

# Algorithmen II

**Peter Sanders, Timo Bingmann**

**Übungen:**

**Sebastian Lamm, Demian Hespe**

Institut für Theoretische Informatik

Web:

[http://algo2.iti.kit.edu/AlgorithmenII\\_WS18.php](http://algo2.iti.kit.edu/AlgorithmenII_WS18.php)

# Organisatorisches

## Vorlesungen:

Mo 09:45–11:15 HS Neue Chemie

Di 15:45–16:30 HS Neue Chemie

## Saalübung:

Di 16:30–17:15 HS Neue Chemie

## Übungsblätter:

14-tägig, jeweils Dienstags, Musterlösung 9 Tage später

1. Blatt: 23.10.2018

# Organisatorisches

**Sprechstunden:**

- Peter Sanders, Dienstag 13:45–14:45 Uhr, Raum 217
- Sebastian Lamm, nach Absprache, Raum 210
- Demian Hespe, nach Absprache, Raum 210

**Ilias Forum:** Link auf

[http://algo2.iti.kit.edu/AlgorithmenII\\_WS18.php](http://algo2.iti.kit.edu/AlgorithmenII_WS18.php)

**Letzte Vorlesung:** 05. Februar 2019

**Klausur:** Dienstag, 19. Februar 2019, 13:00 Uhr

# Materialien

- Folien
- Übungsblätter
- Buch:  
K. Mehlhorn, P. Sanders  
Algorithms and Data Structures — The Basic Toolbox  
Springer 2008. Ca. 40 % der Vorlesung.
- Skript: Minimalistischer Aufschrieb der Sachen, die nicht im Buch stehen, mit Verweisen auf Originalliteratur  
Sprache: **Deutsch**.  
Aber Literatur und ein Teil der Folien auf **Englisch**

## Zum Weiterlesen

- [Dementiev, Dietzfelbinger, Mehlhorn, Sanders] Sequential and **Parallel** Algorithm and Data Structures — The Basic Toolbox
- [Mehlhorn, Näher] **Algorithm Engineering, Flows, Geometrie** The LEDA Platform of Combinatorial and Geometric Computing.
- [Ahuja, Magnanti, Orlin] Network Flows
- [de Berg, Cheong, van Kreveld, Overmars] **Geometrie** Computational Geometry: Algorithms and Applications
- [Gonzalo Navarro] **Succinct Data Structures** Compact Data Structures: A Practical Approach
- [R. Niedermeier] Invitation to Fixed-Parameter Algorithms

# Inhaltsübersicht I

- Algorithm Engineering
- Fortgeschrittene Datenstrukturen am Beispiel von Prioritätslisten
  - addressierbar
  - ganzzahlige Schlüssel
  - (Externspeicher)
- Fortgeschrittene Graphenalgorithmen
  - Kürzeste Wege II: negative Kreise, Potentialmethode
  - Starke Zusammenhangskomponenten
  - Maximale Flüsse und Matchings

## Inhaltsübersicht II

- “Bindestrichalgorithmen”
  - Randomisierte Algorithmen
  - Externe Algorithmen
  - Parallel Algorithmen
  - Stringalgorithmen: sorting, indexing,...
  - Geometrische Algorithmen
  - Approximationsalgorithmen
  - Fixed-Parameter-Algorithmen
  - Onlinealgorithmen

# Zusammenfassung – Rolle der Algorithmik

- Kerndisziplin der Informatik
- Das Herz jeder nichttrivialen Computeranwendung
- Algorithm Engineering führt zu wohlverstandenen Lösungen für eine Vielzahl von Problemen

# “Machine Learning macht das von selbst?”

- Machine Learning Algorithmen sind auch Algorithmen
- Spezialisiertere Algorithmen finden bei wohlverstandenen Problemen bessere, schnellere und **robustere** Lösungen
- Teile Problem in wohlverstanden Teil und den Rest
- Machine Learning für Parameter-Tuning? Oder für Kantengewichte bei Routenplanung?
- Spezialisierte Algorithmen für wohlverstandene Teile, z.B. Feature-Extraktion oder Indexdatenstrukturen?

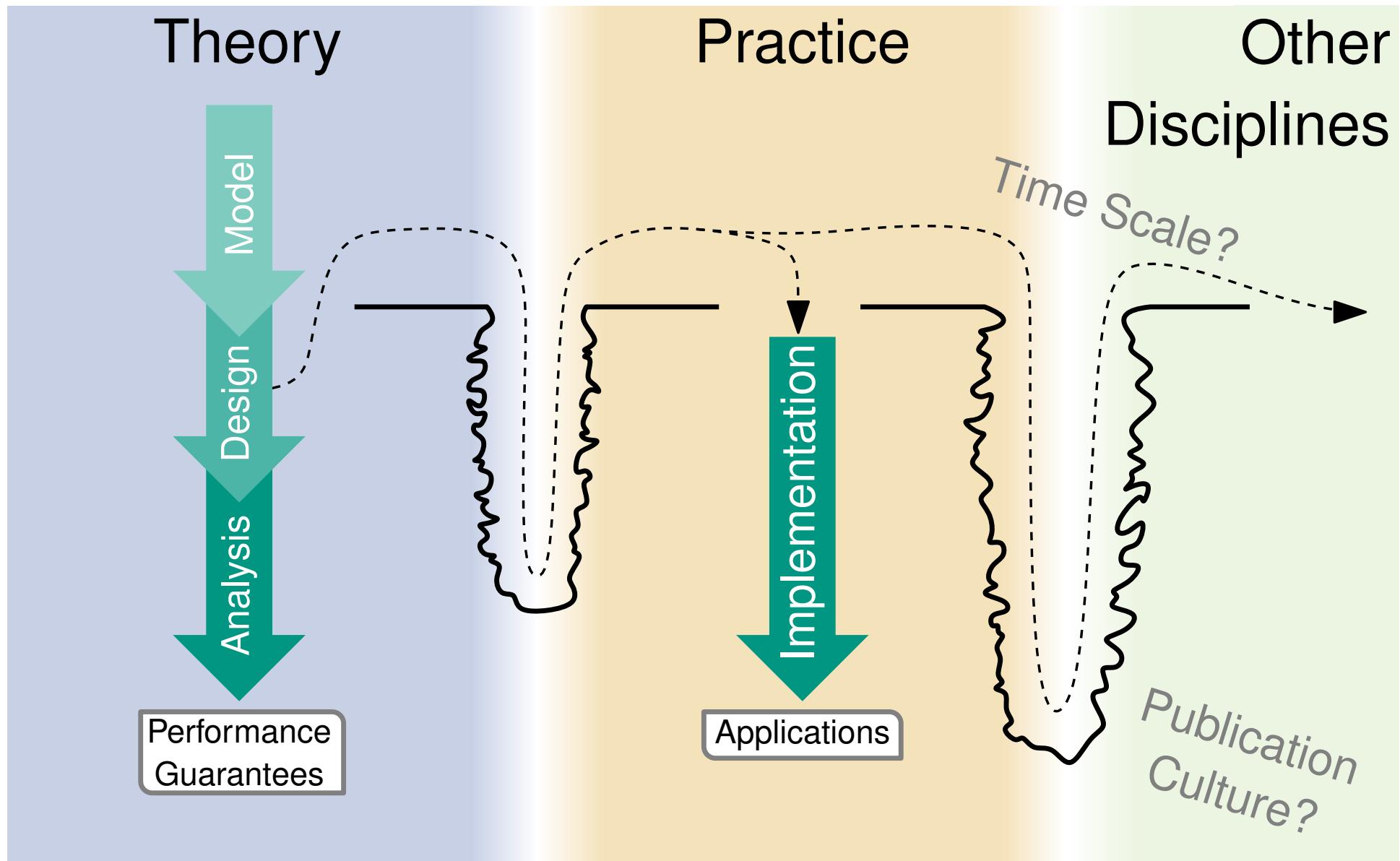


# 1 Algorithm Engineering

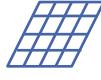
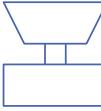
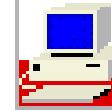
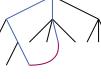
## A detailed definition

- in general
  - [with Kurt Mehlhorn, Rolf Möhring, Petra Mutzel, Dorothea Wagner]
- A few examples, usually sorting
- A little bit on experimental methodology

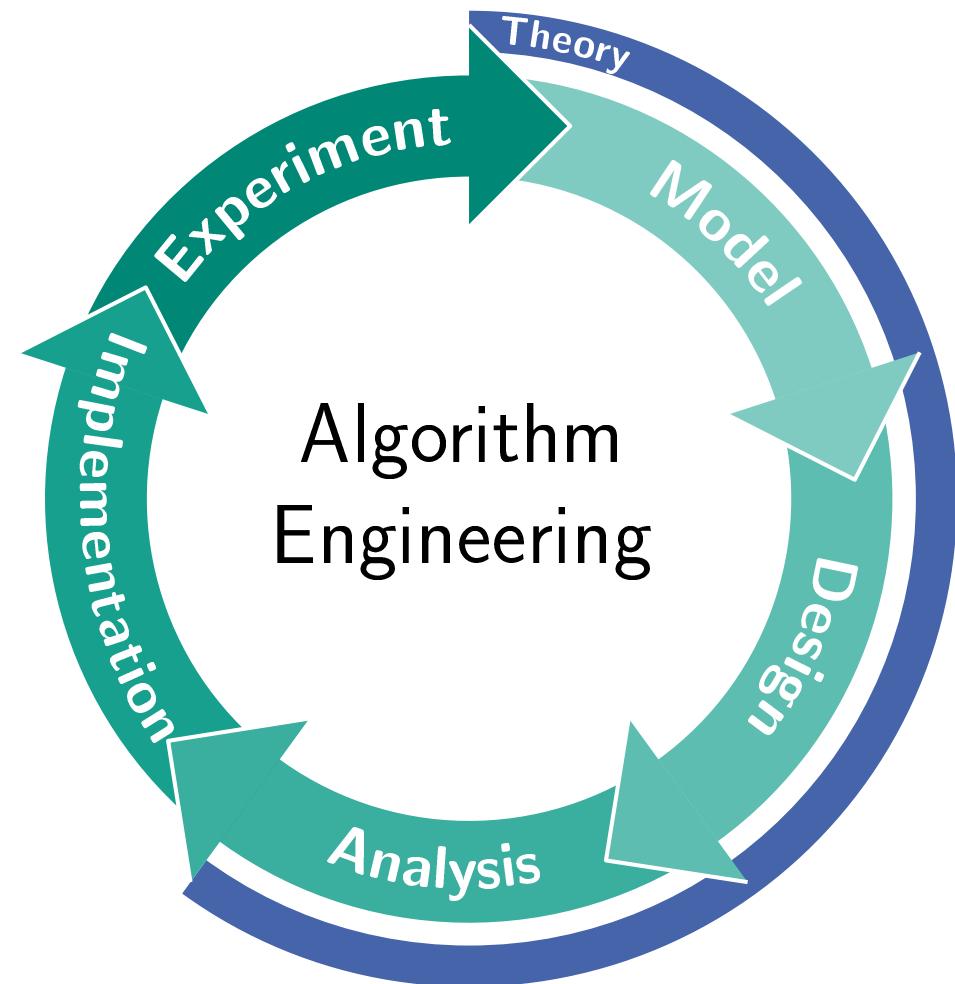
# (Caricatured) Traditional View: Algorithm Theory



# Gaps Between Theory & Practice

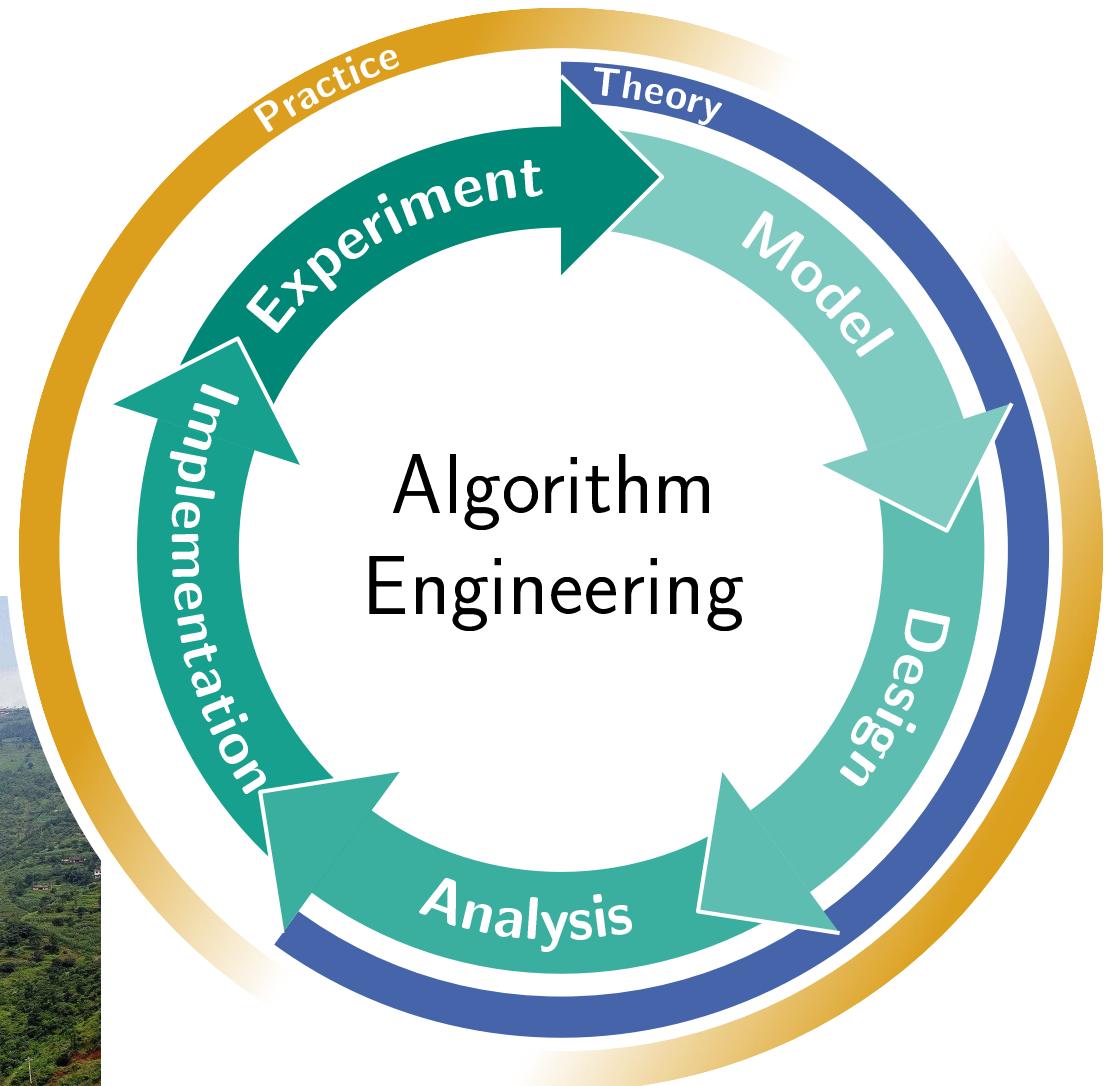
Theory	$\longleftrightarrow$	Practice
simple 	appl. model	 complex
simple 	machine model	 real
complex 	algorithms	 simple
advanced 	data structures	 arrays,...
worst case 	complexity measure	 inputs
asympt. 	efficiency	 42% constant factors

# Algorithmics as Algorithm Engineering



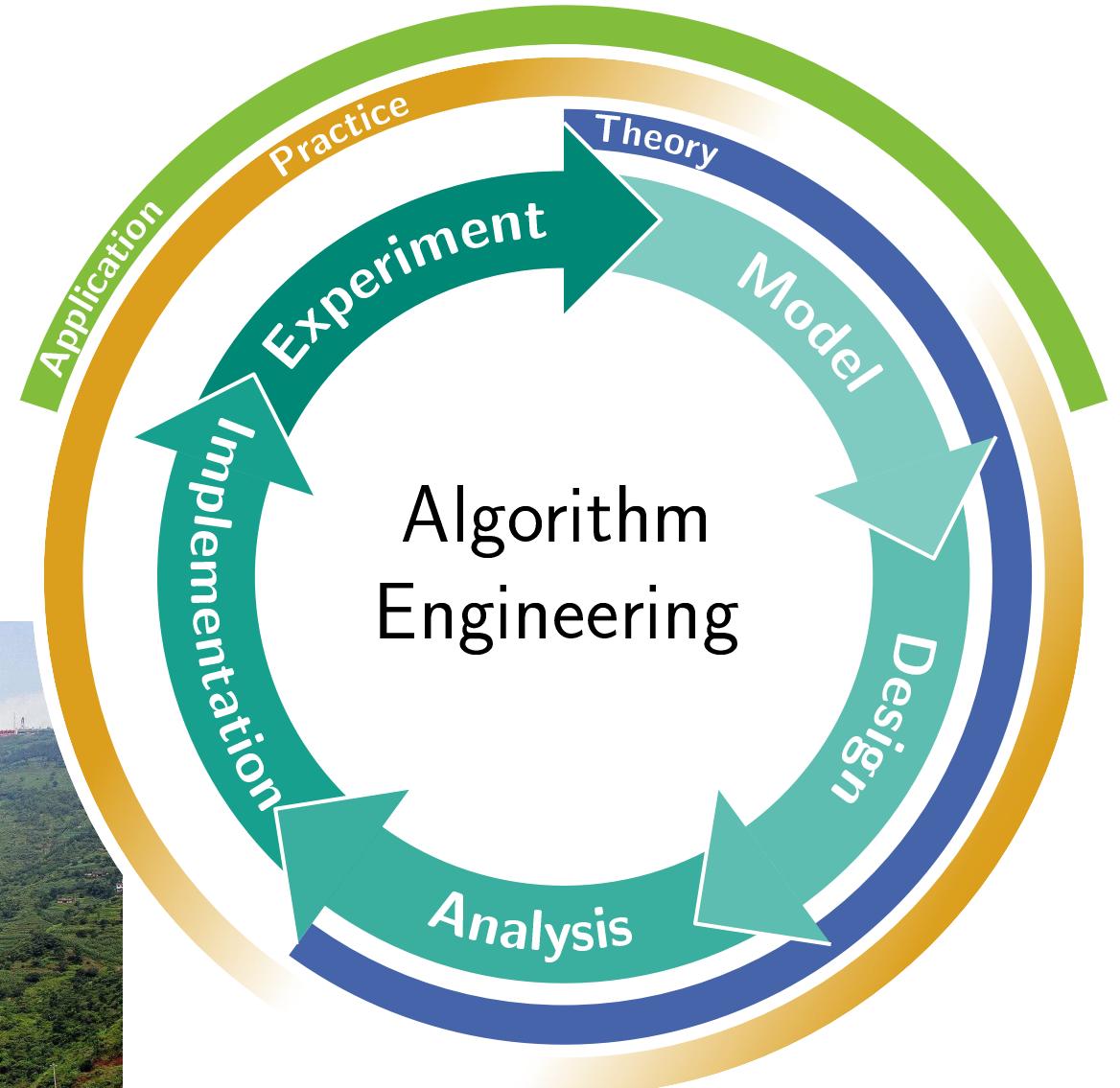
# Algorithmics as Algorithm Engineering

- bridge gaps between theory and practice

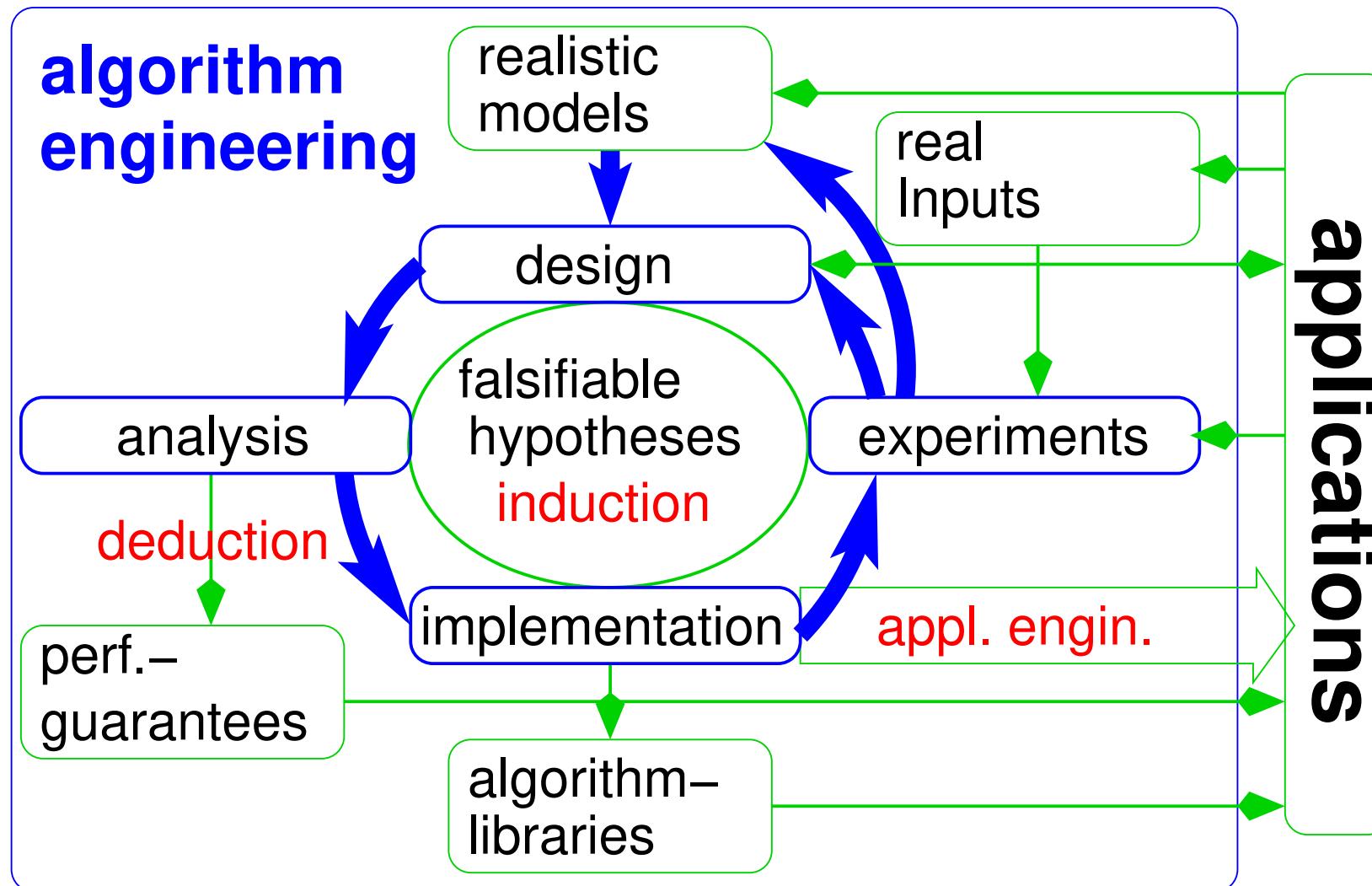


# Algorithmics as Algorithm Engineering

- bridge gaps between theory and practice
- integrated interdisciplinary research



# Algorithmics as Algorithm Engineering





# Bits of History

1843– Algorithms in theory and practice

1950s, 1960s Still infancy

1970s, 1980s Paper and pencil algorithm theory.

Exceptions exist, e.g., [D. Johnson], [J. Bentley]

1986 Term used by [T. Beth],  
lecture “AlgorithmenTechnik” in Karlsruhe.

1988– Library of Efficient Data Types and Algorithms  
(LEDA) [K. Mehlhorn]

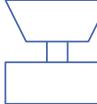
1997– Workshop on Algorithm Engineering  
~~> ESA applied track [G. Italiano]

1997 Term used in US policy paper [Aho, Johnson, Karp, et. al]

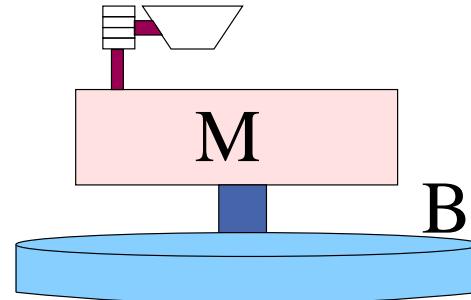
1998 Alex workshop in Italy ~~> ALENEX



# Realistic Models

Theory	↔	Practice
simple 	appl. model	 complex
simple 	machine model	 real

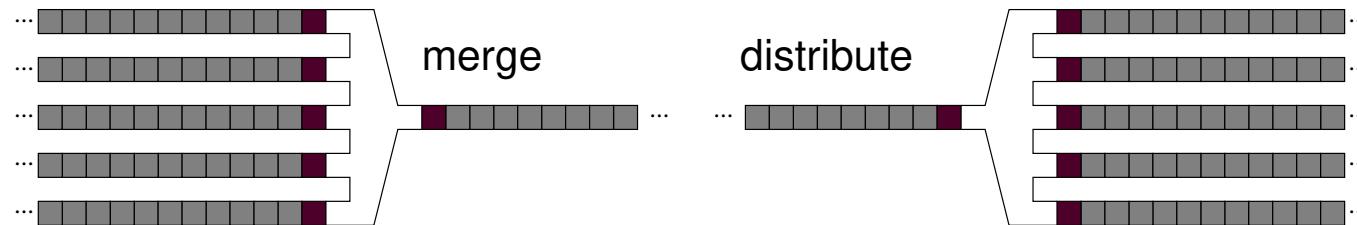
- Careful refinements
- Try to preserve (partial) analyzability / simple results



# Design

of algorithms that work well in **practice**

- simplicity**
- reuse**
- constant factors**
- exploit easy instances**



# Analysis

- Constant factors matter  
Beispiel: quicksort
- Beyond worst case analysis
- Practical algorithms might be difficult to analyze  
(randomization, meta heuristics, ...)

# Implementation

sanity check for algorithms !

## Challenges

Semantic gaps:

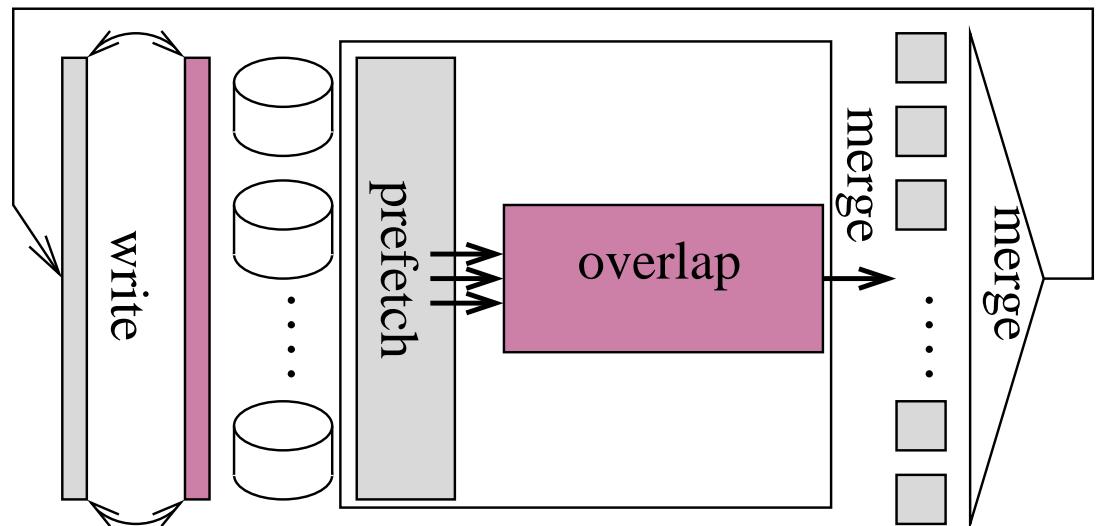
Abstract algorithm



C++...



hardware



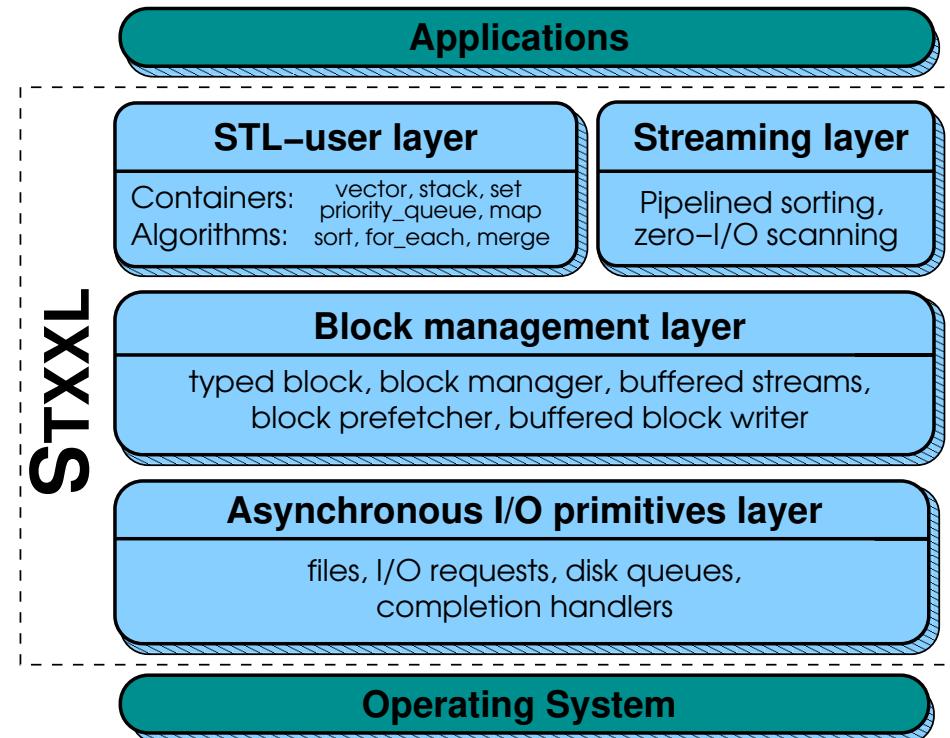
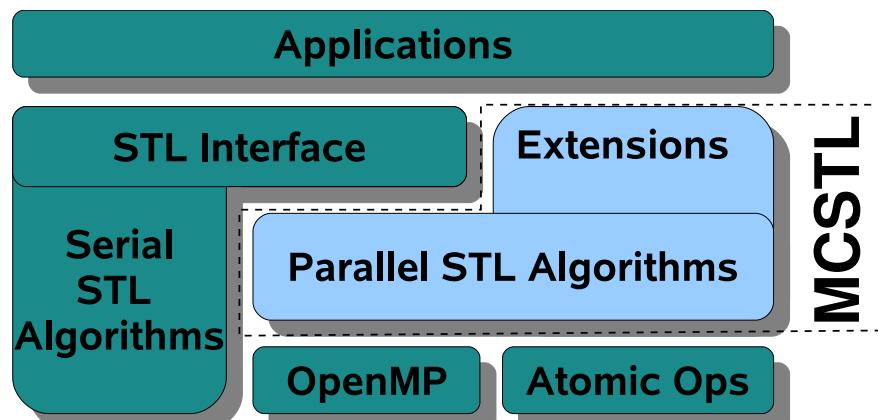
# Experiments

- sometimes a good **surrogate for analysis**
- too much** rather than too little output data
- reproducibility** (10 years!)
- software engineering**

**Stay tuned.**

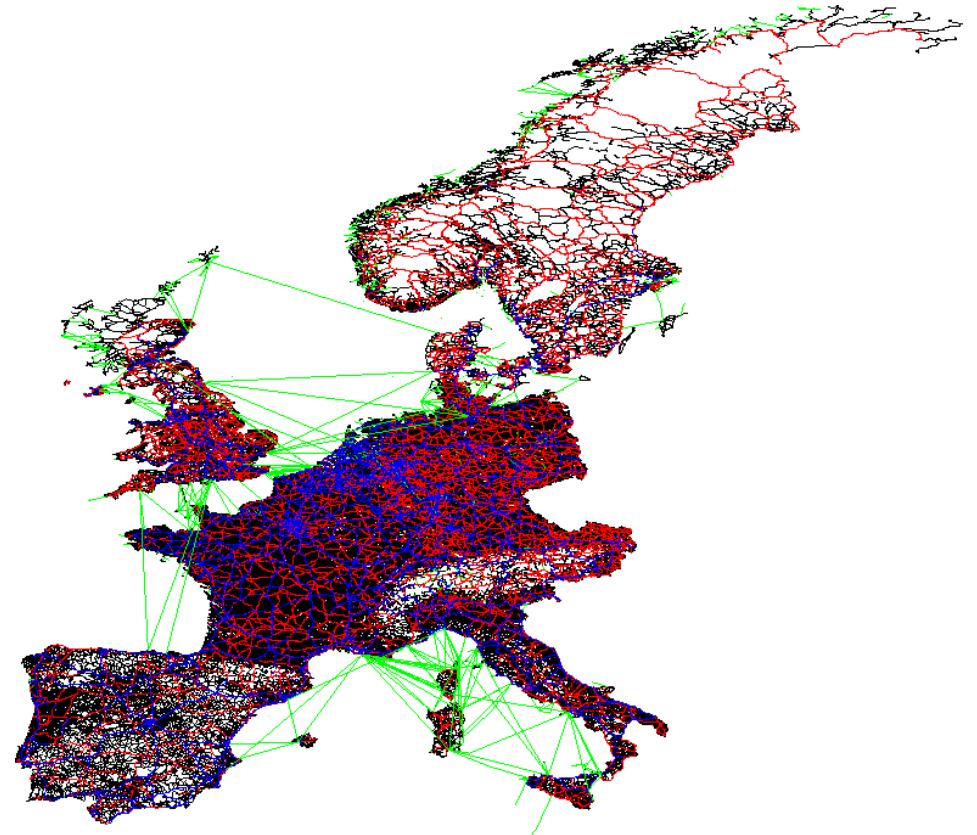
# Algorithm Libraries — Challenges

- software engineering
- standardization, e.g. java.util, C++ STL and BOOST
- performance ↔ generality ↔ simplicity
- applications are a priori unknown
- result checking, verification



# Problem Instances

Benchmark instances are essential for development of practical algorithms





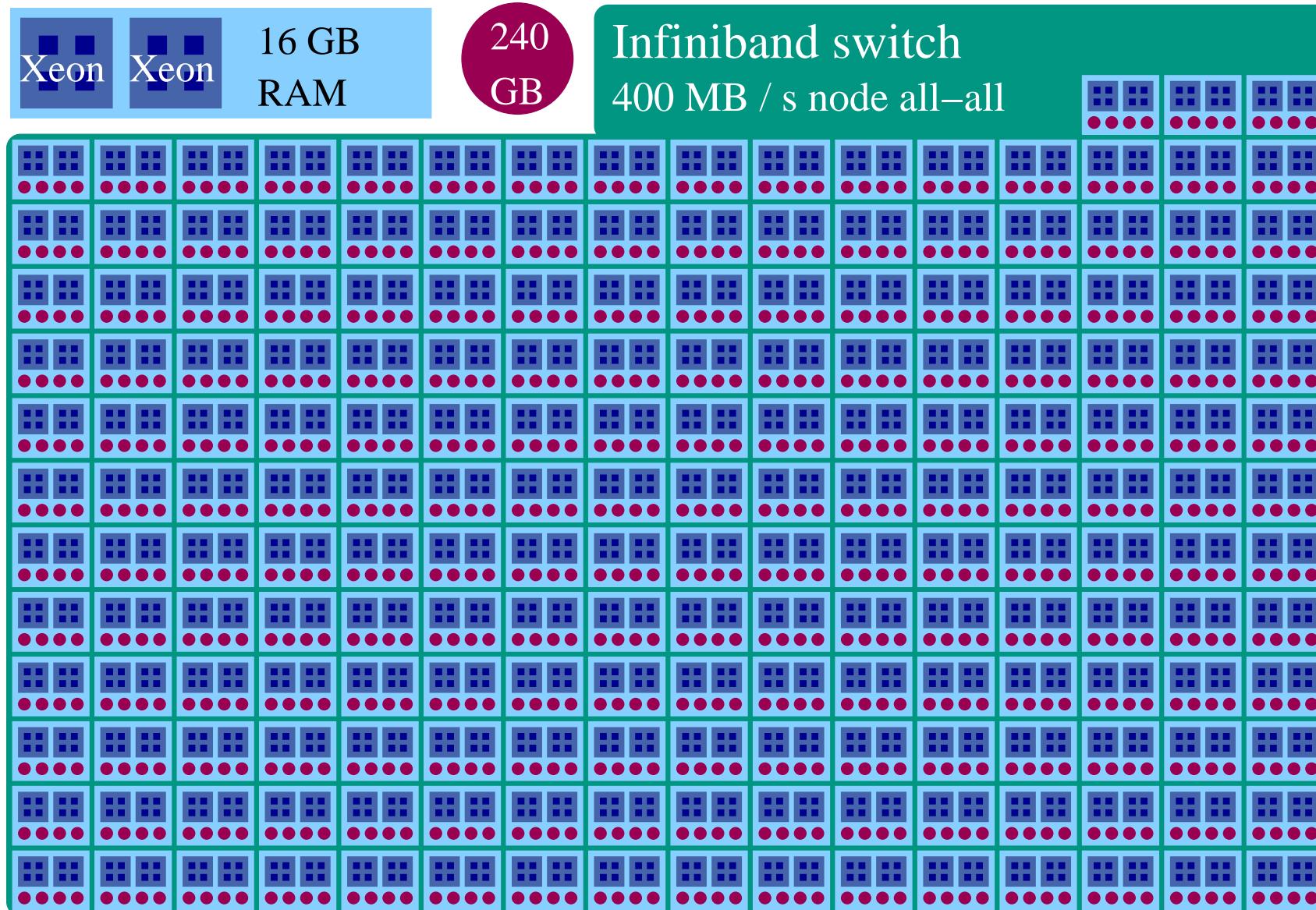
## Example: Sorting Benchmark (Indy)

100 byte records, 10 byte random keys, with file I/O

Category	data volume	performance	improvement
GraySort	100 000 GB	564 GB / min	17×
MinuteSort	955 GB	955 GB / min	> 10×
JouleSort	100 000 GB	3 400 Recs/Joule	???×
JouleSort	1 000 GB	17 500 Recs/Joule	5.1×
JouleSort	100 GB	39 800 Recs/Joule	3.4×
JouleSort	10 GB	43 500 Recs/Joule	5.7×

Also: PennySort

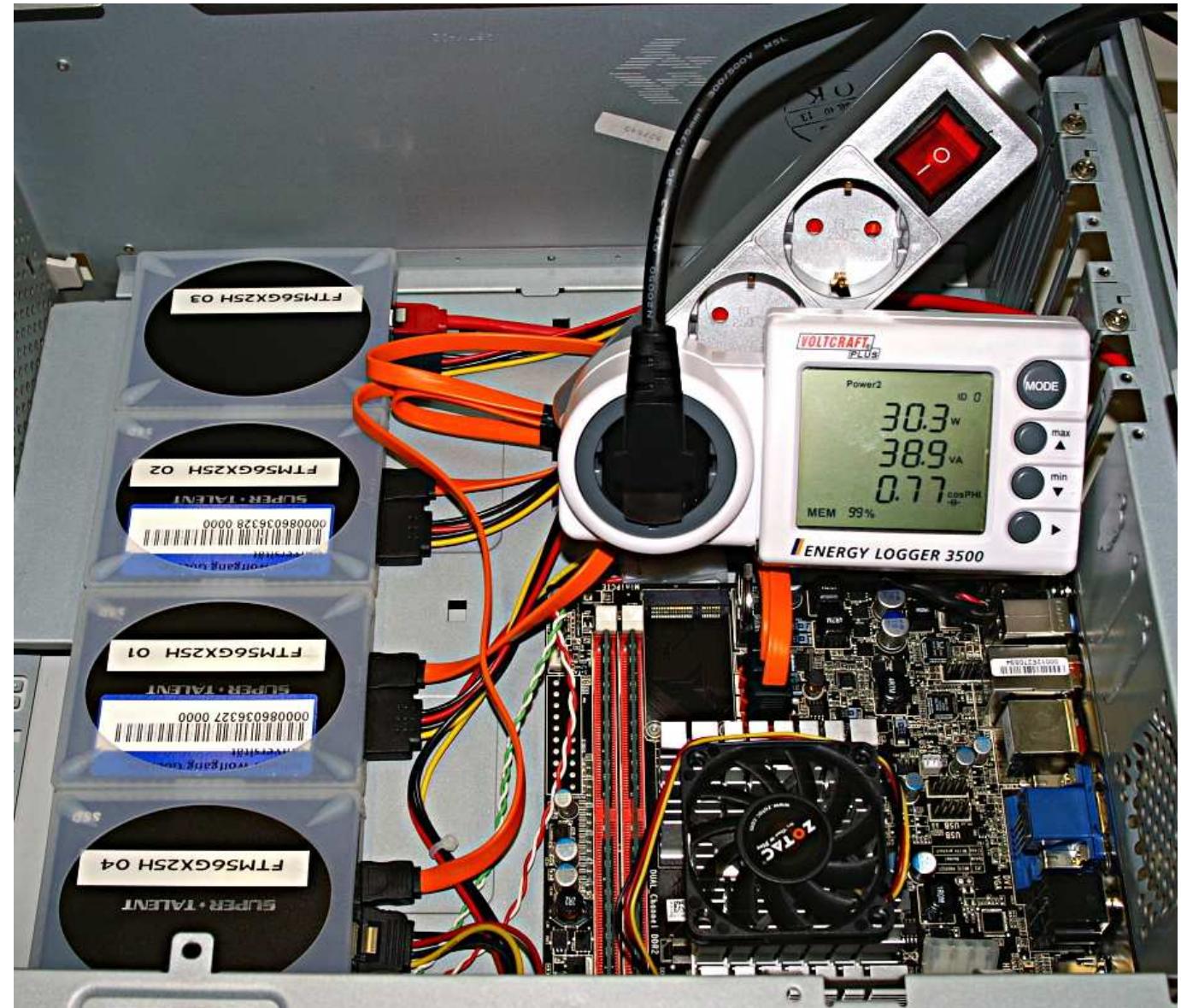
## GraySort: inplace multiway mergesort, exact splitting



# JouleSort

- Intel Atom N330
- 4 GB RAM
- 4 × 256 GB SSD (SuperTalent)

Algorithm similar to  
GraySort



# Applications that “Change the World”

Algorithmics has the potential to SHAPE applications  
(not just the other way round)

[G. Myers]

Bioinformatics: sequencing, proteomics, phylogenetic trees,...



Information Retrieval: Searching, ranking,...

Traffic Planning: navigation, flow optimization,  
adaptive toll, disruption management

Geographic Information Systems: agriculture, environmental  
protection, disaster management, tourism,...

Communication Networks: mobile, P2P, cloud, selfish users,...

## Conclusion:

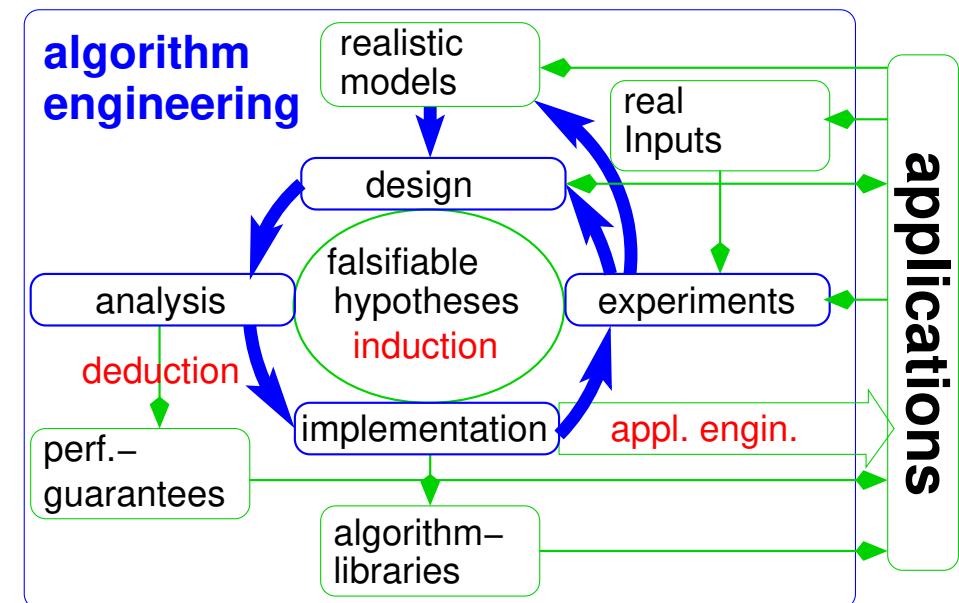
# Algorithm Engineering $\leftrightarrow$ Algorithm Theory

- algorithm engineering is a wider view on algorithmics
  - (but no revolution. None of the ingredients is really new)
- rich methodology
- better coupling to applications
- experimental algorithmics  $\ll$  algorithm engineering
- algorithm theory  $\subset$  algorithm engineering
- sometimes different theoretical questions
- algorithm theory may still yield the strongest, deepest and most persistent results within algorithm engineering

# More On Experimental Methodology

## Scientific Method:

- Experiment need a possible outcome that **falsifies** a hypothesis
- Reproducible
  - keep data/code for at least 10 years
    - + documentation (aka laboratory journal (Laborbuch))
  - clear and detailed description in papers / TRs
  - share instances and code



# Quality Criteria

- Beat the state of the art, globally – (**not your own toy codes or the toy codes used in your community!**)
- Clearly** demonstrate this !
  - both codes use same data ideally from accepted benchmarks  
(not just your favorite data!)
  - comparable machines or fair (conservative) scaling
  - Avoid uncomparabilities like:  
“Yeah we have worse quality but are twice as fast”
  - real world data wherever possible
  - as much **different, fresh** inputs as possible
  - **its fine if you are better just on some (important) inputs**

# Not Here but Important

- describing the setup (machine, compiler, OS, instances, repetitions, . . .)
- finding sources of measurement errors
- reducing measurement errors (averaging, median, unloaded machine . . .)
- measurements in the **creative** phase of experimental algorithmics.

# The Starting Point

- (Several) Algorithm(s)
- A few quantities to be measured: time, space, solution quality, comparisons, cache faults,... There may also be **measurement errors**.
- An unlimited number of potential inputs.  $\rightsquigarrow$  condense to a few characteristic ones (size,  $|V|$ ,  $|E|$ , ... or problem instances from applications)

Usually there is not a lack but an **abundance** of data  $\neq$  many other sciences

# The Process

Waterfall model?

1. Design
2. Measurement
3. Interpretation

Perhaps the paper should at least look like that.

# The Process

- Eventually stop asking questions (Advisors/Referees listen !)
- build measurement tools
- automate (re)measurements
- Choice of Experiments driven by risk and opportunity
- Distinguish mode

**explorative:** many different parameter settings, interactive, short turnaround times

**consolidating:** many large instances, standardized measurement conditions, batch mode, many machines

# Of Risks and Opportunities

Example: Hypothesis = my algorithm is the best

big risk: untried main competitor

small risk: tuning of a subroutine that takes 20 % of the time.

big opportunity: use algorithm for a new application

~~> new input instances

# Fortgeschrittene Graphenalgorithmen

## 2 Kürzeste Wege

Folien teilweise von Rob van Stee

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

Kostenfunktion/Kantengewicht  $c : E \rightarrow \mathbb{R}$

Anfangsknoten  $s$ .

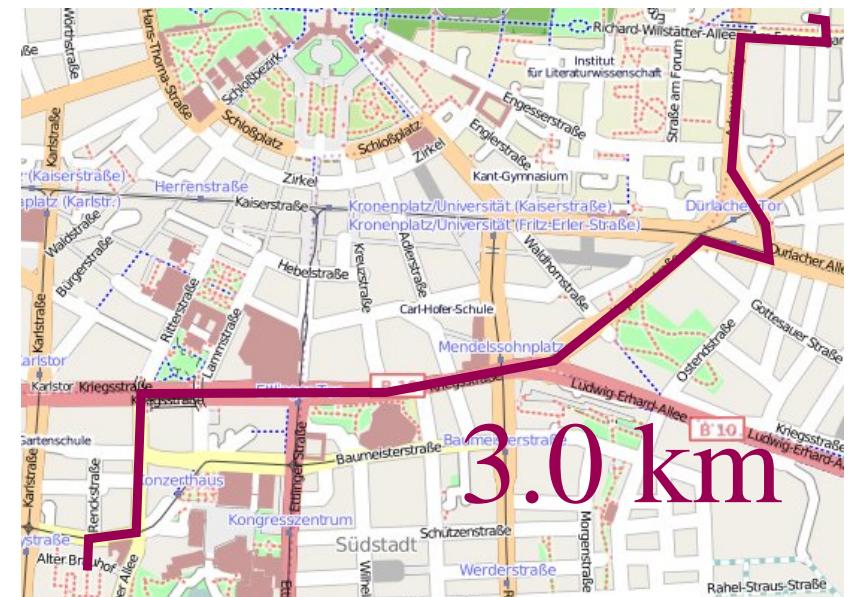
**Ausgabe:** für alle  $v \in V$

Länge  $\mu(v)$  des kürzesten Pfades von  $s$  nach  $v$ ,

$$\mu(v) := \min \{c(p) : p \text{ ist Pfad von } s \text{ nach } v\}$$

$$\text{mit } c(\langle e_1, \dots, e_k \rangle) := \sum_{i=1}^k c(e_i).$$

Oft wollen wir auch „geeignete“ Repräsentation der kürzesten Pfade.



# Allgemeine Definitionen

Wie bei BFS benutzen wir zwei Knotenarrays:

- $d[v]$  = aktuelle (vorläufige) Distanz von  $s$  nach  $v$

**Invariante:**  $d[v] \geq \mu(v)$

- $\text{parent}[v]$  = Vorgänger von  $v$

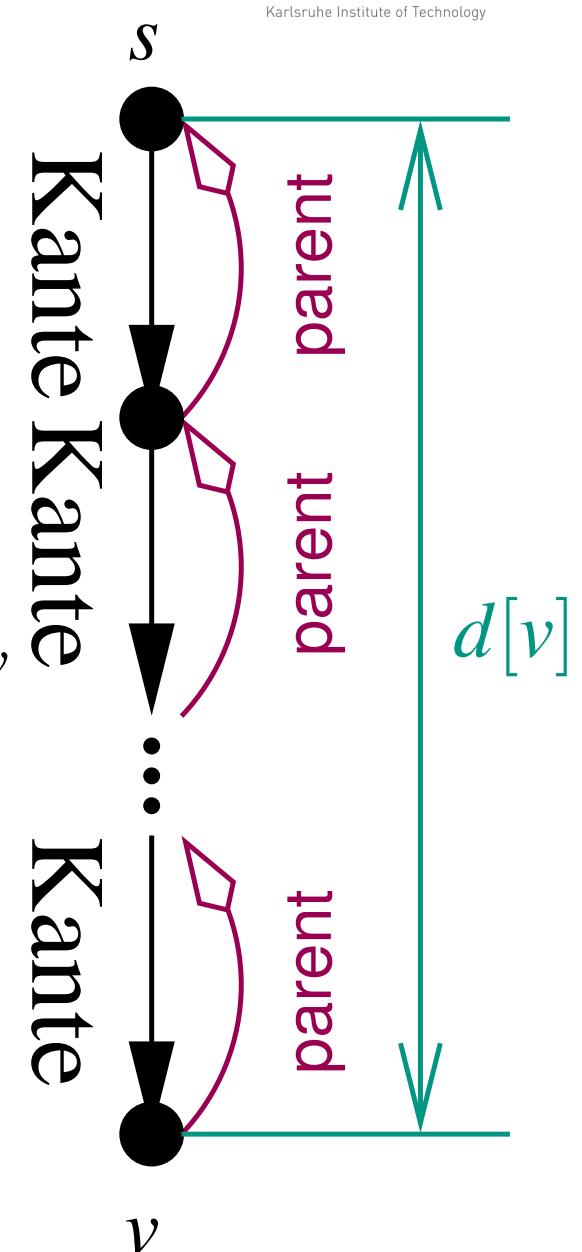
auf dem (vorläufigen) kürzesten Pfad von  $s$  nach  $v$

**Invariante:** dieser Pfad bezeugt  $d[v]$

**Initialisierung:**

$$d[s] = 0, \text{parent}[s] = s$$

$$d[v] = \infty, \text{parent}[v] = \perp$$



## Kante $(u, v)$ relaxieren

falls  $d[u] + c(u, v) < d[v]$

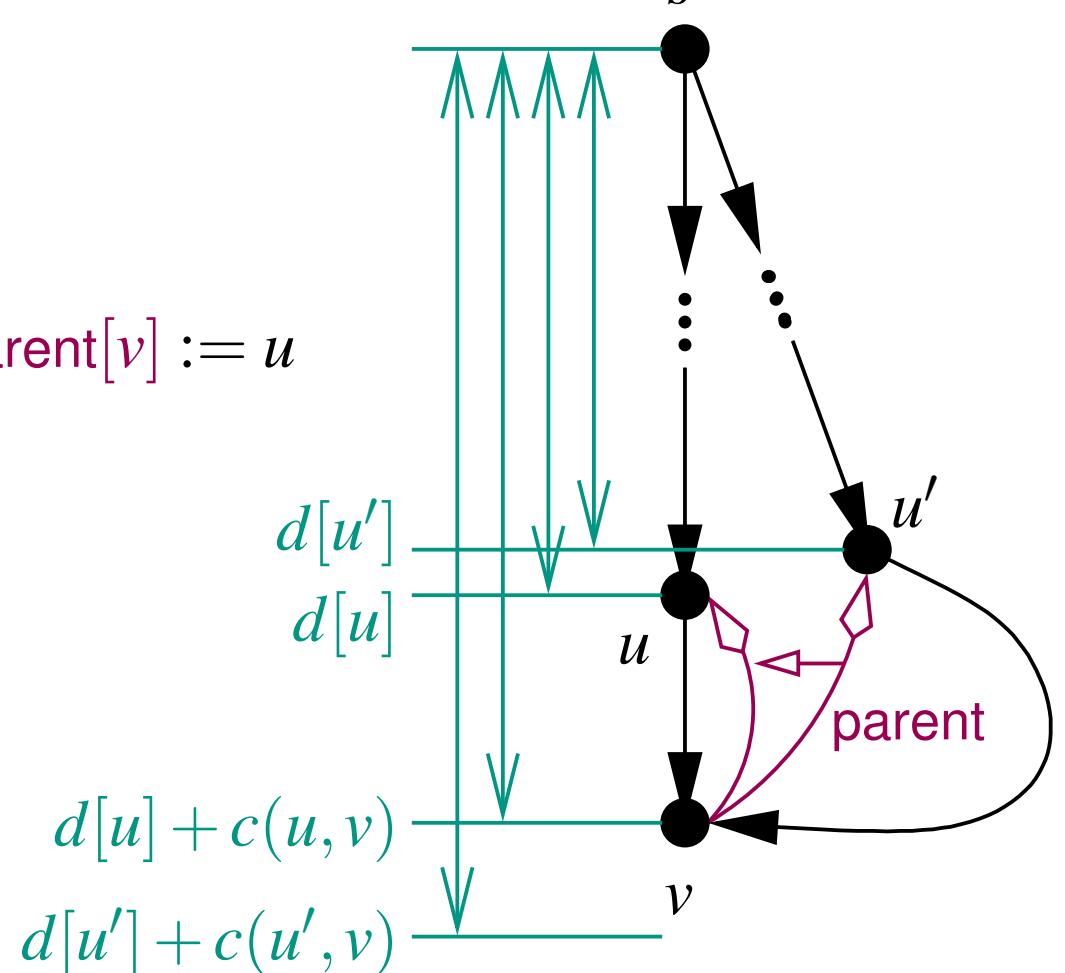
vielleicht  $d[v] = \infty$

setze  $d[v] := d[u] + c(u, v)$  und  $\text{parent}[v] := u$

Invarianten bleiben erhalten!

**Beobachtung:**

$d[v]$  Kann sich mehrmals ändern!



# Dijkstra's Algorithmus: Pseudocode

initialize  $d$ , parent

all nodes are non-scanned

**while**  $\exists$  non-scanned node  $u$  with  $d[u] < \infty$

$u :=$  non-scanned node  $v$  with minimal  $d[v]$

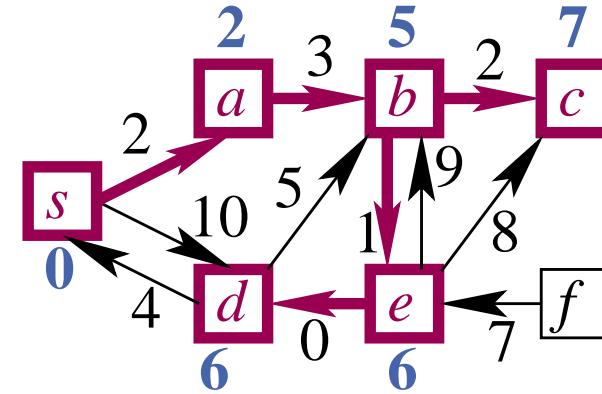
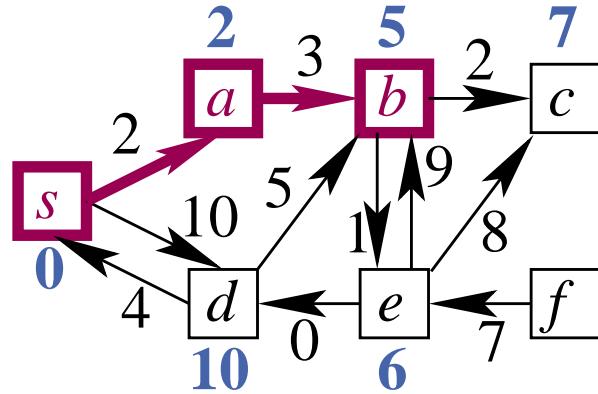
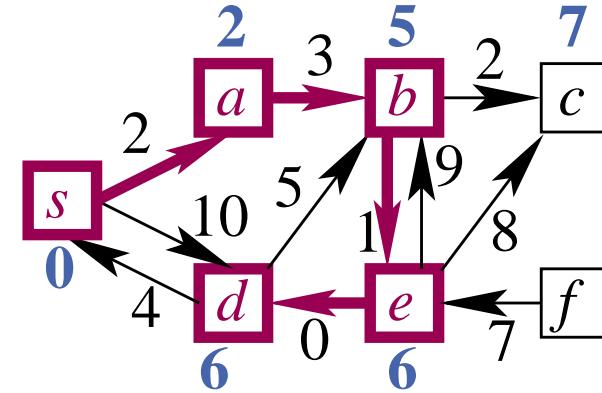
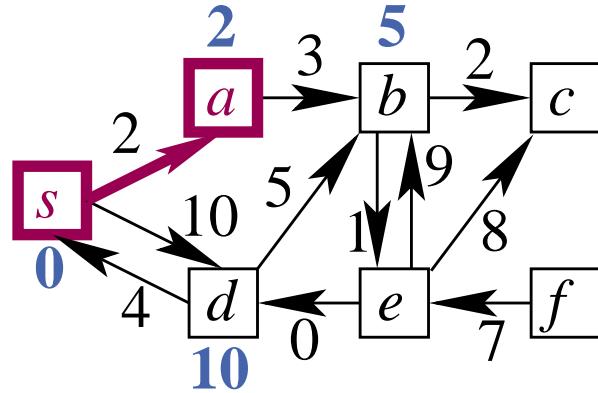
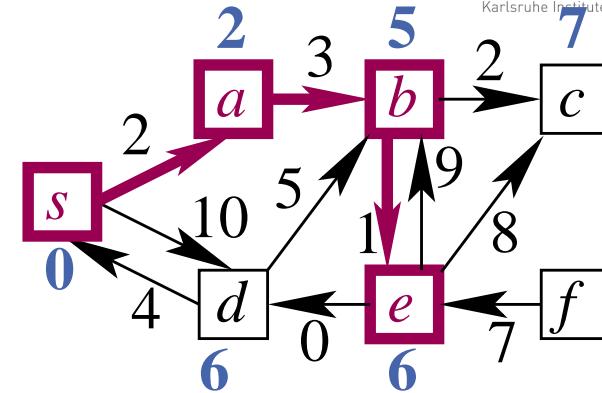
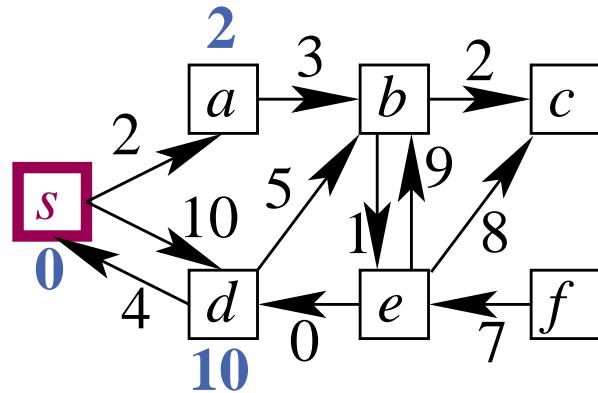
    relax all edges  $(u, v)$  out of  $u$

$u$  is scanned now

**Behauptung:** Am Ende definiert  $d$  die optimalen Entfernungen und parent die zugehörigen Wege (siehe Algo I):

- $v$  erreichbar  $\implies v$  wird irgendwann gescannt
- $v$  gescannt  $\implies \mu(v) = d[v]$

# Beispiel



# Laufzeit

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

Mit Fibonacci-Heapprioritätslisten:

- insert  $O(1)$
- decreaseKey  $O(1)$
- deleteMin  $O(\log n)$  (amortisiert)

$$\begin{aligned} T_{\text{DijkstraFib}} &= O(m \cdot 1 + n \cdot (\log n + 1)) \\ &= O(m + n \log n) \end{aligned}$$

Aber: konstante Faktoren in  $O(\cdot)$  sind hier größer als bei binären Heaps!

# Laufzeit im Durchschnitt

Bis jetzt:  $\leq m$  decreaseKeys ( $\leq 1 \times$  pro Kante)

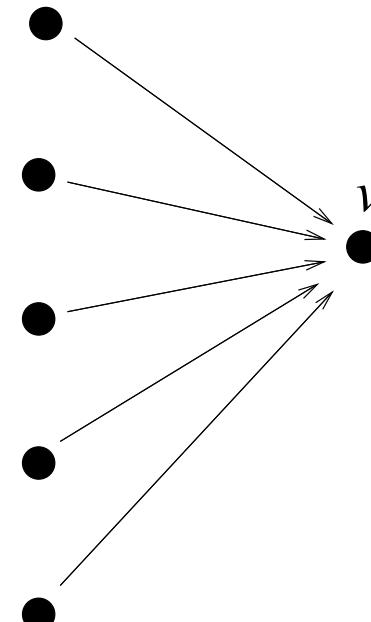
Wieviel decreaseKeys **im Durchschnitt**?

**Modell:**

- Beliebiger Graph  $G$
- Beliebiger Anfangsknoten  $s$
- Beliebige Mengen  $C(v)$   
von Kantengewichten für  
eingehende Kanten von Knoten  $v$

**Gemittelt** wird über alle Zuteilungen  
 $C(v) \rightarrow$  eingehende Kanten von  $v$

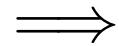
**Beispiel:** alle Kosten unabhängig identisch verteilt



$$\begin{aligned}\text{indegree}(v) &= 5 \\ C(v) &= \{c_1, \dots, c_5\}\end{aligned}$$

## Laufzeit im Durchschnitt

Probabilistische Sichtweise: Zufällige gleichverteilte Auswahl der zu mittelnden Eingaben.



wir suchen den **Erwartungswert** der Laufzeit

**Frage:** Unterschied zu erwarteter Laufzeit bei randomisierten Algorithmen ?

# Laufzeit im Durchschnitt

**Satz 1.**  $\mathbb{E}[\#\text{decreaseKey-Operationen}] = O\left(n \log \frac{m}{n}\right)$

Dann

$$\begin{aligned}\mathbb{E}(T_{\text{DijkstraBHeap}}) &= O\left(m + n \log \frac{m}{n} \cdot T_{\text{decreaseKey}}(n)\right. \\ &\quad \left.+ n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))\right) \\ &= O\left(m + n \log \frac{m}{n} \log n + n \log n\right) \\ &= O\left(m + n \log \frac{m}{n} \log n\right)\end{aligned}$$

(wir hatten vorher  $T_{\text{DijkstraBHeap}} = O((m+n) \log n)$ )

( $T_{\text{DijkstraFib}} = O(m + n \log n)$  schlechtester Fall)

# Lineare Laufzeit für dichte Graphen

$m = \Omega(n \log n \log \log n) \Rightarrow$  lineare Laufzeit.

(nachrechnen)

Also hier u. U. besser als Fibonacci heaps

**Satz 1.**  $\mathbb{E}[\#\text{decreaseKey-Operationen}] = O\left(n \log \frac{m}{n}\right)$

**Satz 1.**  $\mathbb{E}[\#\text{decreaseKey-Operationen}] = O\left(n \log \frac{m}{n}\right)$

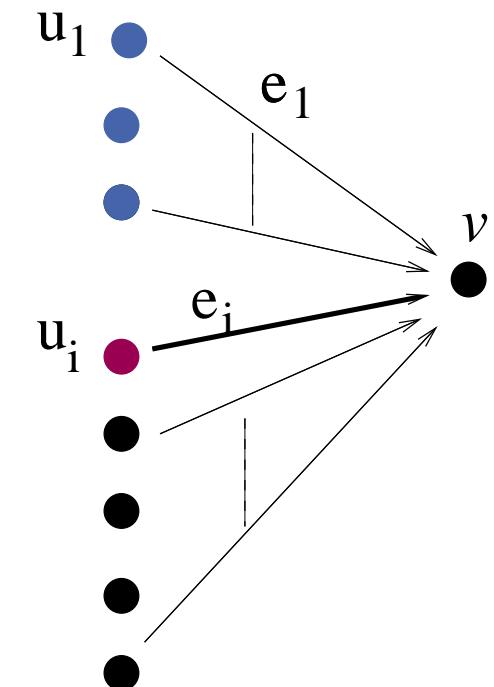
decreaseKey bei Bearbeitung von  $e_i$  nur wenn

$$\mu(u_i) + c(e_i) < \min_{j < i}(\mu(u_j) + c(e_j)).$$

Aber  $\mu(u_i) \geq \mu(u_j)$  für  $j < i$ , also muss gelten:

$$c(e_i) < \min_{j < i} c(e_j).$$

Präfixminimum



**Satz 1.**  $\mathbb{E}[\#\text{decreaseKey-Operationen}] = O\left(n \log \frac{m}{n}\right)$

Kosten in  $C(v)$  erscheinen in **zufälliger Reihenfolge**

Wie oft findet man ein neues Minimum bei zufälliger Reihenfolge?

Harmonische Zahl  $H_k$  (Sect. 2.8, s.u.)

Erstes Minimum: führt zu  $\text{insert}(v)$ .

Also  $\leq H_k - 1 \leq (\ln k + 1) - 1 = \ln k$  erwartete decreaseKeys

**Satz 1.**  $\mathbb{E}[\#\text{decreaseKey-Operationen}] = O\left(n \log \frac{m}{n}\right)$

Für Knoten  $v \leq H_k - 1 \leq \ln k$  decreaseKeys (erwartet) mit  
 $k = \text{indegree}(v)$ .

Insgesamt

$$\sum_{v \in V} \ln \text{indegree}(v) \leq n \ln \frac{m}{n}$$

(wegen Konkavität von  $\ln x$ )

# Präfixminima einer Zufallsfolge

Definiere Zufallsvariable  $M_n$  als Anzahl Präfixminima einer Folge von  $n$  verschiedenen Zahlen (in Abhängigkeit von einer Zufallspermutation)

Definiere Indikatorzufallsvariable  $I_i := 1$  gdw. die  $i$ -te Zahl ein Präfixminimum ist.

$$\begin{aligned} E[M_n] &= E\left[\sum_{i=1}^n I_i\right] \stackrel{\text{Lin. } E[\cdot]}{=} \sum_{i=1}^n E[I_i] \\ &= \sum_{i=1}^n \frac{1}{i} = H_n \text{ wegen } \mathbb{P}[I_i = 1] = \frac{1}{i} \end{aligned}$$

$$\underbrace{x_1, \dots, x_{i-1}}_{< x_i?}, \textcolor{violet}{x_i}, x_{i+1}, \dots, x_n$$

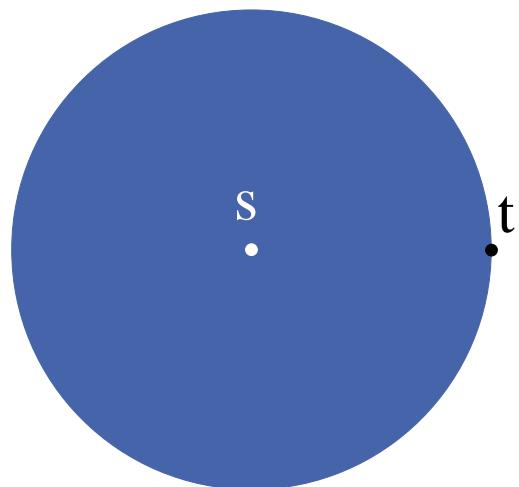
# Monotone ganzzahlige Prioritätslisten

Grundidee: Datenstruktur auf Anwendung **zuschneiden**

Dijkstra's Algorithmus benutzt die **Prioritätsliste monoton**:

Operationen insert und decreaseKey benutzen Distanzen der Form  
 $d[u] + c(e)$

Dieser Wert **nimmt ständig zu**



# Monotone ganzzahlige Prioritätslisten

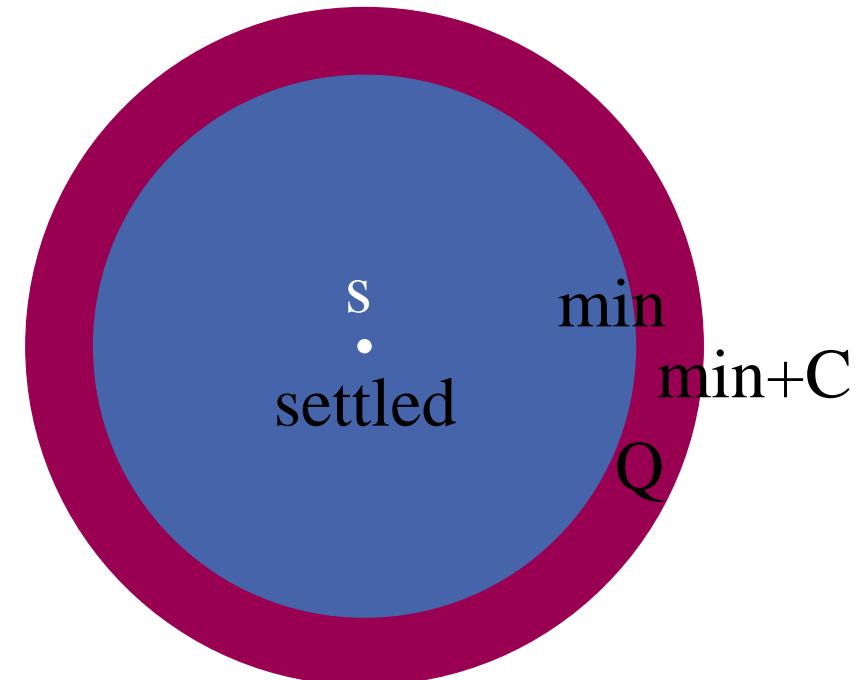
**Annahme:** Alle Kantengewichte sind ganzzahlig und im Intervall  $[0, C]$

$$\implies \forall v \in V : d[v] \leq (n - 1)C$$

Es gilt sogar:

Sei  $\min$  der letzte Wert,  
der aus  $Q$  entfernt wurde.

In  $Q$  sind **immer**  
nur Knoten mit Distanzen im  
Interval  $[\min, \min + C]$ .

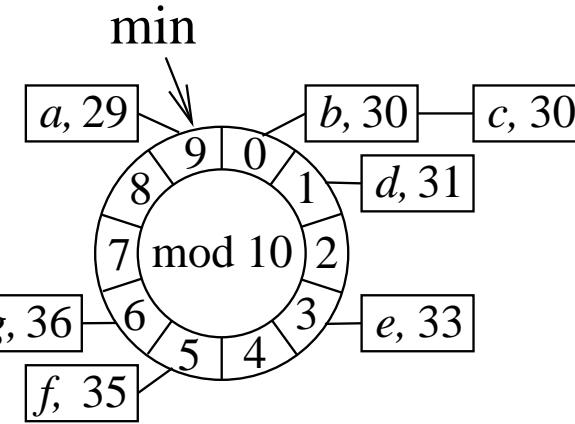


# Bucket-Queue

Zyklisches Array  $B$  von  $C + 1$  doppelt verketteten Listen

Knoten mit Distanz  $d[v]$  wird in  $B[d[v] \bmod (C+1)]$  gespeichert.

Bucket queue with  $C = 9$



Content=

$\langle (a, 29), (b, 30), (c, 30), (d, 31)$   
 $(e, 33), (f, 35), (g, 36) \rangle$

# Operationen

Initialisierung:  $C + 1$  leere Listen,  $\min = 0$

**insert( $v$ )**: fügt  $v$  in  $B[d[v] \bmod (C + 1)]$  ein

**decreaseKey( $v$ )**: entfernt  $v$  aus seiner Liste und

fügt es ein in  $B[d[v] \bmod (C + 1)]$

$O(1)$

$O(1)$

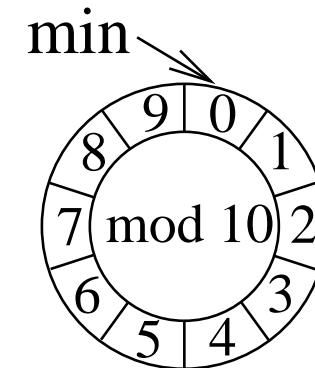
**deleteMin**: fängt an bei Bucket  $B[\min \bmod (C + 1)]$ . Falls der leer ist,  $\min := \min + 1$ , wiederhole.

erfordert Monotonizität!

$\min$  nimmt höchstens  $nC$  mal zu, höchstens  $n$  Elemente werden insgesamt aus Q entfernt  $\Rightarrow$

Gesamtkosten deleteMin-Operationen =  $O(n + nC) = O(nC)$ .

Genauer:  $O(n + \text{maxPathLength})$



# Laufzeit Dijkstra mit Bucket-Queues

$$\begin{aligned} T_{\text{Dijkstra}} &= \mathcal{O}(m \cdot T_{\text{decreaseKey}}(n) \\ &\quad + \text{Kosten deleteMin-Operationen} \\ &\quad + n \cdot T_{\text{insert}}(n))) \end{aligned}$$

$$\begin{aligned} T_{\text{DijkstraBQ}} &= \mathcal{O}(m \cdot 1 + nC + n \cdot 1)) \\ &= \mathcal{O}(m + nC) \text{ oder auch} \\ &= \mathcal{O}(m + \text{maxPathLength}) \end{aligned}$$

Mit Radix-Heaps finden wir sogar  $T_{\text{DijkstraRadix}} = \mathcal{O}(m + n \cdot \log C)$

Idee: nicht alle Buckets gleich groß machen

# Radix-Heaps

Wir verwenden die Buckets  $-1$  bis  $K$ , für  $K = 1 + \lfloor \log C \rfloor$

$\min$  = die zuletzt aus  $Q$  entfernte Distanz

Für jeden Knoten  $v \in Q$  gilt  $d[v] \in [\min, \dots, \min + C]$ .

Betrachte **binäre Repräsentation** der möglichen Distanzen in  $Q$ .

Nehme zum Beispiel  $C = 9$ , binär 1001. Dann  $K = 4$ .

Beispiel 1:  $\min = 10000$ , dann  $\forall v \in Q : d[v] \in [10000, 11001]$

Beispiel 2:  $\min = 11101$ , dann  $\forall v \in Q : d[v] \in [11101, 100110]$

Speichere  $v$  in Bucket  $B[i]$  falls  $d[v]$  und  $\min$  sich **zuerst an der *iten* Stelle unterscheiden**, (in  $B[K]$  falls  $i > K$ , in  $B[-1]$  falls sie sich nicht unterscheiden)

## Definition $msd(a, b)$

Die **Position** der **höchstwertigen** Binärziffer wo  $a$  und  $b$  sich unterscheiden

$a$	1100 <b>1</b> 010	10101 <b>0</b> 0	1110110
$b$	1100 <b>0</b> 101	10101 <b>1</b> 0	1110110
$msd(a, b)$	3	1	-1

$msd(a, b)$  können wir mit Maschinenbefehlen sehr schnell berechnen

# Radix-Heap-Invariante

$v$  ist gespeichert in Bucket  $B[i]$  wo  $i = \min(msd(min, d[v]), K)$ .

Beispiel 1:  $\min = 10000, C = 9, K = 4$

Bucket	$d[v]$ binär	$d[v]$
-1	10000	16
0	10001	17
1	1001*	18,19
2	101**	20–23
3	11***	24–25
4	-	-

(In Bucket 4 wird nichts gespeichert)

# Radix-Heap-Invariante

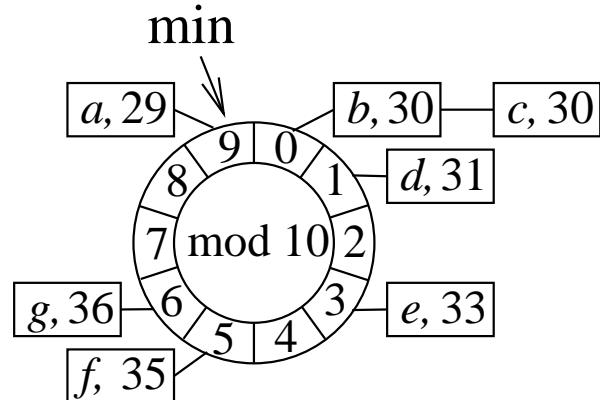
$v$  ist gespeichert in Bucket  $B[i]$  wo  $i = \min(msd(min, d[v]), K)$ .

Beispiel 2:  $\min = 11101$ ,  $C = 9$ , dann  $K = 4$

Bucket	$d[v]$ binär	$d[v]$
-1	11101	29
0	-	-
1	1111*	30,31
2	-	-
3	-	-
4	100000 und höher	32 und höher

Falls  $d[v] \geq 32$ , dann  $msd(\min, d[v]) > 4$ !

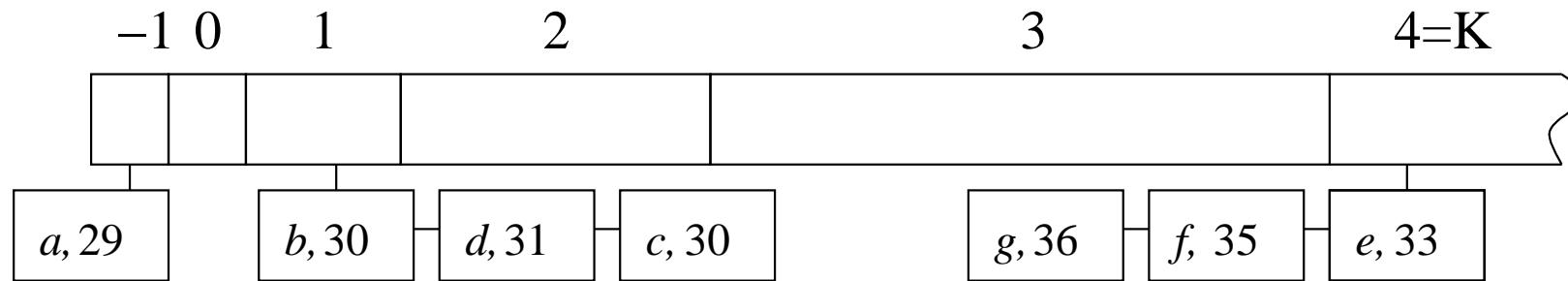
# Bucket-Queues und Radix-Heaps



Bucket queue with  $C = 9$

Content=

$\langle (a, 29), (b, 30), (c, 30), (d, 31), (e, 33), (f, 35), (g, 36) \rangle$



Binary Radix Heap

# Radix Heap: deleteMin

**Function** deleteMin: Element

**if**  $B[-1] = \emptyset$

$i := \min \{j \in 0..K : B[j] \neq \emptyset\}$

move min  $B[i]$  to  $B[-1]$  and to min

**foreach**  $e \in B[i]$  **do** // exactly here invariant is violated !

move  $e$  to  $B[\min(msd(min, d[v]), K)]$

result:=  $B[-1].popFront$

**return** result

$B[0], \dots, B[i-1]$ : leer, also nichts zu tun.

$B[i+1], \dots, B[K]$ : msd bleibt erhalten, weil altes und neues *min*

**gleich** für alle Bits  $j > i$

# Buckets $j > i$ bei Änderung von min

Beispiel:  $\min = 10000, C = 9, K = 4.$

Neues  $\min = 10010$ , war in Bucket 1

	$\min = 10000$		$\min = 10010$	
Bucket	$d[v]$ binär	$d[v]$	$d[v]$ binär	$d[v]$
-1	10000	16	10010	18
0	-	-	10011	19
1	1001*	18,19	-	-
2	101**	20–23	101**	20-23
3	11***	24–25	11***	24-27
4	-	-	-	-

# Bucket $B[i]$ bei Änderung von $\min$

**Lemma:** Elemente  $x$  aus  $B[i]$  gehen zu Buckets mit **kleineren** Indices

Wir zeigen nur den Fall  $i < K$ .

Sei  $\min_o$  der alte Wert von  $\min$ .

Case  $i < K$

	$i$	0	
$\min_o$	$\alpha$	0	
$\min$	$\alpha$	1	
$x$	$\alpha$	1	

## Kosten der deleteMin-Operationen

Bucket  $B[i]$  finden:  $O(i)$

Elemente aus  $B[i]$  verschieben:  $O(|B[i]|)$

Insgesamt  $O(K + |B[i]|)$  falls  $i \geq 0$ ,  $O(1)$  falls  $i = -1$

Verschiebung erfolgt immer nach **kleineren Indices**

Wir zahlen dafür **schon beim insert** (amortisierte Analyse):

es gibt höchstens  $K$  Verschiebungen eines Elements

# Laufzeit Dijkstra mit Radix-Heaps

Insgesamt finden wir amortisiert

- $T_{\text{insert}}(n) = \mathcal{O}(K)$
- $T_{\text{deleteMin}}(n) = \mathcal{O}(K)$
- $T_{\text{decreaseKey}}(n) = \mathcal{O}(1)$

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

$$T_{\text{DijkstraRadix}} = \mathcal{O}(m + n \cdot (K + K)) = \mathcal{O}(m + n \cdot \log C)$$

# Lineare Laufzeit für zufällige Kantengewichte

**Vorher gesehen:** Dijkstra mit binary heaps hat lineare Laufzeit für dichte Graphen ( $m > n \log n \log \log n$ )

**Letzte Folie:**  $T_{\text{DijkstraRadix}} = O(m + n \cdot \log C)$

**Jetzt:** Dijkstra mit Radix-Heaps hat lineare Laufzeit ( $O(m + n)$ ) falls Kantenkosten identisch uniform verteilt in  $0..C$   
– wir brauchen nur eine kleine Änderung im Algorithmus

# Änderung im Algorithmus für zufällige Kantengewichte

Vorberechnung von  $c_{\min}^{in}(v) := \min \{c((u, v) : (u, v) \in E\}$  leichtestes  
eingehendes Kantengewicht.

Beobachtung:  $d[v] \leq \min + c_{\min}^{in}(v)$

$\implies d[v] = \mu(v).$

$\implies$  schiebe  $v$  in Menge  $F$  ungescannter Knoten mit korrekter Distanz

Knoten in  $F$  werden bei nächster Gelegenheit gescannt.

( $\approx F$  als Erweiterung von  $B[-1]$ .)



# Analyse

Ein Knoten  $v$  kommt **nie** in einem Bucket  $i$  mit  $i < \log c_{\min}^{\text{in}}(v)$

Also wird  $v$  höchstens  $K + 1 - \log c_{\min}^{\text{in}}(v)$  mal verschoben

**Kosten von Verschiebungen** sind dann insgesamt höchstens

$$\sum_v (K - \log c_{\min}^{\text{in}}(v) + 1) = n + \sum_v (K - \log c_{\min}^{\text{in}}(v)) \leq n + \sum_e (K - \log c(e)).$$

$K - \log c(e)$  = Anzahl Nullen am Anfang der binären Repräsentation von  $c(e)$  als  $K$ -Bit-Zahl.

$$\mathbb{P}(K - \log c(e) = i) = 2^{-i} \Rightarrow \mathbb{E}(K - \log c(e)) = \sum_{i \geq 0} i 2^{-i} \leq 2$$

Laufzeit =  $\mathcal{O}(m + n)$

# All-Pairs Shortest Paths

Bis jetzt gab es immer einen bestimmten Anfangsknoten  $s$

Wie können wir kürzeste Pfade für **alle** Paare  $(u, v)$  in  $G$  bestimmen?

Annahme: negative Kosten erlaubt, aber keine negativen **Kreise**

Lösung 1:  $n$  mal Bellman-Ford ausführen

... Laufzeit  $O(n^2m)$

Lösung 2: **Knotenpotentiale**

... Laufzeit  $O(nm + n^2 \log n)$ , deutlich schneller

# Knotenpotentiale

Jeder Knoten bekommt ein Potential  $\text{pot}(v)$

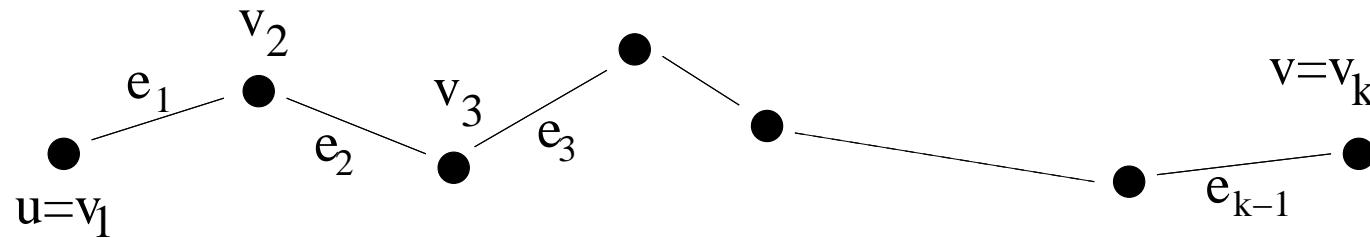
Mit Hilfe der Potentiale definieren wir **reduzierte Kosten**  $\bar{c}(e)$  für Kante  $e = (u, v)$  als

$$\bar{c}(e) = \text{pot}(u) + c(e) - \text{pot}(v).$$

Mit diesen Kosten finden wir die **gleichen** kürzesten Pfade wie vorher!

Gilt für **alle** möglichen Potentiale – wir können sie also frei definieren

# Knotenpotentiale



Sei  $p$  ein Pfad von  $u$  nach  $v$  mit Kosten  $c(p)$ . Dann

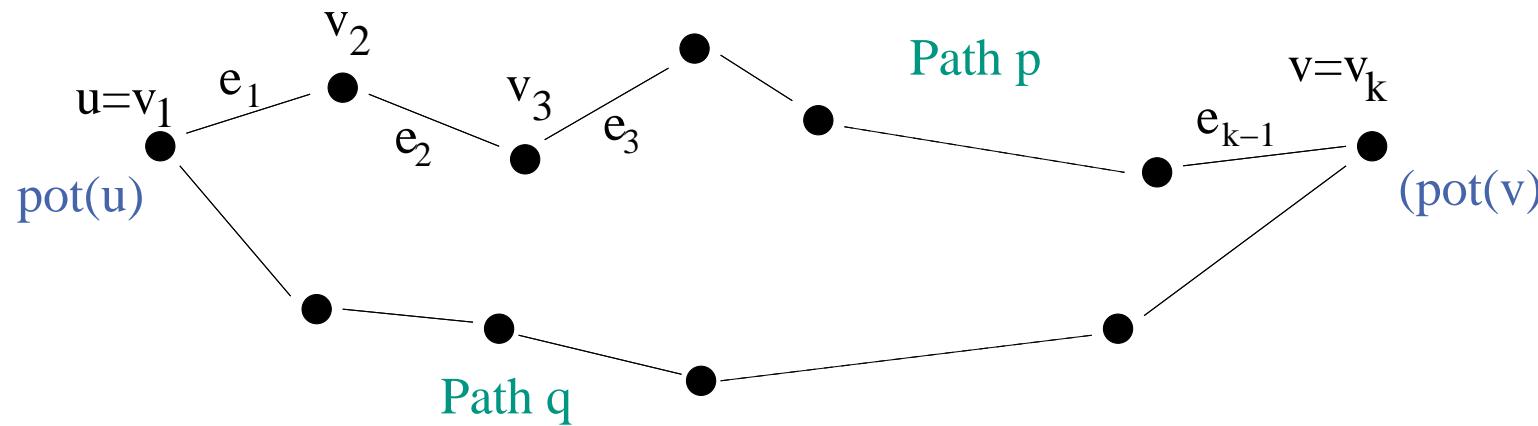
$$\begin{aligned}
 \bar{c}(p) &= \sum_{i=1}^{k-1} \bar{c}(e_i) = \sum_{i=1}^{k-1} (\text{pot}(v_i) + c(e_i) - \text{pot}(v_{i+1})) \\
 &= \text{pot}(v_1) + \sum_{i=1}^{k-1} c(e_i) - \text{pot}(v_k) \\
 &= \text{pot}(v_1) + c(p) - \text{pot}(v_k).
 \end{aligned}$$

# Knotenpotentiale

Sei  $p$  ein Pfad von  $u$  nach  $v$  mit Kosten  $c(p)$ . Dann

$$\bar{c}(p) = \text{pot}(v_1) + c(p) - \text{pot}(v_k).$$

Sei  $q$  ein anderer  $u$ - $v$ -Pfad, dann  $\text{c}(p) \leq \text{c}(q) \Leftrightarrow \bar{c}(p) \leq \bar{c}(q)$ .



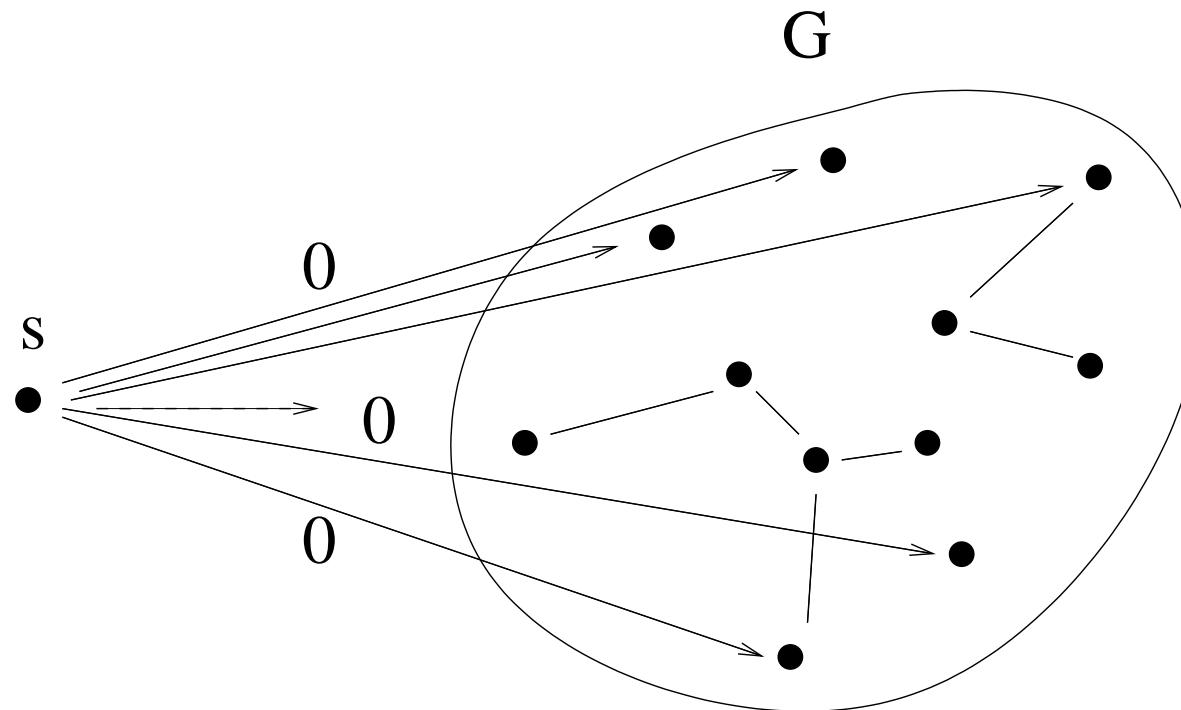
**Definition:**  $\mu(u, v)$  = kürzeste Distanz von  $u$  nach  $v$

# Hilfsknoten

Wir fügen einen **Hilfsknoten  $s$**  an  $G$  hinzu

Für alle  $v \in V$  fügen wir eine Kante  $(s, v)$  hinzu mit Kosten 0

Berechne kürzeste Pfade **von  $s$  aus** mit Bellman-Ford



# Definition der Potentiale

Definiere  $\text{pot}(v) := \mu(v)$  für alle  $v \in V$

Jetzt sind die reduzierten Kosten alle **nicht negativ**: also können wir Dijkstra benutzen! (Evtl.  $s$  wieder entfernen...)

- Keine negativen Kreise, also  $\text{pot}(v)$  wohldefiniert
- Für beliebige Kante  $(u, v)$  gilt

$$\mu(u) + c(e) \geq \mu(v)$$

deshalb

$$\bar{c}(e) = \mu(u) + c(e) - \mu(v) \geq 0$$

# Algorithmus

## All-Pairs Shortest Paths in the Absence of Negative Cycles

neuer Knoten  $s$

**foreach**  $v \in V$  **do** füge Kante  $(s, v)$  ein (Kosten 0) //  $O(n)$

$\text{pot} := \mu := \text{BellmanFordSSSP}(s, c)$  //  $O(nm)$

**foreach** Knoten  $x \in V$  **do** //  $O(n(m + n \log n))$

$\bar{\mu}(x, \cdot) := \text{DijkstraSSSP}(x, \bar{c})$

// zurück zur ursprünglichen Kostenfunktion

**foreach**  $e = (v, w) \in V \times V$  **do** //  $O(n^2)$

$\mu(v, w) := \bar{\mu}(v, w) + \text{pot}(w) - \text{pot}(v)$

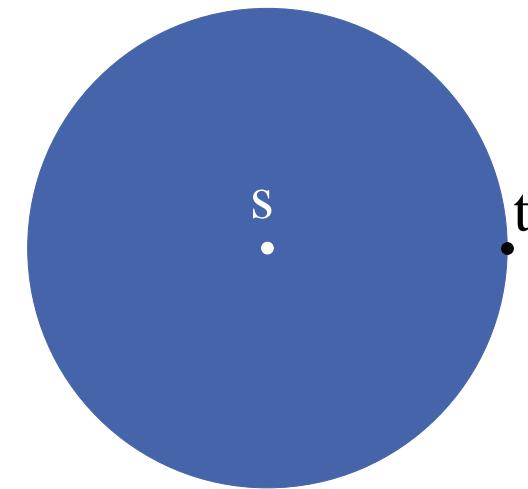
# Laufzeit

- $s$  hinzufügen:  $O(n)$
- Postprocessing:  $O(n^2)$  (zurück zu den ursprünglichen Kosten)
- $n$  mal Dijkstra dominiert

Laufzeit  $O(n(m + n \log n)) = O(nm + n^2 \log n)$

# 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

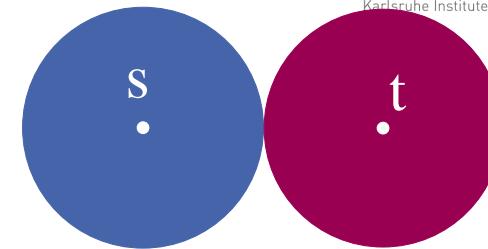
Spart "im Durchschnitt" Hälfte der Scans

Frage: Wieviel spart es (meist) beim Europa-Navi?



# Ideen für Routenplanung

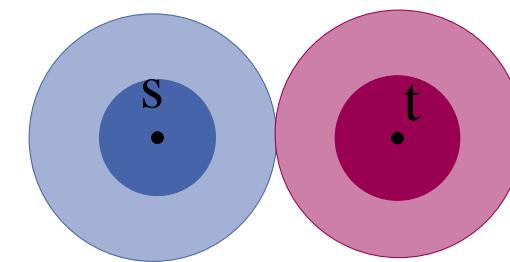
Vorwärts + Rückwärtssuche



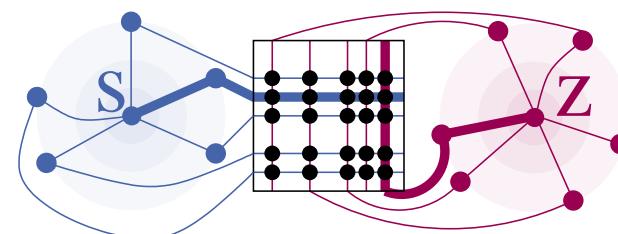
Zielgerichtete Suche



Hierarchien ausnutzen 

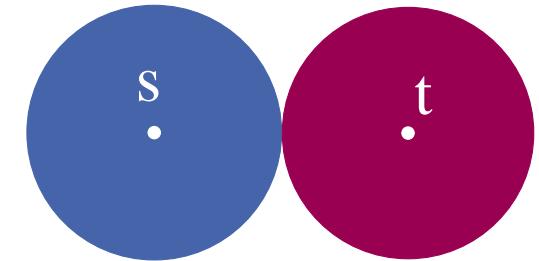


Teilabschnitte tabellieren 



## Bidirektionale Suche

Idee: Suche abwechselnd  $s$  und  $t$



Vorwärtssuche auf normalem Graph  $G = (V, E)$

Rückwärtssuche auf Rückwärtsgraph  $G^r = (V, E^r)$

(Suchrichtungen wechseln in jedem Schritt)

Vorläufige kürzeste Distanz wird in jedem Schritt gespeichert:

$$d[s, t] = \min(d[s, t], d_{\text{forward}}[u] + d_{\text{backward}}[u])$$

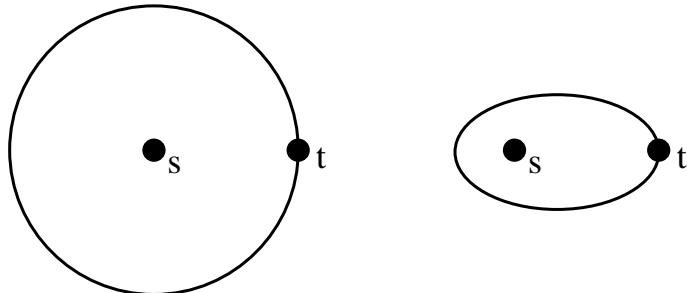
Abbruchkriterium:

Suche scannt Knoten, der in anderer Richtung bereits gescannt wurde.

$$d[s, t] \Rightarrow \mu(s, t)$$

# $A^*$ -Suche

Idee: suche “in die Richtung von  $t$ ”

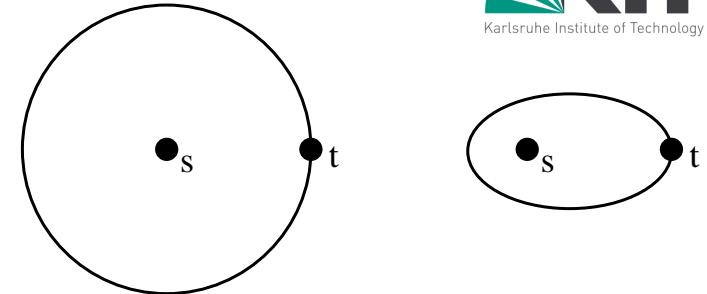


Annahme: Wir kennen eine Funktion  $f(v)$  die  $\mu(v, t)$  schätzt  $\forall v$

Definiere  $\text{pot}(v) = f(v)$  und  $\bar{c}(u, v) = c(u, v) + f(v) - f(u)$

[Oder: in Dijkstra's Algorithmus, entferne nicht  $v$  mit minimalem  $d[v]$  aus  $Q$ , sondern  $v$  mit minimalem  $d[v] + f[v]$ ]

# $A^*$ -Suche



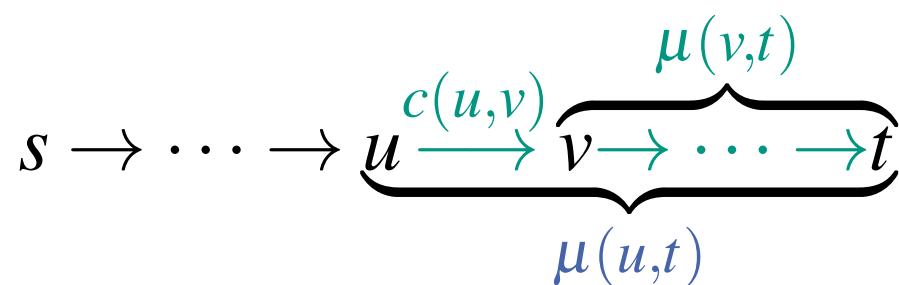
Idee: suche “in die Richtung von  $t$ ”

Annahme: wir kennen eine Funktion  $f(v)$  die  $\mu(v, t)$  schätzt  $\forall v$

Definiere  $\text{pot}(v) = f(v)$  und  $\bar{c}(u, v) = c(u, v) + f(v) - f(u)$

Beispiel:  $f(v) = \mu(v, t)$ .

