Algorithmen II

Peter Sanders, Johannes Fischer (Strings)

Übungen:

Dennis Luxen und Johannes Singler

Skript:

Robert Geisberger, Moritz Kobitzsch, Vitaly Osipov, Christian Schulz, N.N.

Institut für theoretische Informatik, Algorithmik II

Web:

`http://algo2.iti.kit.edu/AlgorithmenII.php`

Organisatorisches

Vorlesungen:

Di 15:45–17:15 HS Neue Chemie

Do 15:45–16:30 Gerthsen-HS

Saalübung:

Do 16:30–17:15 Gerthsen-HS

Übungsblätter: 14-täglich, jeweils Dienstag, Musterlösung 9 Tage später vor Vorlesung/Übung. 1. Blatt: 26.10.2010

Organisatorisches

Sprechstunde:

- Peter Sanders, Dienstag 13:45–14:45 Uhr, Raum 217

Letzte Vorlesung: 10. Februar 2011

Klausur: 3. März 2011, 11:00 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

- [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

- . . . wird noch ergänzt ?

Inhaltsübersicht I

- Algorithm Engineering
- Fortgeschrittene Datenstrukturen
  - (Hashing II $\rightsquigarrow$ unter randomisierte Algorithmen)
  - Effiziente addressierbare Prioritätslisten
  - (Monotone Prioritätslisten) $\rightsquigarrow$ unter kürzeste Wege
  - (Externe Prioritätslisten) $\rightsquigarrow$ unter externe Algorithmen
  - (“succinct” data structures) $\rightsquigarrow$ unter Geometrie.
- Fortgeschrittene Graphenalgorithmen
  - Starke Zusammenhangskomponenten
  - Kürzeste Wege II: negative Kreise, Potentialmethode
  - Maximale Flüsse und Matchings

Inhaltsübersicht II

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

1 Algorithm Engineering

A detailed definition

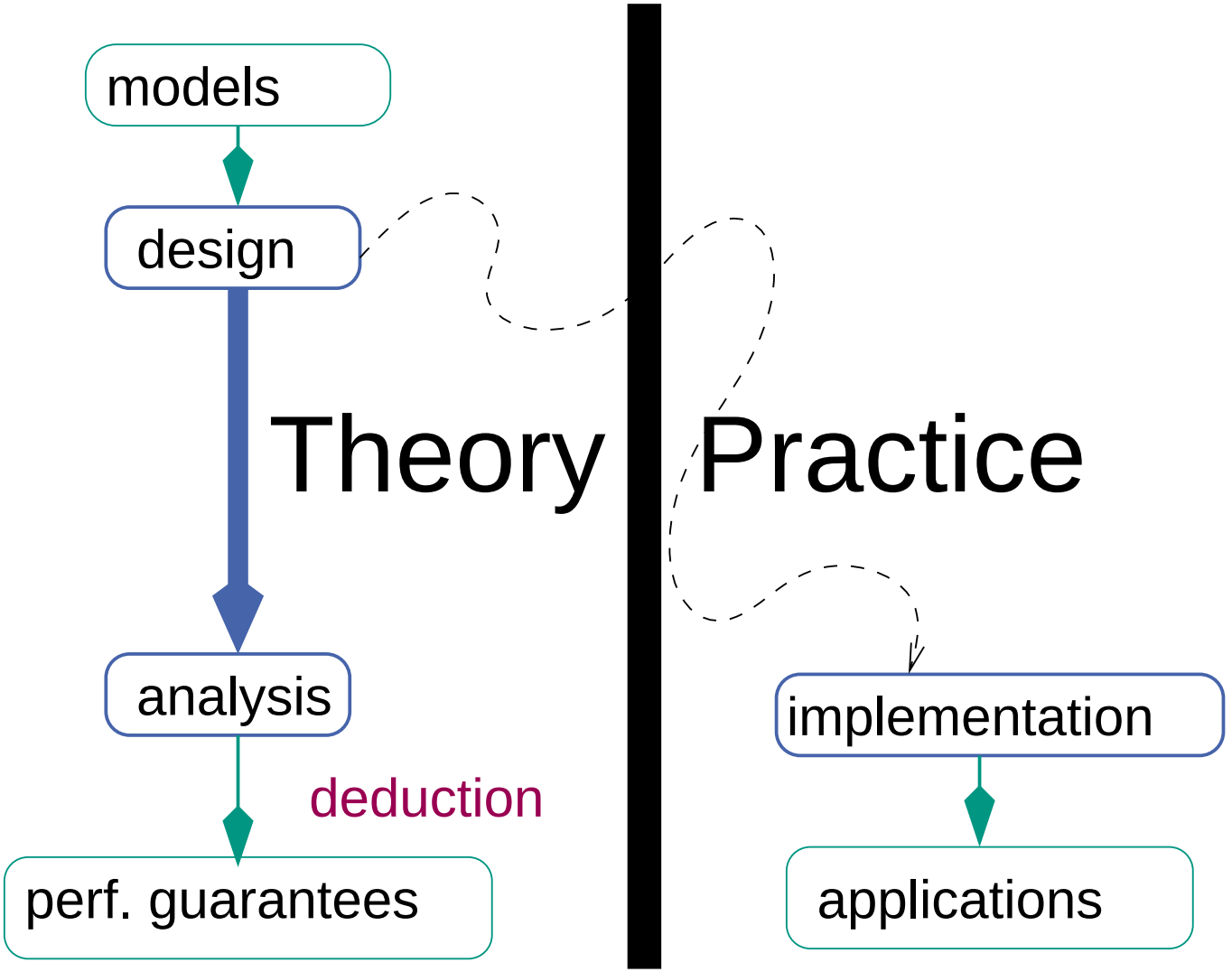
- in general

  [with Kurt Mehlhorn, Rolf Möhring, Petra Mutzel, Dorothea Wagner]

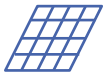
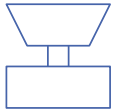
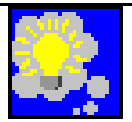
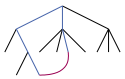
- A few examples, usually sorting

- A little bit on experimental methodology

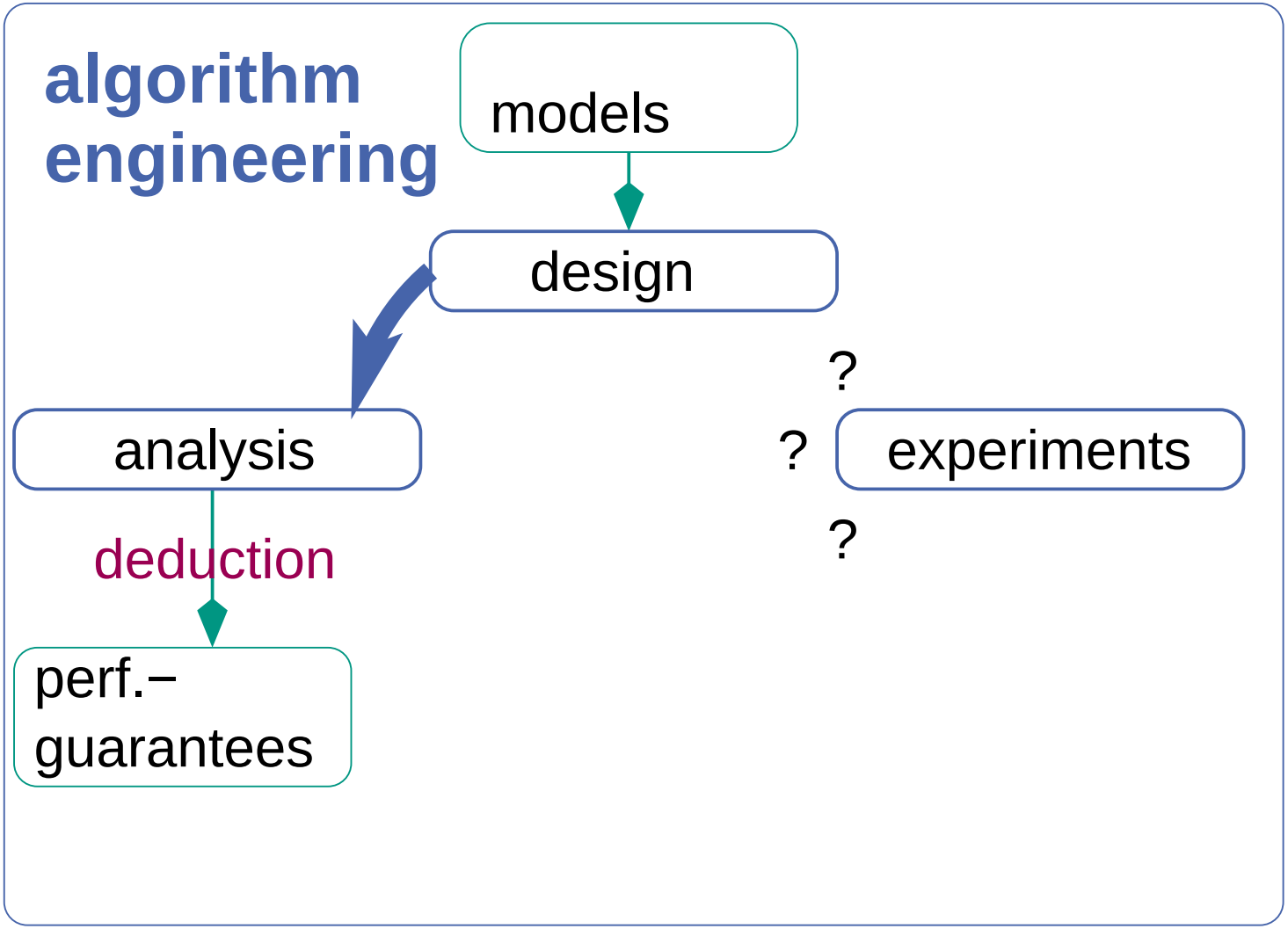
(Caricatured) Traditional View: Algorithm Theory



