$ // best solution found so far
loop
    if $\exists \mathbf{x} \in \mathcal{N}(\mathbf{x}) \cap \mathcal{L} : f(\mathbf{x}) < f(\hat{\mathbf{x}})$ **then** $\hat{\mathbf{x}} := \mathbf{x}$
    else return $\hat{\mathbf{x}}$ // local optimum found

Problem: Lokale Optima



Warum die Nachbarschaft wichtig ist

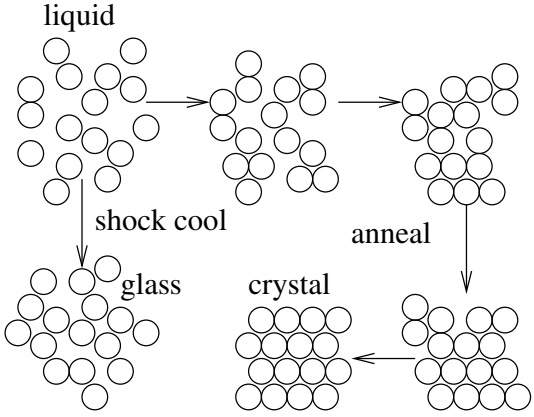


Gegenbeispiel für Koordinatensuche

Jenseits von Hill Climbing

Auch Verschlechterungen akzeptieren.

- Simulated Annealing: physikalische Analogie
- Tabusuche
- ...



Generische Optimierungsmethoden

- Black-Box-Löser
- Greedy
- Dynamische Programmierung
- Systematische Suche
- Lokale Suche
- Evolutionäre Algorithmen

Evolutionäre Algorithmen

Ausführliche Behandlung würde den Rahmen sprengen.

Verallgemeinerung von lokaler Suche:

- $\mathbf{x} \longrightarrow$ Population von Lösungskandidaten
- Reproduktion fitter Lösungen
- Mutation ähnlich lokaler Suche
- zusätzlich: geschlechtliche Vermehrung.
  Idee: erben guter Eigenschaften beider Eltern

Zusammenfassung Vor- und Nachteile (1)

Greedy: Einfach und schnell. Selten optimal. Manchmal Approximationsgarantien.

Systematische Suche: Einfach mit Werkzeugen z. B. (I)LP, SAT, constraint programming. Selbst gute Implementierungen mögen nur mit kleinen Instanzen funktionieren.

Linear Programming: Einfach und einigermaßen schnell. Optimal falls das Modell passt. Rundungsheuristiken ergeben Näherungslösungen

Dynamische Programmierung: Optimale Lösungen falls Teilprobleme optimal und austauschbar sind. Hoher Platzverbrauch.

Integer Linear Programming: Leistungsfähiges Werkzeug für optimale Lösungen. Gute Formulierungen können viel know how erfordern.

Zusammenfassung Vor- und Nachteile (2)

Lokale Suche: Flexibel und einfach. Langsam aber oft gute Lösungen für harte Probleme.

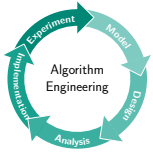
Hill climbing: einfach aber leidet an lokalen Optima.

Simulated Annealing und Tabu Search: Leistungsfähig aber langsam. Tuning kann unschön werden.

Evolutionäre Algorithmen: Ähnliche Vor- und Nachteile wie lokale Suche. Durch geschl. Vermehrung potentiell mächtiger aber auch langsamer und schwieriger gut hinzukriegen. Weniger zielgerichtet.

Index

Karlsruher Institut für Technologie



Algorithmen I – 13. Zusammenfassung

Sommersemester 2025

Peter Sanders | Stand: 30. Juli 2025

Zusammenfassung

- Datenstrukturen
- Algorithmen
- Entwurfstechniken
- Analysetechniken

Zusammenfassung
Datenstrukturen

- (doppelt) verkettete Listen, unbeschränkte (zyklische) Felder, Stapel, FIFOs, deques
- (beschränktes) Hashing: verketten (universell) / lin. Suche
- sortiertes Feld
- Prioritätslisten (binärer Heap) (addressierbar)
- Implizite Repräsentation vollständiger Bäume
- Suchbäume: binär, (a, b)-Baum
- Graphen: Adjazenzfeld / Listen / Matrix
- Union-Find

Zusammenfassung
Algorithmen

- Langzahlmultiplikation
- Insertion-, Merge-, Quick-, Heap-, Bucket-, Radix-sort, Selektion
- BFS, DFS, topologisches Sortieren
- Kürzeste Wege: Dijkstra, Bellman-Ford
- MST: Jarník-Prim, Kruskal
- Rucksack: greedy, (I)LP, dynamische Programmierung (über Profit) , backtracking systematische Suche

Zusammenfassung
Entwurfstechniken

- Iteration/Induktion/Schleifen, Teile-und-Herrsche
- Schleifen- und Datenstruktur-Invarianten
- Randomisierung (universelles Hashing, Quicksort,...)
- Graphenmodelle
- Trennung Mathe $\leftrightarrow$ Funktionalität $\leftrightarrow$ Repräsentation $\leftrightarrow$ Algorithmus $\leftrightarrow$ Implementierung
- Sonderfälle vermeiden
- Zeigerdatenstrukturen
- Datenstrukturen augmentieren (z.B. Teilbaumgrößen)
- Datenstrukturen unbeschränkt machen
- Implizite Datenstrukturen (z.B. Intervallgraphen)

Zusammenfassung
Entwurfstechniken

- Algebra (Karatsuba, univ. Hashfkt., Matrixmultiplikation für Graphen)
- Algorithmenschemata (z.B. DFS, lokale Suche)
- Verwendung abstrakter Problemeigenschaften (z.B. Schnitt/Kreis-Eigenschaft bei MST)
- Black-Box-Löser (z.B. lineare Programmierung)
- Greedy
- Dynamische Programmierung
- Systematische Suche
- Metaheuristiken (z.B. Lokale Suche)

Zusammenfassung
Analysetechniken

- Summen, Rekurrenzen, Induktion, Master-Theorem, Abschätzung
- Asymptotik ($O(\cdot)$, ..., $\omega(\cdot)$), einfache Modelle
- Analyse im Mittel
- Amortisierung (z.B. unbeschränkte Felder)
- Linearität des Erwartungswertes, Indikatorzufallsvariablen
- Kombinatorik ($\approx$ Zählen): univ. Hashfunktionen,
  untere Sortierschranken (Informationsmenge)
- Integrale als Summenabschätzung
- Schleifen/Datenstruktur-(In)varianten
  (z.B. (a, b)-Baum, Union-by-rank)

Zusammenfassung
Weitere Techniken

- Algorithm Engineering
- Parameter Tuning (z.B. Basisfallgröße)
- High-Level Pseudocode
- Dummys und Sentinels (Listen, insertion sort,. . .)
- Speicherverwaltung