Dann gilt:  $\bar{c}(u, v) = c(u, v) + \mu(v, t) - \mu(u, t) = 0$  falls  $(u, v)$  auf dem kürzesten Pfad von  $s$  nach  $t$  liegt.



Also scannt Dijkstra nur die Knoten auf diesem Pfad!

## Benötigte Eigenschaften von $f(v)$

- Konsistenz (reduzierte Kosten nicht negativ):  
 $c(e) + f(v) \geq f(u) \quad \forall e = (u, v)$
- $f(v) \leq \mu(v, t) \quad \forall v \in V$
- $f(t) = 0$  dann können wir aufhören, wenn  $t$  aus  $Q$  entfernt wird

Sei  $p$  irgendein Pfad von  $s$  nach  $t$ .

Alle Kanten auf  $p$  sind relaxiert?  $\Rightarrow d[t] \leq c(p)$ .

Sonst:  $\exists v \in p \cap Q$ , und  $d[t] + f(t) \leq d[v] + f(v)$  weil  $t$  schon entfernt wurde. Deshalb

$$d[t] = d[t] + f(t) \leq d[v] + f(v) \leq d[v] + \mu(v, t) \leq c(p)$$

# Wie finden wir $f(v)$ ?

Wir brauchen Heuristiken für  $f(v)$ .

Strecke im Straßennetzwerk:  $f(v) = \text{euklidischer Abstand } ||v - t||_2$

bringt deutliche aber nicht überragende Beschleunigung

Fahrzeit:  $\frac{||v - t||_2}{\text{Höchstgeschwindigkeit}}$  praktisch nutzlos

Noch besser aber mit Vorberechnung: **Landmarks**

# Landmarks

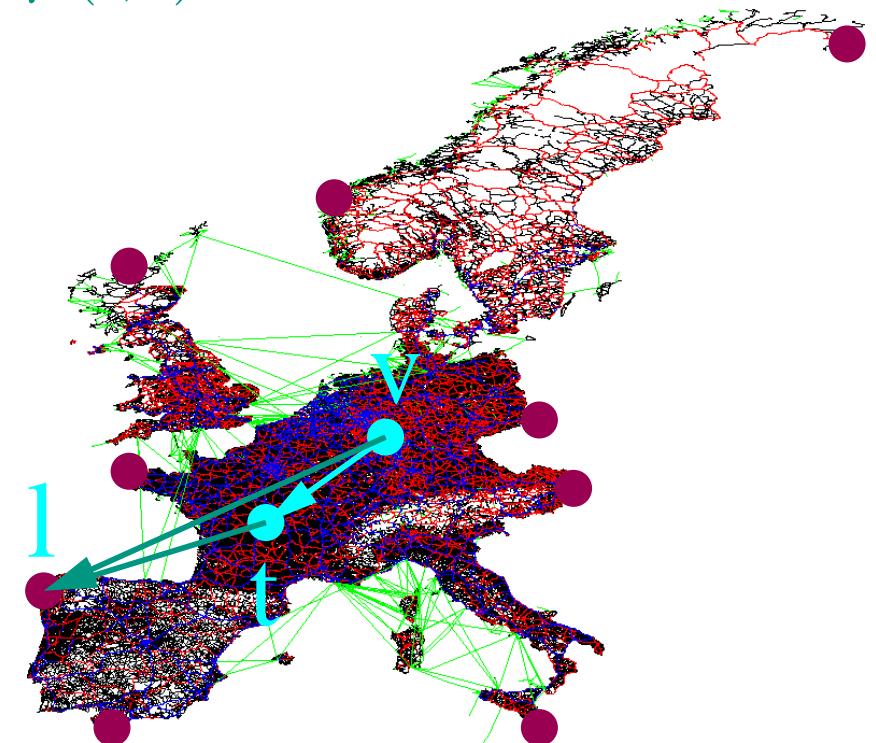
[Goldberg Harrelson 2003]

**Vorberechnung:** Wähle **Landmarkmenge  $L$ .**  $\forall \ell \in L, v \in V$  berechne/speichere  $\mu(v, \ell)$ .

**Query:** Suche Landmark  $\ell \in L$  "hinter" dem Ziel.

Benutze untere Schranke  $f_\ell(v) = \mu(v, \ell) - \mu(t, \ell)$

- + Konzeptuell einfach
- + Erhebliche Beschleunigungen  
( $\approx$  Faktor 20 im Mittel)
- + Kombinierbar mit anderen Techniken
- Landmarkauswahl kompliziert
- hoher Platzverbrauch



## Zusammenfassung Kürzeste Wege

- Nichtriviale Beispiele für Analyse im Mittel. Ähnlich für MST
- Monotone, ganzzahlige Prioritätslisten als Beispiel wie Datenstrukturen auf Algorithmus angepasst werden
- Knotenpotentiale allgemein nützlich in Graphenalgorithmen
- Aktuelle Forschung trifft klassische Algorithmik

# 3 Fortgeschrittene Datenstrukturen

Hier am Beispiel von Prioritätslisten.

Weitere Beispiele:

- Monotone ganzzahlige Prioritätslisten      Kapitel kürzeste Wege
- perfektes Hashing      Kapitel rand. Alg.
- Suchbäume mit fortgeschrittenen Operationen      siehe Buch
- Externe Prioritätslisten      Kapitel externe Algorithmen
- Geometrische Datenstrukturen      Kapitel geom. Algorithmen

## 3.1 Adressierbare Prioritätslisten

**Procedure** build( $\{e_1, \dots, e_n\}$ )  $M := \{e_1, \dots, e_n\}$

**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

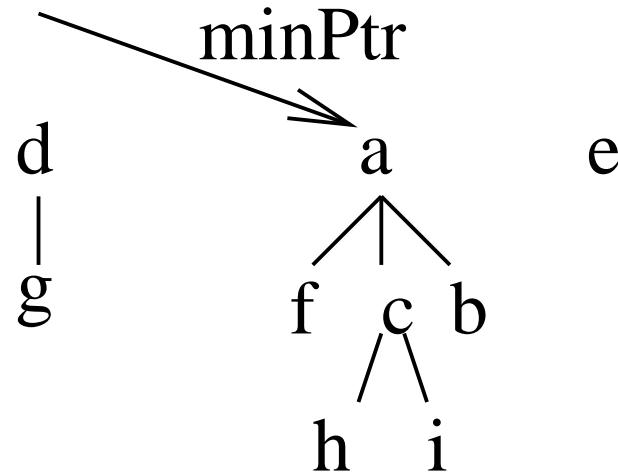
- Dijkstras Algorithmus für **kürzeste Wege**
- Jarník-Prim-Algorithmus für **minimale Spannbäume**
- Bei uns: Hierarchiekonstruktion für **Routenplanung**
- Bei uns: Graphpartitionierung
- Bei uns: disk scheduling

Allgemein:

**Greedy-Algorithmen**, bei denen sich Prioritäten (begrenzt) ändern.

# Grundlegende Datenstruktur

Ein **Wald heap-geordneter Bäume**

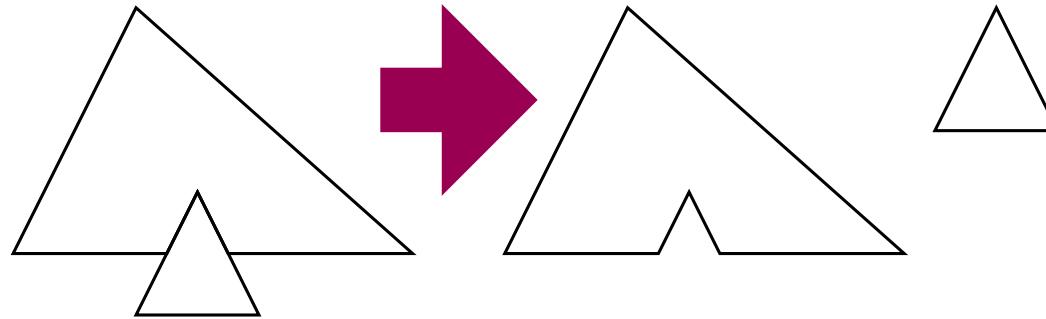


Verallgemeinerung gegenüber binary heap:

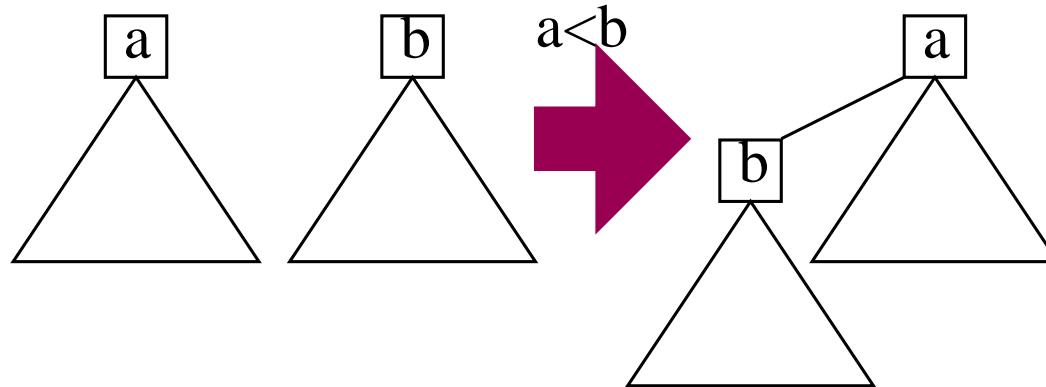
- Baum → Wald
- binär → beliebige Knotengrade

# Wälder Bearbeiten

Cut:



Link:



$\text{union}(a, b) : \text{link}(\min(a, b), \max(a, b))$

# Pairing Heaps (Paarungs-Haufen??)

[Fredman Sedgewick Sleator Tarjan 1986]

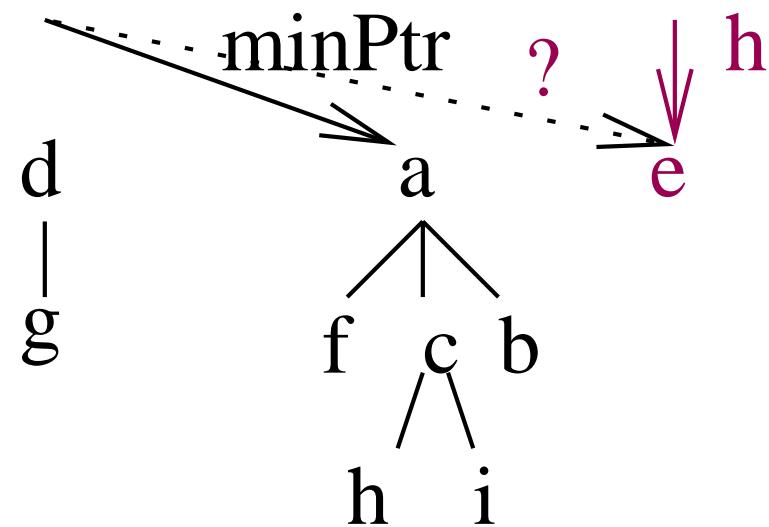
**Procedure** insertItem( $h$  : Handle)

newTree( $h$ )

**Procedure** newTree( $h$  : Handle)

forest:= forest  $\cup \{h\}$

**if**  $*h < \text{min}$  **then** minPtr:=  $h$



**Vorsicht:** sehr einfache Variante aus 1. engl. Auflage.

dt. Ausgabe, 2. Auflage weichen ab

# Pairing Heaps

**Procedure** decreaseKey( $h$  : Handle,  $k$  : Key)

key( $h$ ) :=  $k$

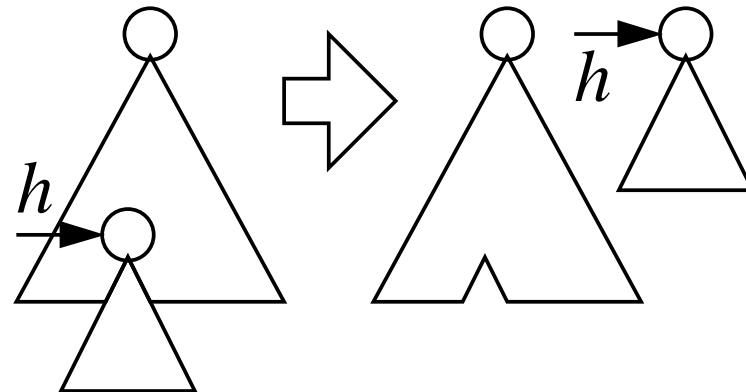
**if**  $h$  is not a root **then** cut( $h$ )

**else** update minPtr

**Procedure** cut( $h$  : Handle)

remove the subtree rooted at  $h$

newTree( $h$ )



# Pairing Heaps

**Function** deleteMin : Handle

*m*:= minPtr

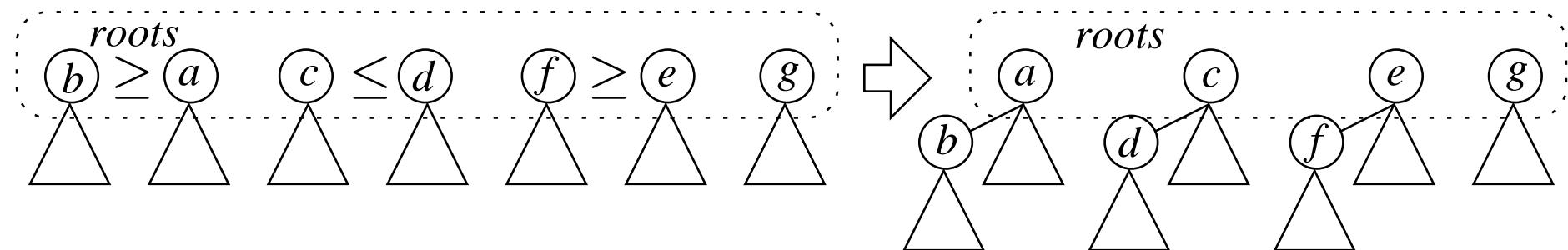
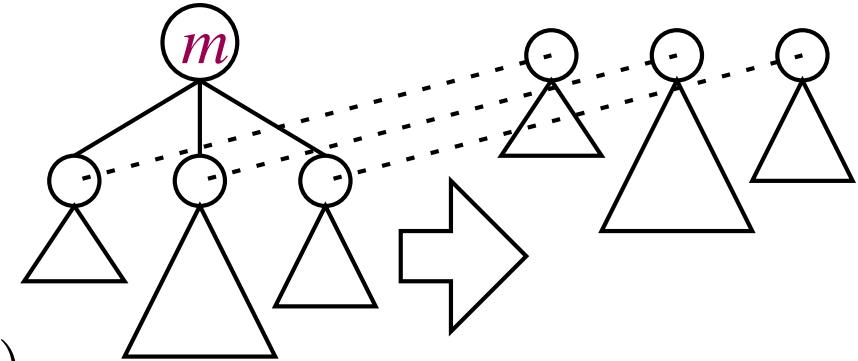
forest:= forest \ {*m*}

**foreach** child *h* of *m* **do** newTree(*h*)

perform pair-wise union operations on the roots in forest

update minPtr

**return** *m*



# Pairing Heaps

**Procedure** merge(*o* : AdressablePQ)

**if** \*minPtr > \*(*o*.minPtr) **then** minPtr:= *o*.minPtr

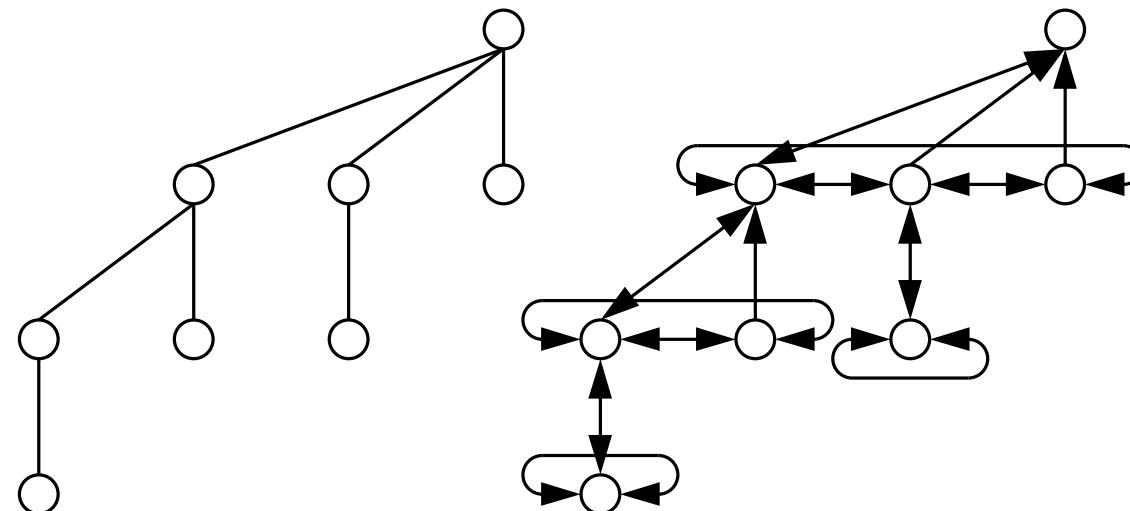
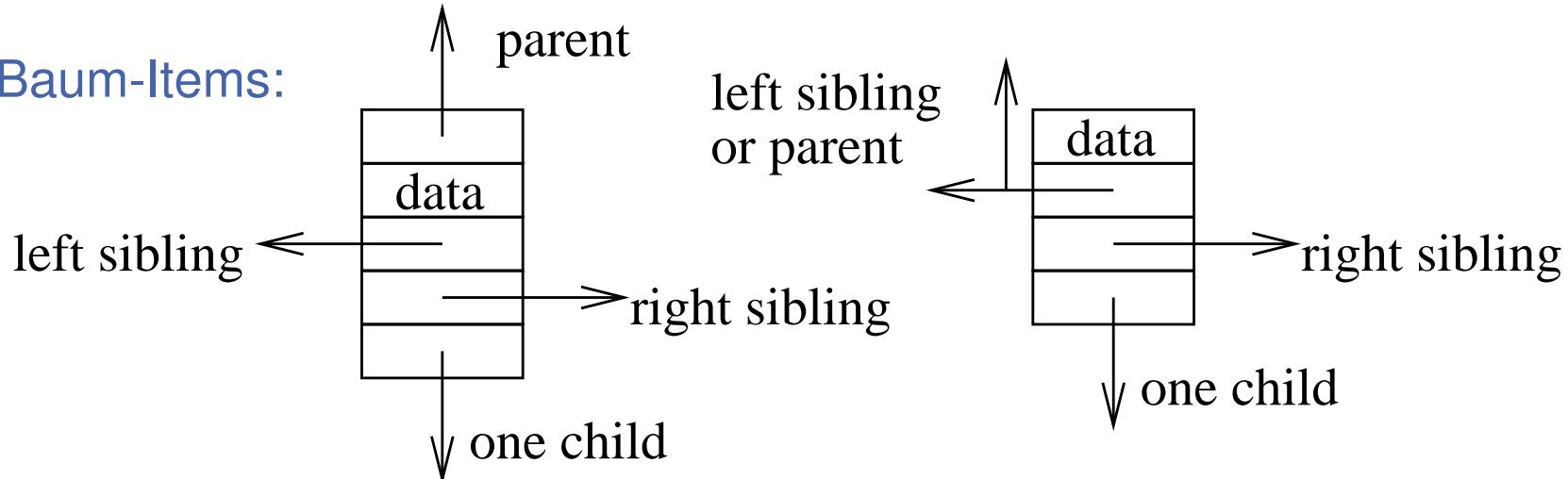
forest:= forest  $\cup$  *o*.forest

*o*.forest:=  $\emptyset$

# Pairing Heaps – Repräsentation

Wurzeln: Doppelt verkettete Liste

Baum-Items:



## Pairing Heaps – Analyse

insert, merge:  $O(1)$

deleteMin, remove:  $O(\log n)$  amortisiert

decreaseKey: unklar!  $O(\log \log n) \leq T \leq O(\log n)$  amortisiert.

In der Praxis sehr schnell.

Beweise: nicht hier.

## Fibonacci Heaps [Fredman Tarjan 1987]

Rang: Anzahl (direkter) Kinder speichern

Vereinigung nach Rang: Union nur für gleichrangige Wurzeln

Markiere Knoten, die ein Kind verloren haben

Kaskadierende Schnitte: Schneide markierte Knoten  
(die also 2 Kinder verloren haben)

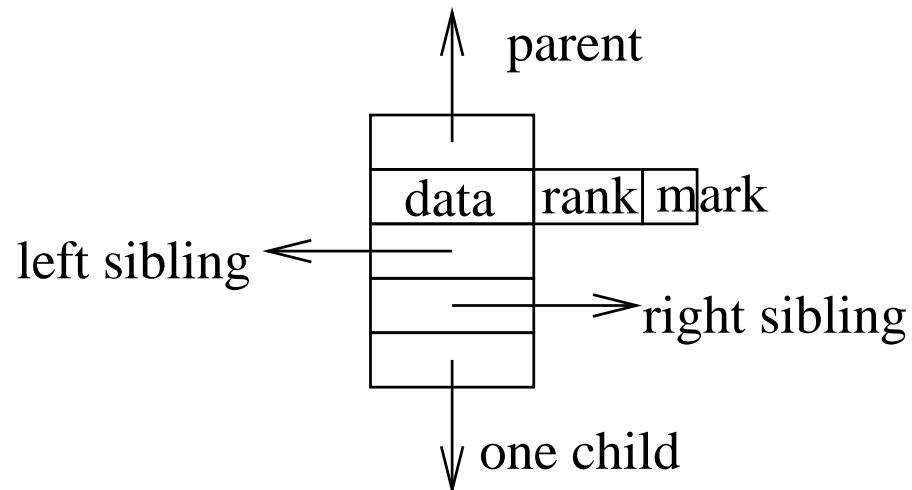
**Satz:** Amortisierte Komplexität  $O(\log n)$  für deleteMin, remove und  
 **$O(1)$**  für alle anderen Operationen  
(d.h.  $Gesamtzeit = O(o + d \log n)$  falls  
 $d = \#\text{deleteMin}$ ,  $o = \#\text{otherOps}$ ,  $n = \max |M|$ )

# Repräsentation

Wurzeln: Doppelt verkettete Liste

(und ein temporäres Feld für deleteMin)

Baum-Items:



**insert, merge:** wie gehabt. Zeit  $O(1)$

# deleteMin mit Union-by-Rank

**Function** deleteMin : Handle

$m := \text{minPtr}$

$\text{forest} := \text{forest} \setminus \{m\}$

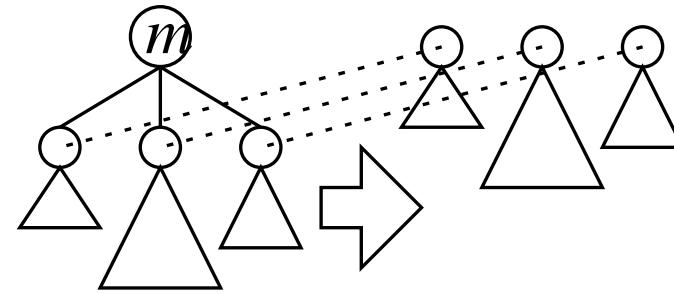
**foreach** child  $h$  of  $m$  **do** newTree( $h$ )

**while**  $\exists a, b \in \text{forest} : \text{rank}(a) = \text{rank}(b)$  **do**

$\text{union}(a, b)$  // increments rank of surviving root

update minPtr

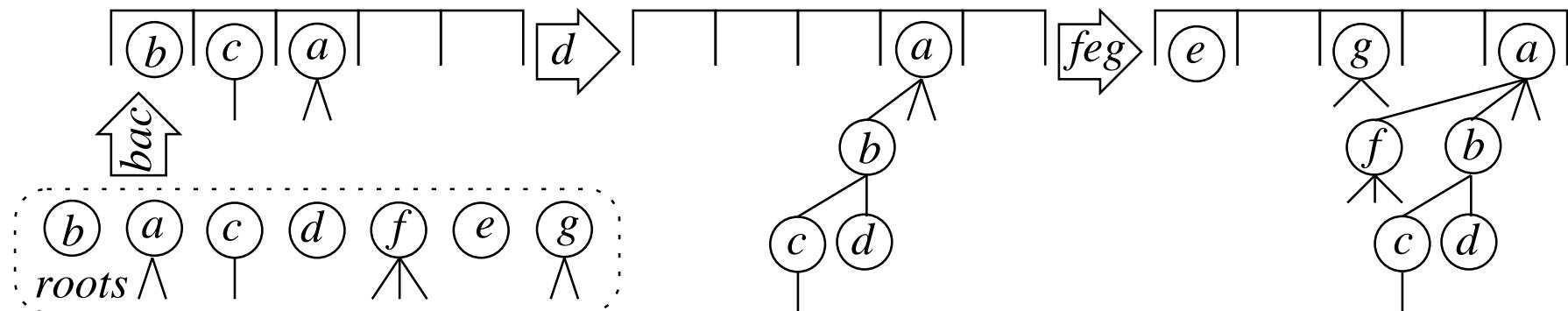
**return**  $m$



# Schnelles Union-by-Rank

Durch rank adressiertes Feld.

Solange link durchführen bis freier Eintrag gefunden.



Analyse: Zeit  $O(\#unions + |\text{forest}|)$

## Amortisierte Analyse von deleteMin

$$\text{maxRank} := \max_{a \in \text{forest}} \text{rank}(a) \text{ (nachher)}$$

Lemma:  $T_{\text{deleteMin}} = O(\text{maxRank})$

Beweis: Kontomethode. Ein Token pro Wurzel

$$\text{rank}(\text{minPtr}) \leq \text{maxRank}$$

~~~ Kosten  $O(\text{maxRank})$  für newTrees und neue Token.

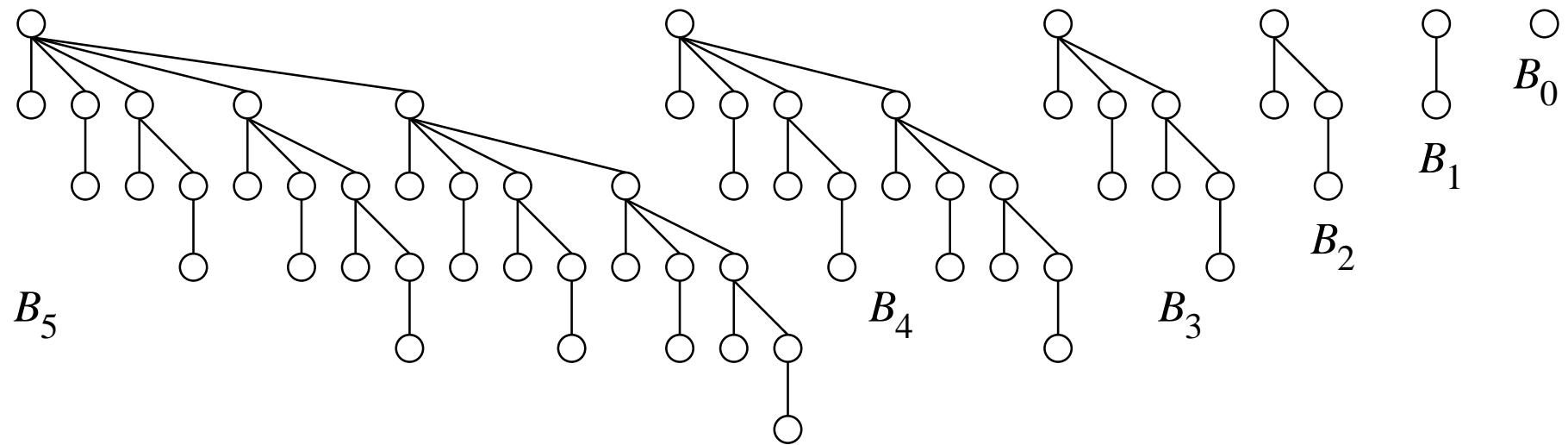
Union-by-rank: Token zählen für

- union Operationen (ein Token wird frei) und
- durchlaufen alter und neuer Wurzeln.

Am Ende gibt es  $\leq \text{maxRank}$  Wurzeln.

# Warum ist maxRank logarithmisch? – Binomialbäume

$2^k + 1 \times \text{insert}, 1 \times \text{deleteMin} \rightsquigarrow \text{rank } k$



[Vuillemain 1978] PQ nur mit Binomialbäumen,  $T_{\text{decreaseKey}} = O(\log n)$ .

Problem: Schnitte können zu kleinen hochrangigen Bäumen führen

# Kaskadierende Schnitte

**Procedure** decreaseKey( $h$  : Handle,  $k$  : Key)

key( $h$ ) :=  $k$

cascadingCut( $h$ )

**Procedure** cascadingCut( $h$ )

**if**  $h$  is not a root **then**

$p$  := parent( $h$ )

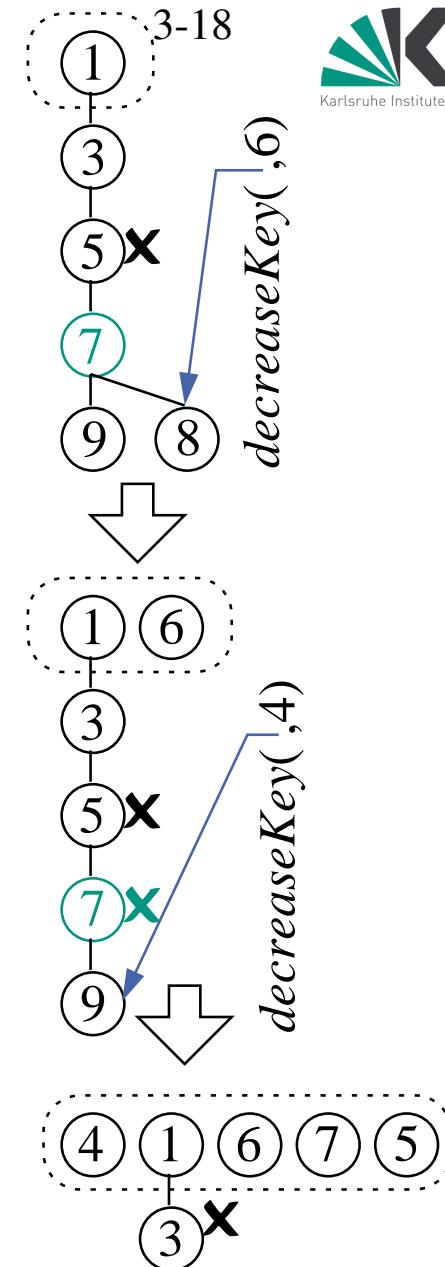
unmark  $h$

cut( $h$ )

**if**  $p$  is marked **then**

cascadingCut( $p$ )

**else** mark  $p$



Wir werden zeigen: kaskadierende Schnitte halten maxRank logarithmisch

Lemma: decreaseKey hat amortisierte Komplexität  $O(1)$

Kontomethode: ( $\approx 1$  Token pro cut oder union)

1 Token für jede Wurzel

2 Token für jeden markierten Knoten

betrachte decreaseKey mit  $k$  konsekutiven markierten Vorgängern:

2k Token werden frei (unmarked nodes)

2 Token für neue Markierung

$k+1$  Token für Ausstattung der neuen Wurzeln

$k+1$  Token für Schnitte

Bleiben 4 Token +  $O(1)$  Kosten für decreaseKey

# Auftritt Herr Fibonacci

$$F_i := \begin{cases} 0 & \text{für } i=0 \\ 1 & \text{für } i=1 \\ F_{i-2} + F_{i-1} & \text{sonst} \end{cases}$$

Bekannt:  $F_{i+1} \geq ((1 + \sqrt{5})/2)^i \geq 1.618^i$  for all  $i \geq 0$ .

Wir zeigen:

Ein Teilbaum mit Wurzel  $v$  mit  $\text{rank}(v) = i$  enthält  $\geq F_{i+2}$  Elemente.

⇒

logarithmische Zeit für deleteMin.

## Beweis:

Betrachte Zeitpunkt als das  $j$ -te Kind  $w_j$  von  $v$  hinzugelinkt wurde:

$w_j$  und  $v$  hatten gleichen Rang  $\geq j - 1$  ( $v$  hatte schon  $j - 1$  Kinder)

$\text{rank}(w_j)$  hat **höchsten um eins abgenommen** (cascading cuts)

$\Rightarrow \text{rank}(w_j) \geq j - 2$  und  $\text{rank}(v) \geq j - 1$

$S_i :=$  untere Schranke für # Knoten mit Wurzel vom Rang  $i$ :

$$S_0 = 1$$

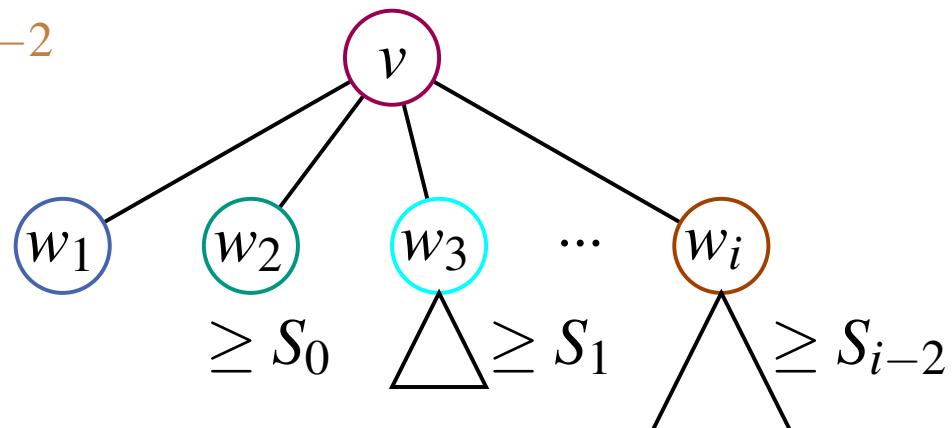
$$S_1 = 2$$

$$S_i \geq 1 + 1 + S_0 + S_1 + \cdots + S_{i-2}$$

für  $i \geq 2$

Diese Rekurrenz

hat die Lösung  $S_i \geq F_{i+2}$



## Addressable Priority Queues: Mehr

- Untere Schranke  $\Omega(\log n)$  für deleteMin, vergleichsbasiert.  
Beweis: Übung
- Worst case Schranken: nicht hier
- Monotone PQs mit **ganzzahligen** Schlüsseln (stay tuned)

### Offene Probleme:

Analyse Pairing Heap, Vereinfachung Fibonacci Heap.

## Zusammenfassung Datenstrukturen

- In dieser Vorlesung Fokus auf Beispiel Prioritätslisten  
(siehe auch kürzeste Wege, externe Algorithmen)
- Heapkonzept trägt weit
- Geschwisterzeiger erlauben Repräsentation beliebiger Bäume mit konstanter Zahl Zeiger pro Item.
- Fibonacci heaps als nichttriviales Beispiel für amortisierte Analyse

# 4 Anwendungen von DFS

# Tiefensuchschema für $G = (V, E)$

unmark all nodes; init

**foreach**  $s \in V$  **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( $u, v : \text{NodId}$ )

            // Explore  $v$  coming from  $u$ .

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

# DFS Nummerierung

init:  $\text{dfsPos} = 1 : 1..n$

$\text{root}(s)$ :  $\text{dfsNum}[s] := \text{dfsPos}++$

$\text{traverseTreeEdge}(v, w)$ :  $\text{dfsNum}[w] := \text{dfsPos}++$

$$u \prec v \Leftrightarrow \text{dfsNum}[u] < \text{dfsNum}[v] .$$

## Beobachtung:

Knoten auf dem Rekursionsstapel sind bzgl.,  $\prec$  sortiert

# Fertigstellungszeit

init:                   finishingTime=1 : 1..n

backtrack( $u, v$ ):    **finishTime**[ $v$ ]:= finishingTime++

# Starke Zusammenhangskomponenten

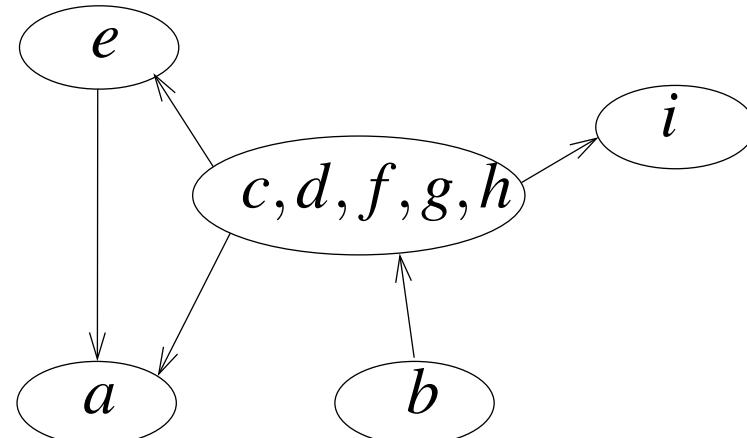
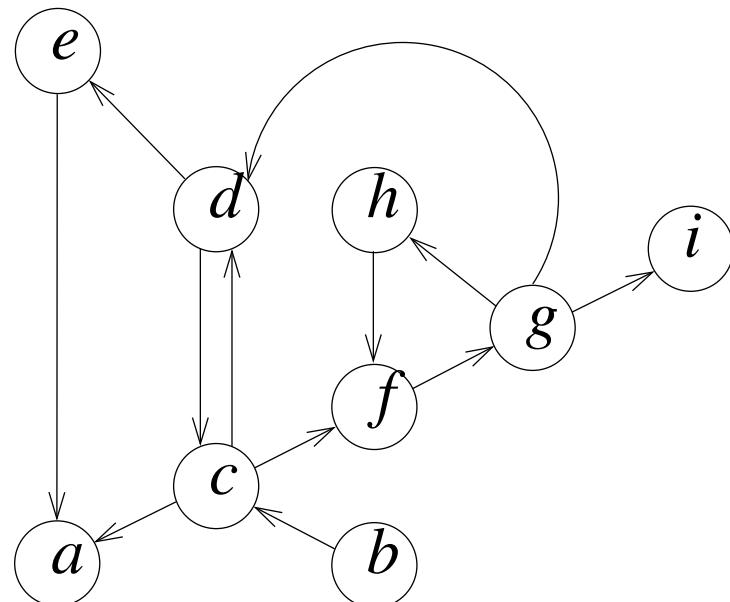
Betrachte die Relation  $\overset{*}{\leftrightarrow}$  mit

$u \overset{*}{\leftrightarrow} v$  falls  $\exists$  Pfad  $\langle u, \dots, v \rangle$  und  $\exists$  Pfad  $\langle v, \dots, u \rangle$ .

**Beobachtung:**  $\overset{*}{\leftrightarrow}$  ist Äquivalenzrelation

Übung

Die Äquivalenzklassen von  $\overset{*}{\leftrightarrow}$  bezeichnet man als starke Zusammenhangskomponenten.



# Starke Zusammenhangskomponenten – Abstrakter Algorithmus

$G_c := (V, \emptyset = E_c)$

**foreach** edge  $e \in E$  **do**

**invariant** SCCs of  $G_c$  are known

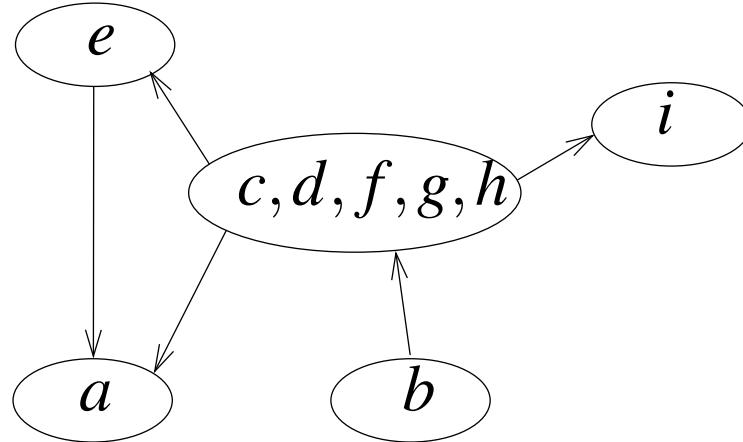
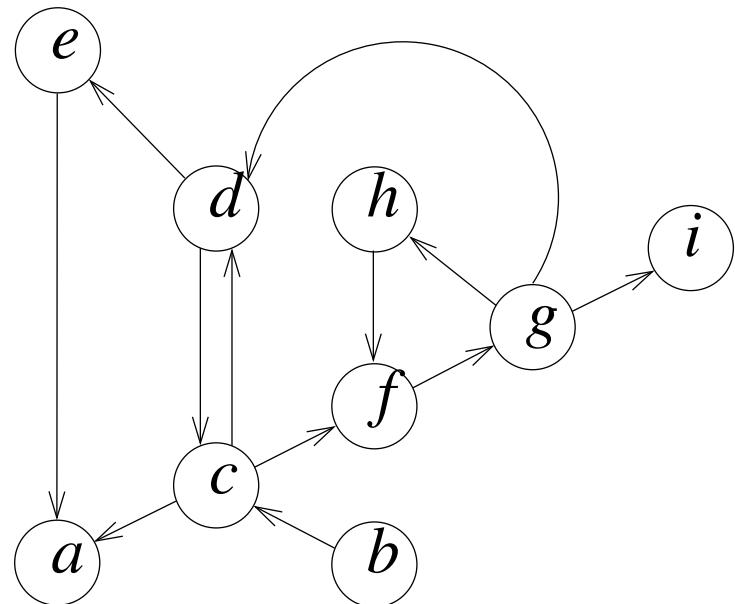
$E_c := E_c \cup \{e\}$

# Schrumpfgraph

$$G_c^s = (V^s, E_c^s)$$

Knoten: SCCs von  $G_c$ .

Kanten:  $(C, D) \in E_c^s \Leftrightarrow \exists (c, d) \in E_c : c \in C \wedge d \in D$



**Beobachtung:** Der Schrumpfgraph ist azyklisch

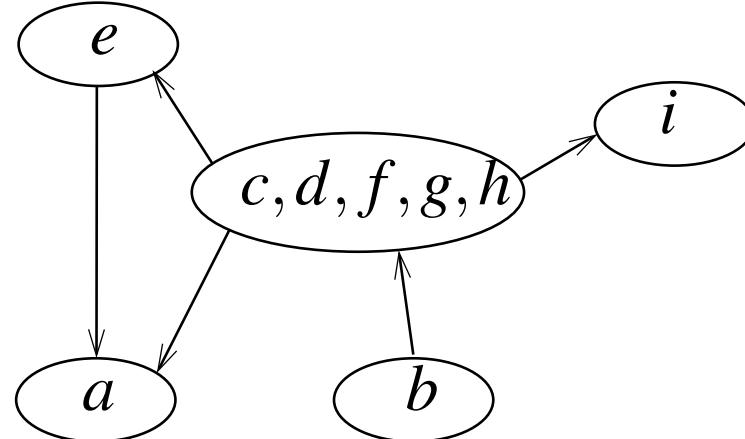
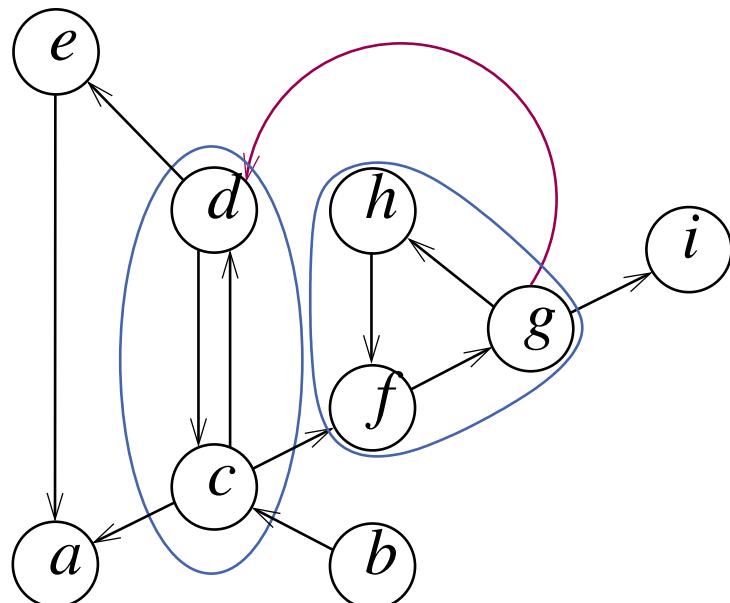
# Auswirkungen einer neuen Kante $e$ auf $G_c$ , $G_c^s$

SCC-intern: Nichts ändert sich

zwischen zwei SCCs:

Kein Kreis: Neue Kante in  $G_c^s$

Kreisschluss: SCCs auf Kreis kollabieren.



## Konkreter: SCCs mittels DFS

[Cherian/Mehlhorn 96, Gabow 2000]

$V_c$  = markierte Knoten

$E_c$  = bisher explorierte Kanten

**Aktive Knoten:** markiert aber nicht finished.

SCCs von  $G_c$ :

nicht erreicht: Unmarkierte Knoten

offen: enthält aktive Knoten

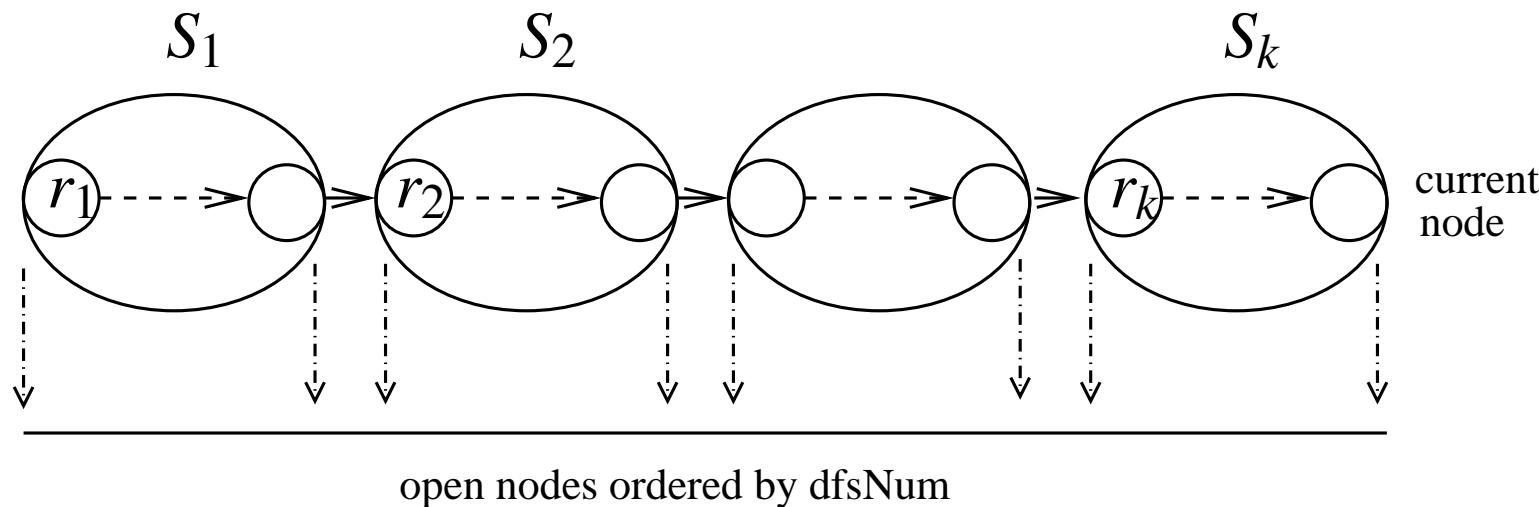
abgeschlossen: alle Knoten finished

**component[w]** gibt Repräsentanten einer SCC an.

Knoten von offenen (abgeschl.) Komponenten heißen offen (abgeschl.)

# Invarianten von $G_c$

1. Kanten von abgeschlossenen Knoten gehen zu abgeschlossenen Knoten
2. Offene Komponenten  $S_1, \dots, S_k$  bilden Pfad in  $G_c^s$ .
3. Repräsentanten partitionieren die offenen Komponenten bzgl. ihrer `dfsNum`.



**Lemma:** Abgeschlossene SCCs von  $G_c$  sind SCCs von  $G$

Betrachte abgeschlossenen Knoten  $v$

und beliebigen Knoten  $w$

in der SCC von  $v$  bzgl.  $G$ .

z.Z.:  $w$  ist abgeschlossen und

in der gleichen SCC von  $G_c$  wie  $v$ .

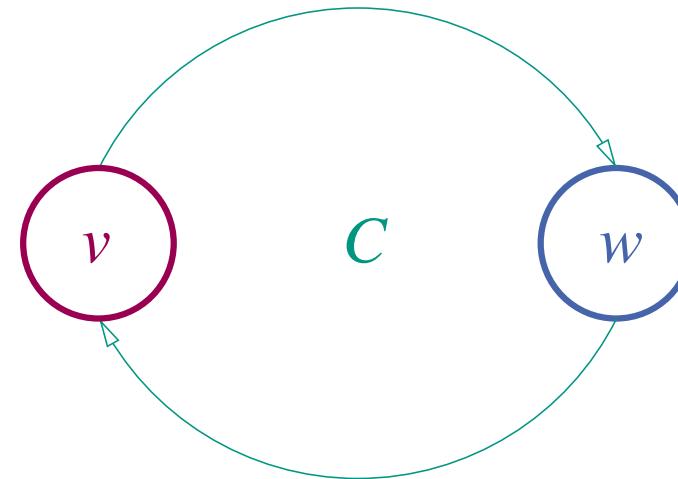
Betrachte Kreis  $C$  durch  $v, w$ .

Inv. 1: **Knoten** von  $C$  sind abgeschlossen.

Abgeschl. Knoten sind finished.

Kanten aus finished Knoten wurden exploriert.

Also sind alle **Kanten** von  $C$  in  $G_c$ . □

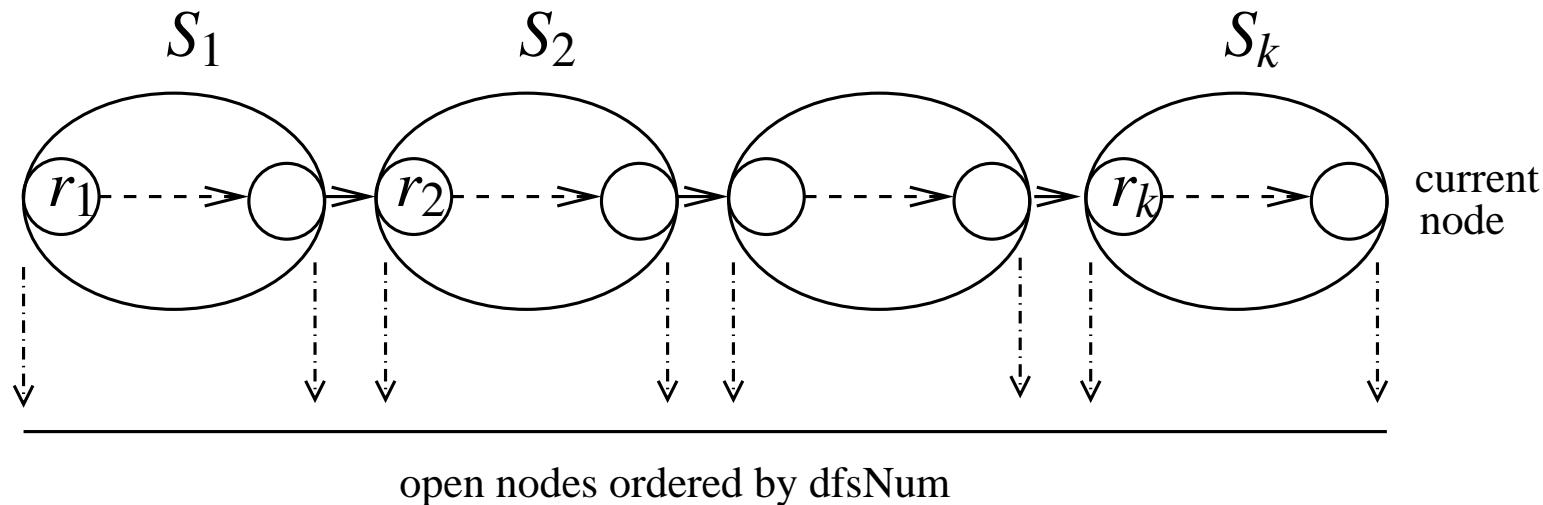


# Repräsentation offener Komponenten

Zwei Stapel aufsteigend sortiert nach dfsNum

oReps: Repräsentanten offener Komponenten

oNodes: Alle offenen Knoten



init

```
component : NodeArray of NodId           // SCC representatives  
oReps=⟨⟩ : Stack of NodId      // representatives of open SCCs  
oNodes=⟨⟩ : Stack of NodId      // all nodes in open SCCs
```

Alle Invarianten erfüllt.

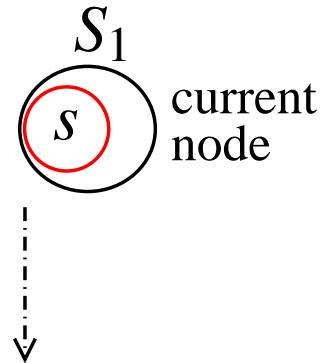
(Weder offene noch geschlossene Knoten)

$\text{root}(s)$

```
oReps.push(s)                                // new open  
oNodes.push(s)                               // component
```

$\{s\}$  ist die einzige offene Komponente.

Alle Invarianten bleiben gültig



---

open nodes ordered by dfsNum

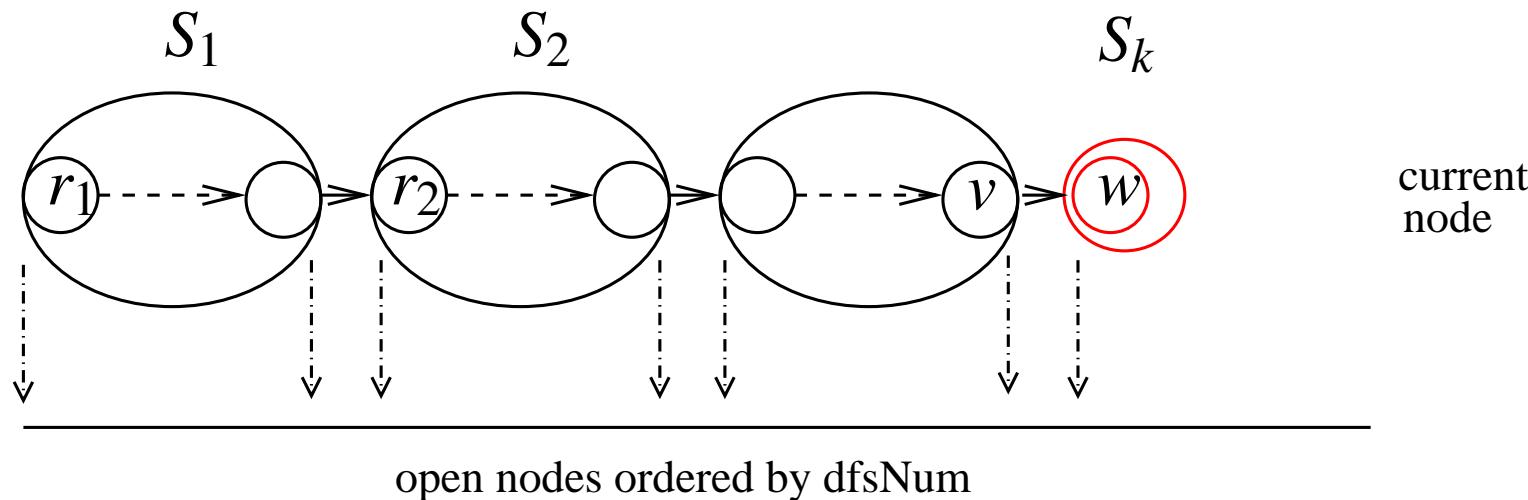
traverseTreeEdge( $v, w$ )

oReps.push( $w$ ) // new open  
oNodes.push( $w$ ) // component

$\{w\}$  ist neue offene Komponente.

$\text{dfsNum}(w) >$  alle anderen.

~~~Alle Invarianten bleiben gültig



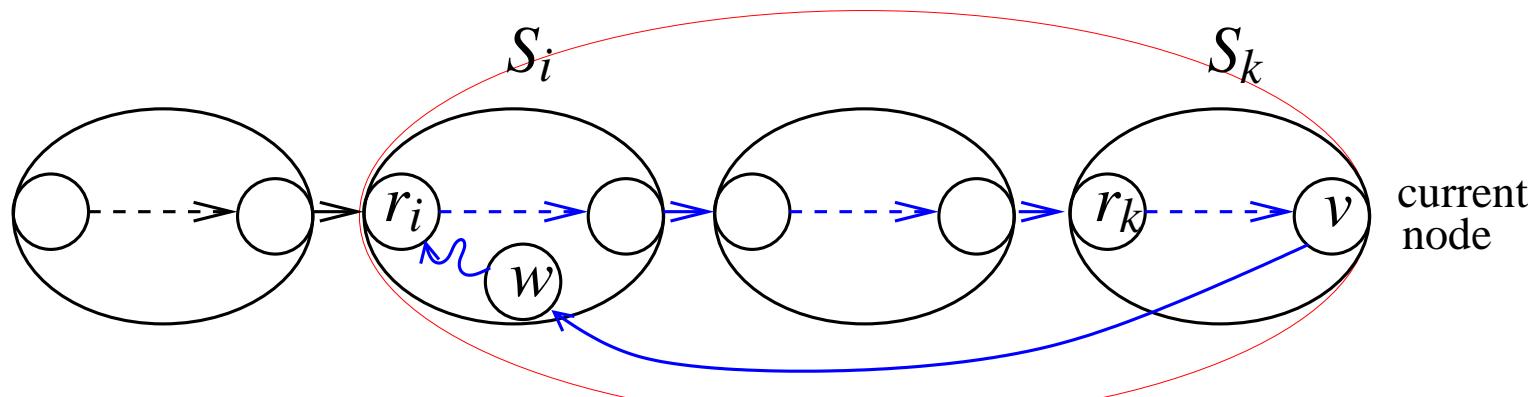
`traverseNonTreeEdge( $v, w$ )`

**if**  $w \in \text{oNodes}$  **then**

**while**  $w \prec \text{oReps.top}$  **do**  $\text{oReps.pop}$

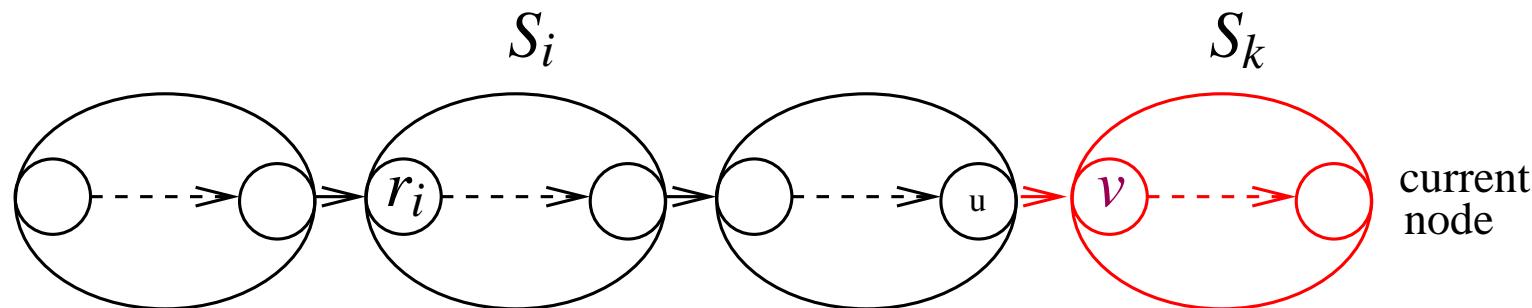
$w \notin \text{oNodes} \rightsquigarrow w$  is abgeschlossen  $\stackrel{\text{Lemma}(*)}{\rightsquigarrow}$  Kante uninteressant

$w \in \text{oNodes}$ : kollabiere offene SCCs auf Kreis



backtrack( $u, v$ )

```
if  $v = \text{oReps.top}$  then
     $\text{oReps.pop}$  // close
repeat // component
     $w := \text{oNodes.pop}$ 
     $\text{component}[w] := v$ 
until  $w = v$ 
```



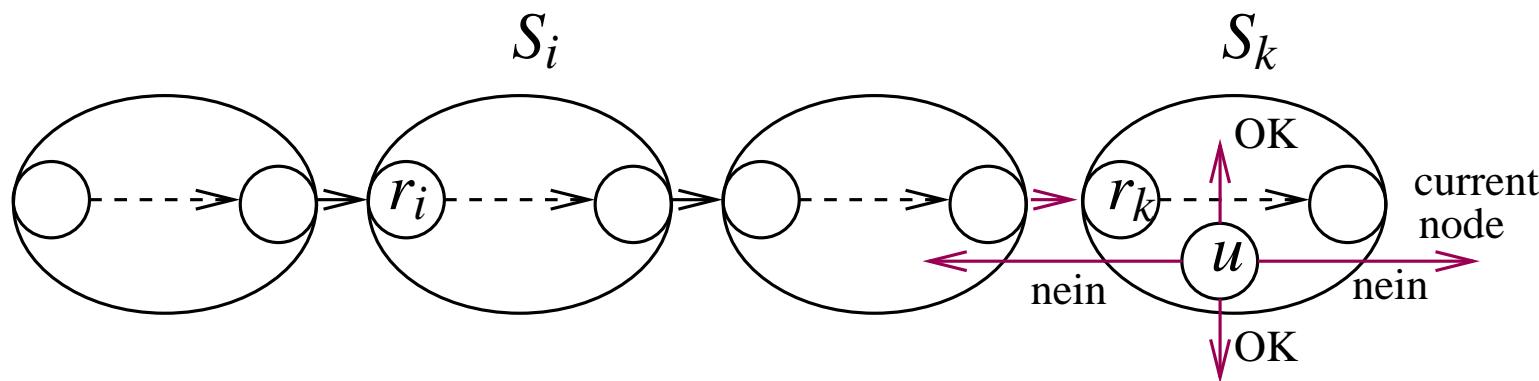
z.Z. Invarianten bleiben erhalten...

backtrack( $u, v$ )

```

if  $v = \text{oReps.top}$  then
     $\text{oReps.pop}$                                 // close
    repeat                                     // component
         $w := \text{oNodes.pop}$ 
         $\text{component}[w] := v$ 
    until  $w = v$ 
```

**Inv. 1:** Kanten von abgeschlossenen Knoten gehen zu abgeschlossenen Knoten.



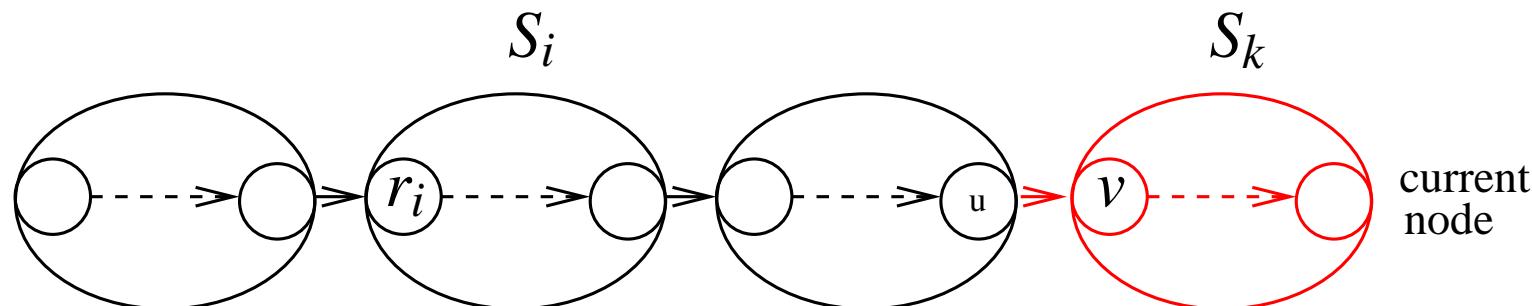
backtrack( $u, v$ )

```

if  $v = \text{oReps.top}$  then
     $\text{oReps.pop}$                                 // close
    repeat                                     // component
         $w := \text{oNodes.pop}$ 
         $\text{component}[w] := v$ 
    until  $w = v$ 
```

**Inv. 2:** Offene Komponenten  $S_1, \dots, S_k$  bilden Pfad in  $G_c^s$

OK. ( $S_k$  wird ggf. entfernt)

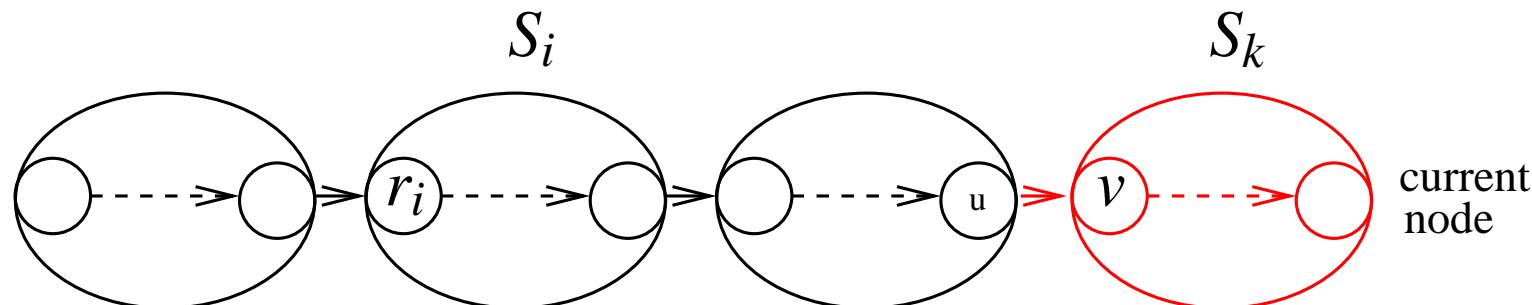


backtrack( $u, v$ )

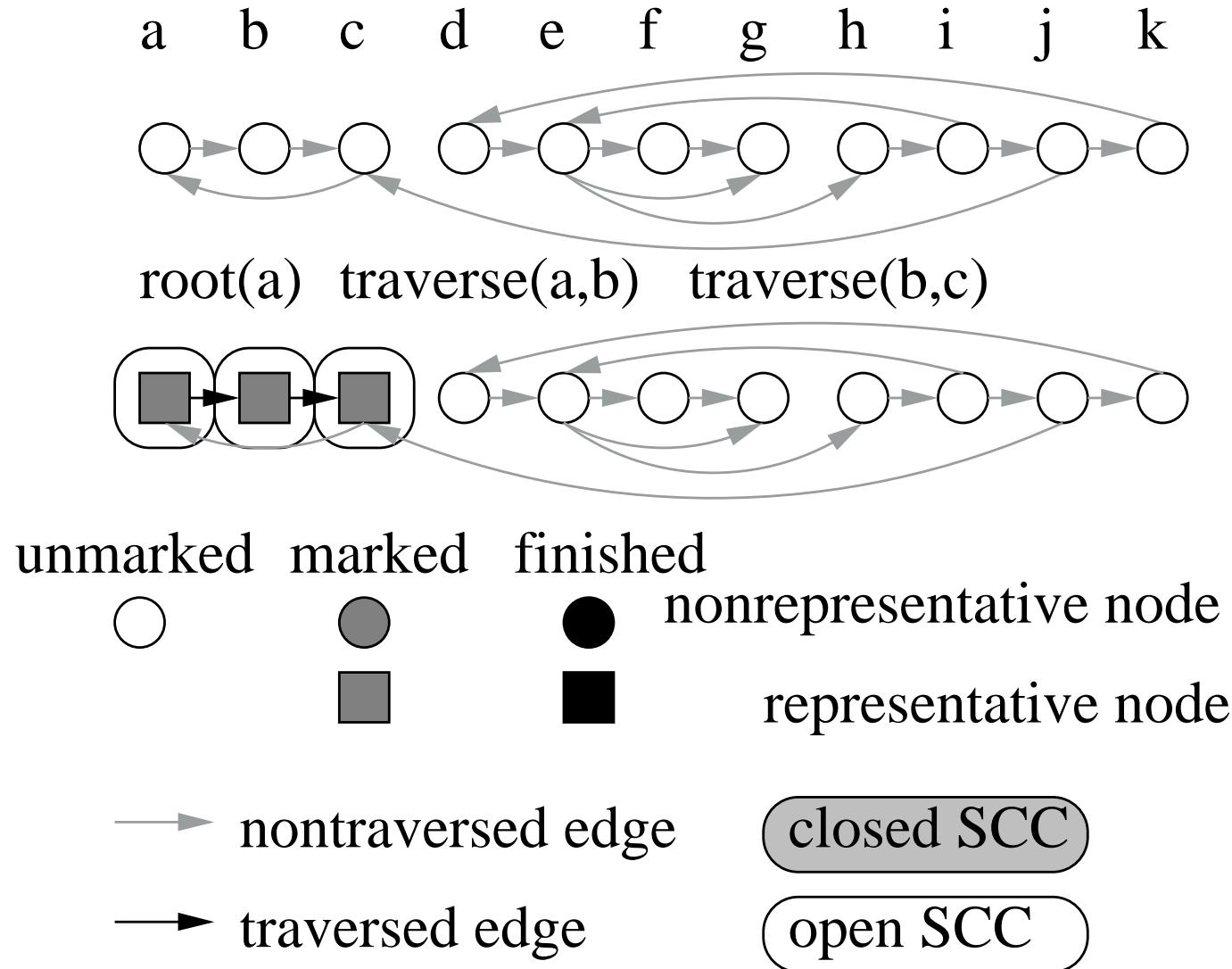
```
if  $v = \text{oReps.top}$  then
     $\text{oReps.pop}$                                 // close
repeat                                         // component
     $w := \text{oNodes.pop}$ 
     $\text{component}[w] := v$ 
until  $w = v$ 
```

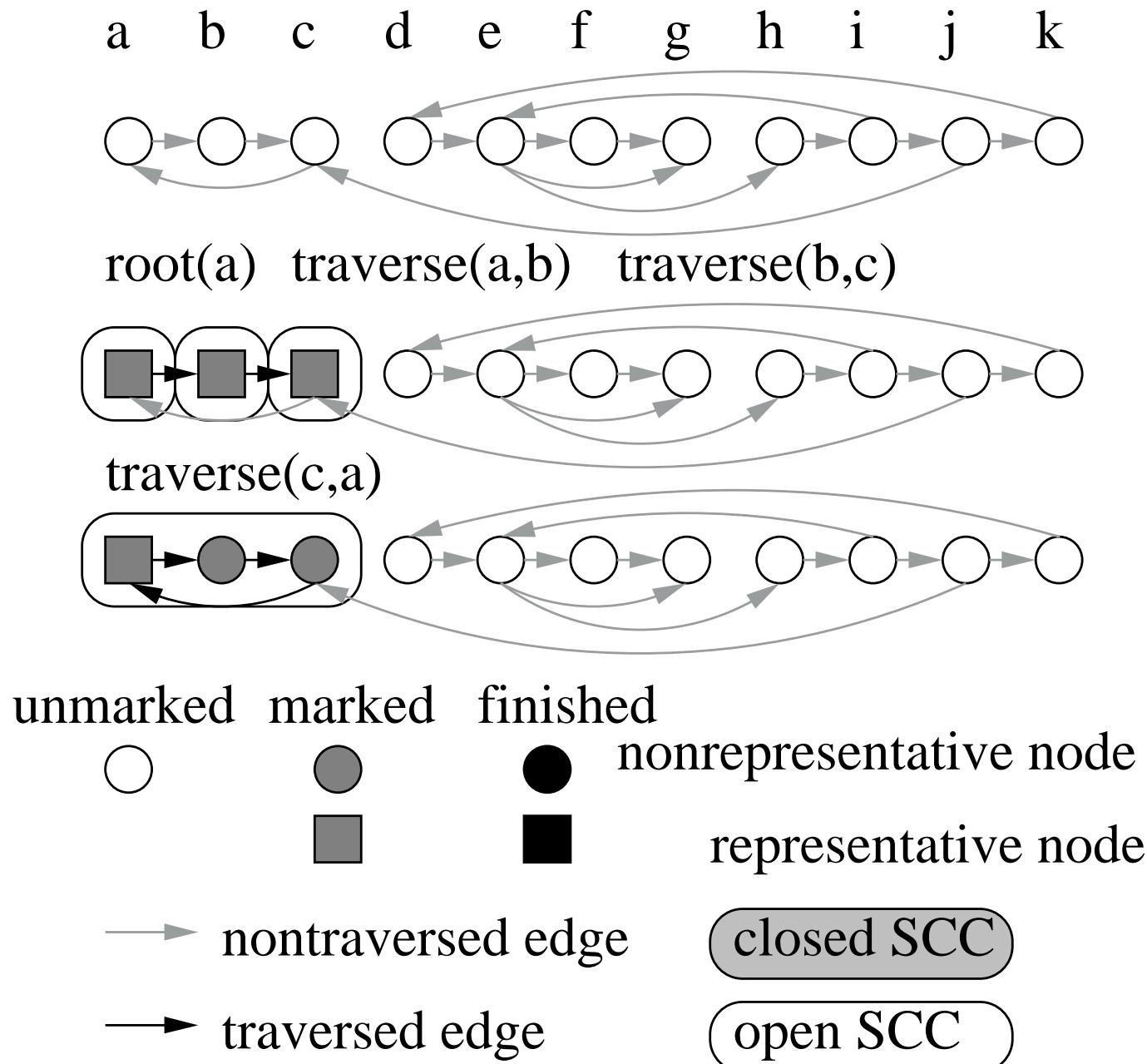
**Inv. 3:** Repräsentanten partitionieren die offenen Komponenten bzgl.  
ihrer dfsNum.

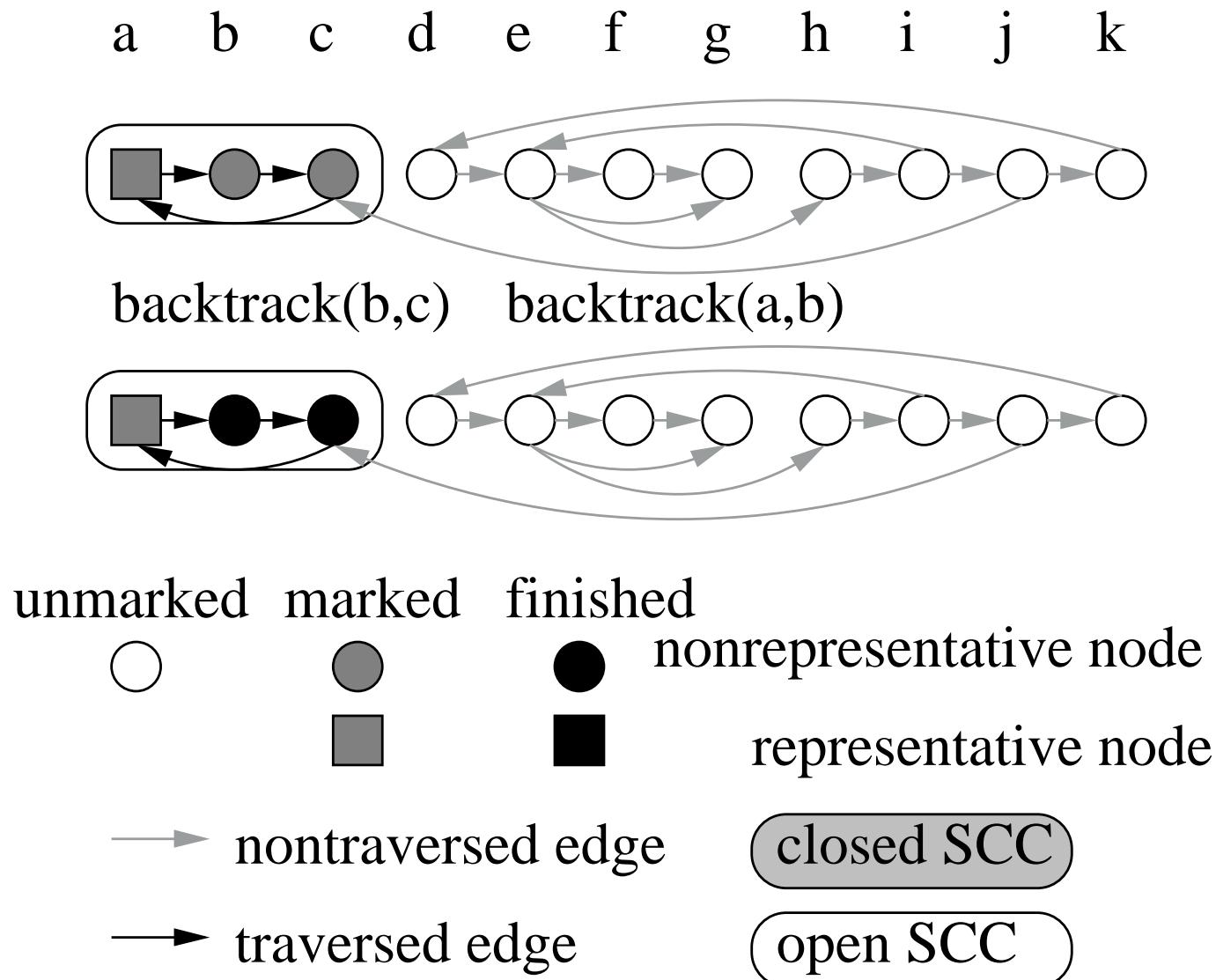
OK. ( $S_k$  wird ggf. entfernt)

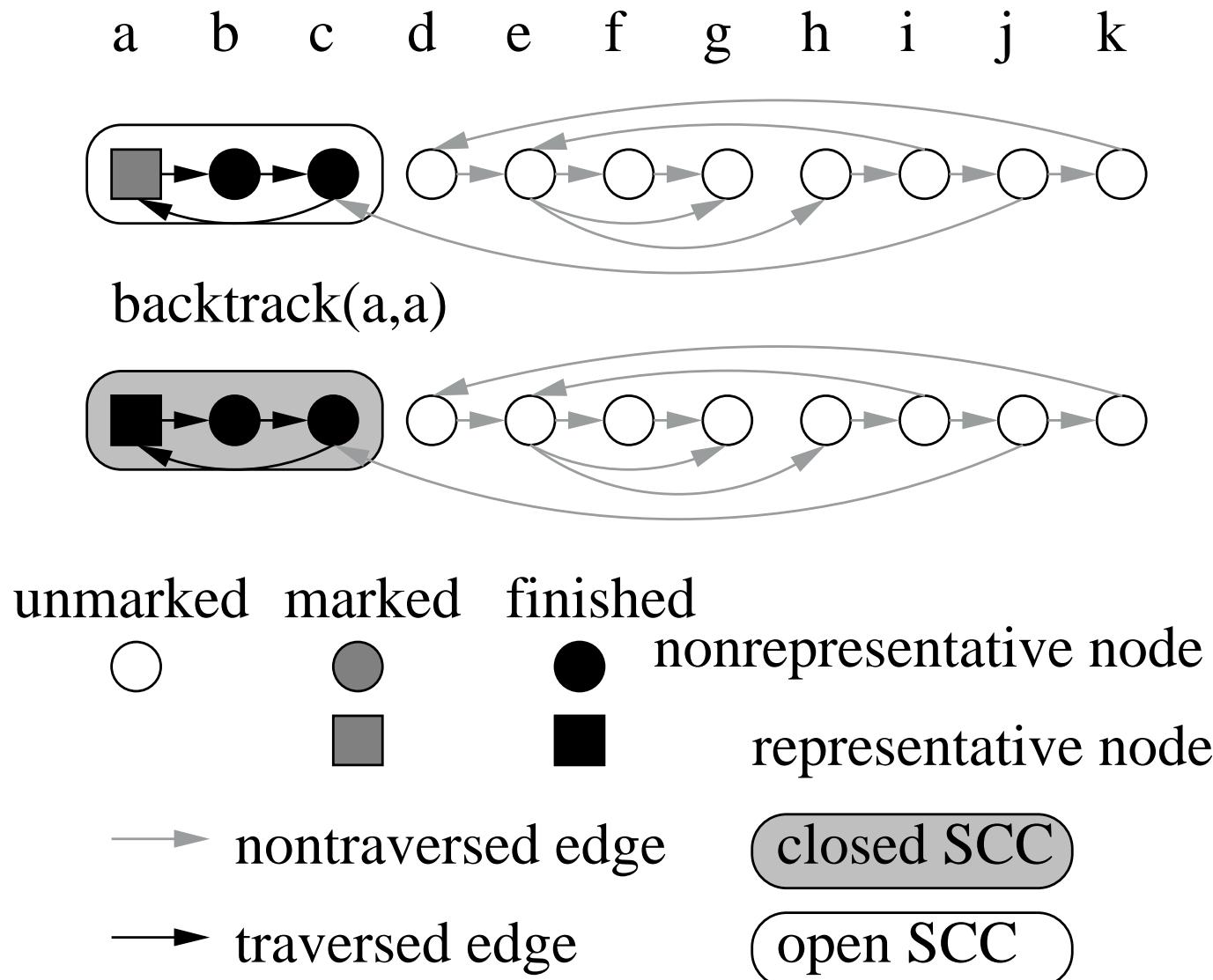


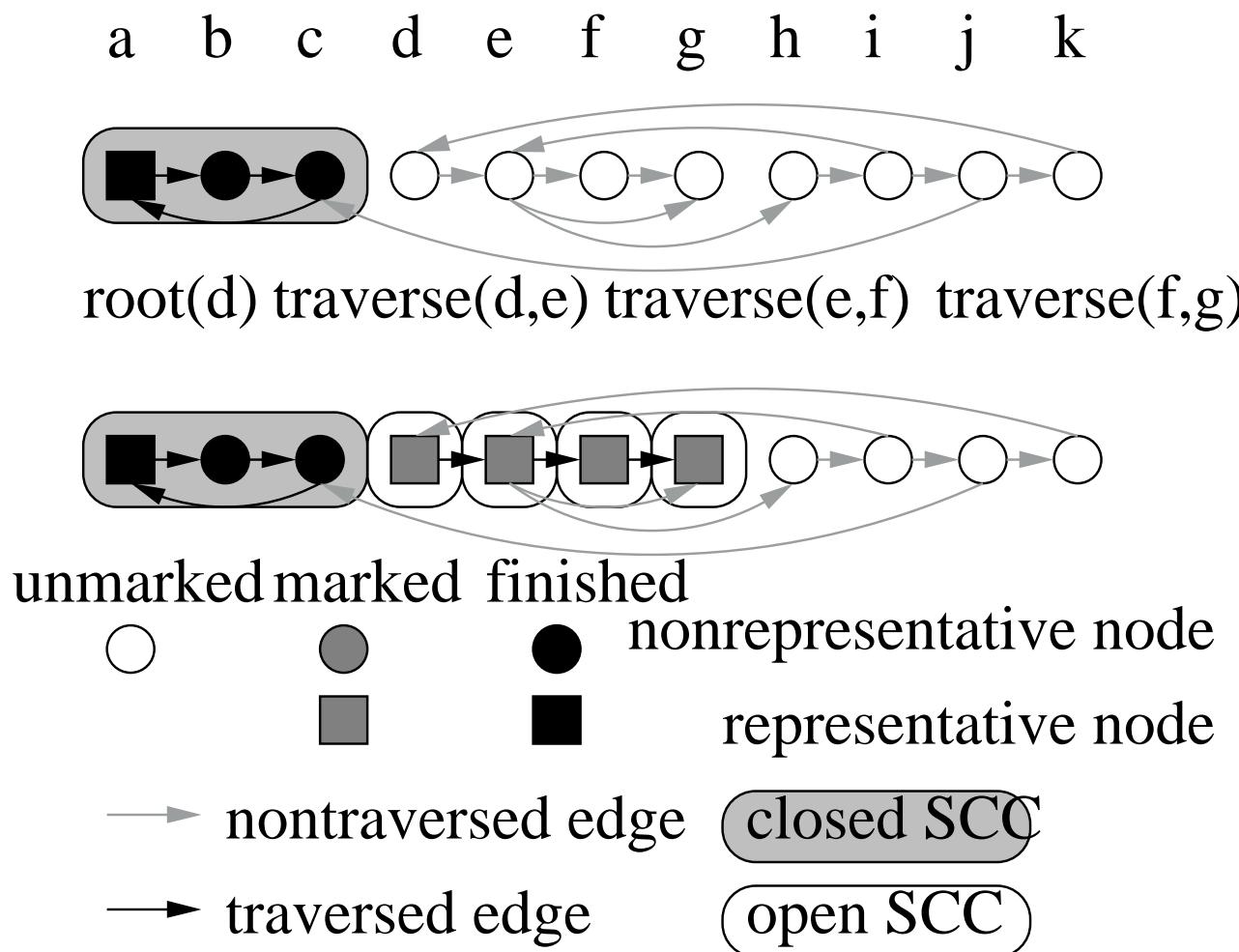
# Beispiel

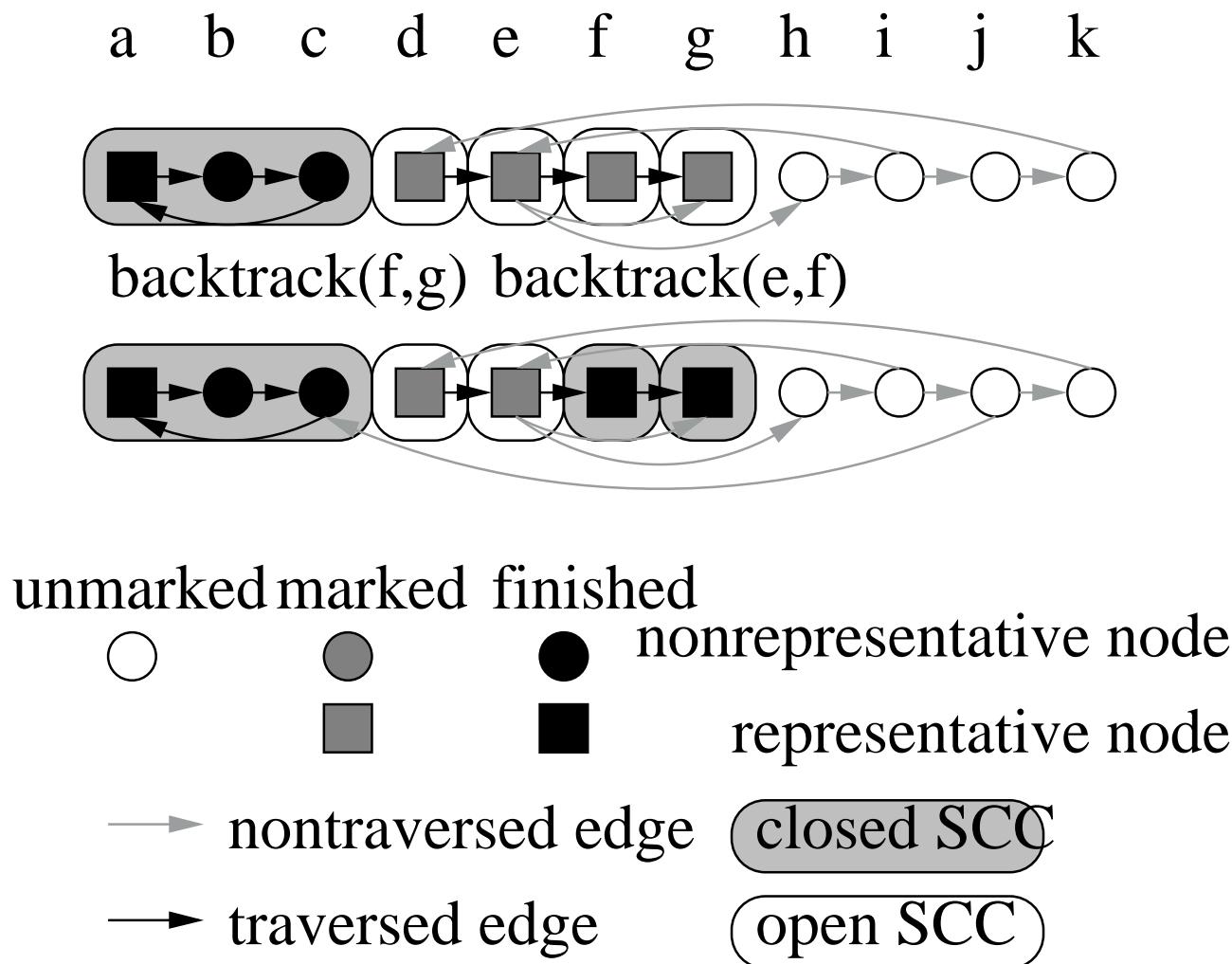


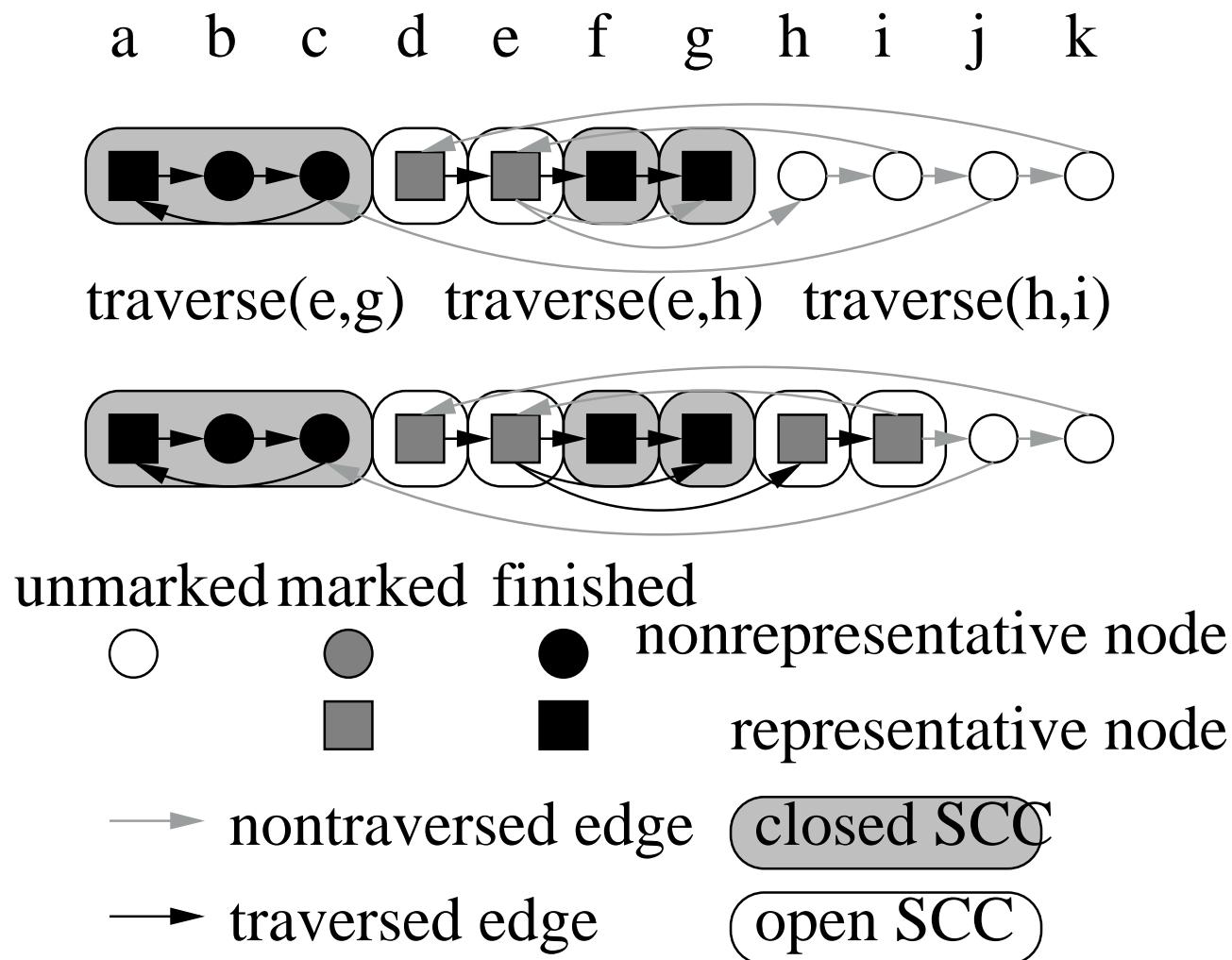


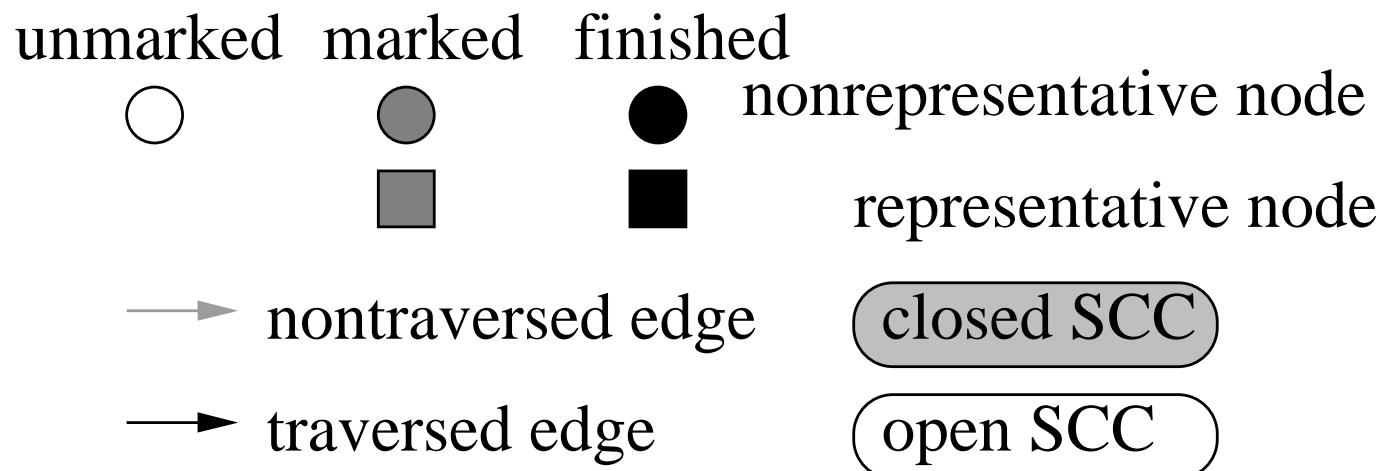
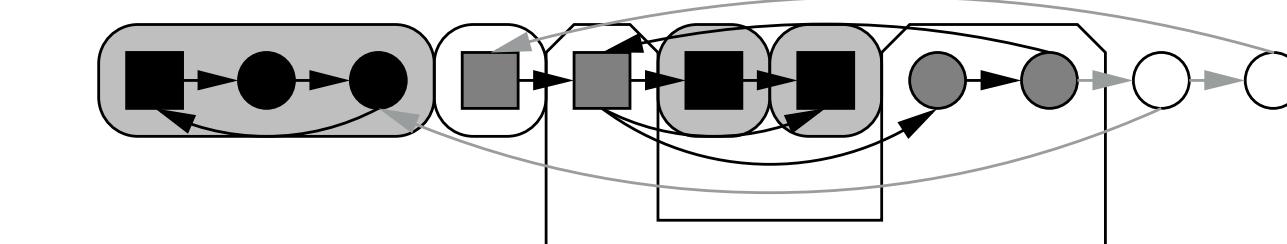
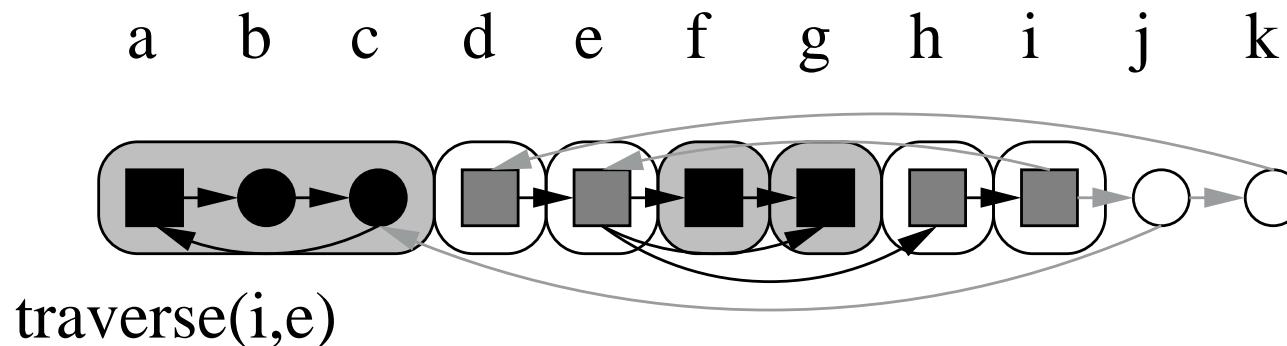


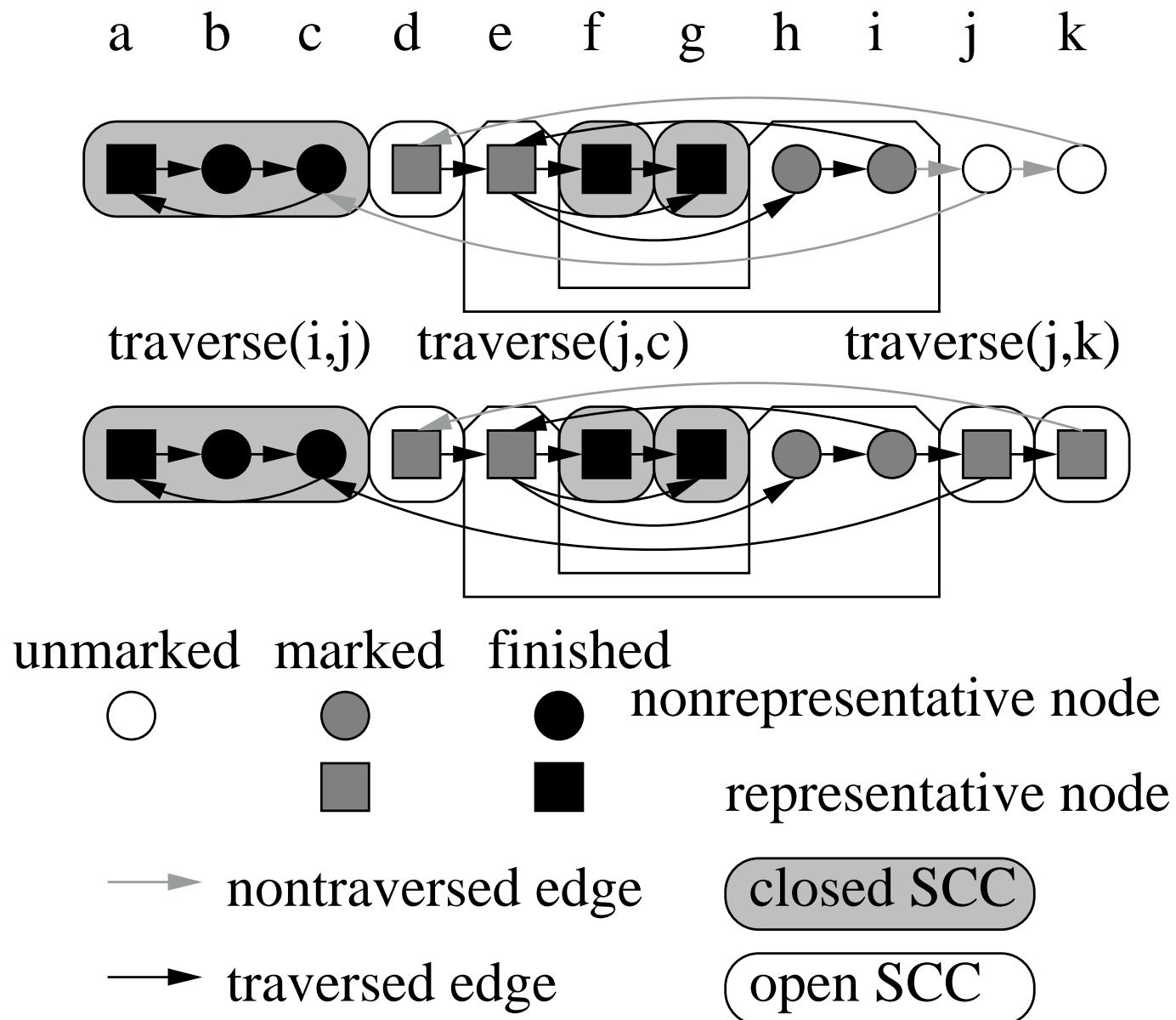


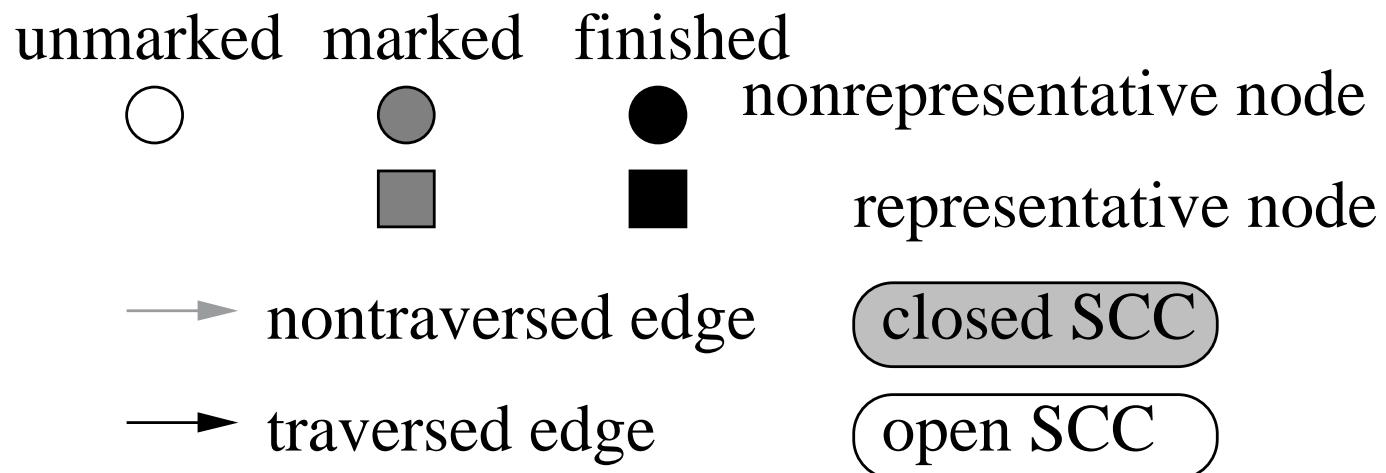
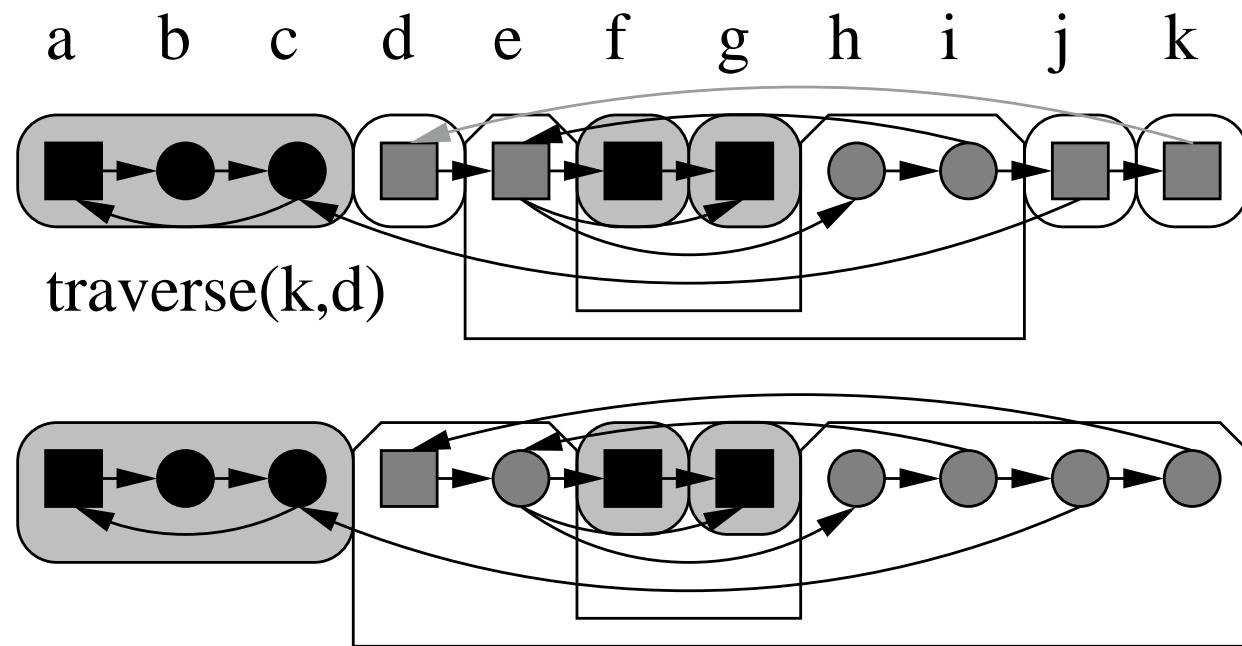