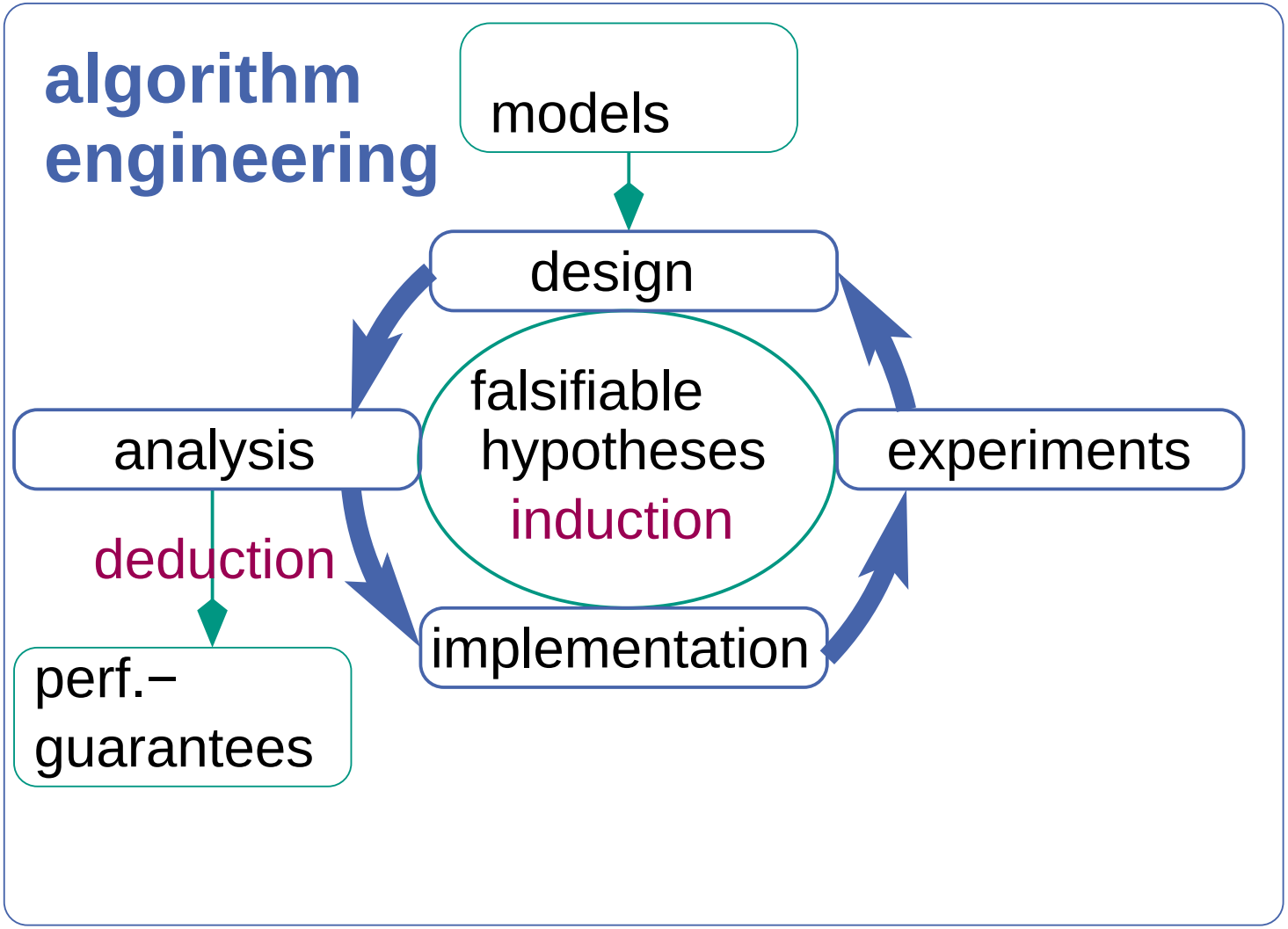
Gaps Between Theory & Practice

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

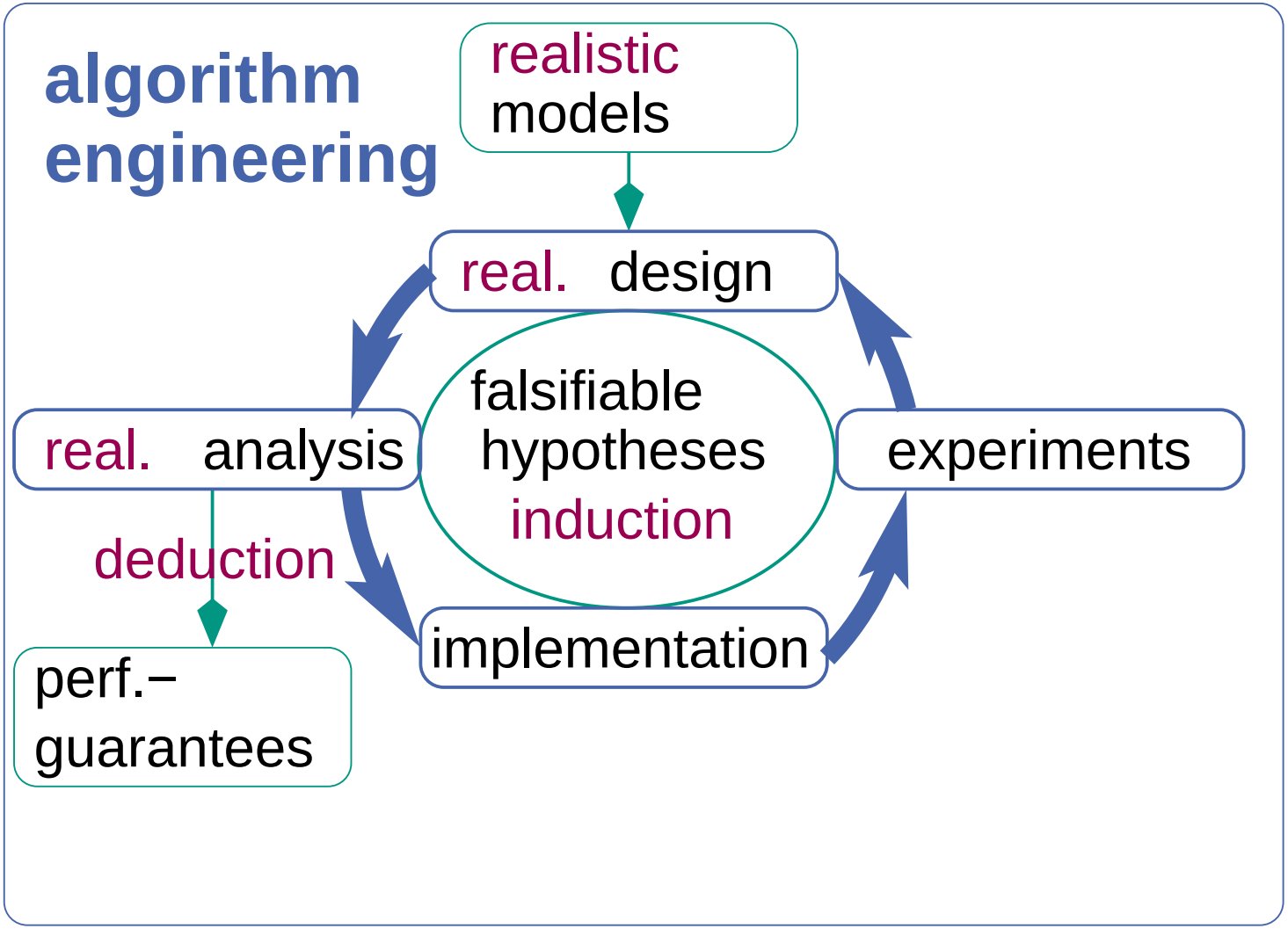
Algorithmics as Algorithm Engineering



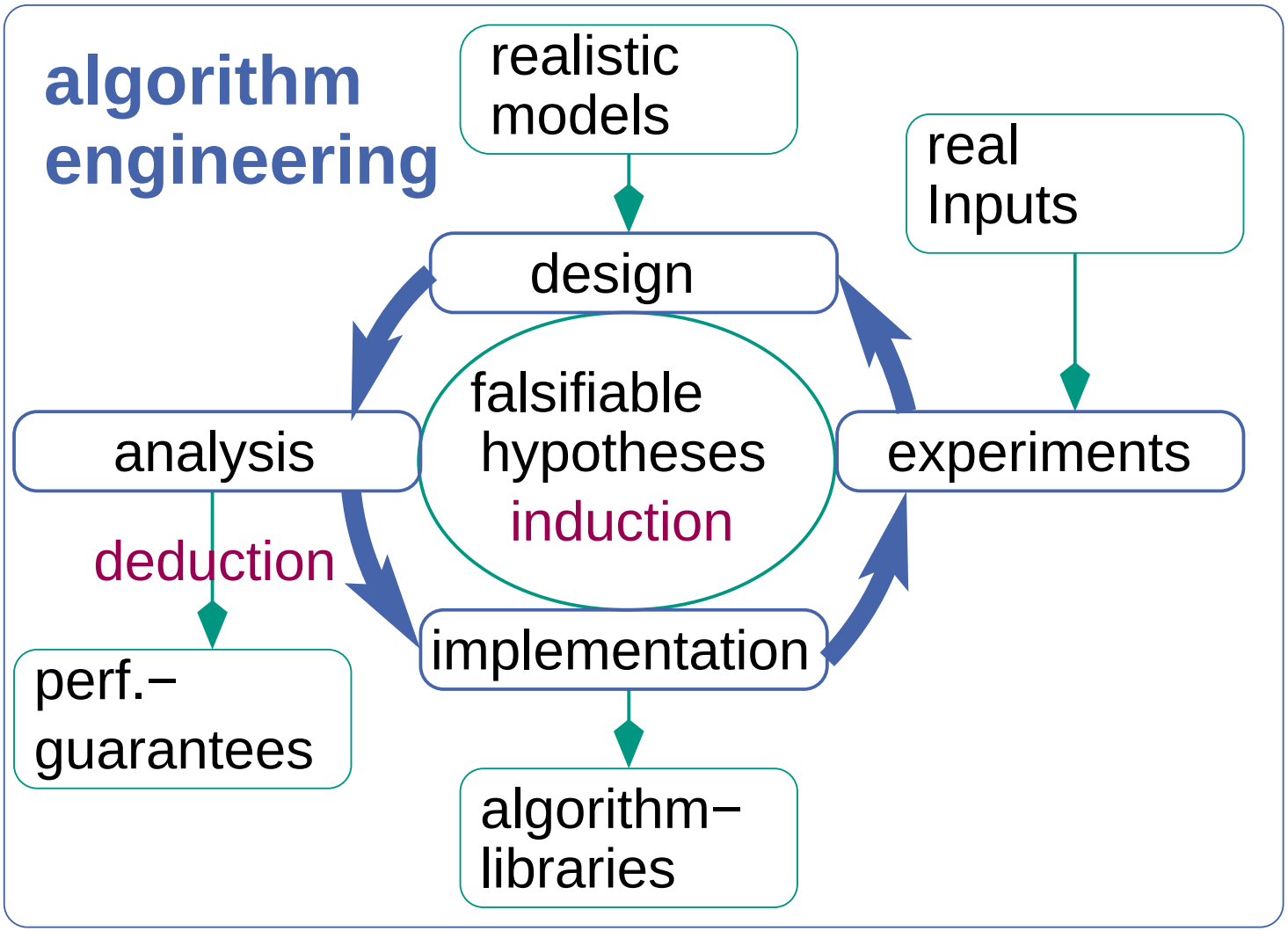
Algorithmics as Algorithm Engineering



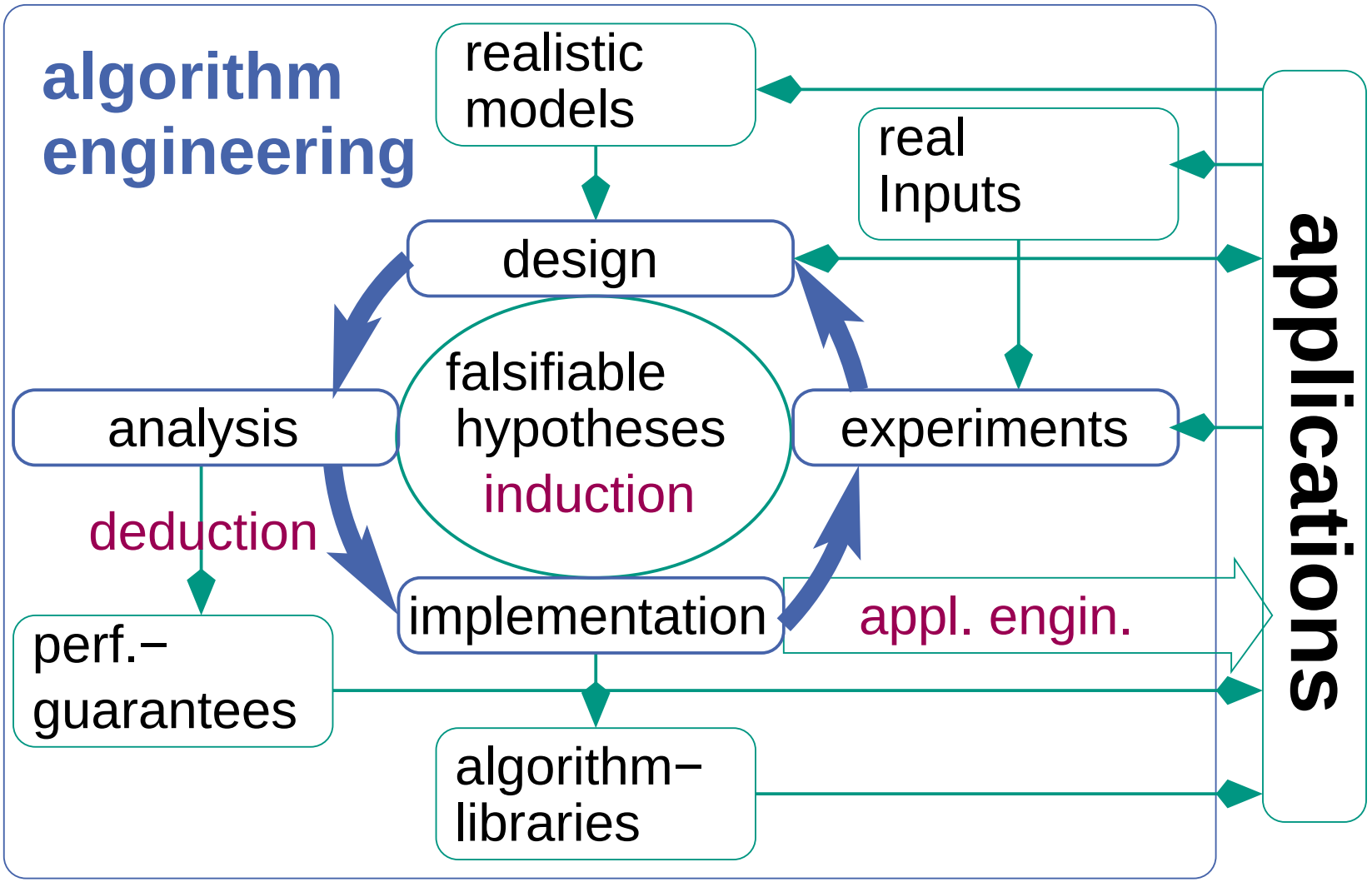
Algorithmics as Algorithm Engineering



Algorithmics as Algorithm Engineering



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]

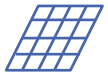
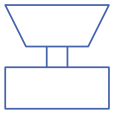
1997– Workshop on Algorithm Engineering

$\rightsquigarrow$ ESA applied track [G. Italiano]

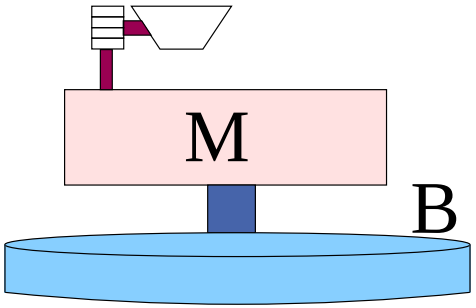
1997 Term used in US policy paper [Aho, Johnson, Karp, et. al]

1998 Alex workshop in Italy $\rightsquigarrow$ ALENEX

Realistic Models

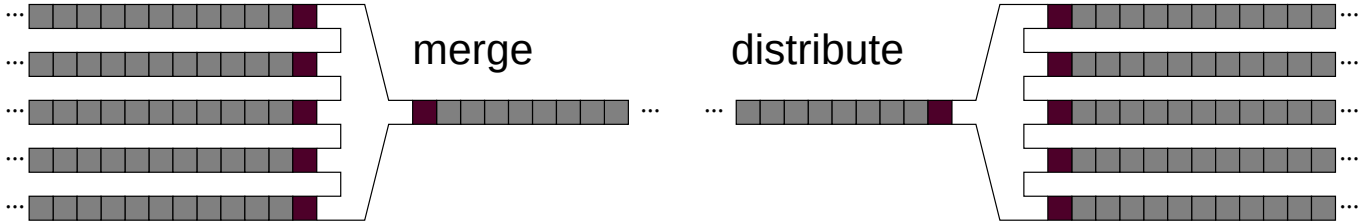
Theory	$\longleftrightarrow$	**Practice**
simple 	**appl. model**	 complex
simple 	**machine model**	 real

- 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:

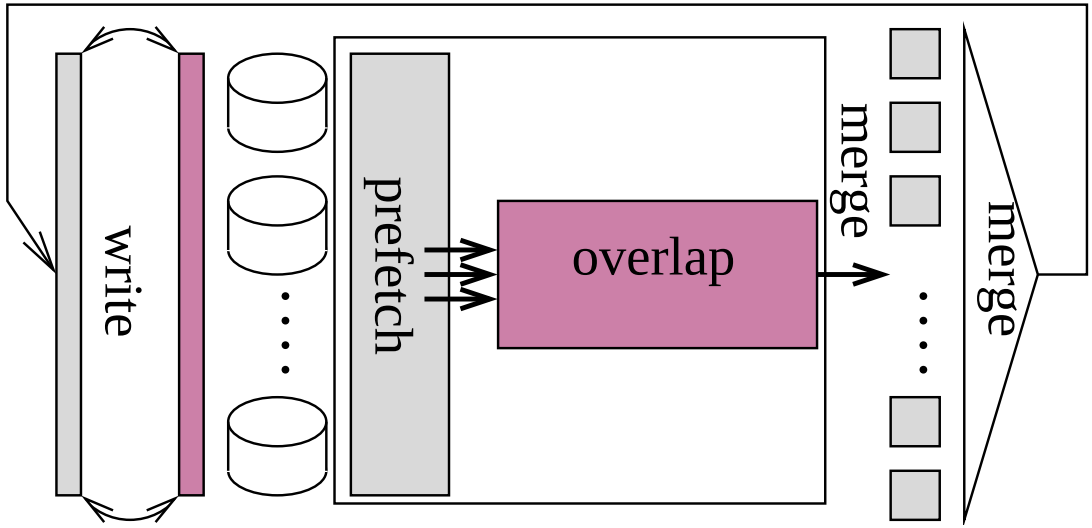
Abstract algorithm

$\leftrightarrow$

C++...

