Algorithmen & Komplexität

Angelika Steger
Institut für Theoretische Informatik

steger@inf.ethz.ch

Idee des Algorithmus von Dijkstra

Grundlegende Idee

Besuche in jedem Schritt den Knoten, der den kürzesten Abstand von den schon besuchten Knoten hat.

Die Abstände zu s werden in jedem Knoten gespeichert.

Datenstrukturen

W	(Menge der Knoten v für die kürzester s-v-Pfad bekannt)
ρ[v]	(aktuell bekannte Länge eines kürzesten Pfades)
pred[v]	(Pointer zu einem Vorgänger)

Algorithmus von Dijkstra

Algorithmus 2.5 Algorithmus von Dijkstra

Eingabe: Ein zusammenhängendes Netzwerk $N = (V, E, \ell)$ mit $\ell \geq 0$ und Knoten $s, t \in V$
Ausgabe: Ein kürzester s–t-Pfad in N

{ *Initialisierung* }
$W := \emptyset$; $\rho[s] := 0$; pred$[s] := nil$;
for all $v \in V \setminus \{s\}$ **do** $\rho[v] := \infty$;

{ *Traversieren und Markieren* }
while $t \notin W$ **do**
    { *Traversieren* }
    Finde ein $x_0 \in V \setminus W$ mit $\rho[x_0] = \min\{\rho[v] \mid v \in V \setminus W\}$;
    $W := W \cup \{x_0\}$;
    { *Markieren* }
    for all $v \in \Gamma(x_0) \cap (V \setminus W)$ mit $\rho[v] > \rho[x_0] + \ell(x_0, v)$ **do**
        $\rho[v] := \rho[x_0] + \ell(x_0, v)$; pred$[v] := x_0$;

Algorithmus von Dijkstra

Satz

Für nicht-negative Gewichtsfunktionen berechnet der Algorithmus von Dijkstra zu einem Startknoten s für alle Knoten $v \in V$ einen kürzesten Pfad von s zu v. Die Laufzeit des Algorithmus ist $O(n^2)$.

Ausblick

Verwendung von Fibonacci Heaps Laufzeit $O(n \log n + m)$ möglich.

Single-Source vs. All-Pairs Shortest Path

Gegeben

Netzwerk N = (V, E, w)

w nicht-negativ

Single-Source Shortest Path Problem

Berechne alle kürzesten Pfade ab einem Startknoten s
Algorithmus von Dijkstra

Laufzeit: $O(n^2)$ [bzw. $O(n \log n + m)$ … später!]

All-Pairs Shortest Path Problem

Bereche die kürzesten Pfade zwischen allen Knotenpaaren

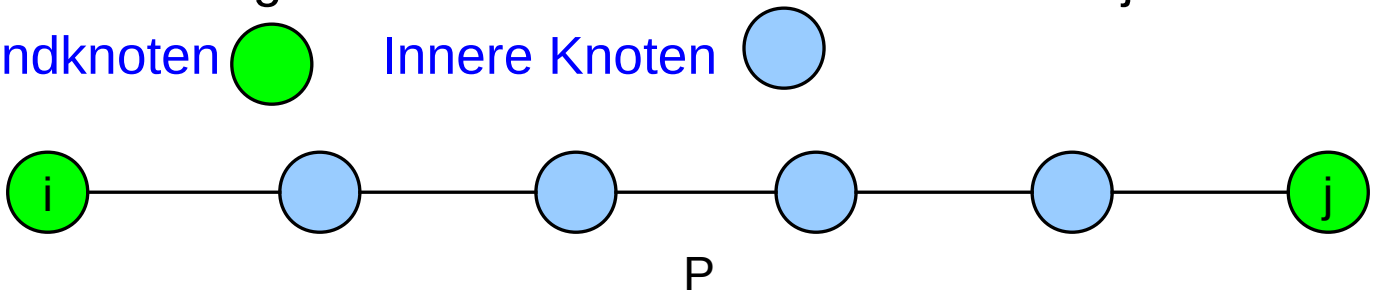
Algorithmus von Floyd-Warshall
Laufzeit: $O(n^3)$

Idee des Algorithmus von Floyd-Warshall

Knotenmenge V = {1, 2, …, n}

P sei beliebiger Pfad zwischen den Knoten i und j

Endknoten Innere Knoten



Invariante

Berechne eine Folge F_k von Matrizen für k = 0,1, 2, …, n

$F_k[i, j]$ speichert die Länge eines kürzesten Pfades zwischen i und j der nur innere Knoten aus der Menge {1, 2, …, k} hat.