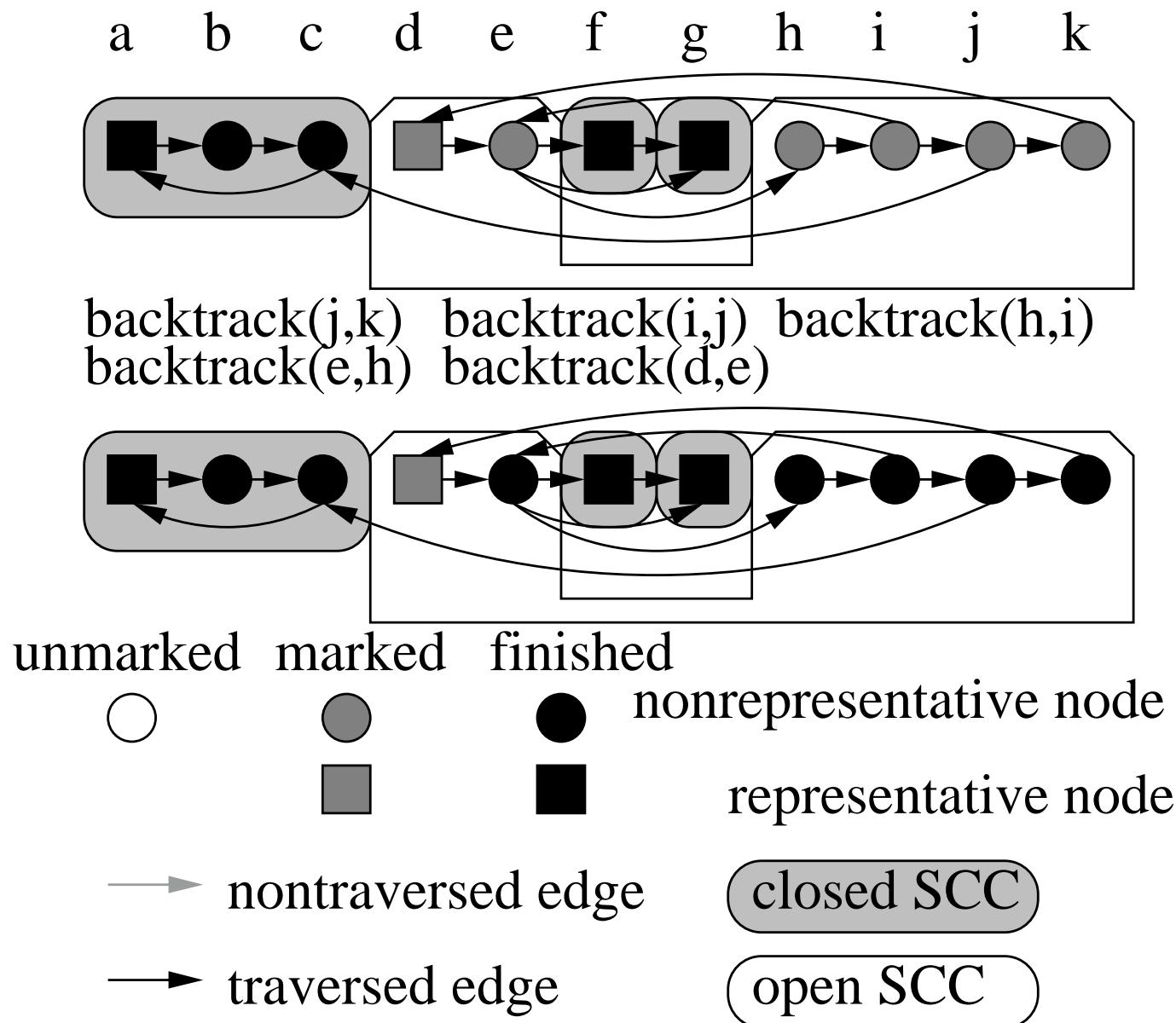


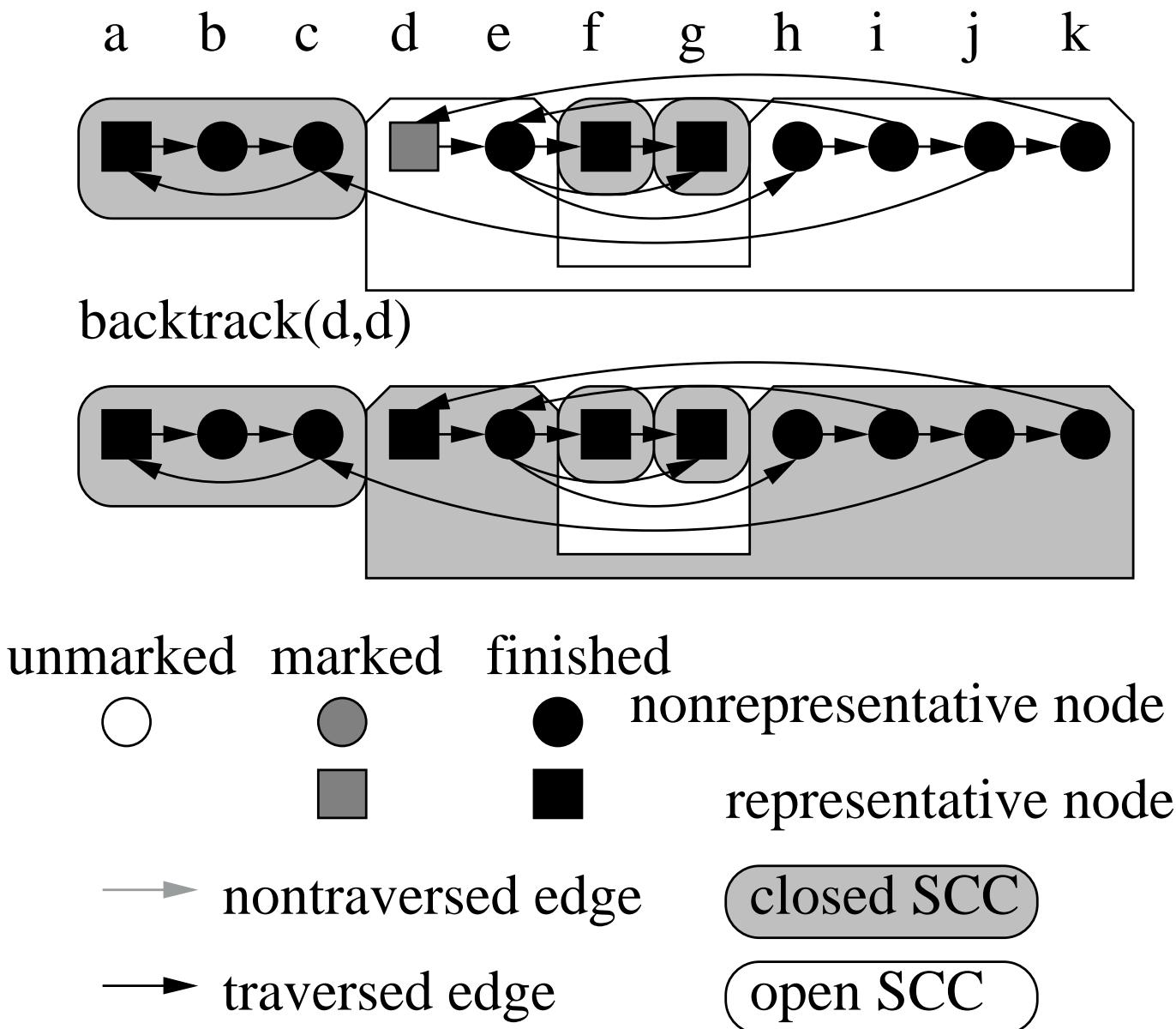












## Zusammenfassung: SCC Berechnung

- Einfache Instantiierung des DFS-Musters
- Nichttrivialer Korrektheitsbeweis
- Laufzeit  $O(m + n)$ : (Jeweils max.  $n$  push/pop Operationen)
- Ein einziger Durchlauf

Implementierungsdetails:

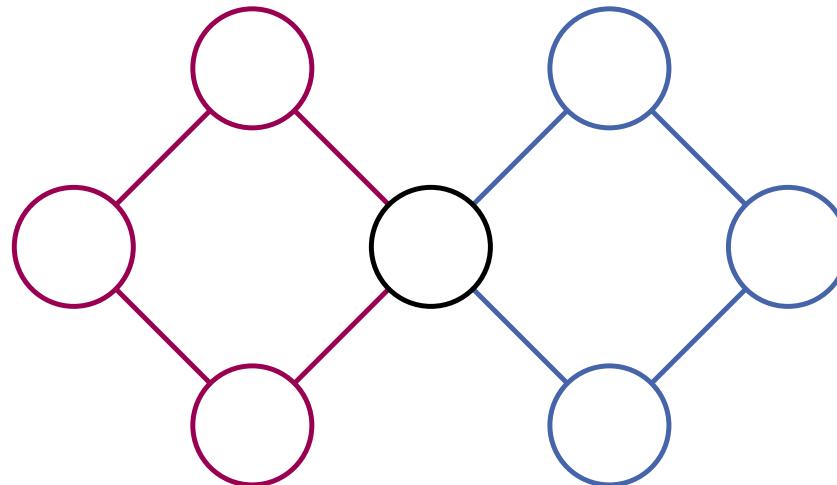
Mehlhorn, Näher, Sanders

Engineering DFS-Based Graph Algorithms

[arxiv.org/abs/1703.10023](https://arxiv.org/abs/1703.10023)

## 2-zusammenhängende Komponenten (ungerichtet)

Bei entfernen eines Knotens bleibt die Komponente zusammenhängend.  
(Partitionierung der Kanten)



Geht in Zeit  $O(m + n)$  mit Algorithmus ähnlich zu SCC-Algorithmus

## Mehr DFS-basierte Linearzeitalgorithmen

- 3-zusammenhängende Komponenten
- Planaritätstest
- Einbettung planarer Graphen

# 5 Maximum Flows and Matchings

[mit Kurt Mehlhorn, Rob van Stee]

Folien auf Englisch

Literatur:

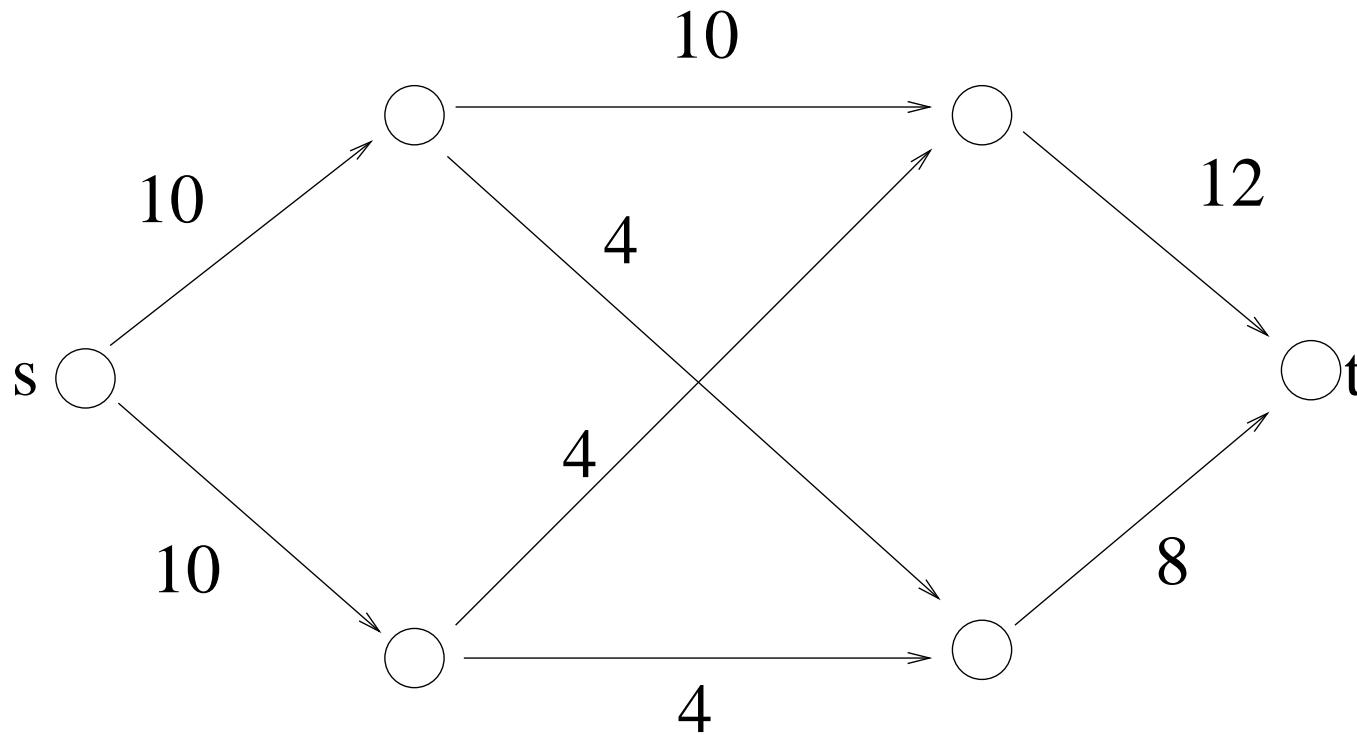
[Mehlhorn / Näher, The LEDA Platform of Combinatorial and Geometric Computing, Cambridge University Press, 1999]

[http://www.mpi-inf.mpg.de/~mehlhorn/ftp/LEDAbook/Graph\\_alg.ps](http://www.mpi-inf.mpg.de/~mehlhorn/ftp/LEDAbook/Graph_alg.ps)

[Ahuja, Magnanti, Orlin, Network Flows, Prentice Hall, 1993]

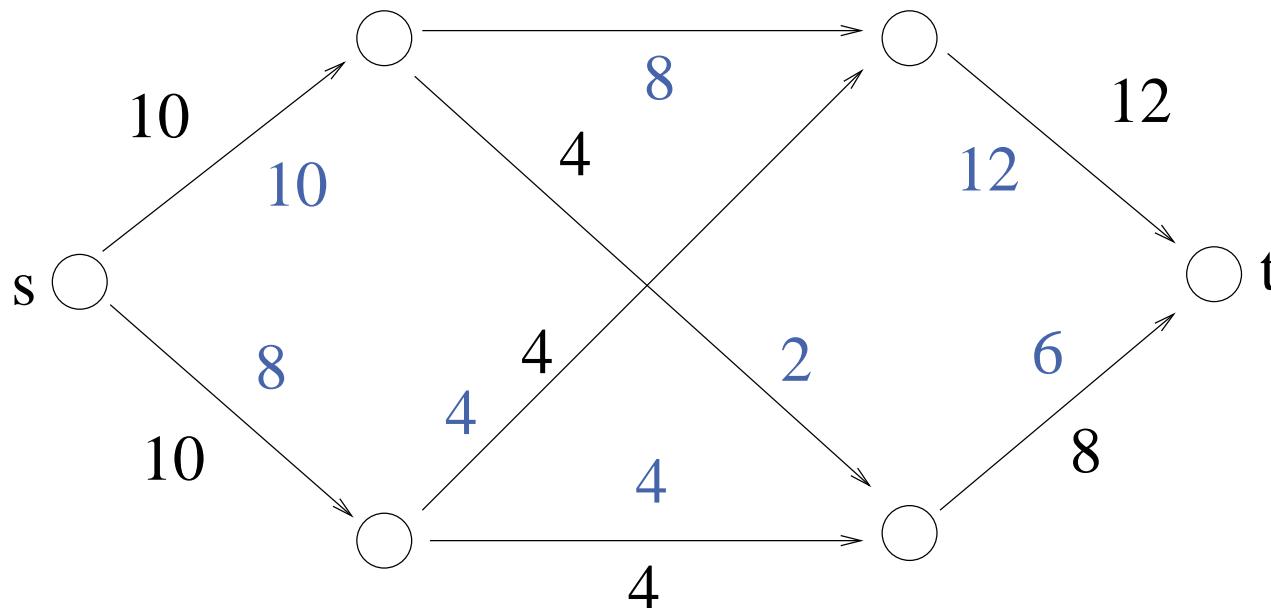
# Definitions: Network

- Network = directed weighted graph with  
**source node  $s$**  and **sink node  $t$**
- $s$  has no incoming edges,  $t$  has no outgoing edges
- Weight  $c_e$  of an edge  $e$  = **capacity** of  $e$  (nonnegative!)



## Definitions: Flows

- Flow = function  $f_e$  on the edges,  $0 \leq f_e \leq c_e \forall e$   
 $\forall v \in V \setminus \{s, t\}$ : total incoming flow = total outgoing flow
- Value of a flow  $\text{val}(f) = \text{total outgoing flow from } s =$   
**total flow going into  $t$**
- Goal: find a flow with maximum value

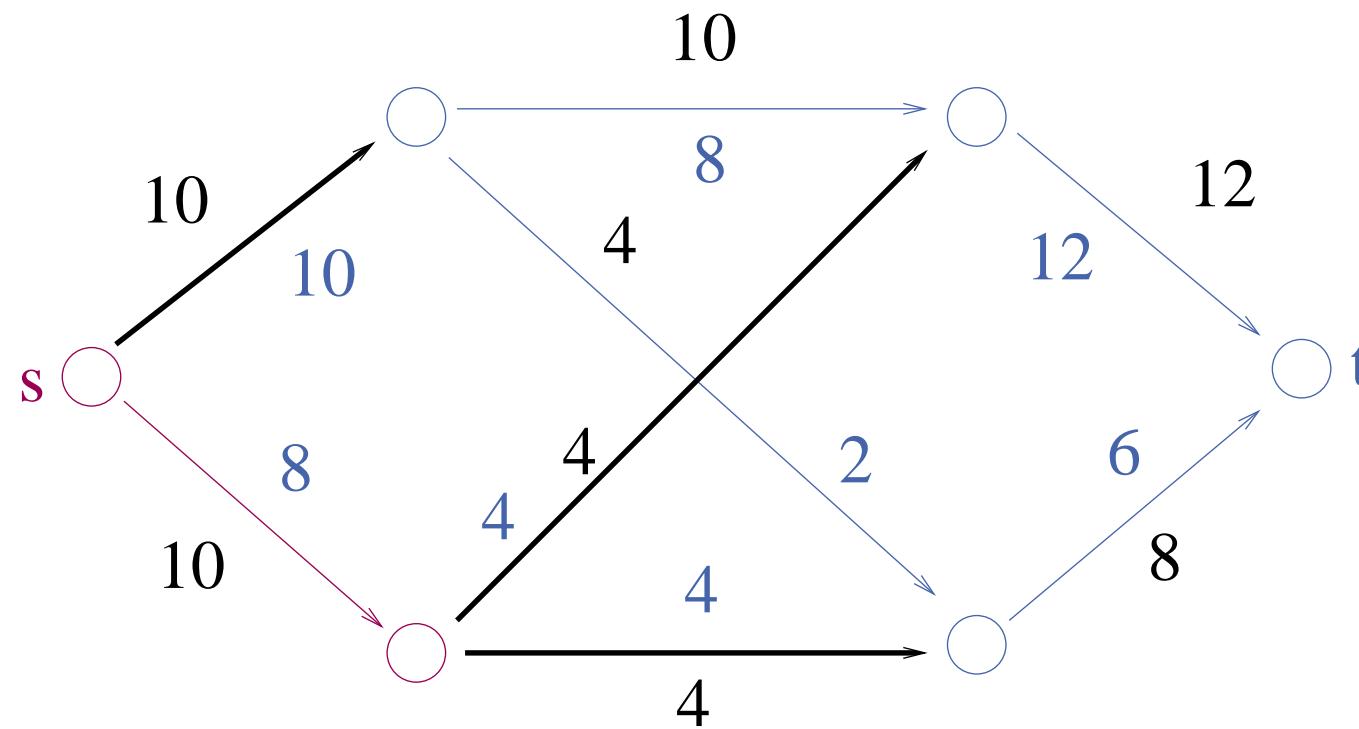


## Definitions: (Minimum) $s$ - $t$ Cuts

An  $s$ - $t$  cut is partition of  $V$  into  $S$  and  $T$  with  $s \in S$  and  $t \in T$ .

The **capacity** of this cut is:

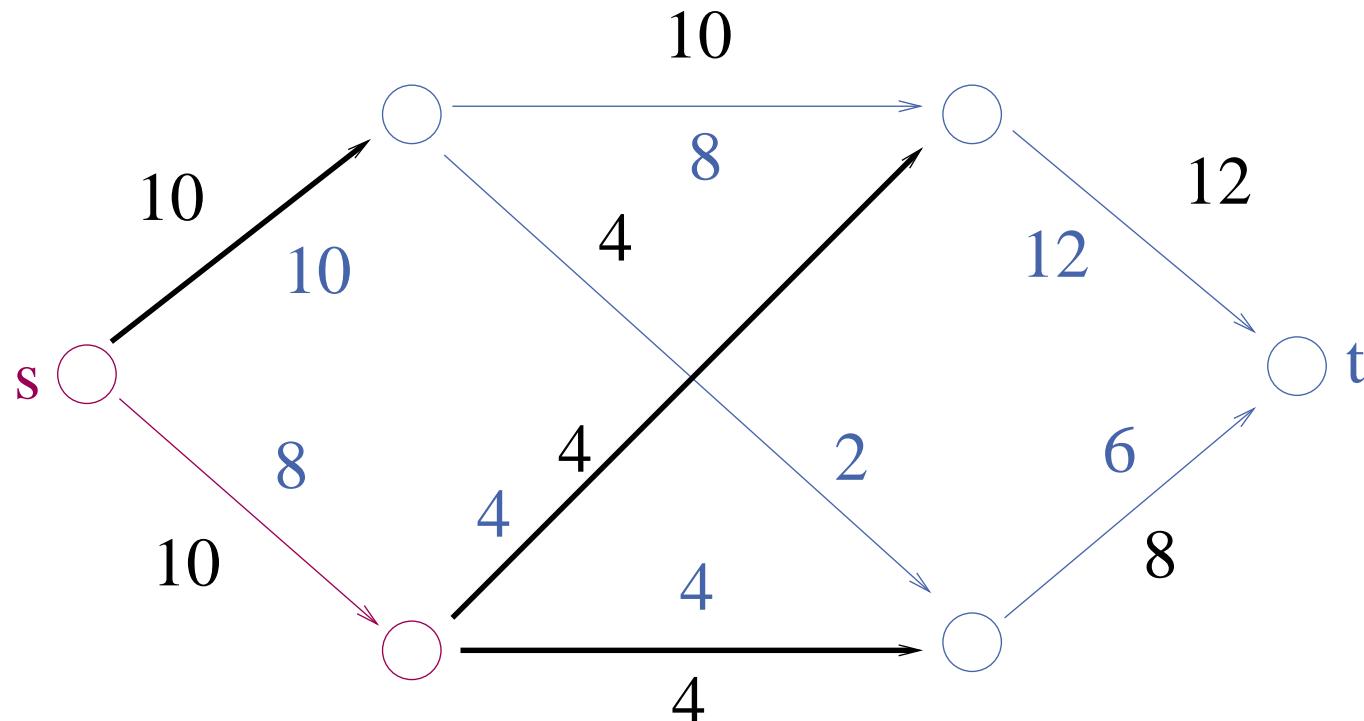
$$\sum \{c_{(u,v)} : u \in S, v \in T\}$$



# Duality Between Flows and Cuts

**Theorem:** [Elias/Feinstein/Shannon, Ford/Fulkerson 1956]

Value of an  $s$ - $t$  max-flow = minimum capacity of an  $s$ - $t$  cut.

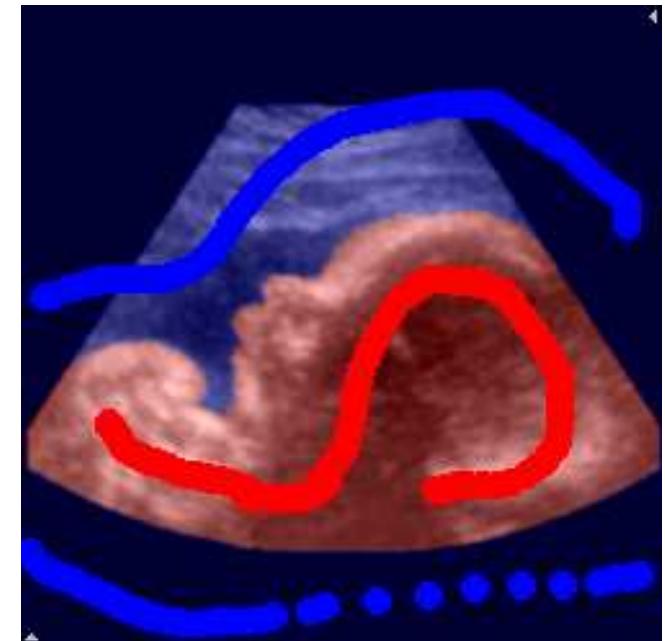


**Proof:** later



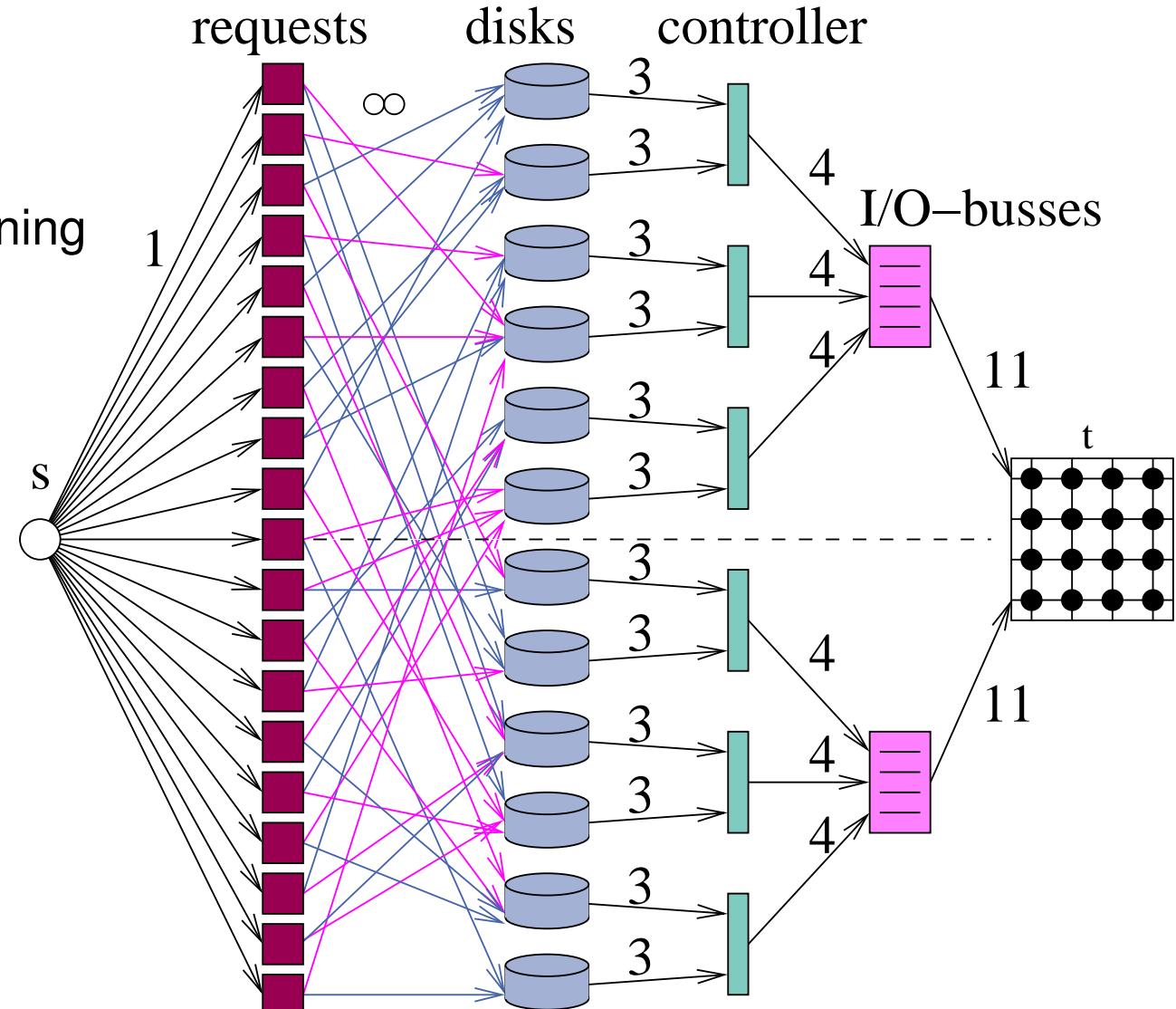
# Applications

- Oil pipes
- Traffic flows on highways
- Image Processing <http://vision.csd.uwo.ca/maxflow-data>
  - segmentation
  - stereo processing
  - multiview reconstruction
  - surface fitting
- disk/machine/tanker scheduling
- matrix rounding
- ...



# Applications in our Group

- multicasting using  
network coding
- balanced  $k$  partitioning
- disk scheduling



## Option 1: linear programming

- Flow variables  $x_e$  for each edge  $e$
- Flow on each edge is at most its capacity
- Incoming flow at each vertex = outgoing flow from this vertex
- Maximize outgoing flow from starting vertex

We can do better!



# Algorithms 1956–now

| Year | Author         | Running time     |                        |
|------|----------------|------------------|------------------------|
| 1956 | Ford-Fulkerson | $O(mnU)$         |                        |
| 1969 | Edmonds-Karp   | $O(m^2n)$        |                        |
| 1970 | Dinic          | $O(mn^2)$        |                        |
| 1973 | Dinic-Gabow    | $O(mn \log U)$   | $n$ = number of nodes  |
| 1974 | Karzanov       | $O(n^3)$         | $m$ = number of arcs   |
| 1977 | Cherkassky     | $O(n^2\sqrt{m})$ | $U$ = largest capacity |
| 1980 | Galil-Naamad   | $O(mn \log^2 n)$ |                        |
| 1983 | Sleator-Tarjan | $O(mn \log n)$   |                        |

| Year | Author                    | Running time   |
|------|---------------------------|--|
| 1986 | Goldberg-Tarjan           | $O(mn \log(n^2/m))$  |
| 1987 | Ahuja-Orlin               | $O(mn + n^2 \log U)$   |
| 1987 | Ahuja-Orlin-Tarjan        | $O(mn \log(2 + n\sqrt{\log U}/m))$                             |
| 1990 | Cheriyan-Hagerup-Mehlhorn | $O(n^3 / \log n)$  |
| 1990 | Alon                      | $O(mn + n^{8/3} \log n)$                                       |
| 1992 | King-Rao-Tarjan           | $O(mn + n^{2+\varepsilon})$                                    |
| 1993 | Philipps-Westbrook        | $O(mn \log n / \log \frac{m}{n} + n^2 \log^{2+\varepsilon} n)$ |
| 1994 | King-Rao-Tarjan           | $O(mn \log n / \log \frac{m}{n \log n})$ if $m \geq 2n \log n$ |
| 1997 | Goldberg-Rao              | $O(\min\{m^{1/2}, n^{2/3}\} m \log(n^2/m) \log U)$             |
| 2014 | Lee-Sidford               | $O(m\sqrt{n} \log^2 U)$  |

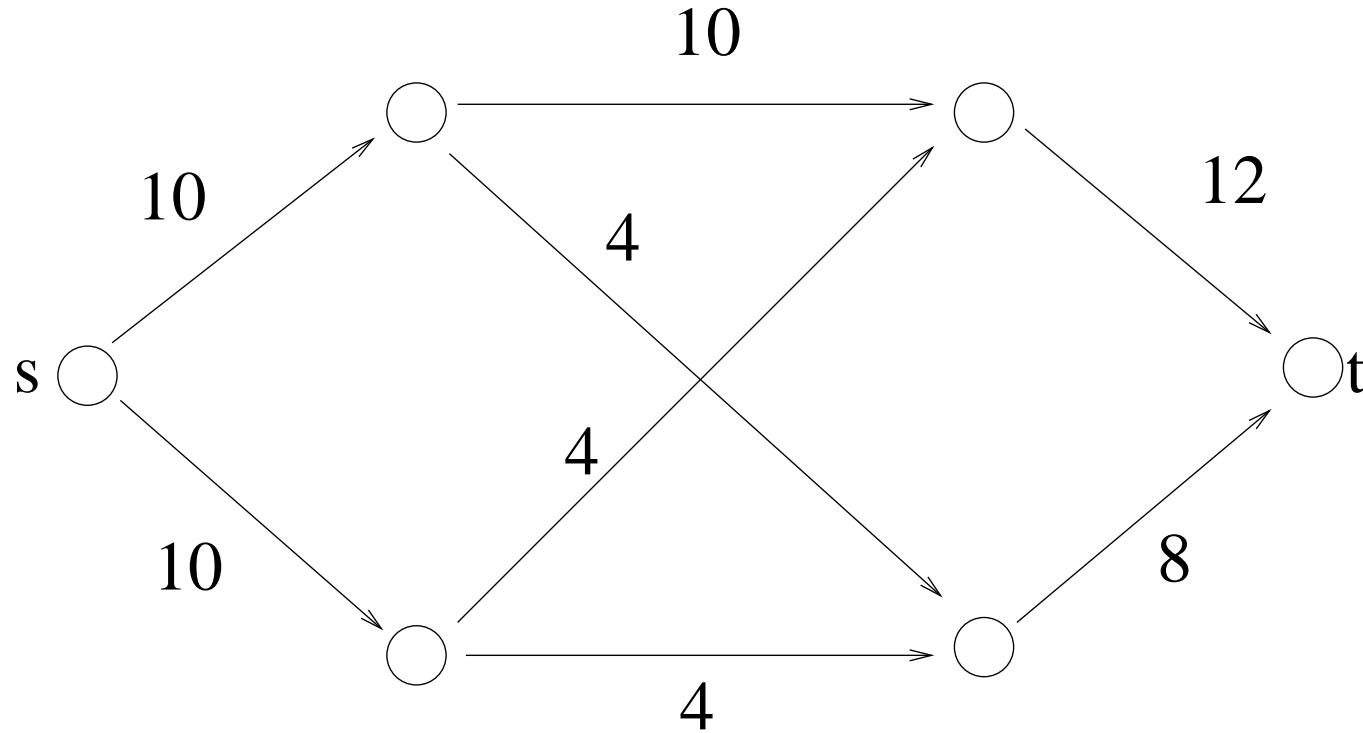
## Augmenting Paths (Rough Idea)

Find a path from  $s$  to  $t$  such that each edge has some **spare capacity**

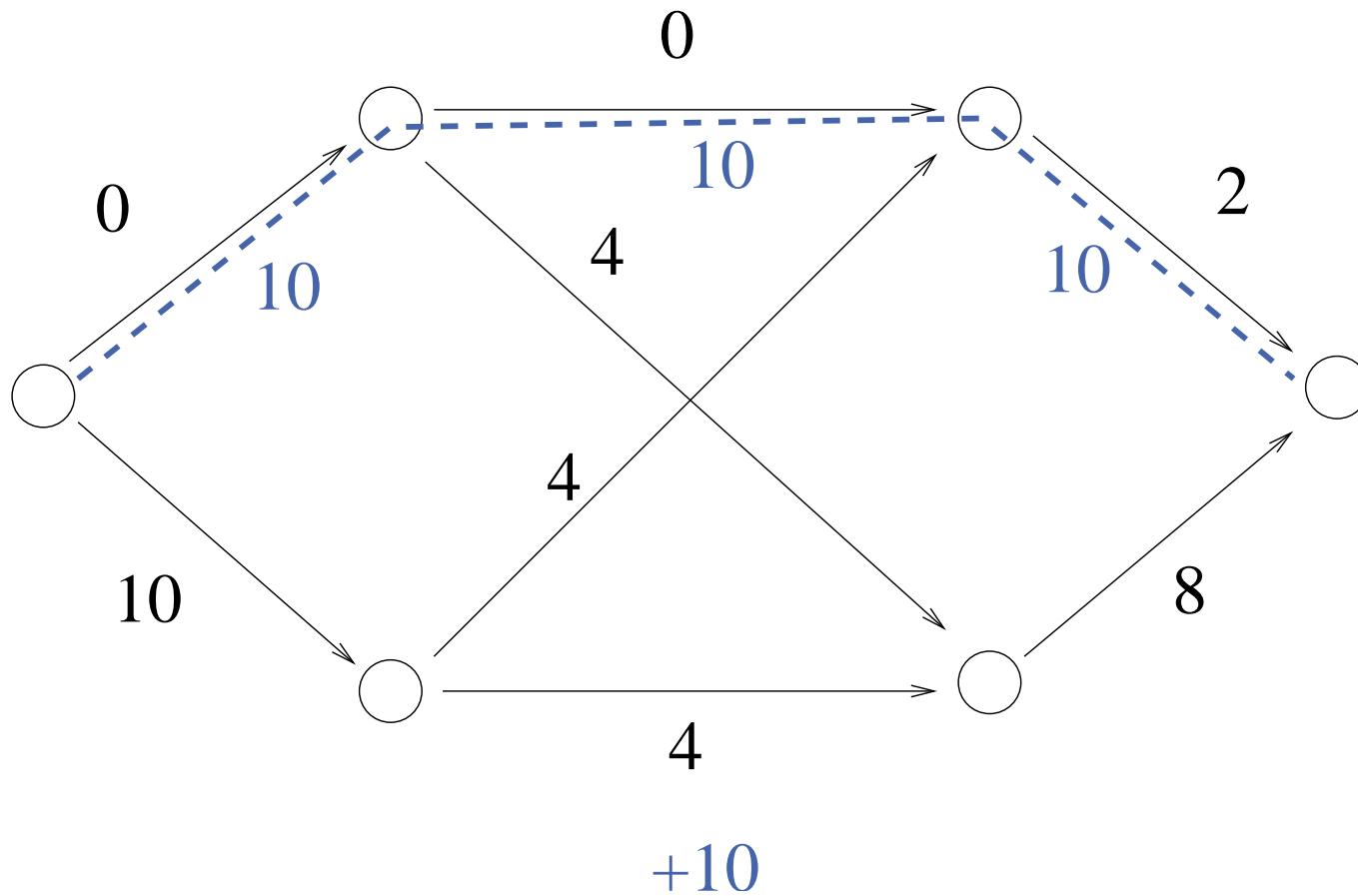
On this path, **saturate** the edge with the smallest spare capacity

**Adjust capacities** for all edges (create residual graph) and repeat

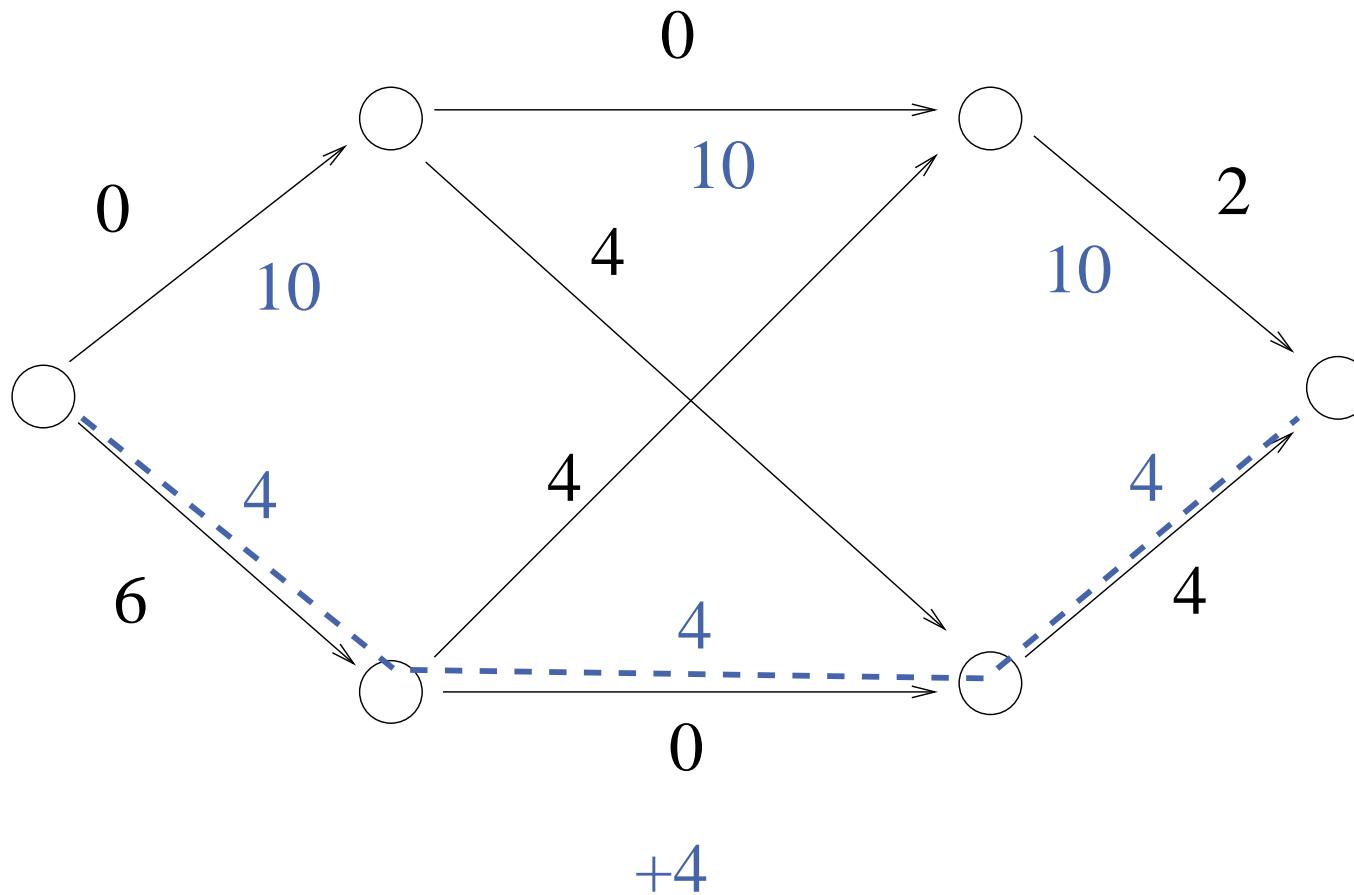
## Example



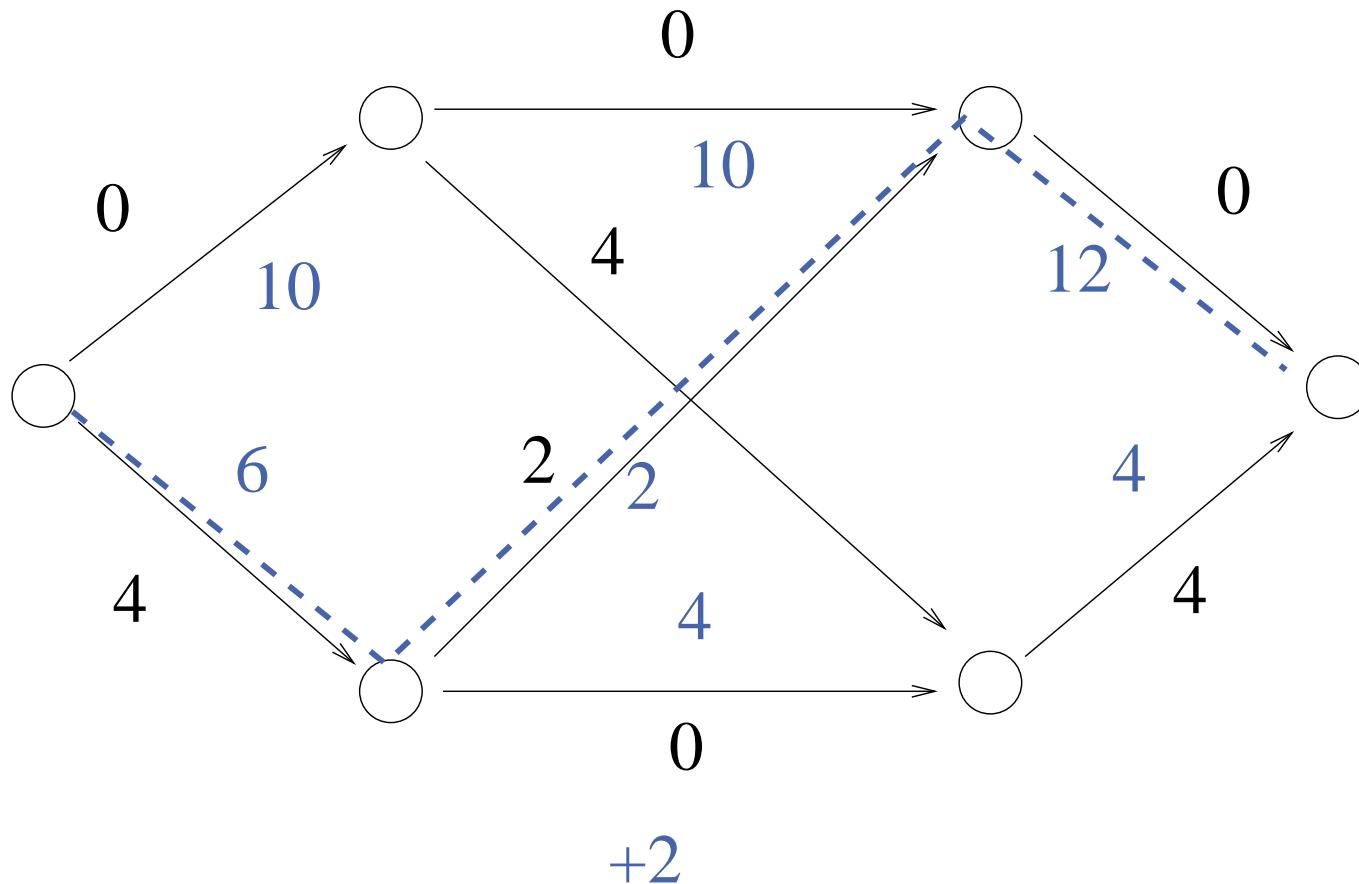
## Example



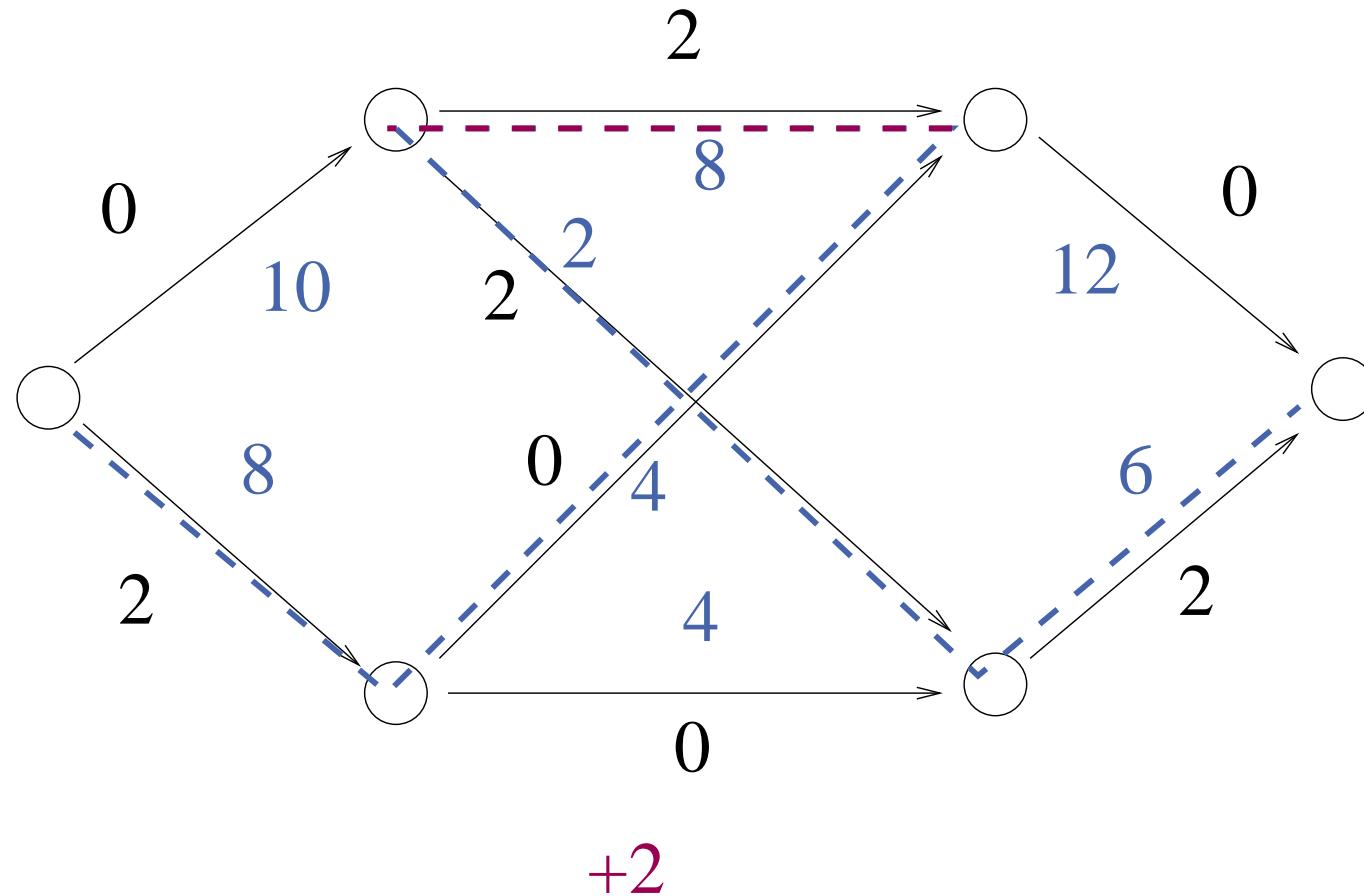
# Example



# Example



# Example

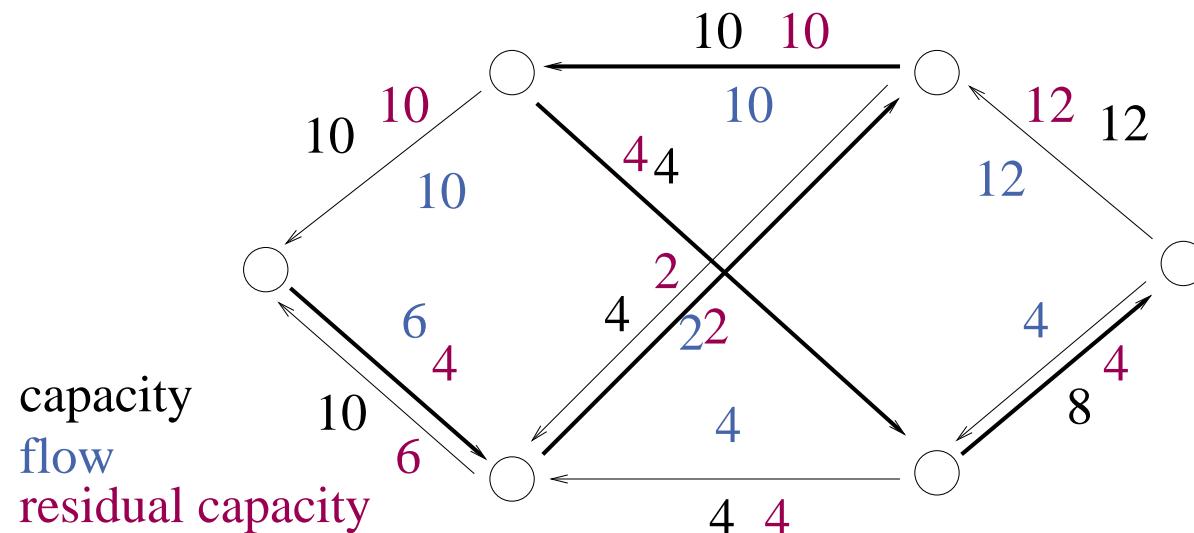


# Residual Graph

Given, network  $G = (V, E, c)$ , flow  $f$

Residual graph  $G_f = (V, E_f, c^f)$ . For each  $e \in E$  we have

$$\begin{cases} e \in E_f \text{ with } c_e^f = c_e - f(e) & \text{if } f(e) < c(e) \\ e^{\text{rev}} \in E_f \text{ with } c_{e^{\text{rev}}}^f = f(e) & \text{if } f(e) > 0 \end{cases}$$



# Augmenting Paths

Find a path  $p$  from  $s$  to  $t$  such that each edge  $e$  has nonzero **residual capacity**  $c_e^f$

$$\Delta f := \min_{e \in p} c_e^f$$

**foreach**  $(u, v) \in p$  **do**

**if**  $(u, v) \in E$  **then**  $f_{(u,v)}^+ = \Delta f$

**else**  $f_{(v,u)}^- = \Delta f$

# Ford Fulkerson Algorithm

**Function** FFMaxFlow( $G = (V, E), s, t, c : E \rightarrow \mathbb{N} : E \rightarrow \mathbb{N}$ )

$f := 0$

**while**  $\exists$  path  $p = (s, \dots, t)$  in  $G_f$  **do**

augment  $f$  along  $p$

**return**  $f$

time  $O(m\text{val}(f))$

# Ford Fulkerson – Correctness

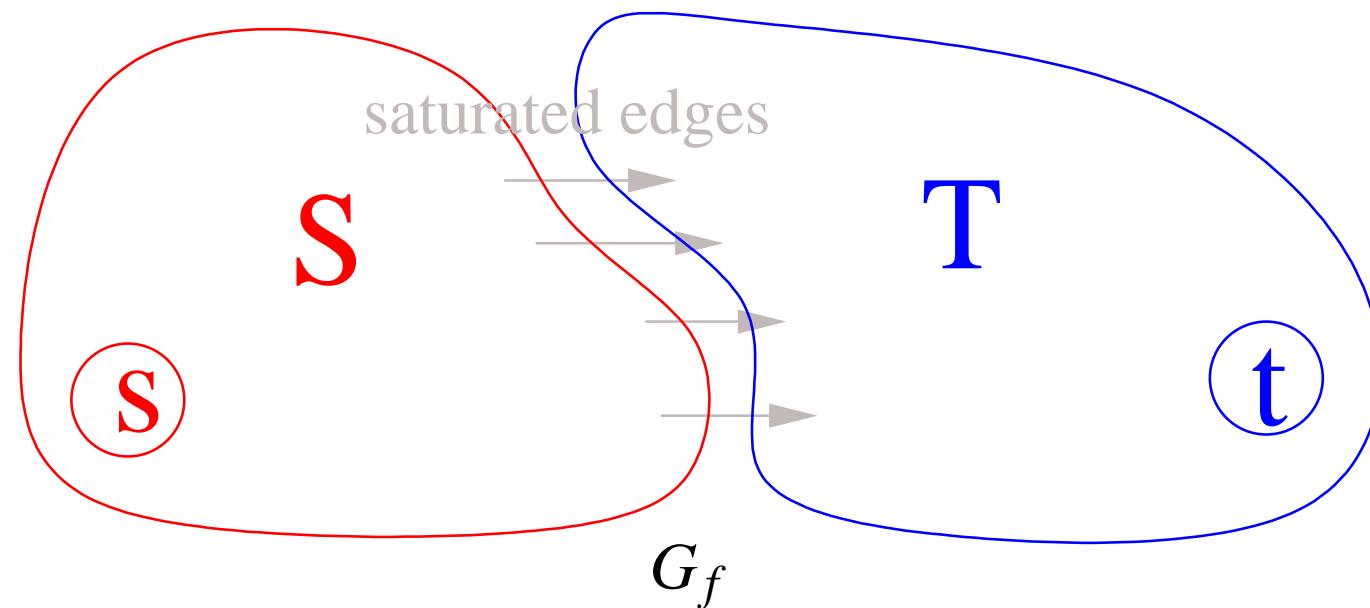
“Clearly” FF computes a feasible flow  $f$ . (Invariant)

Todo: flow value is maximal

At termination: no augmenting paths in  $G_f$  left.

Consider cut  $(S, T := V \setminus S)$  with

$$S := \{v \in V : v \text{ reachable from } s \text{ in } G_f\}$$



# A Basic Observations

**Lemma 1:** For any cut  $(S, T)$ :

$$\mathbf{val}(f) = \overbrace{\sum_{e \in E \cap S \times T} f_e}^{S \rightarrow T \text{ edges}} - \overbrace{\sum_{e \in E \cap T \times S} f_e}^{T \rightarrow S \text{ edges}} .$$

# Ford Fulkerson – Correctness

**Todo:**  $\text{val}(f)$  is maximal when no augmenting paths in  $G_f$  left.

Consider cut  $(S, T := V \setminus S)$  with

$$S := \{v \in V : v \text{ reachable from } s \text{ in } G_f\}.$$

**Observation:**  $\forall (u, v) \in E \cap T \times S : f(u, v) = 0$

otherwise  $c^f(v, u) > 0$  contradicting the definition of  $S$ .

$$\begin{aligned} \text{val}(f) &= \sum_{e \in E \cap S \times T} f_e - \sum_{e \in E \cap T \times S} f_e && \text{Lemma 1} \\ &= \sum_{e \in E \cap S \times T} f_e && \text{Observation above} \\ &= \sum_{e \in E \cap S \times T} c_{(u,v)} = (S, T) \text{ cut capacity} \end{aligned}$$

see next slide

# Max-Flow-Min-Cut theorem

**Theorem:** Max-flow = min-cut

**Proof:**

obvious: any-flow  $\leq$  max-flow  $\leq$  min-cut  $\leq$  any-cut

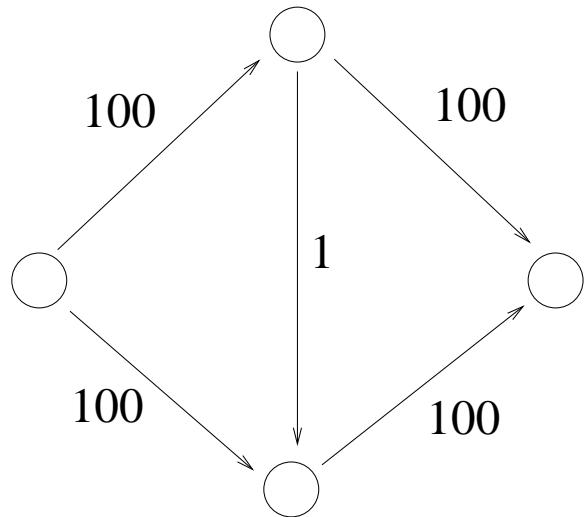
previous slide:

$(S, T)$  flow =  $(S, T)$  cut capacity

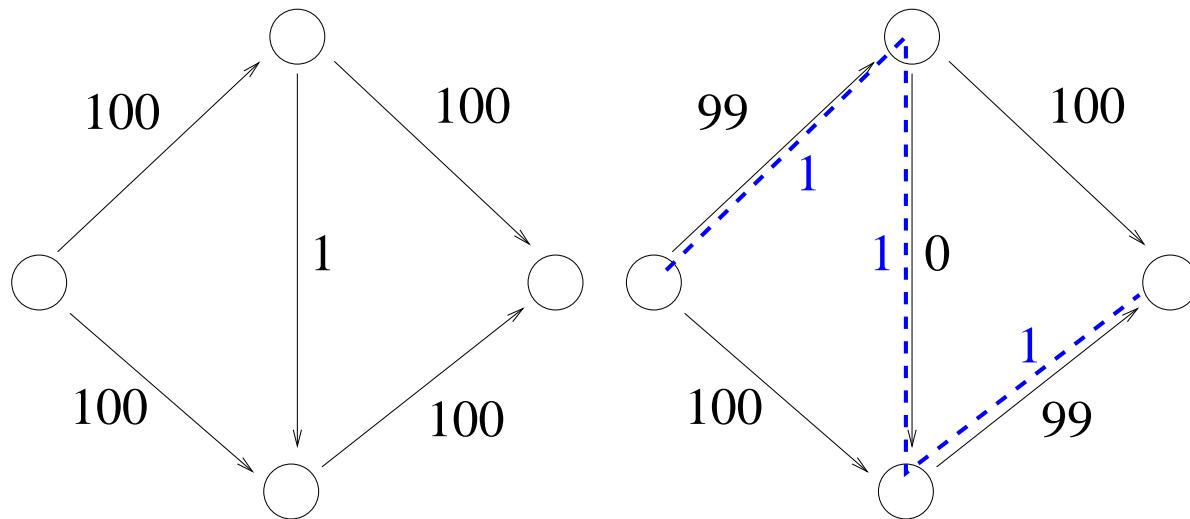
$\Rightarrow$

$(S, T)$  flow = max-flow = min-cut

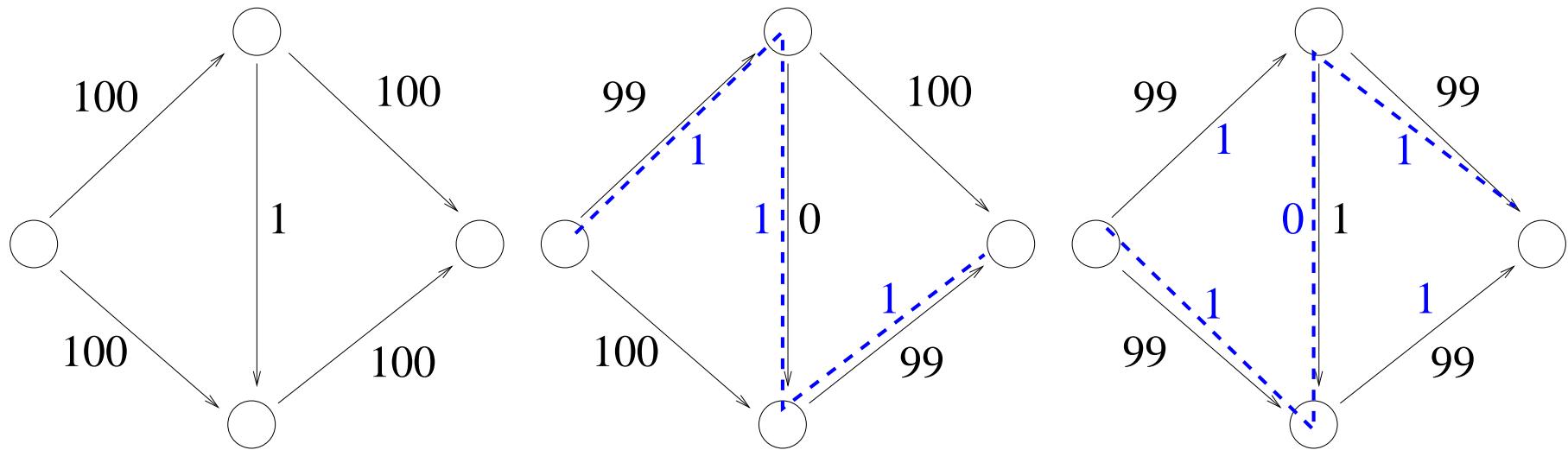
# A Bad Example for Ford Fulkerson



# A Bad Example for Ford Fulkerson



# A Bad Example for Ford Fulkerson





# An Even Worse Example for Ford Fulkerson

[U. Zwick, TCS 148, p. 165–170, 1995]

Let  $r = \frac{\sqrt{5} - 1}{2}$ .

Consider the graph

And the augmenting paths

$$p_0 = \langle s, c, b, t \rangle$$

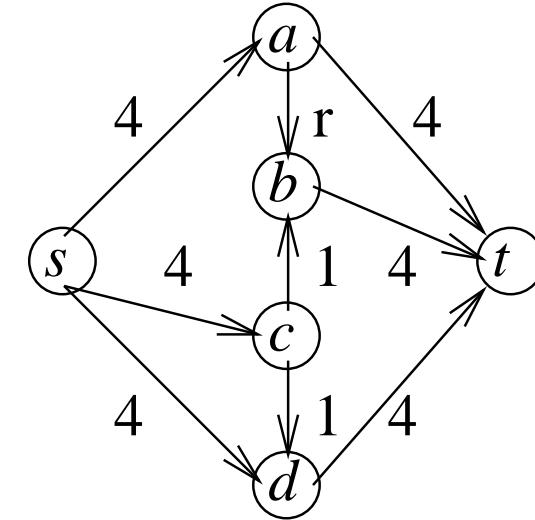
$$p_1 = \langle s, a, b, c, d, t \rangle$$

$$p_2 = \langle s, c, b, a, t \rangle$$

$$p_3 = \langle s, d, c, b, t \rangle$$

The sequence of augmenting paths  $p_0(p_1, p_2, p_1, p_3)^*$  is an infinite sequence of positive flow augmentations.

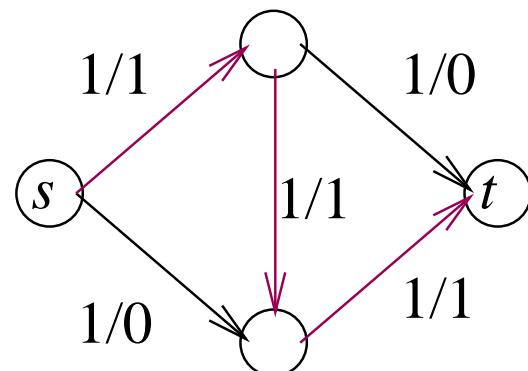
The flow value does **not** converge to the maximum value 9.



# Blocking Flows

$f_b$  is a **blocking flow** in  $H$  if

$$\forall \text{paths } p = \langle s, \dots, t \rangle : \exists e \in p : f_b(e) = c(e)$$



# Dinitz Algorithm

**Function** DinitzMaxFlow( $G = (V, E), s, t, c : E \rightarrow \mathbb{N}$ ) :  $E \rightarrow \mathbb{N}$

$f := 0$

**while**  $\exists$  path  $p = (s, \dots, t)$  in  $G_f$  **do**

$d = G_f.\text{reverseBFS}(t) : V \rightarrow \mathbb{N}$

$L_f = (V, \{(u, v) \in E_f : d(v) = d(u) - 1\})$  // layer graph

find a blocking flow  $f_b$  in  $L_f$

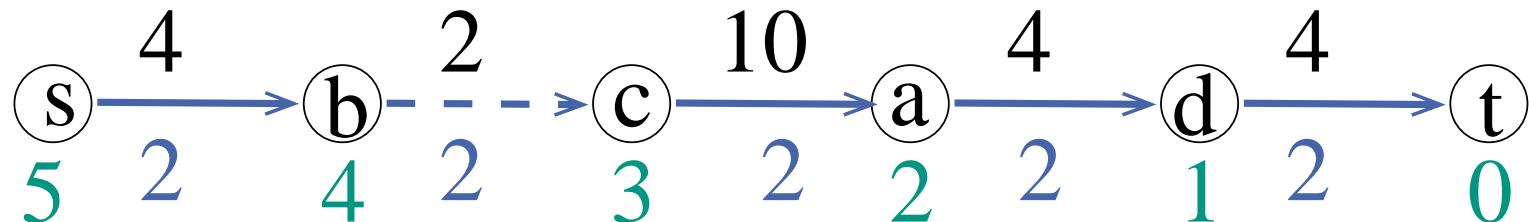
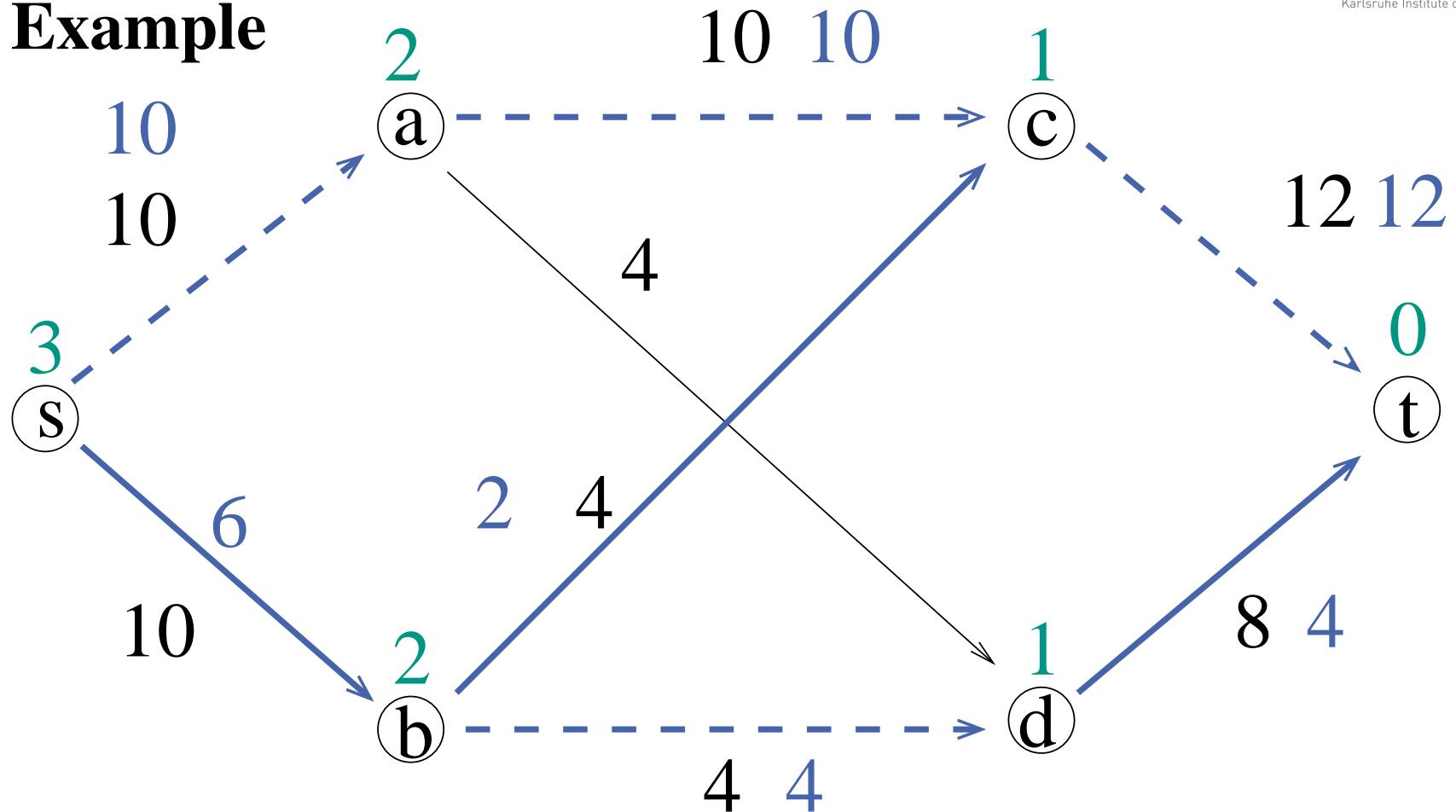
augment  $f += f_b$

**return**  $f$

# Dinitz – Correctness

analogous to Ford-Fulkerson

## Example



# Computing Blocking Flows

Idee: wiederholte DFS nach augmentierenden Pfaden

**Function** blockingFlow( $L_f = (V, E)$ ) :  $E \rightarrow \mathbb{N}$

$p = \langle s \rangle$  : Path;  $f_b = 0$  : Flow

**loop** // Round

$v := p.\text{last}()$

**if**  $v = t$  **then** // breakthrough

$\delta := \min \{c(e) - f_b(e) : e \in p\}$

**foreach**  $e \in p$  **do**

$f_b(e) += \delta$

**if**  $f_b(e) = c(e)$  **then remove**  $e$  from  $E$

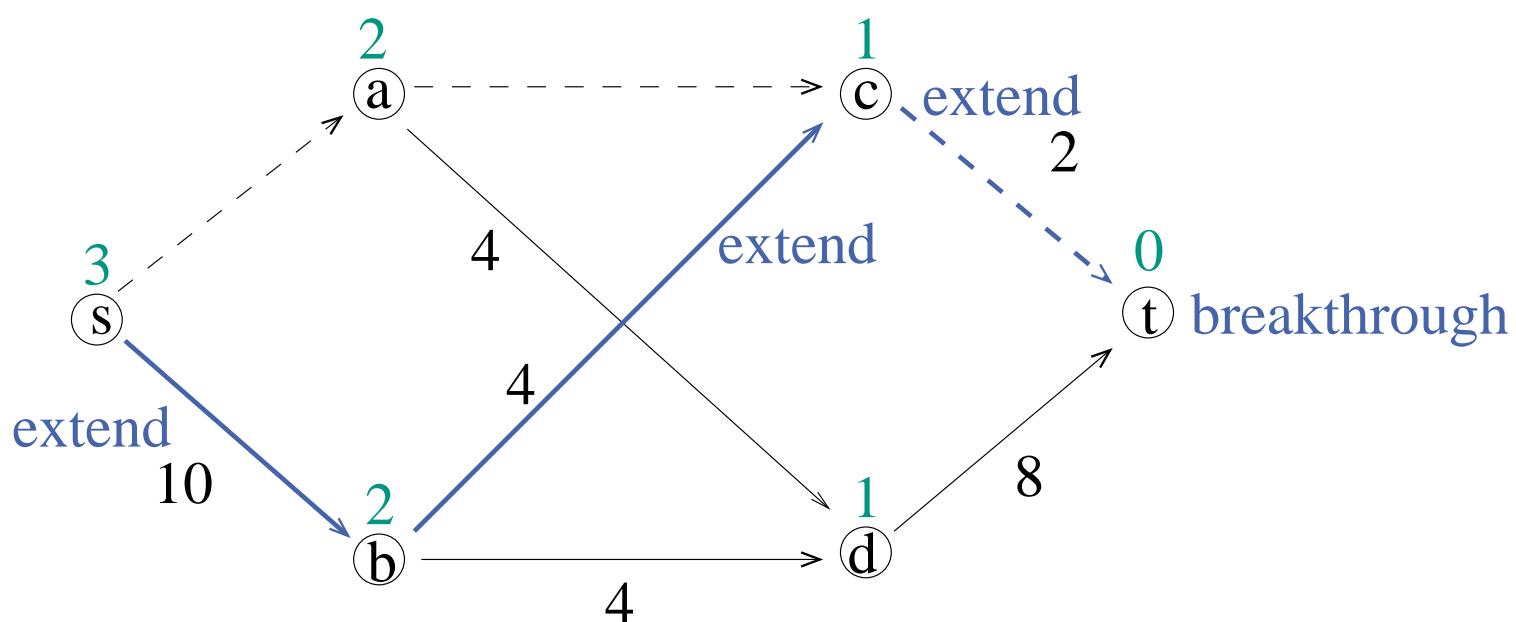
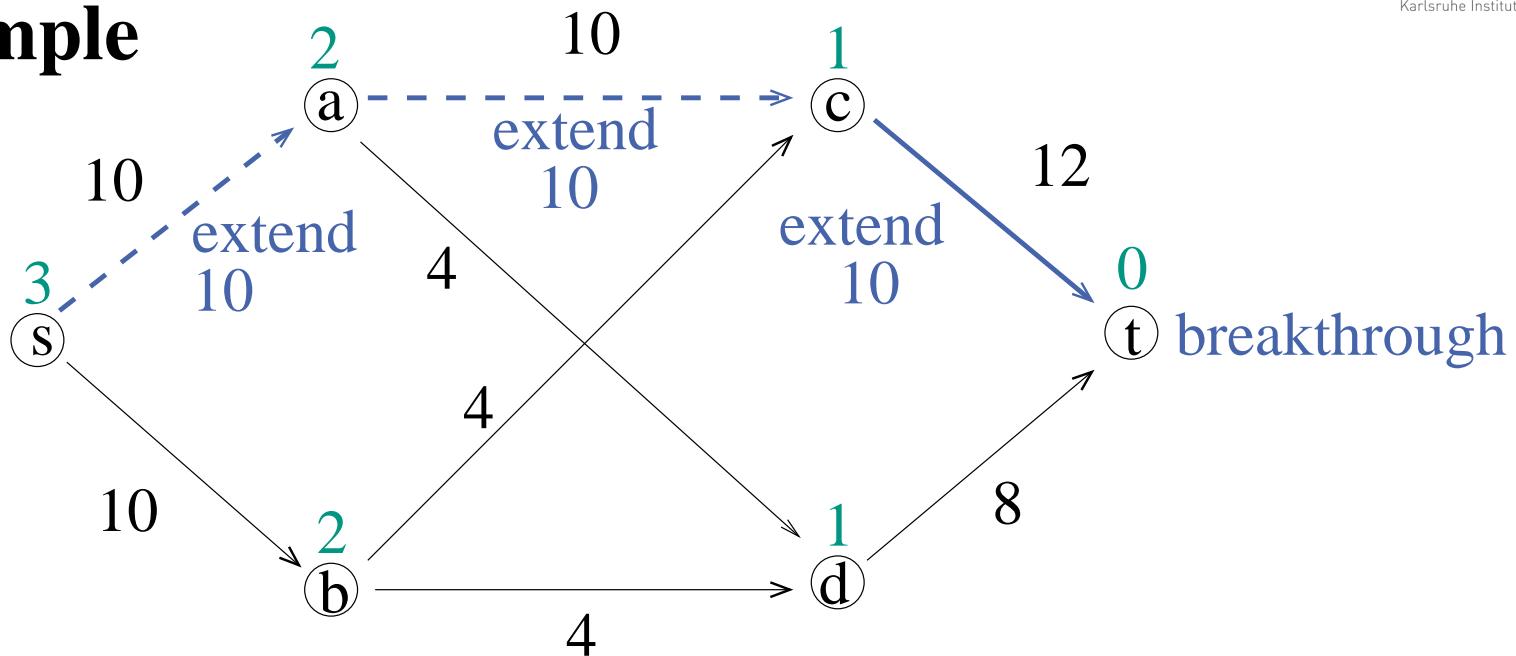
$p := \langle s \rangle$

**else if**  $\exists e = (v, w) \in E$  **then**  $p.\text{pushBack}(w)$  // extend

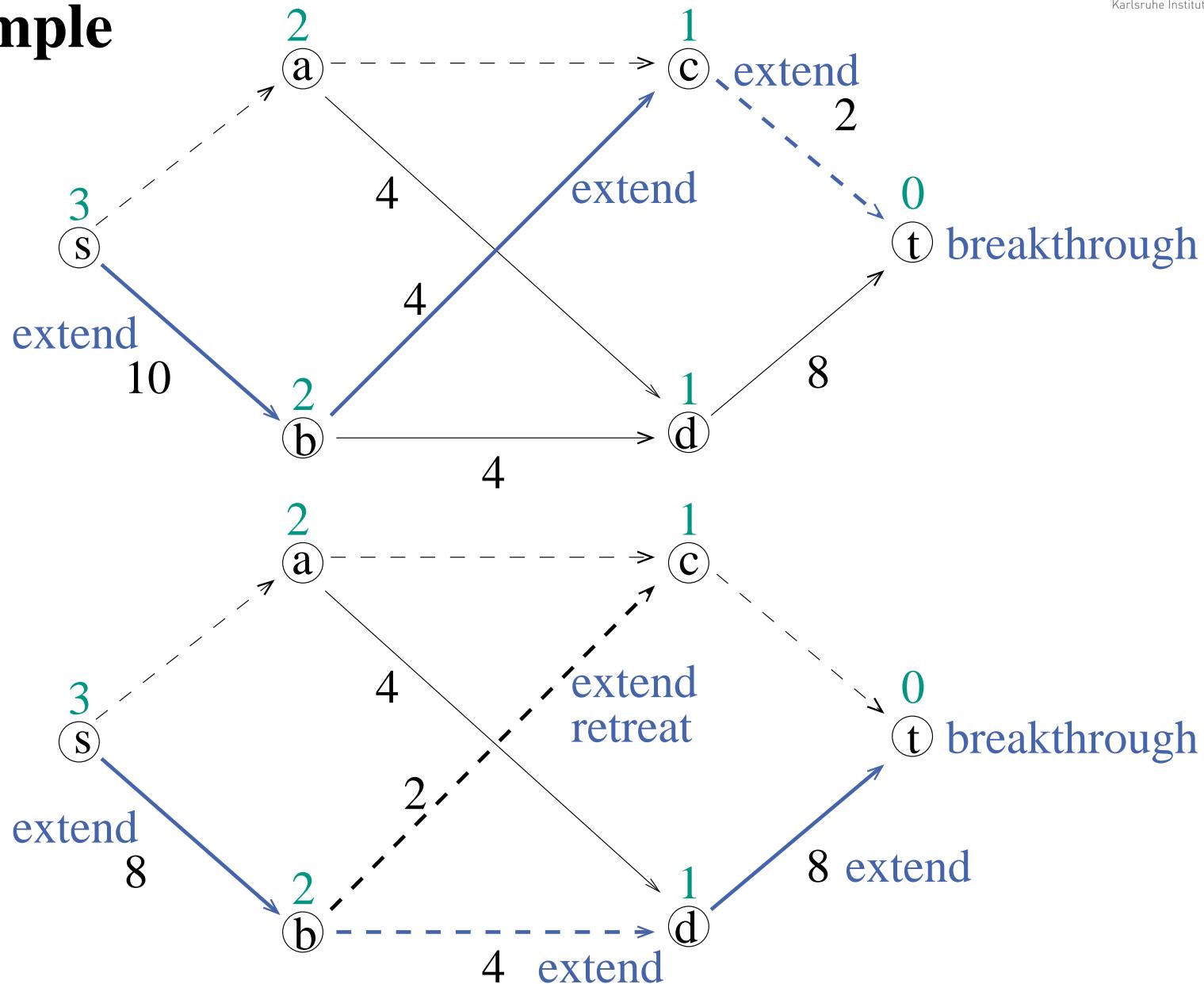
**else if**  $v = s$  **then return**  $f_b$  // done

**else** delete the last edge from  $p$  in  $p$  and  $E$  // retreat

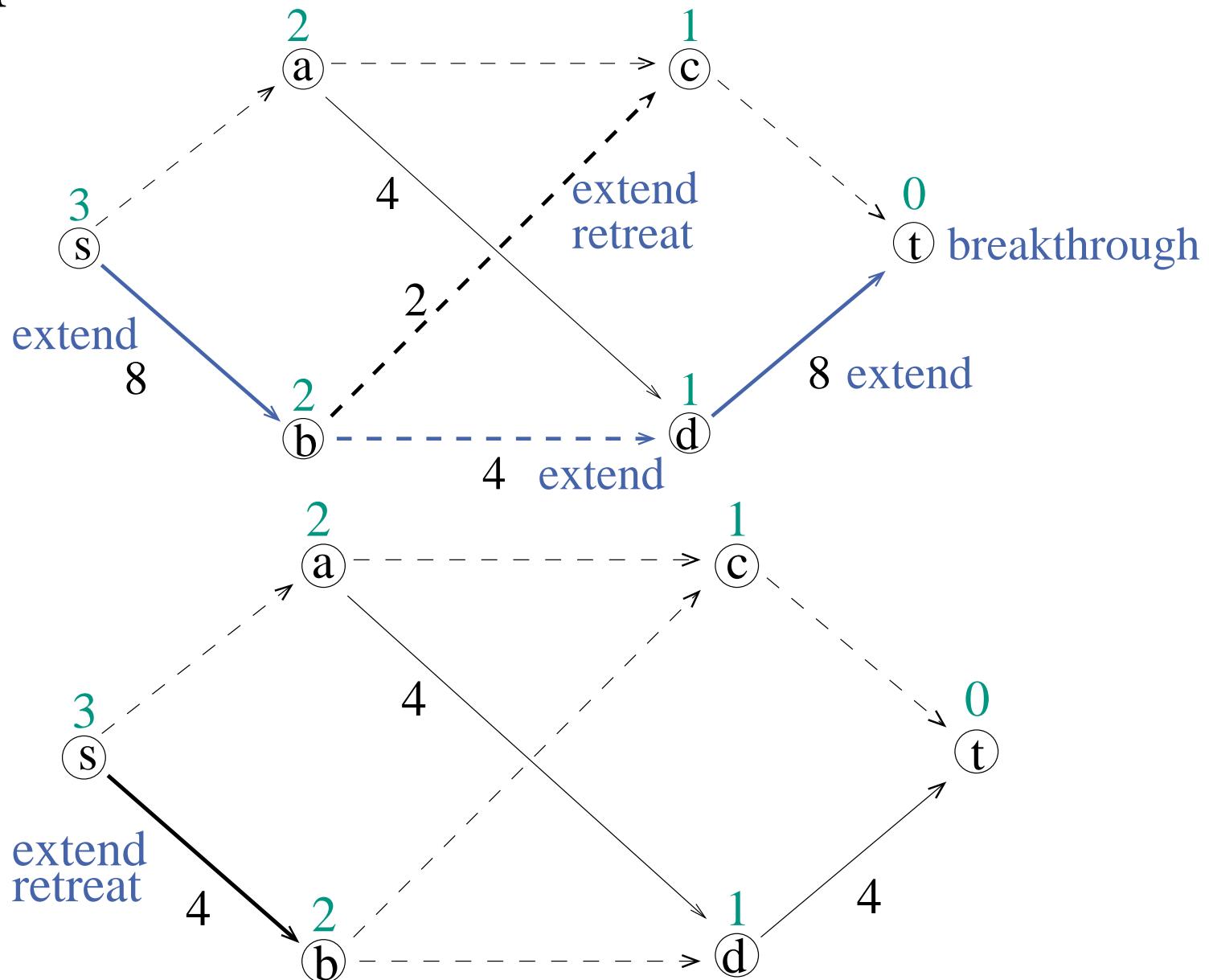
## Example



# Example



# Example



# Blocking Flows Analysis 1

- running time  $\#_{extends} + \#_{retreats} + n \cdot \#_{breakthroughs}$
- $\#_{breakthroughs} \leq m$ 
  - $\geq 1$  edge is saturated
- $\#_{retreats} \leq m$ 
  - one edge is removed
- $\#_{extends} \leq \#_{retreats} + n \cdot \#_{breakthroughs}$ 
  - a retreat cancels 1 extend, a breakthrough cancels  $\leq n$  extends

time is  $O(m + nm) = O(nm)$

## Blocking Flows Analysis 2

### Unit capacities:

breakthroughs saturates **all** edges on  $p$ , i.e., amortized constant cost per edge.

time  $O(m + n)$

## Blocking Flows Analysis 3

Dynamic trees: breakthrough (!), retreat, extend in time  $O(\log n)$

time  $O((m+n)\log n)$

“Theory alert”: In practice, this seems to be slower  
(few breakthroughs, many retreat, extend ops.)

# Dinitz Analysis 1

**Lemma 1.**  $d(s)$  increases by at least one in each round.

*Beweis.* not here



## Dinitz Analysis 2

- $\leq n$  rounds
- time  $O(mn)$  each

time  $O(mn^2)$  (**strongly polynomial**)

time  $O(mn \log n)$  with dynamic trees

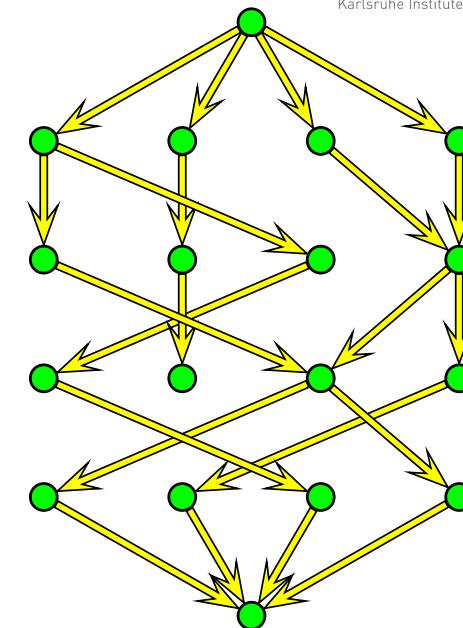
# Dinitz Analysis 3 – Unit Capacities

**Lemma 2.** At most  $2\sqrt{m}$  BF computations:

*Beweis.* Consider iteration  $k = \sqrt{m}$ .

Cut in layergraph induces cut in residual graph of capacity at most  $\sqrt{m}$ .

At most  $\sqrt{m}$  additional phases.



Total time:  $O((m+n)\sqrt{m})$

more detailed analysis:  $O(m \min \{m^{1/2}, n^{2/3}\})$

## Dinitz Analysis 4 – Unit Networks

Unit capacity +  $\forall v \in V : \min \{\text{indegree}(v), \text{outdegree}(v)\} = 1$ :

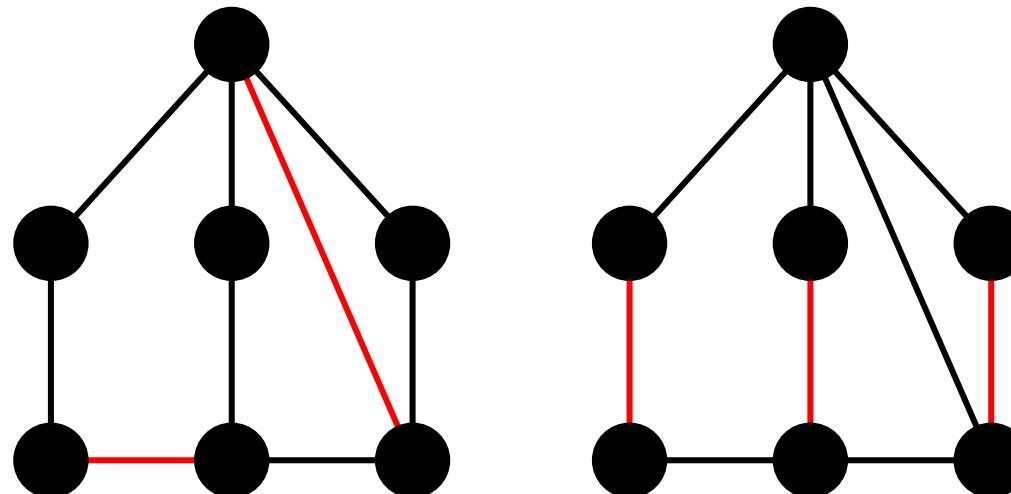
time:  $O((m+n)\sqrt{n})$

# Matching

$M \subseteq E$  is a **matching** in the undirected graph  $G = (V, E)$  iff  
 $(V, M)$  has maximum degree  $\leq 1$ .

$M$  is **maximal** if  $\nexists e \in E \setminus M : M \cup \{e\}$  is a matching.

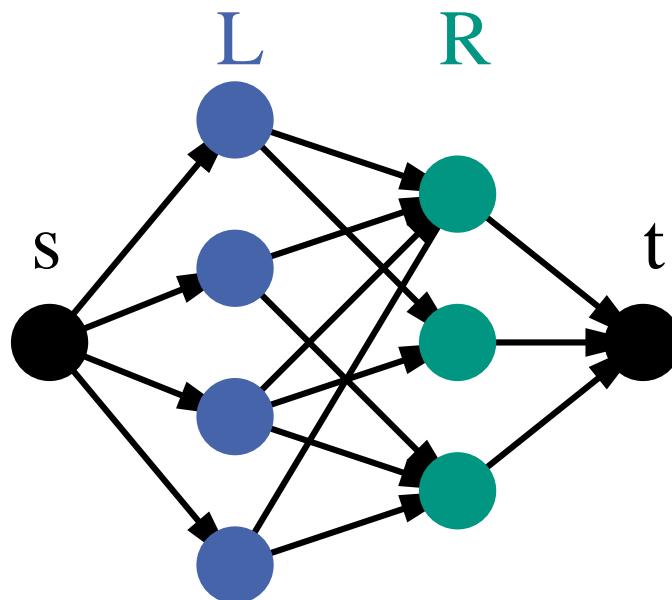
$M$  has **maximum** cardinality if  $\nexists$  matching  $M' : |M'| > |M|$



# Maximum Cardinality Bipartite Matching

in  $(L \cup R, E)$ . Model as a **unit network maximum flow** problem

$$(\{s\} \cup L \cup R \cup \{t\}, \{(s, u) : u \in L\} \cup E \cup \{(v, t) : v \in R\})$$



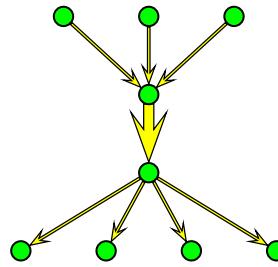
Dinitz algorithm yields  $O((n + m)\sqrt{n})$  algorithm



# Similar Performance for Weighted Graphs?

time:  $O\left(m \min\left\{m^{1/2}, n^{2/3}\right\} \log C\right)$  [Goldberg Rao 97]

**Problem:** Fat edges between layers ruin the argument



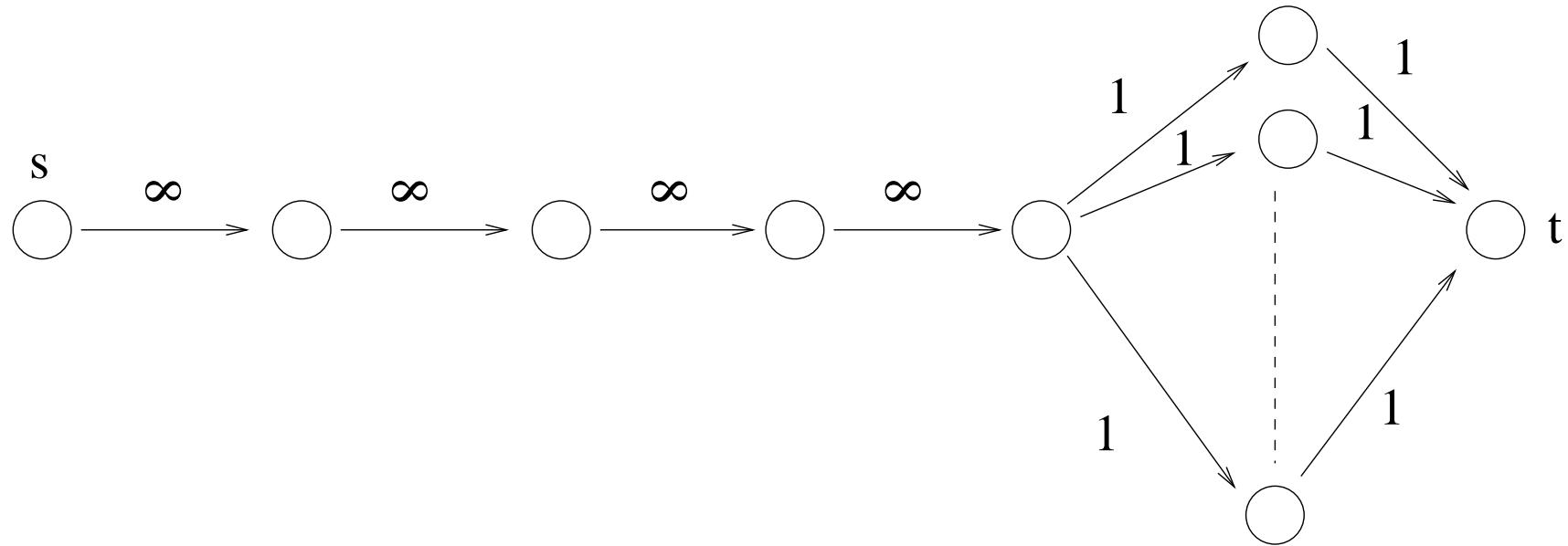
Idea: **scale** a parameter  $\Delta$  from small to large

contract SCCs of fat edges (capacity  $> \Delta$ )

Experiments [Hagerup, Sanders Träff 98]:

Sometimes best algorithm usually slower than **preflow push**

# Disadvantage of augmenting paths algorithms



# Preflow-Push Algorithms

Preflow  $f$ : a flow where the **flow conservation** constraint is relaxed to

$$\text{excess}(v) := \overbrace{\sum_{(u,v) \in E} f_{(u,v)}}^{\text{inflow}} - \overbrace{\sum_{(v,w) \in E} f_{(v,w)}}^{\text{outflow}} \geq 0 .$$

$v \in V \setminus \{s, t\}$  is **active** iff  $\text{excess}(v) > 0$

**Procedure**  $\text{push}(e = (v, w), \delta)$

**assert**  $\delta > 0 \wedge \text{excess}(v) \geq \delta$

**assert** residual capacity of  $e \geq \delta$

$\text{excess}(v)- = \delta$

$\text{excess}(w)+ = \delta$

**if**  $e$  is reverse edge **then**  $f(\text{reverse}(e))- = \delta$

**else**  $f(e)+ = \delta$

# Level Function

Idea: make progress by pushing **towards**  $t$

Maintain

an **approximation**  $d(v)$  of the BFS distance from  $v$  to  $t$  **in**  $G_f$ .