$\leftrightarrow$

hardware



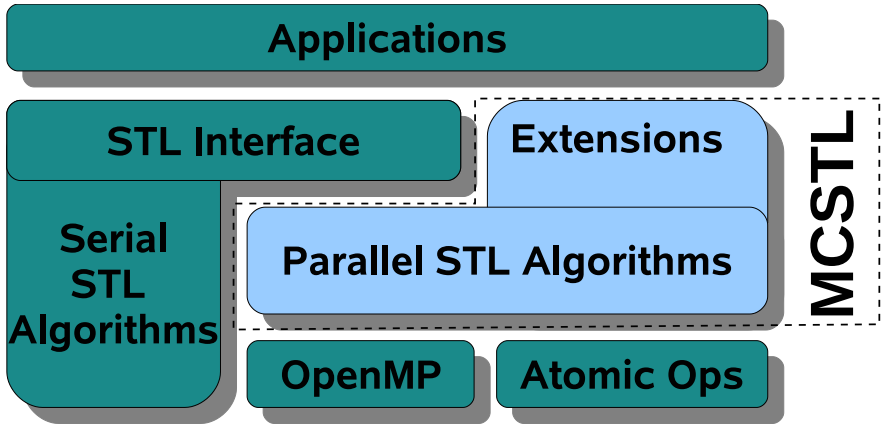
Experiments

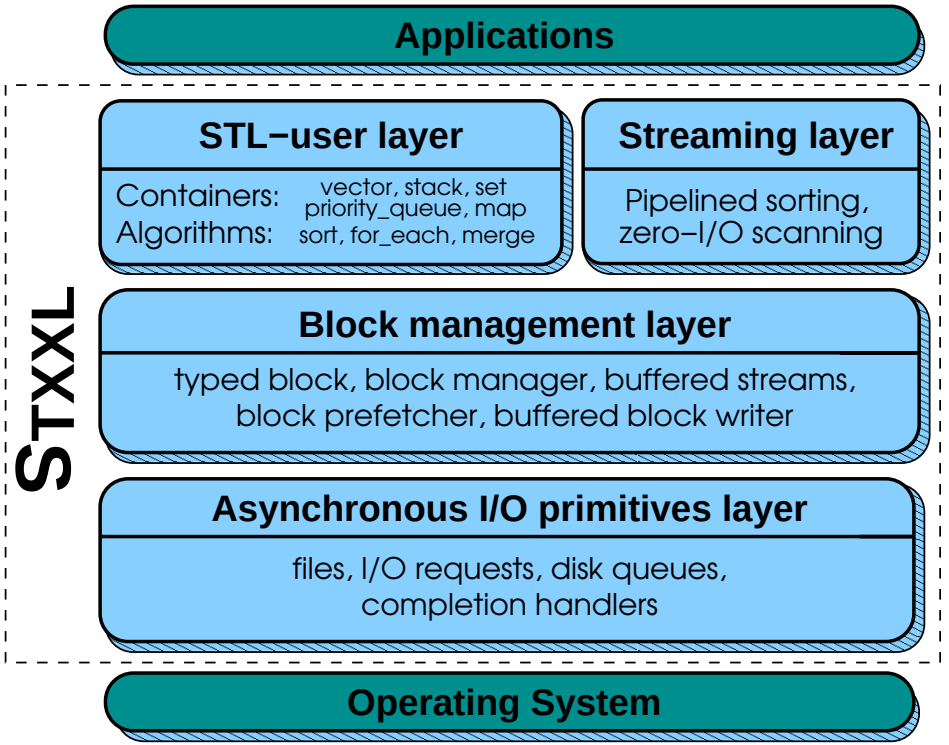
- sometimes a good surrogate for analysis
- too much rather than too little output data
- reproducibility (10 years!)
- software engineering

Stay tuned.

Algorithm Libraries — Challenges

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

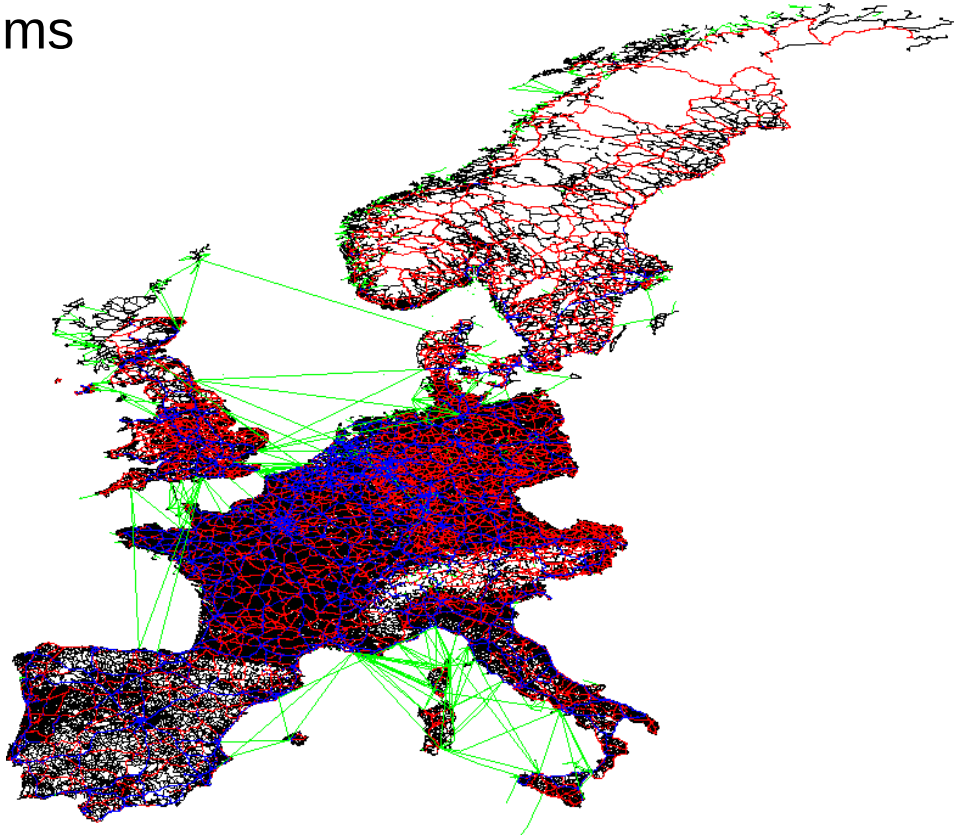




Problem Instances

Benchmark instances for NP-hard problems

- TSP
- Steiner-Tree
- SAT
- set covering
- graph partitioning
- ...



have proved essential for development of practical algorithms

Strange: much less real world instances for polynomial problems (MST, shortest path, max flow, matching...)

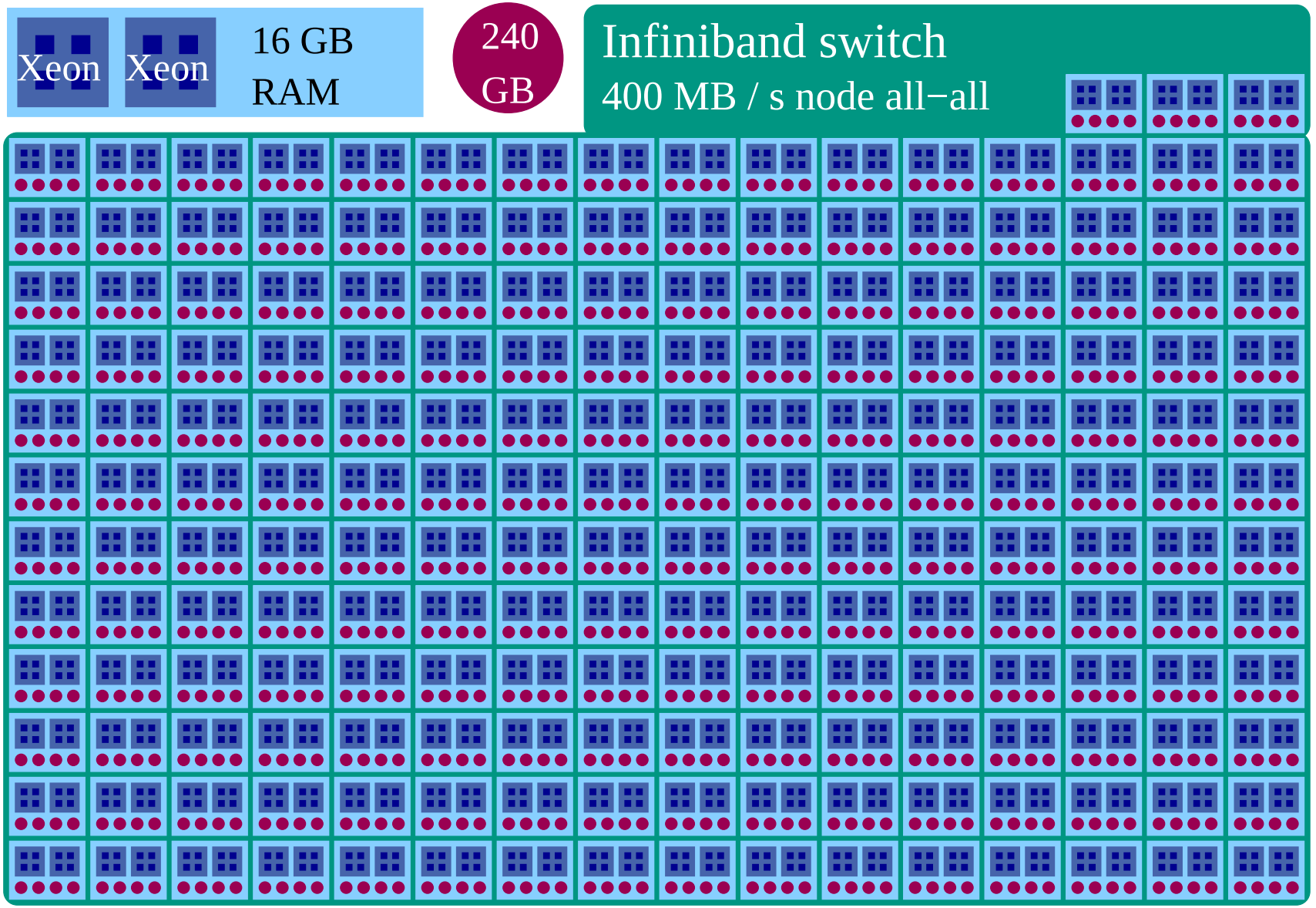
Example: Sorting Benchmark (Indy)

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

Category	data volume	performance	improvement
GraySort	100 TB	564 GB / min	$17\times$
MinuteSort	955 GB	955 GB / min	$>10\times$
JouleSort	1 000 GB	13 400 Recs/Joule	$4\times$
JouleSort	100 GB	35 500 Recs/Joule	$3\times$
JouleSort	10 GB	34 300 Recs/Joule	$3\times$

Also: PennySort

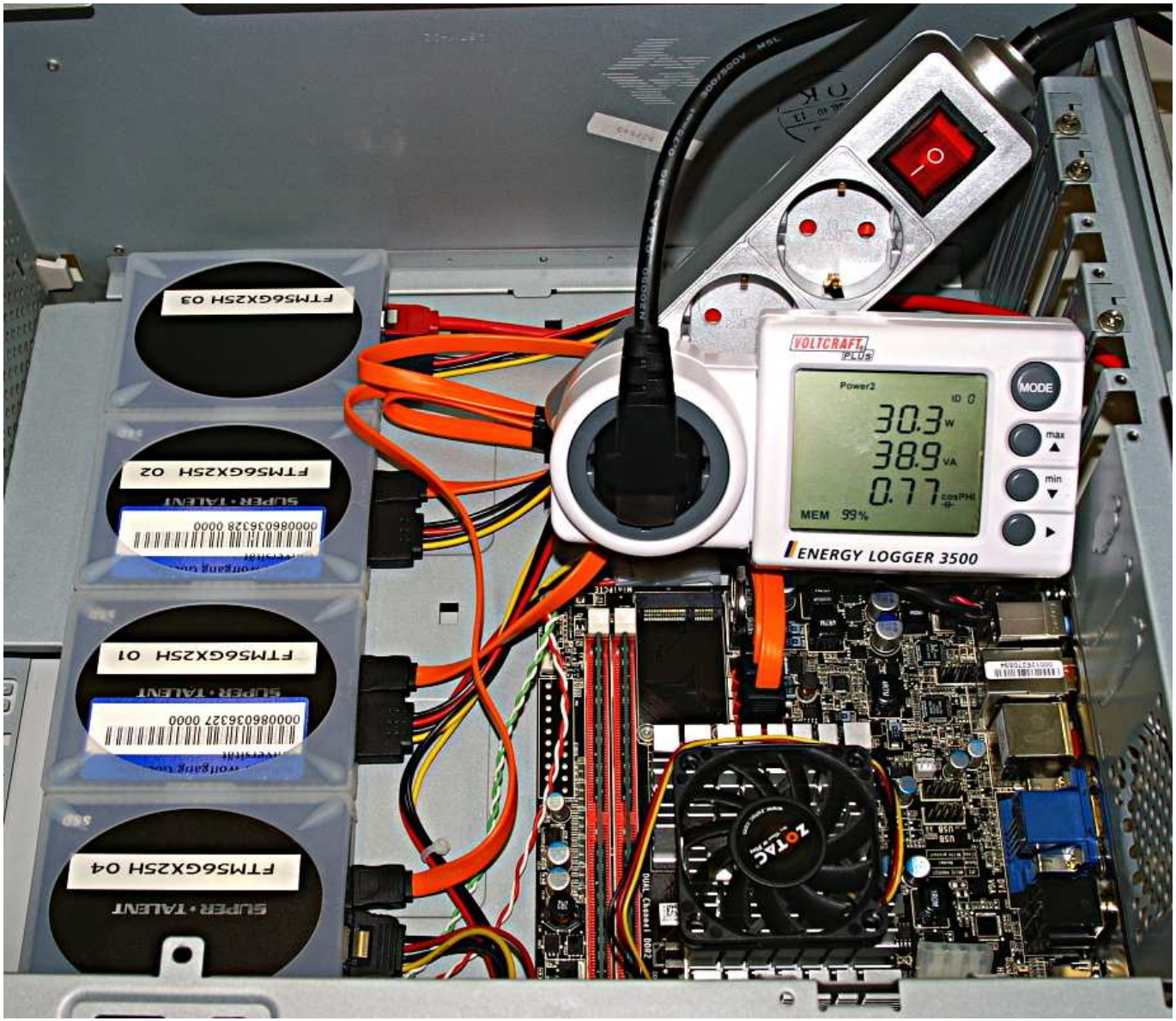
GraySort: inplace multiway mergesort, exact splitting