Algorithmus von Floyd-Warshall

Algorithmus 2.6 Algorithmus von Floyd-Warshall

Eingabe: Ein zusammenhängendes Netzwerk $N = (V, E, \ell)$ mit $\ell \geq 0$ und $V = \{1, \ldots, n\}$

Ausgabe: Die Längen aller kürzesten i–j-Pfade in N, für alle Paare $i, j \in V$

{ *Initialisierung* }

$$\forall\ i,\ j\ \in\ V: \qquad F_0[i,j] := \begin{cases} l(\{i,j\}) & \text{falls } \{i,j\} \in E \\ 0 & \text{falls } i = j \\ \infty & \text{sonst} \end{cases} .$$

{ *Rekursion* }

for $k = 1$ **to** n **do**

$\forall\ i,\ j\ \in\ V: F_k[i,j] := \min\{F_{k-1}[i,j], F_{k-1}[i,k] + F_{k-1}[k,j]\}$;

{ *Ausgabe* }

$F_n[i,j]$ gibt die Länge eines kürzesten i–j-Pfades an.

Graphen mit negativen Kantengewichten

Was passiert wenn Kantengewichte auch negativ sind?

D.h.: Gewichtsfunktion. $w : E \rightarrow \mathbf{Z}$ (statt $w : E \rightarrow \mathbf{N}$)

Wenn es eine negative Kante gibt, so ist die Länge eines kürzesten Weges $-\infty$.

[Begr.: Durchlaufe negative Kante „unendlich oft“ hin und zurück …]

Das Problem einen kürzesten Pfad zu finden ist „schwierig“.

[vgl. Kapitel 5 der Vorlesung]

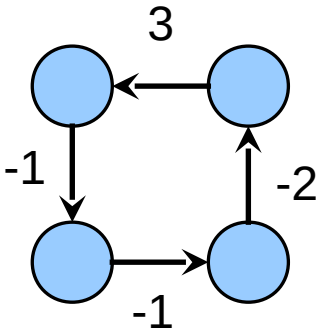
Gerichtete Graphen

Gegeben

Gerichteter Graph: D = (V, A)

Kanten haben Richtungen: 

Allgemeine Gewichtsfunktion. w : A → **Z**



Problem

Wenn der Graph einen negativen gerichteten Kreis enthält, kann dieser beliebig oft durchlaufen werden.

Ziel

Entscheide, ob es einen negativen gerichteten Kreis gibt;

falls nicht: bestimme alle kürzesten gerichteten Pfade.

Beachte: Wenn es keinen negativen gerichteten Kreis gibt, so enthält ein kürzester gerichteter Weg immer einen kürzesten gerichteter Pfad!

Kürzester Pfad vs. kürzester Weg

Betrachte Algorithmus von Floyd-Warshall:

Funktioniert auch für gerichtete Graphen (nach leichter Abänderung der Initialisierung):

- In $F_k[i, j]$ ist nur die Länge eines kürzesten Pfades gespeichert, nicht aber welche Knoten dieser Pfad enthält.
- Bei der Rekursion

  $$F_k[i, j] = \min \{ F_{k-1}[i,j],\ F_{k-1}[i,k] + F_{k-1}[k,j] \}$$

  kann daher nicht a priori sicher gestellt werden, dass Knoten nicht doppelt vorkommen.

ABER: kein gerichteter Kreis => können Weg zu Pfad abkürzen

Modifizierter Floyd-Warshall

Algorithmus 2.7 Modifizierter Algorithmus von Floyd-Warshall

Eingabe: Gerichtetes Netzwerk $N = (V, A, \ell)$ mit $V = \{1, \ldots, n\}$.

Ausgabe: Die Längen aller kürzesten i-j Pfade in N, für alle Paare $i, j \in V$, oder ein Beweis dafür, dass das Netzwerk einen negativen gerichteten Kreis enthält.