**invariant**  $d(t) = 0$

**invariant**  $d(s) = n$

**invariant**  $\forall(v, w) \in E_f : d(v) \leq d(w) + 1$  // no **steep** edges

Edge directions of  $e = (v, w)$

**steep**:  $d(w) < d(v) - 1$

**downward**:  $d(w) < d(v)$

**horizontal**:  $d(w) = d(v)$

**upward**:  $d(w) > d(v)$

**Procedure** genericPreflowPush( $G = (V, E)$ ,  $f$ )

```

forall  $e = (s, v) \in E$  do  $\text{push}(e, c(e))$            // saturate
 $d(s) := n$ 
 $d(v) := 0$  for all other nodes
while  $\exists v \in V \setminus \{s, t\} : \text{excess}(v) > 0$  do           // active node
  if  $\exists e = (v, w) \in E_f : d(w) < d(v)$  then // eligible edge
    choose some  $\delta \leq \min \left\{ \text{excess}(v), c_e^f \right\}$ 
     $\text{push}(e, \delta)$                                 // no new steep edges
  else  $d(v)++$                                 // relabel. No new steep edges

```

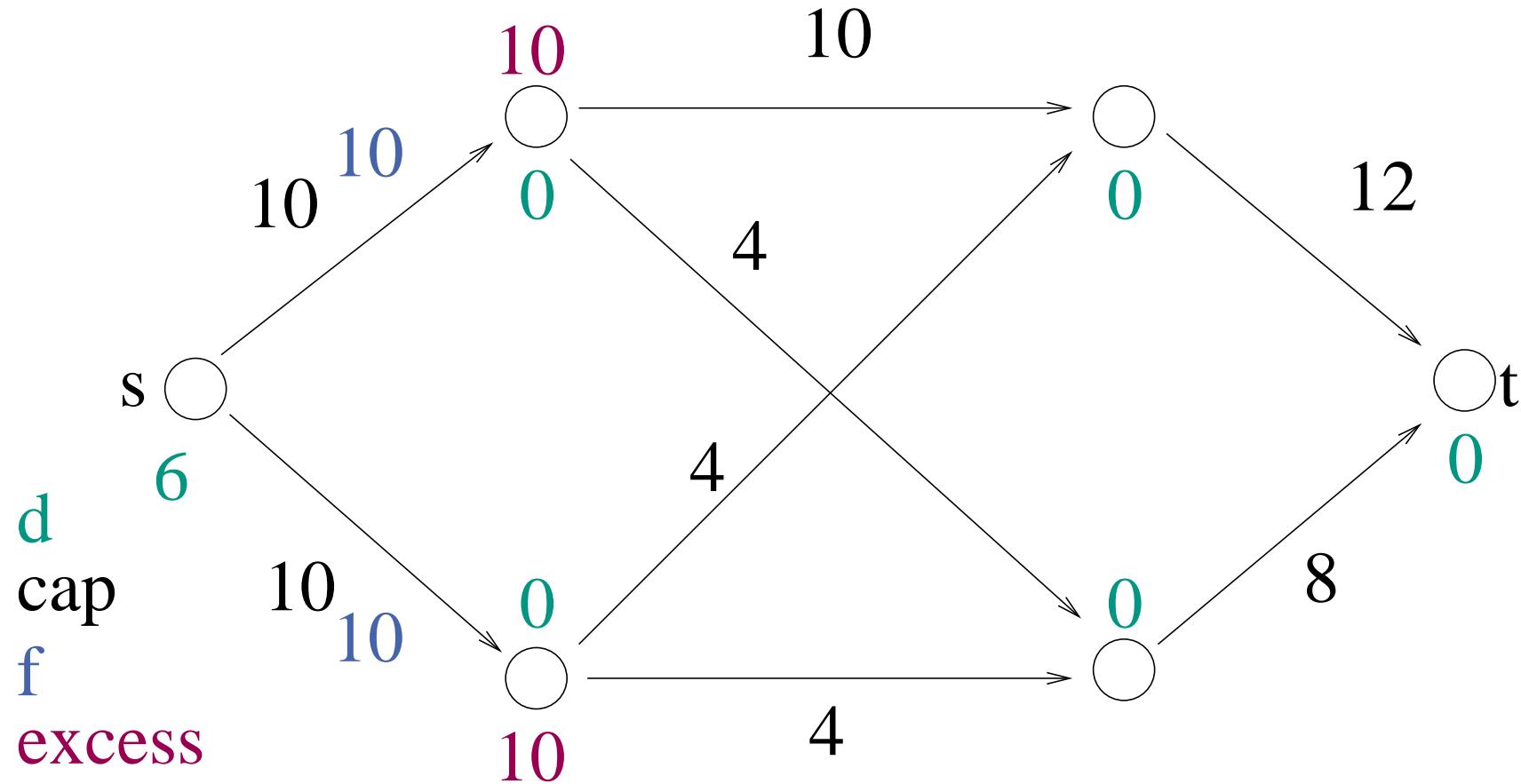
Obvious choice for  $\delta$ :  $\delta = \min \left\{ \text{excess}(v), c_e^f \right\}$

Saturating push:  $\delta = c_e^f$

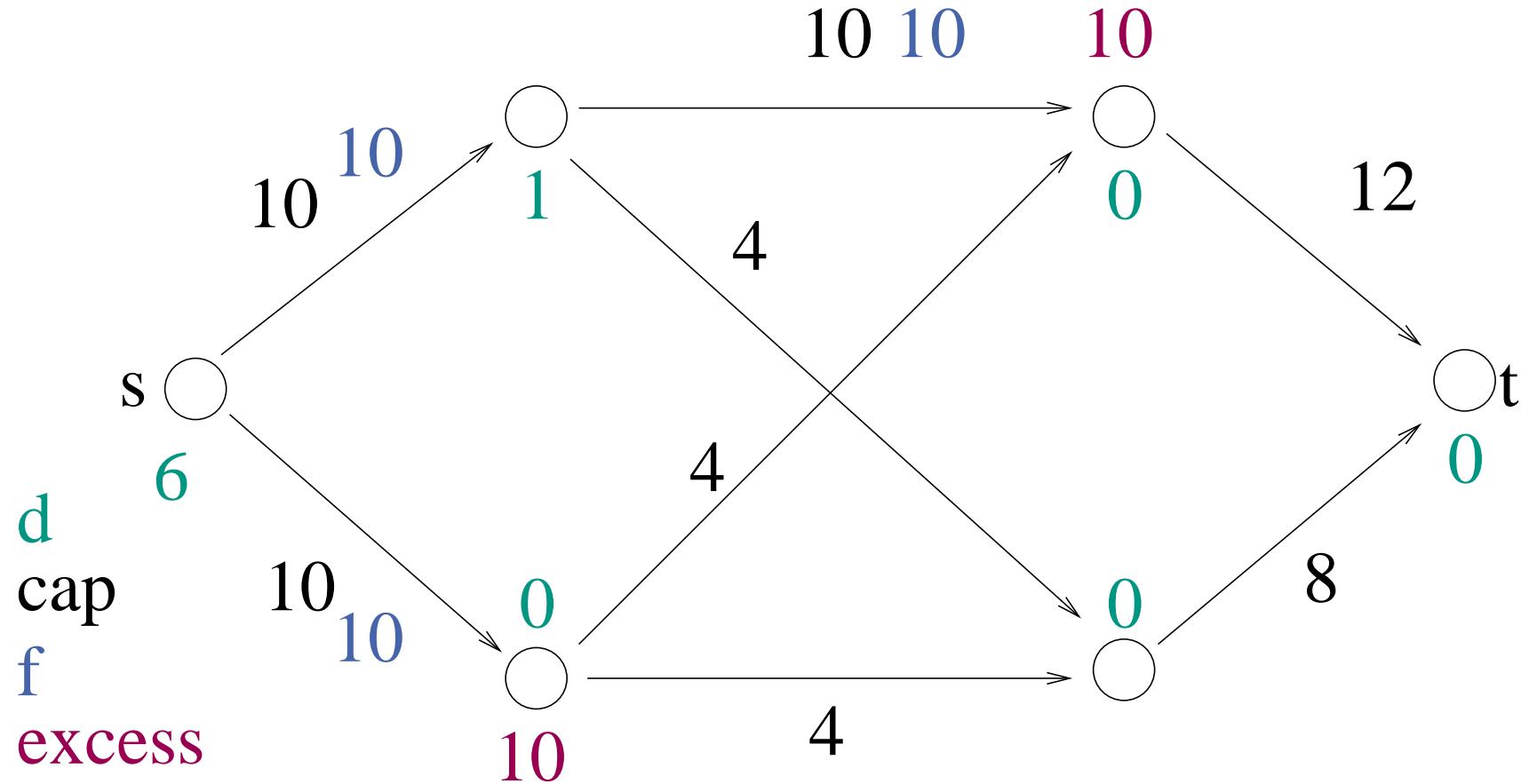
nonsaturating push:  $\delta < c_e^f$

To be filled in: How to select active nodes and eligible edges?

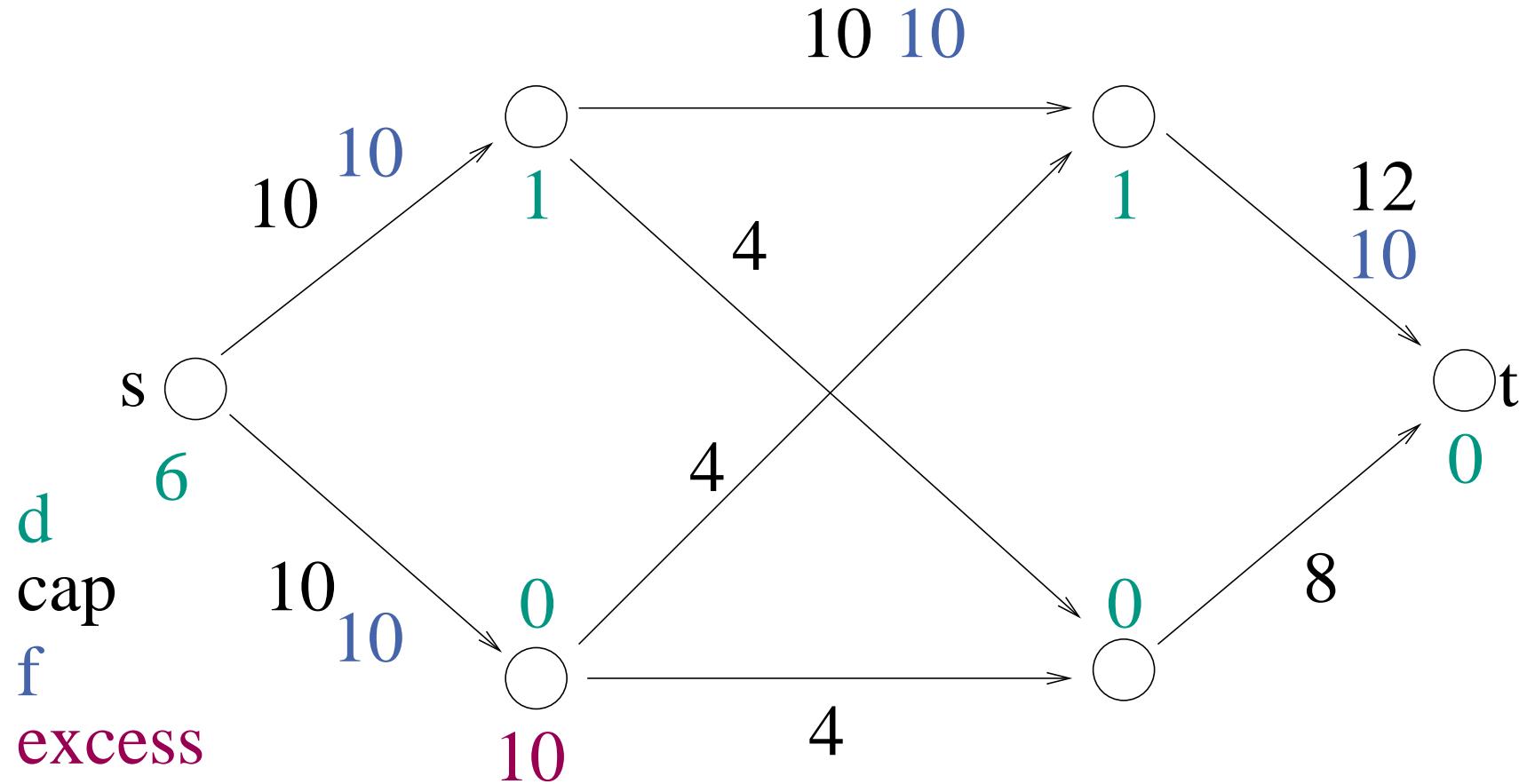
# Example



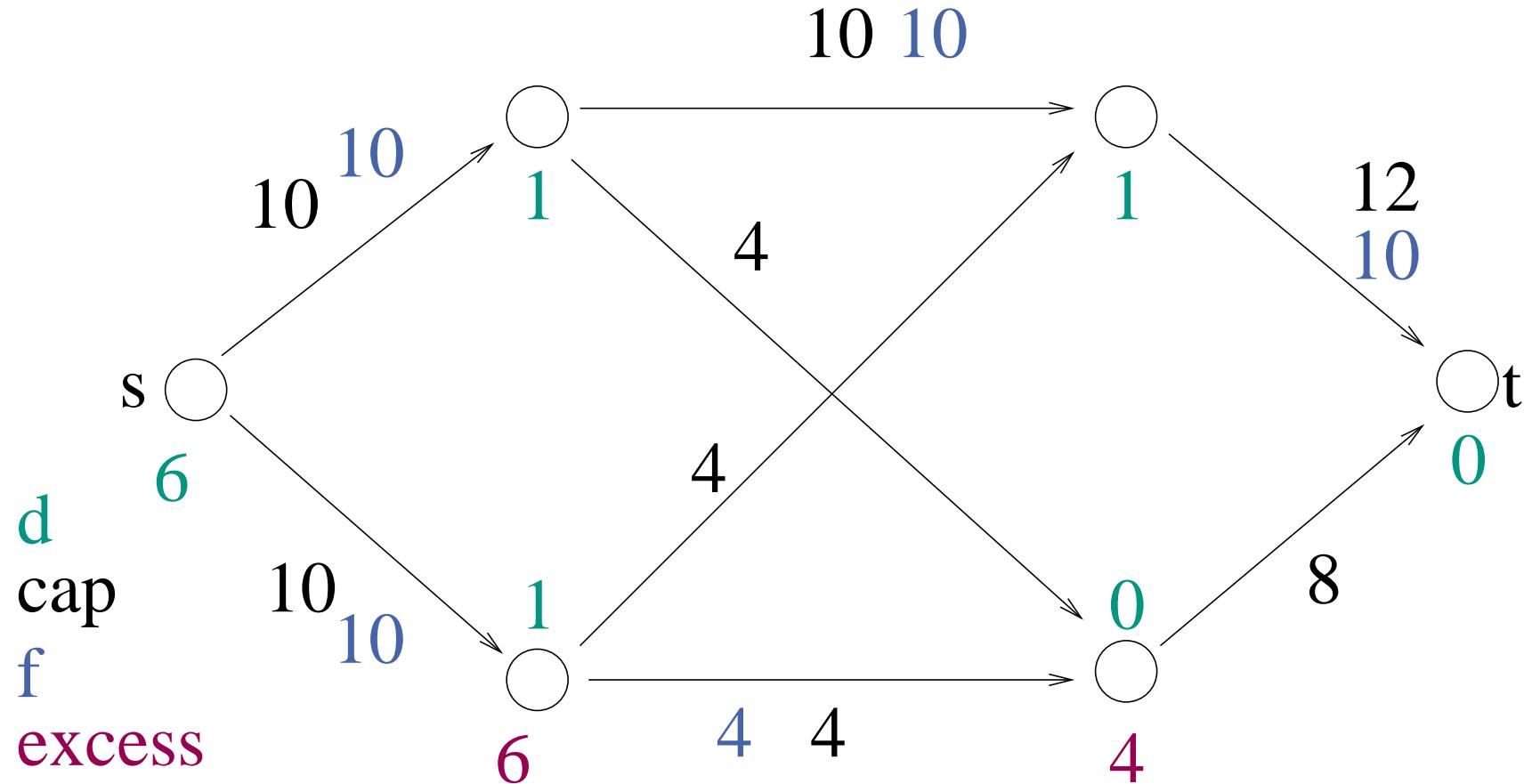
# Example



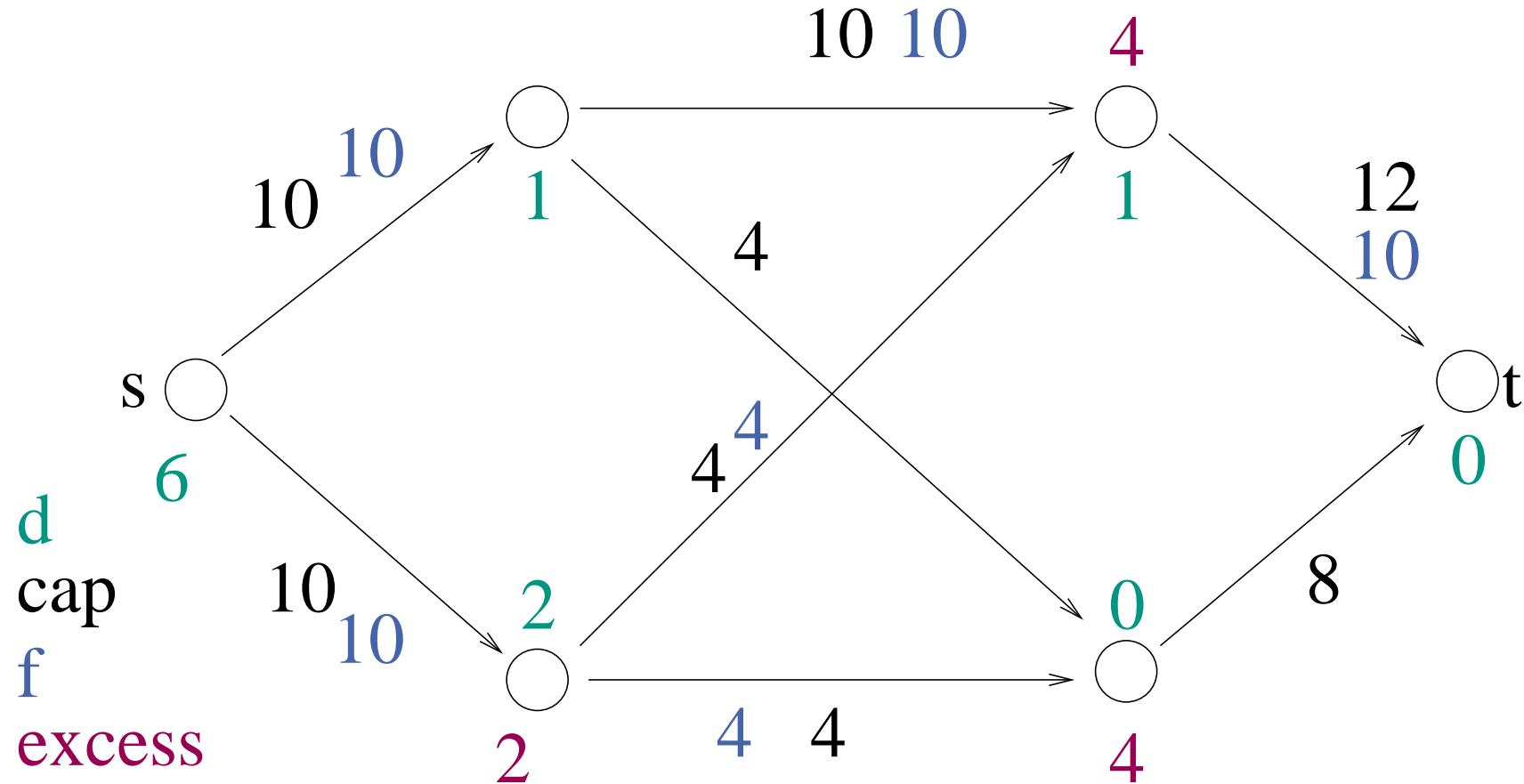
# Example



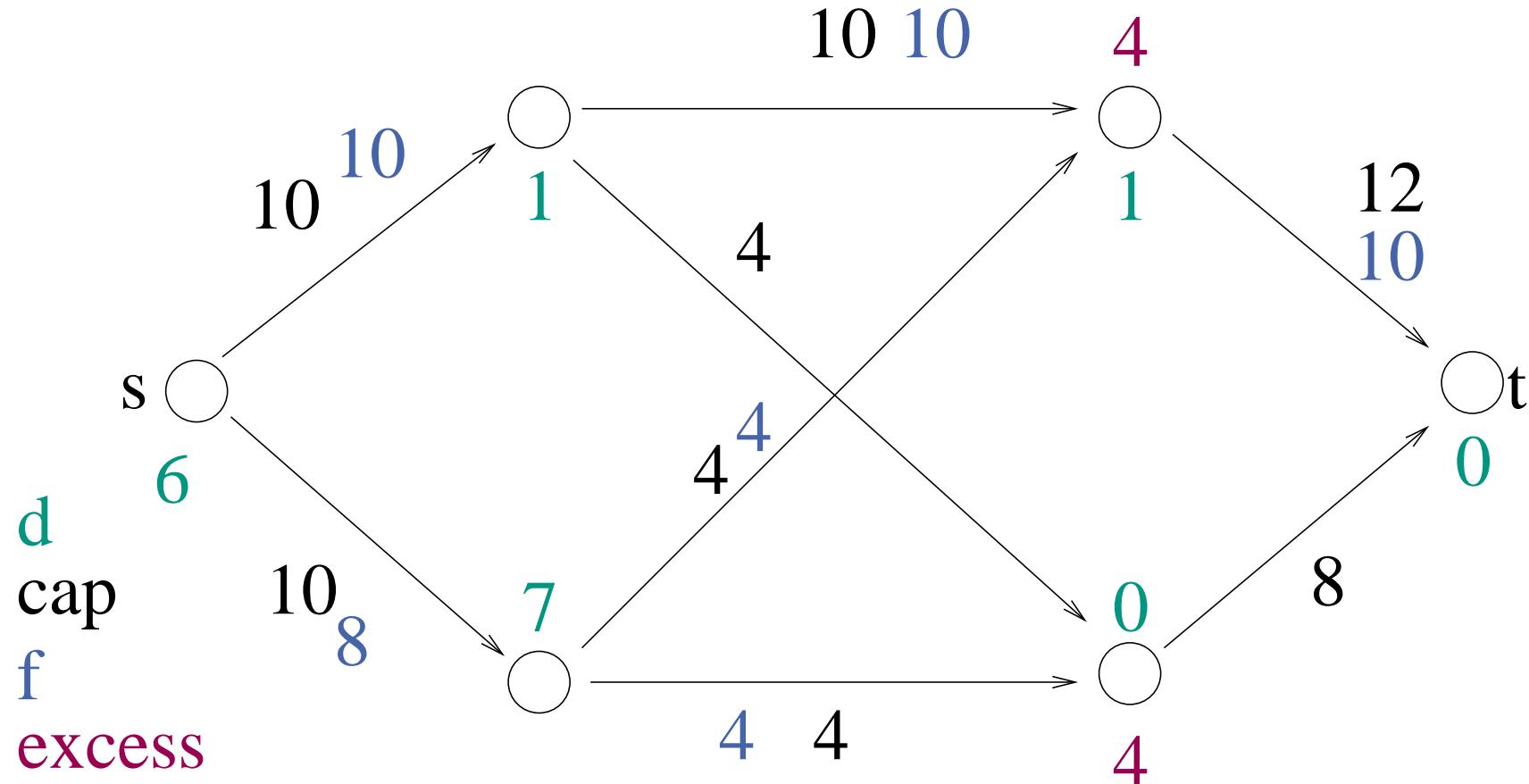
# Example



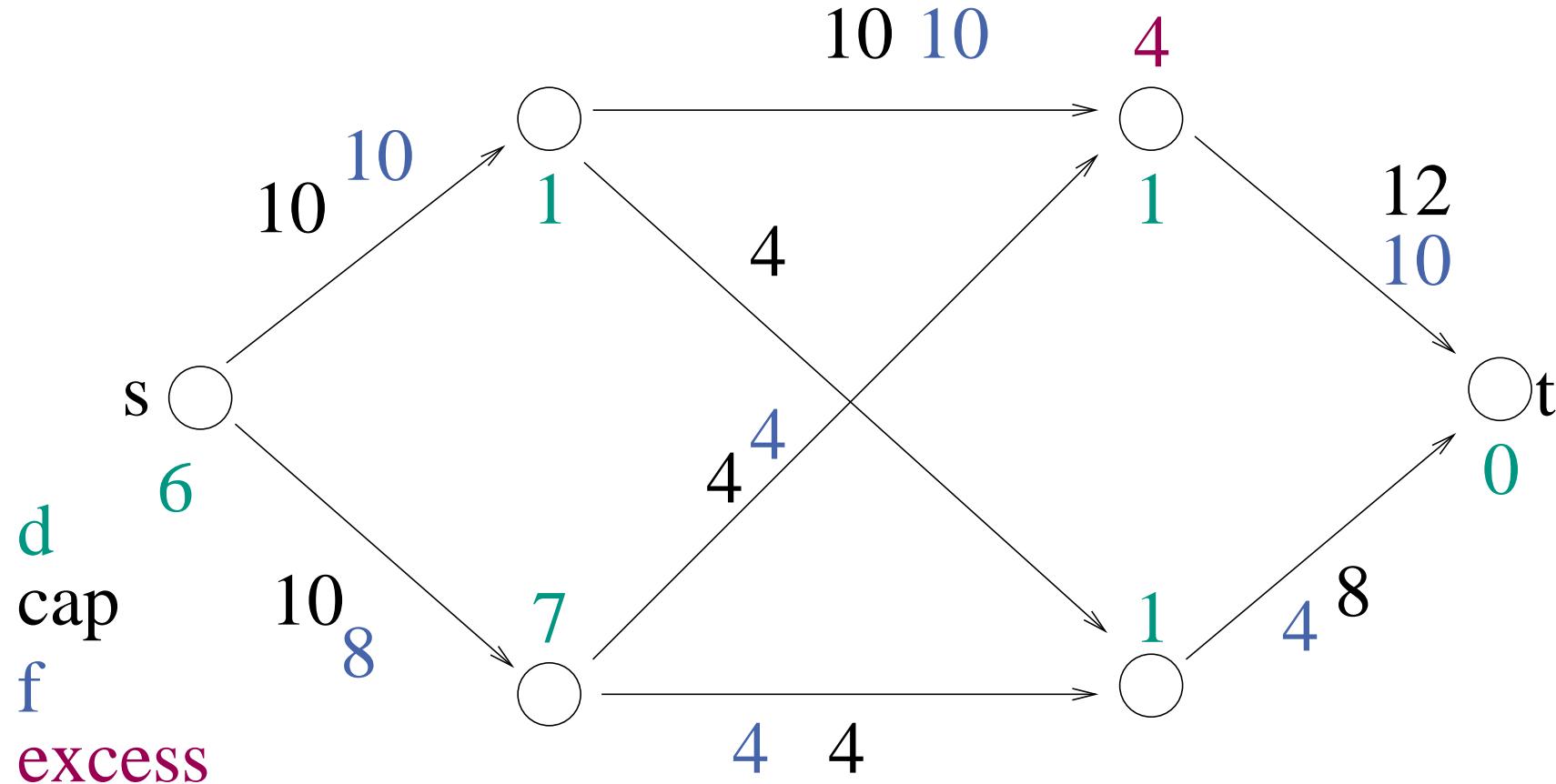
# Example



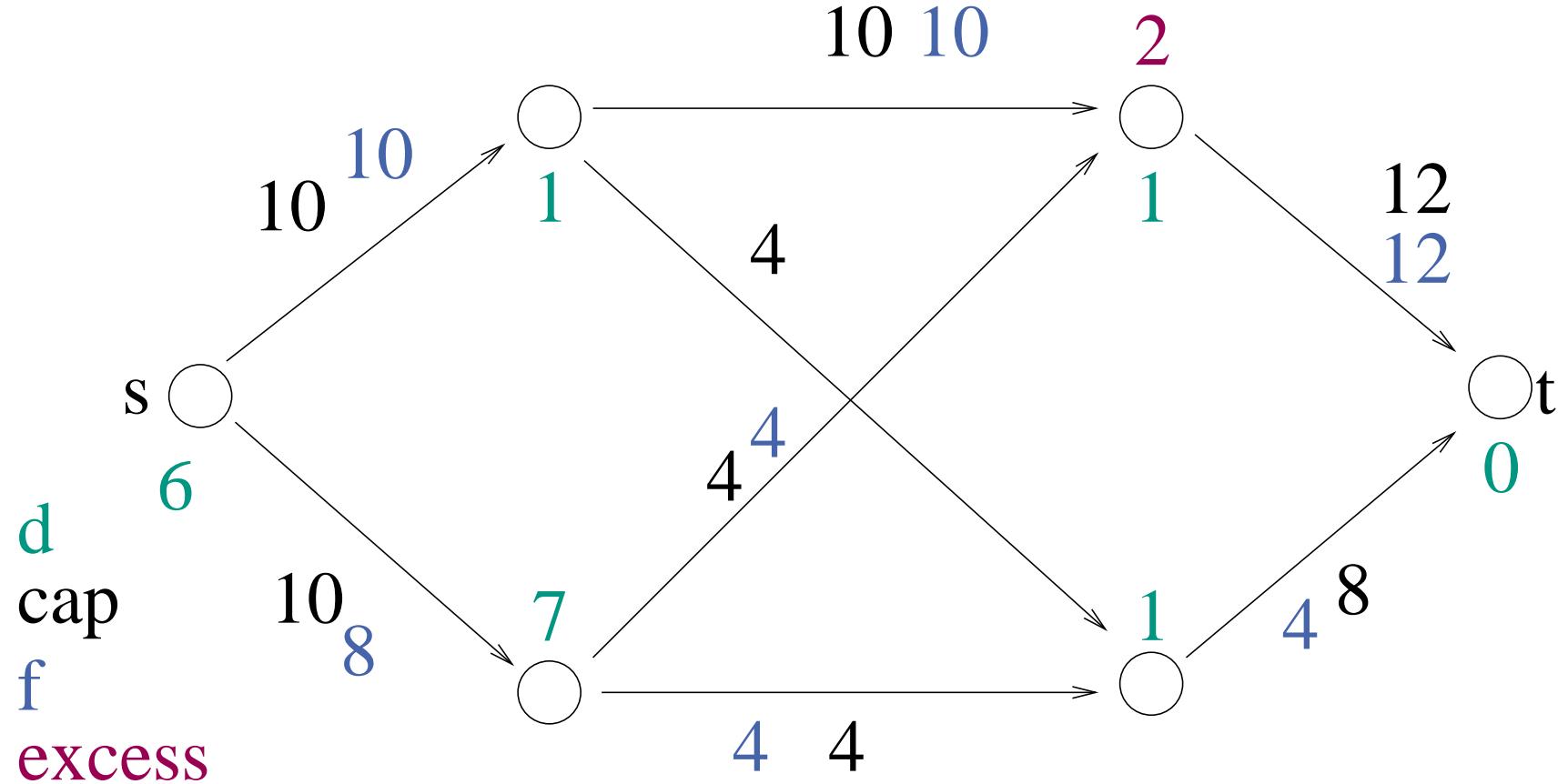
# Example



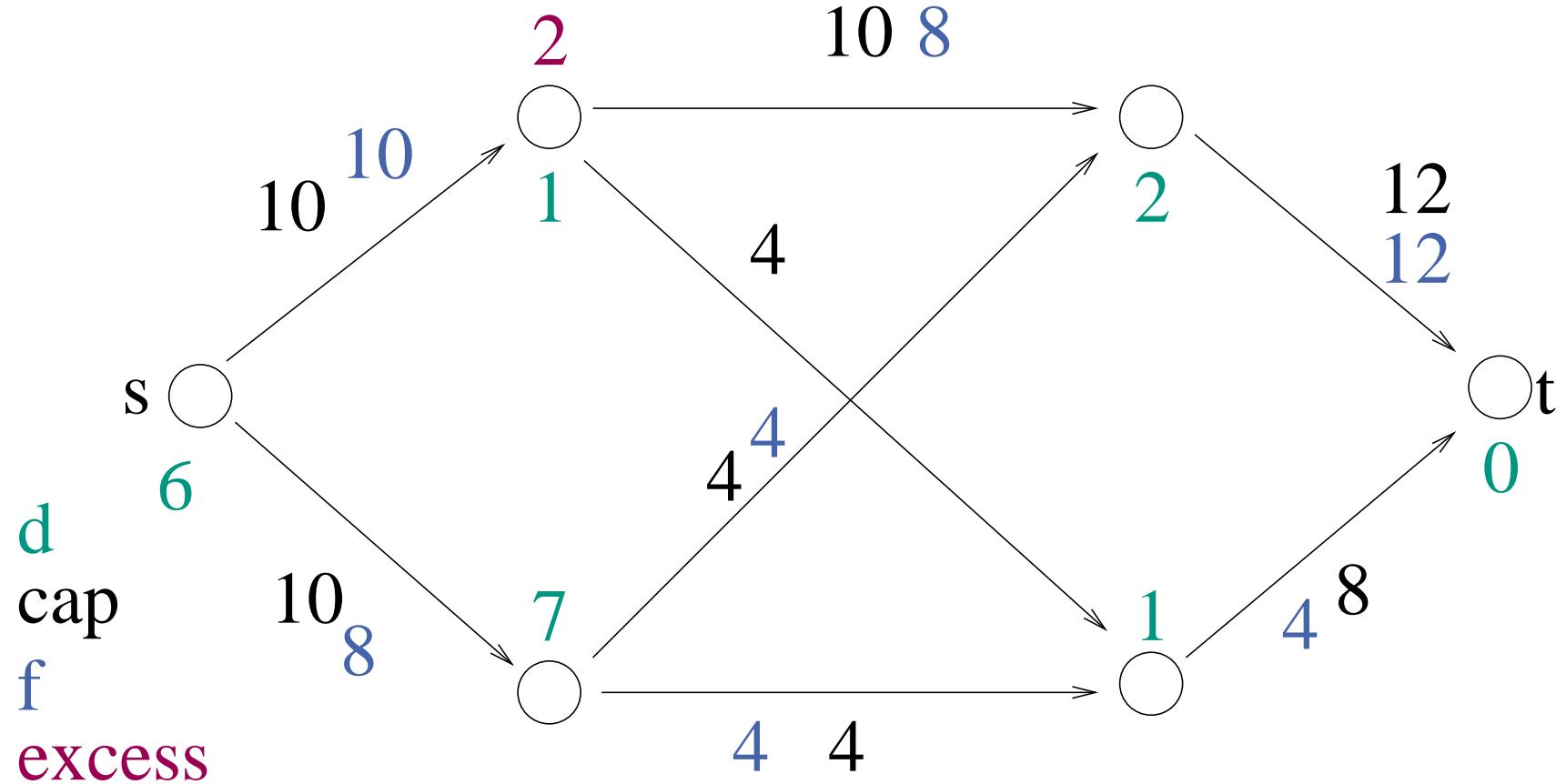
# Example



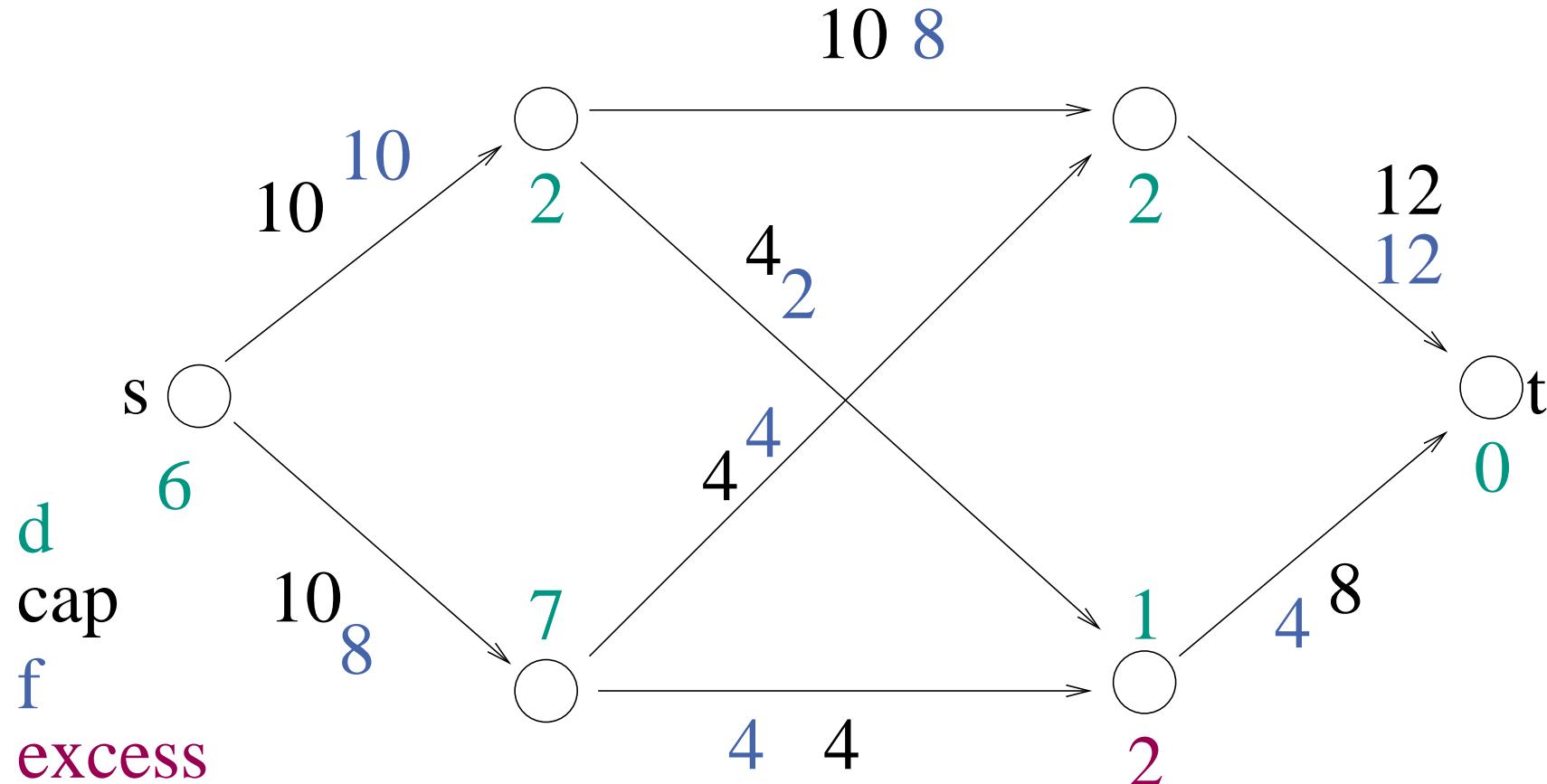
# Example



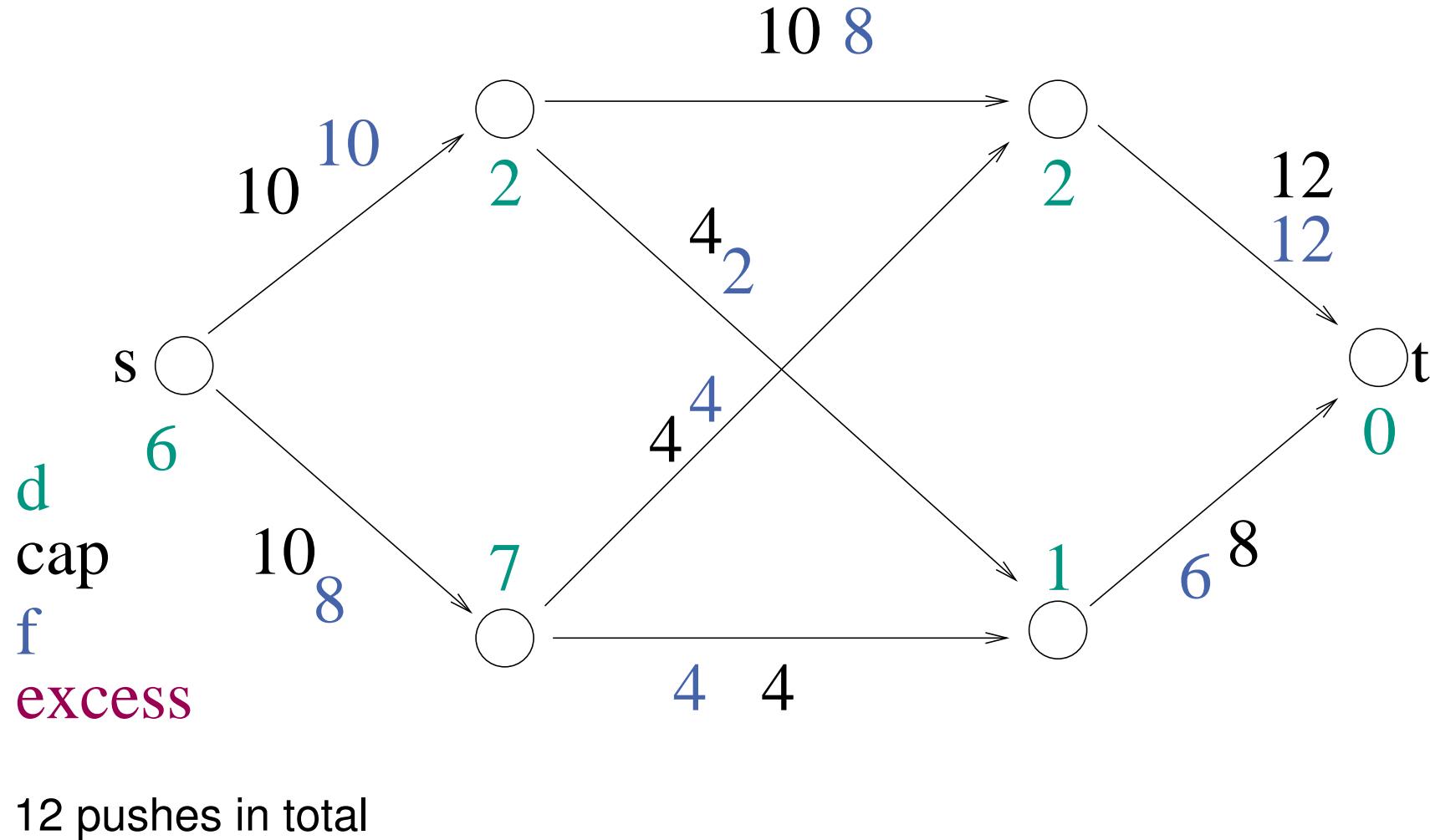
# Example



# Example



# Example



# Partial Correctness

**Lemma 3.** When `genericPreflowPush` terminates  
 $f$  is a **maximal flow**.

*Beweis.*

$f$  is a **flow** since  $\forall v \in V \setminus \{s, t\} : \text{excess}(v) = 0$ .

To show that  $f$  is **maximal**, it suffices to show that

$\nexists$  path  $p = \langle s, \dots, t \rangle \in G_f$  (Max-Flow Min-Cut Theorem):

Since  $d(s) = n, d(t) = 0$ ,  $p$  would have to contain steep edges.

That would be a contradiction. □

**Lemma 4.** For any cut  $(S, T)$ ,

$$\sum_{u \in S} \text{excess}(u) = \sum_{e \in E \cap (T \times S)} f(e) - \sum_{e \in E \cap (S \times T)} f(e),$$

**Proof:**

$$\sum_{u \in S} \text{excess}(u) = \sum_{u \in S} \left( \sum_{(v,u) \in E} f((v,u)) - \sum_{(u,v) \in E} f((u,v)) \right)$$

Contributions of edge  $e$  to sum:

$S$  to  $T$ :  $-f(e)$

$T$  to  $S$ :  $f(e)$

within  $S$ :  $f(e) - f(e) = 0$

within  $T$ : 0

■

**Lemma 5.**

$$\forall \text{ active nodes } v : \text{excess}(v) > 0 \Rightarrow \exists \text{ path } \langle v, \dots, s \rangle \in G_f$$

Intuition: what got there can always go back.

*Beweis.*  $S := \{u \in V : \exists \text{ path } \langle v, \dots, u \rangle \in G_f\}$ ,  $T := V \setminus S$ . Then

$$\sum_{u \in S} \text{excess}(u) = \sum_{e \in E \cap (T \times S)} f(e) - \sum_{e \in E \cap (S \times T)} f(e),$$

$\forall (u, w) \in E_f : u \in S \Rightarrow w \in S$  by Def. of  $G_f$ ,  $S$

$\Rightarrow \forall e = (u, w) \in E \cap (T \times S) : f(e) = 0$  Otherwise  $(w, u) \in E_f$

Hence,  $\sum_{u \in S} \text{excess}(u) \leq 0$

Only the negative excess of  $s$  can outweigh  $\text{excess}(v) > 0$ .

Hence  $s \in S$ . □

**Lemma 6.**

$$\forall v \in V : d(v) < 2n$$

*Beweis.*

Suppose  $v$  is lifted to  $d(v) = 2n$ .

By the Lemma 2, there is a (simple) path  $p$  to  $s$  in  $G_f$ .

$p$  has at most  $n - 1$  nodes

$$d(s) = n.$$

Hence  $d(v) < 2n$ . Contradiction (no steep edges). □

**Lemma 7.** # Relabel operations  $\leq 2n^2$

*Beweis.*  $d(v) \leq 2n$ , i.e.,  $v$  is relabeled at most  $2n$  times.

Hence, at most  $|V| \cdot 2n = 2n^2$  relabel operations. □

**Lemma 8.**  $\# \text{saturating pushes} \leq nm$ 

*Beweis.*

We show that there are **at most  $n$  sat. pushes** over any edge

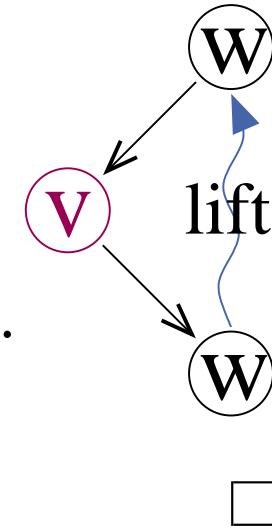
$$e = (v, w).$$

A saturating push( $e, \delta$ ) **removes  $e$**  from  $E_f$ .

Only a **push on  $(w, v)$**  can **reinsert  $e$**  into  $E_f$ .

For this to happen,  $w$  must be **lifted** at least two levels.

Hence, at most  $2n/2 = n$  saturating pushes over  $(v, w)$



**Lemma 9.**  $\# \text{nonsaturating pushes} = O(n^2m)$

if  $\delta = \min \left\{ \text{excess}(v), c_e^f \right\}$

for *arbitrary* node and edge selection rules.  
*(arbitrary-preflow-push)*

*Beweis.*  $\Phi := \sum_{\{v: v \text{ is active}\}} d(v).$  (Potential)

$\Phi = 0$  initially **and** at the end (no active nodes left!)

| Operation          | $\Delta(\Phi)$ | How many times? | Total effect |
|--------------------|----------------|-----------------|--------------|
| relabel            | 1              | $\leq 2n^2$     | $\leq 2n^2$  |
| saturating push    | $\leq 2n$      | $\leq nm$       | $\leq 2n^2m$ |
| nonsaturating push | $\leq -1$      |                 |              |

$\Phi \geq 0$  always. □

# Searching for Eligible Edges

Every node  $v$  maintains a `currentEdge` pointer to its sequence of outgoing edges in  $G_f$ .

**invariant** no edge  $e = (v, w)$  to the left of `currentEdge` is eligible

Reset `currentEdge` at a relabel  $(\leq 2n \times)$

Invariant cannot be violated by a push over a reverse edge  $e' = (w, v)$  since this only happens when  $e'$  is downward, i.e.,  $e$  is upward and hence not eligible.

**Lemma 10.**

*Total cost for searching*  $\leq \sum_{v \in V} 2n \cdot \text{degree}(v) = 4nm = \mathcal{O}(nm)$

**Satz 11.** *Arbitrary Preflow Push finds a maximum flow in time  $O(n^2m)$ .*

*Beweis.*

Lemma 3: partial correctness

Initialization in time  $O(n + m)$ .

Maintain set (e.g., stack, FIFO) of active nodes.

Use reverse edge pointers to implement push.

Lemma 7:  $2n^2$  relabel operations

Lemma 8:  $nm$  saturating pushes

Lemma 9:  $O(n^2m)$  nonsaturating pushes

Lemma 10:  $O(nm)$  search time for eligible edges

---

Total time  $O(n^2m)$

□

## FIFO Preflow push

Examine a node: Saturating pushes until nonsaturating push or relabel.

Examine all nodes in phases (or use FIFO queue).

**Theorem:** time  $O(n^3)$

**Proof:** not here

## Highest Level Preflow Push

Always select active nodes that **maximize  $d(v)$**

Use **bucket priority queue** (insert, increaseKey, deleteMax)

not monotone (!) but **relabels** “pay” for scan operations

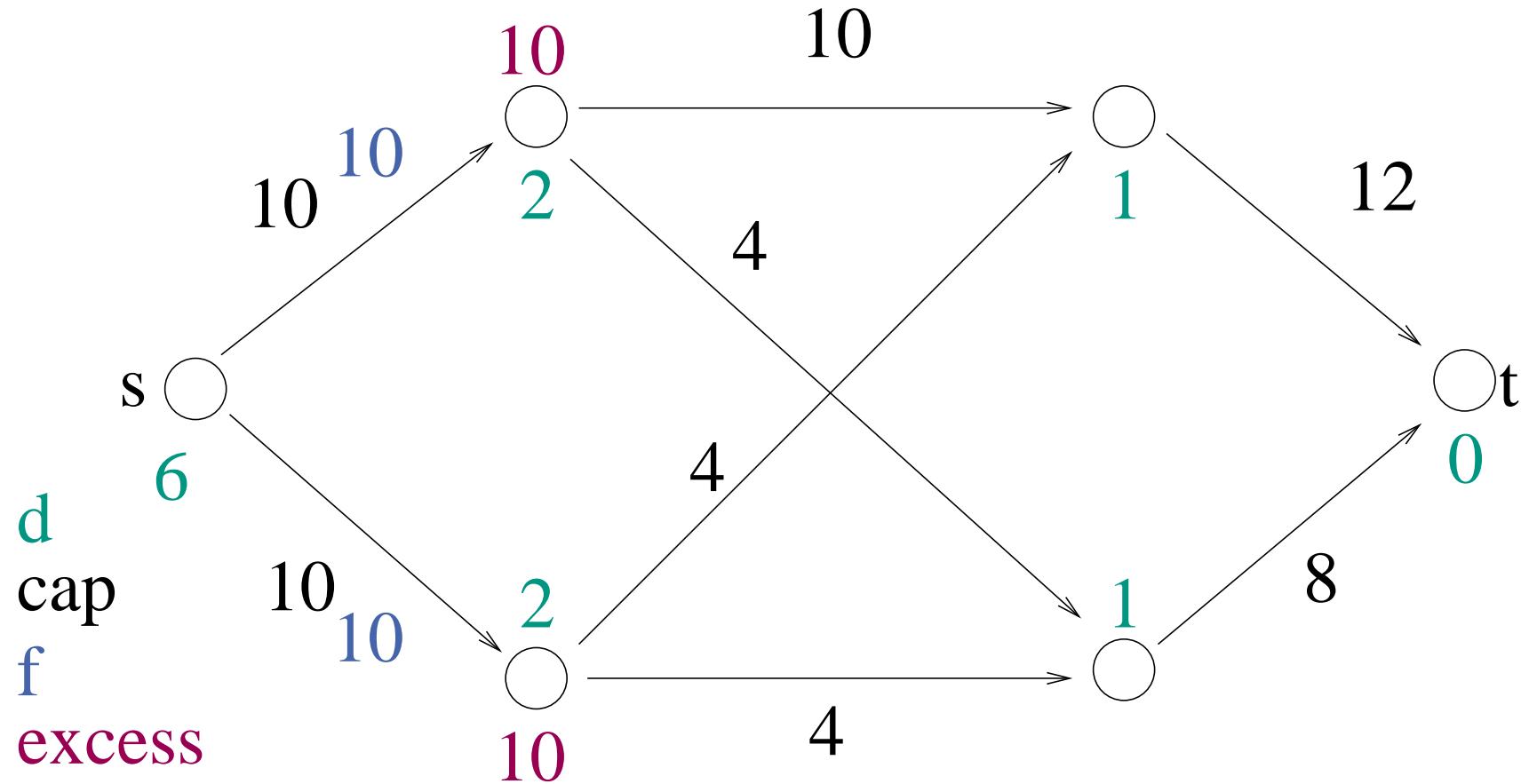
**Lemma 12.** *At most  $n^2\sqrt{m}$  nonsaturating pushes.*

*Beweis.* later

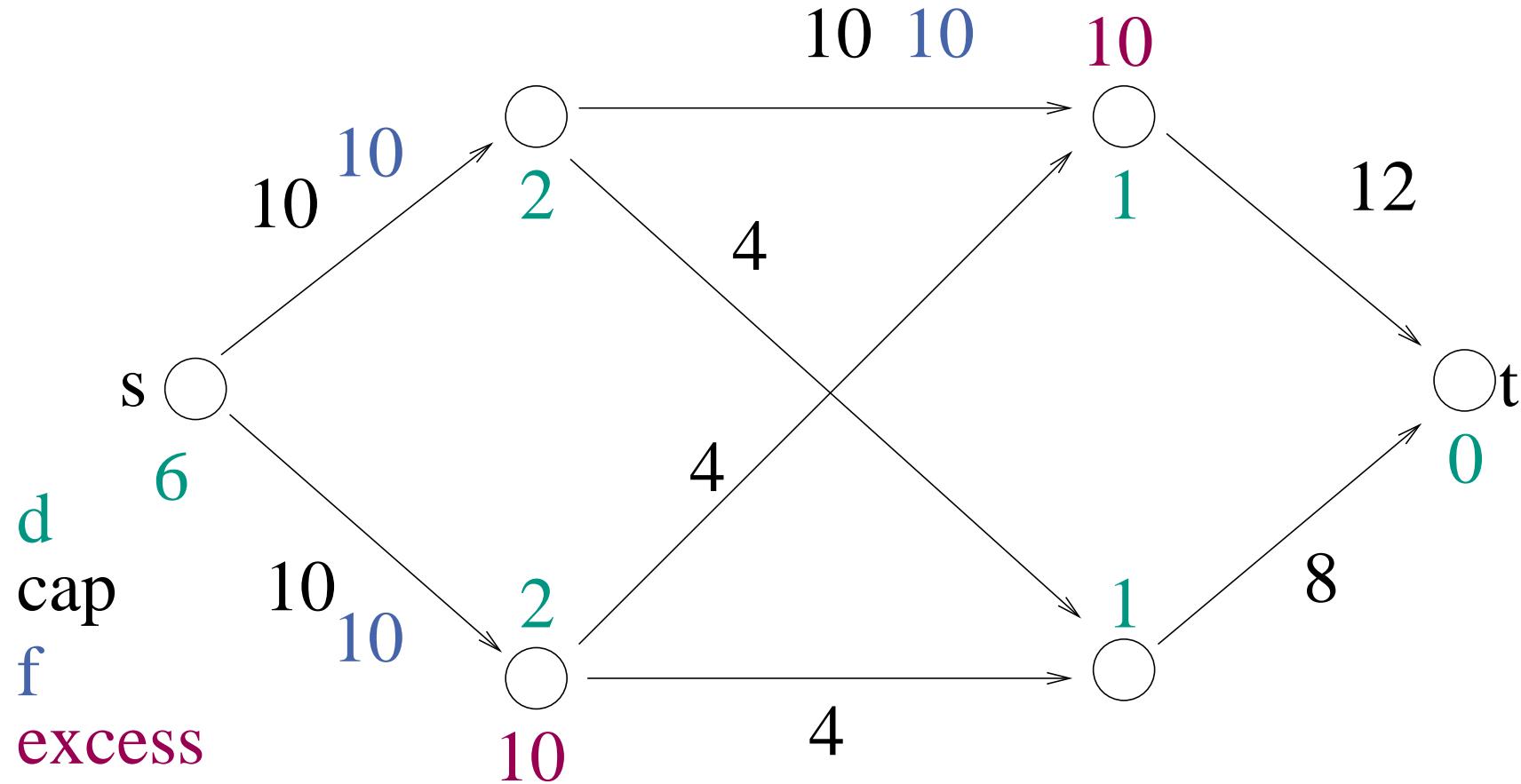
□

**Satz 13.** *Highest Level Preflow Push finds a maximum flow in time  $O(n^2\sqrt{m})$ .*

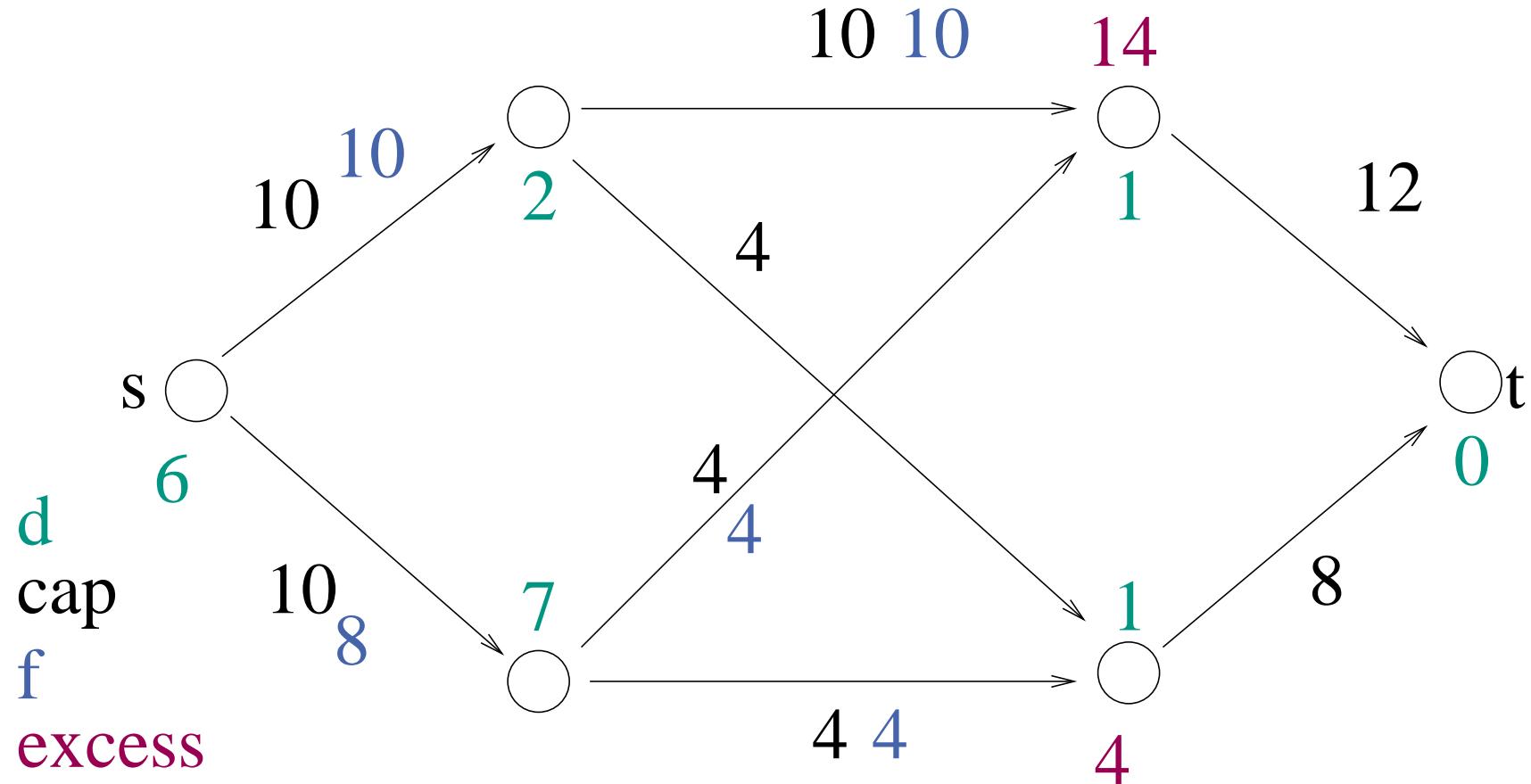
# Example



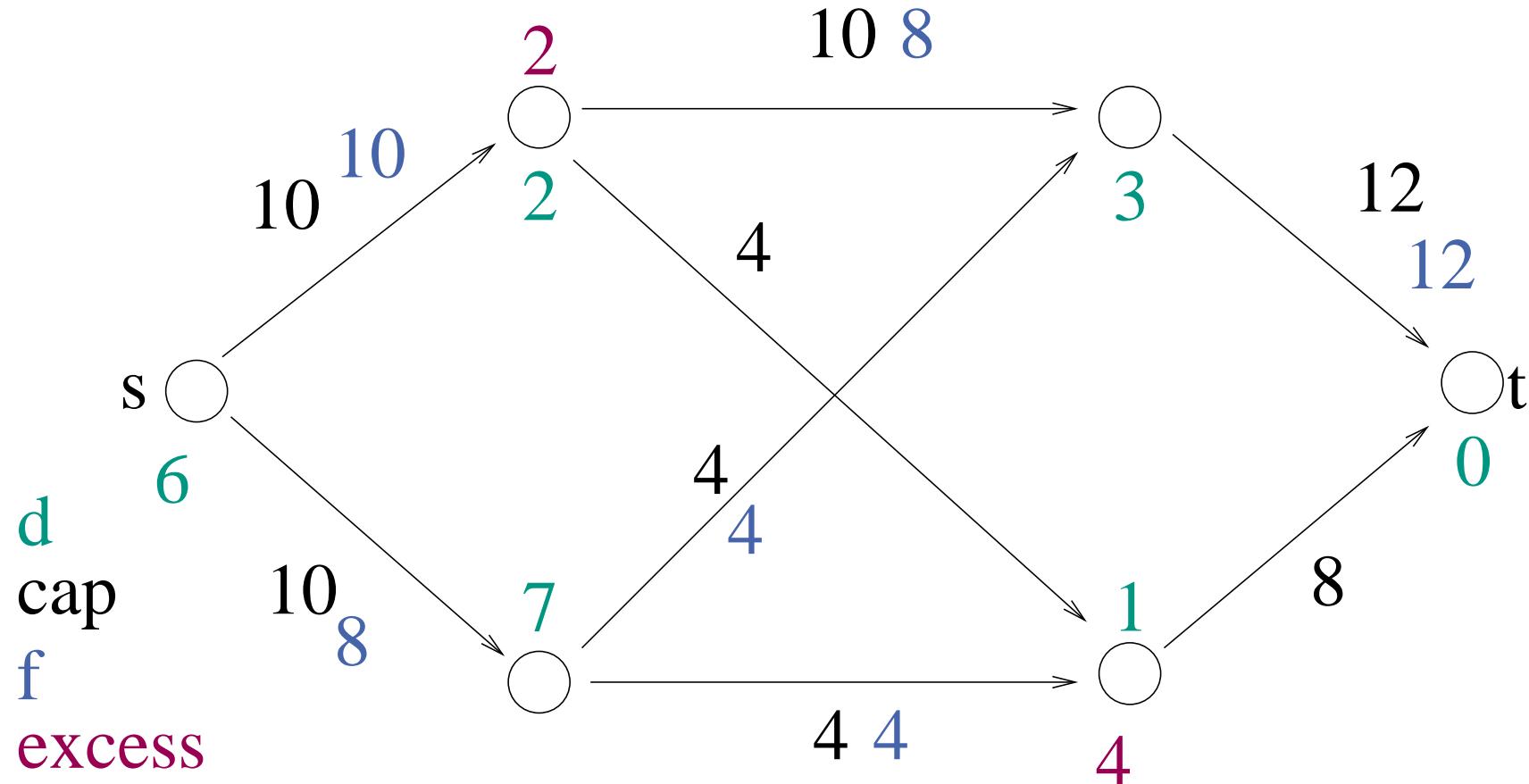
# Example



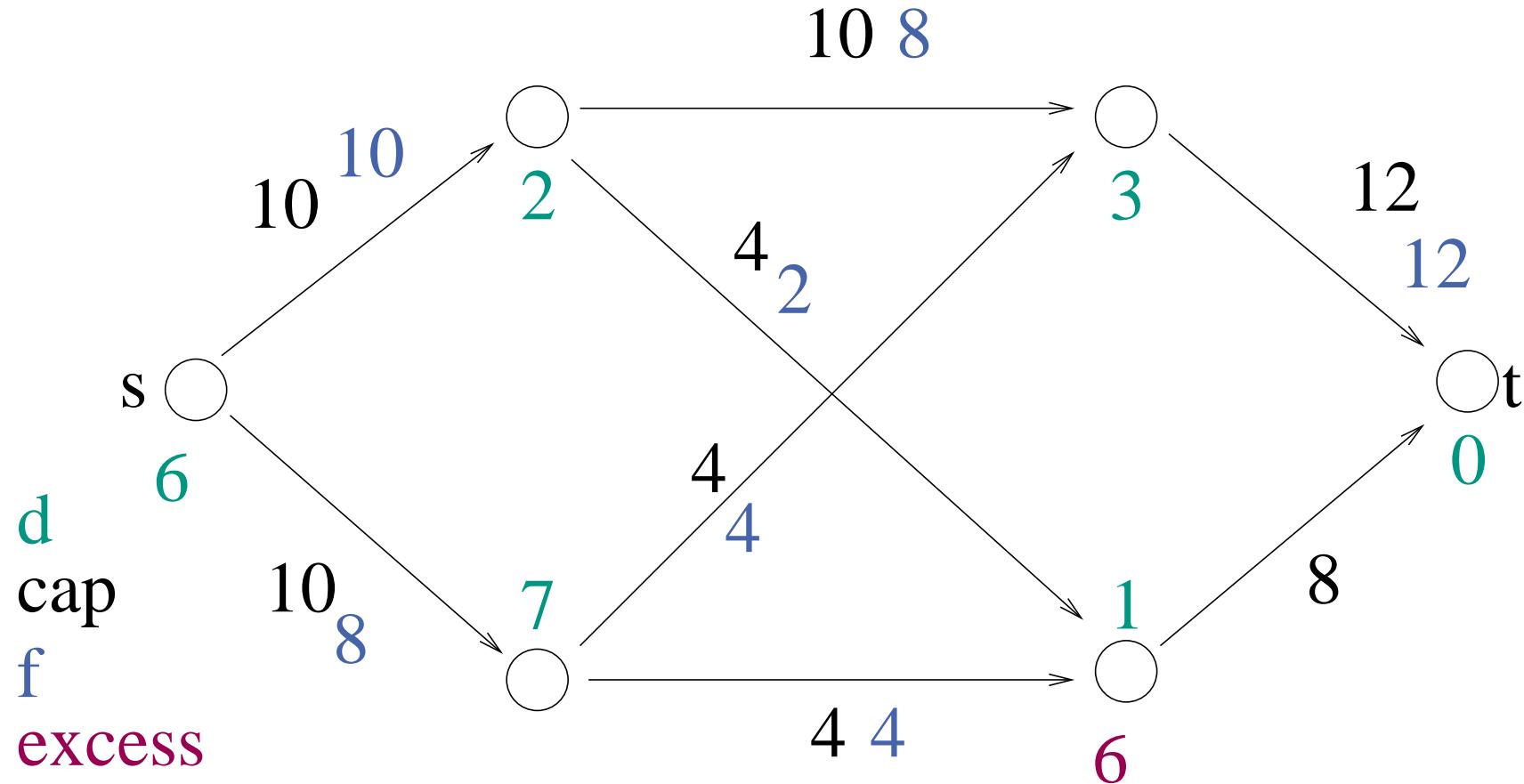
# Example



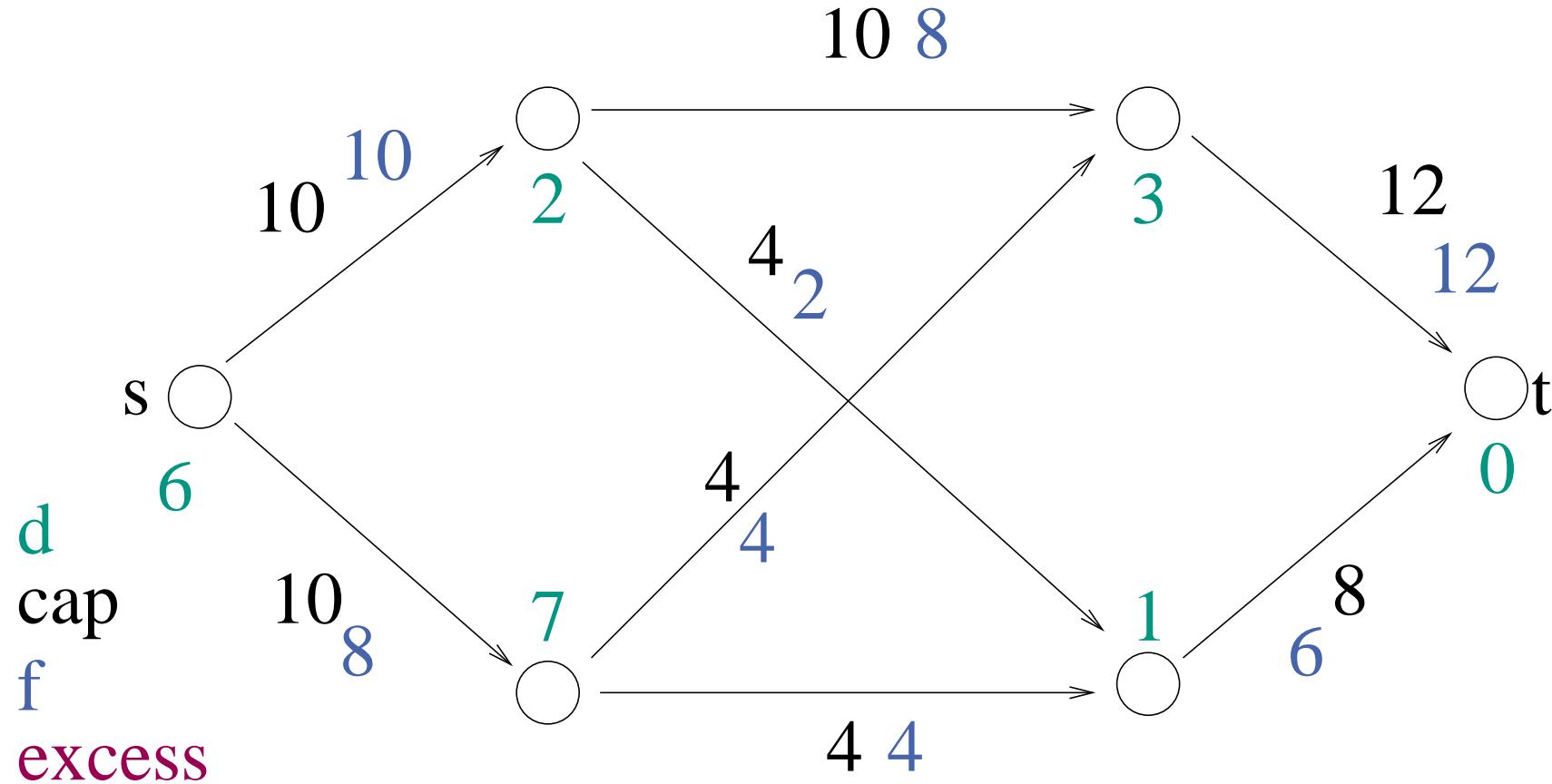
# Example



# Example



# Example



9 pushes in total, 3 less than before



# Proof of Lemma 12

$$K := \sqrt{m}$$

tuning parameter

$$d'(v) := \frac{|\{w : d(w) \leq d(v)\}|}{K}$$

scaled number of dominated nodes

$$\Phi := \sum_{\{v : v \text{ is active}\}} d'(v).$$

(Potential)

$$d^* := \max \{d(v) : v \text{ is active}\}$$

(highest level)

**phase**:= all pushes between two consecutive changes of  $d^*$

**expensive phase**: more than  $K$  pushes

**cheap** phase: otherwise



## Claims:

1.  $\leq 4n^2K$  nonsaturating pushes in all cheap phases together
2.  $\Phi \geq 0$  always,  $\Phi \leq n^2/K$  initially (obvious)
3. a relabel or saturating push increases  $\Phi$  by at most  $n/K$ .
4. a nonsaturating push does not increase  $\Phi$ .
5. an expensive phase with  $Q \geq K$  nonsaturating pushes decreases  $\Phi$  by at least  $Q$ .

Lemma 7 + Lemma 8 + 2. + 3. + 4.  $\Rightarrow$

$$\text{total possible decrease} \leq (2n^2 + nm) \frac{n}{K} + \frac{n^2}{K}$$

This + 5.  $\leq \frac{2n^3 + n^2 + mn^2}{K}$  nonsaturating pushes in expensive phases

This + 1.  $\leq \frac{2n^3 + n^2 + mn^2}{K} + 4n^2K = O(n^2\sqrt{m})$  nonsaturating pushes overall for  $K = \sqrt{m}$

| Operation | Amount |
|-----------|--------|
| Relabel   | $2n^2$ |
| Sat.push  | $nm$   |



## Claims:

1.  $\leq 4n^2K$  nonsaturating pushes in all cheap phases together

We first show that there are at most  $4n^2$  phases  
(changes of  $d^* = \max \{d(v) : v \text{ is active}\}$ ).

$d^* = 0$  initially,  $d^* \geq 0$  always.

Only **relabel** operations increase  $d^*$ , i.e.,

$\leq 2n^2$  increases by Lemma 7 and hence

$\leq 2n^2$  decreases

---

$\leq 4n^2$  changes overall

By definition of a cheap phase, it has at most  $K$  pushes.



## Claims:

1.  $\leq 4n^2K$  nonsaturating pushes in all cheap phases together
2.  $\Phi \geq 0$  always,  $\Phi \leq n^2/K$  initially (obvious)
3. a relabel or saturating push increases  $\Phi$  by at most  $n/K$ .

Let  $v$  denote the relabeled or activated node.

$$d'(v) := \frac{|\{w : d(w) \leq d(v)\}|}{K} \leq \frac{n}{K}$$

A relabel of  $v$  can increase only the  $d'$ -value of  $v$ .

A saturating push on  $(u, w)$  may activate only  $w$ .



## Claims:

1.  $\leq 4n^2K$  nonsaturating pushes in all cheap phases together
2.  $\Phi \geq 0$  always,  $\Phi \leq n^2/K$  initially (obvious)
3. a relabel or saturating push increases  $\Phi$  by at most  $n/K$ .
4. a nonsaturating push does not increase  $\Phi$ .

$v$  is deactivated ( $\text{excess}(v)$  is now 0)

$w$  may be activated

but  $d'(w) \leq d'(v)$  (we do not push flow away from the sink)



## Claims:

1.  $\leq 4n^2K$  nonsaturating pushes in all cheap phases together
2.  $\Phi \geq 0$  always,  $\Phi \leq n^2/K$  initially (obvious)
3. a relabel or saturating push increases  $\Phi$  by at most  $n/K$ .
4. a nonsaturating push does not increase  $\Phi$ .
5. an expensive phase with  $Q \geq K$  nonsaturating pushes decreases  $\Phi$  by at least  $Q$ .

During a phase  $d^*$  remains constant

Each nonsat. push decreases the number of active nodes at level  $d^*$

Hence,  $|\{w : d(w) = d^*\}| \geq Q \geq K$  during an expensive phase

Each nonsat. push across  $(v, w)$  decreases  $\Phi$  by

$$\geq d'(v) - d'(w) \geq |\{w : d(w) = d^*\}| / K \geq K / K = 1$$

■



## Claims:

1.  $\leq 4n^2K$  nonsaturating pushes in all cheap phases together
2.  $\Phi \geq 0$  always,  $\Phi \leq n^2/K$  initially (obvious)
3. a relabel or saturating push increases  $\Phi$  by at most  $n/K$ .
4. a nonsaturating push does not increase  $\Phi$ .
5. an expensive phase with  $Q \geq K$  nonsaturating pushes decreases  $\Phi$  by at least  $Q$ .

Lemma 7 + Lemma 8 + 2. + 3. + 4.  $\Rightarrow$

$$\text{total possible decrease} \leq (2n^2 + nm) \frac{n}{K} + \frac{n^2}{K}$$

This + 5.  $\leq \frac{2n^3 + n^2 + mn^2}{K}$  nonsaturating pushes in expensive phases

This + 1.  $\leq \frac{2n^3 + n^2 + mn^2}{K} + 4n^2K = O(n^2\sqrt{m})$  nonsaturating pushes overall for  $K = \sqrt{m}$

| Operation | Amount |
|-----------|--------|
| Relabel   | $2n^2$ |
| Sat.push  | $nm$   |



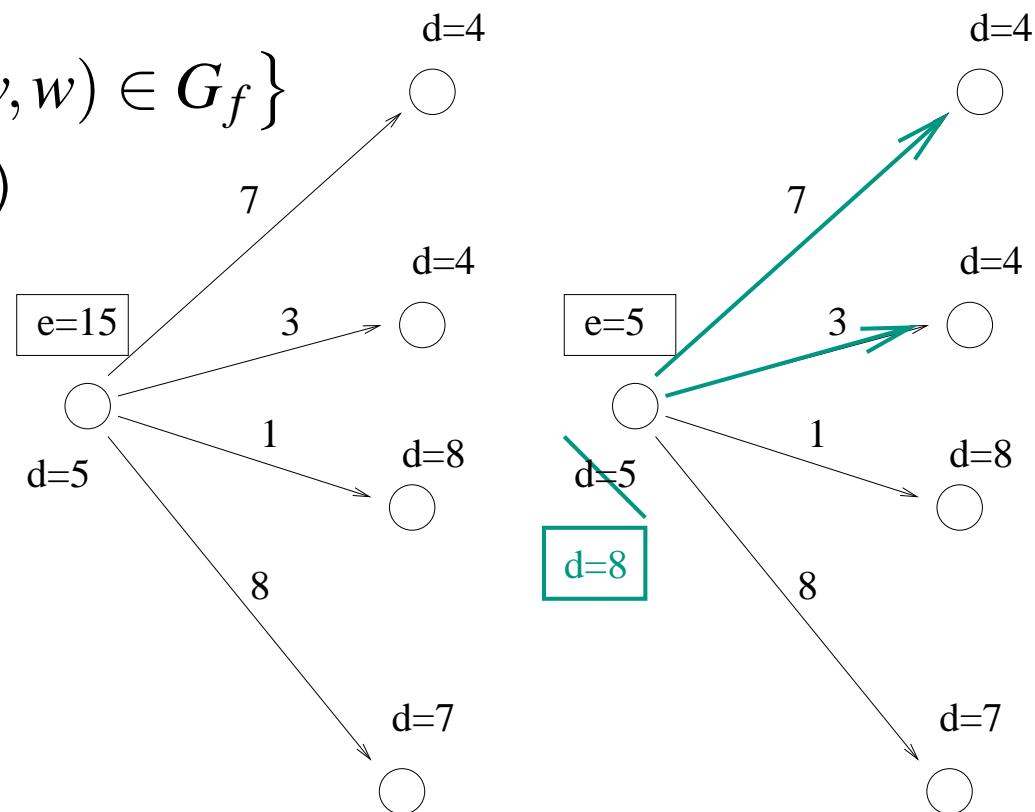
# Heuristic Improvements

Naive algorithm has **best case**  $\Omega(n^2)$ . Why? We can do better.

aggressive local relabeling:

$$d(v) := 1 + \min \{d(w) : (v, w) \in G_f\}$$

(like a sequence of relabels)



# Heuristic Improvements

Naive algorithm has **best case**  $\Omega(n^2)$ . Why? We can do better.

**aggressive local relabeling:**  $d(v) := 1 + \min \{d(w) : (v, w) \in G_f\}$   
(like a sequence of relabels)

**global relabeling:** (initially and every  $O(m)$  edge inspections):  
 $d(v) := G_f.\text{reverseBFS}(t)$  for nodes that can reach  $t$  in  $G_f$ .

Special treatment of nodes with  $d(v) \geq n$ . (**Returning flow** is easy)

**Gap Heuristics.** No node can connect to  $t$  across an empty level:

**if**  $\{v : d(v) = i\} = \emptyset$  **then foreach**  $v$  with  $d(v) > i$  **do**  $d(v) := n$

# Experimental results

We use four classes of graphs:

- Random:  $n$  nodes,  $2n + m$  edges; all edges  $(s, v)$  and  $(v, t)$  exist
- Cherkassky and Goldberg (1997) (two graph classes)
- Ahuja, Magnanti, Orlin (1993)



## Timings: Random Graphs

| Rule | BASIC | Ln    | LRH   | GRH  | GAP  | LEDA |
|------|-------|-------|-------|------|------|------|
| FF   | 5.84  | 6.02  | 4.75  | 0.07 | 0.07 | —    |
|      | 33.32 | 33.88 | 26.63 | 0.16 | 0.17 | —    |
| HL   | 6.12  | 6.3   | 4.97  | 0.41 | 0.11 | 0.07 |
|      | 27.03 | 27.61 | 22.22 | 1.14 | 0.22 | 0.16 |
| MF   | 5.36  | 5.51  | 4.57  | 0.06 | 0.07 | —    |
|      | 26.35 | 27.16 | 23.65 | 0.19 | 0.16 | —    |

$n \in \{1000, 2000\}$ ,  $m = 3n$

FF=FIFO node selection, HL=highest level, MF=modified FIFO

Ln= $d(v) \geq n$  is special,

LRH=local relabeling heuristic, GRH=global relabeling heuristics



## Timings: CG1

| Rule | BASIC | Ln    | LRH   | GRH  | GAP  | LEDA |
|------|-------|-------|-------|------|------|------|
| FF   | 3.46  | 3.62  | 2.87  | 0.9  | 1.01 | —    |
|      | 15.44 | 16.08 | 12.63 | 3.64 | 4.07 | —    |
| HL   | 20.43 | 20.61 | 20.51 | 1.19 | 1.33 | 0.8  |
|      | 192.8 | 191.5 | 193.7 | 4.87 | 5.34 | 3.28 |
| MF   | 3.01  | 3.16  | 2.3   | 0.89 | 1.01 | —    |
|      | 12.22 | 12.91 | 9.52  | 3.65 | 4.12 | —    |

$n \in \{1000, 2000\}$ ,  $m = 3n$

FF=FIFO node selection, HL=highest level, MF=modified FIFO

Ln= $d(v) \geq n$  is special,

LRH=local relabeling heuristic, GRH=global relabeling heuristics



## Timings: CG2

| Rule | BASIC | Ln    | LRH   | GRH  | GAP  | LEDA |
|------|-------|-------|-------|------|------|------|
| FF   | 50.06 | 47.12 | 37.58 | 1.76 | 1.96 | —    |
|      | 239   | 222.4 | 177.1 | 7.18 | 8    | —    |
| HL   | 42.95 | 41.5  | 30.1  | 0.17 | 0.14 | 0.08 |
|      | 173.9 | 167.9 | 120.5 | 0.36 | 0.28 | 0.18 |
| MF   | 45.34 | 42.73 | 37.6  | 0.94 | 1.07 | —    |
|      | 198.2 | 186.8 | 165.7 | 4.11 | 4.55 | —    |

$n \in \{1000, 2000\}$ ,  $m = 3n$

FF=FIFO node selection, HL=highest level, MF=modified FIFO

Ln= $d(v) \geq n$  is special,

LRH=local relabeling heuristic, GRH=global relabeling heuristics



## Timings: AMO

| Rule | BASIC | Ln    | LRH     | GRH    | GAP    | LEDA |
|------|-------|-------|---------|--------|--------|------|
| FF   | 12.61 | 13.25 | 1.17    | 0.06   | 0.06   | —    |
|      | 55.74 | 58.31 | 5.01    | 0.1399 | 0.1301 | —    |
| HL   | 15.14 | 15.8  | 1.49    | 0.13   | 0.13   | 0.07 |
|      | 62.15 | 65.3  | 6.99    | 0.26   | 0.26   | 0.14 |
| MF   | 10.97 | 11.65 | 0.04999 | 0.06   | 0.06   | —    |
|      | 46.74 | 49.48 | 0.1099  | 0.1301 | 0.1399 | —    |

$n \in \{1000, 2000\}$ ,  $m = 3n$

FF=FIFO node selection, HL=highest level, MF=modified FIFO

Ln= $d(v) \geq n$  is special,

LRH=local relabeling heuristic, GRH=global relabeling heuristics



## Asymptotics, $n \in \{5000, 10000, 20000\}$

| Gen  | Rule | GRH  |       |       | GAP  |       |       | LEDA |       |       |
|------|------|------|-------|-------|------|-------|-------|------|-------|-------|
| rand | FF   | 0.16 | 0.41  | 1.16  | 0.15 | 0.42  | 1.05  | —    | —     | —     |
|      | HL   | 1.47 | 4.67  | 18.81 | 0.23 | 0.57  | 1.38  | 0.16 | 0.45  | 1.09  |
|      | MF   | 0.17 | 0.36  | 1.06  | 0.14 | 0.37  | 0.92  | —    | —     | —     |
| CG1  | FF   | 3.6  | 16.06 | 69.3  | 3.62 | 16.97 | 71.29 | —    | —     | —     |
|      | HL   | 4.27 | 20.4  | 77.5  | 4.6  | 20.54 | 80.99 | 2.64 | 12.13 | 48.52 |
|      | MF   | 3.55 | 15.97 | 68.45 | 3.66 | 16.5  | 70.23 | —    | —     | —     |
| CG2  | FF   | 6.8  | 29.12 | 125.3 | 7.04 | 29.5  | 127.6 | —    | —     | —     |
|      | HL   | 0.33 | 0.65  | 1.36  | 0.26 | 0.52  | 1.05  | 0.15 | 0.3   | 0.63  |
|      | MF   | 3.86 | 15.96 | 68.42 | 3.9  | 16.14 | 70.07 | —    | —     | —     |
| AMO  | FF   | 0.12 | 0.22  | 0.48  | 0.11 | 0.24  | 0.49  | —    | —     | —     |
|      | HL   | 0.25 | 0.48  | 0.99  | 0.24 | 0.48  | 0.99  | 0.12 | 0.24  | 0.52  |
|      | MF   | 0.11 | 0.24  | 0.5   | 0.11 | 0.24  | 0.48  | —    | —     | —     |

# Zusammenfassung Flows und Matchings

- Natürliche Verallgemeinerung von kürzesten Wegen:  
ein Pfad  $\rightsquigarrow$  viele Pfade
- viele Anwendungen
- “schwierigste/allgemeinsten” Graph-Probleme, die sich mit  
**kombinatorischen** Algorithmen in **Polynomialzeit** lösen lassen
- Beispiel für nichttriviale Algorithmenanalyse
- **Potentialmethode** ( $\neq$  **Knotenpotentiale**)
- Algorithm Engineering: practical case  $\neq$  worst case.  
**Heuristiken/Details/Eingabeeigenschaften** wichtig
- Datenstrukturen: bucket queues, graph representation, (dynamic trees)

# 6 Randomisierte Algorithmen

verwende Zufall(sbits) zur Beschleunigung/Vereinfachung von Algorithmen

**Las Vegas:** Ergebnis immer korrekt, Laufzeit ist Zufallsvariable.

Schon gesehen:

- quicksort
- hashing

**Monte Carlo:** Ergebnis mit bestimmter Wahrscheinlichkeit  $p$  inkorrekt.  
 $k$ -fache Wiederholung macht Fehlschlagswahrscheinlichkeit exponentiell klein ( $p^k$ ).

Mehr in der Vorlesung Randomisierte Algorithmen von Thomas Worsch

## 6.1 Sortieren – Ergebnisüberprüfung (Checking)

**Permutationseigenschaft** (Sortiertheit: trivial.)

$\langle e_1, \dots, e_n \rangle$  ist Permutation von  $\langle e'_1, \dots, e'_n \rangle$  gdw.

$$q(z) := \prod_{i=1}^n (z - \text{field}(\text{key}(e_i))) - \prod_{i=1}^n (z - \text{field}(\text{key}(e'_i))) = 0,$$

$\mathbb{F}$  sei Körper,  $\text{field} : \text{Key} \rightarrow \mathbb{F}$  sei injektiv.

Beobachtung:  $q$  hat höchstens  $n$  Nullstellen.

Auswertung an **zufälliger** Stelle  $x \in \mathbb{F}$ .

$$\mathbb{P}[q \neq 0 \wedge q(x) = 0] \leq \frac{n}{|\mathbb{F}|}$$

Monte Carlo-Algorithmus, Linearzeit.

Frage: Welchen Körper  $\mathbb{F}$  nehmen wir?

## Sort Checking II – mit Lorenz Hübschle-Schneider

Ist Folge  $E$  eine Permutation von Folge  $E'$ ?

Sei  $h$  zufällige Hash-Funktion mit Wertebereich  $0..U - 1$ ,

$$h(S) := \sum_{e \in S} h(e)$$

**Checker:** return  $h(E) = h(E')$

## Sort Checking II – mit Lorenz Hübschle-Schneider

Ist Folge  $E$  eine Permutation von Folge  $E'$ ?

Sei  $h$  zufällige Hash-Funktion mit Wertebereich  $0..U - 1$ ,

$$h(S) := \sum_{e \in S} h(e)$$

**Checker:** return  $h(E) = h(E')$

Korrekt falls  $E = E'$ .

**Fall  $E \neq E'$ :** Wir zeigen  $\mathbb{P}[h(E) = h(E')] \leq \frac{1}{U}$

Sei  $e$  ein Element, das  $k \times$  in  $E$  vorkommt und  $k' \neq k \times$  in  $E'$ .

$$h(E) = h(E') \Leftrightarrow h(E \setminus e) + kh(e) = h(E' \setminus e) + k'h(e)$$

$$\Leftrightarrow h(e) = \frac{h(E' \setminus e) - h(E \setminus e)}{k - k'} =: x$$

$\mathbb{P}[h(e) = x] \leq \frac{1}{U}$  weil  $x$  unabhängig von  $h(e)$

■

## 6.2 Hashing II

### Perfektes Hashing

Idee: mache  $h$  injektiv.

Braucht  $\Omega(n)$  bits Platz !

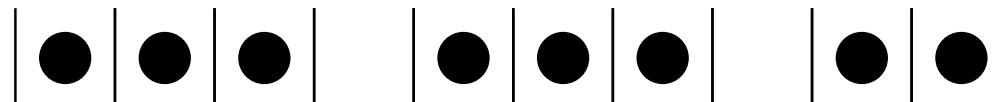
# Here: Fast Space Efficient Hashing

Represent a set of  $n$  elements (with associated information) using space  $(1 + \varepsilon)n$ .

Support operations **insert**, **delete**, **lookup**, (doall) efficiently.

Assume a truly random hash function  $h$

(Trick [Dietzfelbinger, Weidling 2005]: lässt sich rechtfertigen.)





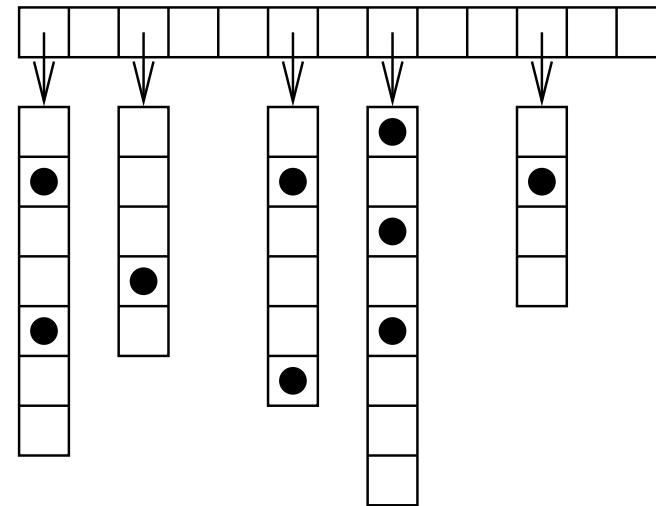
# Related Work

Linear probing:  $E[T_{\text{find}}] \approx \frac{1}{2\varepsilon^2}$

Uniform hashing:  $E[T_{\text{find}}] \approx \frac{1}{\varepsilon}$

Dynamic Perfect Hashing,  
[Dietzfelbinger et al. 94]

Worst case constant time  
for lookup but  $\varepsilon$  is not small.



Approaching the Information Theoretic Lower Bound:

[Brodnik Munro 99, Raman Rao 02]

Space  $(1 + o(1)) \times$  lower bound without associated information

[Botelho Pagh Ziviani 2007] static case.

Simple, fast,  $\approx 3$  bits/element [FiRe/FiPHa: Müller, S, Schulze, Zhou 14]

# Cuckoo Hashing

[Pagh Rodler 01]

Table of size  $(2 + \varepsilon)n$ .

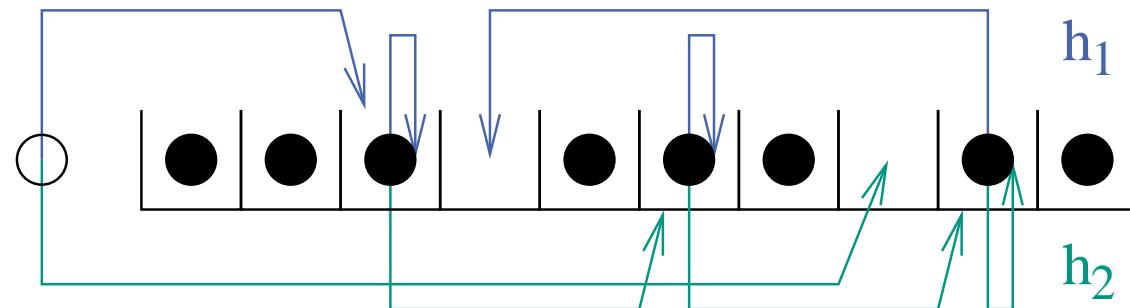
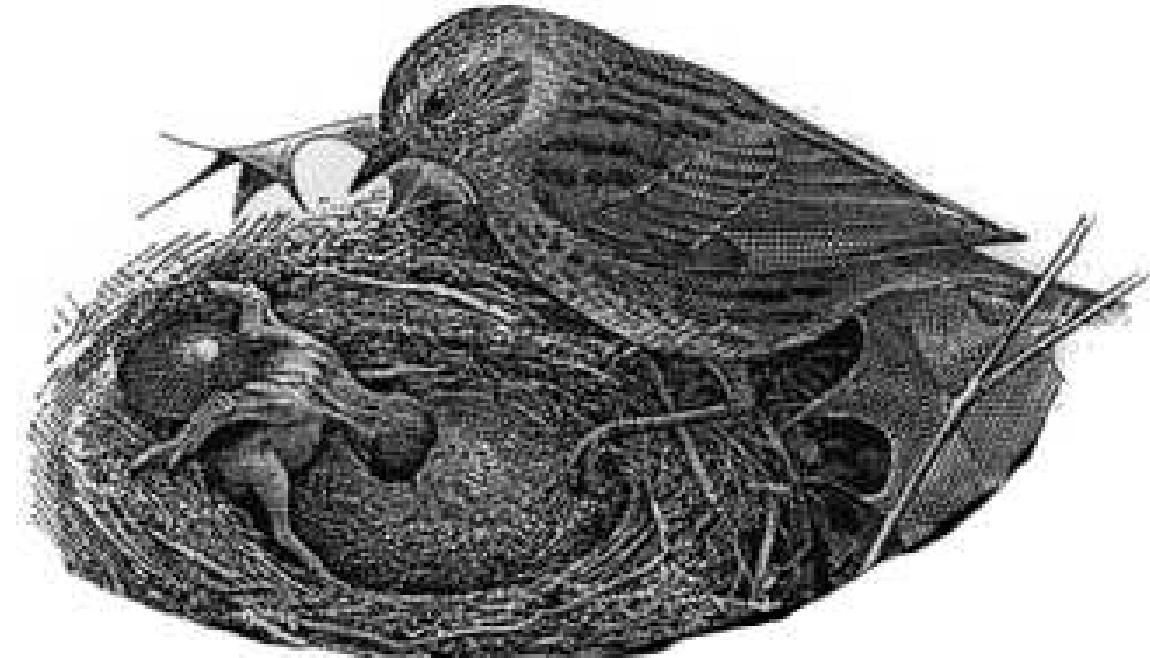
Two choices for each element.

Insert moves elements;

rebuild if necessary.

Very fast lookup and delete.

Expected constant insertion time.



# Cuckoo Hashing – Rebuilds

When needed ?

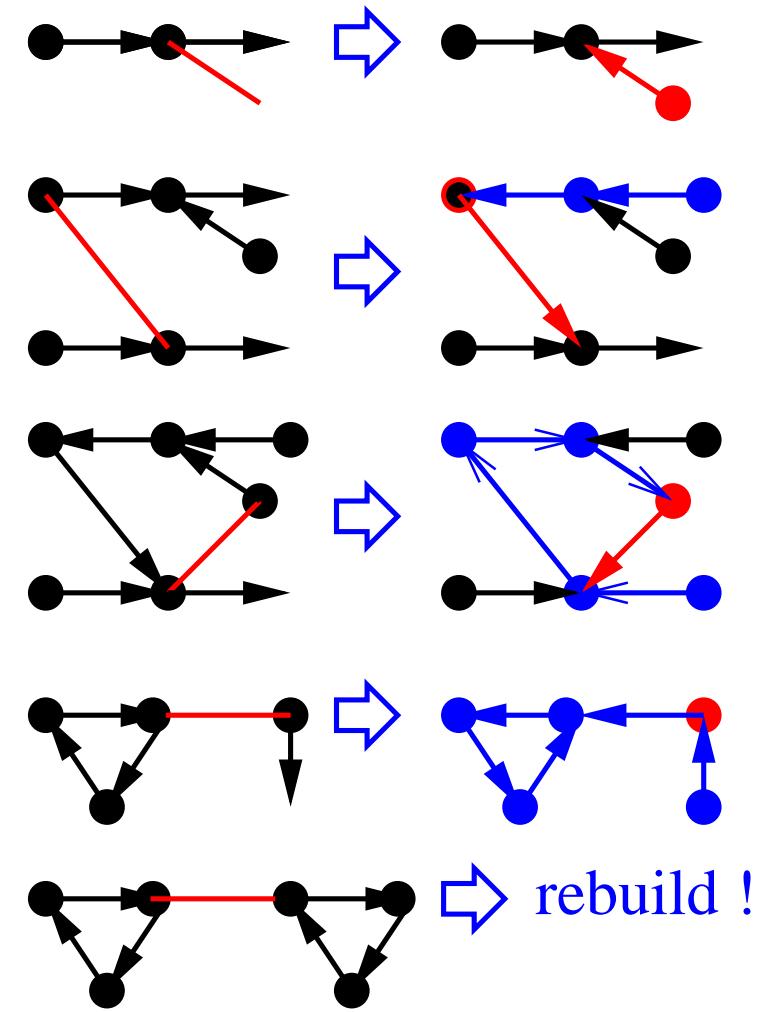
Graph model.

node: table cells

undirected edge: element  $x \rightsquigarrow$   
 edge  $\{h_1(x), h_2(x)\}$

directed:  $(h_2(x), h_1(x))$  means  
 element  $x$  is stored at cell  $h_2(x)$

**Lemma:**  $\text{insert}(x)$  succeeds iff  
 the component containing  $h_1(x), h_2(x)$   
 contains no more edges than nodes.



# Cuckoo Hashing – Rebuilds

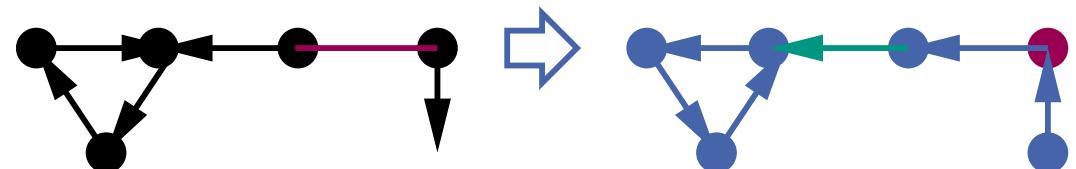
**Lemma:**  $\text{insert}(x)$  succeeds iff  
the component containing  $h_1(x), h_2(x)$   
contains no more edges than nodes.

**Proof outline:** (if-part)

$h_1(x)$  in tree: flip path to root

$h_1(x)$  in pseudotree  $p$ ,  $h_2(x)$  in tree  $t$ :

flip cycle and path to root in  $t$



# Cuckoo Hashing – How Many Rebuilds?

**Theorem:** For truly random hash functions,

$$\Pr[\text{rebuild necessary}] = O(1/n)$$

**Proof:** via random graph theory

# Random Graph Theory

[Erdős, Rényi 1959]

$\mathcal{G}(n, m)$ := sample space of all graphs with  $n$  nodes,  $m$  edges.

A random graph from  $\mathcal{G}(n, m)$  has certain properties **with high probability**, here  $\geq 1 - O(1/n)$ .