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, grid, 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

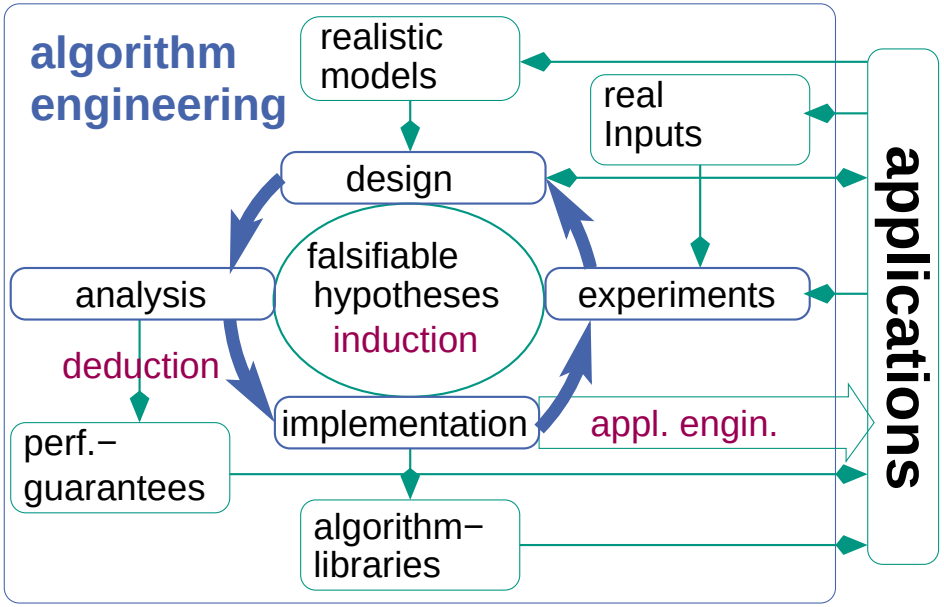
More On Experimental Methodology

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 Criteris

- 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 inputs as possible
  - its fine if you are better just on some (important) inputs

Not Here but Important

- describing the setup
- 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. $\leadsto$ 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

$\rightsquigarrow$ new input instances

2 Fortgeschrittene Datenstrukturen

2.1 Adressierbare Prioritätslisten

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

Function size **return** $|M|$

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

Function min **return** $\min M$

Function deleteMin $e := \min M$; $M := M \setminus \{e\}$; **return** e

Function remove$(h : \text{Handle})$ $e := h$; $M := M \setminus \{e\}$; **return** e

Procedure decreaseKey$(h : \text{Handle}, k : \text{Key})$ **assert** key$(h) \geq k$; key$(h) := k$

Procedure merge(M') $M := M \cup M'$

Adressierbare Prioritätslisten: Anwendungen

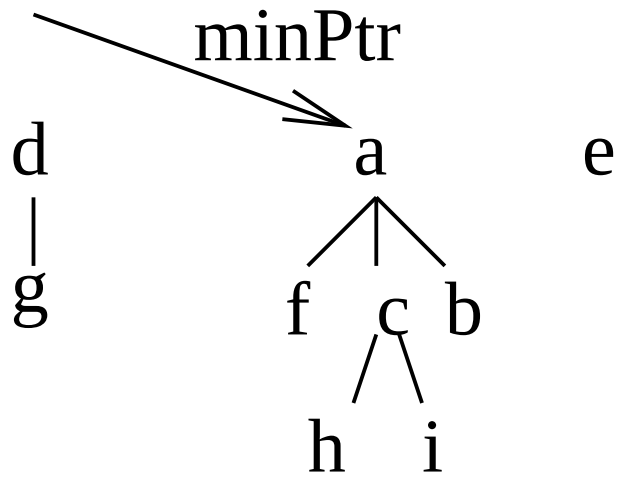
- Dijkstras Algorithmus für kürzeste Wege
- Jarník-Prim-Algorithmus für minimale Spannbäume
- Bei uns: Hierarchiekonstruktion für Routenplanung
- Bei uns: Graphpartitionierung
- Bei uns: disk scheduling

Allgemein:

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

Grundlegende Datenstruktur

Ein Wald heap-geordneter Bäume

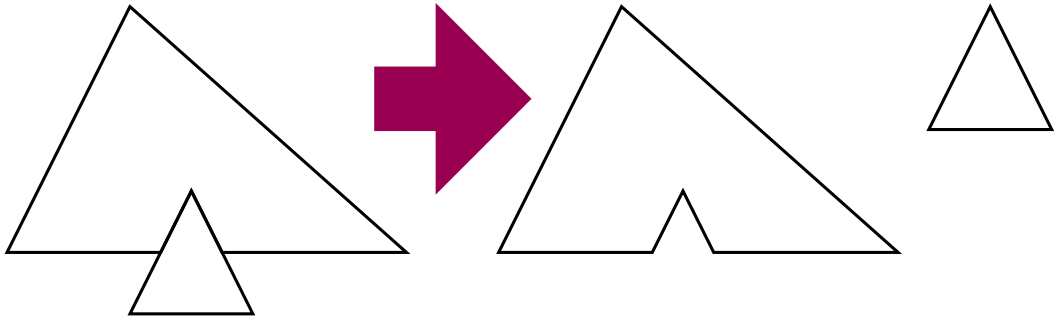


Verallgemeinerung gegenüber binary heap:

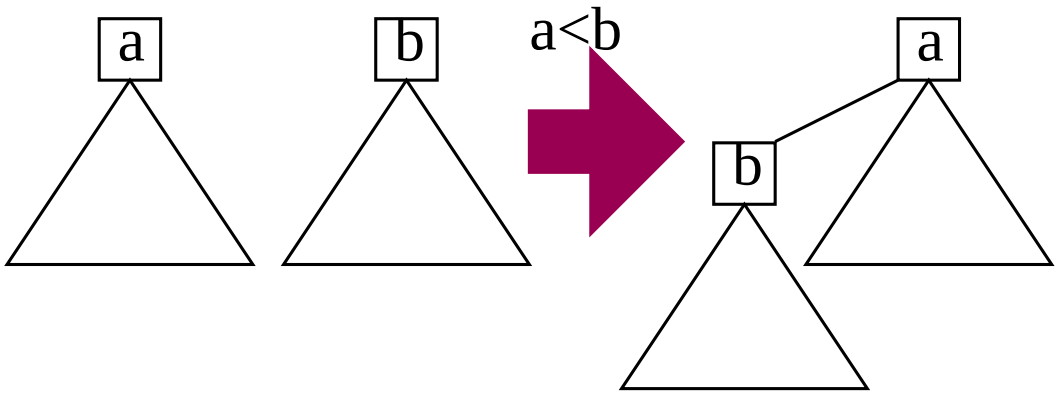
- Baum $\rightarrow$ Wald
- binär $\rightarrow$ beliebige Knotengrade

Wälder Bearbeiten

Cut:



Link:



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

Pairing Heaps (Paarungs-Haufen??)

[Fredman Sedgewick Sleator Tarjan 1986]

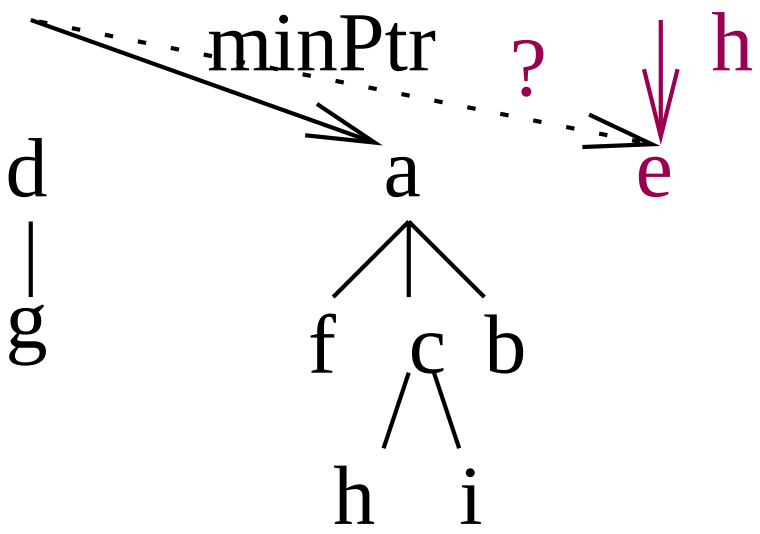
Procedure insertItem$(h$: Handle$)$

newTree(h)

Procedure newTree$(h$: Handle$)$

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

if $*h <$ min **then** minPtr:= h

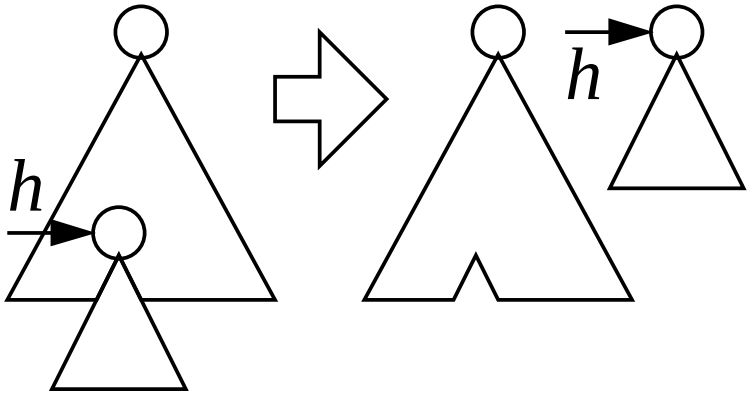


Pairing Heaps

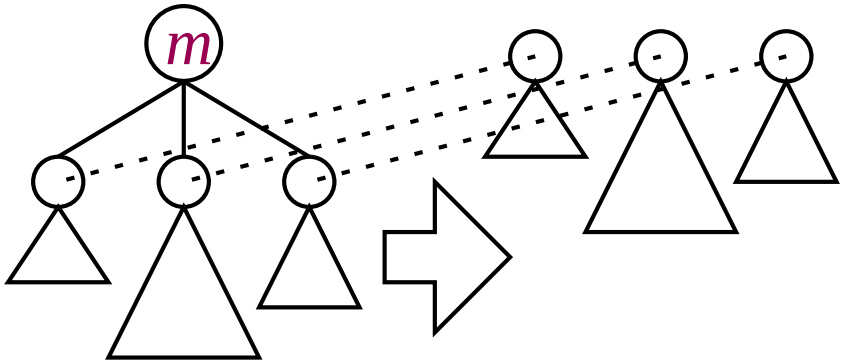
Procedure decreaseKey$(h$: Handle$, k$: Key$)$