{ *Initialisierung* }

$$\forall\ i,\ j\ \in\ V: \qquad F_0[i,j] := \begin{cases} l((i,j)) & \text{falls } (i,j) \in A \\ 0 & \text{falls } i = j \\ \infty & \text{sonst} \end{cases} .$$

{ *Rekursion* }

for $k = 1$ **to** n **do**

$\quad \forall\ i,\ j\ \in\ V : F_k[i,j] := \min\{F_{k-1}[i,j], F_{k-1}[i,k] + F_{k-1}[k,j]\};$

{ *Ausgabe* }

if $(\exists i \in V$ mit $F_n[i,i] < 0)$ **then**

$\quad$ **return** "Graph enthält einen negativen Kreis"

else

$\quad F_n[i,j]$ gibt die Länge eines kürzesten i–j-Pfades an.

Korrektheit

Satz:

Für gerichtete Netzwerke N = (V, A, w) mit beliebiger Gewichtsfunktion w gilt: der modifizierte Floyd-Warshall Algorithmus berechnet in Zeit $O(n^3)$ Werte $F_n[i, j]$ mit:

- Enthält N einen negativen gerichteten Kreis, so gilt $F_n[i, i] < 0$ für mindestens ein i.
- Ansonsten bezeichnet $F_n[i, j]$ die Länge eines kürzesten i-j-Pfades.

Beweisidee:

(1) kein negativer Kreis: Zeige durch Induktion über k:

$F_k[i,j]$ = Länge eines kürzesten Pfades von i nach j, dessen innere Knoten alle in der Menge {1,…,k} enthalten sind

(2) es gibt negativer Kreis: Betrachte negativen Kreis mit grösstem Knoten u …

Kapitel 2.5:

Minimale Spannbäume

Minimaler Spannbaum

Problemstellung

Gegeben

Graph $G = (V, E)$, Gewichtsfunktion $w : E \rightarrow \mathbf{R}$

Spannbaum

$T = (V, E_T)$ Baum mit $E_T \subseteq E$

Gewicht eines Spannbaums

Summe der Kantengewichte des Baumes: $w(T) := \Sigma_{e \in E_T} w(e)$

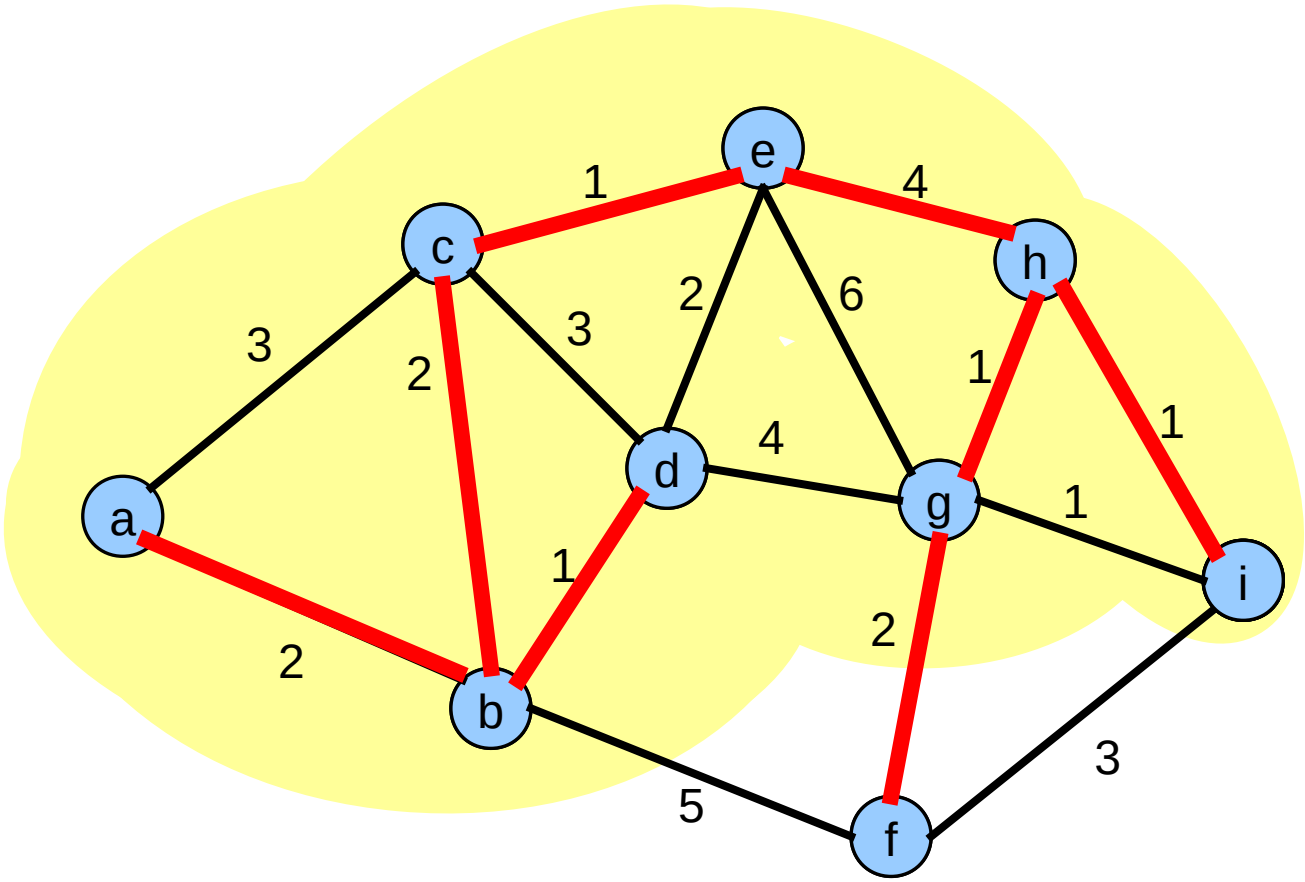
Minimaler Spannbaum

$T = (V, E_T)$ mit $E_T \subseteq E$ mit $w(T) = \min \{ w(T') : T' \text{ Spannbaum} \}$

Gesucht

Minimaler Spannbaum

Beispiel: Algorithmus von Prim



W

Algorithmus von ~~Dijkstra~~ Prim

Algorithmus 2.5 Algorithmus von ~~Dijkstra~~ Prim

Eingabe: Ein zusammenhängendes Netzwerk $N = (V, E, \ell)$ mit ~~$\ell \geq 0$ und Knoten $s, t \in V$~~

Ausgabe: Ein ~~kürzester s-t-Pfad in N~~ minimaler Spannbaum

{ *Initialisierung* } Wähle s ∈V beliebig

$W := \emptyset$; $\rho[s] := 0$; pred$[s] := nil$;

for all $v \in V \setminus \{s\}$ **do** $\rho[v] := \infty$; **F := ⊘;**

{ *Traversieren und Markieren* }

while ~~$t \notin W$~~ **do** W≠V

{ *Traversieren* }

Finde ein $x_0 \in V \setminus W$ mit $\rho[x_0] = \min\{\rho[v] \mid v \in V \setminus W\}$;

$W := W \cup \{x_0\}$; **F := F + {x_0, pred[x_0]};**

{ *Markieren* }

for all $v \in \Gamma(x_0) \cap (V \setminus W)$ mit $\rho[v] >$ ~~$\rho[x_0] +$~~ $\ell(x_0, v)$ **do**