Famous: The evolution of component sizes with increasing  $m$ :

$< (1 - \varepsilon)n/2$ : Trees and Pseudotrees of size  $O(\log n)$

$> (1 + \varepsilon)n/2$ : A “giant” component of size  $\Theta(n)$  (sudden emergence)

$> (1 + \varepsilon)n \ln n/2$ : One single component

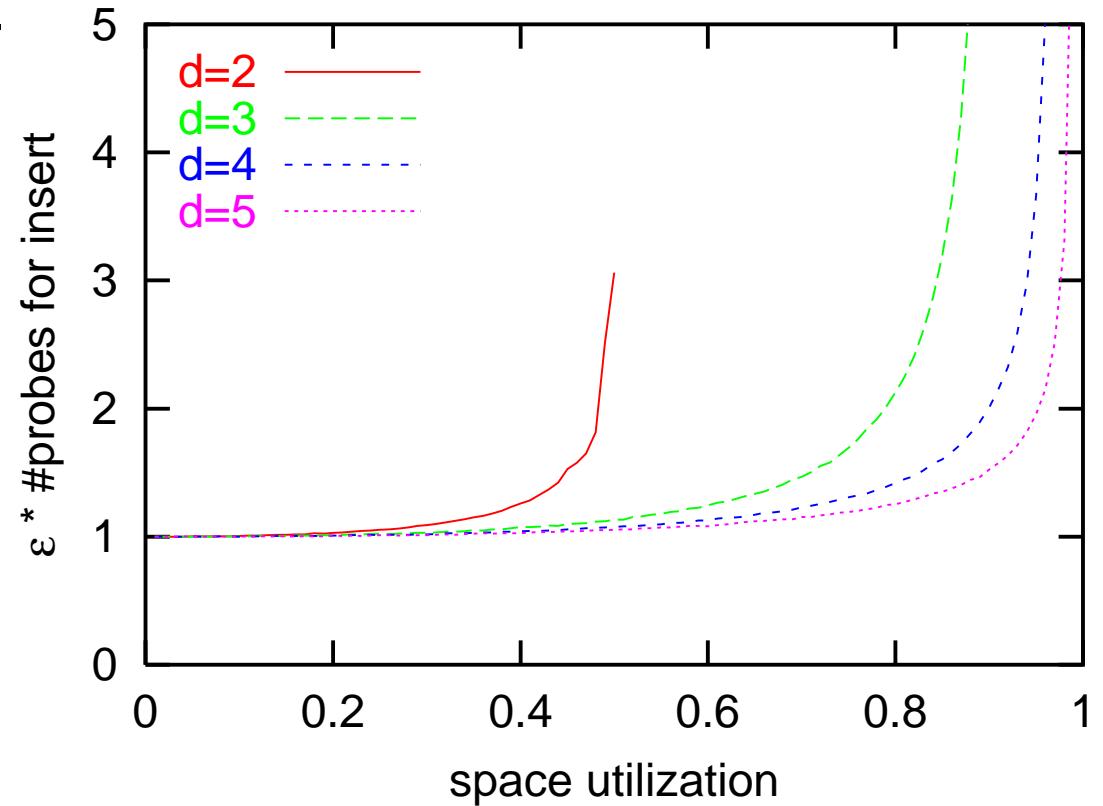


# Space Efficient Cuckoo Hashing

**$d$ -ary:** [Fotakis, Pagh, Sanders, Spirakis 2003]  $d$  possible places.

**Insertion:** BFS, random walk, ...

expected time:  $O\left(\frac{1}{\varepsilon}\right)$  ?



# Space Efficient Cuckoo Hashing

**$d$ -ary:** [Fotakis, Pagh, Sanders, Spirakis 2003]  $d$  possible places.

**blocked:** [Dietzfelbinger, Weidling 2005] cells house  $d$  elements.

Cache efficient !

**blocked,  $d$ -ary, dynamic growing:**

[Maier, Sanders 2017]

# Zusammenfassung: Randomisierte Algorithmen

- einfache, effiziente Algorithmen
- Analyse oft nichttrivial
- z.T. wird aus esoterischer Theorie ein praxisrelevantes Werkzeug,  
z.B. random graph evolution
- Las Vegas versus Monte Carlo
- Brücke zur Algebra: z.B. Checker für Sortieren

# Ausblick: Randomisierte Algorithmen

- externe minimale Spannbäume
- mehr quicksort (strings, parallel)
- kleinster umschließender Kreis
- online paging

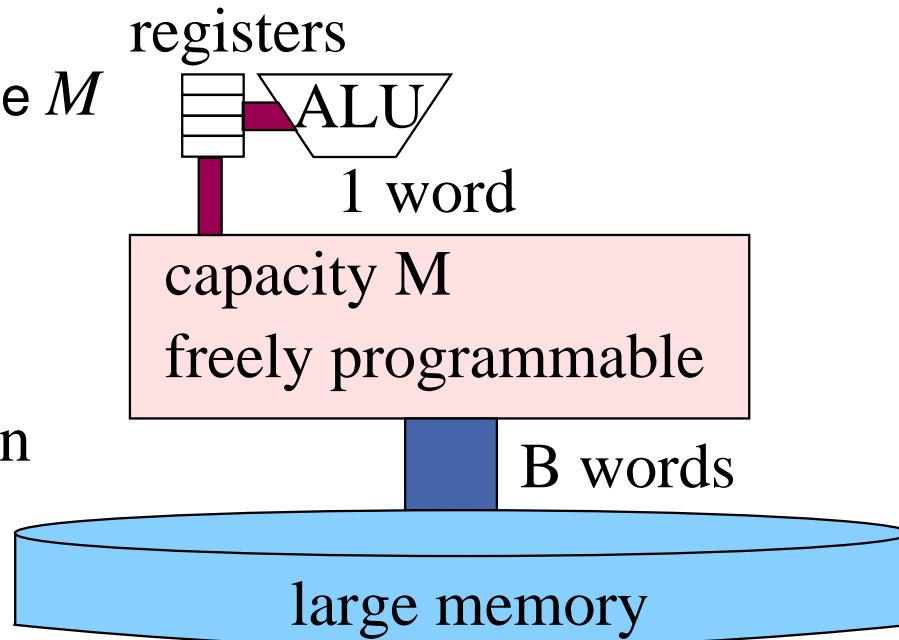
# 7 Externe Algorithmen

## 7.1 Das Sekundärspeichermodell

**M:** Schneller Speicher der Größe  $M$

**B:** Blockgröße

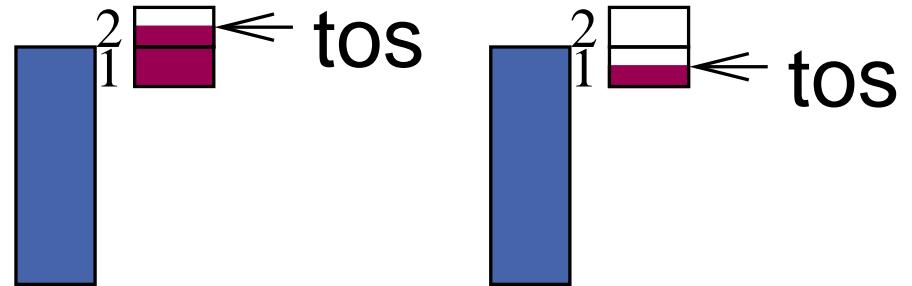
**Analyse:** Blockzugriffe zählen



## 7.2 Externe Stapel

Datei mit Blöcken

2 interne Puffer



**push:** Falls Platz, in Puffer.

Sonst schreibe Puffer **eins** in die Datei (push auf Blockebene)

Umbenennung: Puffer 1 und 2 tauschen die Plätze

**pop:** Falls vorhanden, pop aus Puffer.

Sonst lese Puffer **eins** aus der Datei (pop auf Blockebene)

Analyse: amortisiert  $O(1/B)$  I/Os pro Operation

Aufgabe 1: Beweis.

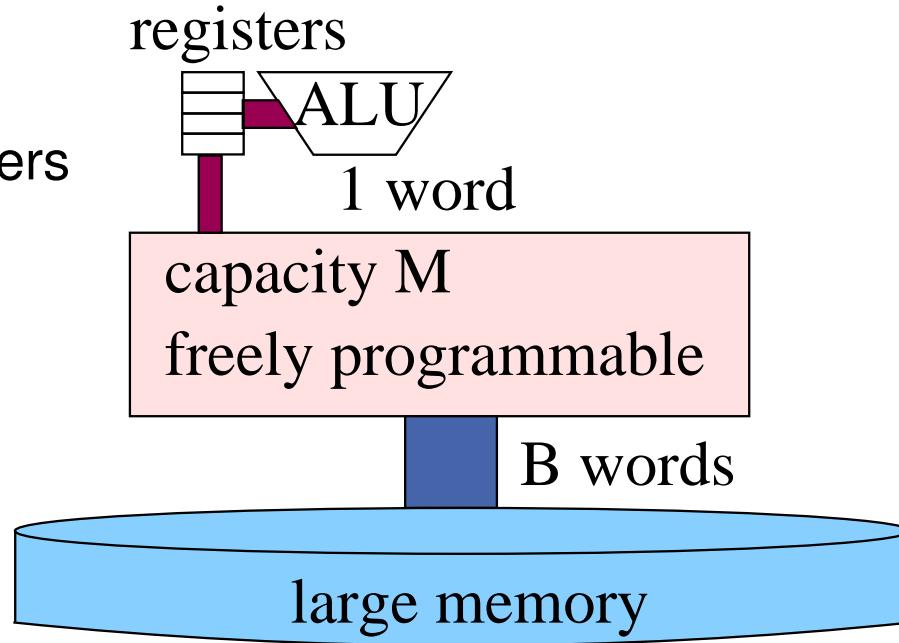
Aufgabe 2: effiziente Implementierung ohne überflüssiges Kopieren

## 7.3 Externes Sortieren

$n$ : Eingabegröße

$M$ : Größe des schnellen Speichers

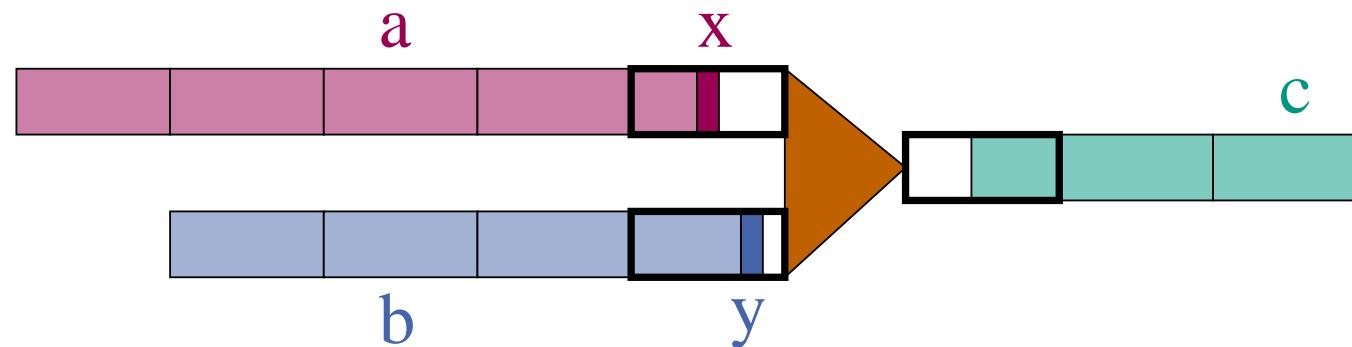
$B$ : Blockgröße



**Procedure** externalMerge( $a, b, c$  :File of Element)

```

 $x := a.\text{readElement}$            // Assume emptyFile.readElement =  $\infty$ 
 $y := b.\text{readElement}$ 
for  $j := 1$  to  $|a| + |b|$  do
    if  $x \leq y$  then    $c.\text{writeElement}(x)$ ;  $x := a.\text{readElement}$ 
    else                   $c.\text{writeElement}(y)$ ;  $y := b.\text{readElement}$ 
  
```



## Externes (binäres) Mischen – I/O-Analyse

Datei  $a$  lesen:  $\lceil |a|/B \rceil \leq |a|/B + 1$ .

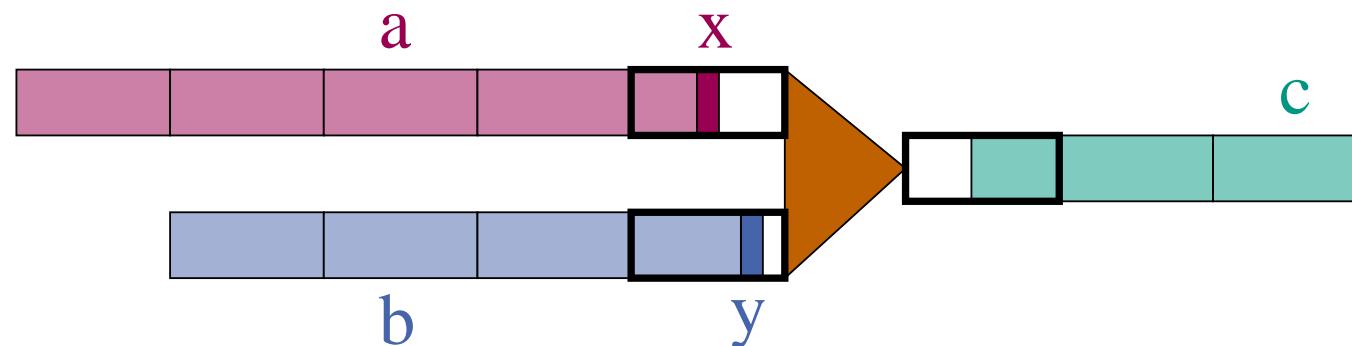
Datei  $b$  lesen:  $\lceil |b|/B \rceil \leq |b|/B + 1$ .

Datei  $c$  schreiben:  $\lceil (|a| + |b|)/B \rceil \leq (|a| + |b|)/B + 1$ .

Insgesamt:

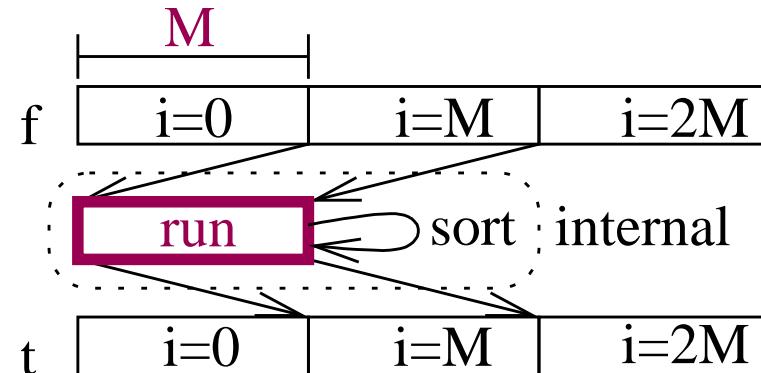
$$\leq 3 + 2 \frac{|a| + |b|}{B} \approx 2 \frac{|a| + |b|}{B}$$

Bedingung: Wir brauchen 3 Pufferblöcke, d.h.,  $M > 3B$ .



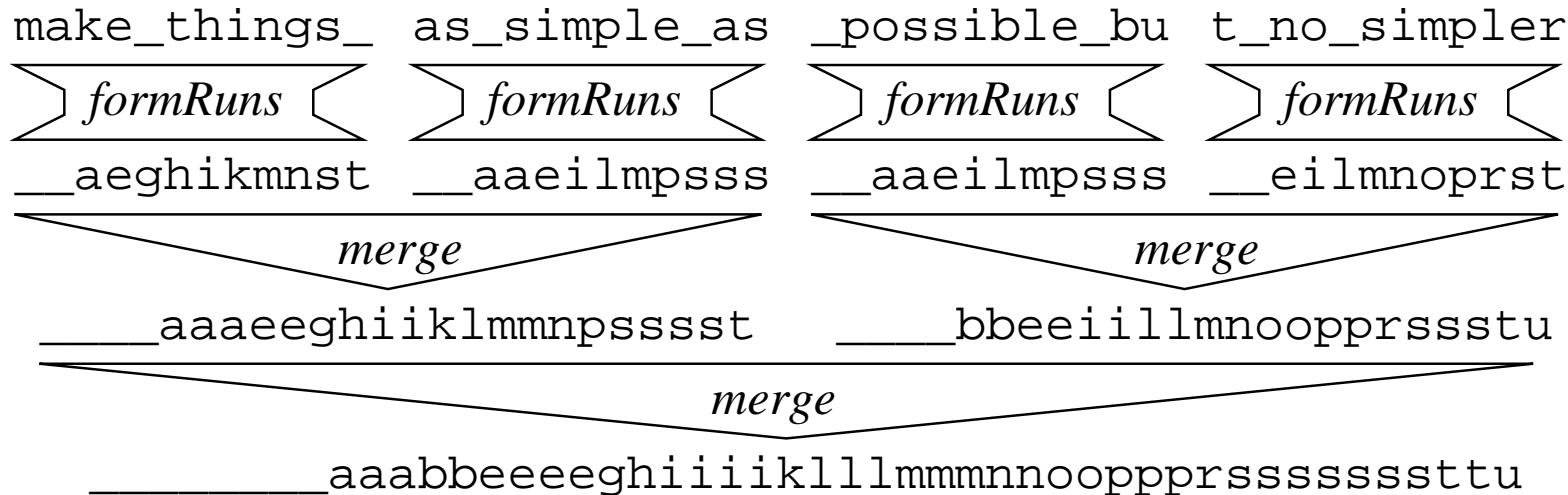
# Run Formation

Sortiere Eingabeportionen der Größe  $M$



$$\text{I/Os: } \approx 2 \frac{n}{B}$$

# Sortieren durch Externes Binäres Mischen



```

Procedure externalBinaryMergeSort // I/Os: ≈
    run formation //  $2n/B$ 
    while more than one run left do //  $\lceil \log \frac{n}{M} \rceil \times$ 
        merge pairs of runs //  $2n/B$ 
    output remaining run //  $\sum : 2\frac{n}{B} \left( 1 + \lceil \log \frac{n}{M} \rceil \right)$ 

```

## Zahlenbeispiel: PC 2018

$$n = 2^{40} \text{ Byte}$$

$$M = 2^{34} \text{ Byte}$$

$$B = 2^{22} \text{ Byte}$$

I/O braucht  $2^{-4}$  s

$$\text{Zeit: } 2 \frac{n}{B} \left( 1 + \left\lceil \log \frac{n}{M} \right\rceil \right) = 2 \cdot 2^{17} \cdot (1 + 6) \cdot 2^{-4} \text{ s} = 2^{16} \text{ s} \approx 32 \text{ h}$$

Idee: 7 Durchläufe  $\rightsquigarrow$  2 Durchläufe

# Mehrwegemischen

**Procedure** multiwayMerge( $a_1, \dots, a_k, c$  :File of Element)

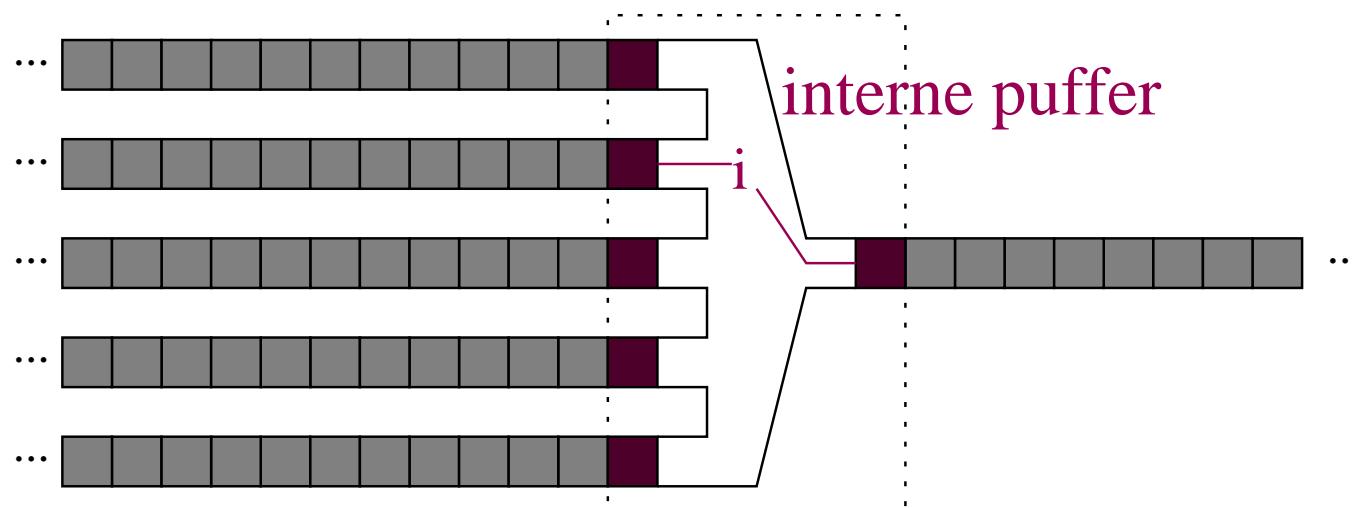
**for**  $i := 1$  **to**  $k$  **do**  $x_i := a_i.\text{readElement}$

**for**  $j := 1$  **to**  $\sum_{i=1}^k |a_i|$  **do**

    find  $i \in 1..k$  that minimizes  $x_i$  // no I/Os!,  $O(\log k)$  time

$c.\text{writeElement}(x_i)$

$x_i := a_i.\text{readElement}$



# Mehrwegemischen – Analyse

I/Os: Datei  $a_i$  lesen:  $\approx |a_i|/B$ .

Datei  $c$  schreiben:  $\approx \sum_{i=1}^k |a_i|/B$

Insgesamt:

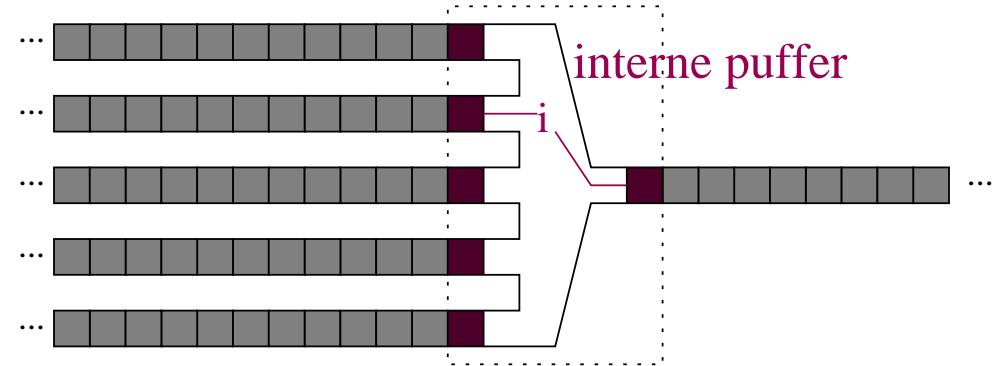
$$\leq \approx 2 \frac{\sum_{i=1}^k |a_i|}{B}$$

Bedingung: Wir brauchen  $k + 1$  Pufferblöcke, d.h.,  $k + 1 < M/B$

(im Folgenden vereinfacht zu  $k < M/B$ )

**Interne Arbeit:** (benutze Prioritätsliste !)

$$O\left(\log k \sum_{i=1}^k |a_i|\right)$$

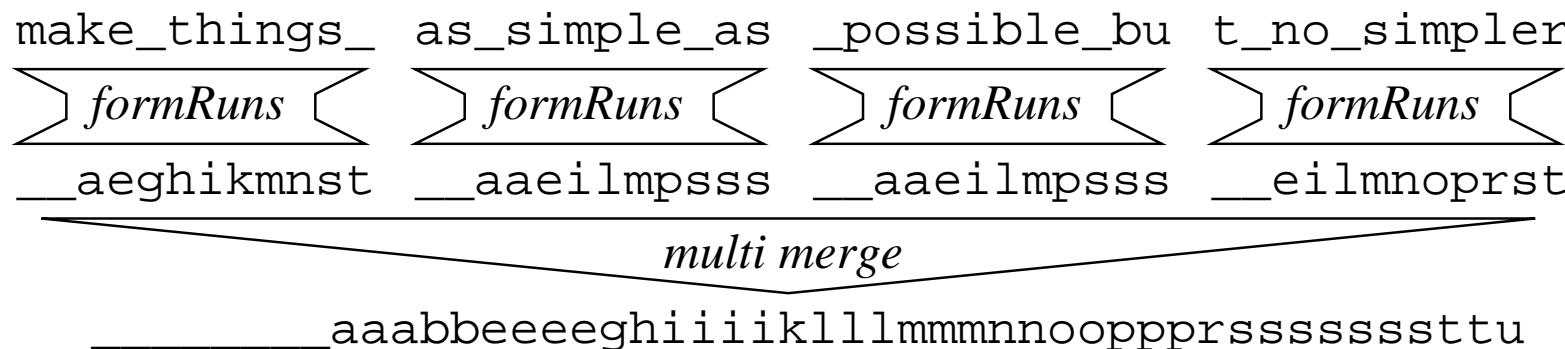


# Sortieren durch Mehrwege-Mischen

- Sortiere  $\lceil n/M \rceil$  runs mit je  $M$  Elementen  $2n/B$  I/Os
  - Mische jeweils  $M/B$  runs  $2n/B$  I/Os
  - bis nur noch ein run übrig ist  $\times \left\lceil \log_{M/B} \frac{n}{M} \right\rceil$  Mischphasen
- 

Insgesamt

$$\text{sort}(n) := \frac{2n}{B} \left( 1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil \right) \text{ I/Os}$$



# Sortieren durch Mehrwege-Mischen

## Interne Arbeit:

$$O\left(\overbrace{n \log M}^{\text{run formation}} + \underbrace{n \log \frac{M}{B}}_{\text{PQ access per phase}} \overbrace{\log_{M/B} \frac{n}{M}}^{\text{phases}}\right) = O(n \log n)$$

## Mehr als eine Mischphase?:

Nicht für Hierarchie Hauptspeicher, Festplatte.

$$\text{Grund } \frac{M}{B} > \frac{\approx 300}{\text{RAM Euro/bit}} \quad \frac{> 1000}{\text{Platte Euro/bit}}$$

## Mehr zu externem Sortieren

Untere Schranke  $\approx \frac{2^{(?)}n}{B} \left( 1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil \right)$  I/Os  
[Aggarwal Vitter 1988]

Obere Schranke  $\approx \frac{2n}{DB} \left( 1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil \right)$  I/Os (erwartet)  
für  $D$  parallele Platten

[Hutchinson Sanders Vitter 2005, Dementiev Sanders 2003]

Offene Frage: deterministisch?

# Externe Prioritätslisten

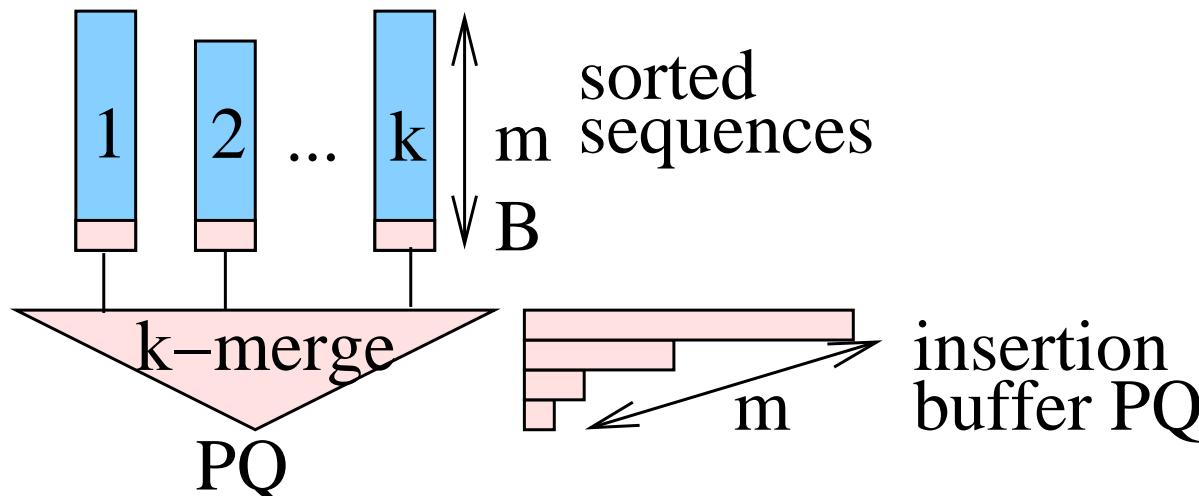
Problem: Binary heaps brauchen

$$\Theta\left(\log \frac{n}{M}\right) \text{ I/Os pro deleteMin}$$

Wir hätten gerne:

$$\Theta\left(\frac{1}{B} \log_{M/B} \frac{n}{M}\right) \text{ I/Os amortisiert}$$

# Mittelgroße PQs – $km \ll M^2/B$ Einfügungen



**Insert:** Anfangs in **insertion buffer**.

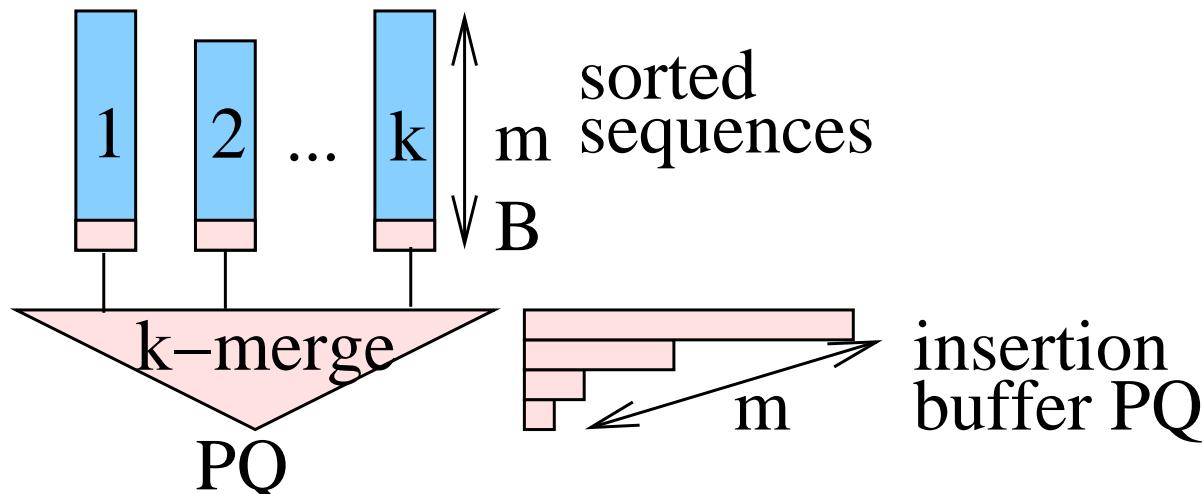
Überlauf →

sort; flush; kleinster Schlüssel in merge-PQ

**Delete-Min:** deleteMin aus der PQ mit kleinerem min

## Analyse – I/Os

`deleteMin`: jedes Element wird  $\leq 1 \times$  gelesen, zusammen mit  $B$  anderen – amortisiert  $1/B$  penalty für `insert`.



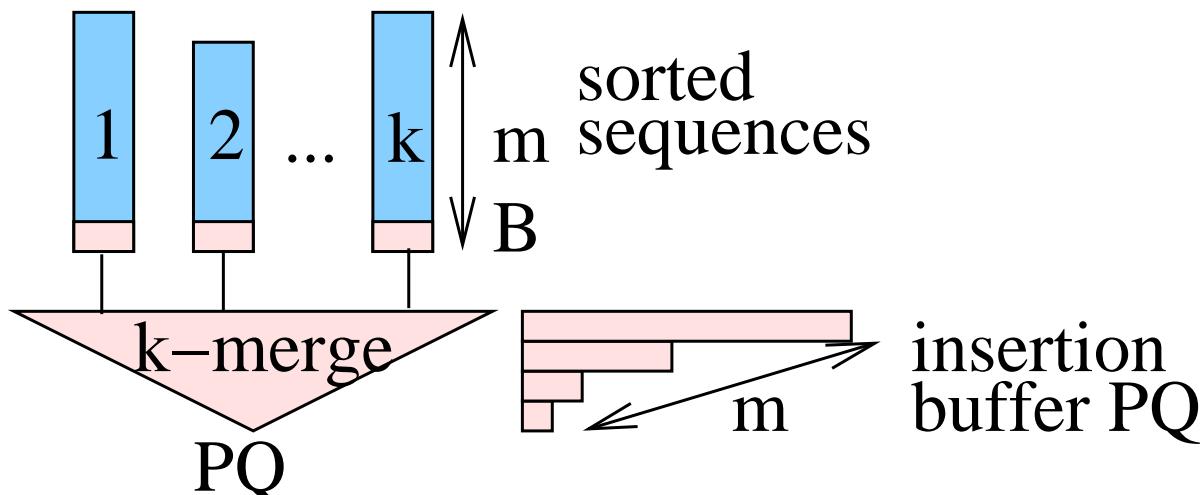
# Analyse – Vergleiche (Maß für interne Arbeit)

deleteMin:  $1 + O(\max(\log k, \log m)) = O(\log m)$

genauere Argumentation: amortisiert  $1 + \log k$  bei geeigneter PQ

insert:  $\approx m \log m$  alle  $m$  Ops. Amortisiert  $\log m$

Insgesamt nur  $\log km$  amortisiert !





# Große Queues

$$\approx \frac{2n}{B} \left( 1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil \right)$$

I/Os für  $n$  Einfügeoperationen

$O(n \log n)$  Arbeit.

[Sanders 1999].

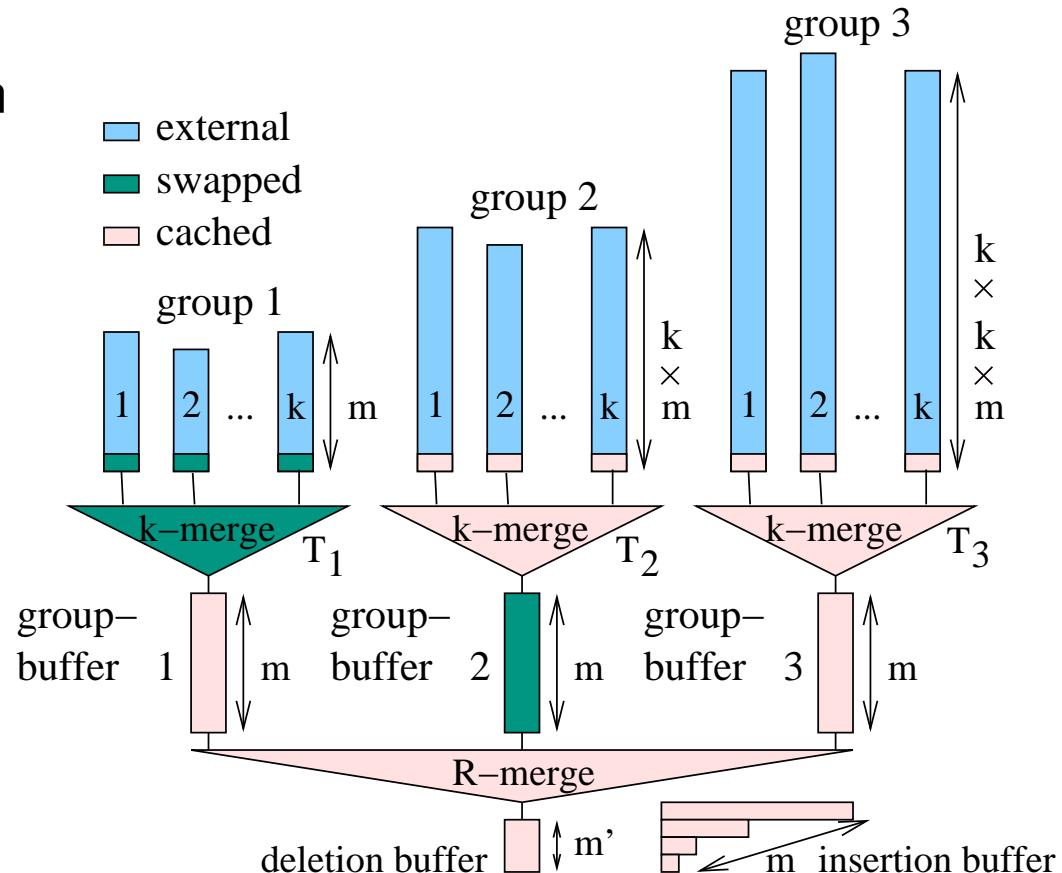
deleteMin:

“amortisiert umsonst”.

Details:

Vorlesung

Algorithm Engineering.



# Experiments

Keys: random 32 bit integers

Associated information: 32 dummy bits

Deletion buffer size: 32 Near optimal

Group buffer size: 256 : performance on

Merging degree  $k$ : 128 all machines tried!

Compiler flags: Highly optimizing, nothing advanced

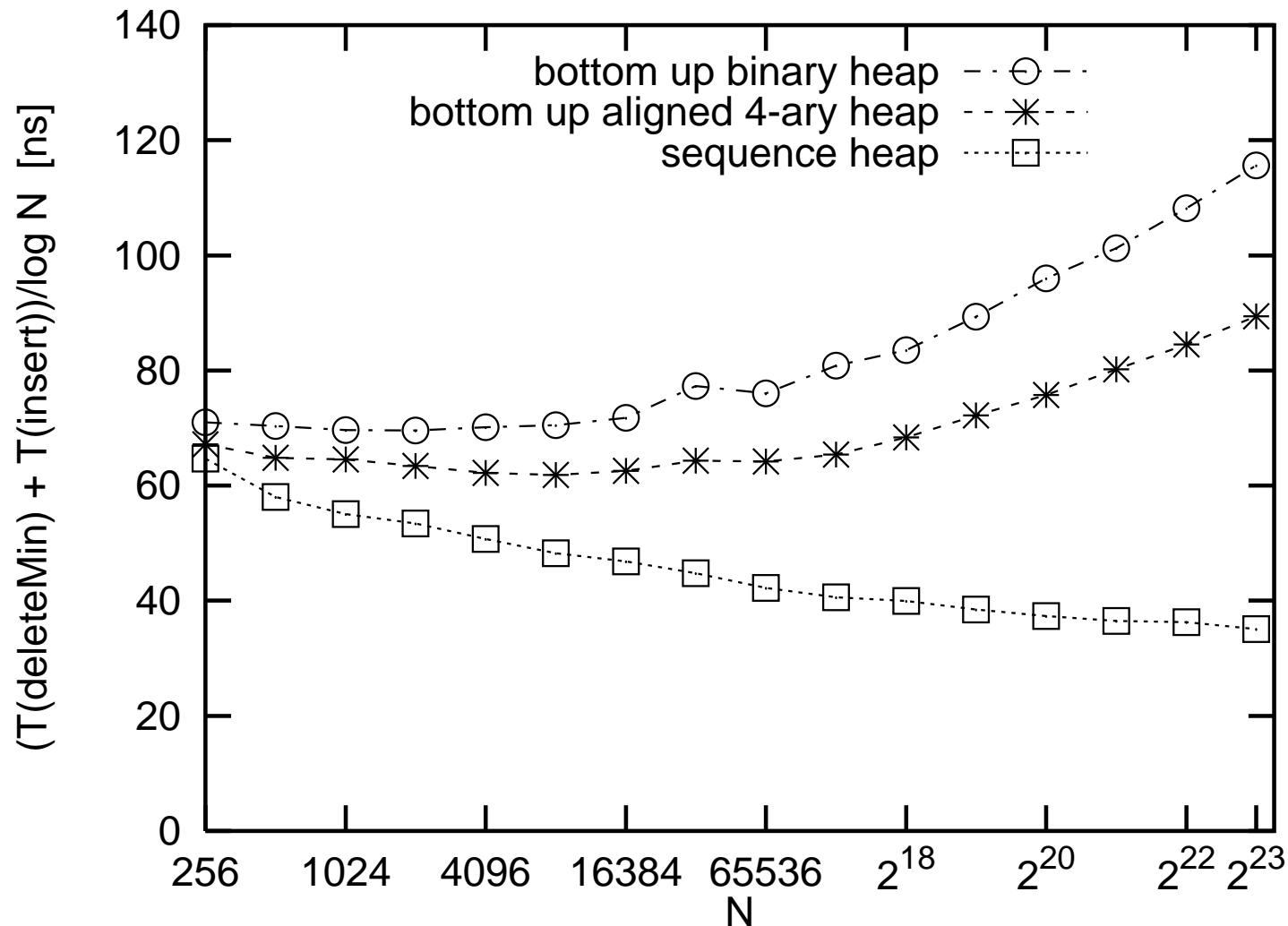
Operation Sequence:

$$(\text{Insert-DeleteMin-Insert})^N (\text{DeleteMin-Insert-DeleteMin})^N$$

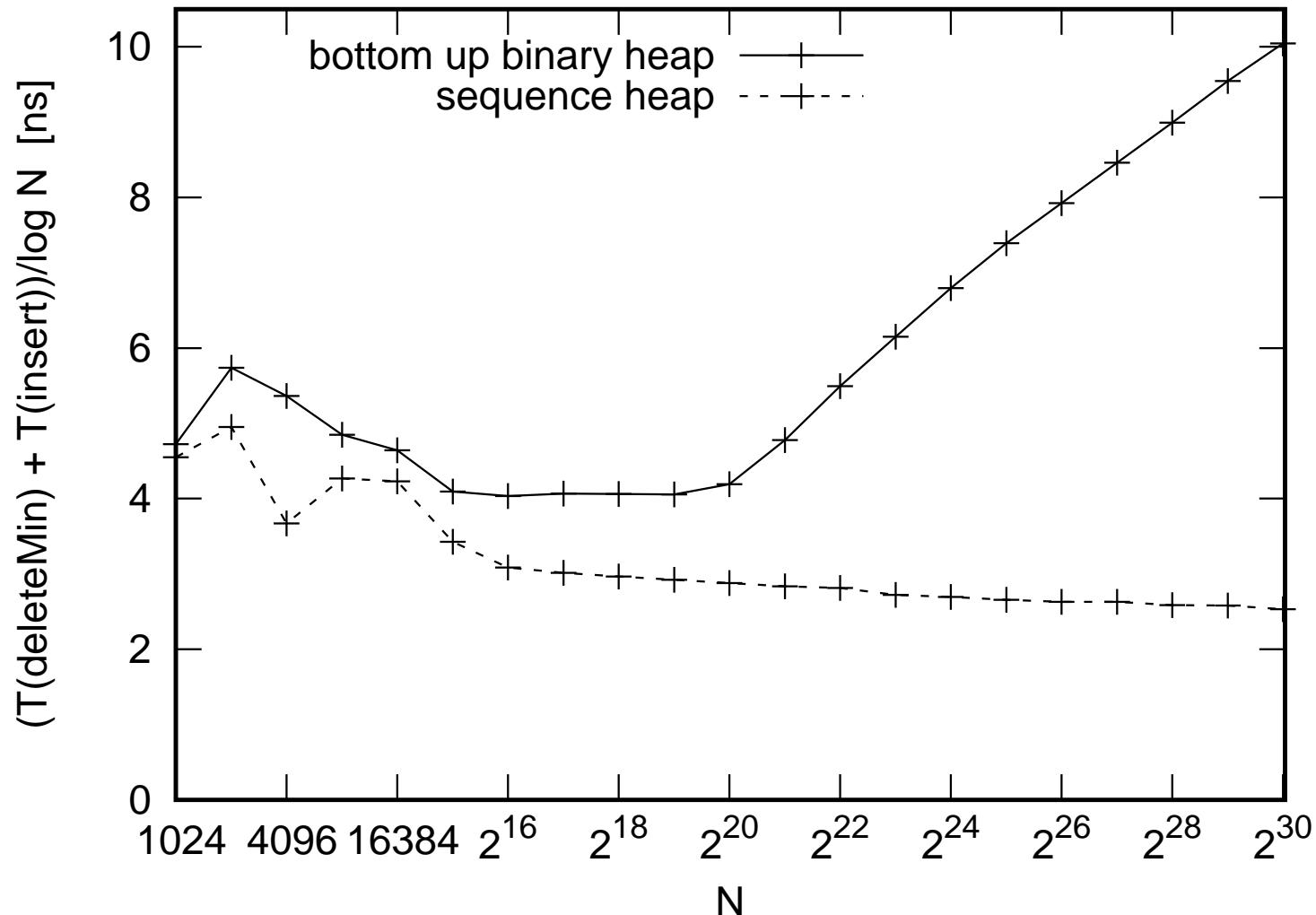
Near optimal performance on all machines tried!



# Alpha-21164, 533 MHz, 1997



# AMD Ryzen 1800X, 16MB L3, 3.6 GHz, 2017



# Minimale Spannbäume

## Semieexterne Kruskal

Annahme:  $M = \Omega(n)$  konstant viele Maschinenworte pro Knoten

**Procedure** seKruskal( $G = (1..n, E)$ )

sort  $E$  by decreasing weight // sort( $m$ ) I/Os

Tc : UnionFind( $n$ )

**foreach**  $(u, v) \in E$  in ascending order of weight **do**

**if** Tc.find( $u$ )  $\neq$  Tc.find( $v$ ) **then**

output  $\{u, v\}$

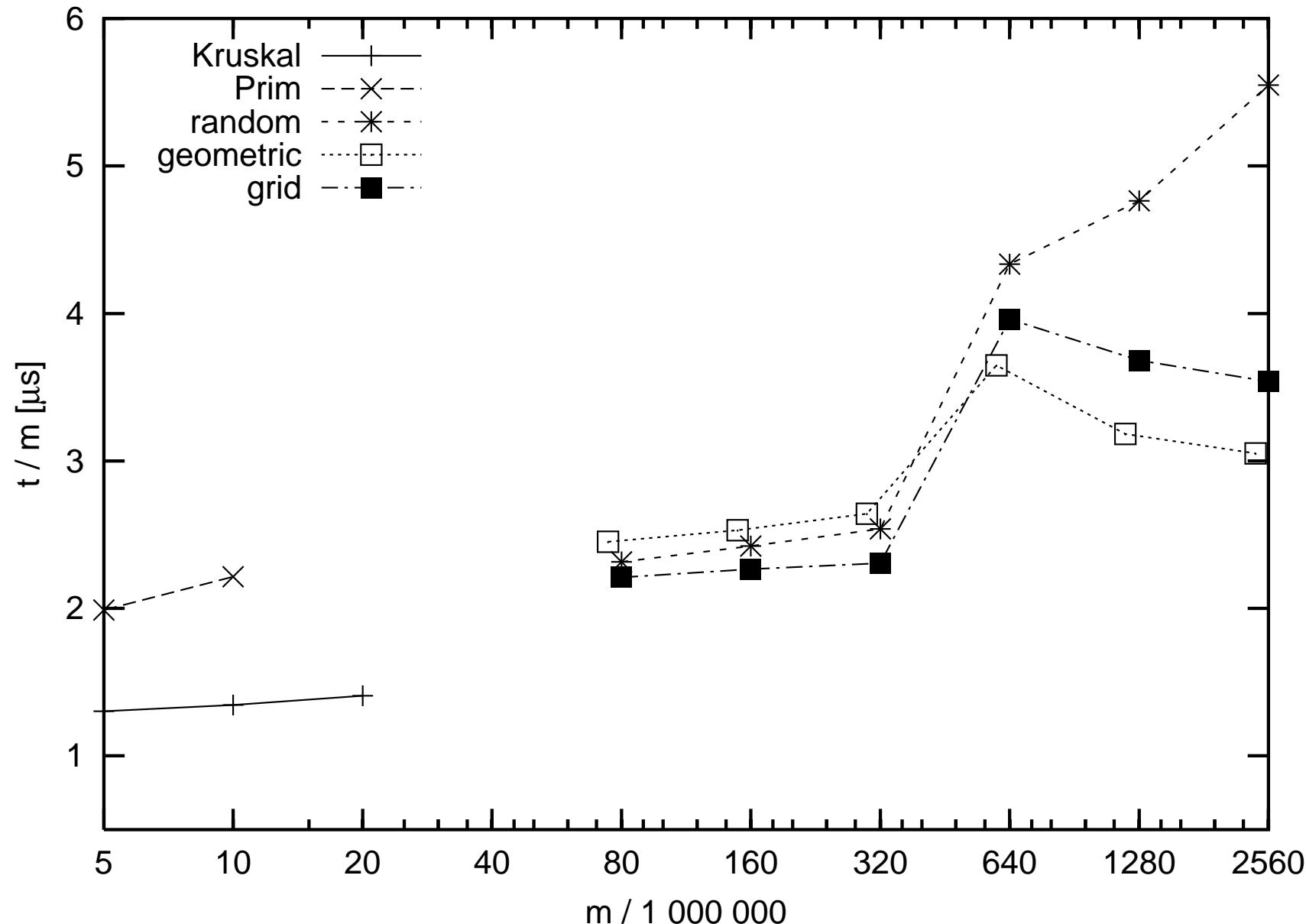
Tc.union( $u, v$ )

// link reicht auch

## Externe MST-Berechnung

- Reduziere Knotenzahl mittels **Kontraktion** von MST-Kanten
  - Details: Vorlesung Algorithm Engineering, Sibeyn's Algorithmus.
  - Implementierung  $\approx$  Sortierer + ext. Prioritätsliste + 1
  - Bildschirmseite. (STXXL Bibliothek)
- benutze semiexternen Algorithmus sobald  $n < M$ .

# Beispiel, Sibeyn's algorithm, $m \approx 2n$



## Mehr zu externen Algorithmen – Basic Toolbox ?

Externe Hashtabellen: geht aber 1 I/O pro Zugriff

Suchbäume:  $(a, 2a)$ -Bäume mit  $a = \Theta(B)$   $\rightsquigarrow \log_B n$  I/Os für Basisoperationen. Brot-und-Butter-Datenstruktur für Datenbanken. Inzwischen auch in Dateisystemen. Viel Tuning: Große Blätter, Caching, ....

BFS: OK bei kleinem Graphdurchmesser

DFS: noch schwieriger. Heuristiken für den **semieexternen** Fall

kürzeste Wege: ähnlich BFS.

# 8 Approximationsalgorithmen

Eine Möglichkeit zum Umgang mit NP-harten Problemen

Beobachtung: Fast alle interessanten Optimierungsprobleme sind  
NP-hart

Auswege:

- Trotzdem optimale Lösungen suchen und riskieren, dass der Algorithmus nicht fertig wird
- Ad-hoc Heuristiken. Man kriegt eine Lösung aber wie gut ist die?
- Approximationsalgorithmen:**
  - Polynomielle Ausführungszeit.
  - Lösungen **garantiert „nah“** am Optimum.
- Problem umdefinieren/spezialisieren

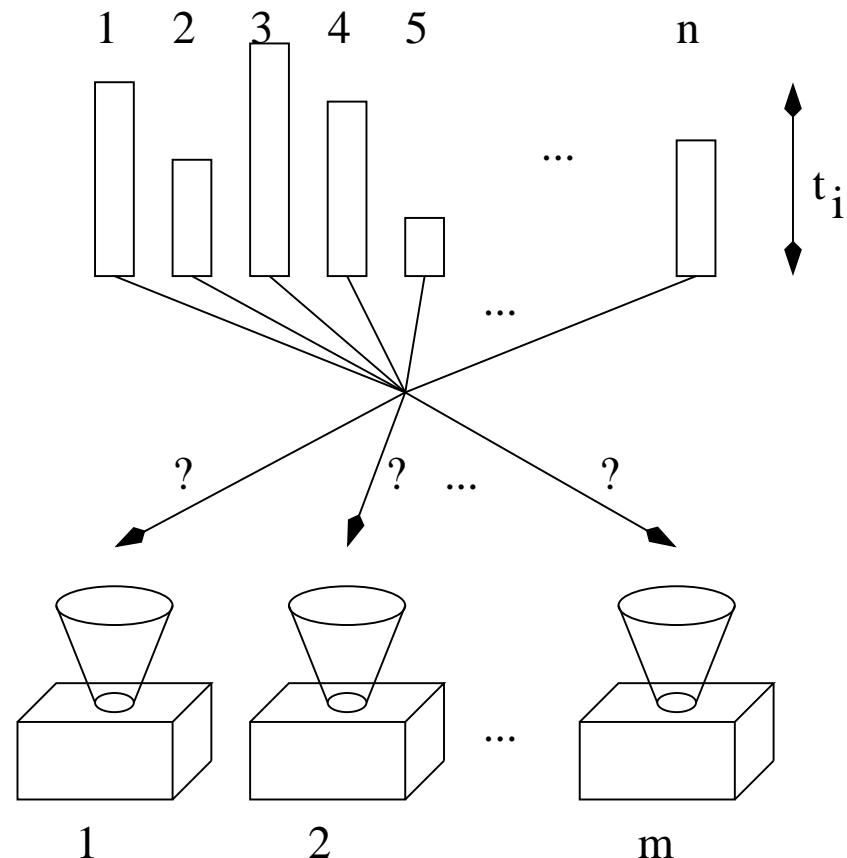
# Scheduling unabhängiger gewichteter Jobs auf parallelen Maschinen

$x(j)$ : Maschine auf der Job  $j$  ausgeführt wird

$L_i$ :  $\sum_{x(j)=i} t_j$ , Last von Maschine  $i$

Zielfunktion: Minimiere Makespan

$$L_{\max} = \max_i L_i$$



Details: Identische Maschinen, unabhängige Jobs, bekannte Ausführungszeiten, offline

## List Scheduling

ListScheduling( $n, m, \mathbf{t}$ )

$J := \{1, \dots, n\}$

array  $L[1..m] = [0, \dots, 0]$

**while**  $J \neq \emptyset$  **do**

    pick **any**  $j \in J$

$J := J \setminus \{j\}$

**//Shortest Queue:**

        pick  $i$  such that  $L[i]$  is minimized

$\mathbf{x}(j) := i$

$L[i] := L[i] + t_j$

**return**  $\mathbf{x}$

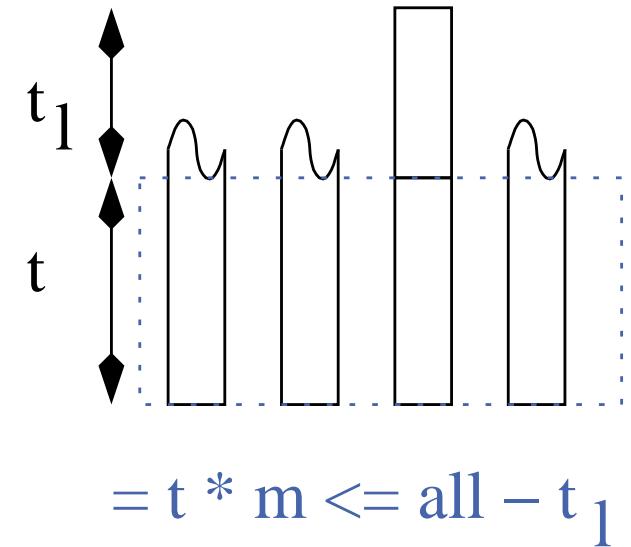
## Viele Kleine Jobs

**Lemma 1.** Falls  $\ell$  der zuletzt beendete Job ist, dann

$$L_{\max} \leq \sum_j \frac{t_j}{m} + \frac{m-1}{m} t_\ell$$

## Beweis

$$L_{\max} = t + t_\ell \leq \sum_{j \neq \ell} \frac{t_j}{m} + t_\ell = \sum_j \frac{t_j}{m} + \frac{m-1}{m} t_\ell$$



## Untere Schranken

**Lemma 2.**  $L_{\max} \geq \sum_j \frac{t_j}{m}$

**Lemma 3.**  $L_{\max} \geq \max_j t_j$

## Der Approximationsfaktor

Definition:

Ein Minimierungsalgorithmus erzielt **Approximationsfaktor  $\rho$**  bezüglich Zielfunktion  $f$  falls er für **alle** Eingaben  $I$ , eine Lösung  $\mathbf{x}(I)$  findet, so dass

$$\frac{f(\mathbf{x}(I))}{f(\mathbf{x}^*(I))} \leq \rho$$

wobei  $\mathbf{x}^*(I)$  die optimale Lösung für Eingabe  $I$  bezeichnet.

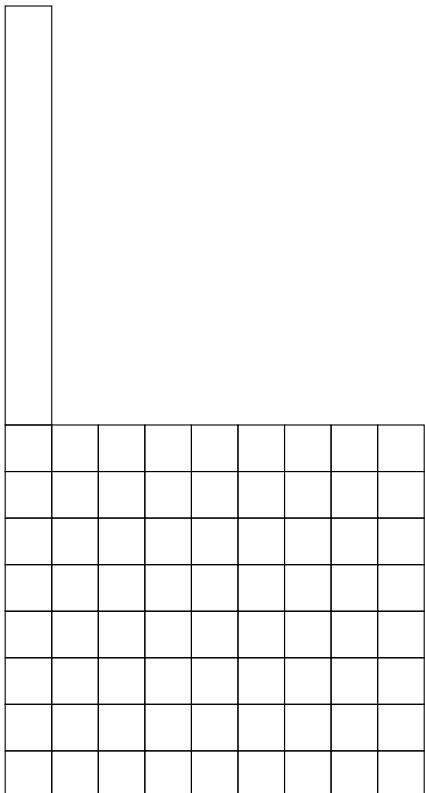
**Satz:** ListScheduling erzielt Approximationsfaktor  $2 - \frac{1}{m}$ .

**Beweis:**

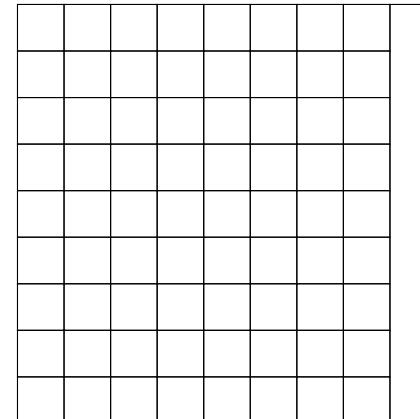
$$\begin{aligned} & \frac{f(\mathbf{x})}{f(\mathbf{x}^*)} \\ & \leq \frac{\sum_j t_j / m}{f(\mathbf{x}^*)} + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} && \text{(obere Schranke Lemma 1)} \\ & \leq 1 + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} && \text{(untere Schranke Lemma 2)} \\ & \leq 1 + \frac{m-1}{m} = 2 - \frac{1}{m} && \text{(untere Schranke Lemma 3)} \end{aligned}$$

## Diese Schranke ist bestmöglich

Eingabe:  $m(m - 1)$  Jobs der Größe 1 und ein Job der Größe  $m$ .



List Scheduling:  $2m-1$



OPT:  $m$

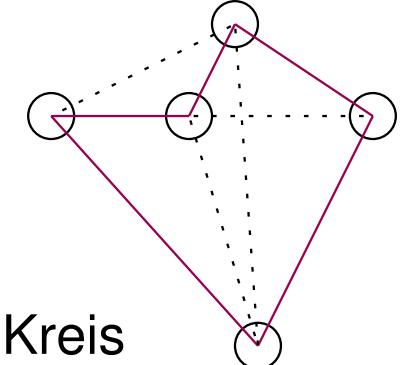
Also ist der Approximationsfaktor  $\geq 2 - 1/m$ .



## Mehr zu Scheduling (siehe Approx-Vorlesung)

- 4/3-Approximation: Sortiere die Jobs nach absteigender Größe.  
Dann List-Scheduling. Zeit  $O(n \log n)$ .
- Schnelle 7/6 Approximation: Rate Makespan (binäre Suche).  
Dann Best Fit Decreasing.
- PTAS ... später ...
- Uniform machines: Maschine  $i$  hat Geschwindigkeit  $v_i$  job  $j$  braucht Zeit  $t_j/v_i$  auf Maschine  $j$ .  $\rightsquigarrow$  relative einfache Verallgemeinerung
- Unrelated Machines Job  $j$  braucht Zeit  $t_{ji}$  auf Maschine  $j$ .  
2-Approximation. Ganz anderer Algorithmus.
- uvam: Andere Zielfunktionen, Reihenfolgebeschränkungen, ...

# Nichtapproximierbarkeit des Handlungsreisendenproblems (TSP)



Gegeben ein Graph  $G = (V, V \times V)$ , finde einen einfachen Kreis  $C = (v_1, v_2, \dots, v_n, v_1)$  so dass  $n = |V|$  und  $\sum_{(u,v) \in C} d(u, v)$  minimiert wird.

**Satz:** Es ist NP-hart das TSP innerhalb irgendeines Faktors  $a$  zu approximieren.

**Beweisansatz:** Es genügt zu zeigen, dass  $\text{HamiltonCycle} \leq_p a\text{-Approximation von TSP}$

# $a$ -Approximation von TSP

**Gegeben:**

Graph  $G = (V, V \times V)$  mit Kantengewichten  $d(u, v)$ ,  
Parameter  $W$ .

Gesucht ist ein Algorithmus, mit folgenden Eigenschaften:

$[G, W]$  wird akzeptiert  $\longrightarrow \exists$  Tour mit Gewicht  $\leq aW$ .

$[G, W]$  wird abgelehnt  $\longrightarrow \nexists$  Tour mit Gewicht  $\leq W$ .

# HamiltonCycle $\leq_p$ $a$ -Approximation von TSP

Sei  $G = (V, E)$  beliebiger ungerichteter Graph.

Definiere  $d(u, v) = \begin{cases} 1 & \text{falls } (u, v) \in E \\ 1 + an & \text{sonst} \end{cases}$

Dann und nur dann, wenn  $G$  einen Hamiltonkreis hat gilt

$\exists$  TSP Tour mit Kosten  $n$

(sonst optimale Kosten  $\geq n + an - 1 > an$ )

Entscheidungsalgorithmus für Hamiltonkreis:

Führe  $a$ -approx TSP auf  $[G, n]$  aus.

Wird akzeptiert

→  $\exists$  Tour mit Gewicht  $\leq an$

→  $\exists$  Tour mit Gewicht  $n$  →  $\exists$  Hamiltonpfad

sonst  $\nexists$  Hamiltonpfad

# TSP mit Dreiecksungleichung

$G$  (ungerichtet) erfüllt die Dreiecksungleichung

$$\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$$

## Metrische Vervollständigung

Betrachte beliebigen ungerichteten Graph  $G = (V, E)$  mit Gewichtsfunktion

$c : E \rightarrow \mathbb{R}_+$ . Definiere

$d(u, v) :=$  Länge des kürzesten Pfades von  $u$  nach  $v$

Beispiel: (ungerichteter) Strassengraph  $\longrightarrow$  Abstandstabelle

# Euler-Touren/-Kreise

Betrachte beliebigen zusammenhängenden ungerichteten  
(Multi-)Graph  $G = (V, E)$  mit  $|E| = m$ .

Ein Pfad  $P = \langle e_1, \dots, e_m \rangle$  ist ein **Euler-Kreis** falls  $\{e_1, \dots, e_m\} = E$ .  
(Jede **Kante** wird genau einmal besucht)

Satz:  $G$  hat Euler-Kreis gdw.  $G$  ist zusammenhängend und  
 $\forall v \in V : \text{Grad}(v)$  ist gerade.

Euler-Kreise lassen sich in Zeit  $O(|E| + |V|)$  finden.

## 2-Approximation durch minimalen Spannbaum

**Lemma 4.**

*Gesamtgewicht eines MST  $\leq$*

*Gesamtgewicht jeder TSP-Tour*

Algorithmus:

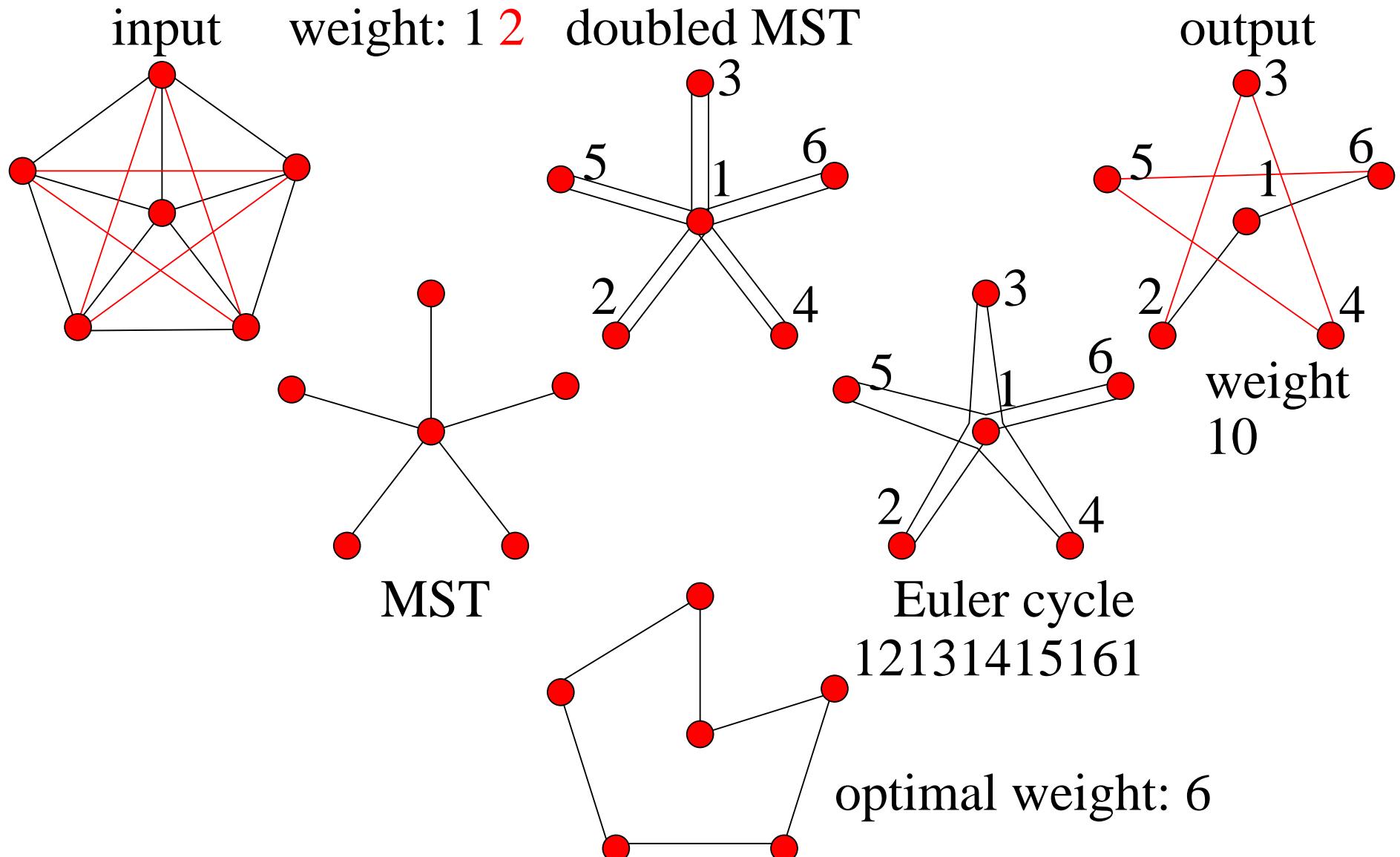
$T := \text{MST}(G)$  // weight( $T$ )  $\leq$  opt

$T' := T$  with every edge doubled // weight( $T'$ )  $\leq 2\text{opt}$

$T'' := \text{EulerKreis}(T')$  // weight( $T''$ )  $\leq 2\text{opt}$

output removeDuplicates( $T''$ ) // shortcutting

## Beispiel



## Beweis von $\text{Gewicht MST} \leq \text{Gewicht TSP-Tour}$

Sei  $T$  die optimale TSP tour

entferne eine Kante

macht  $T$  leichter

nun ist  $T$  ein Spannbaum

der nicht leichter sein kann als der

MST

■

## Allgemeine Technik: Relaxation

hier: ein TSP-Pfad ist ein Spezialfall eines Spannbaums

## Mehr TSP

- Praktisch bessere 2-Approximationen, z.B. lightest edge first
- Relativ einfache aber unpraktische  $\frac{3}{2}$ -Approximation  
(MST+min. weight perfect matching+Euler-Kreis)
- PTAS for Euclidean TSP
- Versuchskanichen für praktisch jede Optimierungsheuristik
- Optimale Lösungen für praktische Eingaben. Faustregel:  
**Falls es in den Speicher passt, lässt sichs lösen.**  
[<http://www.tsp.gatech.edu/concorde.html>]  
sechsstellige Anzahl Codezeilen.
- TSP-artige Anwendungen sind meist komplizierter

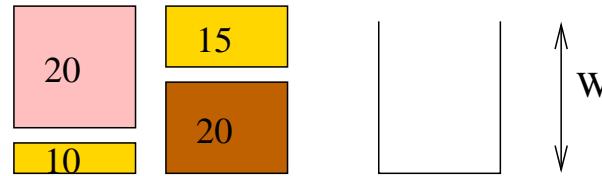
# Pseudopolynomielle Algorithmen

$\mathcal{A}$  ist pseudopolynomieller Algorithmus falls

$$\text{Time}_{\mathcal{A}}(n) \in \mathbf{P}(n)$$

wobei  $n$  die Anzahl Eingabebits ist,  
wenn alle Zahlen unär codiert werden ( $k \equiv 1^k$ ).

# Beispiel Rucksackproblem



- $n$  Gegenstände mit Gewicht  $w_i \in \mathbb{N}$  und profit  $p_i$   
oBdA:  $\forall i \in 1..n : w_i \leq W$
- Wähle eine Teilmenge  $\mathbf{x}$  von Gegenständen
- so dass  $\sum_{i \in \mathbf{x}} w_i \leq W$  und
- maximiere den Profit  $\sum_{i \in \mathbf{x}} p_i$

# Dynamische Programmierung nach Profit

$C(i, P) :=$  kleinste Kapazität für Gegenstände  $1, \dots, i$  die Profit  $\geq P$  ergeben.

**Lemma 5.**

$$\forall 1 \leq i \leq n : C(i, P) = \min(C(i-1, P), C(i-1, P - p_i) + w_i)$$

## Dynamische Programmierung nach Profit

Sei  $\hat{P}$  obere Schranke für den Profit (z.B.  $\sum_i p_i$ ).

Zeit:  $O(n\hat{P})$  pseudo-polynomiell

z.B.  $0..n \times 0..\hat{P}$  Tabelle  $C(i, P)$  spaltenweise ausfüllen

Platz:  $\hat{P} + O(n)$  Maschinenworte plus  $\hat{P}n$  bits.

# Fully Polynomial Time Approximation Scheme

Algorithm  $\mathcal{A}$  ist ein

(Fully) Polynomial Time Approximation Scheme

für  $\begin{matrix} \text{minimization} \\ \text{maximization} \end{matrix}$  Problem  $\Pi$  falls:

Eingabe: Instanz  $I$ , Fehlerparameter  $\varepsilon$

Ausgabequalität:  $f(\mathbf{x}) \stackrel{\leq}{\geq} \left( \frac{1+\varepsilon}{1-\varepsilon} \right) \text{opt}$

Zeit: Polynomiell in  $|I|$  (und  $1/\varepsilon$ )

# Beispiele schranken

| PTAS                             | FPTAS                         |
|----------------------------------|-------------------------------|
| $n + 2^{1/\varepsilon}$          | $n^2 + \frac{1}{\varepsilon}$ |
| $n^{\log \frac{1}{\varepsilon}}$ | $n + \frac{1}{\varepsilon^4}$ |
| $n^{\frac{1}{\varepsilon}}$      | $n/\varepsilon$               |
| $n^{42/\varepsilon^3}$           | :                             |
| $n + 2^{2^{1000/\varepsilon}}$   | :                             |
| :                                | :                             |

# FPTAS für Knapsack

```
P:= maxi pi                                // maximaler Einzelprofit
K:= εP/n                                    // Skalierungsfaktor
p'_i:= ⌊ pi / K ⌋                      // skaliere Profite
x':= dynamicProgrammingByProfit(p', w, C)
gib x' aus
```

**Lemma 6.**  $\mathbf{p} \cdot \mathbf{x}' \geq (1 - \varepsilon)\text{opt}$ .

*Beweis.* Betrachte die optimale Lösung  $\mathbf{x}^*$ .

$$\begin{aligned} \mathbf{p} \cdot \mathbf{x}^* - K\mathbf{p}' \cdot \mathbf{x}^* &= \sum_{i \in \mathbf{x}^*} \left( p_i - K \left\lfloor \frac{p_i}{K} \right\rfloor \right) \\ &\leq \sum_{i \in \mathbf{x}^*} \left( p_i - K \left( \frac{p_i}{K} - 1 \right) \right) = |\mathbf{x}^*|K \leq nK, \end{aligned}$$

also,  $K\mathbf{p}' \cdot \mathbf{x}^* \geq \mathbf{p} \cdot \mathbf{x}^* - nK$ . Weiterhin,

$$K\mathbf{p}' \cdot \mathbf{x}^* \leq K\mathbf{p}' \cdot \mathbf{x}' = \sum_{i \in \mathbf{x}'} K \left\lfloor \frac{p_i}{K} \right\rfloor \leq \sum_{i \in \mathbf{x}'} K \frac{p_i}{K} = \mathbf{p} \cdot \mathbf{x}'. \text{ Also,}$$

$$\mathbf{p} \cdot \mathbf{x}' \geq K\mathbf{p}' \cdot \mathbf{x}^* \geq \mathbf{p} \cdot \mathbf{x}^* - nK = \text{opt} - \varepsilon \underbrace{P}_{\leq \text{opt}} \geq (1 - \varepsilon)\text{opt}$$

□

**Lemma 7.** Laufzeit  $\mathcal{O}(n^3/\varepsilon)$ .

*Beweis.* Die Laufzeit  $\mathcal{O}(n\hat{P}')$  der dynamischen Programmierung dominiert:

$$n\hat{P}' \leq n \cdot (n \cdot \max_{i=1}^n p'_i) = n^2 \left\lfloor \frac{P}{K} \right\rfloor = n^2 \left\lfloor \frac{Pn}{\varepsilon P} \right\rfloor \leq \frac{n^3}{\varepsilon}.$$

□

## Das beste bekannte FPTAS

[Kellerer, Pferschy 04]

$$O\left(\min \left\{ n \log \frac{1}{\varepsilon} + \frac{\log^2 \frac{1}{\varepsilon}}{\varepsilon^3}, \dots \right\} \right)$$

- Weniger buckets  $C_j$  (nichtuniform)
- Ausgefeilte dynamische Programmierung

# Optimale Algorithmen für das Rucksackproblem

Annähernd Linearzeit für fast alle Eingaben! In Theorie und Praxis.

[Beier, Vöcking, An Experimental Study of Random Knapsack Problems, European Symposium on Algorithms, 2004.]

[Kellerer, Pferschy, Pisinger, Knapsack Problems, Springer 2004.]

## 9 Fixed-Parameter-Algorithmen

Praktische Beobachtung: Auch bei NP-harten Problemen können wir u.U. exakte Lösungen finden:  
... für **einfache** Instanzen.

Wie charakterisiert man Einfachheit ?

Durch einen weiteren Parameter  $k$  (neben der Eingabegröße)

Beispiel:  $k = \text{Ausgabegröße}$

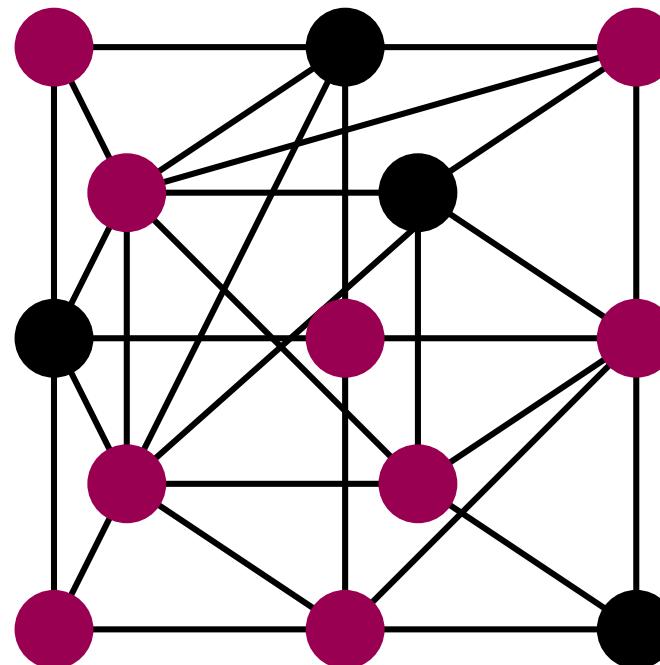
[Niedermeier, Invitation to Fixed Parameter Algorithms, Oxford U.  
Press, 2006]

# Beispiel: VERTEX COVER (Knotenüberdeckung)

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

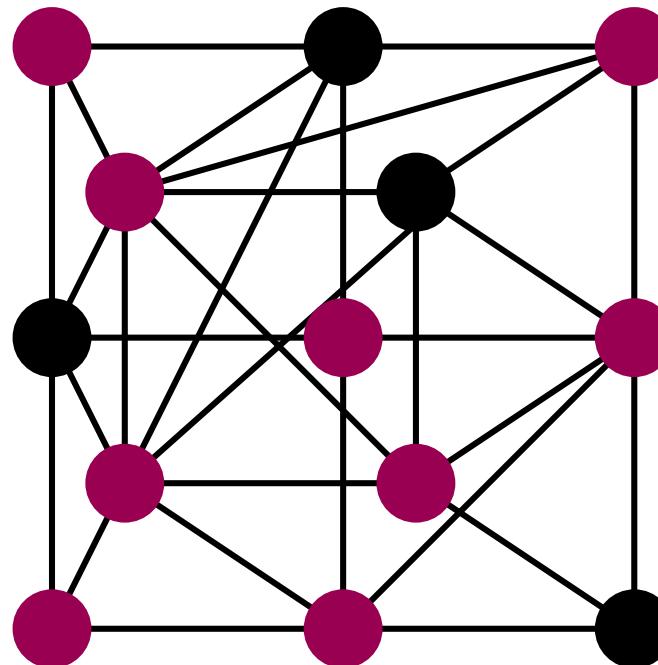
Parameter  $k \in \mathbb{N}$ .

**Frage:**  $\exists V' \subseteq V : |V'| = k \wedge \forall \{u, v\} \in E : u \in V' \vee v \in V'$



# VERTEX COVER Grundlegendes

- Eines der (21) klassischen **NP-harten** Probleme
- Trivialer  $O(n^{k+1})$  Brute-Force-Algorithmus
- Äquivalent zu max. independent set (Komplement)



# Fixed parameter tractable

Eine formale Sprache  $L \in \text{FPT}$  bzgl. Parameter  $k \Leftrightarrow$

$\exists$  Algorithmus mit Laufzeit  $O(f(k) \cdot p(n))$ ,

$f$  berechenbare Funktion, nicht von  $n$  abhängig,

$p$  Polynom, nicht von  $k$  abhängig.

**Beispiele:**  $2^k n^2, k! n^{333}, n + 1 \cdot 1^k$

**Gegenbeispiele:**  $n^k, n^{\log \log k}$

## Beispiel: VERTEX COVER

**Satz:** Vertex Cover ist in FPT bzgl. des Parameters

Ausgabekomplexität

Wir entwickeln Algorithmen mittels zweier auch praktisch wichtiger Entwurfstechniken:

1. **Kernbildung**: (Kernelization) Reduktionsregeln reduzieren Problem auf Größe  $O(f(k))$
2. Systematische **Suche** mit **beschränkter Tiefe**.

# Naive tiefenbeschränkte Suche

**Function** vertexCover( $G = (V, E)$ ,  $k$ ) : Boolean

**if**  $|E| = 0$  **then return** true

**if**  $k = 0$  **then return** false

  pick any edge  $\{u, v\} \in E$

**return** vertexCover( $G - v, k - 1$ )  $\vee$   
    vertexCover( $G - u, k - 1$ )

Operation  $G - v$  removes node  $v$  and its incident edges

# Naive tiefenbeschränkte Suche – Korrektheit

```
Function vertexCover( $G = (V, E)$ ,  $k$ ) : Boolean
    if  $|E| = 0$  then return true           // triviales Problem
    if  $k = 0$  then return false         // unmögliches Problem
    pick any edge  $\{u, v\} \in E$    //  $u$  oder  $v$  müssen im cover sein !
    //Fallunterscheidung:
    return vertexCover( $G - v, k - 1$ ) ∨      // Fall  $v$  in cover
        vertexCover( $G - u, k - 1$ )           // Fall  $u$  in cover
```

# Naive tiefenbeschränkte Suche – Laufzeit

**Function** vertexCover( $G = (V, E), k$ ) : Boolean

```
if  $|E| = 0$  then return true // O(1)
if  $k = 0$  then return false // O(1)
pick any edge  $\{u, v\} \in E$  // O(1)
return vertexCover( $G - v, k - 1$ )  $\vee$  //  $O(n + m) + T(k - 1)$ 
    vertexCover( $G - u, k - 1$ ) //  $T(k - 1)$ 
```

Rekursionstiefe  $k \rightsquigarrow O(2^k)$  rekursive Aufrufe also

Laufzeit  $O(2^k(n + m))$ .

**Formaler:** Lösung der Rekurrenz  $T(k) = (n + m) + 2T(k - 1)$

# Kernbildung für Vertex Cover

Beobachtung:  $\forall v \in V : \text{degree}(v) > k \implies v \in \text{Lösung} \vee \text{unlösbar}$

**Function** kernelVertexCover( $G = (V, E), k$ ) : Boolean

**while**  $\exists v \in V : \text{degree}(v) > k$  **do**

**if**  $k = 0$  **then return** false

$G := G - v$

$k := k - 1$

    remove isolated nodes

**if**  $|E| > k^2$  **then return** false

**return** vertexCover( $G, k$ )

# Kernbildung für Vertex Cover – Korrektheit

Beobachtung:  $\forall v \in V : \text{degree}(v) > k \implies v \in \text{Lösung} \vee \text{unlösbar}$

**Function** kernelVertexCover( $G = (V, E), k$ ) : Boolean

```
while  $\exists v \in V : \text{degree}(v) > k$  do      // siehe Beobachtung
    if  $k = 0$  then return false                //  $m > k = 0 !$ 
     $G := G - v$                             //  $v$  muss in die Lösung!
     $k := k - 1$ 
    remove isolated nodes                  // nutzlose Knoten
    if  $|E| > k^2$  then return false    //  $\leq k$  nodes  $\times \leq k$  neighbors
    return vertexCover( $G, k$ )
```

# Kernbildung für Vertex Cover – Laufzeit

```
Function kernelVertexCover( $G = (V, E), k$ ) : Boolean
    while  $\exists v \in V : \text{degree}(v) > k$  do          //  $\leq k \times$ 
        if  $k = 0$  then return false                  //  $m \geq k > 0 !$ 
         $G := G - v$  //  $v$  muss in die Lösung!      //  $O(n)$ 
         $k := k - 1$ 

        remove isolated nodes                      // nutzlose Knoten
        // Insgesamt  $O(nk)$ 

        if  $|E| > k^2$  then return false //  $\leq k$  nodes  $\times \leq k$  neighbors
        return vertexCover( $G, k$ )                //  $O(2^k k^2)$ 

    Insgesamt  $O(nk + 2^k k^2)$                     Aufgabe:  $O(n + m + 2^k k^2)$ 
```

# Kernbildung für Vertex Cover – Beispiel

**Function** kernelVertexCover( $G = (V, E), k$ ) : Boolean

**while**  $\exists v \in V : \text{degree}(v) > k$  **do**

**if**  $k = 0$  **return** false

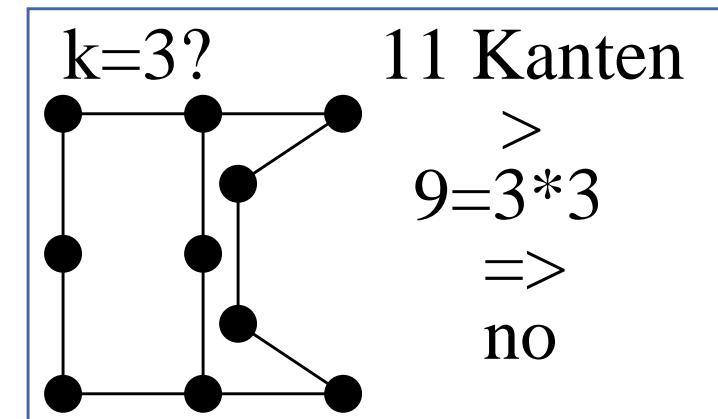
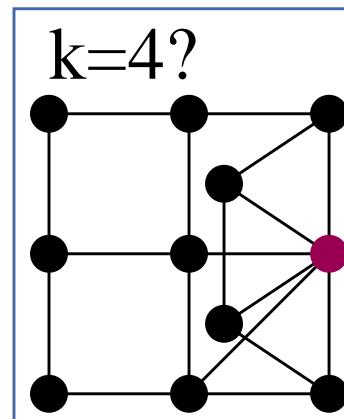
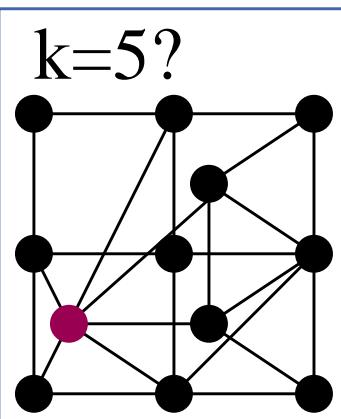
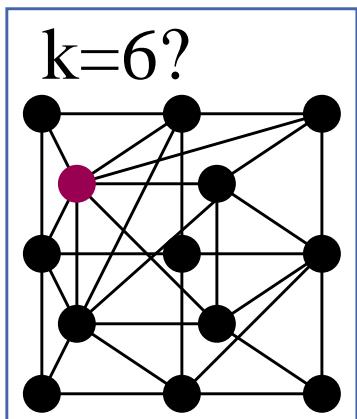
$G := G - v$

$k := k - 1$

remove isolated nodes

**if**  $|E| > k^2$  **then return** false

**return** vertexCover( $G, k$ )





# Reduktionsregeln

0: nicht im cover

1: OBdA Nachbar im cover

Aufgabe: vertex cover für Bäume?

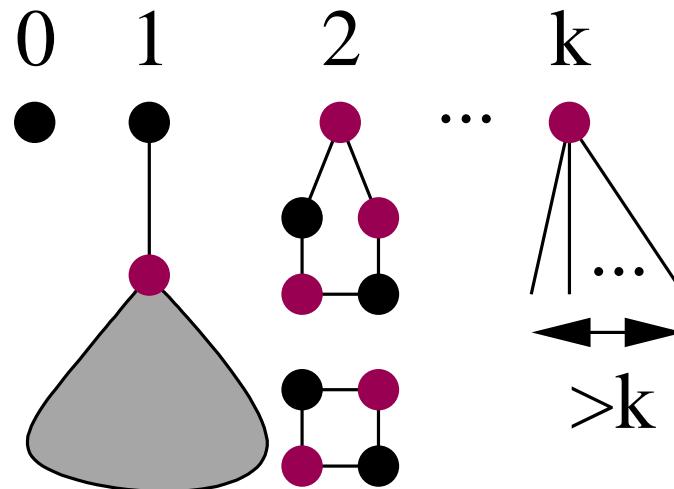
2: geht auch aber komplizierter. Aber,

trivial wenn alle Knoten Grad zwei haben

“Nimm jeden zweiten Knoten”

$> k$ : muss ins cover

Mehr Regeln ?