key$(h) := k$

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



Pairing Heaps



Function deleteMin : Handle

m:= minPtr

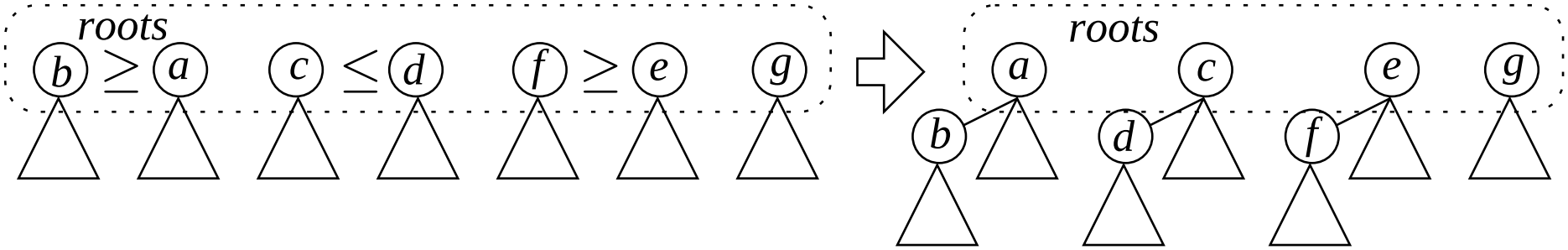
forest:= forest $\setminus \{m\}$

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

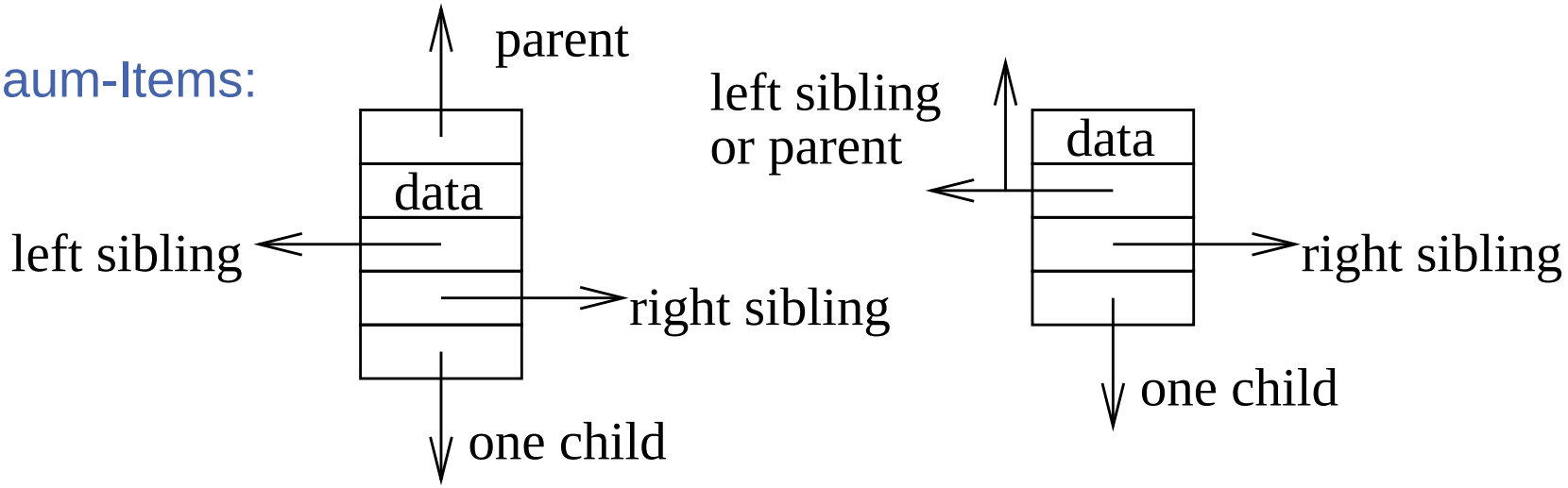
forest:= forest $\cup$ o.forest

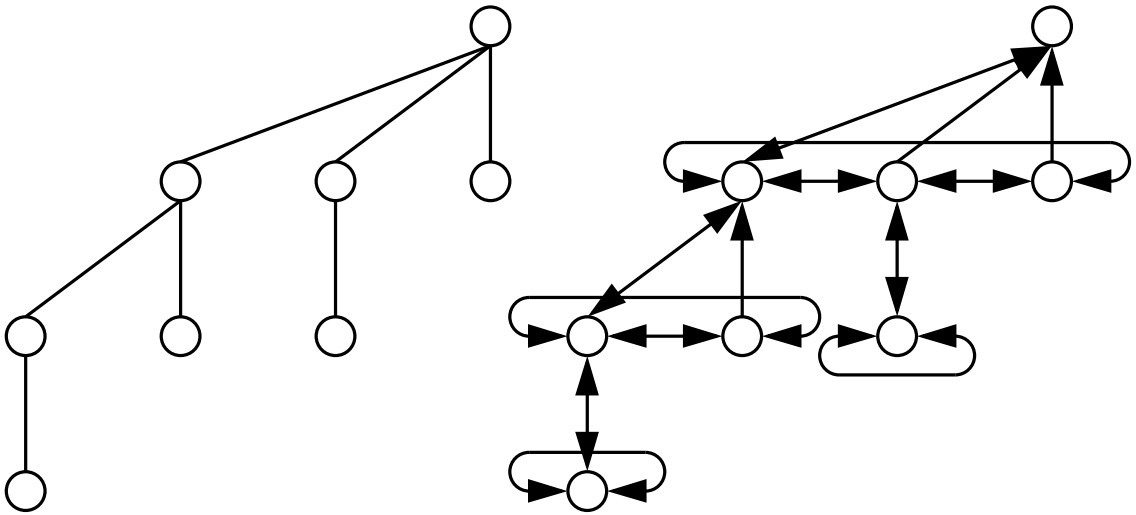
o.forest:= $\emptyset$

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 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 $O(\log n)$ for deleteMin und $O(1)$ für alle anderen Operationen

(d.h. $Gesamtzeit = O(o + d \log n)$ falls

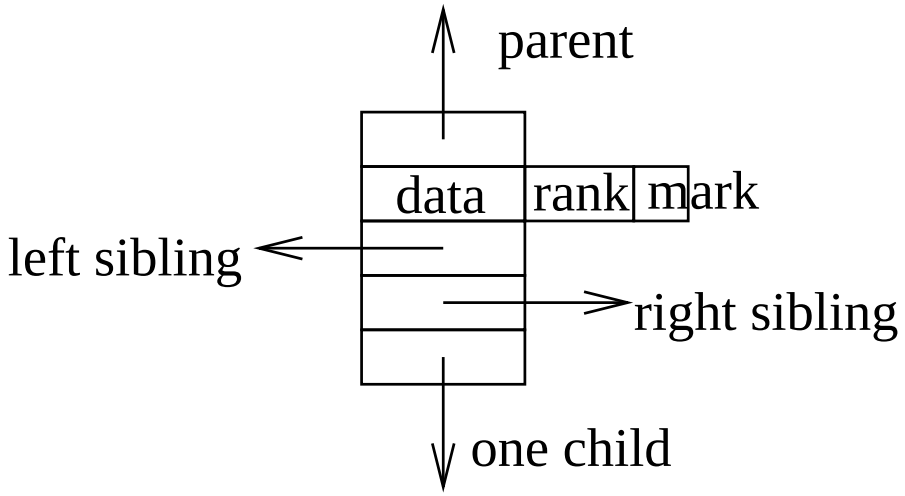
$d =$#deleteMin, $o =$#otherOps, $n = \max |M|$)

Repräsentation

Wurzeln: Doppelt verkettete Liste

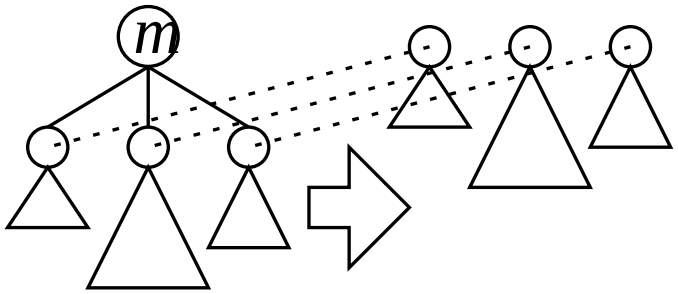
(und ein tempräres Feld für deleteMin)

Baum-Items:



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

deleteMin mit Union-by-Rank



Function deleteMin : Handle

m:= minPtr

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

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

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

union(a, b) // increments rank of surviving root

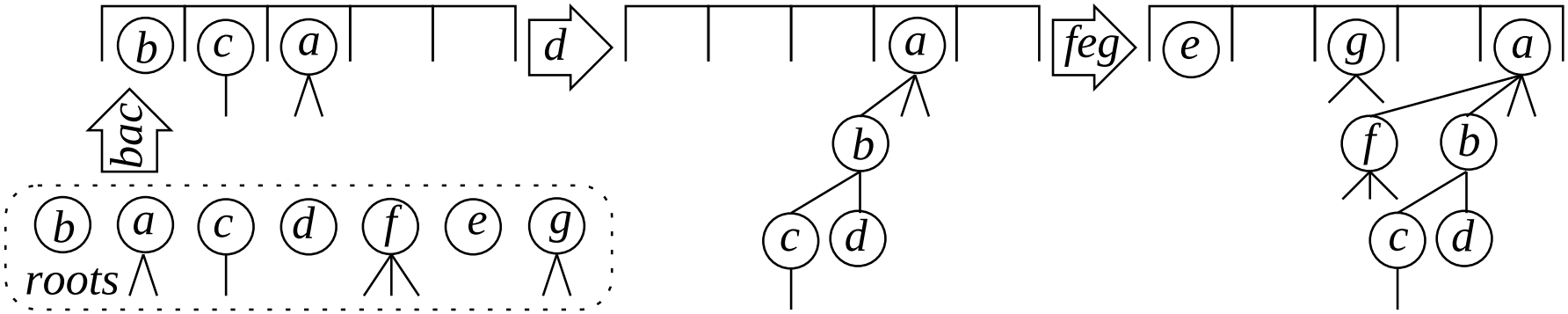
update minPtr

return m

Schnelles Union-by-Rank

Durch rank adressiertes Feld.

Solange link durchführen bis freier Eintrag gefunden.



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

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

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

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

Union-by-rank: Token zahlen für

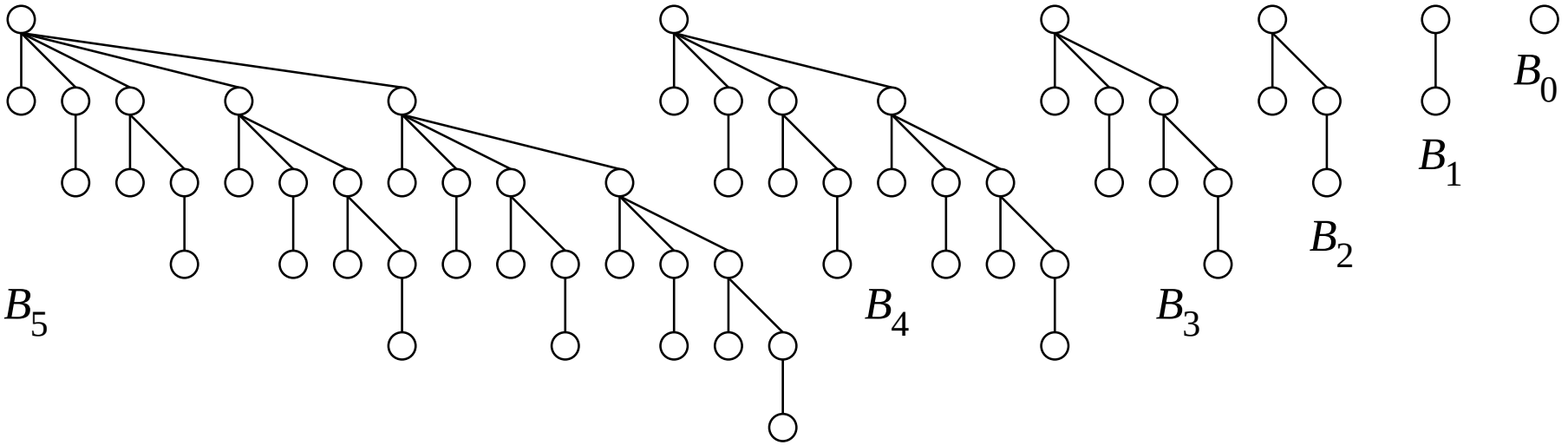
- union Operationen (ein Token wird frei) und
- durchlaufen alter und neuer Wurzeln.

Am Ende gibt es $\leq$ maxRank Wurzeln,

die wir mit neuen Token ausstatten.

Warum ist maxRank logarithmisch? – Binomialbäume

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



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

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)

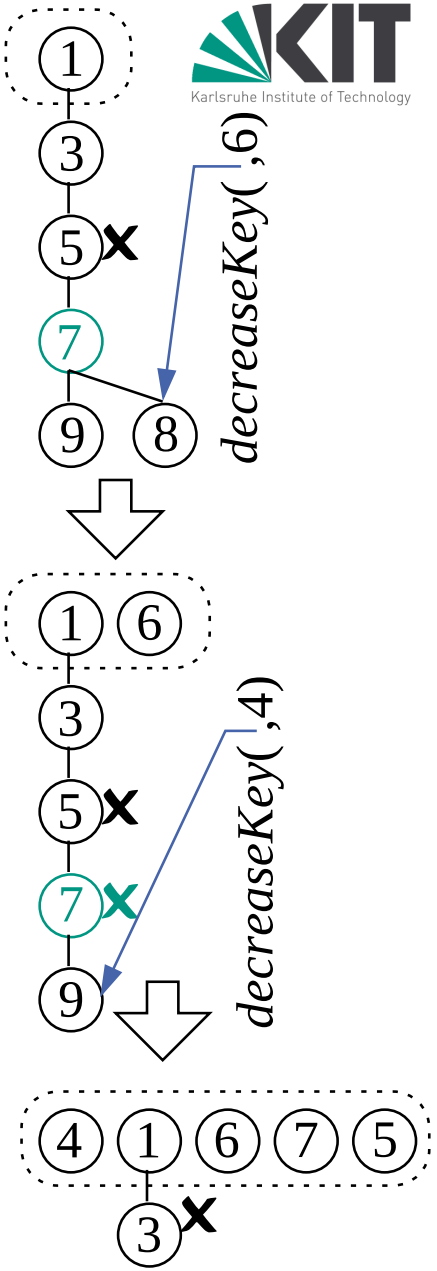
unmark h

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.

$\Rightarrow$

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:

$S_0 = 1$

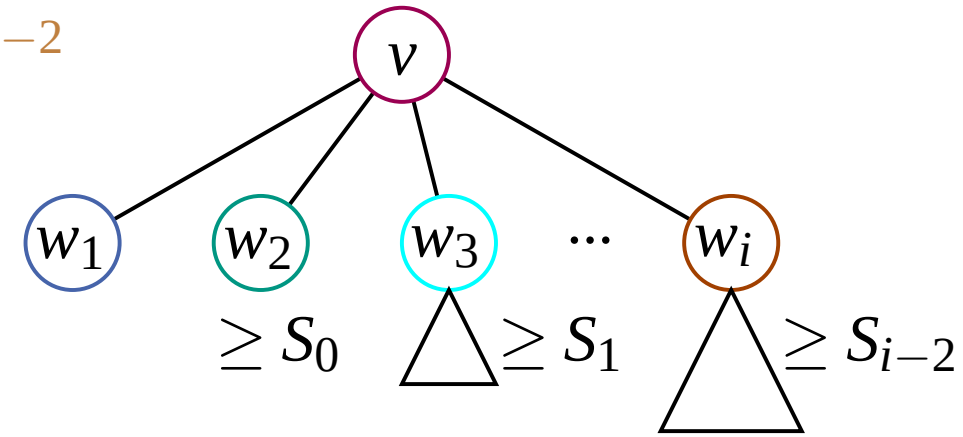
$S_1 = 2$

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

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

Fortgeschrittene Graphenalgorithmen

3 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$(u, v : \text{NodeId})$ // 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:	$\text{dfsPos}=1 : 1..n$
root(s):	$\text{dfsNum}[s]:= \text{dfsPos}++$
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$++$