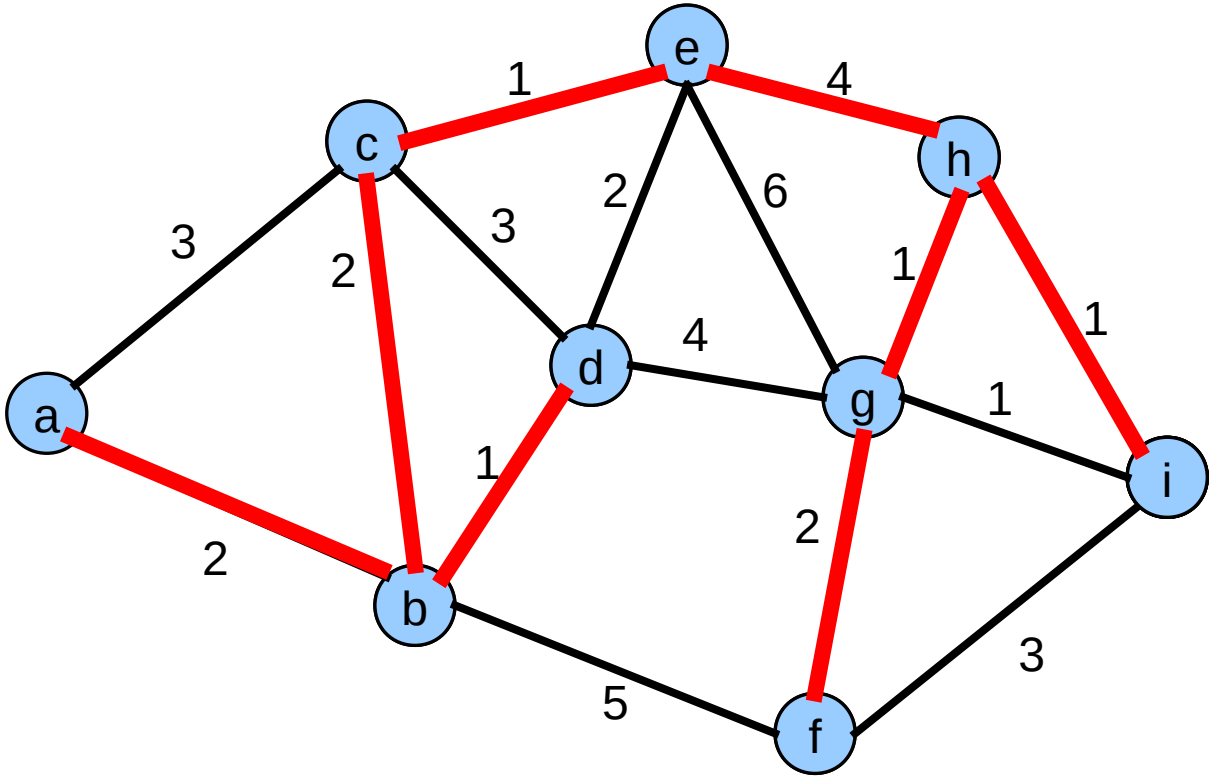
$\rho[v] :=$ ~~$\rho[x_0] +$~~ $\ell(x_0, v)$; pred$[v] := x_0$;

Beweisidee:

Zeige, dass vor und nach jeder Iteration der while-Schleife gilt:

- es gibt einen minimalen Spannbaum **T** mit **F ⊆ T**
- für alle v ε V\W gilt: ρ[v] = min{ ℓ(v,w) : w ε W} und ρ[v] = ℓ(v,pred[v])

Beispiel: Algorithmus von Kruskal



Algorithmus von Kruskal

Algorithmus 2.9 Algorithmus von Kruskal

Eingabe: Ein zusammenhängendes Netzwerk $N = (V, E, \ell)$
Ausgabe: Ein minimaler Spannbaum $T = (V, F)$ in N

{ *Initialisierung* }
Sortiere die Kanten, so dass $\ell(e_1) \leq \ldots \leq \ell(e_m)$.
Setze $F := \emptyset$.

{ *Aufbau des Baums* }
for $i := 1$ **to** m **do**
    if $(V, F \cup \{e_i\})$ ist kreisfrei **then** $F := F \cup \{e_i\}$.

Algorithmus von Kruskal

Satz

Der Algorithmus von Kruskal berechnet für alle Gewichtsfunktionen einen minimalen Spannbaum und benötigt höchstens O(nm) Schritte.

Korrektheit: später …

Laufzeit:

Initialisierung:

Sortieren vom m=|E| Zahlen: O(m log(m)) = O(m log(n))

Schleife:

- wird maximal m oft durchlaufen
- pro Durchlauf ein Test auf Kreisfreiheit … O(n)

Verbesserte Implementierung: O(m log n) … später!