## Verbesserte tiefenbeschränkte Suche

**Function** vertexCover2( $G = (V, E)$ ,  $k$ ) : Boolean

**if**  $|E| = 0$  **then return** true // O(1)

**if**  $k = 0$  **then return** false // O(1)

**if**  $\exists v \in V : \text{degree}(v) = 1$  **then**

**return** vertexCover2( $G - \text{neighbor}(v)$ ,  $k - 1$ )

**if**  $\exists v \in V : \text{degree}(v) \geq 3$  **then**

**return** vertexCover2( $G - v$ ,  $k - 1$ )  $\vee$

vertexCover2( $G - \mathcal{N}(v)$ ,  $k - |\mathcal{N}(v)|$ )

**assert** all nodes have degree 2

**return** vertexCoverCollectionOfCycles( $G, k$ )

**Analyse:** Lösung der Rekurrenz

$$T(k) = (n+m) + T(k-1) + T(k-3) = \mathcal{O}((n+m)1.4656^k)$$

~~~ benutze erzeugende Funktionen

## Weitere Verbesserungen

- Kerne der Größe  $2k$  (wieder mittels Matching-Algorithmen)
- Reduziere Zeit pro rekursivem Aufruf auf  $O(1)$
- Detaillierte Fallunterscheidungen  $\rightsquigarrow$   
kleinere Konstante im exponentiellen Teil  
 $\rightsquigarrow O\left(1.2738^k + kn\right)$

[Chen Kanj Xia 2006]

## Zusammenfassung

- Wichtige Teilklassen NP-harter Probleme können polynomiell lösbar sein
- Kernbildung ist wichtige Vor/Zwischenverarbeitungsstrategie für Optimierungsproblem – auch polynomial lösbar. Zum Beispiel Max-Cardinality matching.
- Brute-force Algorithmen weniger brute-force machen bringt beweisbar exponentielle Beschleunigung.  $\rightsquigarrow$  nichttriviale Analyse- und Entwurfs-Techniken

# 10 Parallele Algorithmen

Schnupperkapitel.

Mehr in der **gleichnamigen Vorlesung**  
sowie im **Vertiefungsfach Parallelverarbeitung**

# Warum Parallelverarbeitung

Geschwindigkeitsteigerung:  $p$  Computer, die gemeinsam an einem Problem arbeiten, lösen es **bis zu**  $p$  mal so schnell. Aber, viele Köche verderben den Brei  $\rightsquigarrow$  gute Koordinationsalgorithmen

Energieersparnis: Zwei Prozessoren mit halber Taktfrequenz brauchen weniger als eine voll getakteter Prozessor.  
(Leistung  $\approx$  Spannung  $\cdot$  Taktfrequenz)

Speicherbeschränkungen von Einzelprozessoren

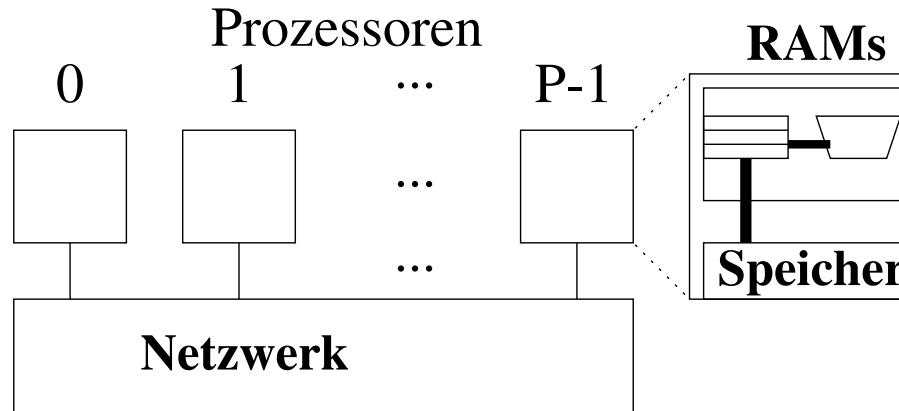
Kommunikationsersparnis: wenn Daten verteilt anfallen kann man sie auch verteilt (vor)verarbeiten

# Parallelverarbeitung am ITI Sanders

- Massiv paralleles Sortieren, Michael Axtmann
- Massiv parallele Graph-Algorithmen, Sebastian Lamm
- Fehlertoleranz, Demian Hespe
- Big-Data Framework Thrill, Timo Bingmann
- Shared Memory Datenstrukturen, Tobias Maier
- (Hyper)Graphpartitionierung, Sebastian Schlag & Yaroslav Akhremtsev
- Kommunikationseff. Alg., Lorenz Hübschle-Schneider
- SAT-Solving und Planungsprobl., Tomas Balyo & Dominik Schreiber
- Geometrische Algorithmen, Daniel Funke

## 10.1 Modell

# Nachrichtengekoppelte Parallelrechner



- Prozessoren sind RAMs
- asynchrone Programmabarbeitung
- Interaktion durch Nachrichtenaustausch

# Kostenmodell für Nachrichtenaustausch

Jedes PE kann  
gleichzeitig maximal eine Nachricht senden und empfangen.

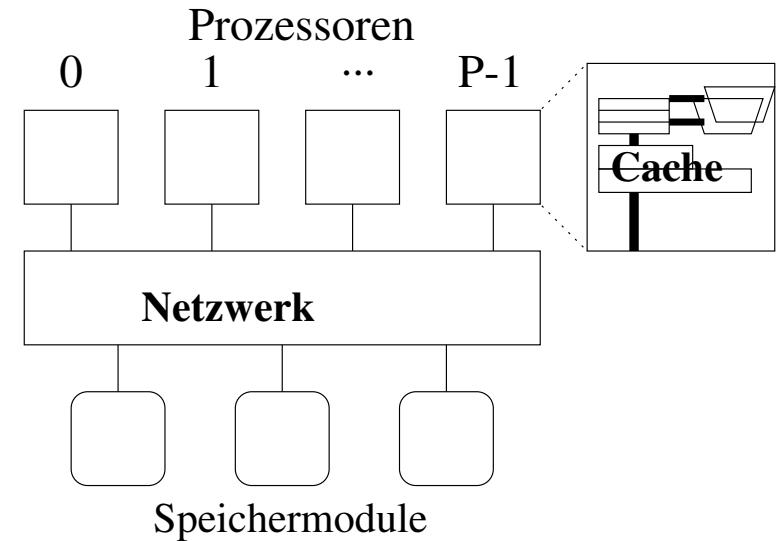
Bei Nachrichtenlänge  $\ell$  dauert das

$$T_{\text{comm}}(\ell) = T_{\text{start}} + \ell T_{\text{byte}}$$

- vollduplex
- Punkt-zu-Punkt
- vollständige Verknüpfung
- i.allg.  $T_{\text{start}} \gg T_{\text{byte}}$  – Alternative: Blockkommunikation **analog Sekundärspeichermodell**

# Warum **kein** (shared memory) Multicore-Modell

- Unklar wie man damit  
**skalierbaren Parallelismus** erreicht
- Unklares **Kostenmaß** bei  
Speicherzugriffskonflikten
- Gute Strategie für Parallelprogrammierung:  
**Verteilt entwerfen, vereint implementieren**
  - ~~> mit verteiltem Speicher decken wir den ganzen Bereich vom Multicore-Smartphone zum Superrechner ab und sind noch einigermaßen Nah an Cloud, Sensor- oder Peer-to-Peer-Netzen



# Formulierung paralleler Algorithmen

Gleicher Pseudocode wie immer.

Single Program Multiple Data Prinzip.

Der Prozessorindex wird genutzt um die Symmetrie zu brechen.

**Procedure** helloWorldParallel

    writeLineAtomic “Hallo, I am PE ” iProc “ out of ”  $p$  “processing elements”

Hallo, I am PE 0 out of 3 processing elements

Hallo, I am PE 2 out of 3 processing elements

Hallo, I am PE 1 out of 3 processing elements

# Analyse paralleler Algorithmen

Im Prinzip nur ein zusätzlicher Parameter:  $p$ .

Finde Ausführungszeit  $T(I, p)$ .

Problem: Interpretation.

Work:  $W = pT(p)$  ist ein Kostenmaß.

Span:  $T_\infty = \sup_p T(p)$  misst Parallelisierbarkeit.

(absoluter) Speedup:  $S = T_{\text{seq}}/T(p)$  Beschleunigung. Benutze  
**besten bekannten** sequentiellen Algorithmus. Relative  
Beschleunigung  $S_{\text{rel}} = T(1)/T(p)$  ist i.allg. was anderes!

Effizienz:  $E = S/p$ . Ziel:  $E \approx 1$  oder wenigstens  $E = \Theta(1)$ .

(Sinnvolles Kostenmaß?) „Superlineare Beschleunigung“:  $E > 1$ .  
(möglich?).

## 10.2 Beispiel: Assoziative Operationen (=Reduktion)

**Satz 1.** Sei  $\oplus$  ein assoziativer Operator, der in konstanter Zeit berechnet werden kann. Dann lässt sich

$$\bigoplus_{i < p} x_i := (\cdots ((x_0 \oplus x_1) \oplus x_2) \oplus \cdots \oplus x_{p-1})$$

in Zeit  $O(\log p)$  berechnen

Beispiele:  $+$ ,  $\cdot$ ,  $\max$ ,  $\min$ , ... (z.B. ? nichkommutativ?)

# Grundidee für $p = 2^k$ (oBdA?)

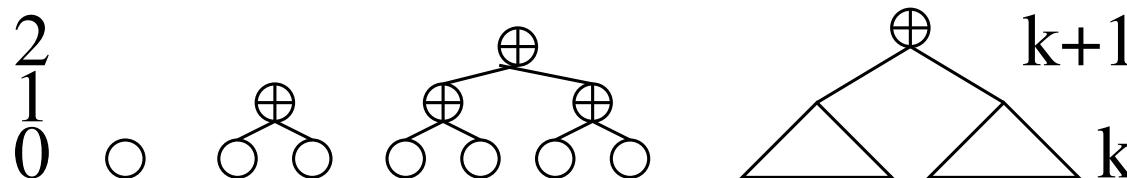
Induktion über  $k$ :

$k = 0$ : trivial

$k \rightsquigarrow k + 1$ :

$$\bigoplus_{i < 2^{k+1}} x_i = \underbrace{\bigoplus_{i < 2^k} x_i}_{\text{Tiefe } k+1} \oplus \underbrace{\bigoplus_{i < 2^k} x_{i+2^k}}_{\text{Tiefe } k \text{ (IA)}}$$

■



## Pseudocode

PE index  $i \in \{0, \dots, p - 1\}$

//Input  $x_i$  located on PE  $i$

active := 1

$s := x_i$

**for**  $0 \leq k < \lceil \log p \rceil$  **do**

**if** active **then**

**if** bit  $k$  of  $i$  **then**

            sync-send  $s$  to PE  $i - 2^k$

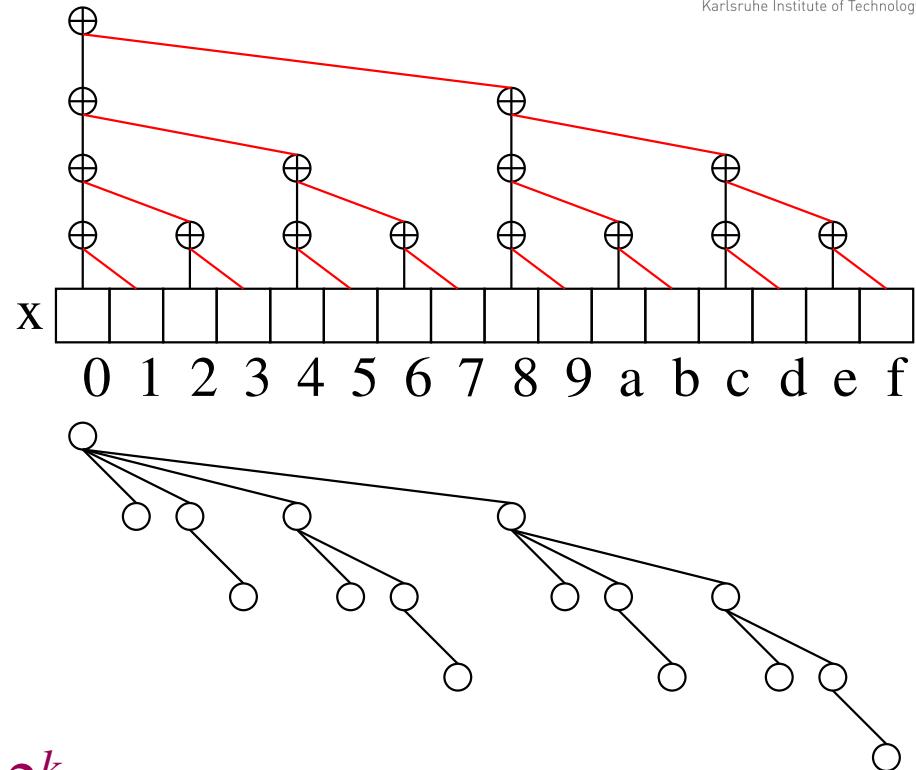
            active := 0

**else if**  $i + 2^k < p$  **then**

            receive  $s'$  from PE  $i + 2^k$

$s := s \oplus s'$

//result is in  $s$  on PE 0



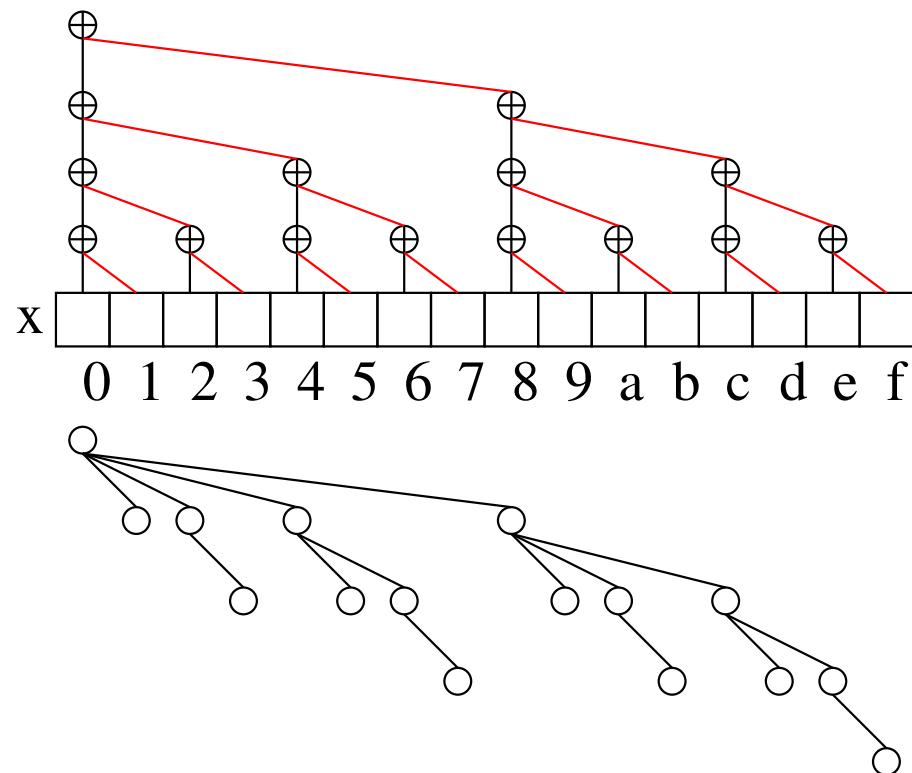
# Analyse

$n$  PEs

Zeit  $O(\log n)$

Speedup  $O(n/\log n)$

Effizienz  $O(1/\log n)$



# Weniger ist Mehr

$p$  PEs

Jedes PE addiert

$n/p$  Elemente sequentiell

Dann parallele Summe

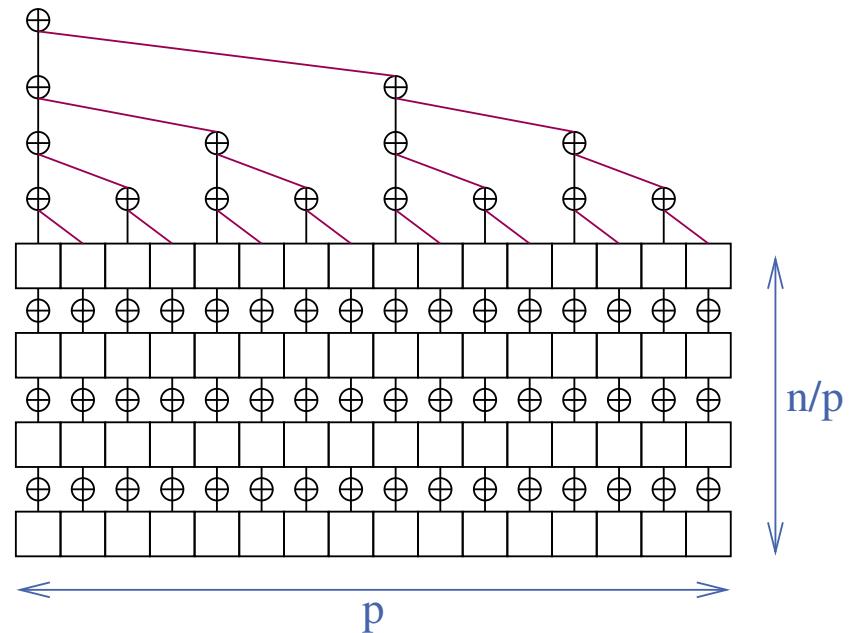
für  $p$  Teilsummen

Zeit  $T_{\text{seq}}(n/p) + \Theta(\log p)$

Effizienz

$$\frac{T_{\text{seq}}(n)}{p(T_{\text{seq}}(n/p) + \Theta(\log p))} = \frac{n}{p(n/p + \Theta(\log(p)))} = \frac{1}{1 + \Theta(p \log(p)) / n}$$

$\approx 1$  falls  $n \gg p \log p$



# Diskussion Reduktionsoperation

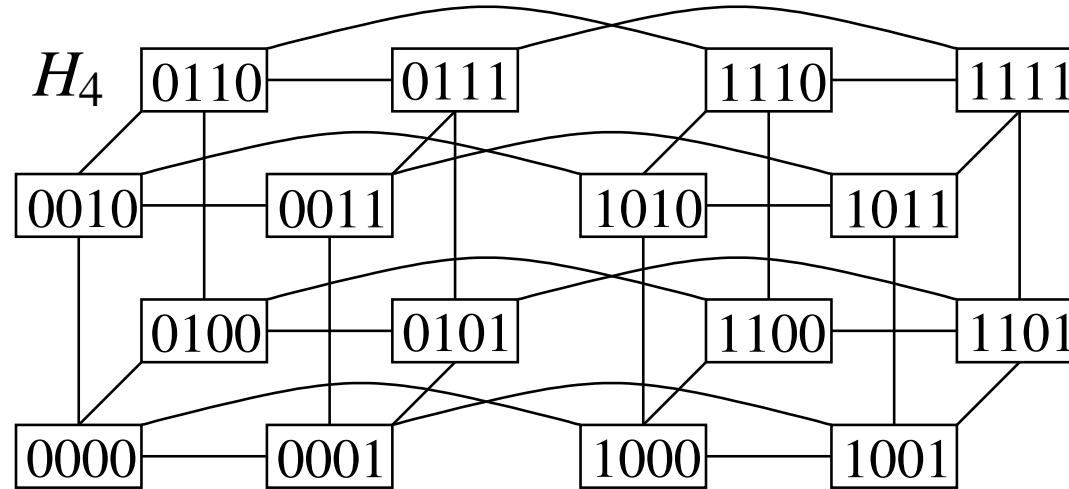
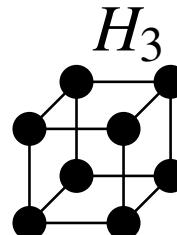
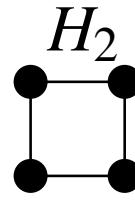
- Binärbaum führt zu logarithmischer Ausführungszeit
- Nützlich auf den meisten Modellen
- Brent's Prinzip: Ineffiziente Algorithmen werden durch Verringerung der Prozessorzahl effizient

# Hyperwürfel

hypercube

$H_0$  ●

$H_1$  ●

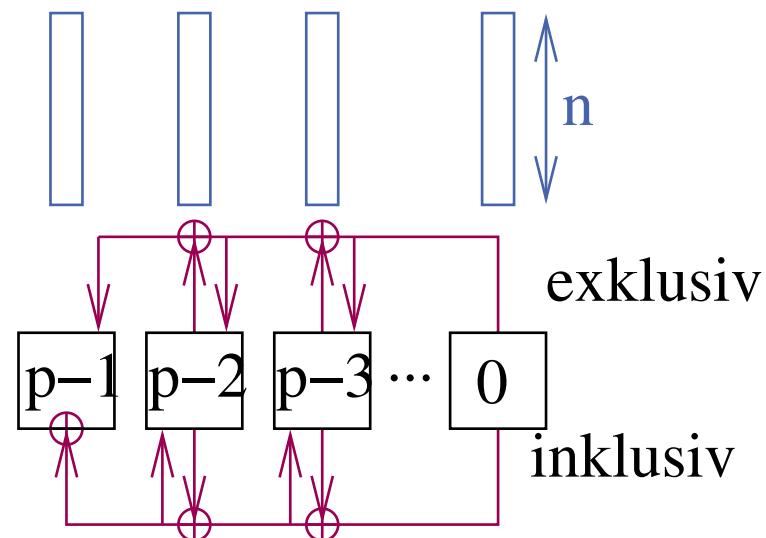


# Präfixsummen

Gesucht

$$x @ i := \bigoplus_{i' \leq i} m @ i'$$

(auf PE  $i$ ) Objekte der Länge  $\ell$



inhärent sequentiell ???

# Hyperwürfelalgorithmus

//view PE index  $i$  as a

// $d$ -bit bit array

**Function** hcPrefix( $m$ )

$x := \sigma := m$

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

**invariant**  $\sigma = \bigoplus_{j=i[k..d-1]}^{i[k..d-1]1^k} m @ j$

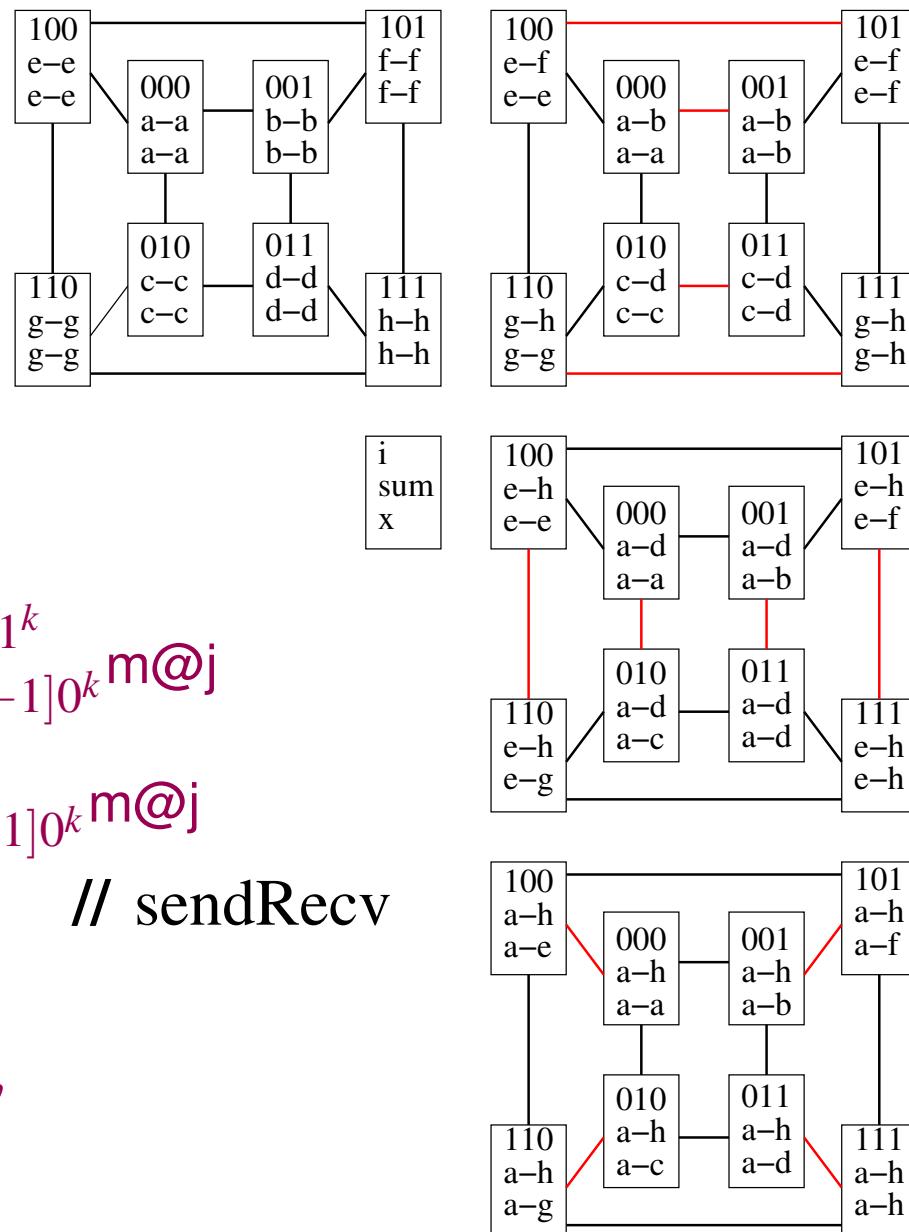
**invariant**  $x = \bigoplus_{j=i[k..d-1]}^{i[j..d-1]0^k} m @ j$

$y := \sigma @ (i \text{ xor } 2^k)$  // sendRecv

$\sigma := \sigma \oplus y$

**if**  $i[k] = 1$  **then**  $x := x \oplus y$

**return**  $x$



# Analyse

$$T_{\text{prefix}} = O((T_{\text{start}} + \ell T_{\text{byte}}) \log p)$$

## Problemchen:

Nichtoptimal bei  $\ell T_{\text{byte}} > T_{\text{start}}$  (analog schon bei Reduktion)

siehe Spezialvorlesung

$$\rightsquigarrow O(T_{\text{start}} \log P + \ell T_{\text{byte}})$$

## 10.3 Sortieren

- Paralleles Quicksort
- Paralleles Mehrwege-Mergesort
- Hier nicht: binäres Mergesort, Radixsort,...

# Paralleles Quicksort

## Sequentiell (vereinfacht)

**Procedure** qSort( $d[]$ ,  $n$ )

**if**  $n = 1$  **then return**

select a **pivot**  $v$

reorder the elements in  $d$  such that

$$d_0 \cdots d_{k-1} \leq v < d_k \cdots d_{n-1}$$

qSort( $[d_0, \dots, d_{k-1}]$ ,  $k$ )

qSort( $[d_{k+1}, \dots, d_{n-1}]$ ,  $n - k - 1$ )

## Anfänger-Parallelisierung

Parallelisierung der rekursiven Aufrufe.

$$T_{\text{par}} = \Omega(n)$$

- Sehr begrenzter Speedup
- Schlecht für distributed Memory

## Theoretiker-Parallelisierung

Zur Vereinfachung:  $n = p$ .

Idee: Auch die Aufteilung parallelisieren.

1. Ein PE stellt den Pivot (z.B. zufällig).
2. Broadcast
3. Lokaler Vergleich
4. „Kleine“ Elemente durchnummernieren (Präfix-Summe)
5. Daten umverteilen
6. Prozessoren aufspalten
7. Parallel Rekursion

## Theoretiker-Parallelisierung

//Let  $i \in 0..p - 1$  and  $p$  denote the ‘local’ PE index and partition size

**Procedure** theoQSort( $d, i, p$ )

**if**  $p = 1$  **then return**

$r :=$  random element from  $0..p - 1$  // same value in entire partition

$v := d @ r$  // broadcast **pivot**

$f := d \leq v$  // 1 iff  $d$  is on left side, 0 otherwise

$j := \sum_{k=0}^i f @ k$  // **prefix sum**, count elements on left side

$p' := j @ (p - 1)$  // broadcast, result is border index

**if**  $f$  **then** send  $d$  to PE  $j - 1$

**else** send  $d$  to PE  $p' + i - j$  //  $i - j = \sum_{k=0}^i d @ k > v$

receive  $d$

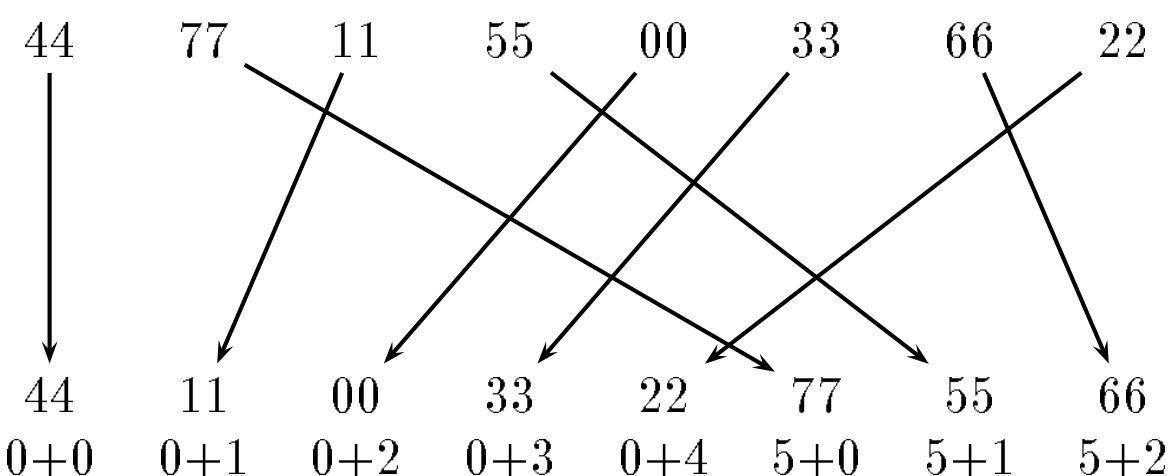
**if**  $i < p'$  **then** join left partition; qsort( $d, i, p'$ )

**else** join right partition; qsort( $d, i - p', p - p'$ )

# Beispiel

pivot  $v = 44$

| PE Nummer                    | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   |
|------------------------------|-----|-----|-----|-----|-----|-----|-----|-----|
| Nr. d. Elemente $\leq$ Pivot | 0   | 1   |     | 2   | 3   |     | 4   |     |
| Nr. d. Elemente $>$ Pivot    | 0   |     | 1   |     |     | 2   |     |     |
| Wert vorher                  | 44  | 77  | 11  | 55  | 00  | 33  | 66  | 22  |
|                              |     |     |     |     |     |     |     |     |
| Wert nachher                 | 44  | 11  | 00  | 33  | 22  | 77  | 55  | 66  |
| PE Nummer                    | 0+0 | 0+1 | 0+2 | 0+3 | 0+4 | 5+0 | 5+1 | 5+2 |



```
int pQuickSort(int item, MPI_Comm comm)
{ int iP, nP, small, allSmall, pivot;
  MPI_Comm newComm; MPI_Status status;
  MPI_Comm_rank(comm, &iP); MPI_Comm_size(comm, &nP);

  if (nP == 1) { return item; }
  else {
    pivot = getPivot(item, comm, nP);
    count(item <= pivot, &small, &allSmall, comm, nP);
    if (item <= pivot) {
      MPI_Bsend(&item, 1, MPI_INT, small - 1 , 8, comm);
    } else {
      MPI_Bsend(&item, 1, MPI_INT, allSmall+iP-small, 8, comm);
    }
    MPI_Recv(&item, 1, MPI_INT, MPI_ANY_SOURCE, 8, comm, &status);
    MPI_Comm_split(comm, iP < allSmall, 0, &newComm);
    return pQuickSort(item, newComm); } }
```

```
/* determine a pivot */

int getPivot(int item, MPI_Comm comm, int nP)
{ int pivot = item;
  int pivotPE = globalRandInt(nP); /* from random PE */
  /* overwrite pivot by that one from pivotPE */
  MPI_Bcast(&pivot, 1, MPI_INT, pivotPE, comm);
  return pivot;
}

/* determine prefix-sum and overall sum over value */
void
count(int value,int *sum,int *allSum,MPI_Comm comm,int nP)
{ MPI_Scan(&value, sum, 1, MPI_INT, MPI_SUM, comm);
  *allSum = *sum;
  MPI_Bcast(allSum, 1, MPI_INT, nP - 1, comm);
}
```

# Analyse

- pro Rekursionsebene:
  - $2 \times$  broadcast
  - $1 \times$  Präfixsumme
- ~~~ Zeit  $O(T_{\text{start}} \log p)$
- erwartete Rekursionstiefe:  $O(\log p)$

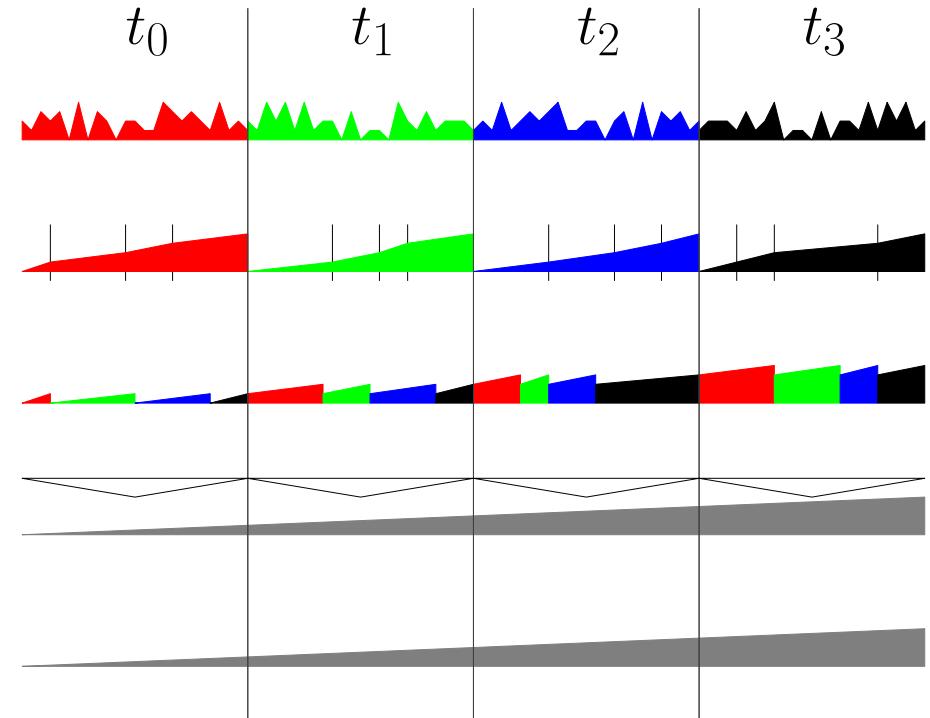
Erwartete Gesamtzeit:  $O(T_{\text{start}} \log^2 p)$

## Verallgemeinerung für $n \gg p$ nach Schema F?

- Jedes PE hat i.allg. „große“ und „kleine“ Elemente.
- Aufteilung geht nicht genau auf
- Präfixsummen weiterhin nützlich
- Unterm Strich ist Zeit  $O\left(\frac{n \log n}{p} + \log^2 p\right)$  möglich
- Bei verteiltem Speicher stört, dass jedes Element  $\Omega(\log p)$  mal transportiert wird.  
 $\rightsquigarrow \dots \rightsquigarrow$  Zeit  $O\left(\frac{n}{p}(\log n + T_{\text{byte}} \log p) + T_{\text{start}} \log^2 p\right)$

# Paralleles Sortieren durch Mehrwegemischen

1.  $p$  Prozessoren sortieren  
je  $n/p$  Elemente lokal
2. Finde pivots so dass  
 $n/p$  Elemente zwischen  
zwei benachbarten pivots liegen

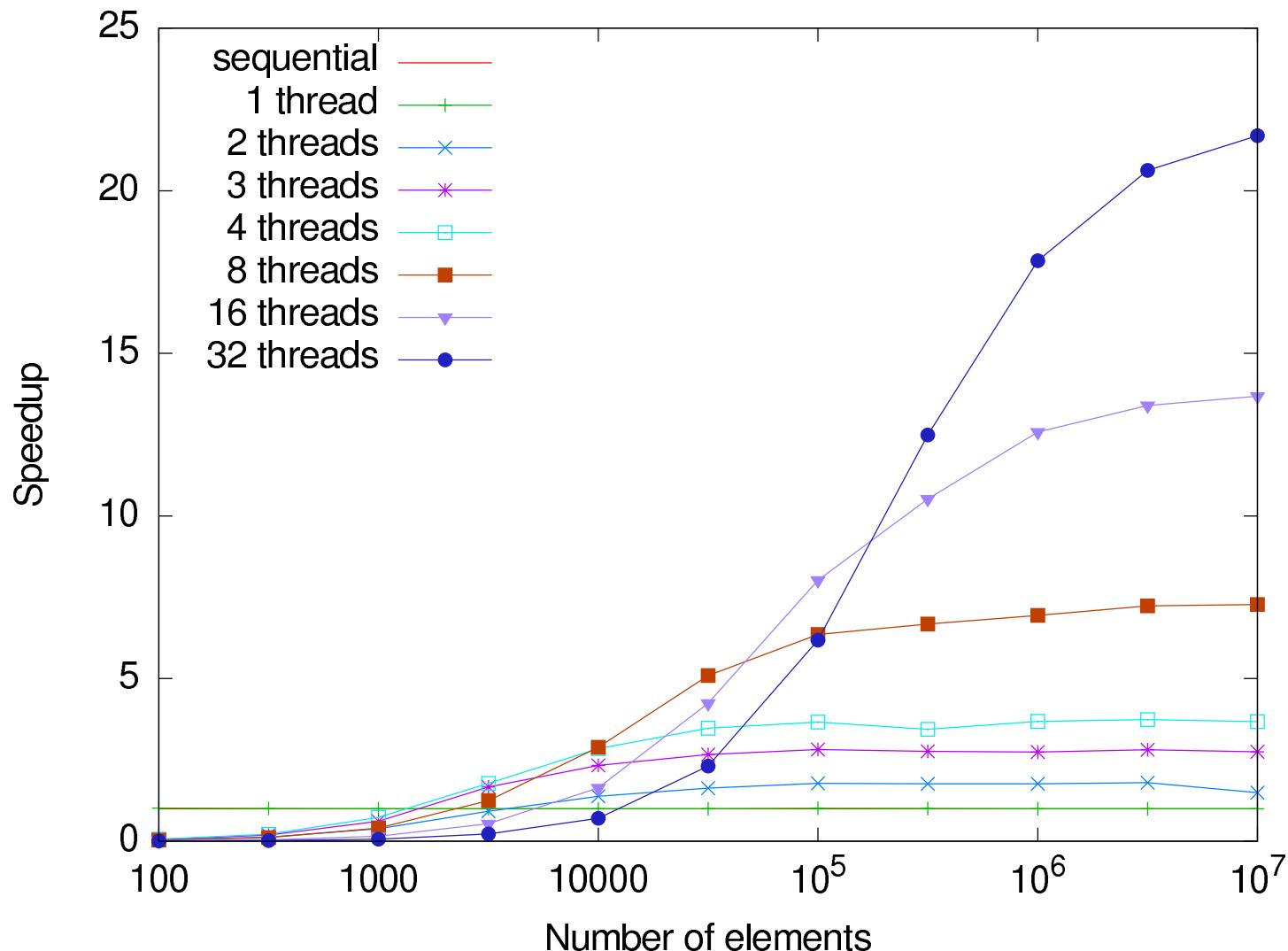


3. Jeder Prozessor macht  
Mehrwegemischen für alle  
Elemente zwischen zwei benachbarten Pivots.

Mehr in “Parallel Algorithmen”

# Messungen Sparc T1 – 8 Kerne, 128 bit Elemente

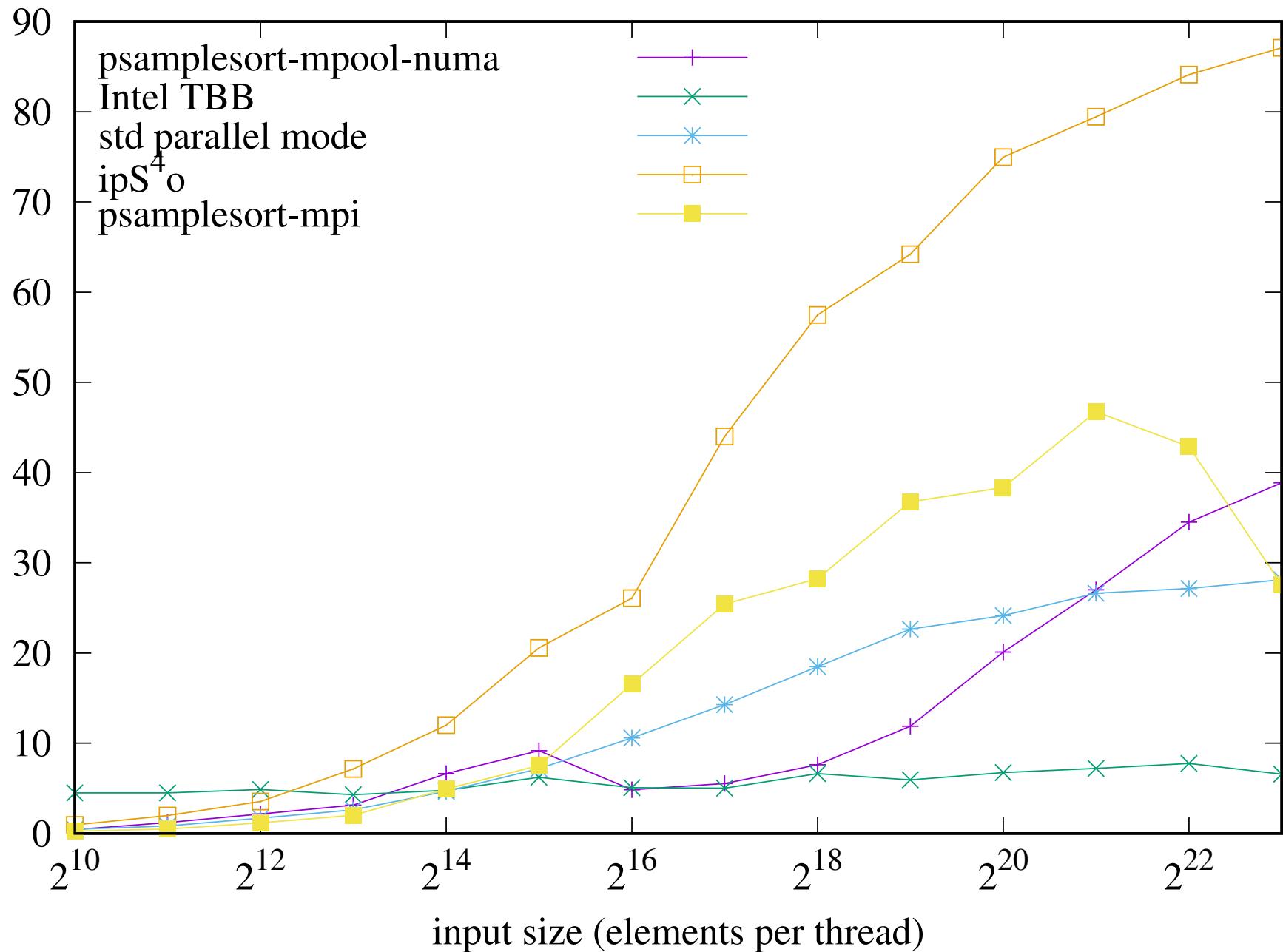
[Sanders Singler 2007]



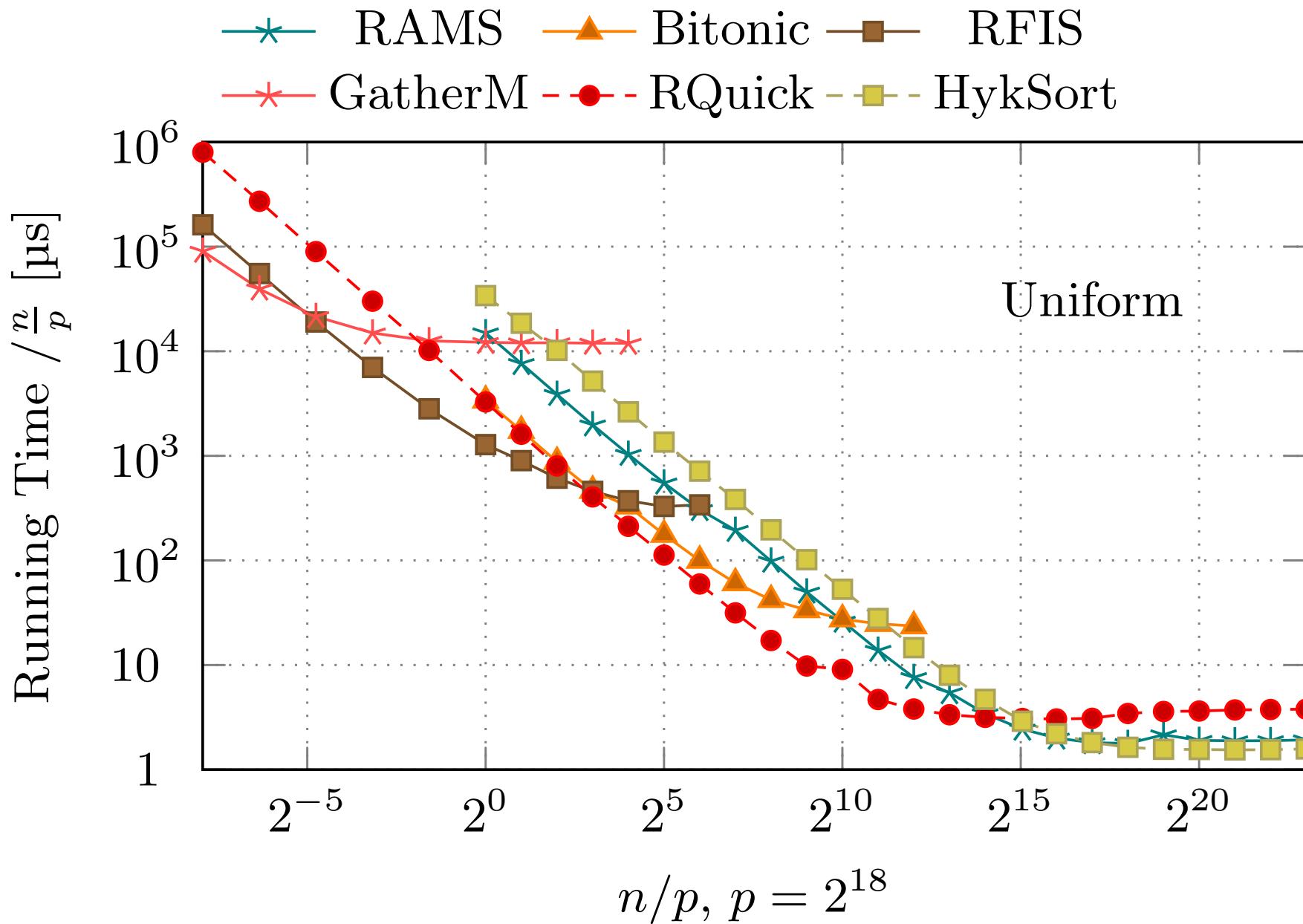
## Mehr zu parallelem Sortieren

- Theoretikeralgorithmen mit Laufzeit  $O(\log p)$  bei  $p = n$
- Praktikable Algorithmen mit Laufzeit  $O(\log p)$  für  $n = O(\sqrt{p})$  –  
wenn schnell wichtiger als Effizient ist, z.B. base case
- Sample sort:  $k$ -Wegeverallgemeinerung von quicksort
- Paralleles externes Sortieren

# Experiments Speedup on 4× Intel E7-8890 v3 – 72 cores



# Messungen Axtmann Sanders ALENEX 2017



# Mehr zu parallelen Algorithmen – Parallelisierung der Basic Toolbox ?

Verteilte Hashtabellen: geht oft teuer  $\rightsquigarrow$  [Maier S Dementiev 16]

Prioritätslisten, Suchbäume: ähnliches Problem. Am ehesten **batched updates**  $\rightsquigarrow$  [Hübschle-Schn. S Müller 16][Akhremtsev S 16]

BFS: OK bei kleinem Graphdurchmesser

DFS:  $\approx$  inhärent nichtparallelisierbar

kürzeste Wege: ähnlich BFS. Aber all-to-all, Vorberechnungen OK,  
multiobjective OK [Mandow S 13]

MST: Ja! Aber u.U. große konstante Faktoren

Optimierungstechniken: dynamische Programmierung OK. Greedy???  
LP schwierig, Metaheuristiken teils OK

# Mehr zu parallelen Algorithmen – Die parallele Basic Toolbox

- Kollektive Kommunikation: **Reduktion**, **Präfixsumme**, allg.  
Nachrichtenaustaus, Gossiping,...
- Lastverteilung

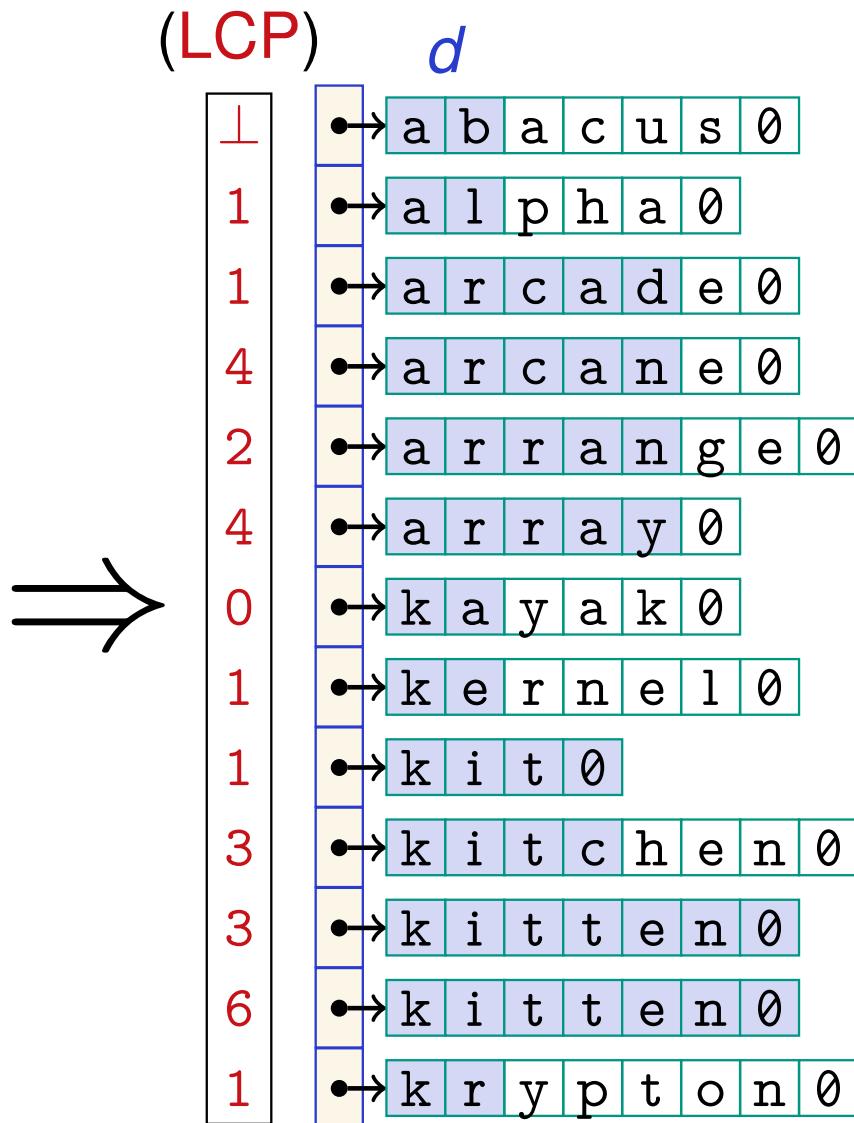
# 11 Stringology

## (Zeichenkettenalgorithmen)

- Strings sortieren
- Patterns suchen
  - Pattern vorverarbeiten
  - Text vorverarbeiten
    - \* Invertierte Indizes
    - \* Suffix Trees / Suffix Arrays
- Datenkompression
- Pattern suchen in komprimierten Indizes

# Strings Sortieren

|     |           |
|-----|-----------|
| • → | array 0   |
| • → | kit 0     |
| • → | arrange 0 |
| • → | kayak 0   |
| • → | kernel 0  |
| • → | kitchen 0 |
| • → | kitten 0  |
| • → | arcade 0  |
| • → | kitten 0  |
| • → | abacus 0  |
| • → | krypton 0 |
| • → | alpha 0   |
| • → | arcane 0  |



Eingabe:  $n$  Strings mit  $N$  Zeichen insgesamt.

# Sortieren: Most Significant Digit Radix Sort

array

kit

arrange

kayak

kernel

kitchen

kitten

arcade

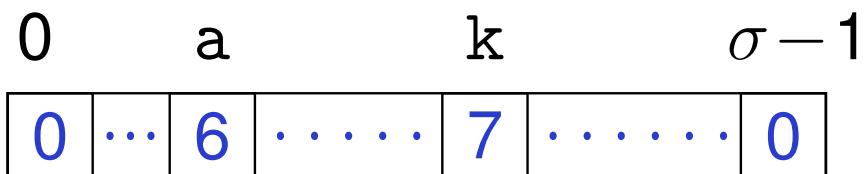
kite

abacus

krypton

alpha

arcadic



# Sortieren: Most Significant Digit Radix Sort

array

kit

arrange

kayak

kernel

kitchen

kitten

arcade

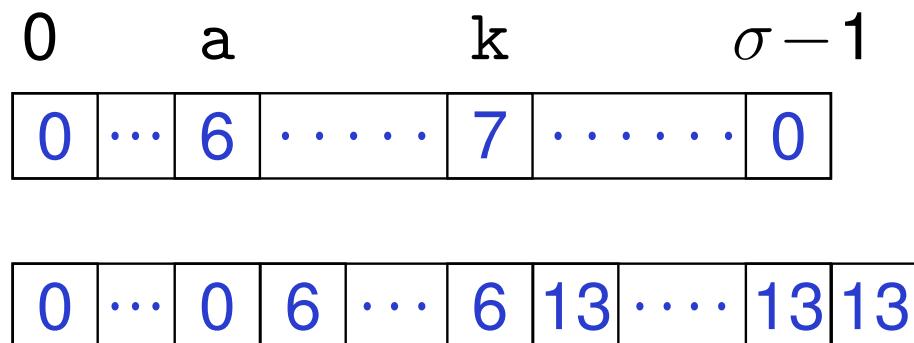
kite

abacus

krypton

alpha

arcadic

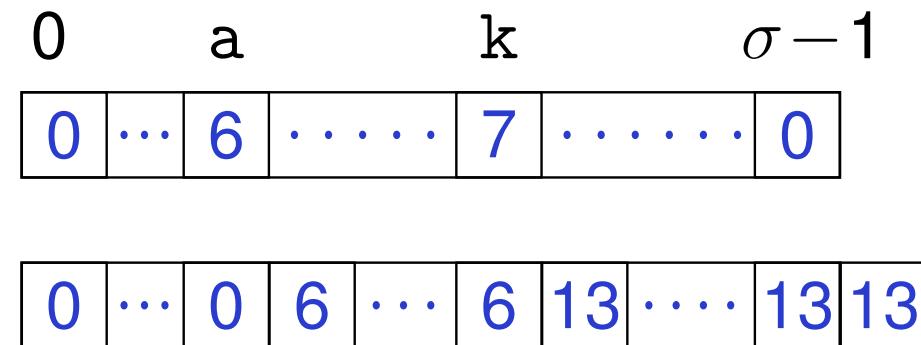


# Sortieren: Most Significant Digit Radix Sort

---

|                |
|----------------|
| array          |
| kit            |
| arrange        |
| kayak          |
| kernel         |
| <u>kitchen</u> |
| kitten         |
| arcade         |
| kite           |
| abacus         |
| krypton        |
| alpha          |
| arcadic        |

---

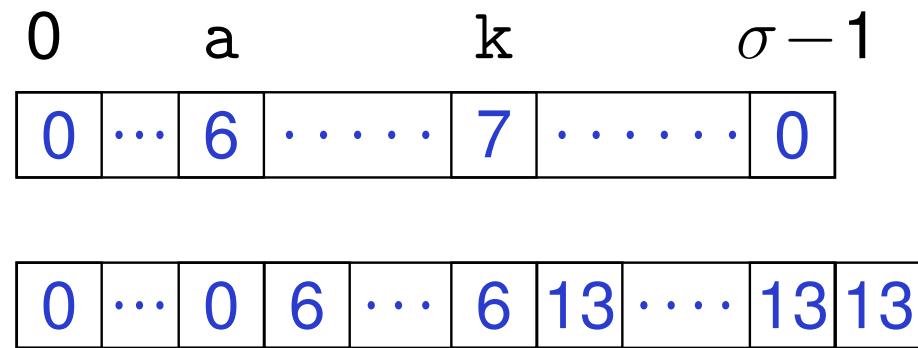


# Sortieren: Most Significant Digit Radix Sort

---

|               |
|---------------|
| a r r a y     |
| a r r a n g e |
| a r c a d e   |
| a b a c u s   |
| a l p h a     |
| a r c a i c   |
| k i t         |
| k a y a k     |
| k e r n e l   |
| k i t c h e n |
| k i t t e n   |
| k i t e       |
| k r y p t o n |

---



- Laufzeit:  
 $\mathcal{O}(d + r\sigma + n \log \sigma)$
- Varianten: out-of-place und in-place!

# Sortieren: Most Significant Digit Radix Sort

|         |
|---------|
| array   |
| kit     |
| arrange |
| kayak   |
| kernel  |
|         |
| kitchen |
| kitten  |
| arcade  |
| kite    |
|         |
| abacus  |
|         |
| krypton |
| alpha   |
| arcadic |

| 0     | a   | k   | $\sigma - 1$ |
|-------|-----|-----|--------------|
| 0     | ... | 6   | .....        |
| 0     | ... | 6   | 13           |
| $p_1$ | 0   | ... | 2            |
| $p_2$ | 0   | ... | 2            |
| $p_3$ | 0   | ... | 2            |

# Sortieren: Most Significant Digit Radix Sort

|         |
|---------|
| array   |
| kit     |
| arrange |
| kayak   |
| kernel  |
|         |
| kitchen |
| kitten  |
| arcade  |
| kite    |
|         |
| abacus  |
|         |
| krypton |
| alpha   |
| arcadic |

|  | 0 | a   | k | $\sigma - 1$ |
|--|---|-----|---|--------------|
|  | 0 | ... | 6 | .....        |
|  | 0 | ... | 6 | 7            |
|  | 0 | ... | 6 | .....        |
|  | 0 | ... | 6 | 0            |

|       |   |     |   |       |   |       |   |
|-------|---|-----|---|-------|---|-------|---|
| $p_1$ | 0 | ... | 2 | ..... | 3 | ..... | 0 |
| $p_2$ | 0 | ... | 2 | ..... | 3 | ..... | 0 |
| $p_3$ | 0 | ... | 2 | ..... | 1 | ..... | 0 |

|       |   |     |   |   |     |    |    |      |    |    |
|-------|---|-----|---|---|-----|----|----|------|----|----|
| $p_1$ | 0 | ... | 0 | 6 | ... | 6  | 13 | .... | 13 | 13 |
| $p_2$ | 0 | ... | 2 | 6 | ... | 9  | 13 | .... | 13 | 13 |
| $p_3$ | 0 | ... | 4 | 6 | ... | 12 | 13 | .... | 13 | 13 |

# Sortieren: Most Significant Digit Radix Sort

---

**Algorithm :** Sequential Radix Sort “CE0”, adapted from [KR08; Ran07]

---

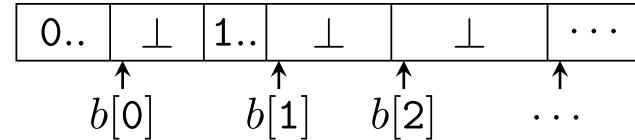
```

1 Function RadixSortCE0( $\mathcal{S}, h$ )
  Input :  $\mathcal{S} = [s_0, \dots, s_{n-1}]$  an array of  $n$  strings with common prefix  $h$ .
   $c := [0, \dots, 0]$  // Allocate  $|\Sigma|$  integer counters initialized with zero,
  for  $i = 0, \dots, n - 1$  do  $c[s_i[h]]++$  // and count character occurrences.
   $b := [0, \perp, \dots, \perp]$  // Calculate exclusive prefix sum of counters
  for  $i = 1, \dots, n - 1$  do  $b[i] := b[i - 1] + c[i - 1]$  // as bucket pointers.
   $\mathcal{T} := \text{allocate}(n, \text{string pointer})$  // Allocate temporary array for sorted output.
  for  $i = 0, \dots, n - 1$  do // Reorder
     $\mathcal{T}[b[s_i[h]]] := \text{move}(s_i)$  // into
     $b[s_i[h]]++$  // buckets
  copy( $\mathcal{T} \rightarrow \mathcal{S}$ ), deallocate( $\mathcal{T}, b$ ).
   $x := c[0]$  // Track beginning of bucket as  $x$ ,
  for  $i = 1, \dots, |\Sigma| - 1$  do // recurse into every unfinished bucket,
    StringSort( $\mathcal{S}[x..x + c[i]]$ ,  $h + 1$ ) // except for the first (zero-termination),
     $x := x + c[i]$  // which contains all fully sorted strings.

  Output : The array  $\mathcal{S}$  is fully sorted lexicographically.

```

---



# Strings Sortieren: Multikey Quicksort

Auch: MKQS / ternary quicksort

**Function** mkqSort( $S$  : Sequence **of** String,  $\ell$  :  $\mathbb{N}$ ) : Sequence **of** String

**assert**  $\forall e, e' \in S : e[1..\ell-1] = e'[1..\ell-1]$

**if**  $|S| \leq 1$  **then return**  $S$  // base case

pick  $p \in S$  uniformly at random // pivot string

**return** concatenation of  $\text{mkqSort}(\langle e \in S : e[\ell] < p[\ell] \rangle, \ell),$

$\text{mkqSort}(\langle e \in S : e[\ell] = p[\ell] \rangle, \ell + 1)$ , and

$\text{mkqSort}(\langle e \in S : e[\ell] > p[\ell] \rangle, \ell)$

- Laufzeit:  $O(|S| \log |S| + \sum_{t \in S} |t|)$
- genauer:  $O(|S| \log |S| + d)$  ( $d$ : Summe der eindeutigen Präfixe)
- Übung: **in-place**!

# Strings Sortieren: Multikey Quicksort

S A A L  
B I E N E  
E H R E  
H A U S  
A R M  
M I E T E  
T A S S E  
M O R D  
H A N D  
S E E  
H U N D  
H A L L E  
N A C H T

# Strings Sortieren: Multikey Quicksort

S A A L  
B I E N E  
E H R E  
H A U S  
A R M  
M I E T E  
T A S S E  
M O R D  
*p* H A N D  
S E E  
H U N D  
H A L L E  
N A C H T

# Strings Sortieren: Multikey Quicksort

B I E N E

E H R E

A R M

H A U S

H A N D

H U N D

H A L L E

S A A L

T A S S E

M I E T E

M O R D

S E E

N A C H T

# Strings Sortieren: Multikey Quicksort

|          |           |                  |
|----------|-----------|------------------|
|          | B I E N E | A R M            |
| <i>p</i> | E H R E   | B I E N E        |
|          | A R M     | E H R E          |
|          | H A U S   | H A U S          |
|          | H A N D   | <i>p</i> H A N D |
|          | H U N D   | H U N D          |
|          | H A L L E | H A L L E        |
|          | S A A L   | S A A L          |
|          | T A S S E | T A S S E        |
|          | M I E T E | M I E T E        |
|          | M O R D   | M O R D          |
|          | S E E     | S E E            |
|          | N A C H T | N A C H T        |

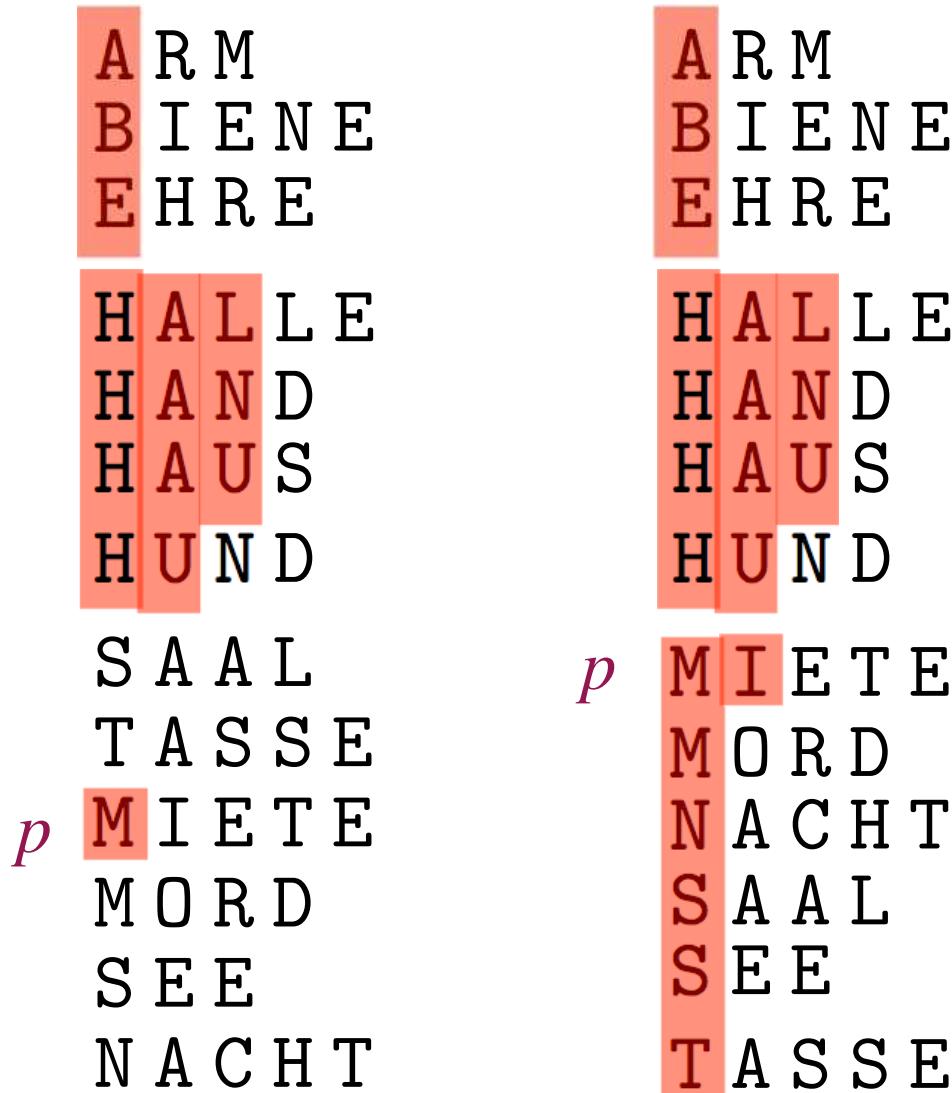
# Strings Sortieren: Multikey Quicksort

|          |           |                  |
|----------|-----------|------------------|
|          | A R M     | A R M            |
|          | B I E N E | B I E N E        |
|          | E H R E   | E H R E          |
|          | H A U S   | H A U S          |
| <i>p</i> | H A N D   | <i>p</i> H A N D |
|          | H U N D   | H A L L E        |
|          | H A L L E | H U N D          |
|          | S A A L   | S A A L          |
|          | T A S S E | T A S S E        |
|          | M I E T E | M I E T E        |
|          | M O R D   | M O R D          |
|          | S E E     | S E E            |
|          | N A C H T | N A C H T        |

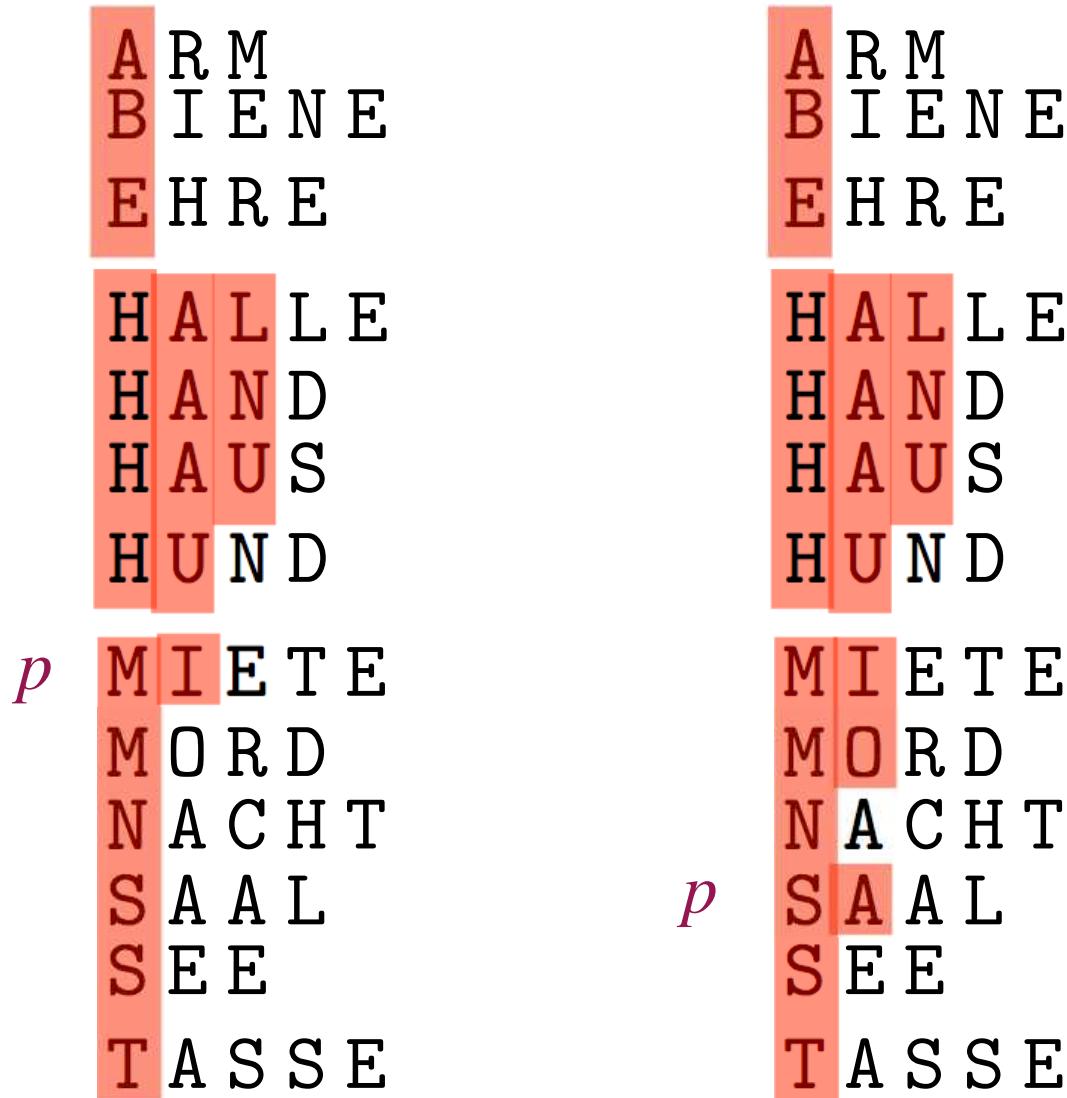
# Strings Sortieren: Multikey Quicksort

|           |             |
|-----------|-------------|
| A R M     | A R M       |
| B I E N E | B I E N E   |
| E H R E   | E H R E     |
| H A U S   | H A L L E   |
| p H A N D | H A N D     |
| H A L L E | H A U S     |
| H U N D   | H U N D     |
| S A A L   | S A A L     |
| T A S S E | T A S S E   |
| M I E T E | p M I E T E |
| M O R D   | M O R D     |
| S E E     | S E E       |
| N A C H T | N A C H T   |

# Strings Sortieren: Multikey Quicksort



# Strings Sortieren: Multikey Quicksort



# Strings Sortieren (ohne Endzeichen)

**Function**  $\text{mkqSort}(S : \text{Sequence of String}, \ell : \mathbb{N}) : \text{Sequence of String}$

**if**  $|S| \leq 1$  **then return**  $S$

$S_{\perp} \leftarrow \langle e \in S : |e| = \ell \rangle; S \leftarrow S \setminus S_{\perp}$

select pivot  $p \in S$

$S_{<} \leftarrow \langle e \in S : e[\ell] < p[\ell] \rangle$

$S_{=} \leftarrow \langle e \in S : e[\ell] = p[\ell] \rangle$

$S_{>} \leftarrow \langle e \in S : e[\ell] > p[\ell] \rangle$

**return** concatenation of  $S_{\perp}$ ,

$\text{mkqSort}(S_{<}, \ell)$ ,

$\text{mkqSort}(S_{=}, \ell + 1)$ , and

$\text{mkqSort}(S_{>}, \ell)$

# Strings Sortieren – Laufzeitanalyse

Hauptarbeit in den Buchstabenvergleichen. Zwei Fälle:

- $e[\ell] = p[\ell]$ : Ordne den Vergleich dem Zeichen  $e[\ell]$  zu.
  - $e[\ell]$  wird danach nicht mehr betrachtet (Rekursion mit  $\ell + 1$ )
  - Maximale Vergleichsanzahl pro String  $e$ ? Maximale Länge des längstes gemeinsamen Präfix von  $e$  mit  $e' \in S$ .
- $e[\ell] \neq p[\ell]$ : Ordne den Vergleich dem String  $e$  zu.
  - $e$  wird zu  $S_<$  oder  $S_>$  zugeordnet. Mit optimaler Pivotwahl sind beide Mengen höchstens  $|S|/2$ .
  - Nach höchstens  $\log |S|$  Schritten ist  $e$  richtig sortiert.



# Strings Sortieren: Algorithmen-Übersicht

## Sequentielle Basis-Algorithmen

- Radix Sort  $O(d + n \log \sigma)$  [McIlroy et al. '95]
- Multikey Quicksort  $O(d + n \log n)$  exp. [Bentley, Sedgewick '97]
- Burstsot  $O(d + n \log \sigma)$  exp. [Sinha, Zobel '04]
- Binary LCP-Mergesort  $O(d + n \log n)$  [Ng, Kakehi '08]

## Theoretische Parallel Algorithmen

- “Optimal Parallel String Algorithms: . . .” [Hagerup '94]  
 $O(\log N / \log \log N)$  time and  $O(N \log \log N)$  work on CRCW PRAM

## Praktische Parallel und neue Basis-Algorithmen

- Parallel Super Scalar String Sample Sort ( $pS^5$ ) [B, Sanders, ESA'13]
- Parallel  $K$ -way LCP-aware Merge(sort) [B, et al. Algorithmica'17]



# Vergleich Sequentielle Algorithmen

**Experiment [B'18]:** Vergleich von 39 sequentiellen String Sorting Algorithmen und getunten Varianten auf sieben Eingaben und sechs Maschinen.

Unten: Repräsentative Auswahl der Basis-Algorithmen.

| Rang | Algorithmus        | GeoM | Rang | Algorithmus              | GeoM |
|------|--------------------|------|------|--------------------------|------|
| 1    | KR.radixsort-CE6   | 1.27 | 10   | B.Seq-S <sup>5</sup> -UI | 1.67 |
| 2    | KRB.radixsort-CE3s | 1.28 | 14   | R.burstsort-vec          | 1.72 |
| 3    | KR.radixsort-CE7   | 1.28 | 16   | SZ.burstsortA            | 1.78 |
| 4    | KRB.radixsort-Cl3s | 1.33 | 23   | BS.mkqs                  | 2.36 |
| 5    | R.mkqs-cache8      | 1.34 | 28   | NK.LCP-Mergesort         | 2.96 |
| 6    | KR.radixsort-CE2   | 1.49 | 34   | MBM.radixsort            | 4.82 |
| 7    | KR.radixsort-DB    | 1.55 | 35   | AN.ForwardRadix16        | 4.88 |

## Ergebnisse:

- Hardware-spezifische Beschleunigungen sind sehr wichtig.
- Cachen von Zeichen reduziert Random-Zugriffe aber kostet Speicher.

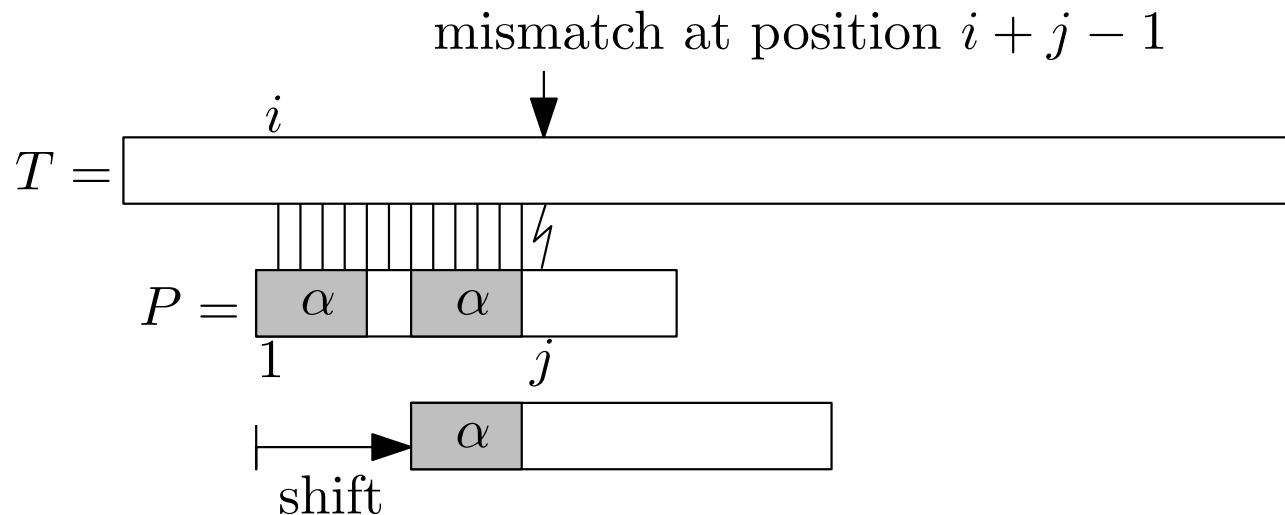
# Naives Pattern Matching

- Aufgabe: Finde alle Vorkommen von  $P$  in  $T$ 
  - $n$ : Länge von  $T$
  - $m$ : Länge von  $P$
- naiv in  $O(nm)$  Zeit

```
i, j := 1                                // indexes in T and P
while i ≤ n - m + 1
    while j ≤ m and ti+j-1 = pj do j++  // compare characters
    if j > m then print "P occurs at position i in T"
    i++                                     // advance in T
    j := 1                                    // restart
```

# Knuth-Morris-Pratt (1977)

- besserer Algorithmus in  $O(n + m)$  Zeit
- Idee: beachte bereits gematchten Teil



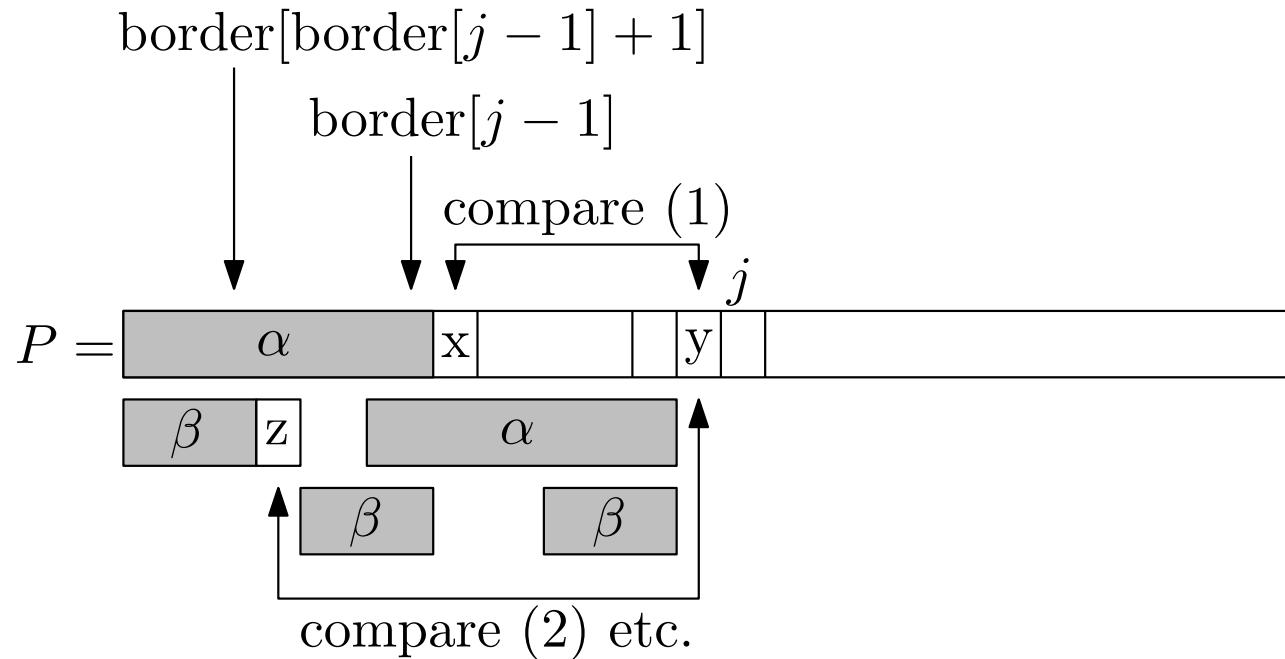
- $\text{border}[j] = \text{längstes echtes Präfix von } P_1 \dots j-1, \text{ das auch (echtes) Suffix von } P_1 \dots j-1 \text{ ist.}$   $\text{border}[1] := -1, \text{ border}[2] = 0.$

# Knuth-Morris-Pratt (1977)

```
i := 1                                // index in T
j := 1                                // index in P
while i ≤ n - m + 1
    while j ≤ m and ti+j-1 = pj do j++ // compare characters
    if j > m then
        print "P occurs at position i in T"
    i := i + j - border[j] - 1           // advance in T
    j := max{1, border[j] + 1} // skip first border[j] characters of P
```

# Berechnung des Border-Arrays

- seien die Werte bis zur Position  $j - 1$  bereits berechnet



# Berechnung des Border-Arrays

□ in  $O(m)$  Zeit:

`border[1] := -1`

`i := border[1]` // position in  $P$

**for**  $j = 2, \dots, m + 1$

**while**  $i \geq 0$  and  $p_{i+1} \neq p_{j-1}$  **do**  $i = \text{border}[i + 1]$

$i++$

`border[j] := i`

# Volltextsuche von Langsam bis Superschnell

**Gegeben:** Text  $S$  ( $n := |S|$ ), Muster (Pattern)  $P$  ( $m := |P|$ ),  $n \gg m$

**Gesucht:** Alle/erstes/nächstes Vorkommen von  $P$  in  $S$

naiv:  $O(nm)$

$P$  vorverarbeiten:  $O(n + m)$

Mit Fehlern: ???

$S$  vorverarbeiten: Textindizes. Erstes Vorkommen:

Invertierter Index: gute heuristik

Suffix Array:  $O(m \log n) \dots O(m)$

# Invertierter Index

- 1 The old night keeper keeps the keep in the town
- 2 In the big old house in the big old gown
- 3 The house in the town had the big old keep
- 4 Where the old night keeper never did sleep
- 5 The night keeper keeps the keep in the night
- 6 And keeps in the dark and sleeps in the light

| term $t$ | $f_t$ | Invertierte Liste für $t$         |
|----------|-------|-----------------------------------|
| and      | 1     | (6,2)                             |
| big      | 2     | (2,2), (3,1)                      |
| dark     | 1     | (6,1)                             |
| did      | 1     | (4,1)                             |
| gown     | 1     | (2,1)                             |
| had      | 1     | (3,1)                             |
| house    | 2     | (2,1), (3,1)                      |
| in       | 5     | (1,1), (2,2), (3,1), (5,1), (6,2) |
| keep     | 3     | (1,1), (3,1), (5,1)               |
| ...      | ...   | ...                               |

# Etwas “Stringology”-Notation

Alphabet  $\Sigma$ : Menge  $\{a, b, c, \dots\}$ .

String  $S$ : Array  $S[0..n) := S[0..n-1] := [S[0], \dots, S[n-1]]$   
von Buchstaben aus  $\Sigma$ .

Suffix:  $S_i := S[i..n)$

Endmarkierungen:  $S[n] := S[n+1] := \dots := 0$

0 ist kleiner als alle anderen Zeichen

$S = \text{banana} :$

|   |        |
|---|--------|
| 0 | banana |
| 1 | anana  |
| 2 | nana   |
| 3 | ana    |
| 4 | na     |
| 5 | a      |

# Suffixe Sortieren

Sortiere die Menge  $\{S_0, S_1, \dots, S_{n-1}\}$

von Suffixen des Strings  $S$  der Länge  $n$

(Alphabet  $[1, n] = \{1, \dots, n\}$ )

in lexikographische Reihenfolge.

- suffix  $S_i = S[i, n]$  für  $i \in [0..n-1]$

$S = \text{banana}:$

|   |        |   |        |
|---|--------|---|--------|
| 0 | banana | 5 | a      |
| 1 | anana  | 3 | ana    |
| 2 | nana   | 1 | anana  |
| 3 | ana    | 0 | banana |
| 4 | na     | 4 | na     |
| 5 | a      | 2 | nana   |

# Anwendungen

- Volltextsuche
- Burrows-Wheeler Transformation (*bzip2* Kompressor)
- Ersatz für kompliziertere Suffixbäume
- Bioinformatik: Wiederholungen suchen (in  $O(n)$  möglich?) ,...

## Volltextsuche

Suche Muster (pattern)  $P[0..m]$  im Text  $S[0..n]$   
mittels Suffix-Tabelle  $SA$  of  $S$ .

Binäre Suche:  $O(m \log n)$  gut für kurze Muster

Binäre Suche mit lcp:  $O(m + \log n)$  falls wir die  
längsten gemeinsamen (common) Präfixe  
zwischen verglichenen Zeichenketten vorberechnen

Suffix-Baum:  $O(n)$  kann aus  $SA$  berechnet werden

# Suffix-Baum

[Weiner '73][McCreight '76][Ukkonen '95]

kompakterter Trie der Suffixe

$S = \text{banana}0$

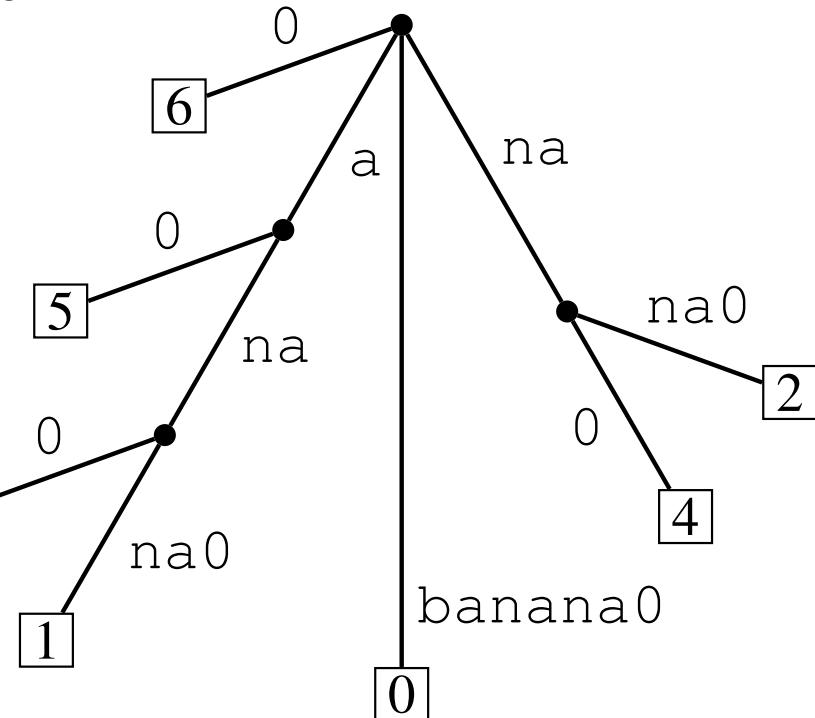
Zeit  $O(n)$  [Farach 97] für ganzzahlige  
Alphabete

Mächtigstes Werkzeug  
der Stringology?

Hoher Platzverbrauch

Effiziente direkte Konstruktion ist  
kompliziert

kann aus SA in Zeit  $O(n)$  abgelesen  
werden



# Suche in Suffix-Bäumen

- Suche (alle/ein) Vorkommen von  $P_{1..m}$  in  $T$ :
- Wenn ausgehende Kanten Arrays der Größe  $|\Sigma|$ :
  - $O(m)$  Suchzeit
  - $O(n|\Sigma|)$  Gesamtplatz
- Wenn ausgehende Kanten Arrays der Größe prop. zur #Kinder:
  - $O(m \log |\Sigma|)$  Suchzeit
  - $O(n)$  Platz

## Alphabet-Modell

Geordnetes Alphabet: Zeichen können nur **verglichen** werden

Konstante Alphabetgröße: endliche Menge  
deren Größe nicht von  $n$  abhängt.

Ganzzahliges Alphabet: Alphabet ist  $\{1, \dots, \sigma\}$   
für eine ganze Zahl  $\sigma \geq 2$

# Geordnetes → ganzzahliges Alphabet

Sortiere die Zeichen von  $S$

Ersetze  $S[i]$  durch seinen Rang

012345      135024

banana     $\rightarrow$  aaabnn

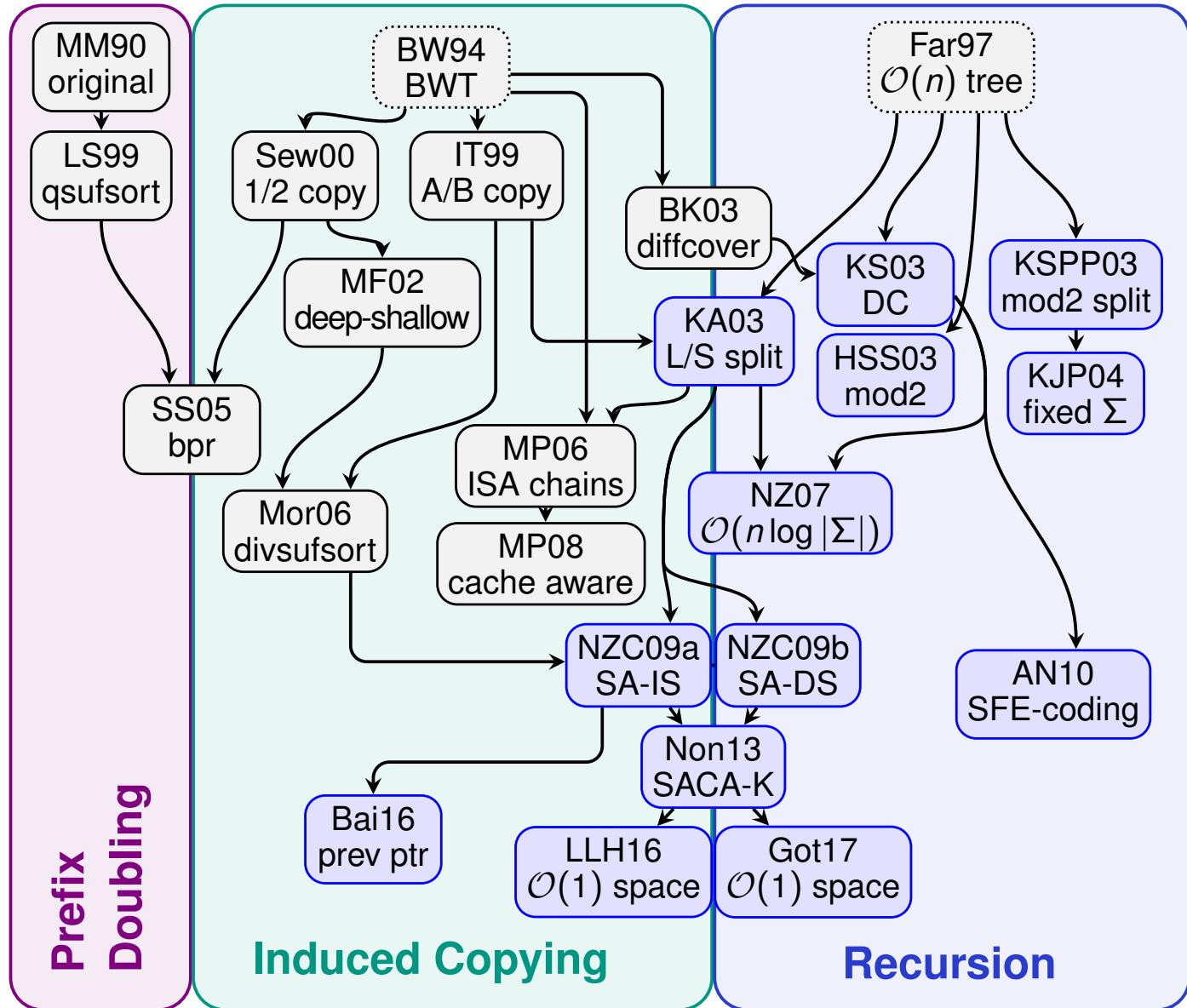
213131     $\leftarrow$  111233

## Verallgemeinerung: Lexikographische Namen

Sortiere die  $k$ -Tupel  $S[i..i+k)$  für  $i \in 1..n$

Ersetze  $S[i]$  durch den Rang von  $S[i..i+k)$  unter den Tupeln

# Suffix Array Konstruktionsalgorithmen



# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

| $i$ | $SA_0[i]$ | $T_i$                        | $ISA_0[i]$ | $L$ |
|-----|-----------|------------------------------|------------|-----|
| 0   | 0         | t o b e o r n o t t o b e \$ | 0          | -   |
| 1   | 1         | o b e o r n o t t o b e \$   | 1          | -   |
| 2   | 2         | b e o r n o t t o b e \$     | 2          | -   |
| 3   | 3         | e o r n o t t o b e \$       | 3          | -   |
| 4   | 4         | o r n o t t o b e \$         | 4          | -   |
| 5   | 5         | r n o t t o b e \$           | 5          | -   |
| 6   | 6         | n o t t o b e \$             | 6          | -   |
| 7   | 7         | o t t o b e \$               | 7          | -   |
| 8   | 8         | t t o b e \$                 | 8          | -   |
| 9   | 9         | t o b e \$                   | 9          | -   |
| 10  | 10        | o b e \$                     | 10         | -   |
| 11  | 11        | b e \$                       | 11         | -   |
| 12  | 12        | e \$                         | 12         | -   |
| 13  | 13        | \$                           | 13         | -   |

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

| $i$ | $SA_1[i]$ | $T_i$                           | $ISA_1[i]$ | $L$ |
|-----|-----------|---------------------------------|------------|-----|
| 0   | 13        | 0 \$                            | 11         | -1  |
| 1   | 2         | 1 b e o r n o t t o b e \$      | 6          | 2   |
| 2   | 11        | 1 b e \$                        | 1          | -   |
| 3   | 3         | 3 e o r n o t t o b e \$        | 3          | 2   |
| 4   | 12        | 3 e \$                          | 6          | -   |
| 5   | 6         | 5 n o t t o b e \$              | 10         | -1  |
| 6   | 1         | 6 o b e o r n o t t o b e \$    | 5          | 4   |
| 7   | 4         | 6 o r n o t t o b e \$          | 6          | -   |
| 8   | 7         | o t t o b e \$                  | 11         | -   |
| 9   | 10        | o b e \$                        | 11         | -   |
| 10  | 5         | 10 r n o t t o b e \$           | 6          | -1  |
| 11  | 0         | 11 t o b e o r n o t t o b e \$ | 1          | 3   |
| 12  | 8         | t t o b e \$                    | 3          | -   |
| 13  | 9         | t o b e \$                      | 0          | -   |

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

| $i$ | $SA_1[i]$ | $T_i$ | $ISA[SA[i] + 1]$               |  | $ISA_1[i]$ | $L$ |
|-----|-----------|-------|--------------------------------|--|------------|-----|
| 0   | 13        | 0     | \$                             |  | 11         | -1  |
| 1   | 2         | 1     | b e 3 o r n o t t o b e \$     |  | 6          | 2   |
| 2   | 11        |       | b e 3 \$                       |  | 1          | -   |
| 3   | 3         | 3     | e o 6 r n o t t o b e \$       |  | 3          | 2   |
| 4   | 12        |       | e \$ 0                         |  | 6          | -   |
| 5   | 6         | 5     | n o t t o b e \$               |  | 10         | -1  |
| 6   | 1         | 6     | o b 1 e o r n o t t o b e \$   |  | 5          | 4   |
| 7   | 4         |       | o r 10 1 o t t o b e \$        |  | 6          | -   |
| 8   | 7         |       | o t 11 ; o b e \$              |  | 11         | -   |
| 9   | 10        |       | o b 1 e \$                     |  | 11         | -   |
| 10  | 5         | 10    | r n o t t o b e \$             |  | 6          | -1  |
| 11  | 0         | 11    | t o 6 o e o r n o t t o b e \$ |  | 1          | 3   |
| 12  | 8         |       | t t 11 ; b e \$                |  | 3          | -   |
| 13  | 9         |       | t o 6 o e \$                   |  | 0          | -   |

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

| $i$ | $SA_2[i]$ | $T_i$ | $ISA[SA[i] + 1]$             | $ISA_1[i]$ | $L$ |
|-----|-----------|-------|------------------------------|------------|-----|
| 0   | 13        | 0     | \$                           | 11         | -1  |
| 1   | 2         | 1     | b e 3 r n o t t o b e \$     | 6          | 2   |
| 2   | 11        |       | b e 3 \$                     | 1          | -   |
| 3   | 12        | 3     | e \$ 0                       | 3          | -3  |
| 4   | 3         | 3     | e o 6 r n o t t o b e \$     | 6          | -   |
| 5   | 6         | 5     | n o t t o b e \$             | 10         | -   |
| 6   | 1         | 6     | o b 1 e o r n o t t o b e \$ | 5          | 2   |
| 7   | 10        |       | o b 1 e \$                   | 6          | -   |
| 8   | 4         | 10    | o r 10 i o t t o b e \$      | 11         | -3  |
| 9   | 7         | 11    | o t 11 ; o b e \$            | 11         | -   |
| 10  | 5         | 10    | r n o t t o b e \$           | 6          | -   |
| 11  | 0         | 11    | t o 6 e o r n o t t o b e \$ | 1          | 2   |
| 12  | 9         |       | t o 6 e \$                   | 3          | -   |
| 13  | 8         |       | t t 11 , b e \$              | 0          | -1  |

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

| $i$ | $SA_2[i]$ | $T_i$ | $ISA[SA[i] + 1]$             | $ISA_2[i]$ | $L$ |
|-----|-----------|-------|------------------------------|------------|-----|
| 0   | 13        | 0     | \$                           | 11         | -1  |
| 1   | 2         | 1     | b e 3 r n o t t o b e \$     | 6          | 2   |
| 2   | 11        | 1     | b e 3 \$                     | 1          | -   |
| 3   | 12        | 3     | e \$ 0                       | 4          | -3  |
| 4   | 3         | 4     | e o 6 r n o t t o b e \$     | 8          | -   |
| 5   | 6         | 5     | n o t t o b e \$             | 10         | -   |
| 6   | 1         | 6     | o b 1 e o r n o t t o b e \$ | 5          | 2   |
| 7   | 10        | 6     | o b 1 e \$                   | 9          | -   |
| 8   | 4         | 8     | o r 10 i o t t o b e \$      | 13         | -3  |
| 9   | 7         | 9     | o t 11 ; o b e \$            | 11         | -   |
| 10  | 5         | 10    | r n o t t o b e \$           | 6          | -   |
| 11  | 0         | 11    | t o 6 e o r n o t t o b e \$ | 1          | 2   |
| 12  | 9         | 11    | t o 6 e \$                   | 3          | -   |
| 13  | 8         | 13    | t t 11 , b e \$              | 0          | -1  |

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

| i  | SA <sub>2</sub> [i] | T <sub>i</sub> | ISA[SA[i] + 2]                 | ISA <sub>2</sub> [i] | L  |
|----|---------------------|----------------|--------------------------------|----------------------|----|
| 0  | 13                  | 0              | \$                             | 11                   | -1 |
| 1  | 2                   | 1              | b e o 8 r n o t t o b e \$     | 6                    | 2  |
| 2  | 11                  | 1              | b e \$ 0                       | 1                    | -  |
| 3  | 12                  | 3              | e \$                           | 4                    | -3 |
| 4  | 3                   | 4              | e o r n o t t o b e \$         | 8                    | -  |
| 5  | 6                   | 5              | n o t t o b e \$               | 10                   | -  |
| 6  | 1                   | 6              | o b e 4 o r n o t t o b e \$   | 5                    | 2  |
| 7  | 10                  | 6              | o b e 3 \$                     | 9                    | -  |
| 8  | 4                   | 8              | o r n o t t o b e \$           | 13                   | -3 |
| 9  | 7                   | 9              | o t t o b e \$                 | 11                   | -  |
| 10 | 5                   | 10             | r n o t t o b e \$             | 6                    | -  |
| 11 | 0                   | 11             | t o b 1 e o r n o t t o b e \$ | 1                    | 2  |
| 12 | 9                   | 11             | t o b 1 e \$                   | 3                    | -  |
| 13 | 8                   | 13             | t t o b e \$                   | 0                    | -1 |

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

| $i$ | $SA_4[i]$ | $T_i$ | $ISA[SA[i] + 2]$             | $ISA_2[i]$ | $L$ |
|-----|-----------|-------|------------------------------|------------|-----|
| 0   | 13        | 0     | \$                           | 11         | -11 |
| 1   | 11        | 1     | b e \$0                      | 6          | -   |
| 2   | 2         | 1     | b e o8r n o t t o b e \$     | 1          | -   |
| 3   | 12        | 3     | e \$                         | 4          | -   |
| 4   | 3         | 4     | e o r n o t t o b e \$       | 8          | -   |
| 5   | 6         | 5     | n o t t o b e \$             | 10         | -   |
| 6   | 10        | 6     | o b e3\$                     | 5          | -   |
| 7   | 1         | 6     | o b e4o r n o t t o b e \$   | 9          | -   |
| 8   | 4         | 8     | o r n o t t o b e \$         | 13         | -   |
| 9   | 7         | 9     | o t t o b e \$               | 11         | -   |
| 10  | 5         | 10    | r n o t t o b e \$           | 6          | -   |
| 11  | 0         | 11    | t o b1e o r n o t t o b e \$ | 1          | 2   |
| 12  | 9         | 11    | t o b1e \$                   | 3          | -   |
| 13  | 8         | 13    | t t o b e \$                 | 0          | -1  |

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

| i  | SA <sub>4</sub> [i] | T <sub>i</sub> | ISA[SA[i] + 2]     | ISA <sub>4</sub> [i] | L   |
|----|---------------------|----------------|--------------------|----------------------|-----|
| 0  | 13                  | 0              | \$                 | 11                   | -11 |
| 1  | 11                  | 1              | b e \$0            | 7                    | -   |
| 2  | 2                   | 2              | b e o8r            | 2                    | -   |
| 3  | 12                  | 3              | e \$               | 4                    | -   |
| 4  | 3                   | 4              | e o                | 8                    | -   |
| 5  | 6                   | 5              | r n o t t o b e \$ | 10                   | -   |
| 6  | 10                  | 6              | n o t t o b e \$   | 5                    | -   |
| 7  | 1                   | 7              | o b e3\$           | 9                    | -   |
| 8  | 4                   | 8              | o b e4o            | 13                   | -   |
| 9  | 7                   | 9              | r n o t t o b e \$ | 11                   | -   |
| 10 | 5                   | 10             | n o t t o b e \$   | 6                    | -   |
| 11 | 0                   | 11             | r n o t t o b e \$ | 1                    | 2   |
| 12 | 9                   |                | t o b1e            | 3                    | -   |
| 13 | 8                   |                | t o b1e \$         | 0                    | -1  |

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

| i  | SA <sub>4</sub> [i] | T <sub>i</sub> | ISA[SA[i] + 4]                 | ISA <sub>4</sub> [i] | L   |
|----|---------------------|----------------|--------------------------------|----------------------|-----|
| 0  | 13                  | 0              | \$                             | 11                   | -11 |
| 1  | 11                  | 1              | b e \$                         | 7                    | -   |
| 2  | 2                   | 2              | b e o r n o t t o b e \$       | 2                    | -   |
| 3  | 12                  | 3              | e \$                           | 4                    | -   |
| 4  | 3                   | 4              | e o r n o t t o b e \$         | 8                    | -   |
| 5  | 6                   | 5              | n o t t o b e \$               | 10                   | -   |
| 6  | 10                  | 6              | o b e \$                       | 5                    | -   |
| 7  | 1                   | 7              | o b e o r n o t t o b e \$     | 9                    | -   |
| 8  | 4                   | 8              | o r n o t t o b e \$           | 13                   | -   |
| 9  | 7                   | 9              | o t t o b e \$                 | 11                   | -   |
| 10 | 5                   | 10             | r n o t t o b e \$             | 6                    | -   |
| 11 | 0                   | 11             | t o b e o r 8 n o t t o b e \$ | 1                    | 2   |
| 12 | 9                   |                | t o b e \$ 0                   | 3                    | -   |
| 13 | 8                   | 13             | t t o b e \$                   | 0                    | -1  |

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

| $i$ | $SA_8[i]$ | $T_i$ | $ISA[SA[i] + 4]$               | $ISA_4[i]$ | $L$ |
|-----|-----------|-------|--------------------------------|------------|-----|
| 0   | 13        | 0     | \$                             | 11         | -14 |
| 1   | 11        | 1     | b e \$                         | 7          | -   |
| 2   | 2         | 2     | b e o r n o t t o b e \$       | 2          | -   |
| 3   | 12        | 3     | e \$                           | 4          | -   |
| 4   | 3         | 4     | e o r n o t t o b e \$         | 8          | -   |
| 5   | 6         | 5     | n o t t o b e \$               | 10         | -   |
| 6   | 10        | 6     | o b e \$                       | 5          | -   |
| 7   | 1         | 7     | o b e o r n o t t o b e \$     | 9          | -   |
| 8   | 4         | 8     | o r n o t t o b e \$           | 13         | -   |
| 9   | 7         | 9     | o t t o b e \$                 | 11         | -   |
| 10  | 5         | 10    | r n o t t o b e \$             | 6          | -   |
| 11  | 9         | 11    | t o b e \$ 0                   | 3          | -   |
| 12  | 0         |       | t o b e o r 8 n o t t o b e \$ | 1          | -   |
| 13  | 8         | 13    | t t o b e \$                   | 0          | -   |

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

| $i$ | $SA_8[i]$ | $T_i$ | $ISA[SA[i] + 4]$               | $ISA_8[i]$ | $L$ |
|-----|-----------|-------|--------------------------------|------------|-----|
| 0   | 13        | 0     | \$                             | 12         | -14 |
| 1   | 11        | 1     | b e \$                         | 7          | -   |
| 2   | 2         | 2     | b e o r n o t t o b e \$       | 2          | -   |
| 3   | 12        | 3     | e \$                           | 4          | -   |
| 4   | 3         | 4     | e o r n o t t o b e \$         | 8          | -   |
| 5   | 6         | 5     | n o t t o b e \$               | 10         | -   |
| 6   | 10        | 6     | o b e \$                       | 5          | -   |
| 7   | 1         | 7     | o b e o r n o t t o b e \$     | 9          | -   |
| 8   | 4         | 8     | o r n o t t o b e \$           | 13         | -   |
| 9   | 7         | 9     | o t t o b e \$                 | 11         | -   |
| 10  | 5         | 10    | r n o t t o b e \$             | 6          | -   |
| 11  | 9         | 11    | t o b e \$ 0                   | 3          | -   |
| 12  | 0         | 12    | t o b e o r 8 n o t t o b e \$ | 1          | -   |
| 13  | 8         | 13    | t t o b e \$                   | 0          | -   |

# SA mit Präfix Verdopplung [Larsson, Sadakane'99]

**Idee (aus [MM93]):** Sortiere Suffixe bis zu  $2h$  Zeichen tief mittels Informationen über die Ordnung der Suffixe bis zur Tiefe  $h$ .

**Def.:** Die  $h$ -Ordnung ist die lexikographische Ordnung bis zur Tiefe  $h$ .

**Original [MM93]:** Sortiere alle Suffix  $s_i$  mit dem Rang von  $s_i$  in einer  $h$ -Ordnung als ersten Sortier-Schlüssel und dem Rang von  $s_i + h$  als zweiten Sortier-Schlüssel. Dies ergibt eine  $2h$ -Ordnung.

**Definitionen [LS99]:** Wenn  $SA_h$  in einer  $h$ -Ordnung sind, dann

1. nennt man eine maximale Sequenz aufeinanderfolgender Suffixe in  $SA_h$  mit den gleichen  $h$  initialen Zeichen eine  $h$ -Gruppe.
2. Eine  $h$ -Gruppe der Länge eins nennt man sortiert oder Singleton, anderenfalls heißt eine Gruppe unsortiert. Eine maximale Sequenz von sortierten Gruppen heißt eine konsolidierte sortierte Gruppe.

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

Algorithmus [LS'99]: Präfix Verdopplung

Gegeben ist ein Text  $T$  der Länge  $n := |T|$ .

1. Enumeriere alle Suffixe in einem Array  $\text{SA}_0$  durch  $\text{SA}_0[i] = i$ .
2. Sortiere  $\text{SA}_0$  nach  $T[i]$  für Index  $i$  und erhalte  $\text{SA}_1$ . Setze  $h = 1$ .
3. Für  $i = 0, \dots, n$  setze  $\text{ISA}_1[i]$  auf den aktuellen Rang der 1-Gruppe von Suffix  $i$ . Dies ist der **kleinste Index** in  $\text{SA}_1[i]$  der ein Suffix mit dem gleichen ersten Buchstaben wie das Suffix  $i$ .
4. Für jede unsortierte Gruppe in  $\text{SA}_1[a..b]$  setzte  $L[a]$  auf dessen Länge. Für jede konsolidierte sortierte Gruppe in  $\text{SA}_1[a..b]$ , setze  $L[a]$  auf seine **negierte** Länge.

# SA mit Präfix Verdopplung

[Larsson, Sadakane'99]

5. Sortiere jede unsortierte Gruppe in  $\text{SA}_h$  (mit multikey quicksort), mittels  $\text{ISA}_h[\text{SA}_h[i] + h]$  als den Sortier-Schlüssel für Index  $i$ . Verwende  $L$  um die unsortierten Gruppen zu finden. Erhalte  $\text{SA}_{2h}$ .
6. Betrachte Folgen gleicher Schlüssel in den eben sortierten Gruppen: markiere Singletons mit  $-1$  in  $L$  und unsortierte  $2h$ -Gruppen mit positiven Längen. Konsolidiere die negativen Sprungwerte aufeinanderfolgende sortierte Gruppen.
7. Aktualisiere  $\text{ISA}_{2h}$  für alle bearbeiteten Gruppen, setze  $h := 2h$ .
8. Wenn  $\text{SA}_h$  noch unsortierte Gruppen enthält ( $L[0] \neq -n$ ), goto 5.

Laufzeit:  $\mathcal{O}(n \log n)$ . Platz:  $8n (+ 1n)$ . Worst-case Eingabe:  $[a^n]$ .

# Suffixtabellen

aus

## Linear Work Suffix Array Construction

Juha Kärkkäinen, Peter Sanders, Stefan Burkhardt

Journal of the ACM

Seiten 1–19, Nummer 6, Band 53.

# Ein erster Teile-und-Herrsche-Ansatz

1.  $SA^1 = \text{sort } \{S_i : i \text{ ist ungerade}\}$  (Rekursion)
2.  $SA^0 = \text{sort } \{S_i : i \text{ ist gerade}\}$  (einfach mittels  $SA^1$ )
3. Mische  $SA^0$  und  $SA^1$  (schwierig)

Problem: wie vergleicht man gerade und ungerade Suffixe?

[Farach 97] hat einen Linearzeitalgorithmus für  
Suffix-**Baum**-Konstruktion entwickelt, der auf dieser Idee beruht.  
Sehr **kompliziert**.

Das war auch der einzige bekannte Algorithmus für Suffix-**Tabellen**  
(lässt sich leicht aus S-Baum ablesen.)