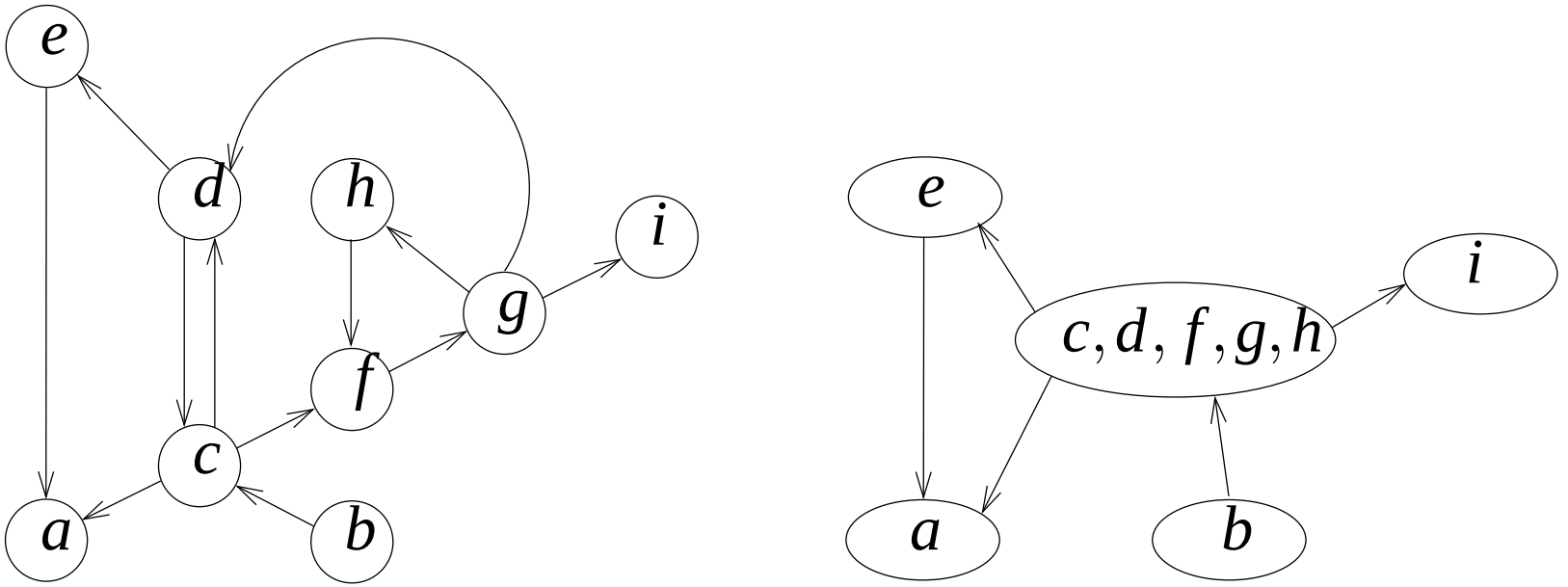
Starke Zusammenhangskomponenten

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

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

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

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



Starke Zusammenhangskomponenten – Abstrakter Algorithmus

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

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

invariant SCCs of G_c are known

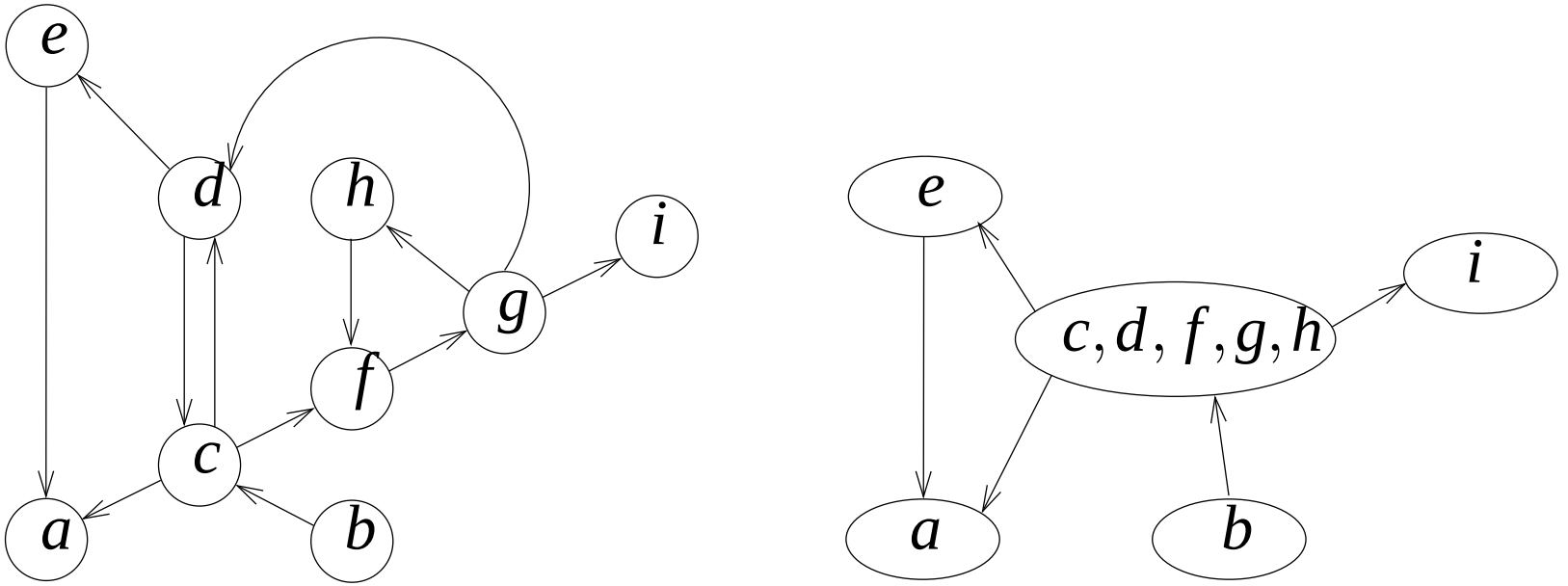
$E_c := E_c \cup \{e\}$

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

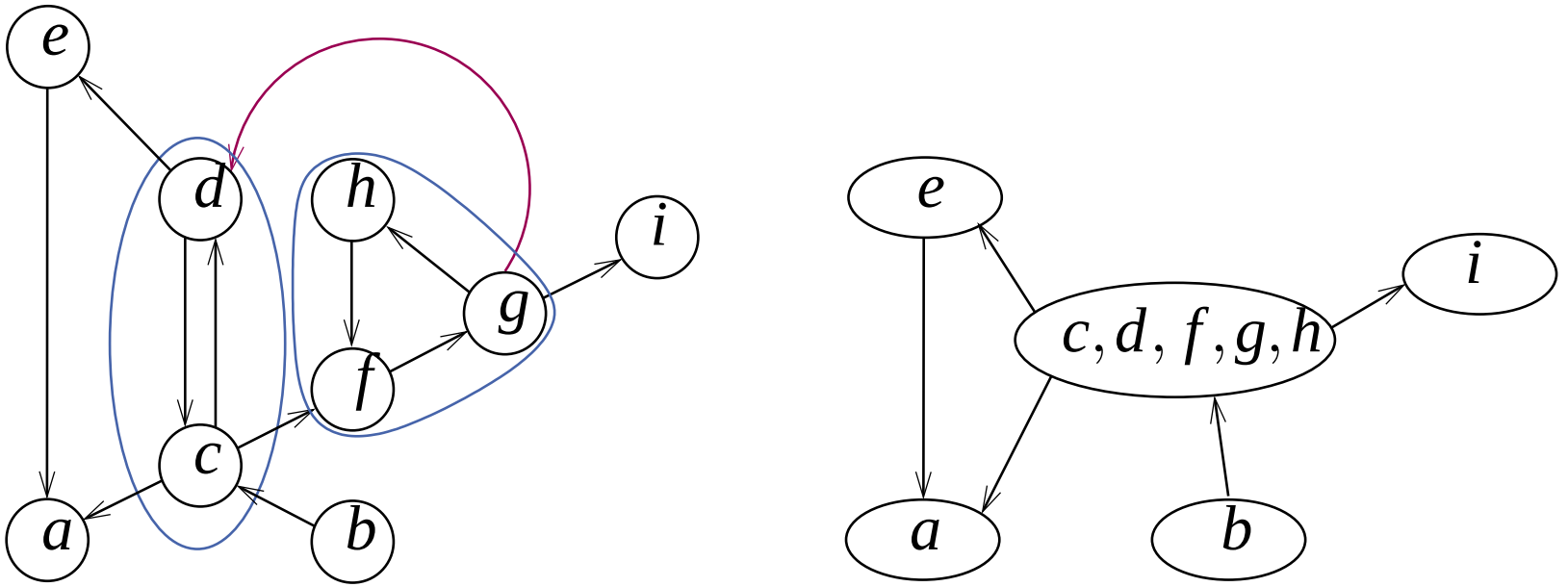
Auswirkungen einer neuen Kante e auf G_c, G_c^s

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

[Cheriyan/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

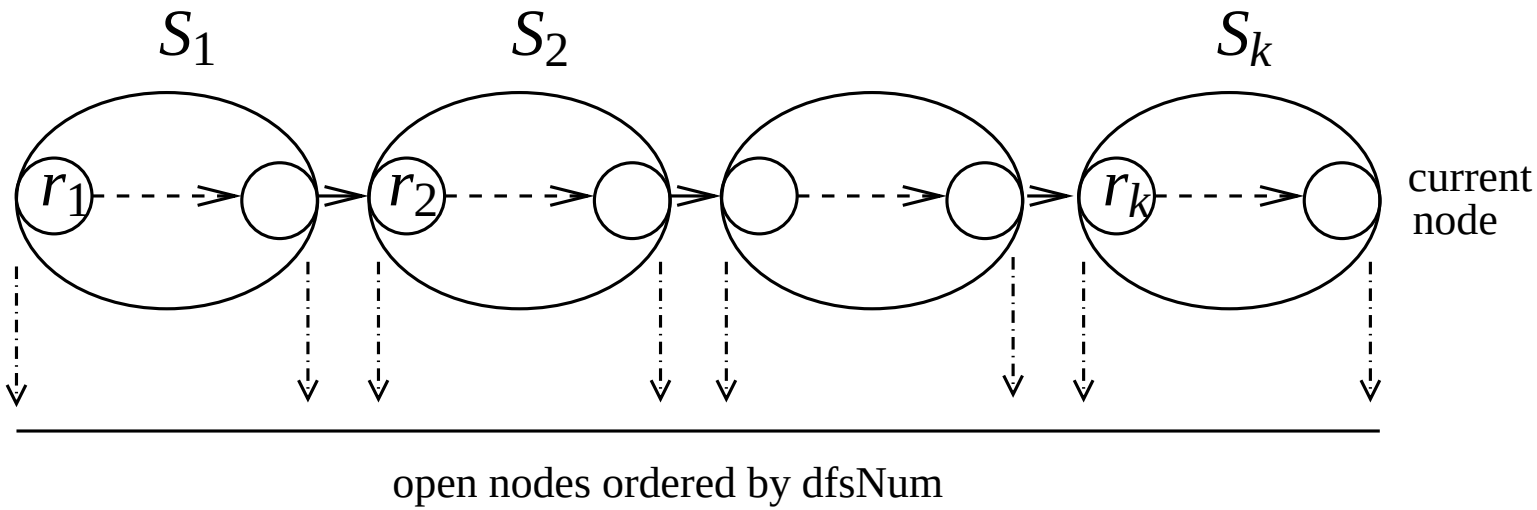
abgeschlossen: alle Knoten finished

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 $S_1,\ldots,S_k$ 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

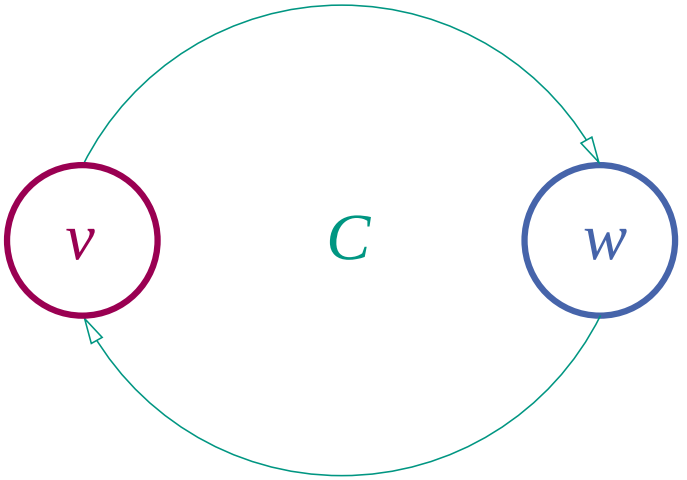
Betrachte abgeschlossenen Knoten v

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.

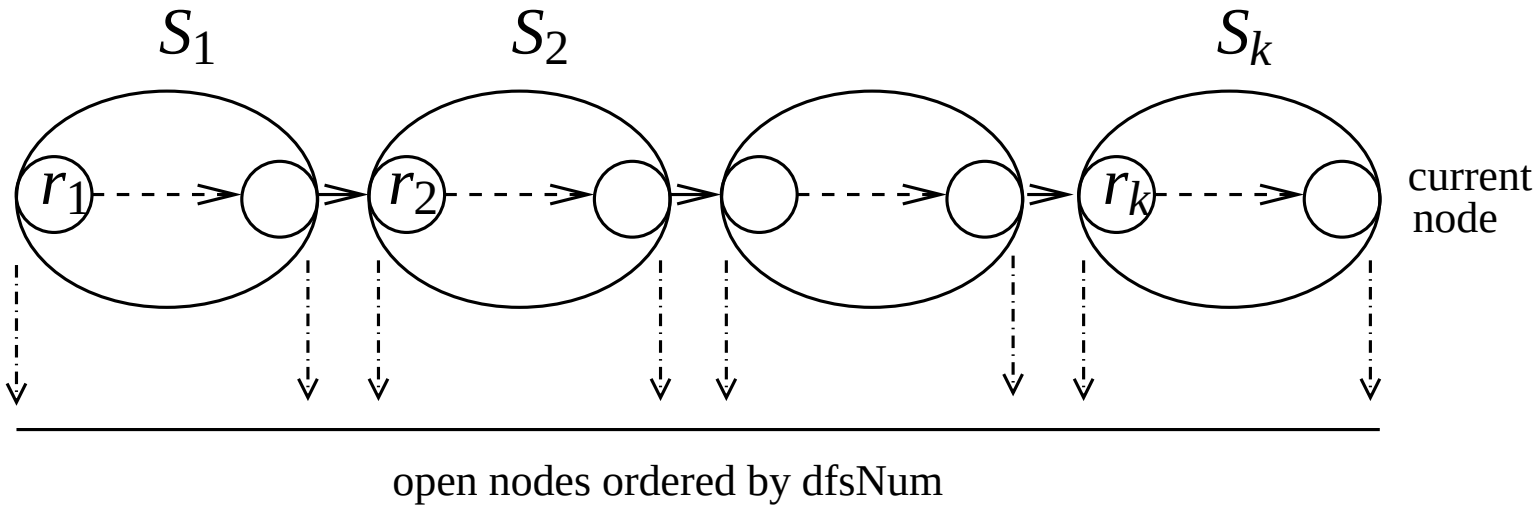
Also sind alle Kanten von C in G_c. □

Repräsentation offener Komponenten

Zwei Stapel aufsteigend sortiert nach dfsNum

oReps: Repräsentanten offener Komponenten

oNodes: Alle offenen Knoten



init

component : NodeArray **of** NodeId // SCC representatives

oReps=⟨⟩ : Stack **of** NodeId // representatives of open SCCs

oNodes=⟨⟩ : Stack **of** NodeId // all nodes in open SCCs

Alle Invarianten erfüllt.