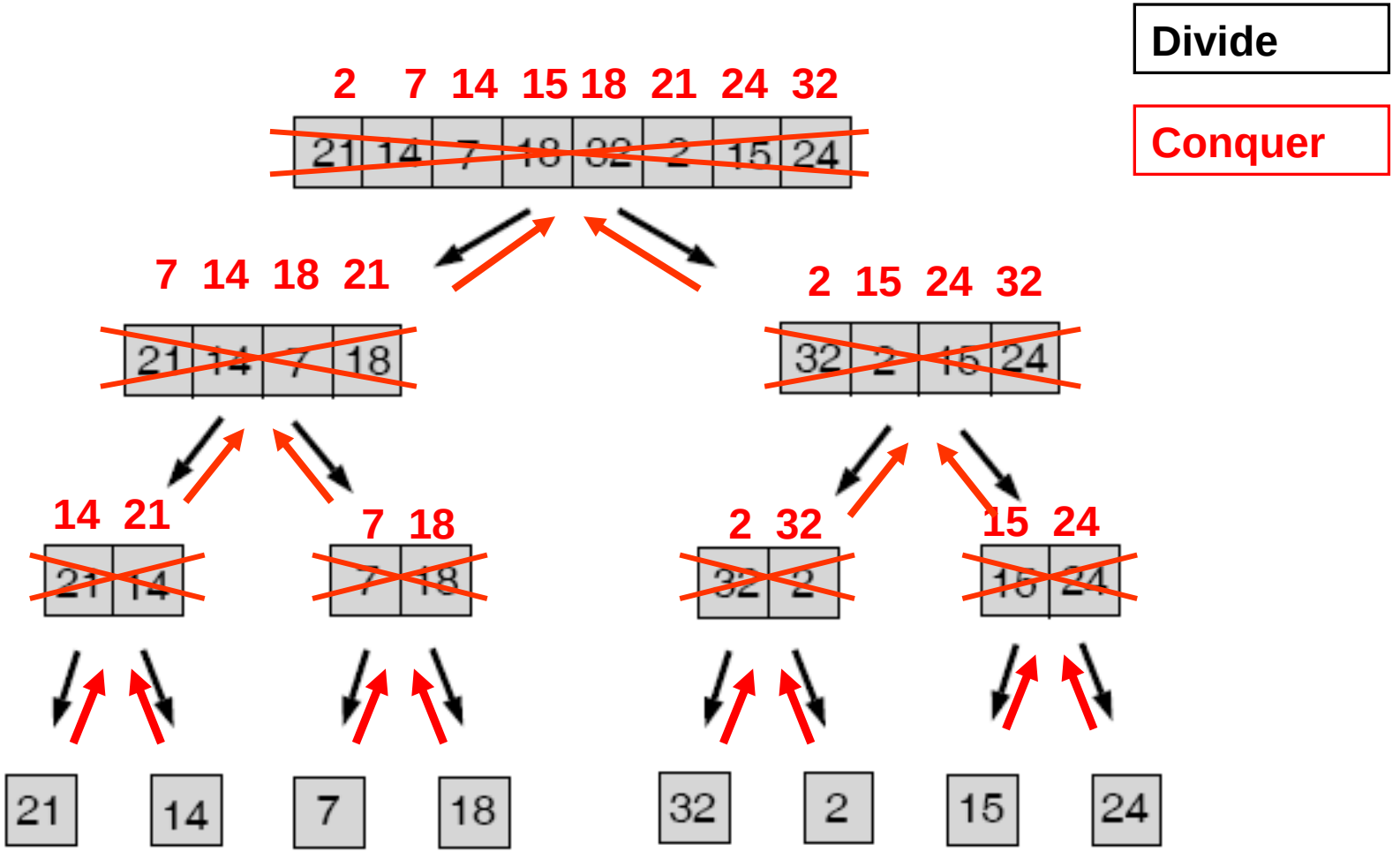
Kapitel 3:

Algorithmische Grundprinzipien

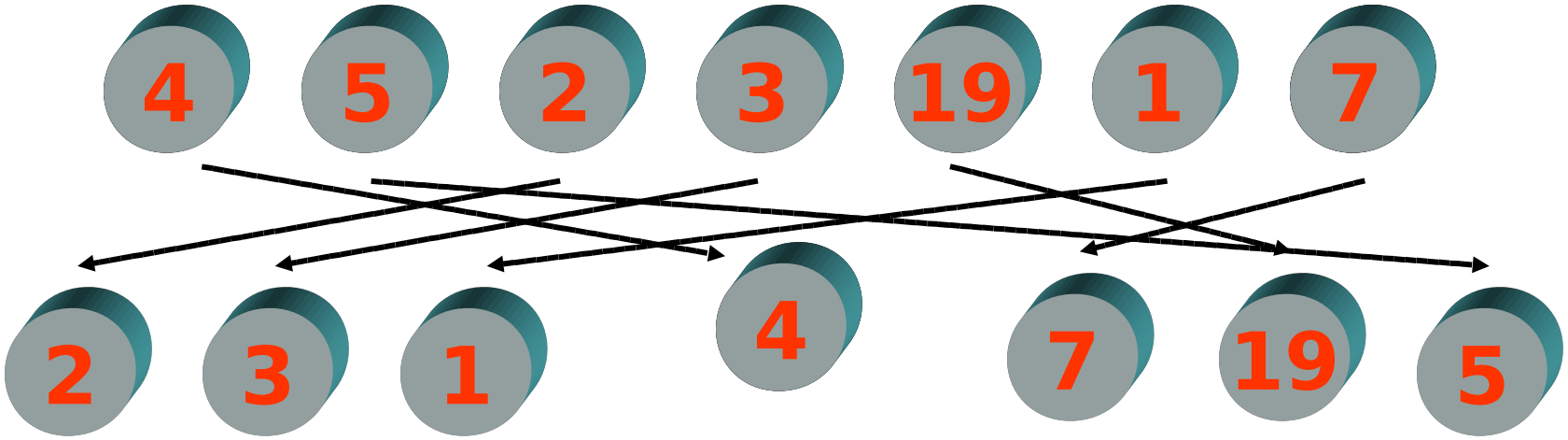
Kapitel 3.1:

Divide & Conquer Algorithmen

MergeSort - Beispiel



QuickSort - Beispiel



Zahlen kleiner als 4

Zahlen grösser als 4

Divide

Wähle erstes Element und teile übrige Elemente in „kleiner“ und „grösser“ auf

Conquer

Algorithmus von Strassen

A und B sind $2^k x 2^k$ Matrizen:

$$C = A \cdot B$$

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Divide

Zurückführung auf 7 Multiplikationen von $2^{k-1} x 2^{k-1}$ Matrizen
Rekursive Anwendung.

Conquer

Berechne C (17 Additionen von $2^{k-1} x 2^{k-1}$ Matrizen)

Kap. 3: Algorithmische Grundprinzipien

3.1 Divide & Conquer Verfahren

MergeSort	$C_n = C_{\lceil n/2 \rceil} + C_{\lfloor n/2 \rfloor} + n-1$
QuickSort	
Binäre Suche	$B_n = B_{\lceil n/2 \rceil} + 1$
Alg. von Strassen	$T_n = 7\, T_{\lceil n/2 \rceil} + 15\, n^2$

Das Master-Theorem

erlaubt Gaussklammern

Satz 3.1 *(Master-Theorem) Seien $\alpha \geq 1$, $\beta > 1$ und $C \geq 0$ Konstanten und sei $f(n)$ eine positive Funktion. Weiter seien $c_1(n), \ldots, c_\alpha(n)$ Funktionen mit $|c_i(n)| \leq C$ für alle $1 \leq i \leq \alpha$ und $n \in \mathbb{N}$. Ist dann $T(n)$ eine Funktion mit $T(1) = 0$, die für $n \geq 1$ die Rekursionsgleichung*

$$T(n) = T(n/\beta + c_1(n)) + \cdots + T(n/\beta + c_\alpha(n)) + f(n)$$

erfüllt, dann gilt

$$T(n) = \begin{cases} \Theta(n^{\log_\beta \alpha}), & \text{falls } f(n) = O(n^{\log_\beta \alpha - \epsilon}) \text{ für ein } \epsilon > 0, \\ \Theta(f(n) \log n), & \text{falls } f(n) = \Theta(n^{\log_\beta \alpha} (\log n)^\delta) \text{ für ein } \delta \geq 0, \\ \Theta(f(n)), & \text{falls } f(n) = \Omega(n^{\log_\beta \alpha + \epsilon}) \text{ für ein } \epsilon > 0. \end{cases}$$

Mergesort: $T_n = T_{\lceil n/2 \rceil} + T_{\lfloor n/2 \rfloor} + n - 1$

also $\alpha = 2$, $\beta = 2$, $f(n) = n - 1$

Kapitel 3:

Algorithmische Grundprinzipien

Kapitel 3.2:

Dynamische Programmierung