# SA<sup>1</sup> berechnen

- Erstes Zeichen weglassen.

banana → anana

- Ersetze Buchstabenpaare durch Ihre lexikographischen Namen

|    |    |                |
|----|----|----------------|
| an | an | a <sub>0</sub> |
|----|----|----------------|

 → 221

- Rekursion

⟨1, 21, 221⟩

- Rückübersetzen

⟨a, ana, anana⟩

# Berechne $SA^0$ aus $SA^1$

|   |       |   |   |       |
|---|-------|---|---|-------|
| 1 | anana | ⇒ | 5 | a     |
| 3 | ana   |   | 3 | ana   |
| 5 | a     |   | 1 | anana |

Ersetze  $S_i$ ,  $i \bmod 2 = 0$  durch  $(S[i], r(S_{i+1}))$

mit  $r(S_{i+1}) :=$  Rang von  $S_{i+1}$  in  $SA^1$

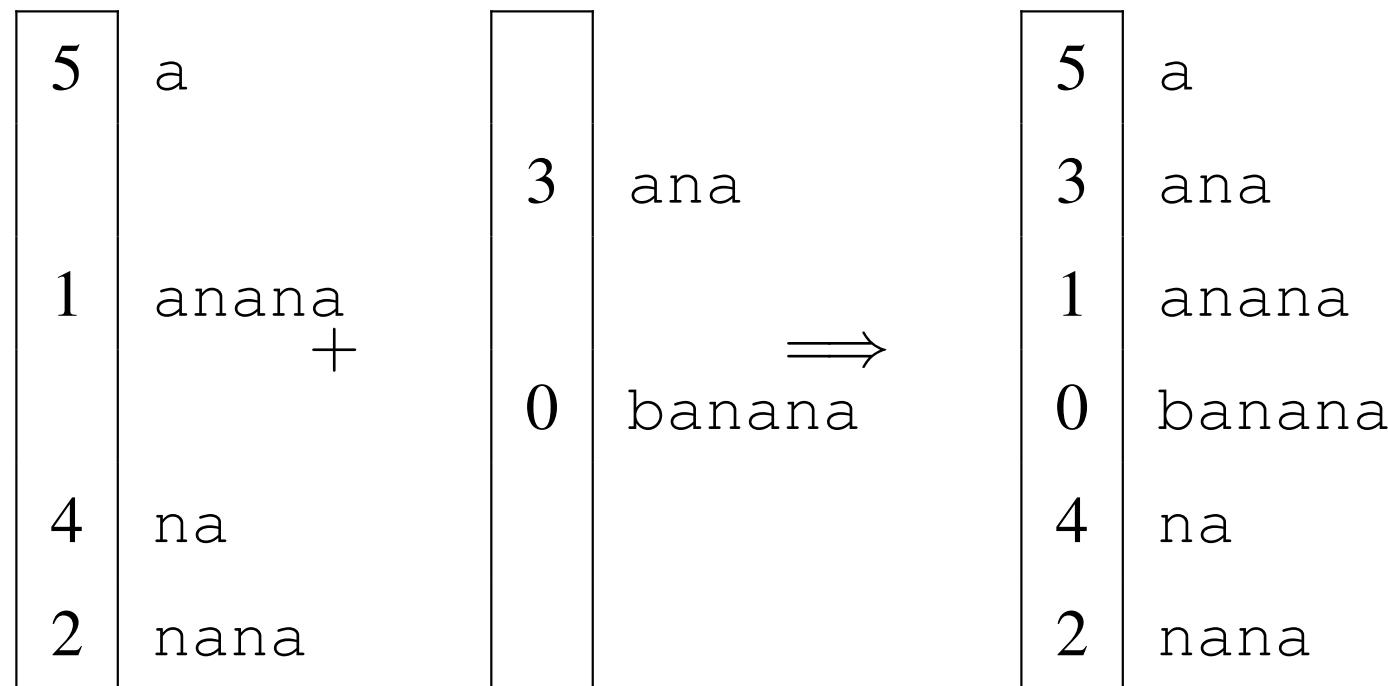
|   |   |   |         |   |   |        |
|---|---|---|---------|---|---|--------|
| 0 | b | 3 | (anana) | ⇒ | 0 | banana |
| 2 | n | 2 | (ana)   |   | 4 | na     |
| 4 | n | 1 | (a)     |   | 2 | nana   |

Radix-Sort

# Asymmetrisches Divide-and-Conquer

1.  $SA^{12} = \text{sort } \{S_i : i \bmod 3 \neq 0\}$  (Rekursion)
2.  $SA^0 = \text{sort } \{S_i : i \bmod 3 = 0\}$  (einfach mittels  $SA^{12}$ )
3. Mische  $SA^{12}$  und  $SA^0$  (**einfach!**)

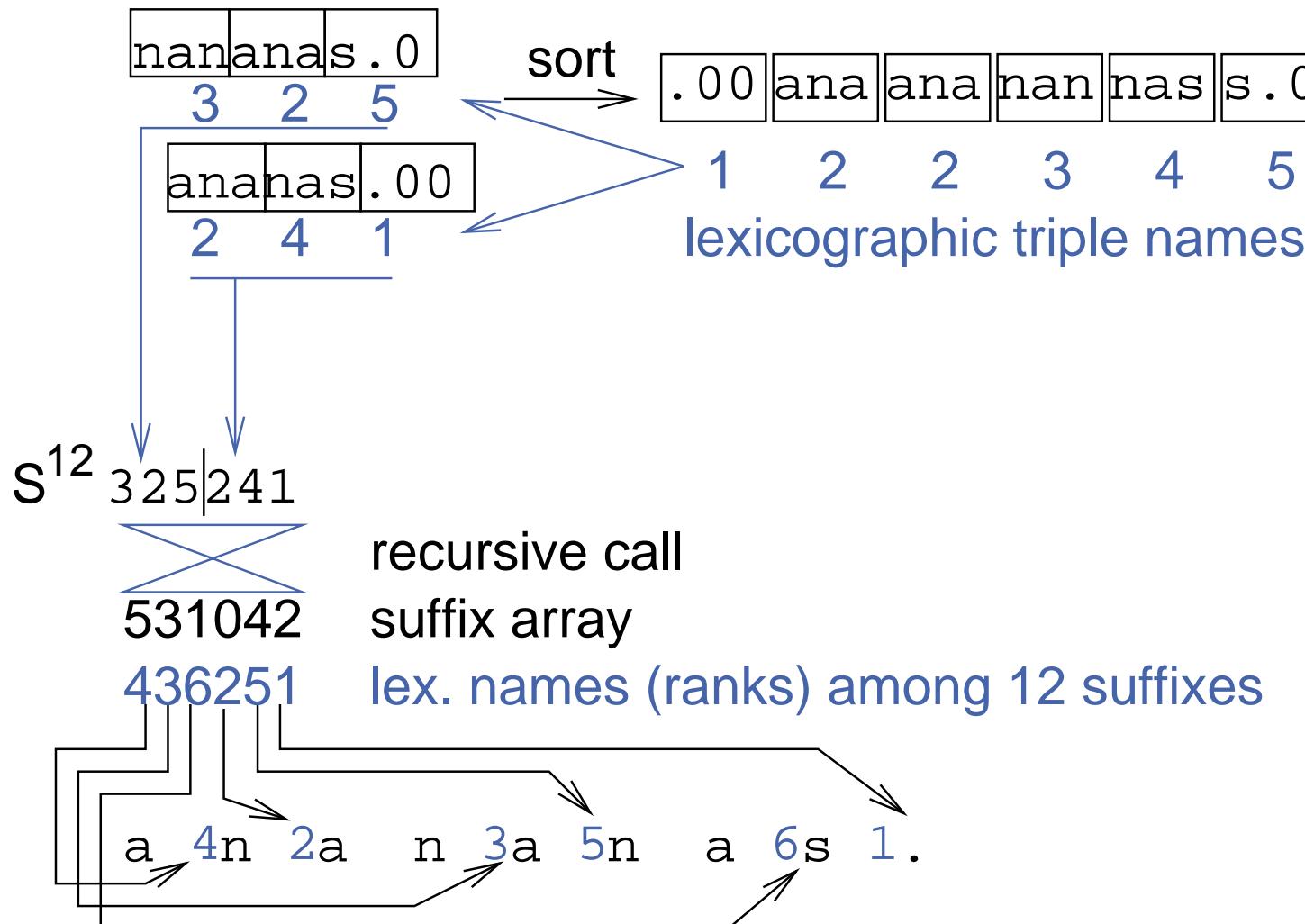
$S = \text{banana}$



# Rekursion, Beispiel

012345678

S anananas .



# Rekursion

- sortiere Tripel  $S[i..i+2]$  für  $i \bmod 3 \neq 0$   
(LSD Radix-Sortieren)
- Finde lexikographische Namen  $S'[1..2n/3]$  der Tripel,  
(d.h.,  $S'[i] < S'[j]$  gdw  $S[i..i+2] < S[j..j+2]$ )
- $S^{12} = [S'[i] : i \bmod 3 = 1] \circ [S'[i] : i \bmod 3 = 2]$ ,  
Suffix  $S_i^{12}$  von  $S^{12}$  repräsentiert  $S_{3i+1}$   
Suffix  $S_{n/3+i}^{12}$  von  $S^{12}$  repräsentiert  $S_{3i+2}$
- Rekursion auf  $(S^{12})$  (Alphabetgröße  $\leq 2n/3$ )
- Annotiere die 12-Suffixe mit ihrer Position in rek. Lösung

# Least Significant Digit First Radix Sort

Hier: Sortiere  $n$  3-Tupel von ganzen Zahlen  $\in [0..n]$  in  
**lexikographische** Reihenfolge

Sortiere nach 3. Position

Elemente sind nach Pos. 3 sortiert

Sortiere **stabil** nach 2. Position

Elemente sind nach Pos. 2,3 sortiert

Sortiere **stabil** nach 1. Position

Elemente sind nach Pos. 1,2,3 sortiert

# Stabiles Ganzzahliges Sortieren

Sortiere  $a[0..n)$  nach  $b[0..n)$  mit  $\text{key}(a[i]) \in [0..n]$

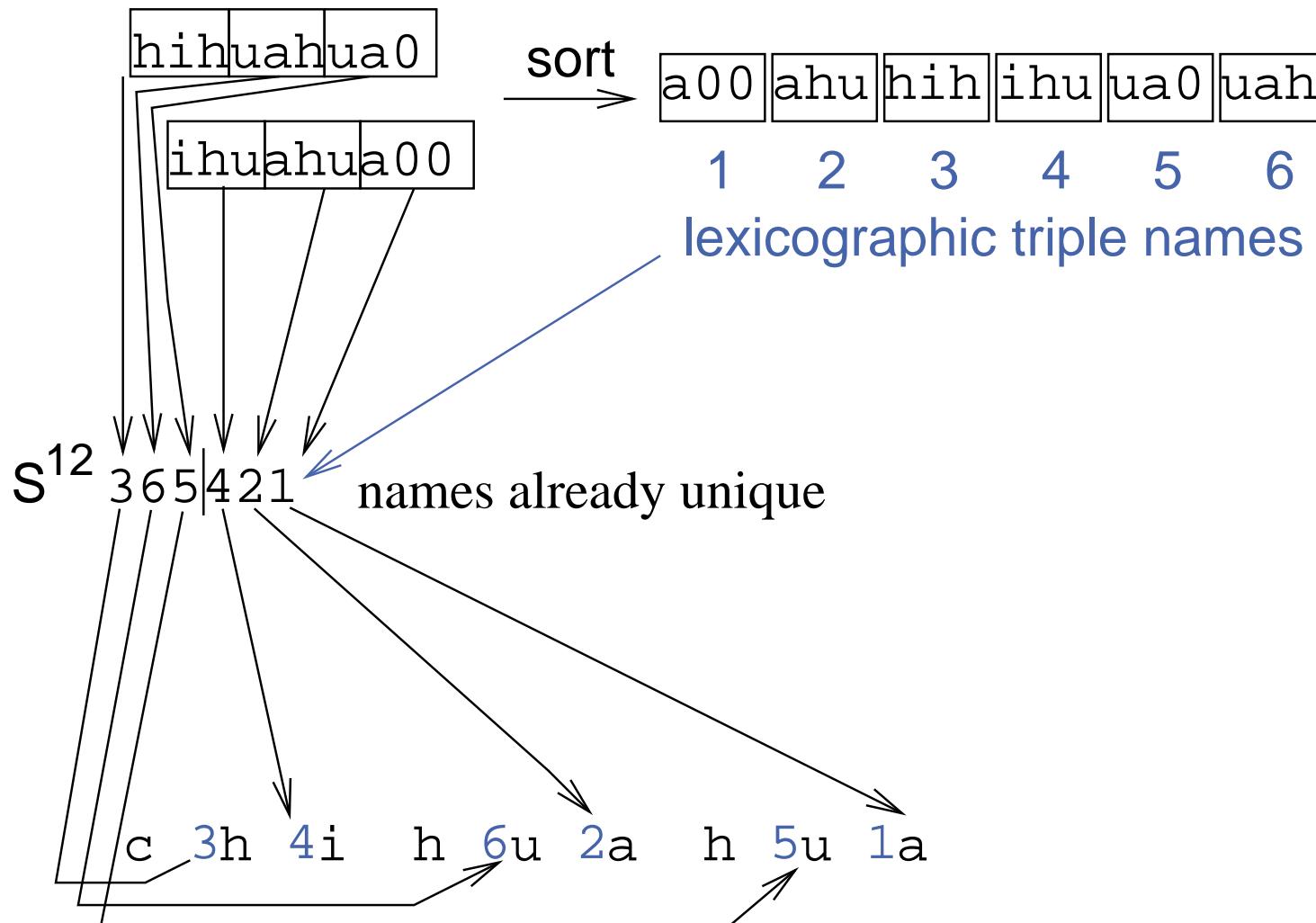
```
c[0..n] := [0, ..., 0]                                Zähler
for i ∈ [0..n) do c[a[i]]++                         zähle
s := 0
for i ∈ [0..n) do (s, c[i]) := (s + c[i], s)          Präfixsummen
for i ∈ [0..n) do b[c[a[i]]++] := a[i]                bucket sort
```

Zeit  $O(n)$  !

# Rekursions-Beispiel: Einfacher Fall

012345678

S chihuahua



# Sortieren der mod 0 Suffixe

|   |                                 |
|---|---------------------------------|
| 0 | c 3 (h 4 i h 6 u 2 a h 5 u 1 a) |
| 1 |                                 |
| 2 |                                 |
| 3 | h 6 (u 2 a h 5 u 1 a)           |
| 4 |                                 |
| 5 |                                 |
| 6 | h 5 (u 1 a)                     |
| 7 |                                 |
| 8 |                                 |

Benutze Radix-Sort (LSD-Reihenfolge bereits bekannt)

# Mische $SA^{12}$ und $SA^0$

$$0 < 1 \Leftrightarrow c_n < c_n$$

$$0 < 2 \Leftrightarrow cc_n < cc_n$$

3: h 6 u 2 (ahua)

6: h 5 u 1 (a)

0: c 3 h 4 (ihuahua)

|    |       |   |   |           |
|----|-------|---|---|-----------|
| 4: | ( 6 ) | u | 2 | (ahua)    |
| 7: | ( 5 ) | u | 1 | (a)       |
| 2: | ( 4 ) | i | h | 6 (uahua) |
| 1: | ( 3 ) | h | 4 | (ihuahua) |
| 5: | ( 2 ) | a | h | 5 (ua)    |
| 8: | ( 1 ) | a | 0 | 0 (0)     |

↓

8: a  
 5: ahua  
 0: chihuahua  
 1: hihuahua  
 6: hua  
 3: huahua  
 2: ihuahua  
 7: ua  
 4: uahua

# Analyse

1. Rekursion:  $T(2n/3)$  plus

Tripel extrahieren:  $O(n)$  (forall  $i, i \bmod 3 \neq 0$  do ...)

Tripel sortieren:  $O(n)$

(e.g., LSD-first radix sort — 3 Durchgänge)

Lexikographisches benenennen:  $O(n)$  (scan)

Rekursive Instanz konstruieren:  $O(n)$  (forall names do ...)

2.  $SA^0 =$ sortiere  $\{S_i : i \bmod 3 = 0\}$ :  $O(n)$

(1 Radix-Sort Durchgang)

3. mische  $SA^{12}$  and  $SA^0$ :  $O(n)$

(gewöhnliches Mischen mit merkwürdiger Vergleichsfunktion)

Insgesamt:  $T(n) \leq cn + T(2n/3)$

$\Rightarrow T(n) \leq 3cn = O(n)$

# Implementierung: Vergleichs-Operatoren

```
inline bool leq(int a1, int a2, int b1, int b2) {  
    return(a1 < b1 || a1 == b1 && a2 <= b2);  
}  
  
inline bool leq(int a1, int a2, int a3, int b1, int b2, int b3) {  
    return(a1 < b1 || a1 == b1 && leq(a2,a3, b2,b3));  
}
```

# Implementierung: Radix-Sortieren

```
// stably sort a[0..n-1] to b[0..n-1] with keys in 0..K from r
static void radixPass(int* a, int* b, int* r, int n, int K)
{ // count occurrences
    int* c = new int[K + 1];                                // counter array
    for (int i = 0; i <= K; i++) c[i] = 0;                  // reset counters
    for (int i = 0; i < n; i++) c[r[a[i]]]++;             // count occurrences
    for (int i = 0, sum = 0; i <= K; i++) { // exclusive prefix sums
        int t = c[i]; c[i] = sum; sum += t;
    }
    for (int i = 0; i < n; i++) b[c[r[a[i]]]++] = a[i]; // sort
    delete [] c;
}
```



# Implementierung: Tripel Sortieren

```
void suffixArray(int* s, int* SA, int n, int K) {  
    int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2;  
    int* s12 = new int[n02 + 3]; s12[n02]= s12[n02+1]= s12[n02+2]=0;  
    int* SA12 = new int[n02 + 3]; SA12[n02]=SA12[n02+1]=SA12[n02+2]=0;  
    int* s0 = new int[n0];  
    int* SA0 = new int[n0];  
  
    // generate positions of mod 1 and mod 2 suffixes  
    // the "+(n0-n1)" adds a dummy mod 1 suffix if n%3 == 1  
    for (int i=0, j=0; i < n+(n0-n1); i++) if (i%3 != 0) s12[j++] = i;  
  
    // lsb radix sort the mod 1 and mod 2 triples  
    radixPass(s12 , SA12, s+2, n02, K);  
    radixPass(SA12, s12 , s+1, n02, K);  
    radixPass(s12 , SA12, s , n02, K);
```

# Implementierung: Lexikographisches Benennen

```
// find lexicographic names of triples
int name = 0, c0 = -1, c1 = -1, c2 = -1;
for (int i = 0; i < n02; i++) {
    if (s[SA12[i]] != c0 || s[SA12[i]+1] != c1 || s[SA12[i]+2] != c2) {
        name++;
        c0 = s[SA12[i]];
        c1 = s[SA12[i]+1];
        c2 = s[SA12[i]+2];
    }
    if (SA12[i] % 3 == 1) { s12[SA12[i]/3] = name; } // left half
    else                  { s12[SA12[i]/3 + n0] = name; } // right half
```

# Implementierung: Rekursion

```
// recurse if names are not yet unique
if (name < n02) {
    suffixArray(s12, SA12, n02, name);
    // store unique names in s12 using the suffix array
    for (int i = 0; i < n02; i++) s12[SA12[i]] = i + 1;
} else // generate the suffix array of s12 directly
    for (int i = 0; i < n02; i++) SA12[s12[i] - 1] = i;
```

# Implementierung: Sortieren der mod 0 Suffixe

```
for (int i=0, j=0; i < n02; i++)  
    if (SA12[i] < n0) s0[j++] = 3*SA12[i];  
radixPass(s0, SA0, s, n0, K);
```



# Implementierung: Mischen

```

for (int p=0, t=n0-n1, k=0; k < n; k++) {
#define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)
    int i = GetI(); // pos of current offset 12 suffix
    int j = SA0[p]; // pos of current offset 0 suffix
    if (SA12[t] < n0 ?
        leq(s[i], s12[SA12[t] + n0], s[j], s12[j/3]) :
        leq(s[i], s[i+1], s12[SA12[t]-n0+1], s[j], s[j+1], s12[j/3+n0]))
    { // suffix from SA12 is smaller
        SA[k] = i; t++;
        if (t == n02) { // done --- only SA0 suffixes left
            for (k++; p < n0; p++, k++) SA[k] = SA0[p];
        }
    } else {
        SA[k] = j; p++;
        if (p == n0) { // done --- only SA12 suffixes left
            for (k++; t < n02; t++, k++) SA[k] = GetI();
        }
    }
}
delete [] s12; delete [] SA12; delete [] SA0; delete [] s0; }

```

## Verallgemeinerung: Differenzenüberdeckungen

Ein Differenzenüberdeckung  $D$  modulo  $v$  ist eine Teilmenge von  $[0, v)$ , so dass  $\forall i \in [0, v) : \exists j, k \in D : i \equiv k - j \pmod{v}$ .

Beispiel:

$\{1, 2\}$  ist eine Differenzenüberdeckung modulo 3.

$\{1, 2, 4\}$  ist eine Differenzenüberdeckung modulo 7.

- Führt zu platzeffizienterer Variante
- Schneller für kleine Alphabete

## Verbesserungen / Verallgemeinerungen

- tuning
- größere Differenzenüberdeckungen
- Kombiniere mit den besten Alg. für einfache Eingaben

[Manzini Ferragina 02, Schürmann Stoye 05, Yuta Mori 08]

# Suffixtabellenkonstruktion: Zusammenfassung

- einfache, direkte, Linearzeit für Suffixtabellenkonstruktion
- einfach anpassbar auf fortgeschrittene Berechnungsmodelle
- Verallgemeinerung auf Diff-Überdeckungen ergibt platzeffiziente Implementierung

# Suche in Suffix Arrays

Given:  $T$ , SA,  $P$ .

$l := 1; r := n + 1$

**while**  $l < r$  **do** //search left index

$q := \lfloor \frac{l+r}{2} \rfloor$

**if**  $P >_{\text{lex}} T_{\text{SA}[q] \dots \min\{\text{SA}[q]+m-1, n\}}$

**then**  $l := q + 1$  **else**  $r := q$

$s := l; l--; r := n$

**while**  $l < r$  **do** //search right index

$q := \lceil \frac{l+r}{2} \rceil$

**if**  $P =_{\text{lex}} T_{\text{SA}[q] \dots \min\{\text{SA}[q]+m-1, n\}}$

**then**  $l := q$  **else**  $r := q - 1$

**return**  $[s, l]$

- Zeit  $O(m \log n)$  (geht auch:  $O(m + \log n)$ )

# LCP-Array

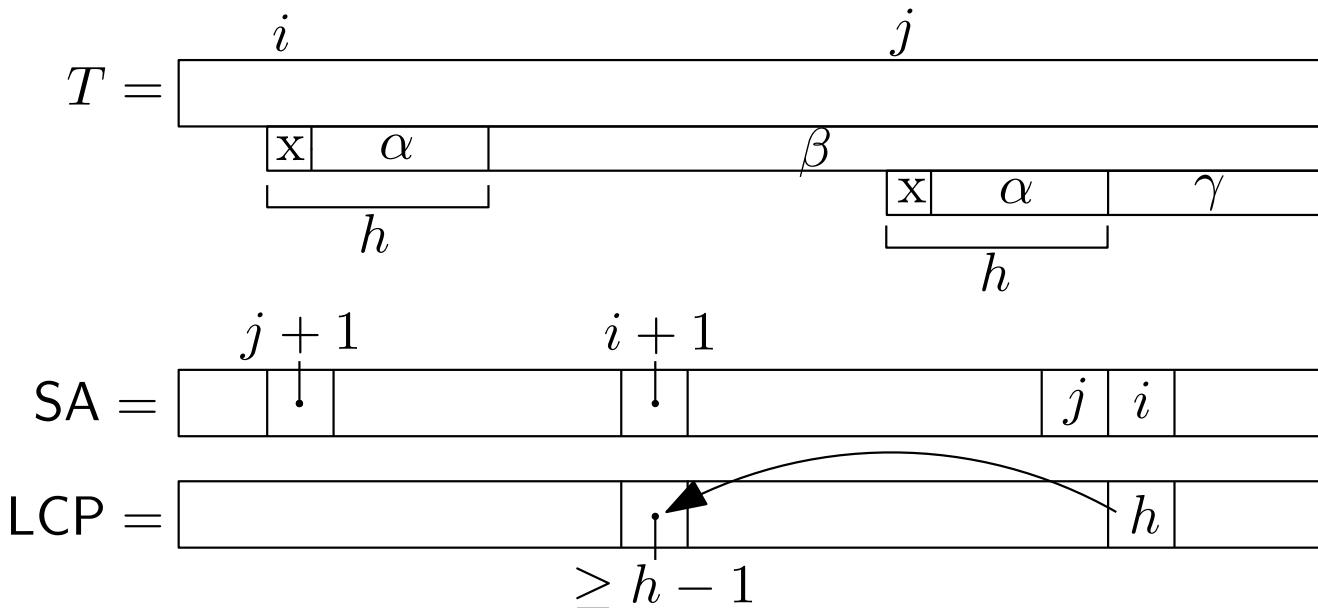
speichert Längen der längsten gemeinsamen Präfixe lexikographisch benachbarter Suffixe!

$S = \text{banana}$ :

|   |        |      |   |        |       |         |              |
|---|--------|------|---|--------|-------|---------|--------------|
| 0 | banana | SA = | 5 | a      | LCP = | $\perp$ | a            |
| 1 | anana  |      | 3 | ana    |       | 1       | <b>a</b> na  |
| 2 | nana   |      | 1 | anana  |       | 3       | <b>an</b> a  |
| 3 | ana    |      | 0 | banana |       | 0       | ba           |
| 4 | na     |      | 4 | na     |       | 0       | na           |
| 5 | a      |      | 2 | nana   |       | 2       | <b>na</b> na |

# LCP-Array: Berechnung

- naiv  $O(n^2)$
- inverses Suffix-Array:  $\text{SA}^{-1}[\text{SA}[i]] = i$  (wo steht  $i$  in SA?)
- For all  $1 \leq i < n$ :  $\text{LCP}[\text{SA}^{-1}[i+1]] \geq \text{LCP}[\text{SA}^{-1}[i]] - 1$ .



# LCP-Array: Berechnung

[KLAAP '01]

- For all  $1 \leq i < n$ :  $\text{LCP}[\text{SA}^{-1}[i+1]] \geq \text{LCP}[\text{SA}^{-1}[i]] - 1$ .

$h := 0, \text{LCP}[1] := 0$

**for**  $i = 1, \dots, n$  **do**  
**if**  $\text{SA}^{-1}[i] \neq 1$  **then**

**while**  $t_{i+h} = t_{\text{SA}[\text{SA}^{-1}[i]-1]+h}$  **do**  $h++$

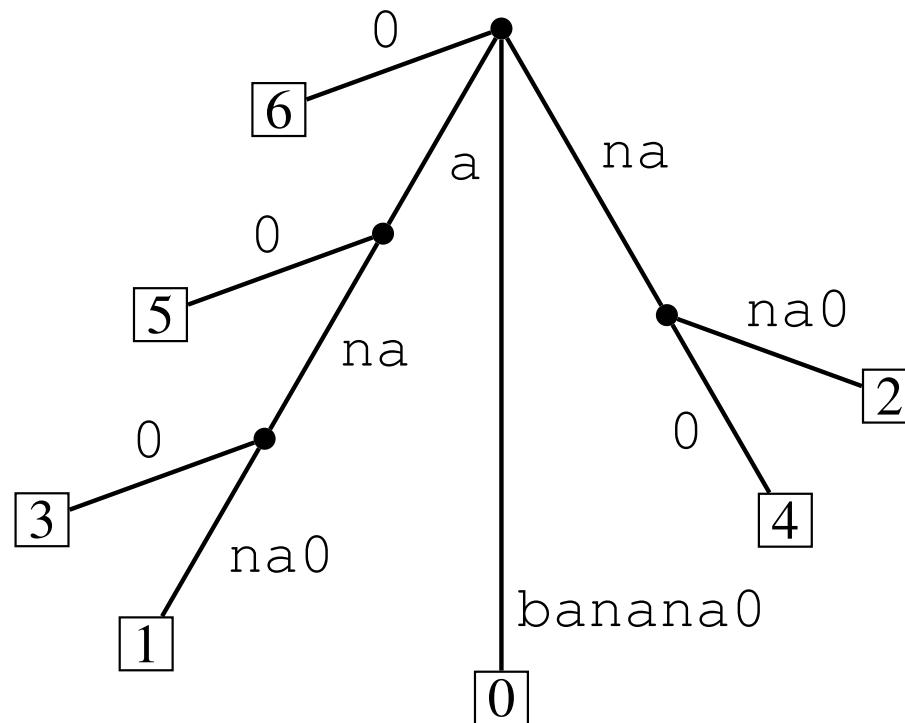
$\text{LCP}[\text{SA}^{-1}[i]] := h$

$h := \max(0, h - 1)$

- Zeit:  $O(n)$

# Suffix-Baum aus SA und LCP

$S = \text{banana}0$



- naiv:  $O(n^2)$
- mit Suffix-Array: in lexikographischer Reihenfolge

## Suffix-Baum aus SA und LCP

- LCP-Werte helfen!
- Betrachte nur **rechtensten Pfad!**
- Finde tiefsten Knoten mit String-Tiefe  $\leq \text{LCP}[i]$   $\rightsquigarrow$  **Einfügepunkt!**

|       | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|-------|---|---|---|---|---|---|---|
| SA =  | 6 | 5 | 3 | 1 | 0 | 4 | 2 |
| LCP = | 0 | 0 | 1 | 3 | 0 | 0 | 2 |

- Zeit  $O(n)$

# Datenkompression

Problem: bei naiver Speicherung verbrauchen Daten sehr viel Speicherplatz / Kommunikationsbandbreite. Das lässt sich oft reduzieren.

## Varianten:

- Verlustbehaftet (mp3, jpg, ...) / **Verlustfrei** (Text, Dateien, Suchmaschinen, Datenbanken, medizin. Bildverarbeitung, Profifotografie, ...)
- 1D** (text, Zahlenfolgen,...) / 2D (Bilder) / 3D (Filme)
- nur Speicherung** / mit Operationen ( $\rightsquigarrow$  succinct data structures)

# Verlustfreie Textkompression

**Gegeben:** Alphabet  $\Sigma$

String  $S = \langle s_1, \dots, s_n \rangle \in \Sigma^*$

Textkompressionsalgorithms  $f : S \rightarrow f(S)$  mit  $|f(S)|$  (z.B. gemessen in bits) möglichst klein.

# Theorie Verlustfreier Textkompression

Informationstheorie. Zum Beispiel

**Entropie:**  $H(S) = \sum_{c \in \Sigma} p(c) \log(1/p(c))$  wobei

$p(c) = |\{s_i : s_i = c\}|/n$  die relative Häufigkeit von  $c$  ist.

untere Schranke für **# bits** pro Zeichen falls Text einer Zufallsquelle entspränge.

~~> Huffman-Coding ist annähernd optimal! (Entropiecodierung) ????

Schon eher:

**Entropie höherer Ordnung** betrachte Teilstrings fester Länge

“Ultimativ”: Kolmogorov Komplexität. Leider nicht berechenbar.

# Theorie Verlustfreier Textkompression

**Entropie höherer Ordnung:** Gegeben ein Text  $S$  der Länge  $n$  über dem Alphabet  $\Sigma$ . Wir definieren  $N(\omega, S)$  als Konkatenation aller Zeichen, die in  $S$  auf Vorkommen von  $\omega \in \Sigma^k$  folgen. Wir definieren die **empirische Entropie der Ordung  $k$**  wie folgt:

$$H_k(S) = \sum_{\omega \in \Sigma^k} \frac{|N(\omega, S)|}{n} H(N(\omega, S))$$

Beispiel:  $S = \text{ananas}$ ,  $k = 2$

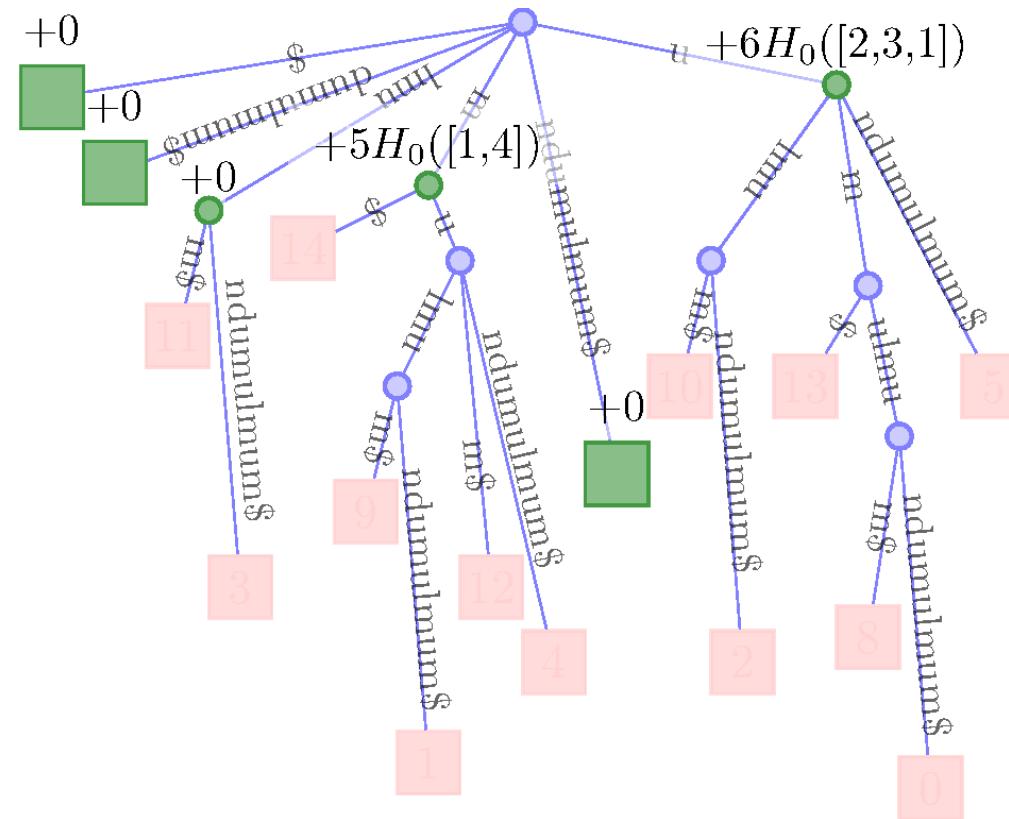
$$N(\text{an}, S) = \text{aa}, N(\text{na}, S) = \text{ns}, N(\text{as}, S) = \varepsilon.$$

$$H_2(\text{ananas}) = \frac{2}{6} H(\text{ns}) = \frac{1}{3} \text{ bits}$$



# Theorie Verlustfreier Textkompression

$H_k$ : Berechnung mittels Suffixbaumes (Beispiel:  $H_1$ )



# Theorie Verlustfreier Textkompression

Werte der empirischen Entropie in der Praxis

$H_k(S)$  in bits und (# eindeutiger Kontexte/ $|S|$  in Prozent)

| $k$ | dblp.xm |     | DNA  |     | english |     | proteins |      |
|-----|---------|-----|------|-----|---------|-----|----------|------|
| 0   | 5.26    | 0.0 | 1.97 | 0.0 | 4.53    | 0.0 | 4.20     | 0.0  |
| 1   | 3.48    | 0.0 | 1.93 | 0.0 | 3.62    | 0.0 | 4.18     | 0.0  |
| 2   | 2.17    | 0.0 | 1.92 | 0.0 | 2.95    | 0.0 | 4.16     | 0.0  |
| 3   | 1.43    | 0.1 | 1.92 | 0.0 | 2.42    | 0.0 | 4.07     | 0.0  |
| 4   | 1.05    | 0.4 | 1.91 | 0.0 | 2.06    | 0.3 | 3.83     | 0.1  |
| 5   | 0.82    | 1.3 | 1.90 | 0.0 | 1.84    | 1.0 | 3.16     | 1.7  |
| 6   | 0.70    | 2.7 | 1.88 | 0.0 | 1.67    | 2.7 | 1.50     | 17.4 |

# Wörterbuchbasierte Textkompression

Grundidee: wähle  $\Sigma' \subseteq \Sigma^*$  und ersetze  $S \in \Sigma^*$  durch  
 $S' = \langle s'_1, \dots, s'_k \rangle \in \Sigma'^*$ , so dass  $S = s'_1 \cdot s'_2 \cdots s'_k$ . (mit ‘ $\cdot$ ’= Zeichenkettenkonkatenation.)

Platz  $n \lceil \log \Sigma \rceil \rightarrow k \lceil \log \Sigma' \rceil$  mit Entropiecodierung der Zeichen aus  $\Sigma'$  sogar  $k \text{Entropie}(S')$

**Problem:** zusätzlicher Platz für Wörterbuch.

OK für sehr große Datenbestände.

## Wörterbuchbasierte Textkompression – Beispiel

Volltextsuchmaschinen verwenden oft  $\Sigma' :=$  durch Leerzeichen (etc.)

separierte Wörter der zugrundeliegenden natürlichen Sprache.

Spezialbehandlung von Trennzeichen etc.

Gallia est omnis divisa in partes tres, ...

→ gallia est omnis divisa in partes tres ...

# Lempel-Ziv Kompression (LZ)

Idee: baue Wörterbuch “on the fly” bei Codierung und Decodierung.  
Ohne explizite Speicherung!

# Naive Lempel-Ziv Kompression (LZ)

**Procedure** naiveLZCompress( $\langle s_1, \dots, s_n \rangle, \Sigma$ )

```
D :=  $\Sigma$                                 // Init Dictionary
p :=  $s_1$                                 // current string
for  $i := 2$  to  $n$  do
    if  $p \cdot s_i \in D$  then  $p := p \cdot s_i$ 
    else
        output code for  $p$ 
         $D := D \cup p \cdot s_i$ 
         $p := s_i$ 
    output code for  $p$ 
```

# Naive LZ Dekompression

**Procedure** naiveLZDecode( $\langle c_1, \dots, c_k \rangle$ )

$D := \Sigma$

output decode( $c_1$ )

**for**  $i := 2$  **to**  $k$  **do**

**if**  $c_i \in D$  **then**

$D := D \cup \text{decode}(c_{i-1}) \cdot \text{decode}(c_i)[1]$

**else** // where  $s[1]$  is the first letter.

$D := D \cup \text{decode}(c_{i-1}) \cdot \text{decode}(c_{i-1})[1]$

    output decode( $c_i$ )

# LZ Beispiel: abracadabra

| #  | $p$     | output | input | $D \cup =$ |
|----|---------|--------|-------|------------|
| 1  | $\perp$ | -      | a     | a,b,c,d,r  |
| 2  | a       | a      | b     | ab         |
| 3  | b       | b      | r     | br         |
| 4  | r       | r      | a     | ra         |
| 5  | a       | a      | c     | ac         |
| 6  | c       | c      | a     | ca         |
| 7  | a       | a      | d     | ad         |
| 8  | d       | d      | a     | da         |
| 9  | a       | -      | b     | -          |
| 10 | ab      | ab     | r     | abr        |
| 11 | r       | -      | a     | -          |
| -  | ra      | ra     | -     | -          |

## LZ-Verfeinerungen

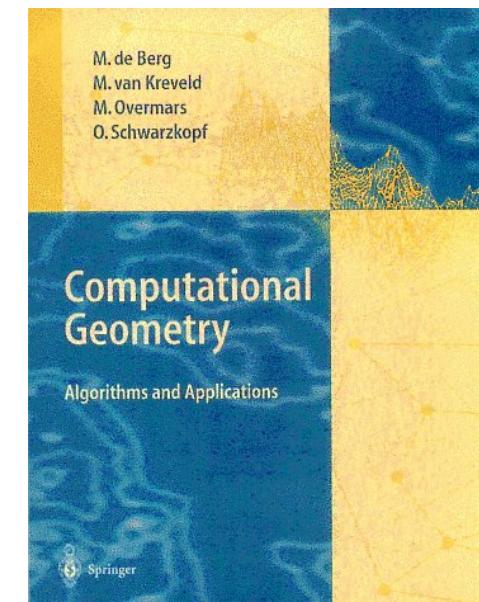
- Wörterbuchgröße begrenzen, z.B.  $|D| \leq 4096 \rightsquigarrow 12\text{bit codes.}$
- Von vorn wenn Wörterbuch voll  $\rightsquigarrow$  Blockweise arbeiten
- Kodierung mit **variabler Zahl Bits** (z.B. Huffman, arithmetic coding)
- Selten benutzte Wörterbucheinträge löschen ???
- Wörterbuch effizient implementieren:  
(universelles) hashing
- verwendet in zip/gzip mit Huffman Kodierung

# 12 Geometrische Algorithmen

- Womit beschäftigen sich geom. Algorithmen?
- Schnitt von Strecken: Bentley-Ottmann-Algorithmus
- Konvexe Hüllen
- Kleinste einschließende Kugel
- Range Search

Quelle:

[Computational Geometry – Algorithms and Applications  
de Berg, van Kreveld, Overmars, Schwarzkopf  
Springer, 1997]

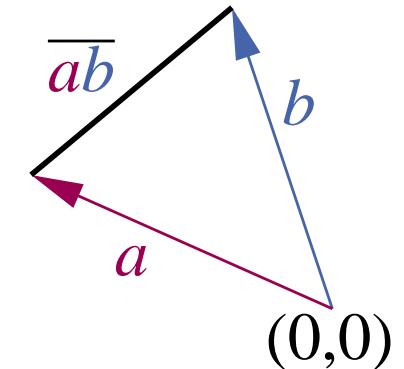


# Elementare geometrische Objekte

Punkte:  $x \in \mathbb{R}^d$

Strecken:  $\overline{ab} := \{\alpha a + (1 - \alpha)b : \alpha \in [0, 1]\}$

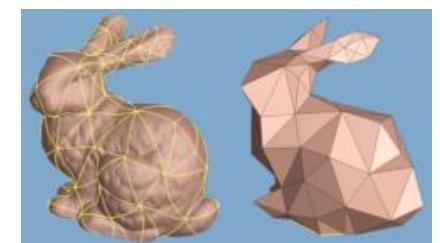
uvam: Halbräume, Ebenen, Kurven,...



## Dimension $d$ :

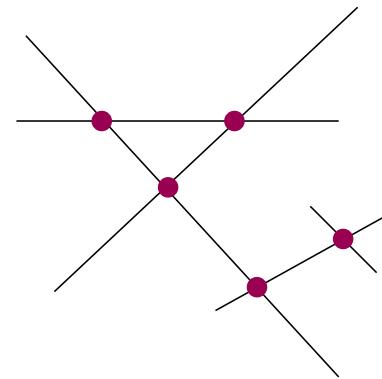
- 1: Oft trivial. Gilt i. allg. nicht als geometrisches Problem
  - 2: Geogr. Informationssysteme (GIS), Bildverarbeitung,...
  - 3: Computergrafik, Simulationen,...
  - $\geq 4$ : Optimierung, Datenbanken, maschinelles Lernen,...
- curse of dimensionality!

$n$ : Anzahl vorliegender Objekte



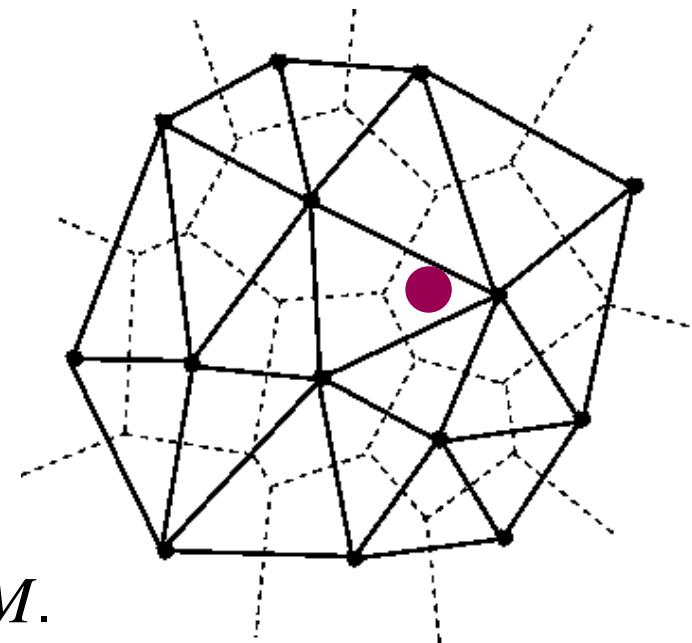
# Typische Fragestellungen

- Schnittpunkte zwischen  $n$  Strecken



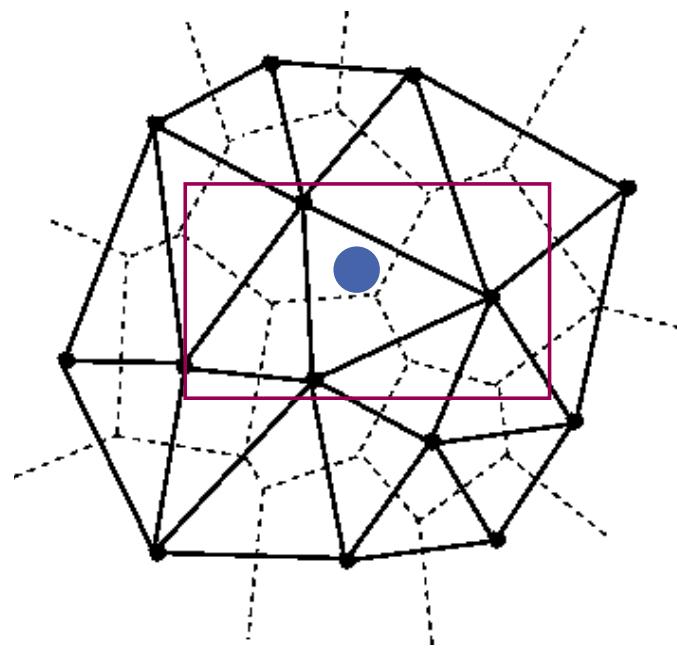
# Typische Fragestellungen

- Schnittpunkte zwischen  $n$  Strecken
- Konvexe Hülle
- Triangulation von Punktmenzen  
(2D, verallgemeinerbar)  
z.B. **Delaunaytriangulierung**:  
Kein Dreieck enthält weiteren Punkt
- Voronoi-Diagramme**: Sei  $x \in M \subseteq \mathbb{R}^d$ .  
 $\forall y \in \mathbb{R}^d$  bestimme nächstes Element aus  $M$ .  
(Unterteilung von  $M$  in  $n$  Voronoizellen)
- Punklokalisierung**: Geg. Unterteilung von  $R^d$ ,  $x \in R^d$ :  
in welchem Teil liegt  $x$  ?



# Datenstrukturen für Punktmengen

- nächsten Nachbarn berechnen
- Bereichsanfragen
- ...



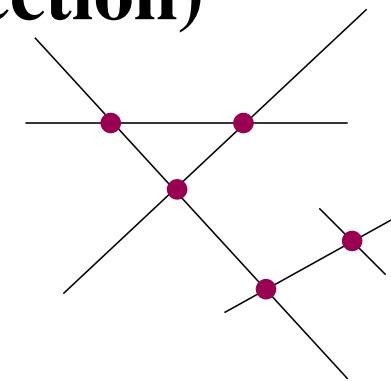
# Mehr Fragestellungen

- Sichtbarkeitsberechnungen
- Lineare Programmierung
- Geometrische Versionen von Optimierungsproblemen
  - Kürzeste Wege, z.B.  
*energieeffiziente Kommunikation in Radionetzwerken*
  - minimale Spannbäume  
*reduzierbar auf Delaunay-Triangulierung + Graphalgorithmus*
  - Matchings
  - Handlungsreisendenproblem
  - ...
- ...

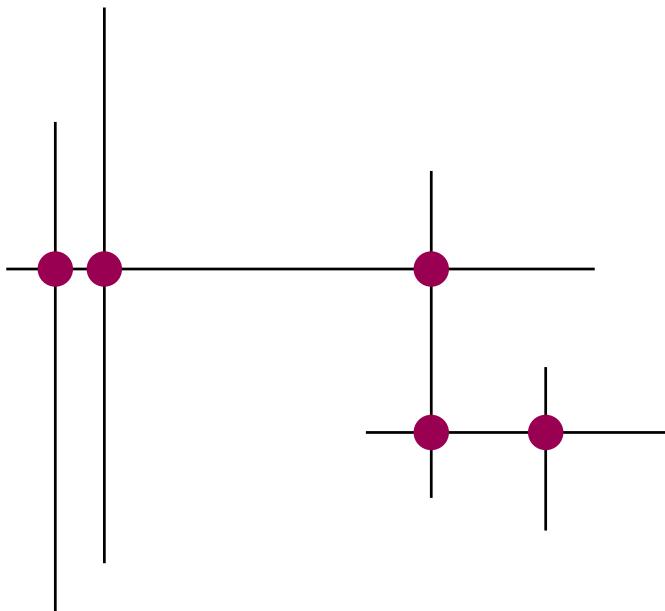
## 12.1 Streckenschnitt (line segment intersection)

**Gegeben:**  $S = \{s_1, \dots, s_n\}$ ,  $n$  Strecken

**Gesucht:** Schnittpunkte  $\bigcup_{s,t \in S} s \cap t$

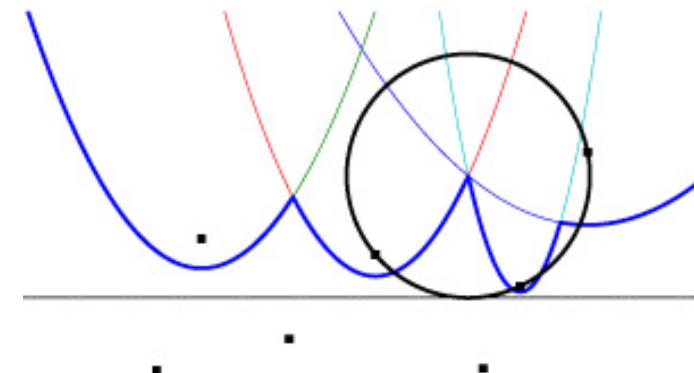


Zum Warmwerden: Orthogonaler Streckenschnitt – die Strecken sind parallel zur x- oder y-Achse



# Streckenschnitt: Anwendungen

- Schaltungsentwurf: wo kreuzen sich Leiterbahnen?
- **GIS**: Strassenkreuzungen, Brücken,...
- Erweiterungen: z.B. Graphen benachbarter Strecken/Flächen aufbauen/verarbeiten
- Noch allgemeiner:  
**Plane-Sweep**-Algorithmen für andere Fragestellungen  
(z.B. Konstruktion von konvexen Hüllen oder Voronoidiagrammen)



# Streckenschnitt: Naiver Algorithmus

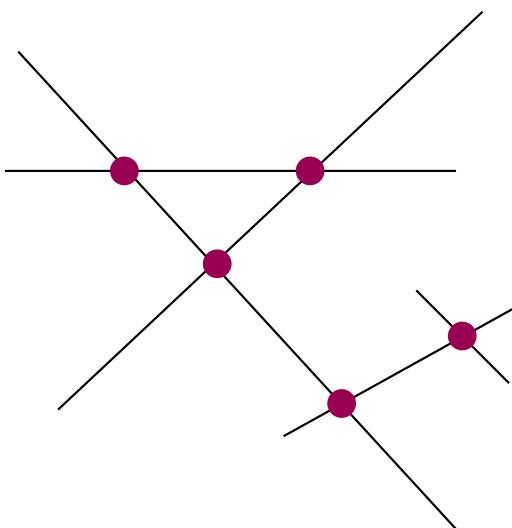
**foreach**  $\{s, t\} \subseteq S$  **do**

**if**  $s \cap t \neq \emptyset$  **then**

        output  $\{s, t\}$

Problem: Laufzeit  $\Theta(n^2)$ .

Zu langsam für große Datenmengen

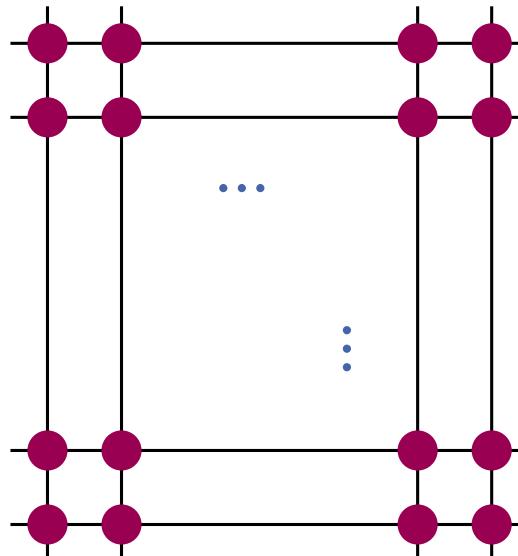


# Streckenschnitt: Untere Schranke

$\Omega(n + k)$  mit  $k :=$  Anzahl ausgegebener Schnitte.

Vergleichsbasiert:  $\Omega(n \log n + k)$  (Beweis: nicht hier)

Beobachtung  $k = \Theta(n^2)$  ist möglich, aber reale Eingaben haben meist  $k = O(n)$ .

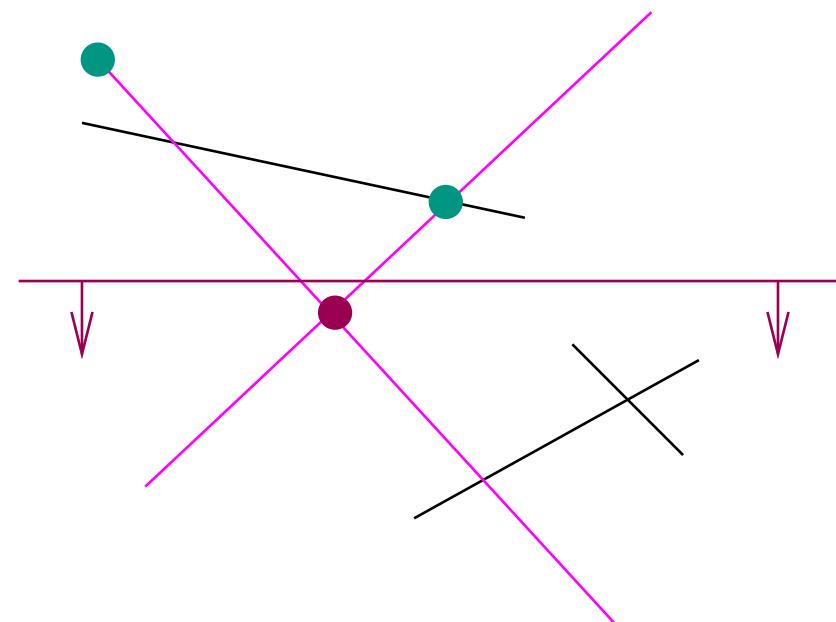


# Idee: Plane-Sweep-Algorithmen

(Waagerechte) Sweep-Line  $\ell$  läuft von oben nach unten.

Invariante: Schnittpunkte oberhalb von  $\ell$  wurden korrekt ausgegeben.

**Ansatz:** speichere jeweils Segmente, die  $\ell$  schneiden und finde deren Schnittpunkte.



# Plane-Sweep für orth. Streckenschnitt

Erstmal nur: Schnitte zwischen vertikalen und horizontalen Strecken.

$T = \langle \rangle$  : SortedSequence **of** Segment

**invariant**  $T$  stores the vertical segments intersecting  $\ell$

$Q := \text{sort}(\langle (y, s) : \exists \text{hor. seg. } s \text{ at } y \text{ or } \exists \text{vert. seg. } s \text{ starting/ending at } y \rangle)$

//tie breaking: vert. starting events first, vert. finishing events last

**foreach**  $(y, s) \in Q$  in descending order **do**

**if**  $s$  is a vertical segment and **starts** at  $y$  **then**  $T.\text{insert}(s)$

**else if**  $s$  is a vertical segment and **ends** at  $y$  **then**  $T.\text{remove}(s)$

**else** //we have a horizontal segment  $s = \overline{(x_1, y)(x_2, y)}$

**foreach**  $t = \overline{(x, y_1)(x, y_2)} \in T$  with  $x \in [x_1, x_2]$  **do**

output  $\{s, t\}$

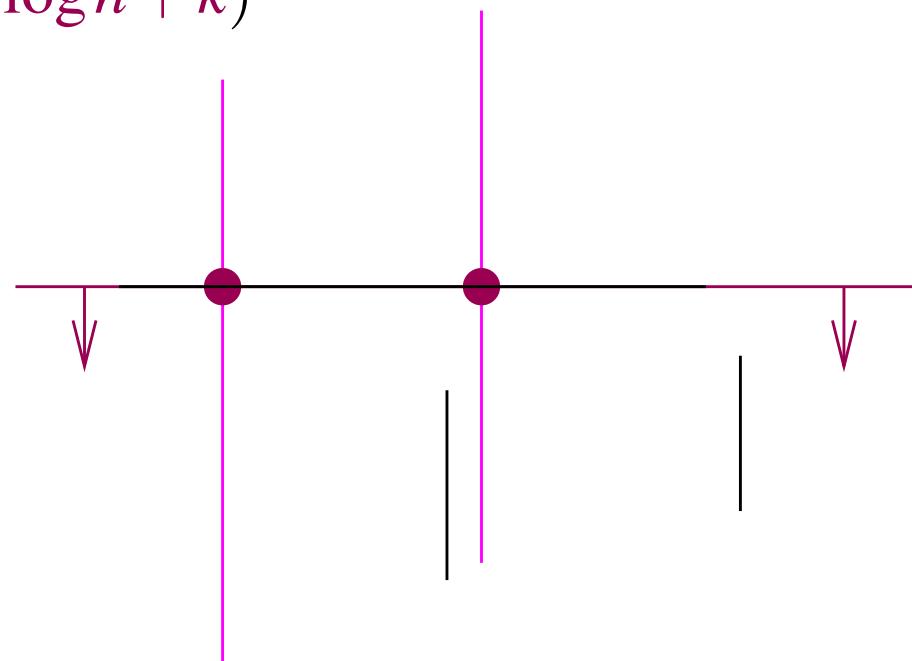
# Analyse orth. Streckenschnitt

insert:  $O(\log n)$  ( $\leq n \times$ )

remove:  $O(\log n)$  ( $\leq n \times$ )

rangeQuery:  $O(\log n + k_s)$ ,  $k_s$  Schnitte mit hor. Segment  $s$

Insgesamt:  $O(n \log n + \sum_s k_s) = O(n \log n + k)$



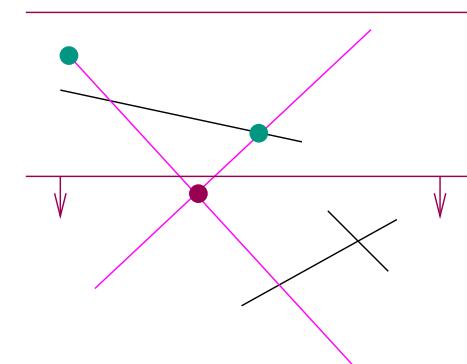
# Verallgemeinerung – aber erstmal “nicht ganz”

Annahme zunächst:

Allgemeine Lage, d.h. hier

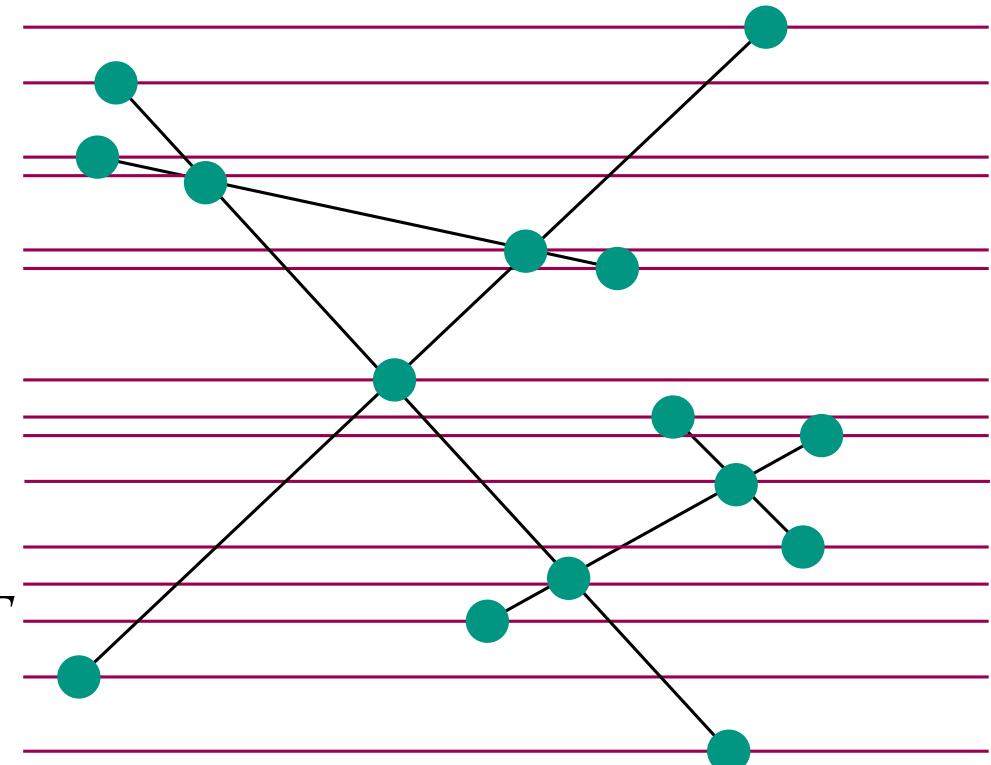
- Keine horizontalen Strecken
- Keine Überlappungen
- Schnittpunkte jeweils im Inneren von genau zwei Strecken

**Beobachtung:** kleine zuf. Perturbationen produzieren allg. Lage.



# Verallgemeinerung – Grundidee

- Plane-Sweep mit Sweep-Line  $\ell$
- Status  $T$  := nach  $x$  geordnete Folge der  $\ell$  schneidenden Strecken
- Ereignis:= Statusänderung
  - Startpunkte
  - Endpunkte
  - Schnittpunkte
- Schnitttest nur für Segmente, die an einem Ereignispunkt in  $T$  benachbart sind.



# Verallgemeinerung – Korrektheit

**Lemma:**

$s \cap t = \{(x, y)\} \longrightarrow \exists \text{ Ereignis : } s, t \text{ werden Nachbarn auf } \ell$

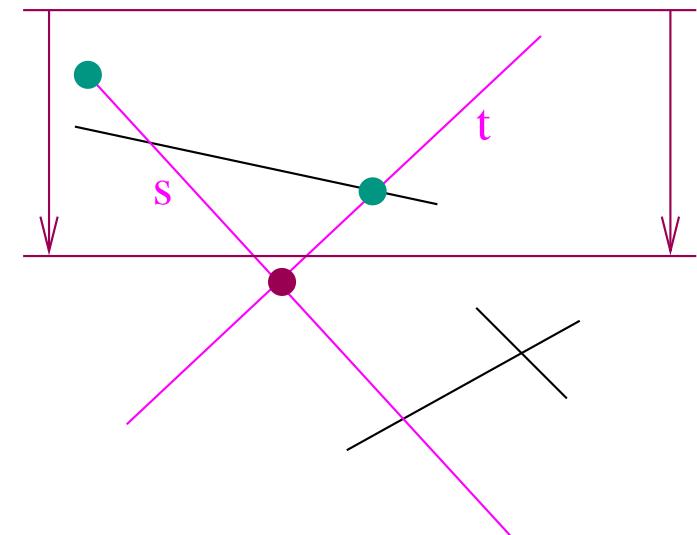
**Beweis:**

Anfangs :  $T = \langle \rangle \longrightarrow s, t \text{ sind nicht in } \ell \text{ benachbart.}$

@ $y + \varepsilon$  :  $s, t$  sind in  $\ell$  benachbart.

$\longrightarrow$

$\exists \text{ Ereignis bei dem } s \text{ und } t \text{ Nachbarn werden.}$



# Verallgemeinerung – Implementierung

$T = \langle \rangle$  : SortedSequence of Segment

**invariant**  $T$  stores the relative order of the segments intersecting  $\ell$

$Q$  : MaxPriorityQueue

$Q := Q \cup \left\{ (\max \{y, y'\}, \text{start}, s) : s = \overline{(x, y)(x', y')} \in S \right\} // O(n \log n)$

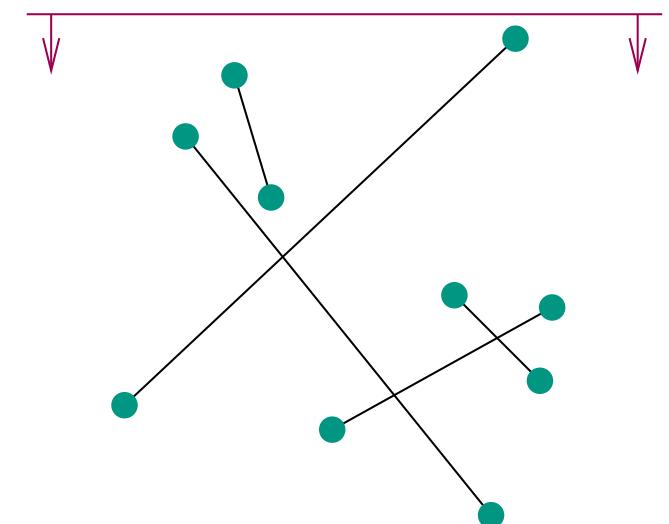
$Q := Q \cup \left\{ (\min \{y, y'\}, \text{finish}, s) : s = \overline{(x, y)(x', y')} \in S \right\} // O(n \log n)$

**while**  $Q \neq \emptyset$  **do**

$(y, \text{type}, s) := Q.\text{deleteMax}$

$// O((n+k) \log n)$

    handleEvent( $y, \text{type}, s, T, Q$ )



```

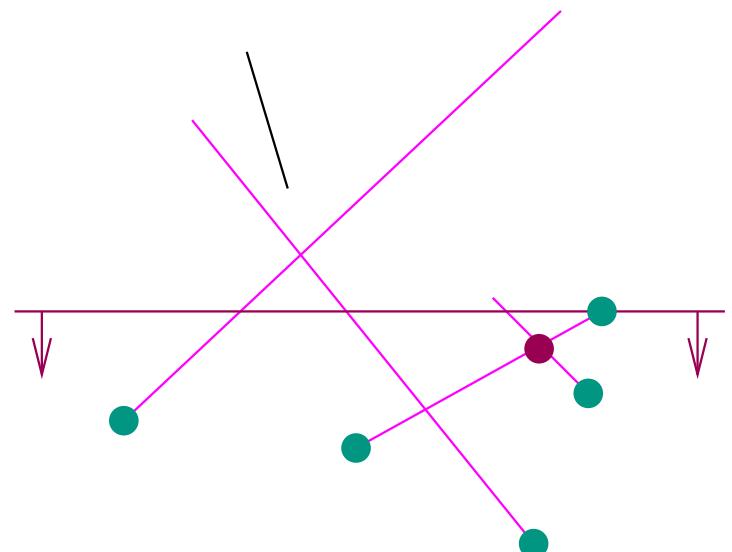
handleEvent( $y, \text{start}, s, T, Q$ ) //  $n \times$ 
     $h := T.\text{insert}(s)$  //  $O(\log n)$ 
    prev := pred( $h$ ) //  $O(1)$ 
    next := succ( $h$ ) //  $O(1)$ 
    findNewEvent(prev,  $h$ )
    findNewEvent( $h$ , next)

```

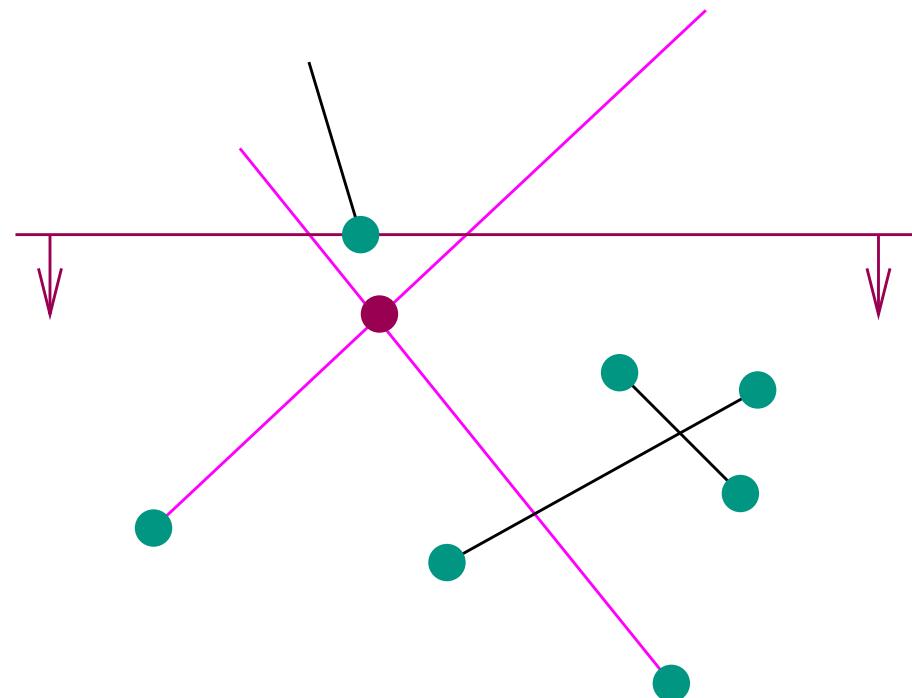
**Procedure** findNewEvent( $s, t$ ) //  $O(1 + \log n)$

**if**  $*s$  and  $*t$  intersect at  $y' < y$  **then**

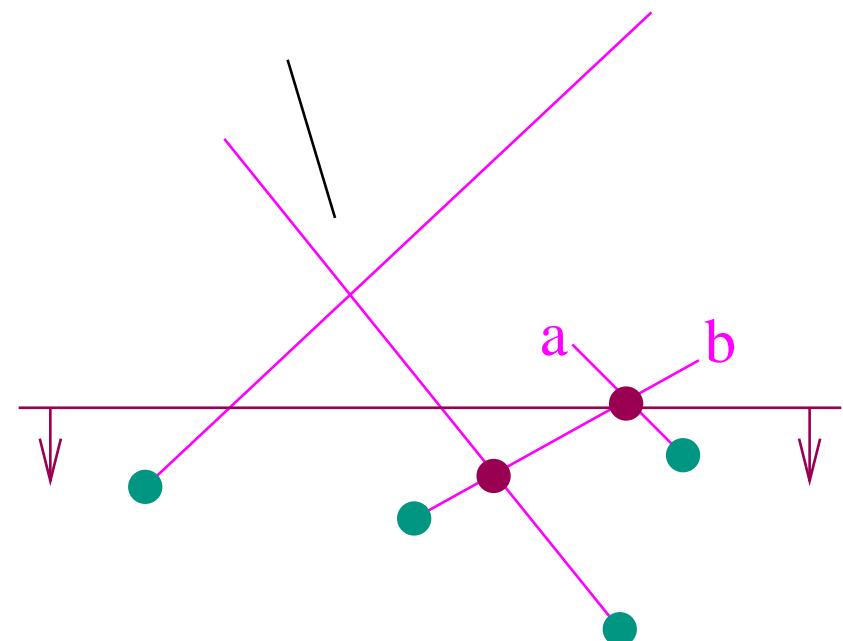
$Q.\text{insert}((y', \text{intersection}, (s, t)))$



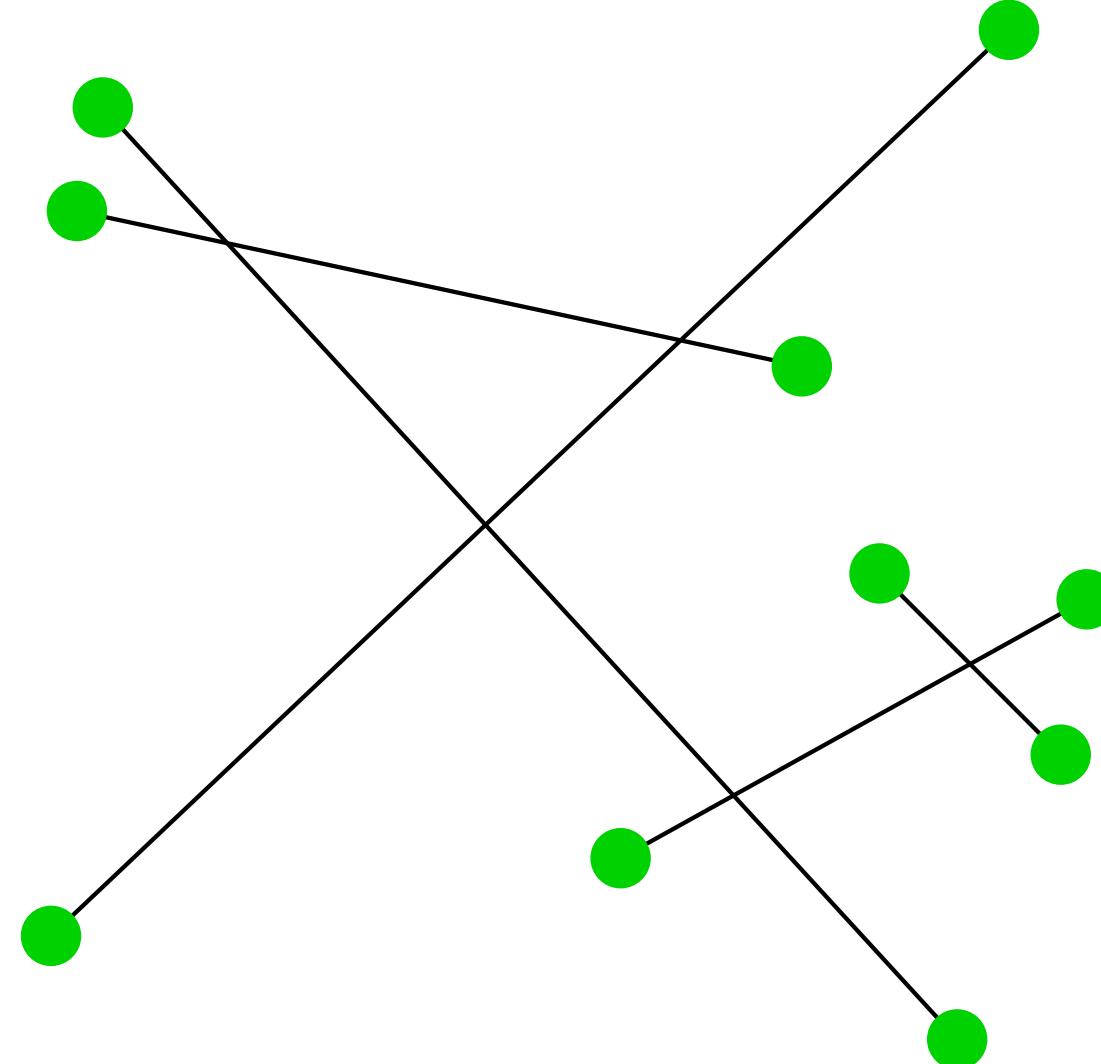
```
handleEvent( $y, \text{finish}, s, T, Q$ ) //  $n \times$ 
     $h := T.\text{locate}(s)$  //  $O(\log n)$ 
    prev := pred( $h$ ) //  $O(1)$ 
    next := succ( $h$ ) //  $O(1)$ 
     $T.\text{remove}(s)$  //  $O(\log n)$ 
    findNewEvent(prev, next)
```



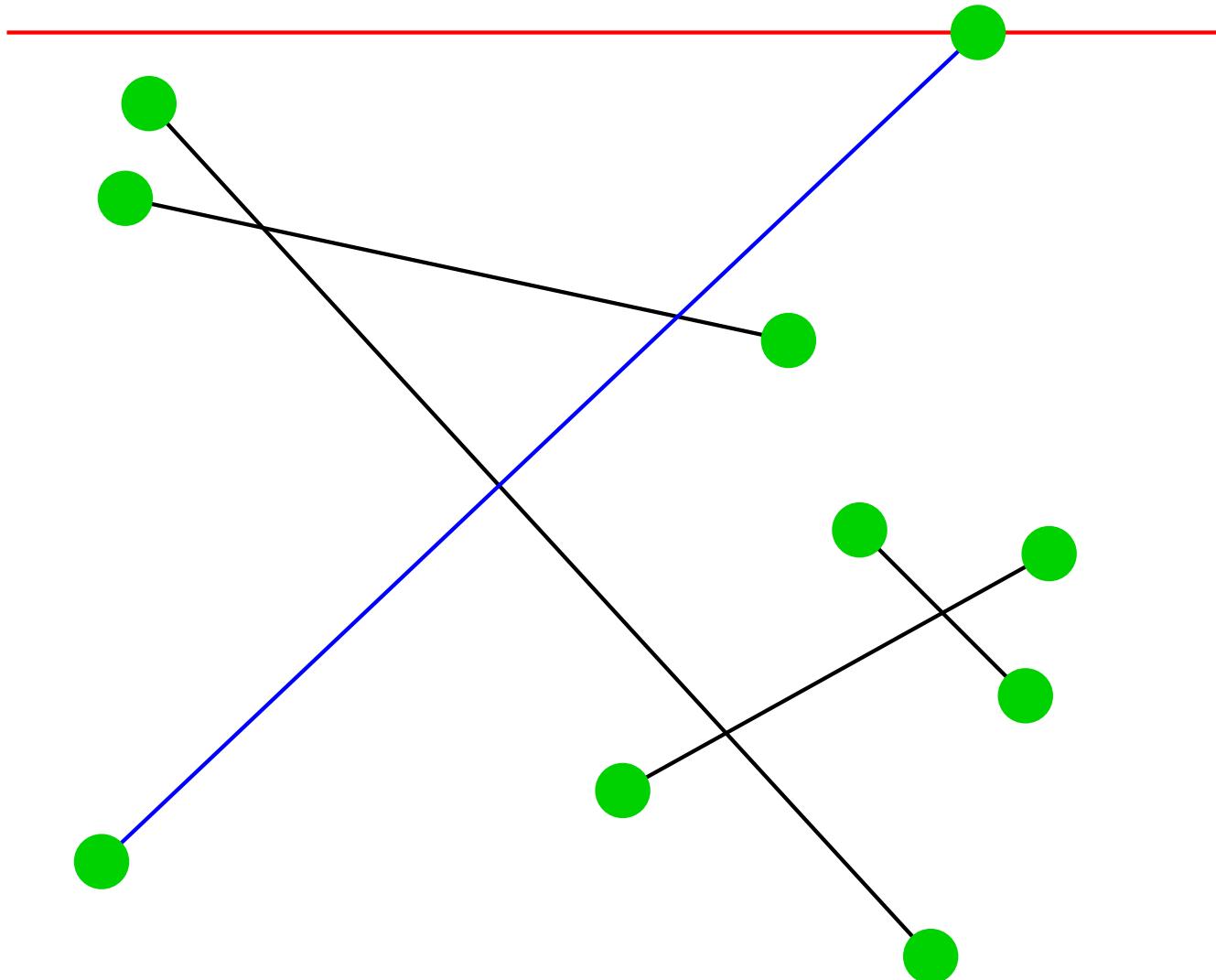
```
handleEvent(y, intersection, (a,b), T, Q)           // k×  
    output (*s ∩ *t)                                // O(1)  
    T.swap(a,b)                                     // O(log n)  
    prev:= pred(b)                                  // O(1)  
    next:= succ(a)                                   // O(1)  
    findNewEvent(prev,b)  
    findNewEvent(a,next)
```