(Weder offene noch geschlossene Knoten)

root(s)

oReps.push(s) // new open

oNodes.push(s) // component

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

Alle Invarianten bleiben gültig

traverseTreeEdge(v, w)

oReps.push(w) // new open

oNodes.push(w) // component

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

dfsNum$(w) >$ alle anderen.

$\rightsquigarrow$Alle Invarianten bleiben gültig

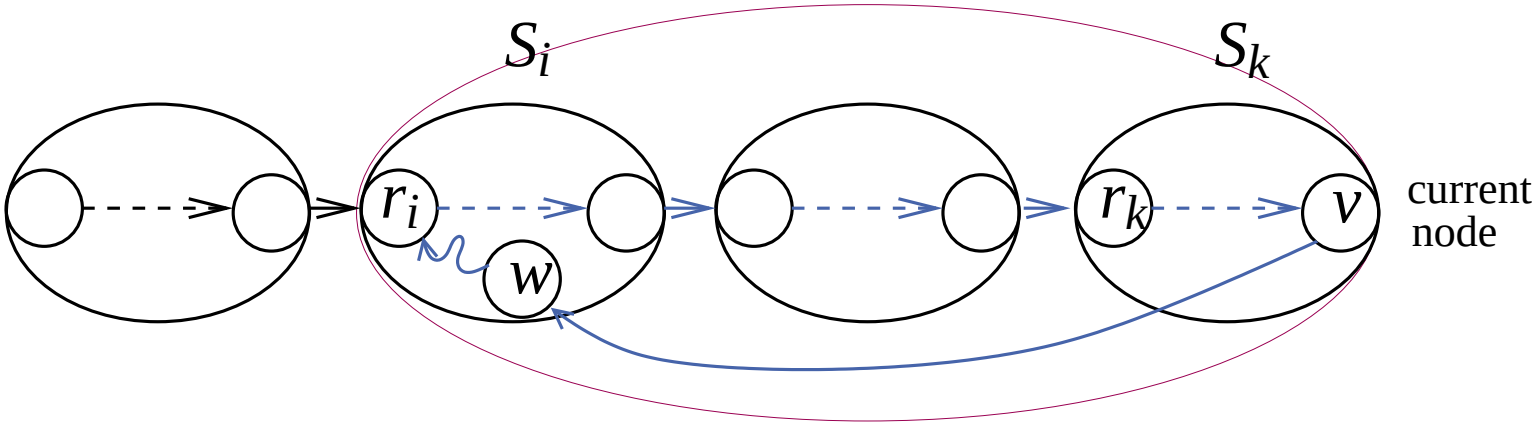
traverseNonTreeEdge(v, w)

if $w \in$ oNodes **then**

while $w \prec$ oReps.top **do** oReps.pop

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

$w \in$ oNodes: kollabiere offene SCCs auf Kreis



backtrack(u, v)

if $v =$ oReps.top **then**

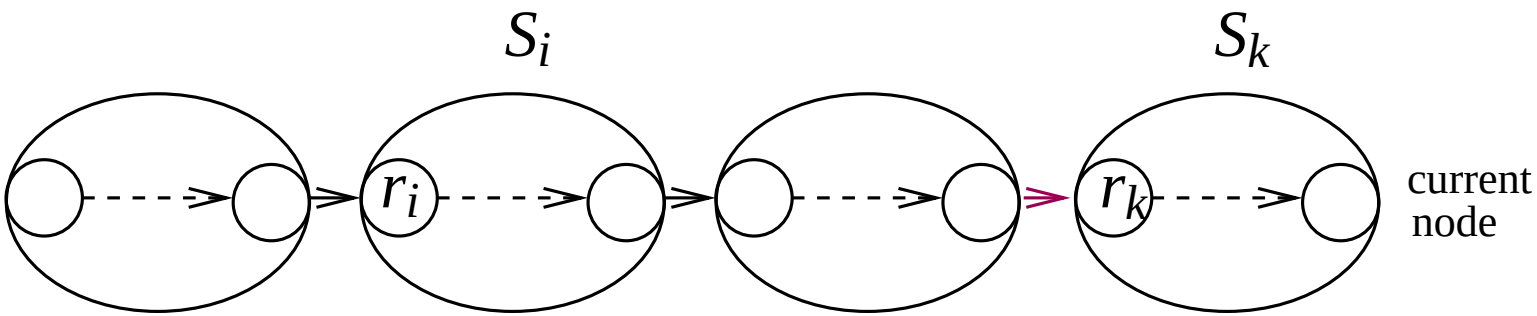
oReps.pop // close

repeat // component

w:= oNodes.pop

component$[w]:= v$

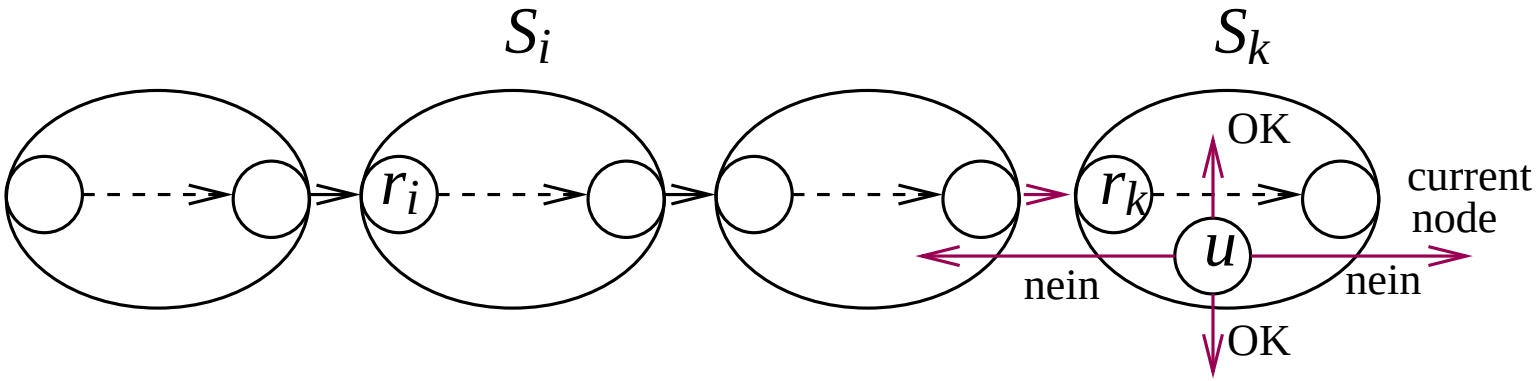
until $w = v$



z.Z. Invarianten bleiben erhalten. . .

```
backtrack(u, v)
    if v = oReps.top then
        oReps.pop                                   // close
        repeat                                      // component
            w:= oNodes.pop
            component[w]:= v
        until w = v
```

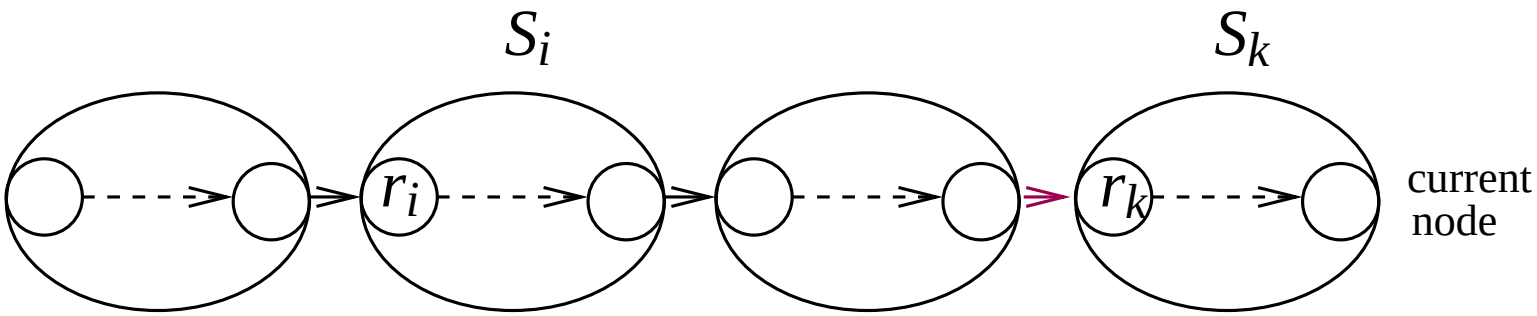
Inv. 1: Kanten von abgeschlossenen Knoten gehen zu abgeschlossenen Knoten.



```
backtrack(u, v)
    if v = oReps.top then
        oReps.pop                                   // close
        repeat                                      // component
            w:= oNodes.pop
            component[w]:= v
        until w = v
```

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

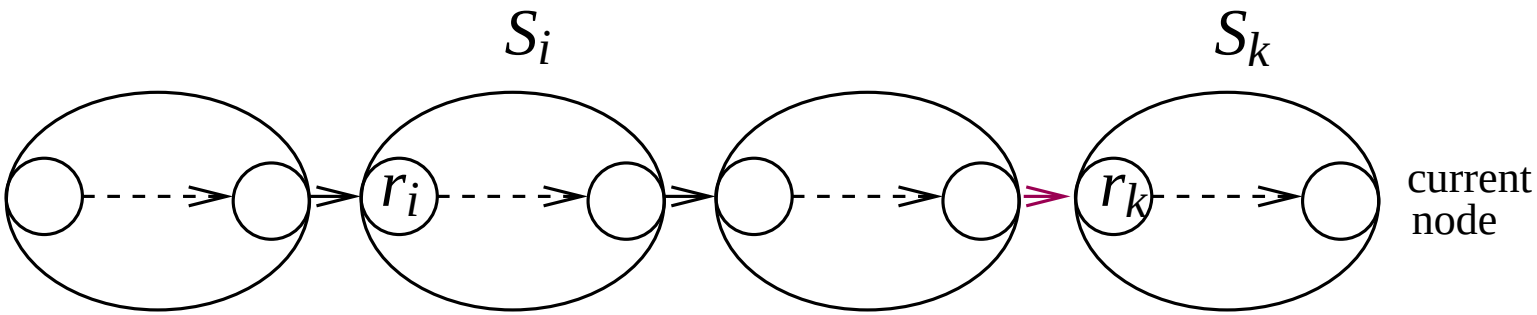
OK. (S_k wird ggf. entfernt)



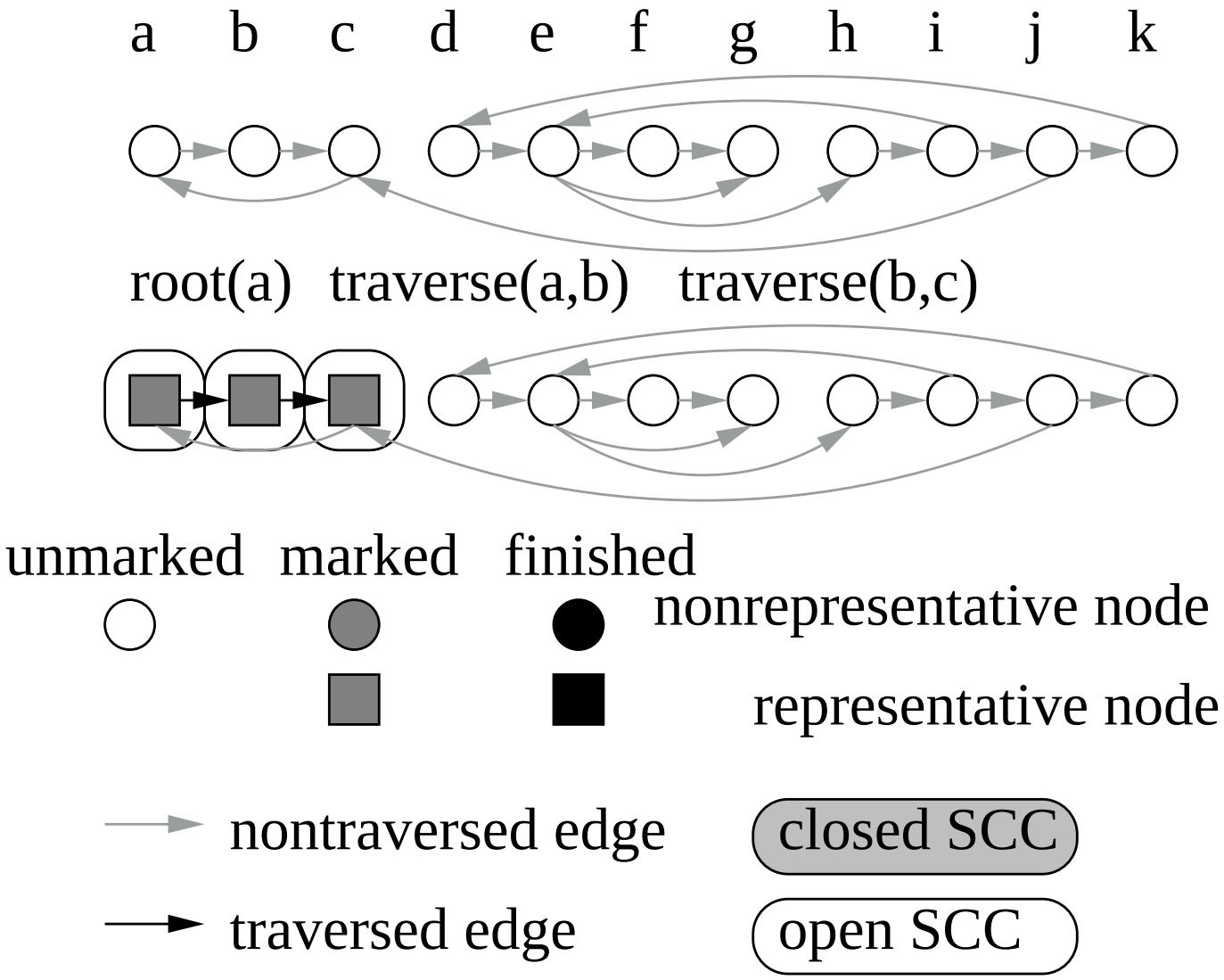
```
backtrack(u, v)
    if v = oReps.top then
        oReps.pop                                   // close
        repeat                                      // component
            w:= oNodes.pop
            component[w]:= v
        until w = v
```