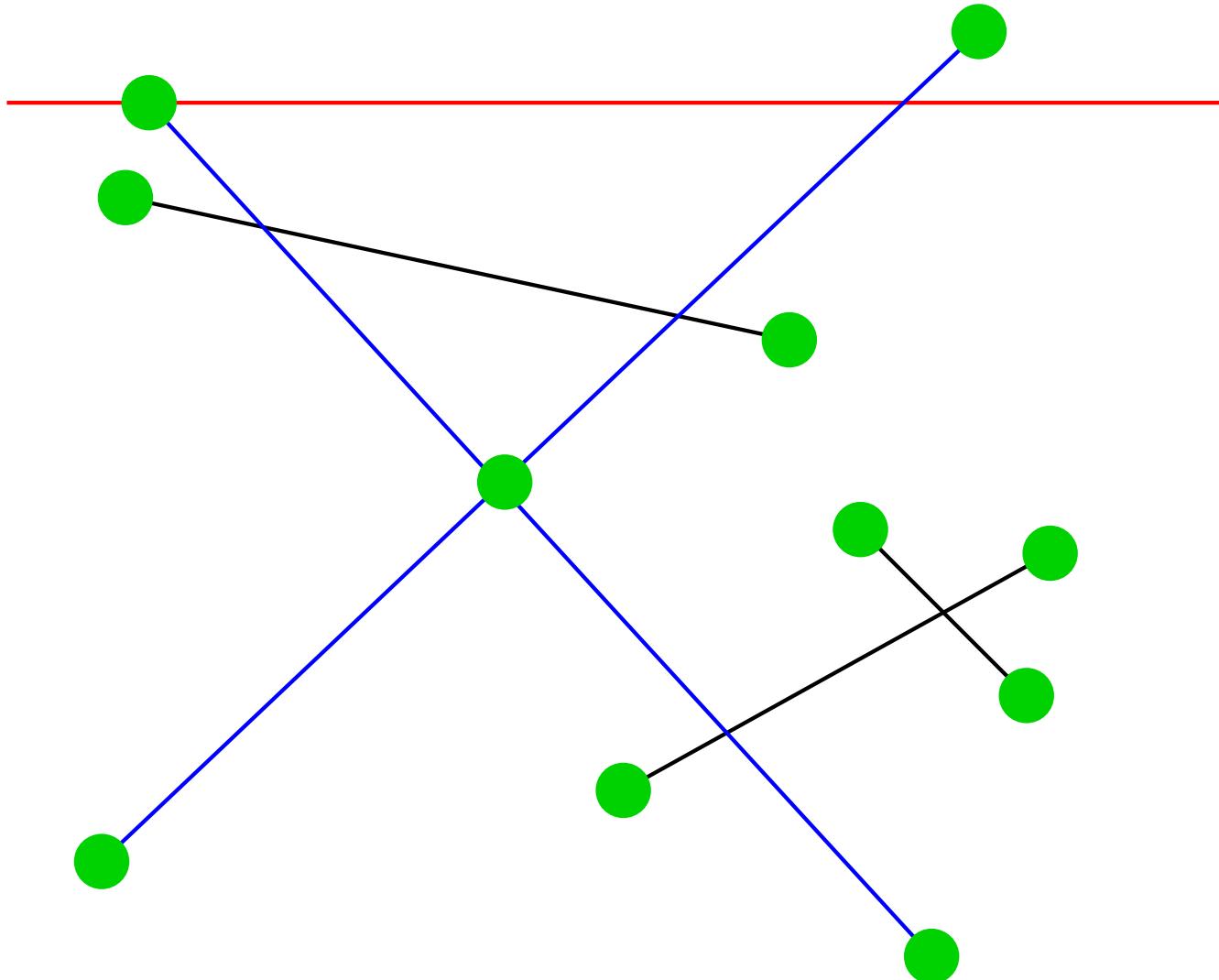
# Verallgemeinerung – Beispiel



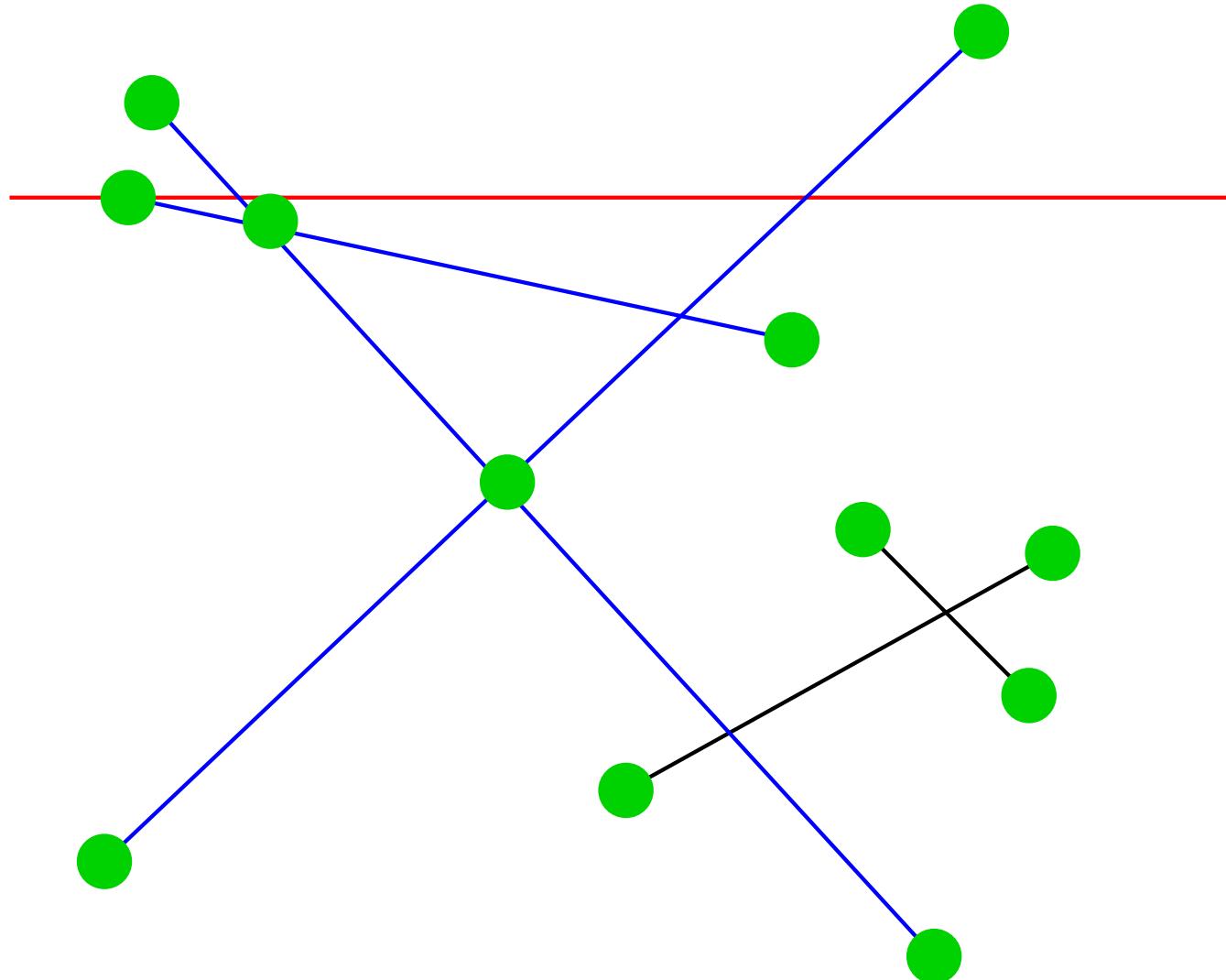
## Verallgemeinerung – Beispiel



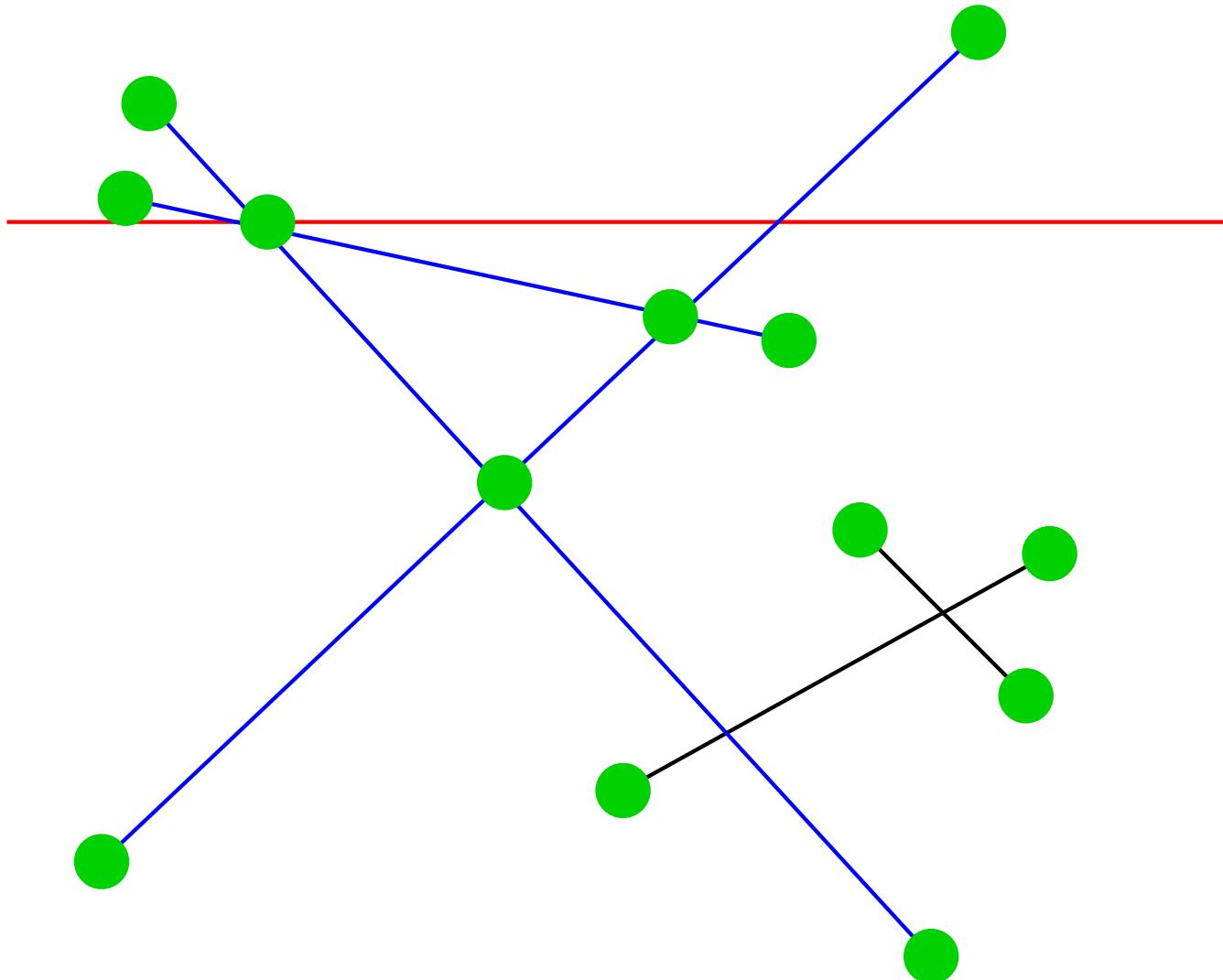
## Verallgemeinerung – Beispiel



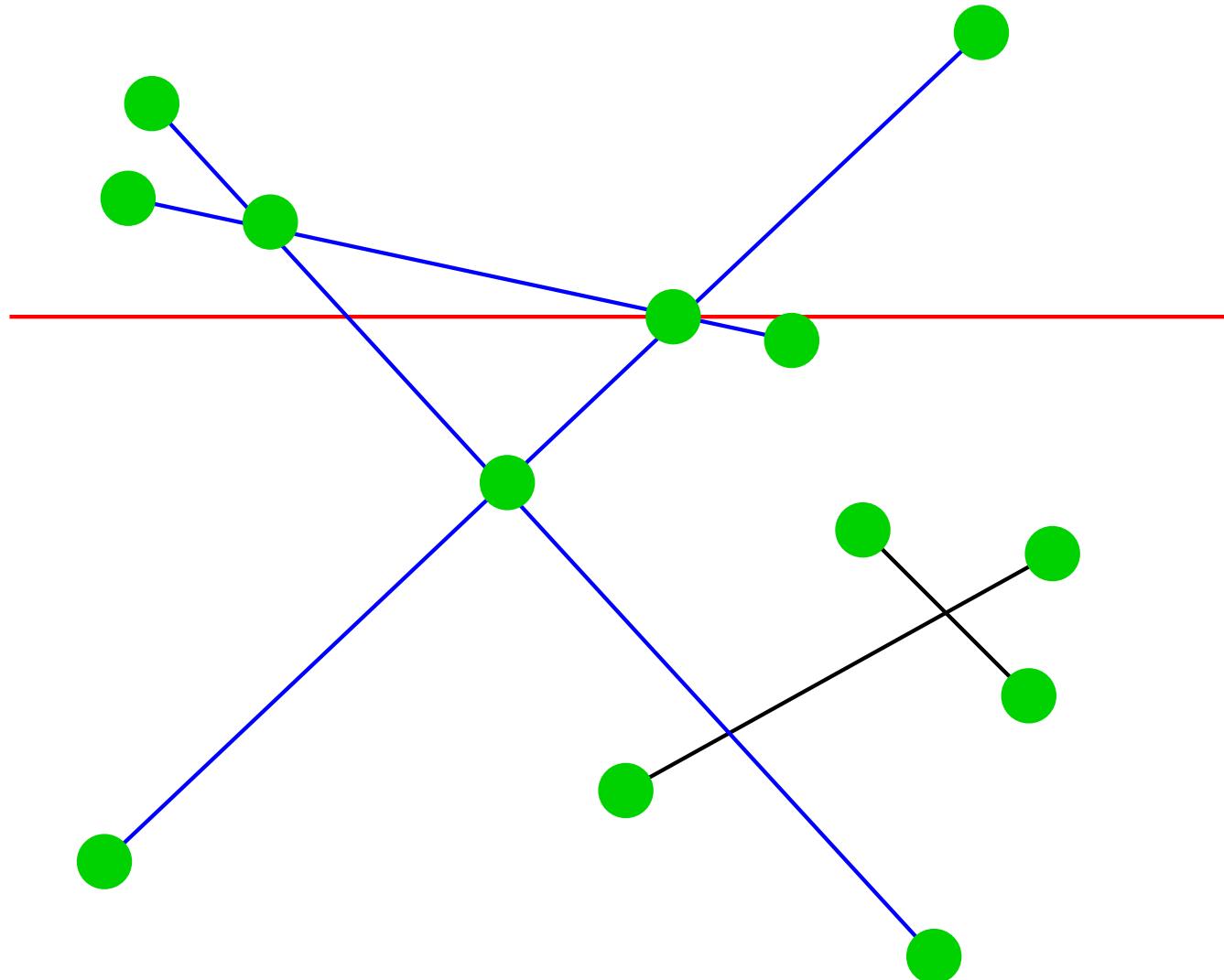
## Verallgemeinerung – Beispiel



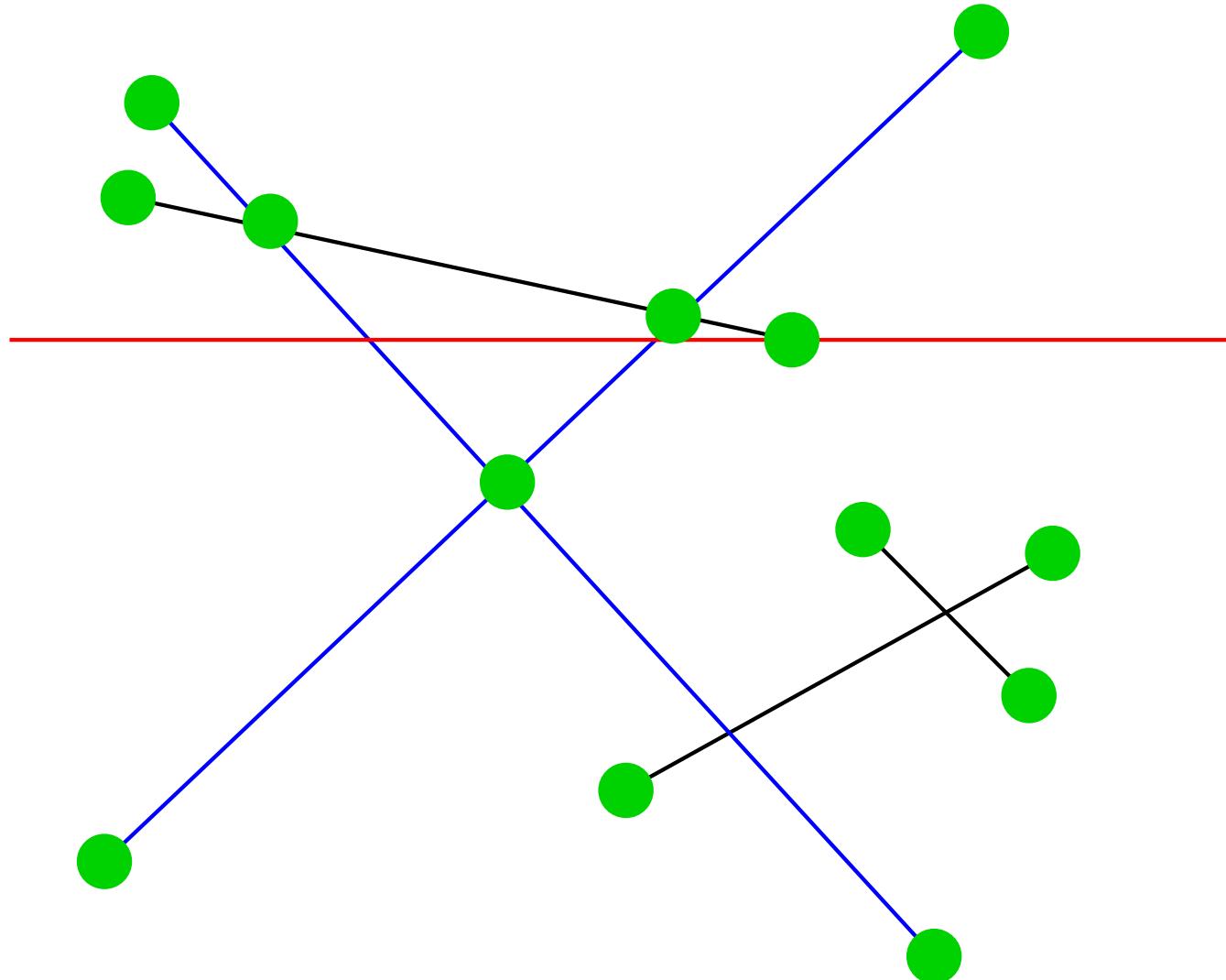
## Verallgemeinerung – Beispiel



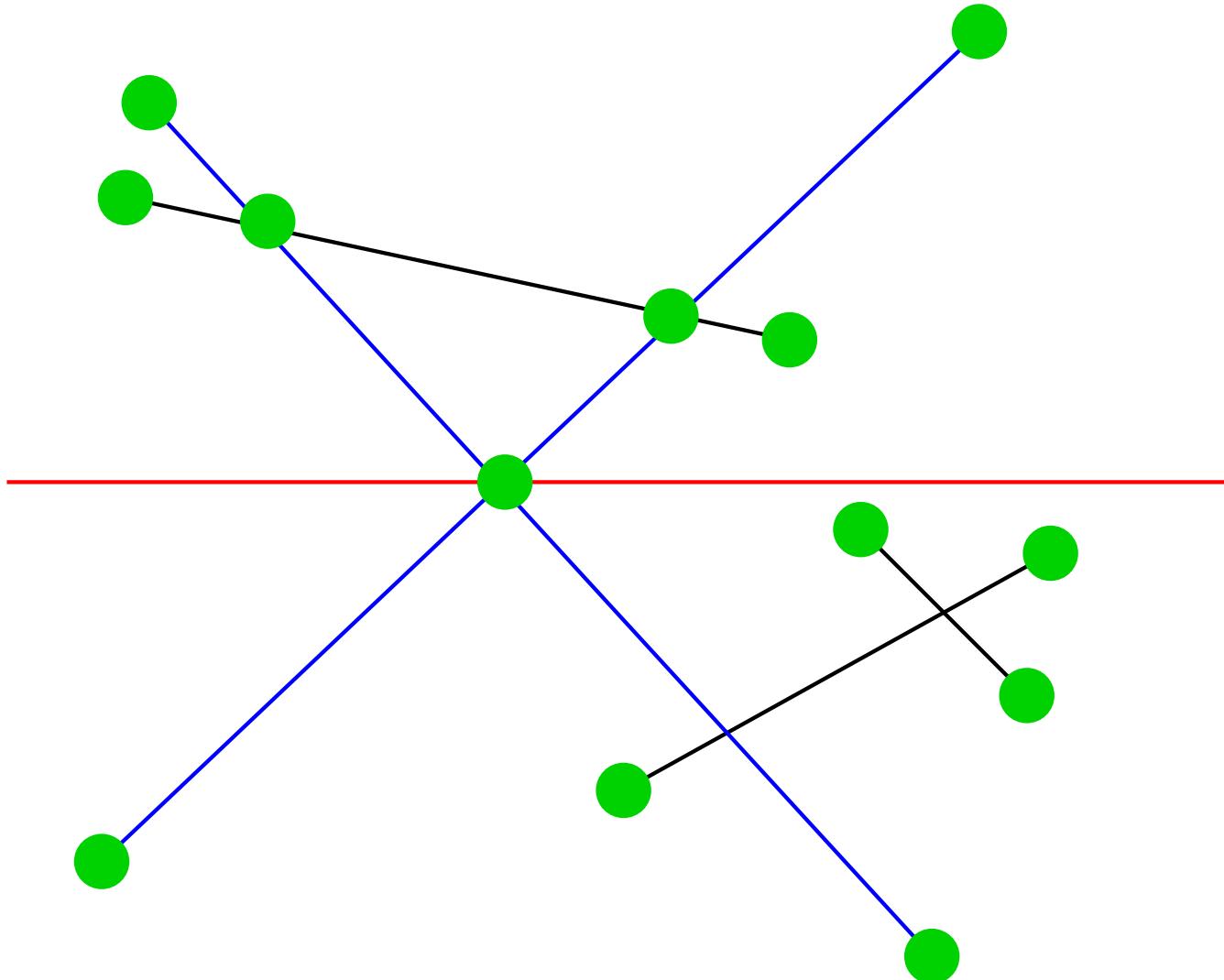
## Verallgemeinerung – Beispiel



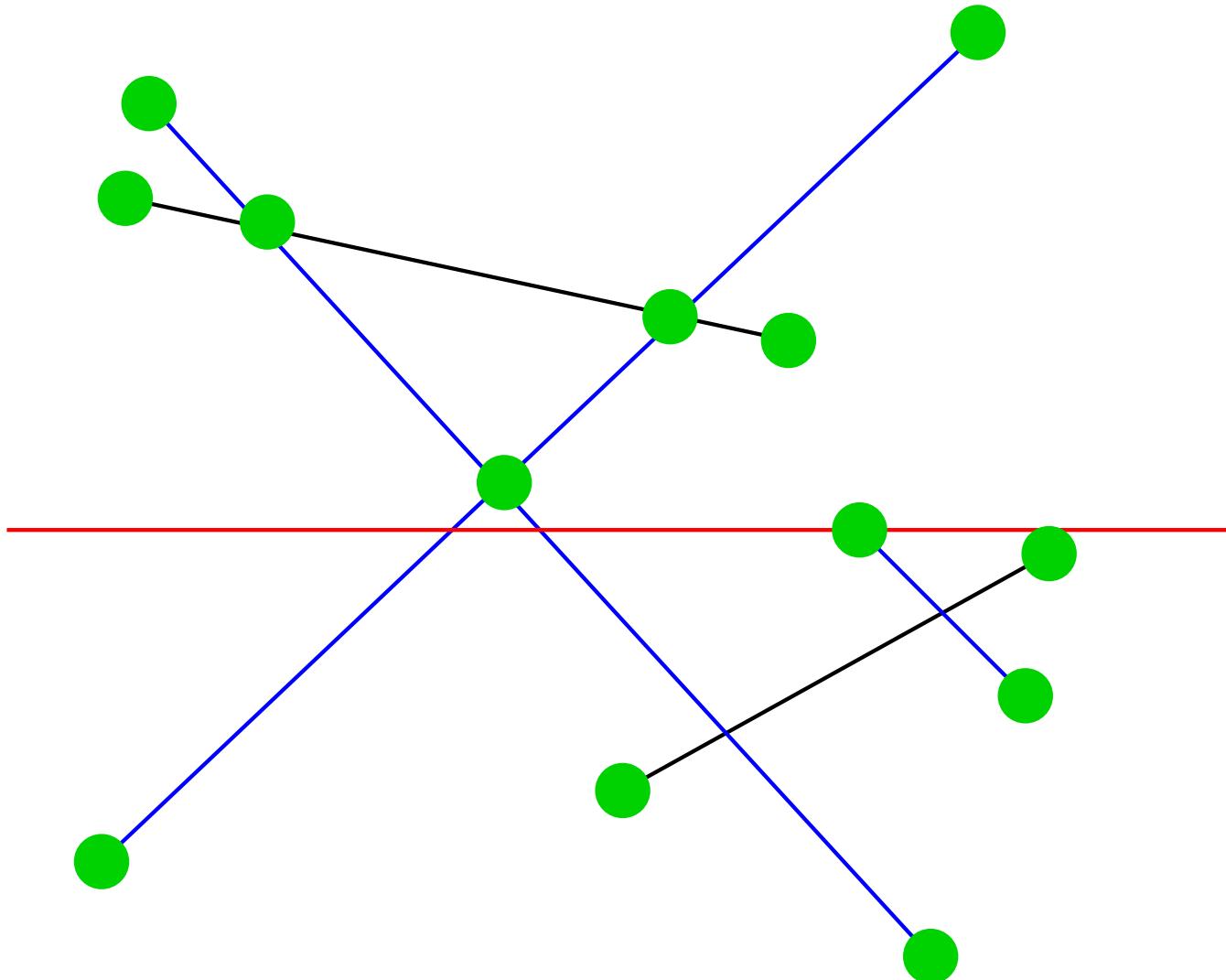
## Verallgemeinerung – Beispiel



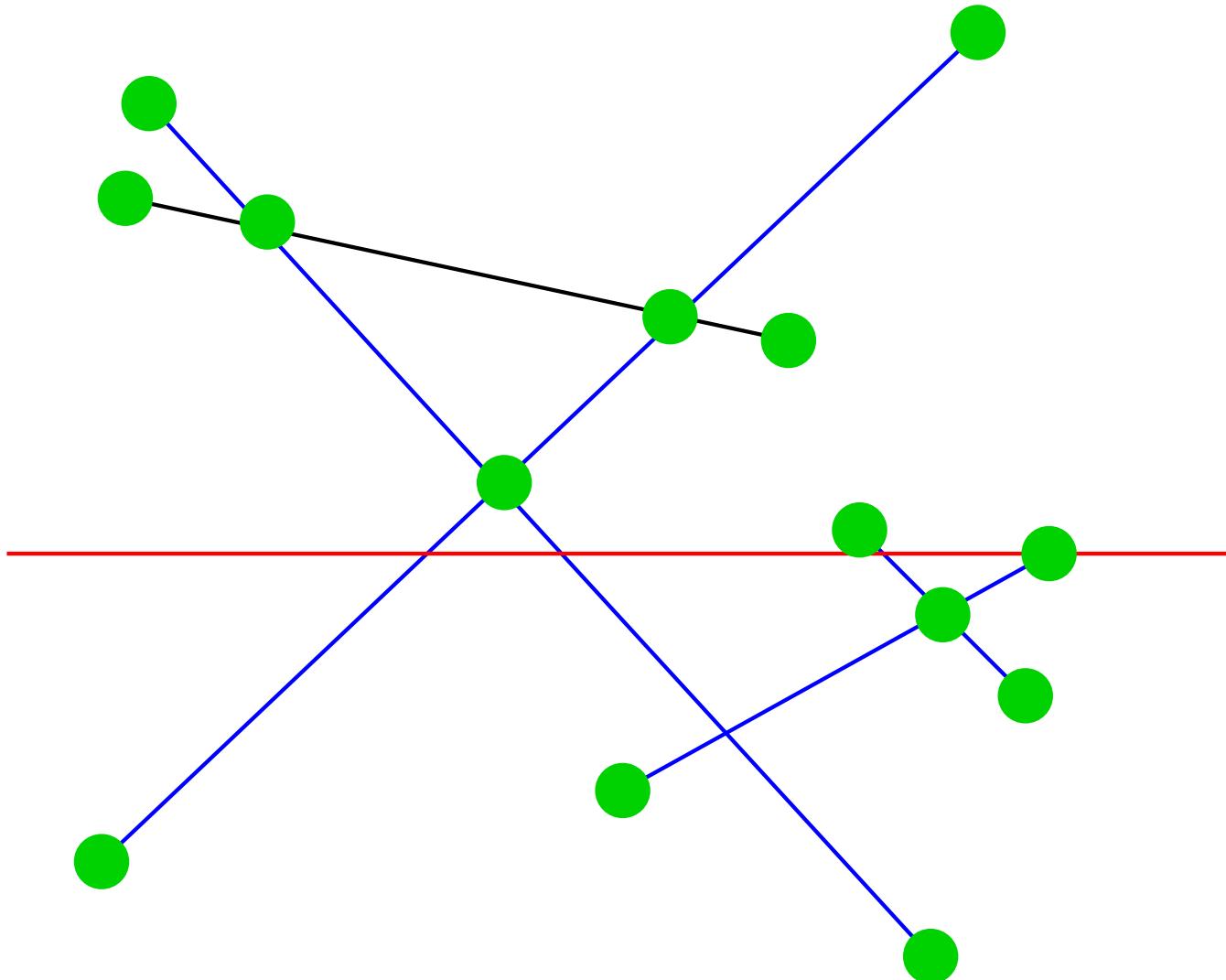
## Verallgemeinerung – Beispiel



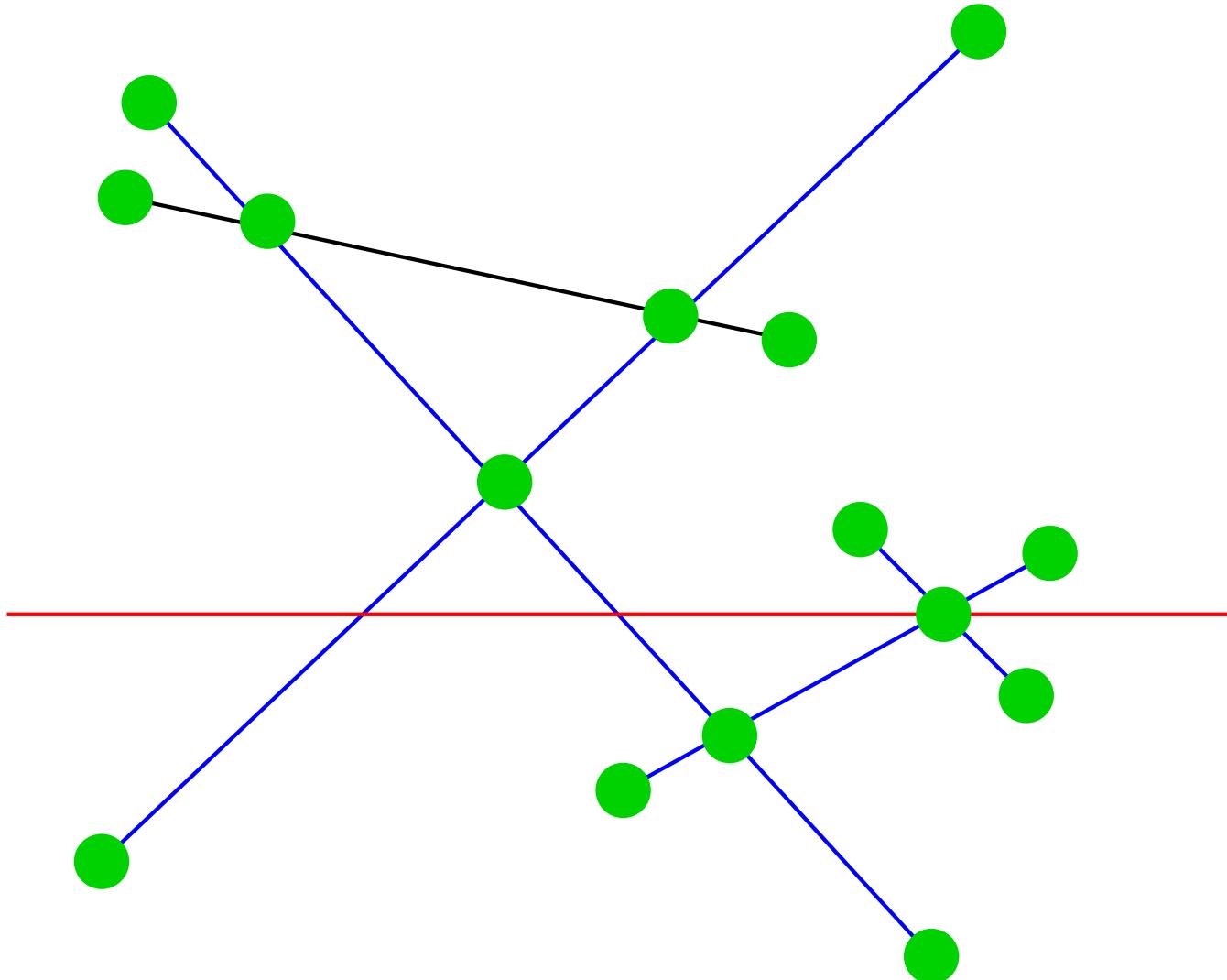
## Verallgemeinerung – Beispiel



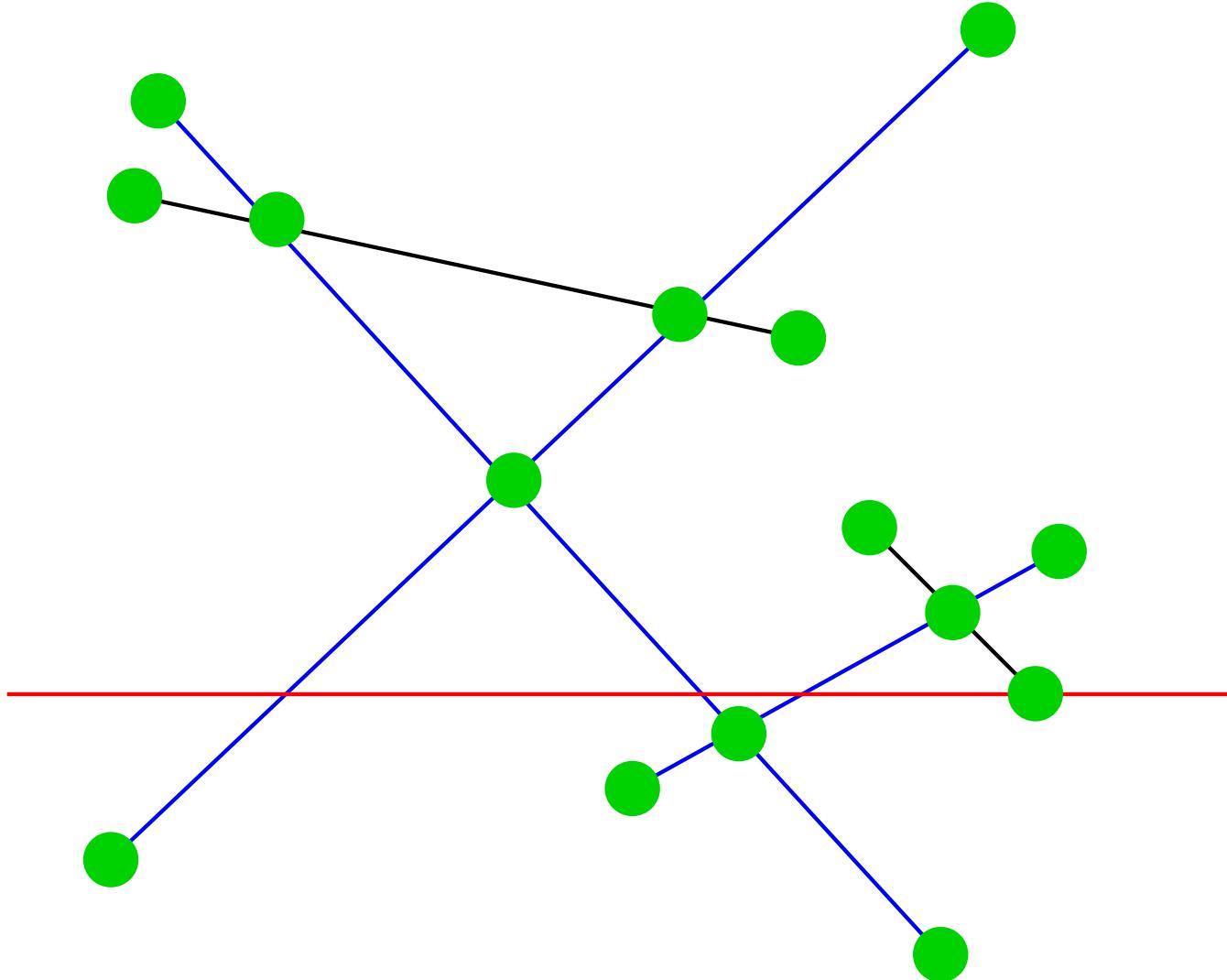
# Verallgemeinerung – Beispiel



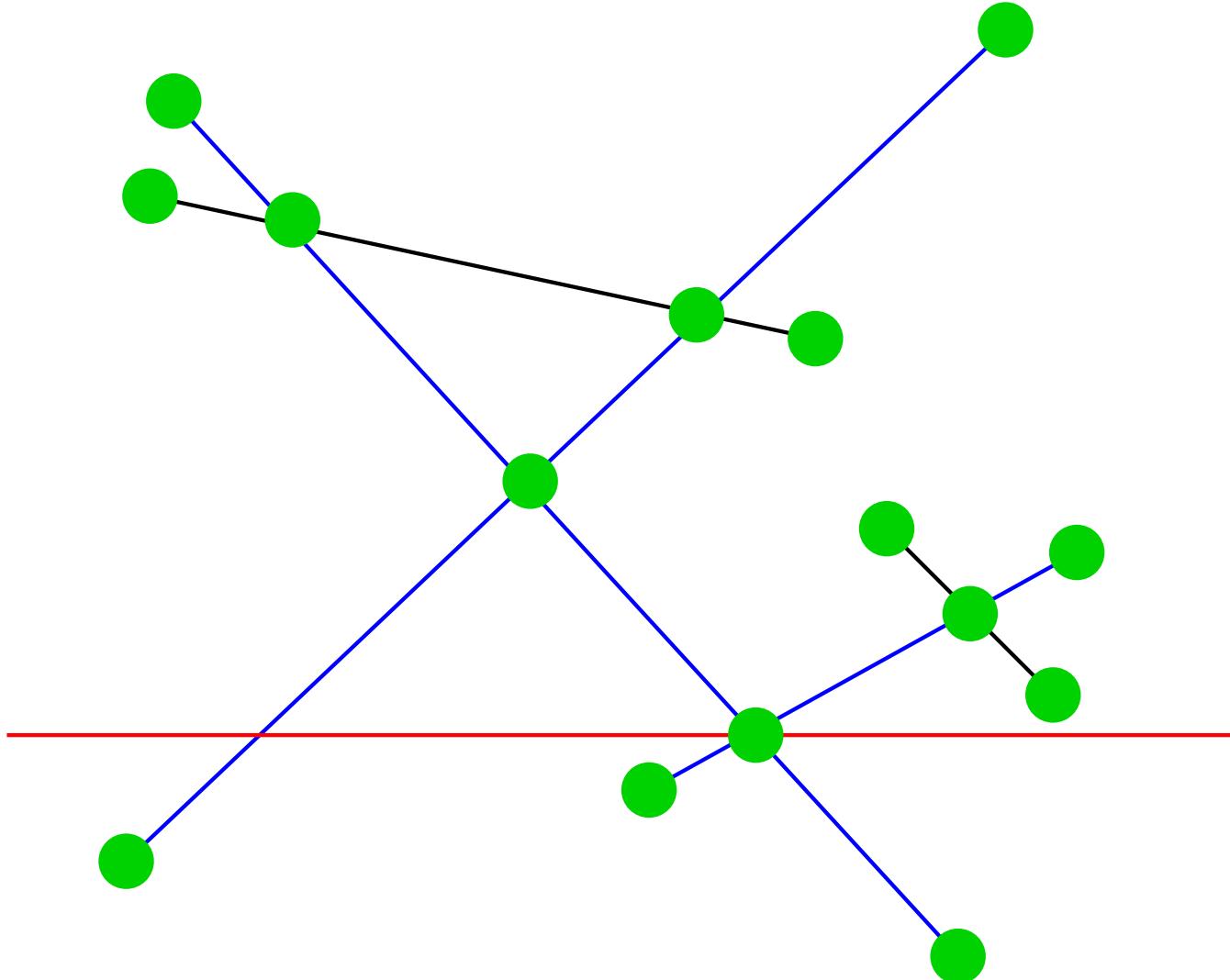
## Verallgemeinerung – Beispiel



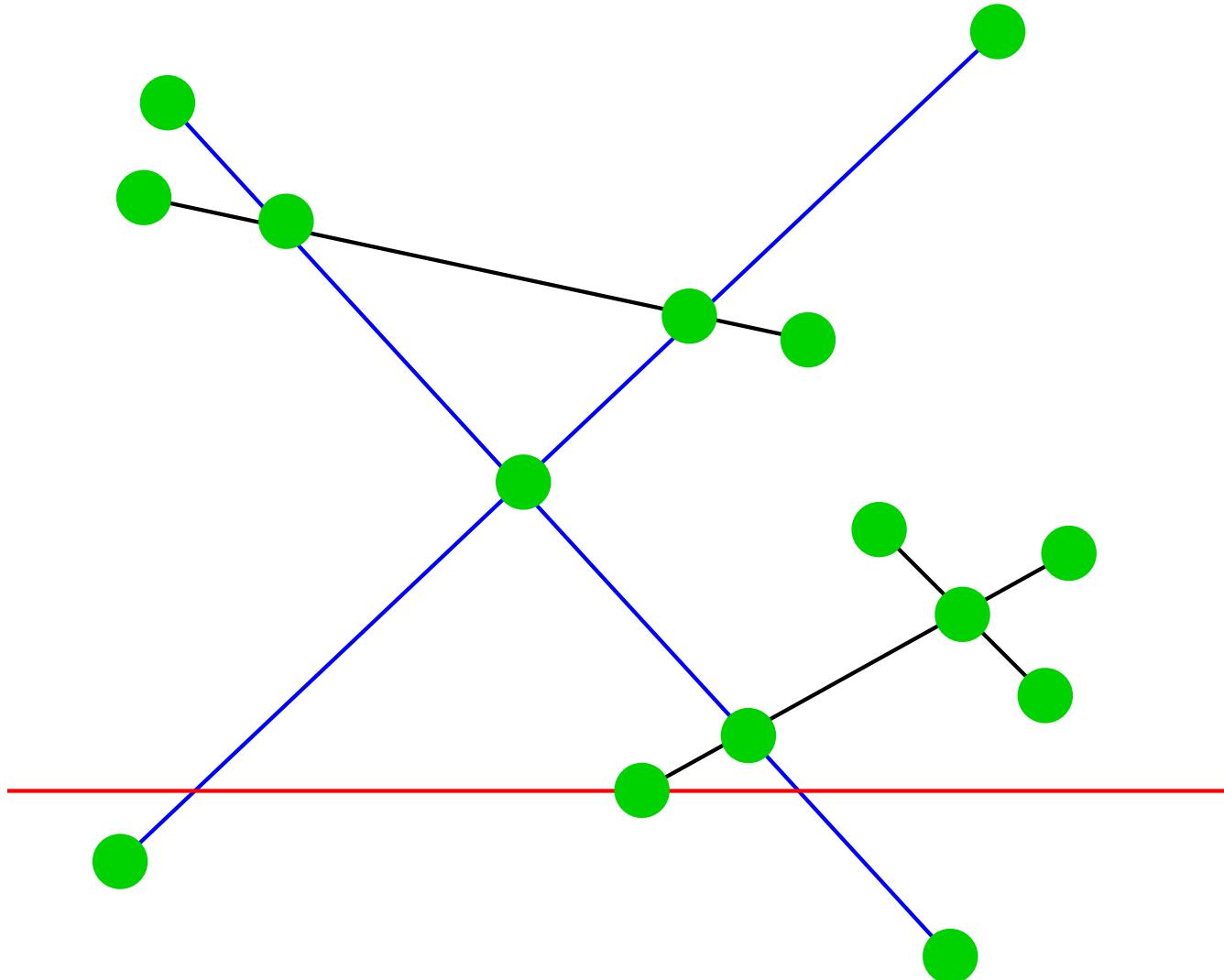
## Verallgemeinerung – Beispiel



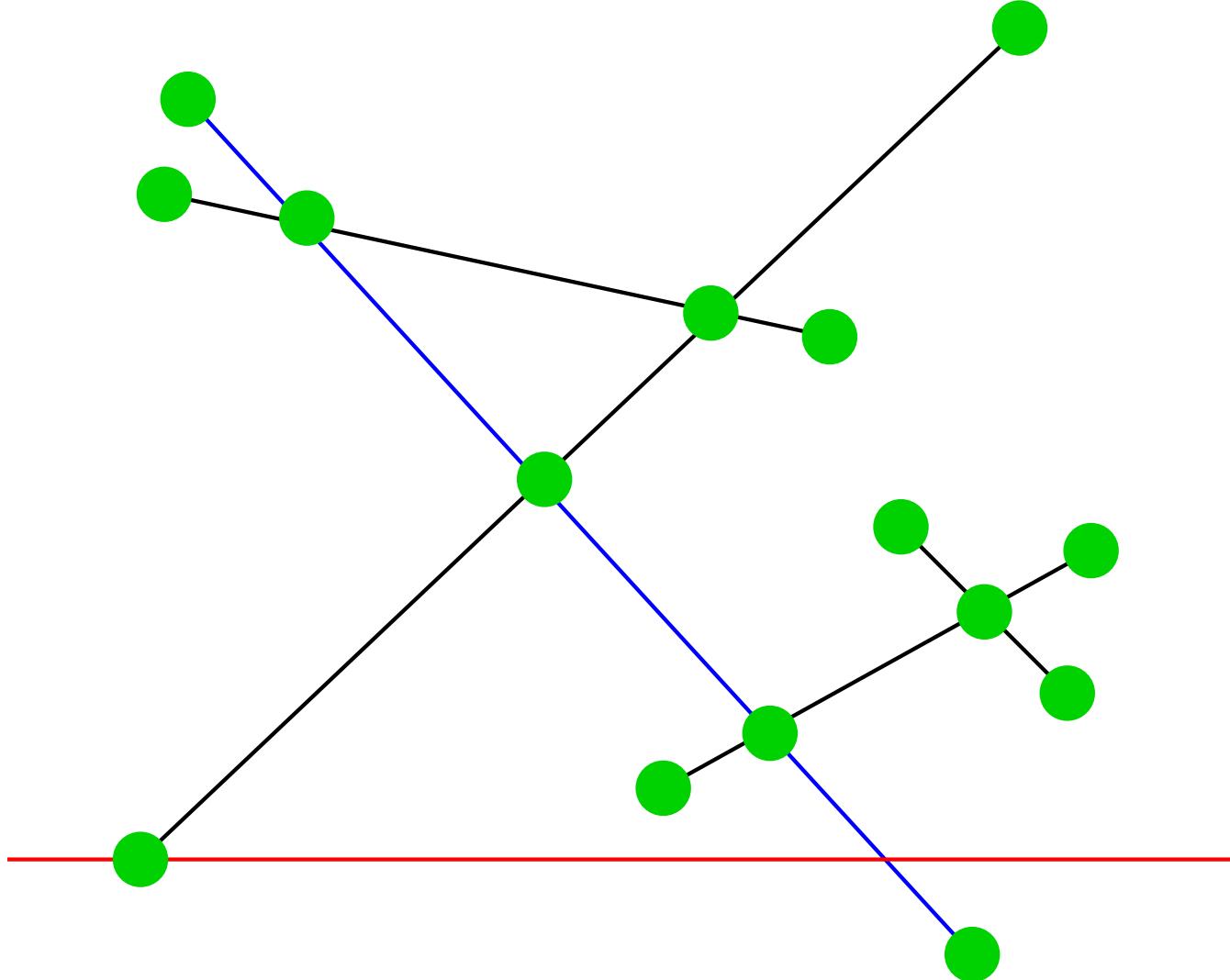
## Verallgemeinerung – Beispiel



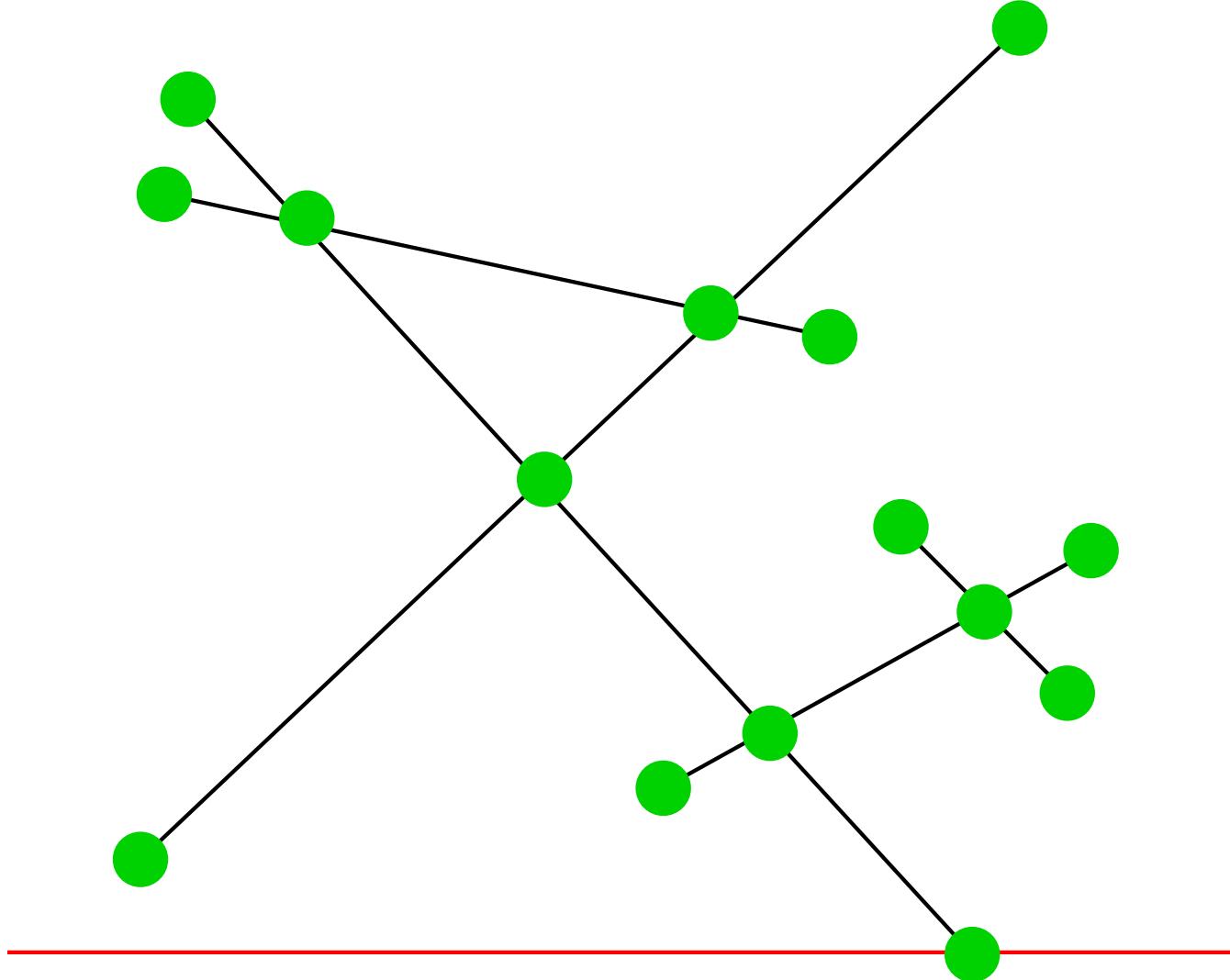
## Verallgemeinerung – Beispiel



## Verallgemeinerung – Beispiel



## Verallgemeinerung – Beispiel



## Verallgemeinerung – Analyse

Insgesamt:  $O((n + k) \log n)$

# Verallgemeinerung – jetzt (fast) wirklich

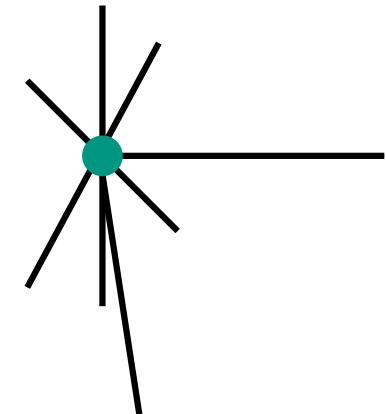
Verbleibende Annahme: Keine Überlappungen

Ordnung für  $Q$ :  $(x, y) \prec (x', y') \Leftrightarrow y > y' \vee y = y' \wedge x < x'$   
(verquere lexikographische Ordnung)

Interpretation: infinitesimal ansteigende Sweep-Line



$Q$  speichert Mengen von Ereignissen mit gleichem  $(x, y)$





handleEvent( $p = (x, y)$ )

$U :=$  segments starting at  $p$  // from  $Q$

$C :=$  segments with  $p$  in their interior // from  $T$

$L :=$  segments finishing at  $p$  // from  $Q$

**if**  $|U| + |C| + |L| \geq 2$  **then** report intersection @  $p$

$T.\text{remove}(L \cup C)$

$T.\text{insert}(C \cup U)$  such that order just below  $p$  is correct

**if**  $U \cup C = \emptyset$  **then**

    findNewEvent( $T.\text{findPred}(p), T.\text{findSucc}(p), p$ )

**else**

    findNewEvent( $T.\text{findPred}(p), T.\text{findLeftmost}(p), p$ )

    findNewEvent( $T.\text{findRightmost}(p), T.\text{findSucc}(p), p$ )

**findNewEvent**( $s, t, p$ )

**if**  $s$  and  $t$  intersect at a point  $p' \succ p$  **then**

**if**  $p' \notin Q$  **then**  $Q.\text{insert}(p')$

## Überlappungen finden

Für jede Strecke  $s$  berechne die Gerade  $g(s)$ , auf der  $s$  liegt

Sortiere  $S$  nach  $g(s)$

1D Überlappungsproblem für jede auftretende Gerade.

## Platzverbrauch

Im Moment:  $\Theta(n + k)$

Reduktion auf  $O(n)$ :

lösche Schnittpunkte zwischen nicht benachbarten Strecken aus  $T$ .

Die werden ohnehin wieder eingefügt wenn sie wieder benachbart werden.

## Mehr Linienschnitt

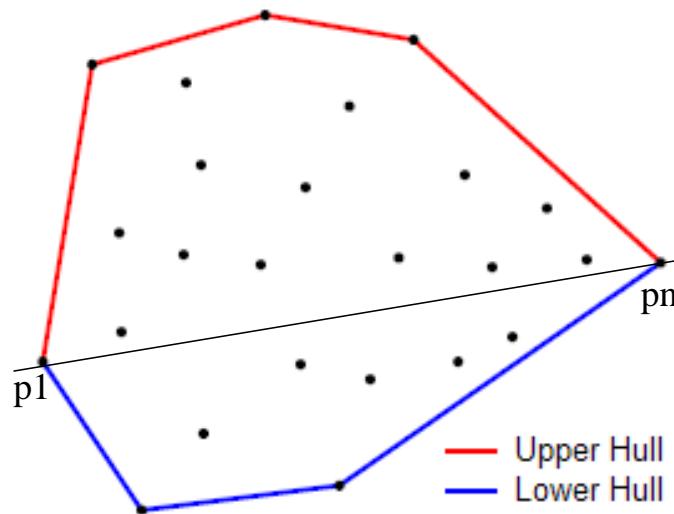
- [Bentley Ottmann 1979] Zeit  $O((n+k)\log n)$
- [Chazelle Edelsbrunner 1988] Zeit  $O(n \log n + k)$
- [Pach Sharir 1991] Zeit  $O((n+k)\log n)$ , Platz  $O(n)$
- [Mulmuley 1988] erwartete Zeit  $O(n \log n + k)$ , Platz  $O(n)$
- [Balaban 1995] Zeit  $O(n \log n + k)$ , Platz  $O(n)$

## 12.2 2D Konvexe Hülle

**Gegeben:** Menge  $P = \{p_1, \dots, p_n\}$  von Punkten in  $\mathbb{R}^2$

**Gesucht:** Konvexes Polygon  $K$  mit Eckpunkten aus  $P$  und  $P \subseteq K$ .

Wir geben einen einfachen Algorithmus, der in Zeit  $O(\text{sort}(n))$  läuft.



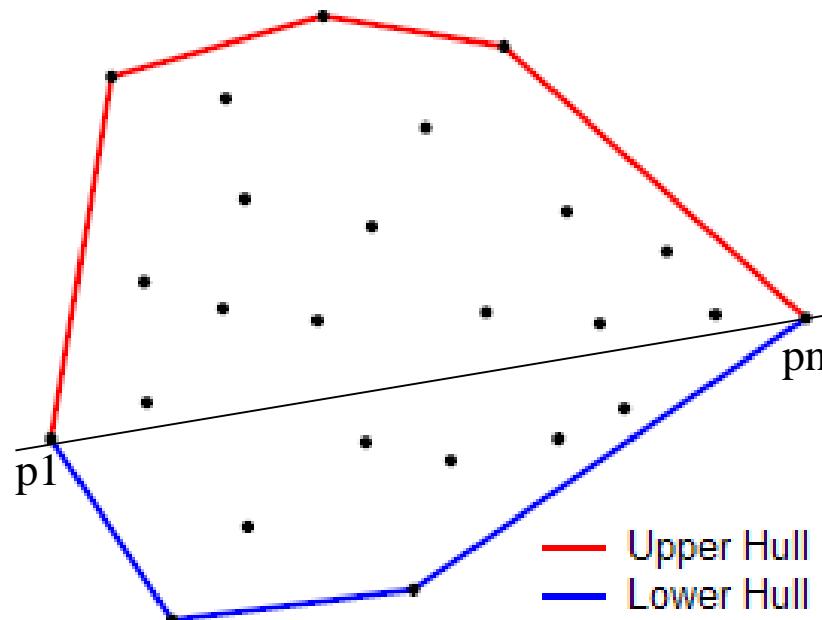
# Konvexe Hülle

sortiere  $P$  lexikographisch nach  $(x, y)$ , d.h., ab jetzt

$$p_1 < p_2 < \dots < p_n$$

OBdA:

Wir berechnen nur die obere Hülle von Punkten oberhalb von  $\overline{p_1 p_n}$



# Graham's Scan [Graham 1972, Andrew 1979]

**Function** `upperHull( $p_1, \dots, p_n$ )`

$L = \langle p_n, p_1, p_2 \rangle$  : Stack of Point

**invariant**  $L$  is the upper hull of  $\langle p_n, p_1, \dots, p_i \rangle$

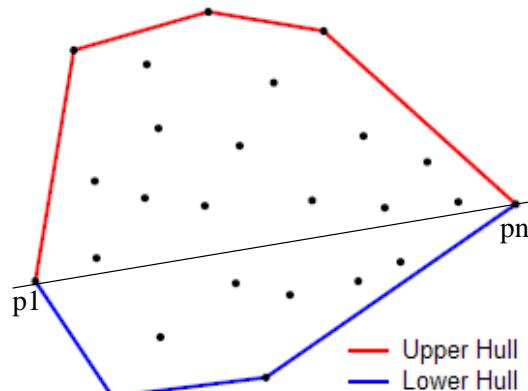
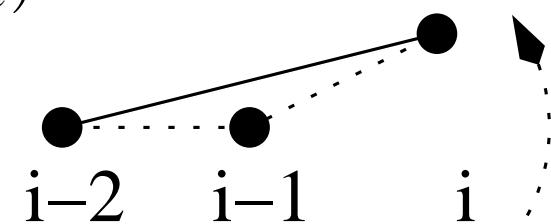
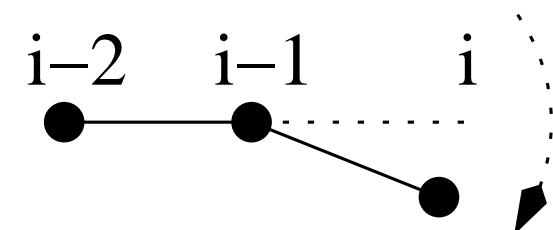
**for**  $i := 3$  **to**  $n$  **do**

**while**  $\neg \text{rightTurn}(L.\text{secondButlast}, L.\text{last}, p_i)$  **do**

$L.\text{pop}$

$L := L \circ \langle p_i \rangle$

**return**  $L$



# Graham's Scan – Beispiel

**Function**  $\text{upperHull}(p_1, \dots, p_n)$

$L = \langle p_n, p_1, p_2 \rangle$  : Stack **of** Point

**invariant**  $L$  is the upper hull of  $\langle p_n, p_1, \dots, p_i \rangle$

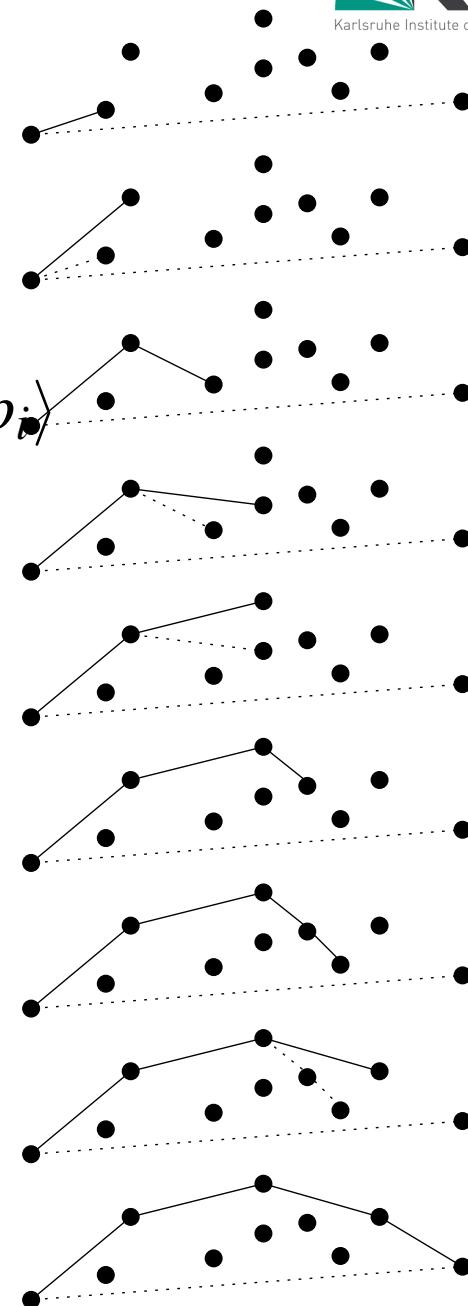
**for**  $i := 3$  **to**  $n$  **do**

**while**  $\neg \text{rightTurn}(L.\text{secondButlast},$   
 $L.\text{last}, p_i)$  **do**

$L.\text{pop}$

$L := L \circ \langle p_i \rangle$

**return**  $L$



# Graham's Scan – Analyse

**Function** upperHull( $p_1, \dots, p_n$ )

$L = \langle p_n, p_1, p_2 \rangle$  : Stack **of** Point

**invariant**  $L$  is the upper hull of  $\langle p_n, p_1, \dots, p_i \rangle$

**for**  $i := 3$  **to**  $n$  **do**

**while**  $\neg \text{rightTurn}(L.\text{secondButlast}, L.\text{last}, p_i)$  **do**

$L.\text{pop}$

$L := L \circ \langle p_i \rangle$

**return**  $L$

Sortieren + $O(n)$

Wieviele Iterationen der While-Schleife insgesamt?

## 3D Konvexe Hülle

Geht in Zeit  $O(n \log n)$  [Preparata Hong 1977]

**Konvexe Hülle,  $d \geq 4$**

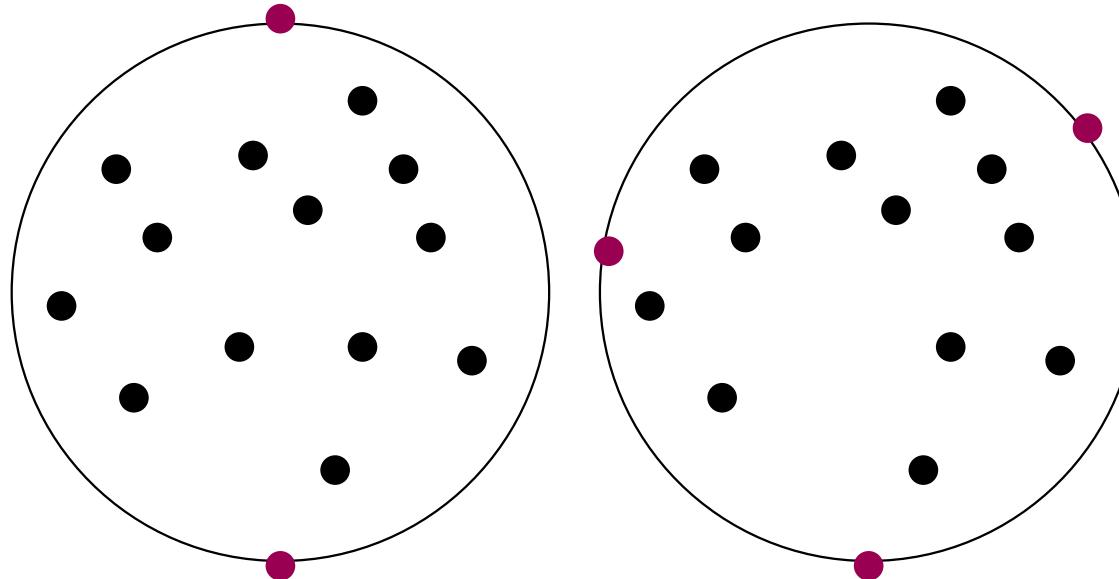
Ausgabekomplexität  $O\left(n^{\lfloor d/2 \rfloor}\right)$

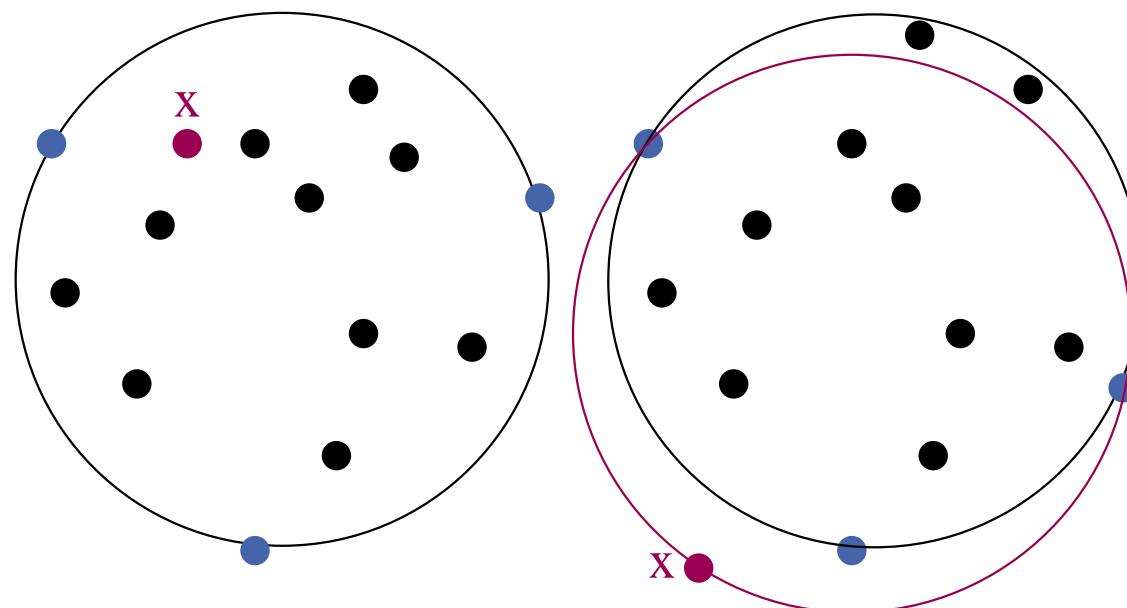
## 12.3 Kleinstes einschließende Kugel

**Gegeben:** Menge  $P = \{p_1, \dots, p_n\}$  von Punkten in  $\mathbb{R}^d$

**Gesucht:** Kugel  $K$  mit minimalem Radius, so dass  $P \subseteq K$ .

Wir geben einen einfachen Algorithmus, der in erwarteter Zeit  $O(n)$  läuft. [Welzl 1991].



**Function**  $\text{smallestEnclosingBallWithPoints}(P, Q)$ **if**  $|P| = 0 \vee |Q| = d + 1$  **then return**  $\text{ball}(Q)$ pick random  $x \in P$  $B := \text{smallestEnclosingBallWithPoints}(P \setminus \{x\}, Q)$ **if**  $x \in B$  **then return**  $B$ **return**  $\text{smallestEnclosingBallWithPoints}(P \setminus \{x\}, Q \cup \{x\})$ 

# Kleinste einschließende Kugel – Korrektheit

**Function**  $\text{smallestEnclosingBallWithPoints}(P, Q)$

**if**  $|P| = 1 \vee |Q| = d + 1$  **then return**  $\text{ball}(Q)$

pick random  $x \in P$

$B := \text{smallestEnclosingBallWithPoints}(P \setminus \{x\}, Q)$

**if**  $x \in B$  **then return**  $B$

**return**  $\text{smallestEnclosingBallWithPoints}(P \setminus \{x\}, Q \cup \{x\})$

z.Z.:  $x \notin B \rightarrow x$  ist auf dem Rand von  $\text{sEB}(P)$

Wir zeigen Kontraposition:

$x$  nicht auf dem Rand von  $\text{sEB}(P)$

$\rightarrow \text{sEB}(P) = \text{sEB}(P \setminus \{x\}) = B$

z.Z.: sEBs sind eindeutig!

Also  $x \in B$

**Lemma:** sEB( $P$ ) ist eindeutig bestimmt.

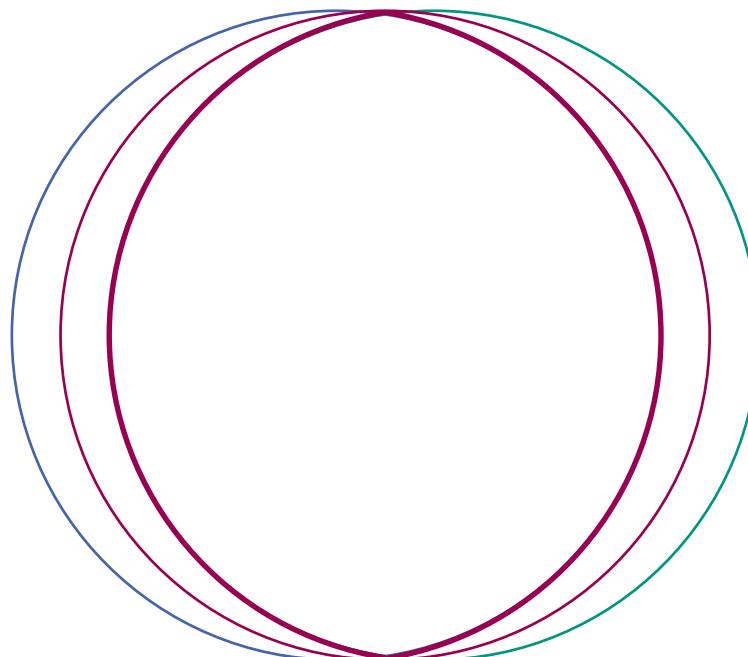
**Beweis:** Annahme,  $\exists$ sEBs  $B_1 \neq B_2$

$$\rightarrow P \subseteq B_1 \wedge P \subseteq B_2$$

$$\rightarrow P \subseteq B_1 \cap B_2 \subseteq \text{sEB}(B_1 \cap B_2) =: B$$

Aber dann ist  $\text{radius}(B) < \text{radius}(B_1)$

Widerspruch zur Annahme, dass  $B_1$  eine sEB ist. □



# Kleinste einschließende Kugel – Analyse

Wir zählen die erwartete Anzahl der Tests  $x \in B, T(p, q)$ .

$$T(p, d+1) = T(1, p) = 0 \quad \text{Basis der Rekurrenz}$$

$$T(p, q) \leq 1 + T(p-1, q) + \mathbb{P}[x \notin B] T(p, q+1)$$

$$\leq 1 + T(p-1, q) + \frac{d+1-q}{p} T(p, q+1)$$

# Kleinste einschließende Kugel – Analyse, $d = 2$

$$T(p, d+1) = T(1, p) = 0$$

$$T(p, q) \leq 1 + T(p-1, q) + \frac{d+1-q}{p} T(p, q+1)$$

$$T(p, 2) \leq 1 + T(p-1, 2) + \frac{1}{p} T(p, 3) \leq 1 + T(p-1, 2) \leq p$$

$$\begin{aligned} T(p, 1) &\leq 1 + T(p-1, 1) + \frac{2}{p} T(p, 2) \\ &\leq 1 + T(p-1, 1) + \frac{2}{p} p = 3 + T(p-1, 1) \leq 3p \end{aligned}$$

$$\begin{aligned} T(p, 0) &\leq 1 + T(p-1, 0) + \frac{3}{p} T(p, 1) \\ &\leq 1 + T(p-1, 0) + \frac{3}{p} 3p = 10 + T(p-1, 0) \leq 10p \end{aligned}$$

# Kleinste einschließende Kugel – Analyse

| $d$ | $T(p, 0)$ |
|-----|-----------|
| 1   | $3n$      |
| 2   | $10n$     |
| 3   | $41n$     |
| 4   | $206n$    |

Allgemein  $T(p, 0) \geq d!n$

# Ähnliche Randomisierte Linearzeitalgorithmen

- Lineare Programmierung mit konstantem  $d$  [Seidel 1991]
- Kleinster einschließender Ellipsoid, Kreisring,...
- Support-Vector-Machines (maschinelles Lernen)
- Alles wo (LP-type problem [Sharir Welzl 1992])
  - $O(1)$  Objekte das Optimum festlegen
  - Objekt  $x$  hinzufügen
    - Lösung bleibt gleich oder ist an Lösungsdef. beteiligt

## 12.4 2D Bereichssuche (range search)

**Daten:**  $P = \{p_1, \dots, p_n\} \subseteq \mathbb{R}^2$

**Anfragen:** achsenparallele Rechtecke  $Q = [x, x'] \times [y, y']$

finde  $P \cap Q$  (range reporting)

oder  $k = |P \cap Q|$  (range counting)

Vorverarbeitung erlaubt.

Vorverarbeitungszeit?  $O(n \log n)$

Platz?  $O(n)$  ?

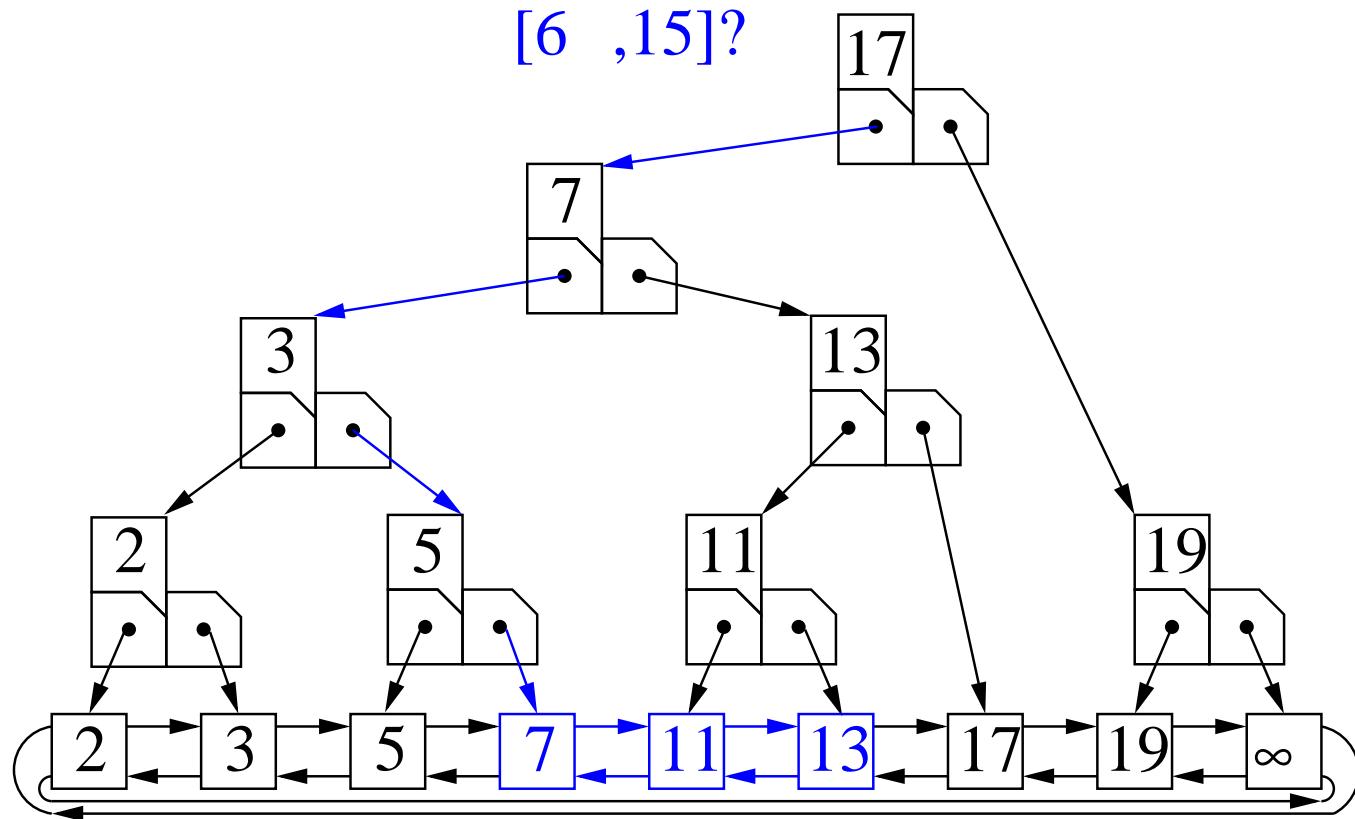
Anfragebearbeitung?

- Counting:  $O(\log n)$

- Reporting:  $O(k + \log n)$  oder wenigstens  $O(k \cdot \log n)$

# 1D Bereichssuche

Suchbaum



Zählanfragen: Teilbaumgrößen speichern

Sogar dynamisch !

## Reduktion auf $1..n \times 1..n$

vereinfachende Annahme: Koordinaten paarweise verschieden.

Ersetze Koordinaten  $P = \{(x_1, y_1), \dots, (x_n, y_n)\}$  durch ihren Rang:

$x_i \rightarrow$  Rang von  $x_i$  in  $\{x_1, \dots, x_n\}$

$y_i \rightarrow$  Rang von  $y_i$  in  $\{y_1, \dots, y_n\}$

# Reduktion auf $1..n \times 1..n$

Ersetze Koordinaten  $P = \{(x_1, y_1), \dots, (x_n, y_n)\}$  durch ihren Rang:

$P_x := \text{sort}(\langle x_1, \dots, x_n \rangle); \quad P_y := \text{sort}(\langle y_1, \dots, y_n \rangle)$

$P := \{( \text{binarySearch}(x, P_x), \text{binarySearch}(y, P_y) ) : (x, y) \in P\}$

**Function** rangeQuery( $[x, x'] \times [y, y']$ )

$x := \text{binarySearchSucc}(x, P_x); \quad x' := \text{binarySearchPred}(x', P_x)$

$y := \text{binarySearchSucc}(y, P_y); \quad y' := \text{binarySearchPred}(y', P_y)$

$R := \text{intRangeQuery}([x, x'] \times [y, y'])$

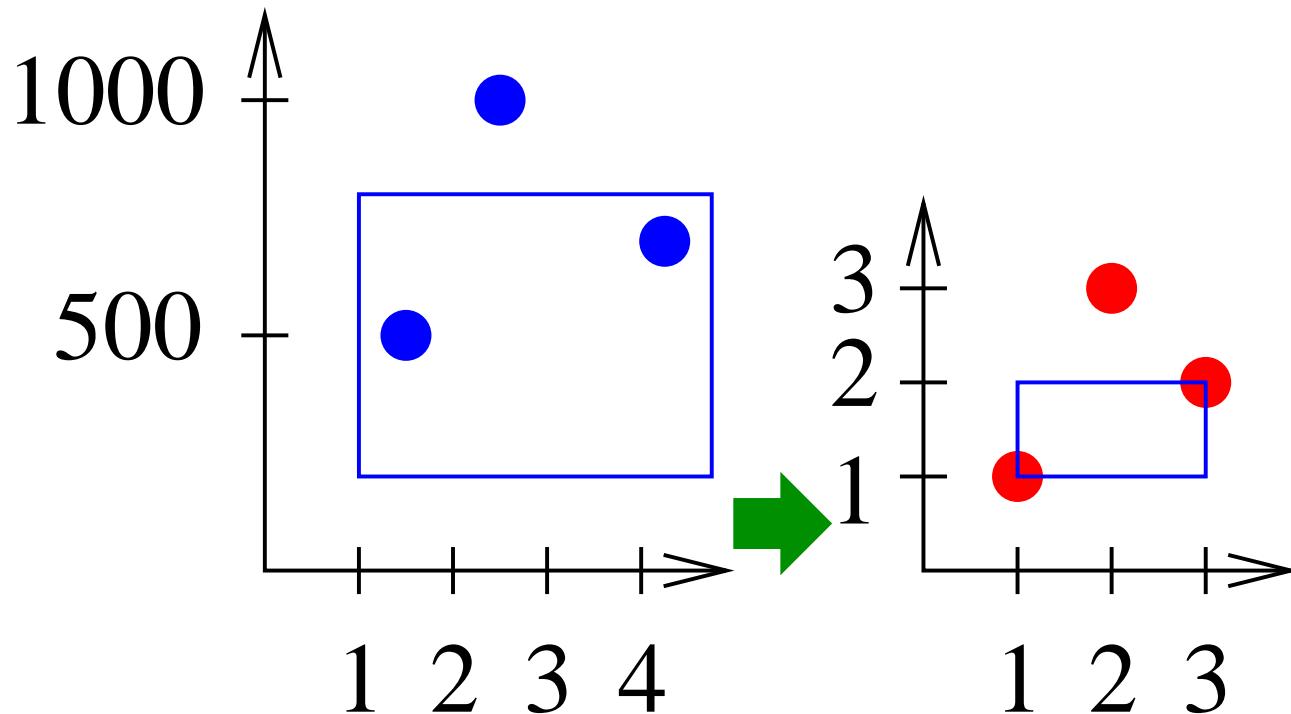
**return**  $\{(P_x[x], P_y[y]) : (x, y) \in R\}$

A Konstruktionszeit  $O(n \log n)$ , Anfragezeit

$O(\log n) + T_{\text{intRangeQuery}}(n)$

# Beispiel

$$\{(2.5, 1000), (1.4, 500), (4.2, 700)\} \rightarrow \{(2, 3), (1, 1), (3, 2)\}$$



# Wavelet Tree

[Chazelle 1988, Grossi/Gupta/Vitter 2003, Mäkinen/Navarro 2007]

**Class** WaveletTree( $X = \langle x_1, \dots, x_n \rangle$ )// represents  $(x_1, 1), \dots, (x_n, n)$

//Constructor:

**if**  $n < n_0$  **then** store  $X$  directly

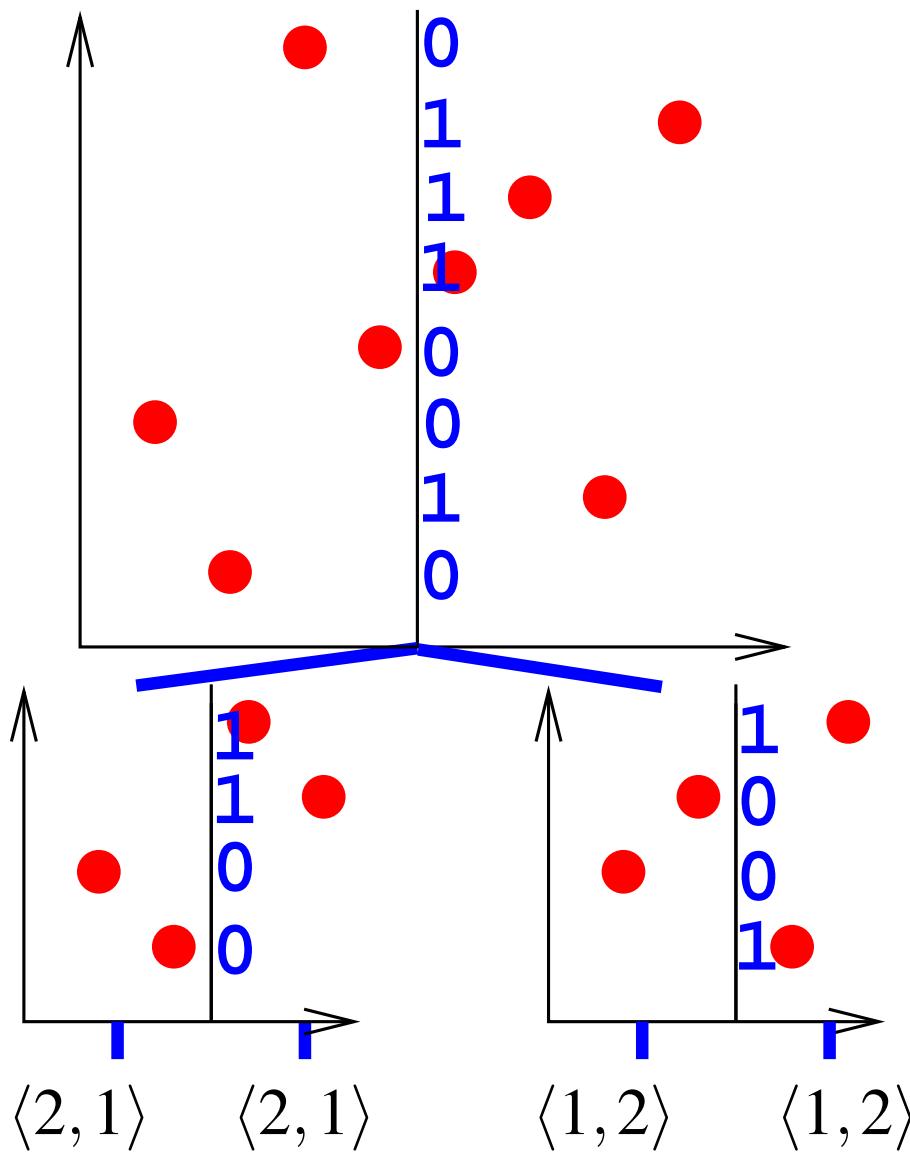
**else**

store bitvector  $b$  with  $b[i] = 1$  iff  $x_i > \lfloor n/2 \rfloor$

$\ell :=$  WaveletTree( $\langle x_i : x_i \leq \lfloor n/2 \rfloor \rangle$ )

$r :=$  WaveletTree( $\langle x_i - \lfloor n/2 \rfloor : x_i > \lfloor n/2 \rfloor \rangle$ )

# Beispiel

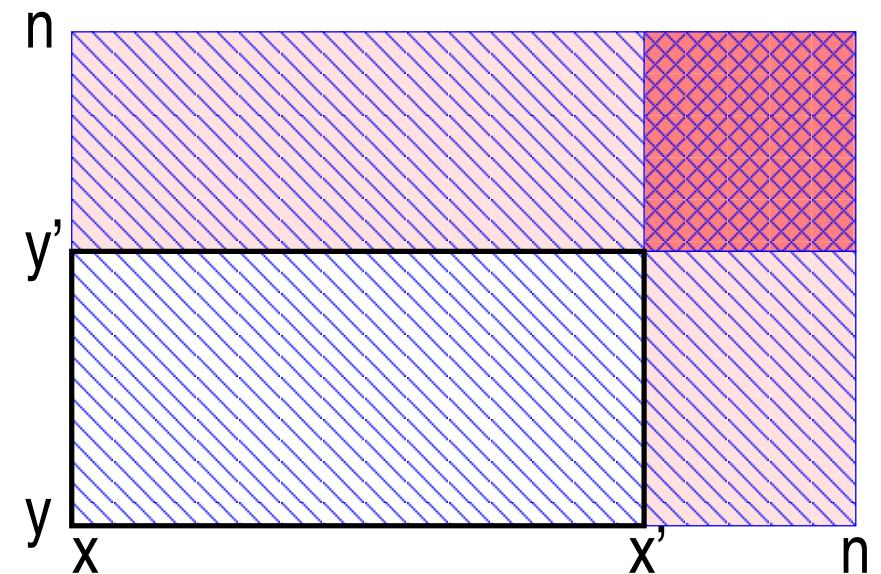


# Wavelet Tree Counting Query

**Function** intRangeCount( $[x, x'] \times [y, y']$ )

**return**

```
intDominanceCount( $x, y$ ) –  
intDominanceCount( $x', y$ ) –  
intDominanceCount( $x, y'$ ) +  
intDominanceCount( $x', y'$ )
```

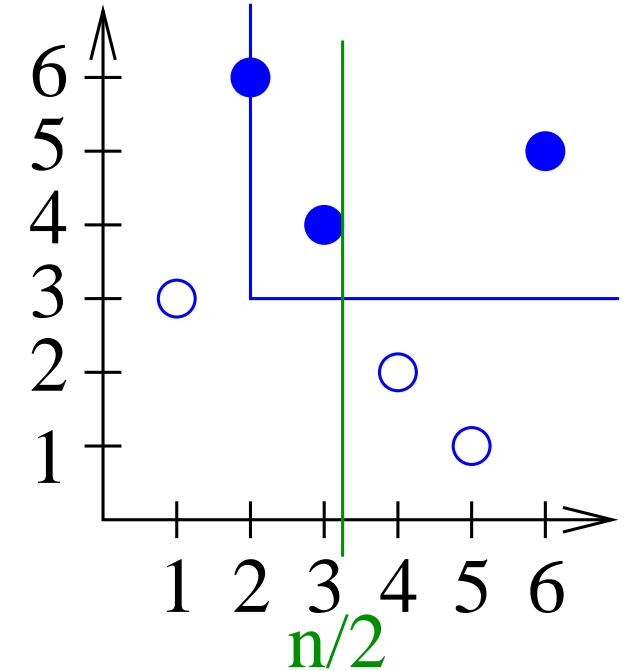


# Wavelet Tree Dominance Counting Query

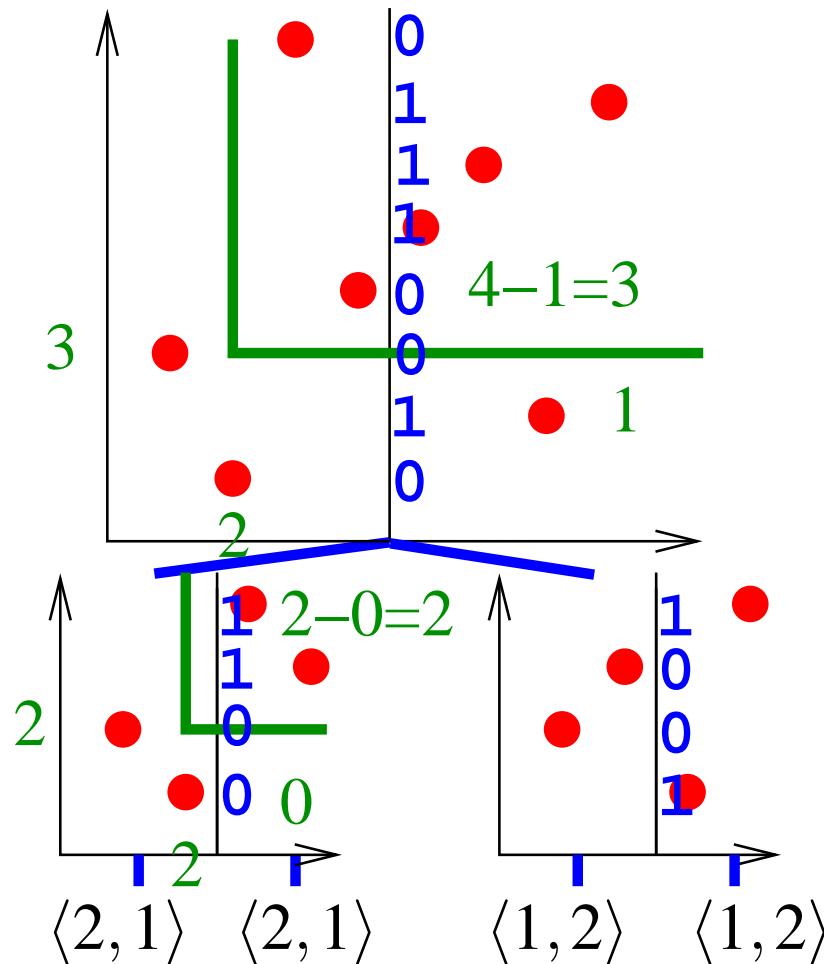
```

Function intDominanceCount( $x, y$ )           //  $|[x, n] \times [y, n] \cap P|$ 
  if  $n \leq n_0$  then return  $|[x, n] \times [y, n] \cap P|$     // brute force
   $y_r := b.rank(y)$           // Number of els  $\leq y$  in right half
  if  $x \leq \lfloor n/2 \rfloor$  then
    return  $\ell.intDominanceCount(x, y - y_r) + \lceil n/2 \rceil - y_r$ 
  else
    return  $r.intDominanceCount(x - \lfloor n/2 \rfloor, y_r)$ 

```



# Beispiel



# Analyse

Nur ein rekursiver Aufruf.

Rekursionstiefe  $O(\log n)$ .

rank in konstanter Zeit (s.u.)

Zeit  $O(\log n)$

# Wavelet Tree Dominance Reporting Query

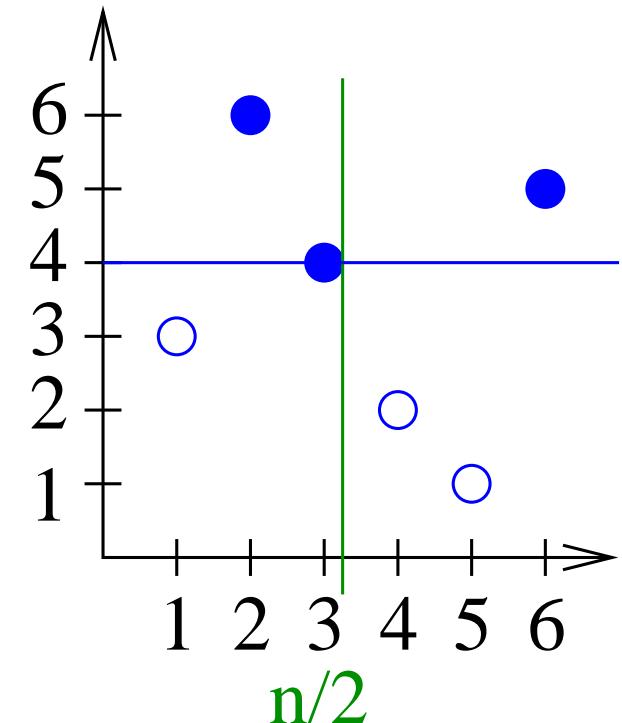
```
Function intDominanceReporting( $x, y$ )           //  $[x, n] \times [y, n] \cap P$ 
    if  $n \leq n_0$  then return  $[x, n] \times [y, n] \cap P$       // brute force
     $R := \emptyset$                                          // Result
     $y_r := b.rank(y)$                                 // Number of els  $\leq y$  in right half
    if  $x \leq \lfloor n/2 \rfloor$  then                      // Both halves interesting
        if  $y - y_r < \frac{n}{2}$  then  $R := R \cup \ell.intDominanceReporting(x, y - y_r)$ 
        if  $y_r < \frac{n}{2}$  then  $R := R \cup r.oneSidedReporting(y_r)$ 
    else if  $y_r < \frac{n}{2}$  then  $R := R \cup r.intDominanceReporting(x - \lfloor n/2 \rfloor, y_r)$ 
    return  $R$ 
```



```

Function oneSidedReporting( $y$ ) //  $[1, n] \times [y, n] \cap P$ 
  if  $n \leq n_0$  then return  $[1, n] \times [y, n] \cap P$  // brute force
   $y_r := b.rank(y)$  // Number of els  $\leq y$  in right half
   $R := \emptyset$ 
  if  $y_r < \frac{n}{2}$  then  $R := R \cup r.oneSidedReporting(y_r)$ 
  if  $y - y_r < \frac{n}{2}$  then  $R := R \cup \ell.oneSidedReporting(y - y_r)$ 
  return  $R$ 

```



# Analyse

Rekurrenz

$$T(n_0, 0) = O(1)$$

$$T(n, 0) = T(n/2, 0) + O(1) \implies T(n, 0) = O(\log n)$$

$$T(n, k) = T(n/2, k') + T(n/2, k - k') + O(1) \text{ also}$$

$$\begin{aligned} T(n, k) &\leq ck' \log n + c(k - k') \log n + c = c(1 + k \log n) = \\ &O(k \log n) \end{aligned}$$

Zeit  $O(k + \log n)$  braucht zusätzlichen Faktor  $\log n$  Platz.

Z.B. komplette Listen auf allen Ebenen speichern

Übungsaufgabe?

# Allgemeine Reporting Query

4-seitig

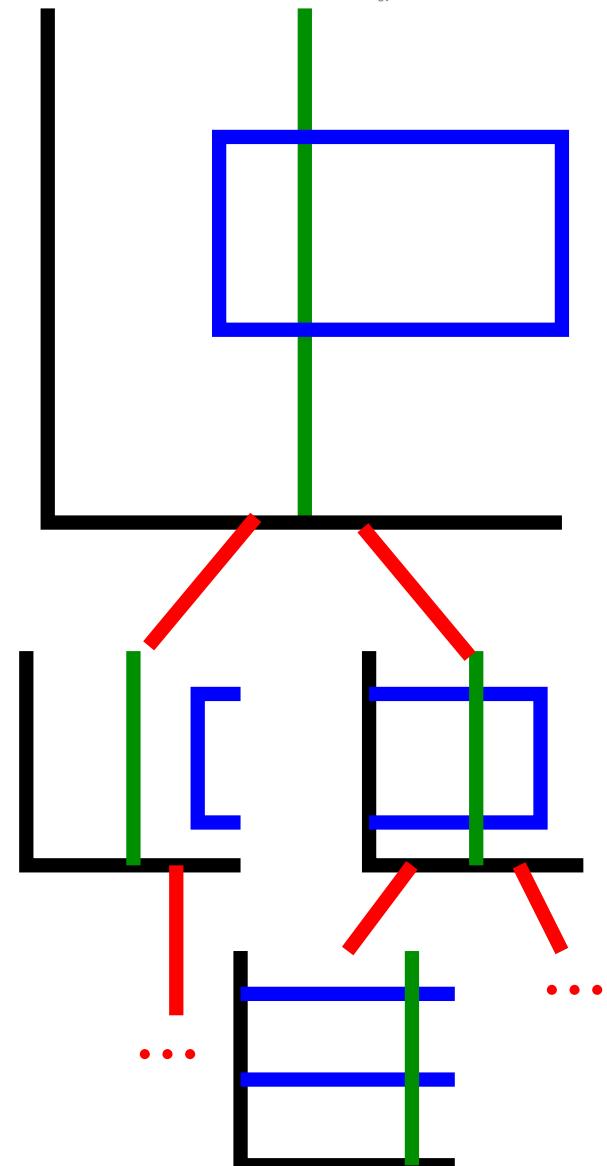
~~~

3-seitig (2 Varianten)

~~~

y-range

Analog oneSidedReporting (zwei Ranks statt einem)



# Bitvektoren $v$ mit rank in $O(1)$

Wähle  $B = \Theta(\log n)$ .

Vorberechnung  $\text{bRank}[i] := v.\text{rank}(iB)$

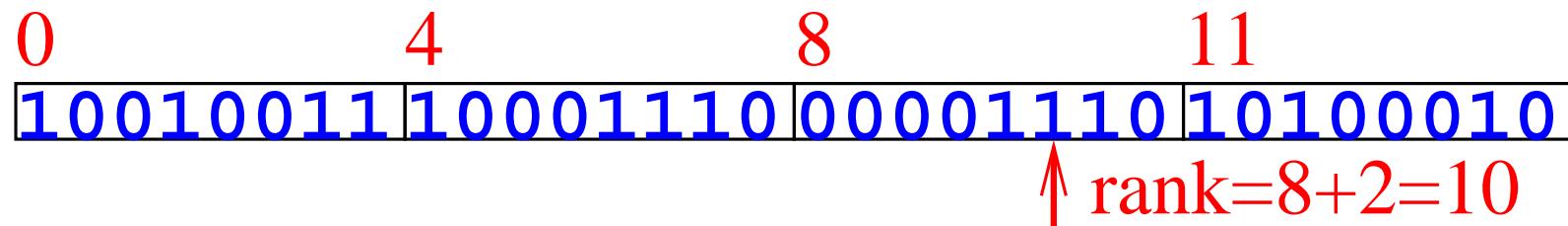
Zeit  $O(n)$ , Platz  $O(n)$  bits

Reduktion auf logarithmische Eingabegröße:

**Function**  $\text{rank}(j)$    **return**  $\text{bRank}[j \text{ div } B] + \text{rank}(v[B(j \text{ div } B)..j])$

Logarithmische Größe:

Maschinenbefehl (population count) oder Tabellenzugriff (z.B. Größe  $\sqrt{n}$  Zahlen)

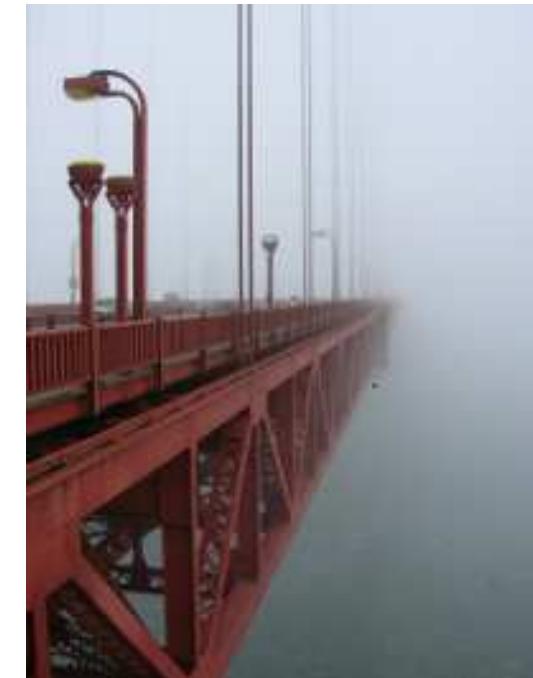


## Mehr zu Bitvektoren

- weitere wichtige Operation  $b.\text{select}(i)$ := Position des  $i$ -ten 1-bits.  
Ebenfalls  $O(1)$
- Informationstheoretisch asympt. optimaler Platz  $n + o(n)$  bits möglich.
- Grundlage für weitere succinct data structures
- Beispiel: Baum mit Platz  $2n + o(n)$  bits und Navigation in konstanter Zeit.

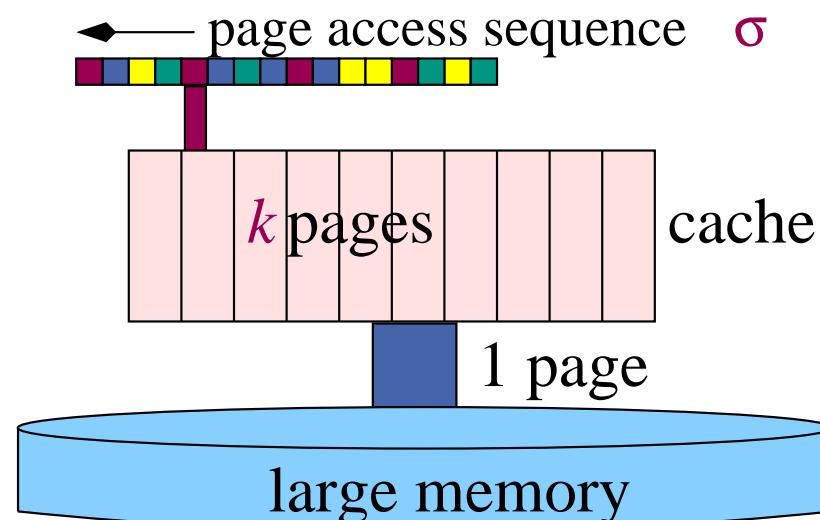
# 13 Onlinealgorithmen [z.T. von Rob van Stee]

- Information is revealed to the algorithm **in parts**
- Algorithm needs to process each part **before** receiving the next
- There is **no information** about the future  
(in particular, no probabilistic assumptions!)
- How well can an algorithm do  
compared to an algorithm that **knows everything?**
- Lack of **knowledge** vs. lack of processing power



# Examples

- Renting Skis etc.
- Paging in a virtual memory system
- Routing in communication networks
- Scheduling machines in a factory, where orders arrive over time
- Google placing advertisements



# Competitive analysis

- Idea: compare online algorithm ALG to offline algorithm OPT
- Worst-case performance measure
- Definition:

$$C_{ALG} = \sup_{\sigma} \frac{ALG(\sigma)}{OPT(\sigma)}$$

(we look for the input that results in worst **relative** performance)

- Goal:
  - find ALG with **minimal**  $C_{ALG}$

## A typical online problem: ski rental

- Renting skis costs 50 euros, buying them costs 300 euros
- You do not know in advance how often you will go skiing
- Should you rent skis or buy them?



## A typical online problem: ski rental

- Renting skis costs 50 euros, buying them costs 300 euros
- You do not know in advance how often you will go skiing
- Should you rent skis or buy them?
- Suggested algorithm: buy skis on the sixth trip
- Two questions:
  - How good is this algorithm?
  - Can you do better?



## Upper bound for ski rental

- You plan to buy skis on the sixth trip
- If you make five trips or less, you pay optimal cost  
(50 euros per trip)
- If you make at least six trips, you pay 550 euros
- In this case OPT pays at least 300 euros
- Conclusion: algorithm is  $\frac{11}{6}$ -competitive:  
it never pays more than  $\frac{11}{6}$  times the optimal cost

## Lower bound for ski rental

- Suppose you buy skis **earlier**, say on trip  $x < 6$ .

You pay  $300 + 50(x - 1)$ , OPT pays only  $50x$

$$\frac{250 + 50x}{50x} = \frac{5}{x} + 1 \geq 2.$$

- Suppose you buy skis **later**, on trip  $y > 6$ .

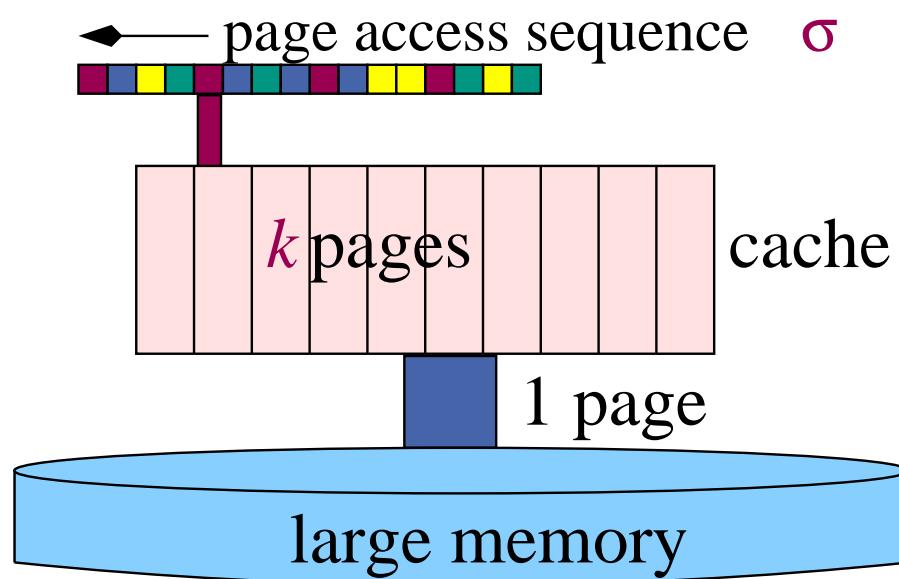
You pay  $300 + 50(y - 1)$ , OPT pays only  $300$

$$\frac{250 + 50y}{300} = \frac{5 + y}{6} \geq 2.$$

- Idea: do not pay the large cost (buy skis) until you would have paid **the same amount** in small costs (rent)

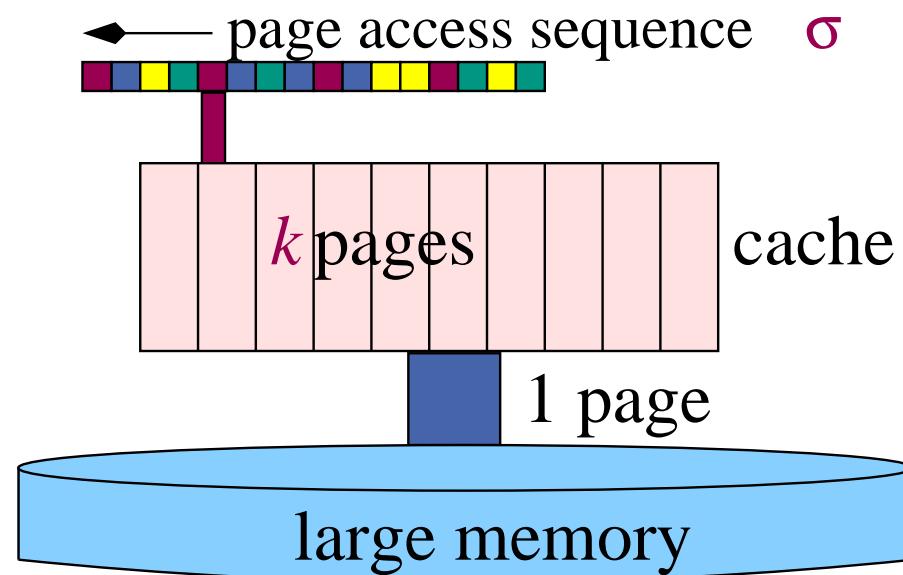
# Paging

- Computers usually have a small amount of fast memory (cache)
- This can be used to store data (pages) that are often used
- Problem when the cache is full and a new page is requested
- Which page should be thrown out (evicted)?



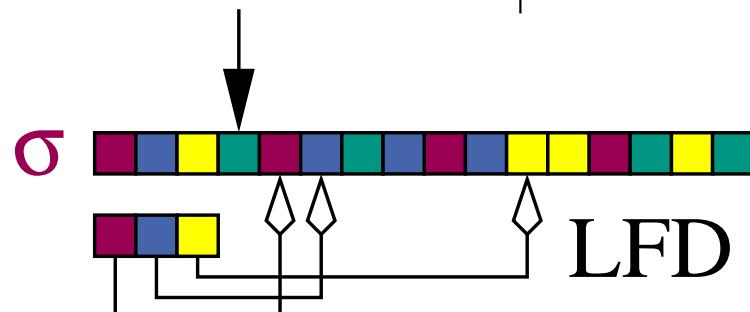
# Definitions

- $k$  = size of cache (number of pages)
- We assume that access to the cache is free, since accessing main memory costs much more
- Thus, a cache hit costs 0 and a miss (fault) costs 1
- The goal is to minimize the number of page faults



# Paging algorithms

| algorithm |                          | which page to evict                |
|-----------|--------------------------|------------------------------------|
| LIFO      | Last In First Out        | newest                             |
| FIFO      | First In First Out       | oldest                             |
| LFU       | Least Frequently used    | requested least often              |
| LRU       | Least Recently Used      | requested least recently           |
| FWF       | Flush When Full          | all                                |
| LFD       | Longest Forward Distance | (re)requested latest in the future |

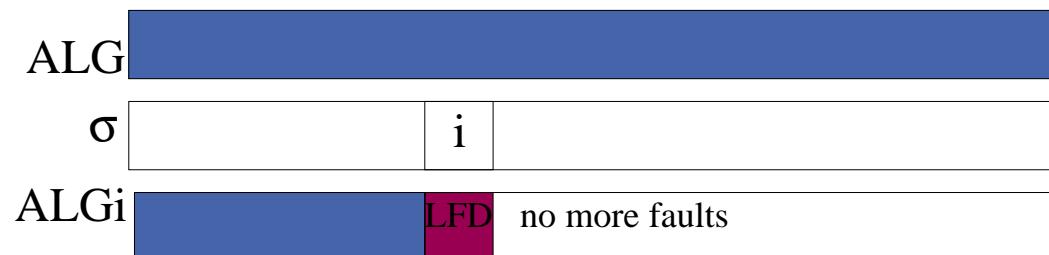


# Longest Forward Distance is optimal

We show: any optimal offline algorithm can be changed to act like LFD without increasing the number of page faults.

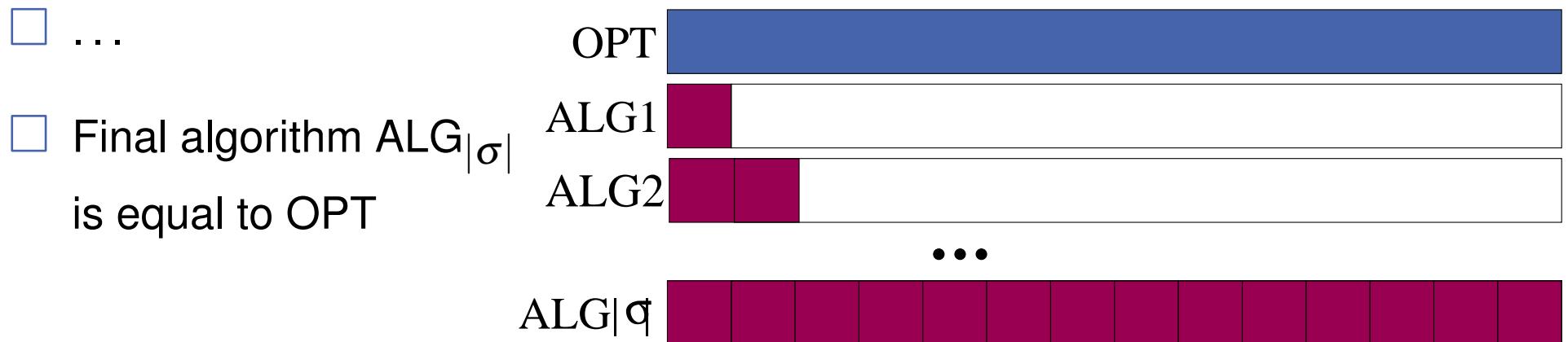
Inductive claim: given an algorithm ALG, we can create  $\text{ALG}_i$  such that

- ALG and  $\text{ALG}_i$  act identically on the first  $i - 1$  requests
- If request  $i$  causes a fault (for both algorithms),  
 $\text{ALG}_i$  evicts page with longest forward distance
- $\text{ALG}_i(\sigma) \leq \text{ALG}(\sigma)$



# Using the claim

- Start with a given request sequence  $\sigma$  and an optimal offline algorithm ALG
- Use the claim for  $i = 1$  on ALG to get  $\text{ALG}_1$ , which evicts the LFD page on the first request (if needed)
- Use the claim for  $i = 2$  on  $\text{ALG}_1$  to get  $\text{ALG}_2$



# Proof of the claim

not this time

# Comparison of algorithms

- OPT is not online, since it looks forward
- Which is the best online algorithm?
- LIFO is **not** competitive: consider an input sequence

$$p_1, p_2, \dots, p_{k-1}, \textcolor{violet}{p_k}, p_{k+1}, p_k, p_{k+1}, \dots$$

- LFU is also **not** competitive: consider

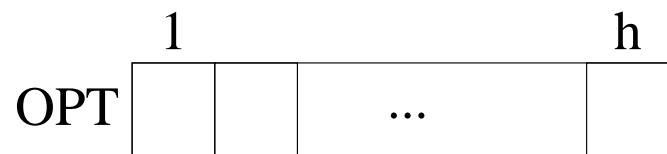
$$p_1^m, p_2^m, \dots, p_{k-1}^m, (\textcolor{violet}{p_k}, p_{k+1})^{m-1}$$

## A general lower bound

- To illustrate the problem, we show a lower bound for **any** online paging algorithm ALG
- There are  $k + 1$  pages
- At all times, ALG has  $k$  pages in its cache
- There is always one page missing: request this page at each step
- OPT only faults **once every  $k$  steps**  
⇒ **lower bound of  $k$**  on the competitive ratio

## Resource augmentation

- We will compare an online algorithm ALG to an optimal offline algorithm which has a smaller cache
- We hope to get more realistic results in this way
- Size of offline cache =  $h < k$
- This problem is known as  $(h, k)$ -paging



## Conservative algorithms

- An algorithm is **conservative** if it has at most  $k$  page faults on any request sequence that contains at most  $k$  distinct pages
- The request sequence may be **arbitrarily long**
- LRU and FIFO are conservative
- LFU and LIFO are **not** conservative (recall that they are not competitive)

# Competitive ratio

**Theorem:** Any conservative algorithm is  $\frac{k}{k-h+1}$ -competitive

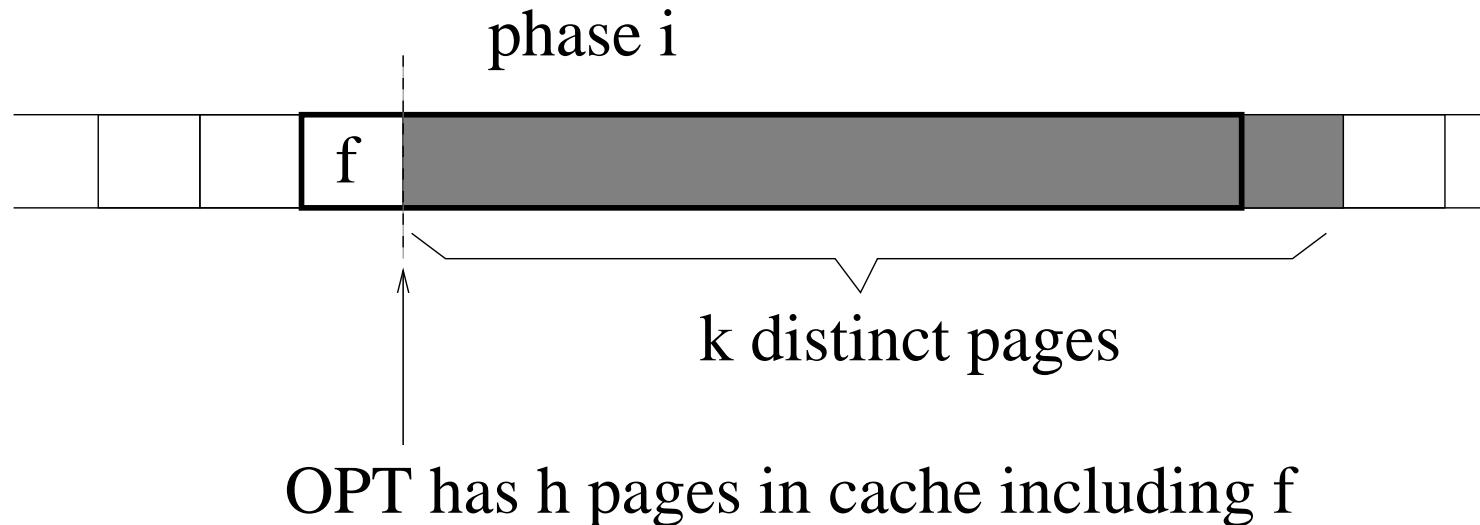
**Proof:** divide request sequence  $\sigma$  into **phases**.

- Phase 0 is the empty sequence
- Phase  $i > 0$  is the maximal sequence following phase  $i - 1$  that contains at most  $k$  distinct pages

Phase partitioning **does not depend on algorithm**. A conservative algorithm has at most  $k$  faults per phase.

## Counting the faults of OPT

Consider some phase  $i > 0$ , denote its first request by  $f$



Thus OPT has at least  $k - (h - 1) = k - h + 1$  faults on the grey requests

# Conclusion

- In each phase, a conservative algorithm has  $k$  faults
- To each phase except the last one, we can assign (charge)  
 $k - h + 1$  faults of OPT
- Thus

$$\text{ALG}(\sigma) \leq \frac{k}{k - h + 1} \cdot \text{OPT}(\sigma) + r$$

where  $r \leq k$  is the number of page faults of ALG in the last phase

- This proves the theorem

## Notes

- For  $h = k/2$ , we find that conservative algorithms are 2-competitive
- The previous lower bound construction does not work for  $h < k$
- In practice, the “competitive ratio” of LRU is a small constant
- Resource augmentation can give better (more realistic) results than pure competitive analysis

## New results (Panagiotou & Souza, STOC 2006)

- Restrict the adversary to get more “natural” input sequences
- Locality of reference**: most consecutive requests to pages have short distance
- Typical memory access patterns**: consecutive requests have either short or long distance compared to the cache size

## Randomized algorithms

- Another way to avoid the lower bound of  $k$  for paging is to use a **randomized algorithm**
- Such an algorithm is allowed to use random bits in its decision making
- Crucial is **what the adversary knows** about these random bits

## Three types of adversaries

- Oblivious**: knows only the probability distribution that ALG uses,  
determines input in advance
- Adaptive online**: knows random choices made so far, bases input  
on these choices
- Adaptive offline**: knows random choices in advance (!)

Randomization **does not help** against adaptive offline adversary

We focus on the **oblivious** adversary

## Marking Algorithm

- marks pages which are requested
- never evicts a marked page
- When **all** pages are marked **and there is a fault**, unmark everything  
(but mark the page which caused the fault)  
(new **phase**)

## Marking Algorithms

Only difference is eviction strategy

- LRU
- FWF
- RMARK: Evict an unmarked page chosen uniformly at random

# Competitive ratio of RMARK

**Theorem:** RMARK is  $2H_k$ -competitive

where

$$H_k = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k} \leq \ln k + 1$$

is the  $k$ -the harmonic number



# Analysis of RMARK

Consider a phase with  $m$  new pages

(that are not cached in the beginning of the phase)

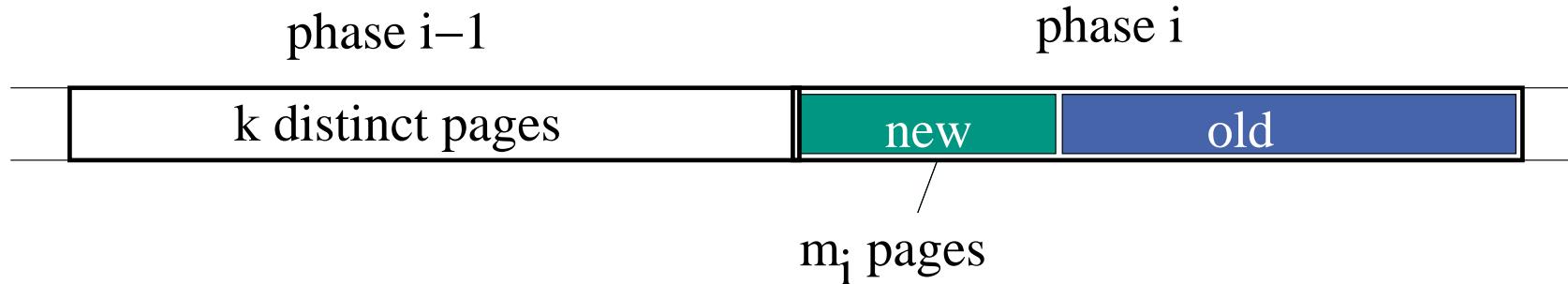
Miss probability when  $j + 1$ st old page becomes marked

$$1 - \frac{\# \text{ old unmarked cached pages}}{\# \text{ old unmarked pages}} \leq 1 - \frac{k - m - j}{k - j} = \frac{m}{k - j}$$

Overall expected number of faults (including new pages):

$$m + \sum_{j=0}^{k-m-1} \frac{m}{k-j} = m + m \sum_{i=m+1}^k \frac{1}{i} = m(1 + H_k - H_m) \leq mH_k$$

# Lower bound for OPT



- There are  $m_i$  new pages in phase  $i$
- Thus, in phases  $i-1$  and  $i$  together,  $k + m_i$  pages are requested
- OPT makes at least  $m_i$  faults in phases  $i$  and  $i-1$  **for any  $i$**
- Total number of OPT faults is at least  $\frac{1}{2} \sum_i m_i$

# Upper bound for RMARK

- Expected number of faults in phase  $i$  is at most  $m_i H_k$  for RMARK
- Total expected number of faults is at most  $H_k \sum_i m_i$
- OPT has at least  $\frac{1}{2} \sum_i m_i$  faults
- Conclusion: RMARK is  $2H_k$ -competitive

## Randomized lower bound

**Theorem:** No randomized can be better than  $H_k$ -competitive against an oblivious adversary.

**Proof:** not here

## Discussion

- $H_k \ll k$
- The upper bound for RMARK holds against an oblivious adversary  
(the input sequence is **fixed in advance**)
- No algorithm can be better than  $H_k$ -competitive
- Thus, RMARK is optimal apart from a factor of 2
- There is a (more complicated) algorithm that is  $H_k$  competitive

## Why competitive analysis?

There are many models for “decision making in the absence of complete information”

- Competitive analysis leads to algorithms that would not otherwise be considered
- Probability distributions are rarely known precisely
- Assumptions about distributions must often be unrealistically crude to allow for mathematical tractability
- Competitive analysis gives a **guarantee** on the performance of an algorithm, which is essential in e.g. financial planning

# Disadvantages of competitive analysis

- Results can be too pessimistic (adversary is too powerful)
  - Resource augmentation
  - Randomization
  - Restrictions on the input
- Unable to distinguish between some algorithms that perform differently in practice
  - Paging: LRU and FIFO
  - more refined models