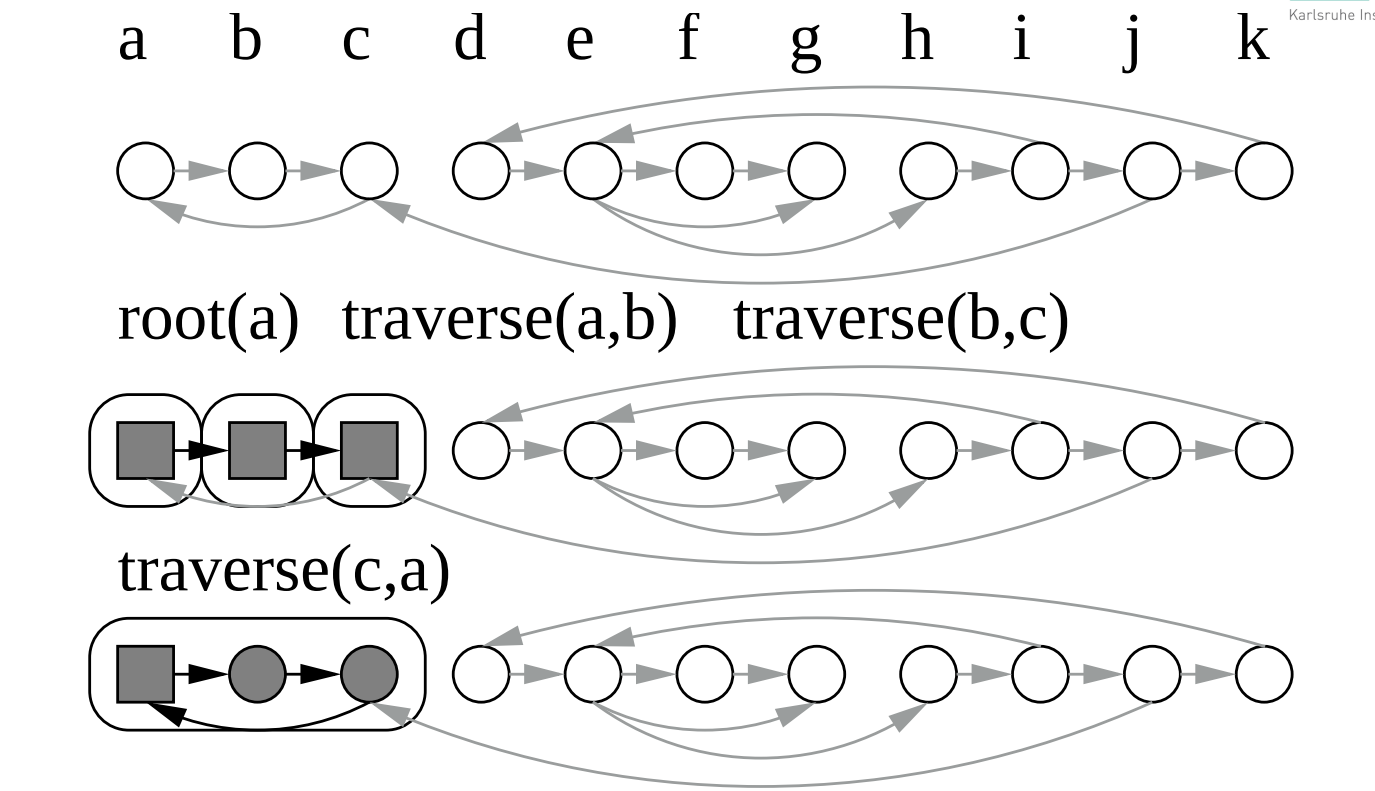
Inv. 3: Repräsentanten partitionieren die offenen Komponenten bzgl. ihrer dfsNum.

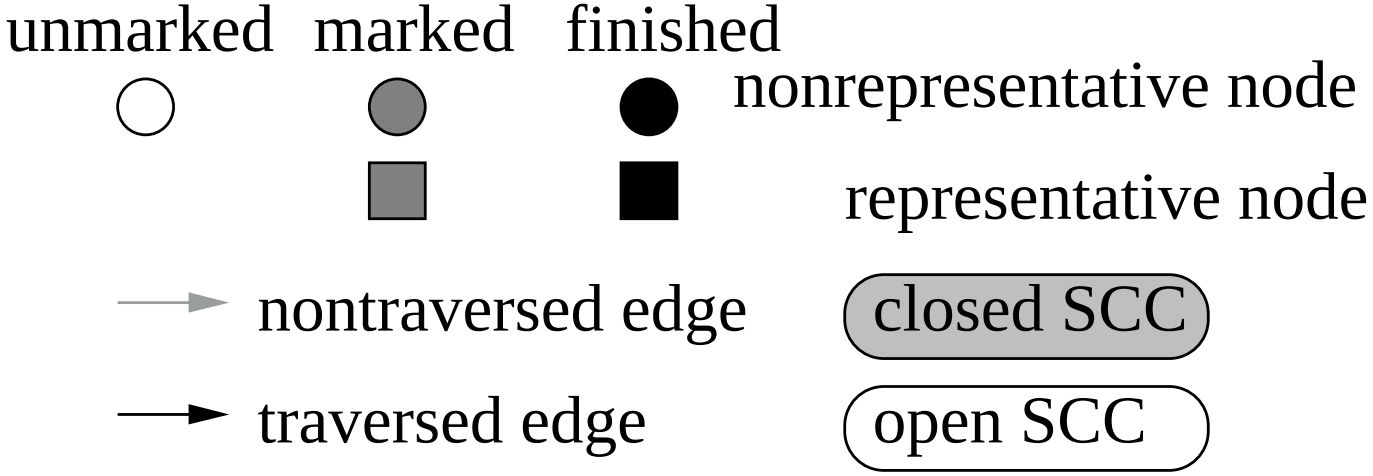
OK. (S_k wird ggf. entfernt)

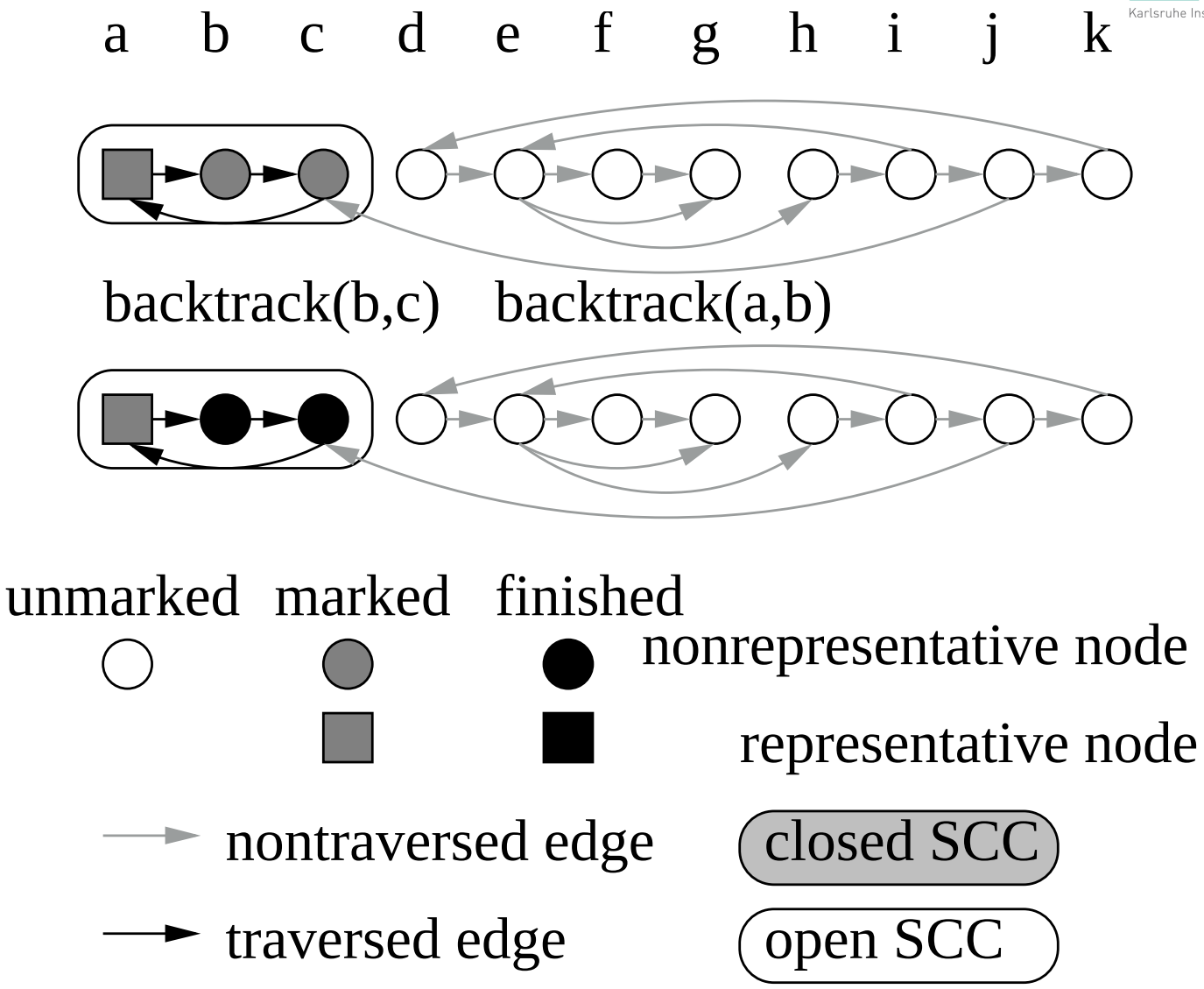


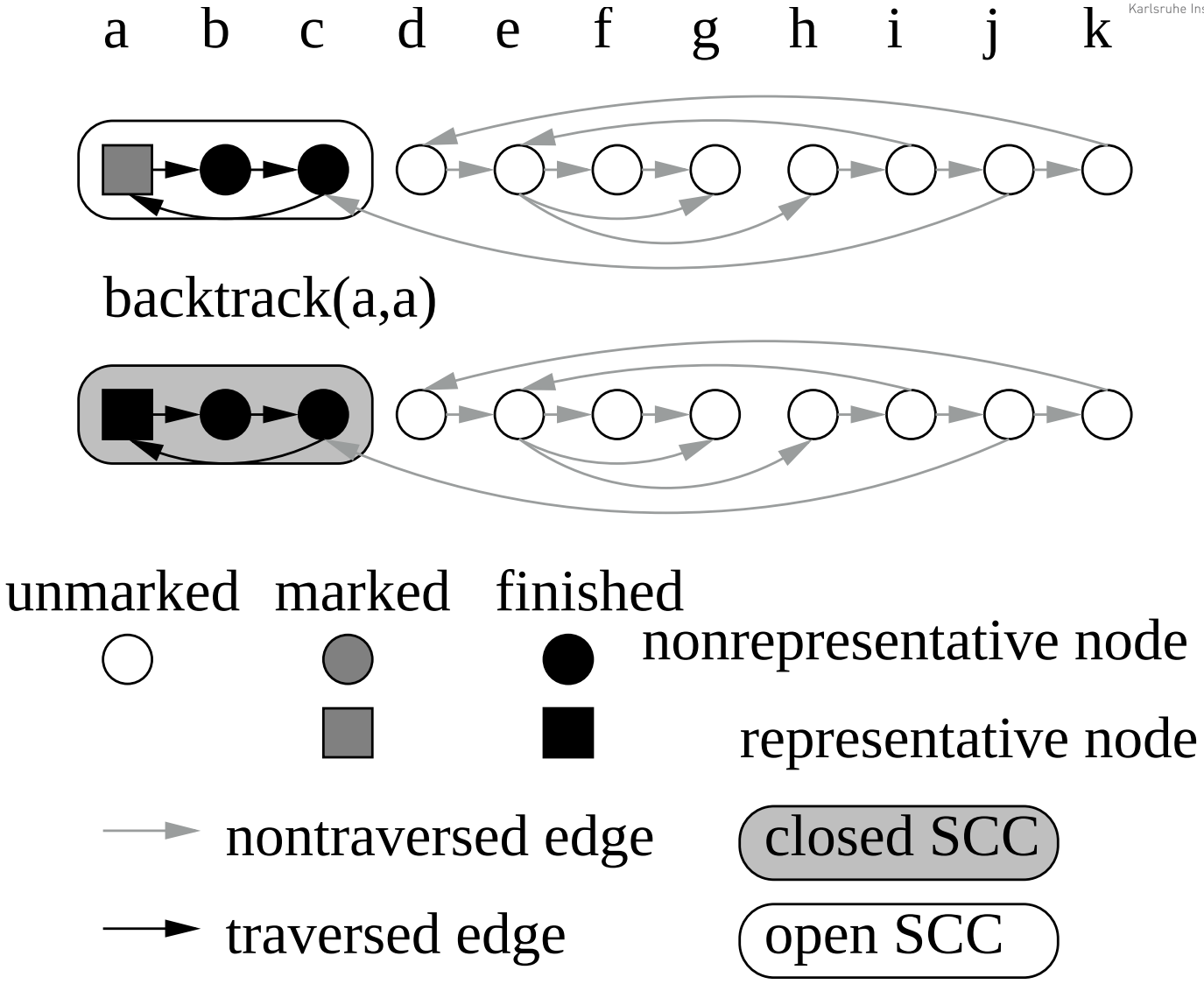
Beispiel

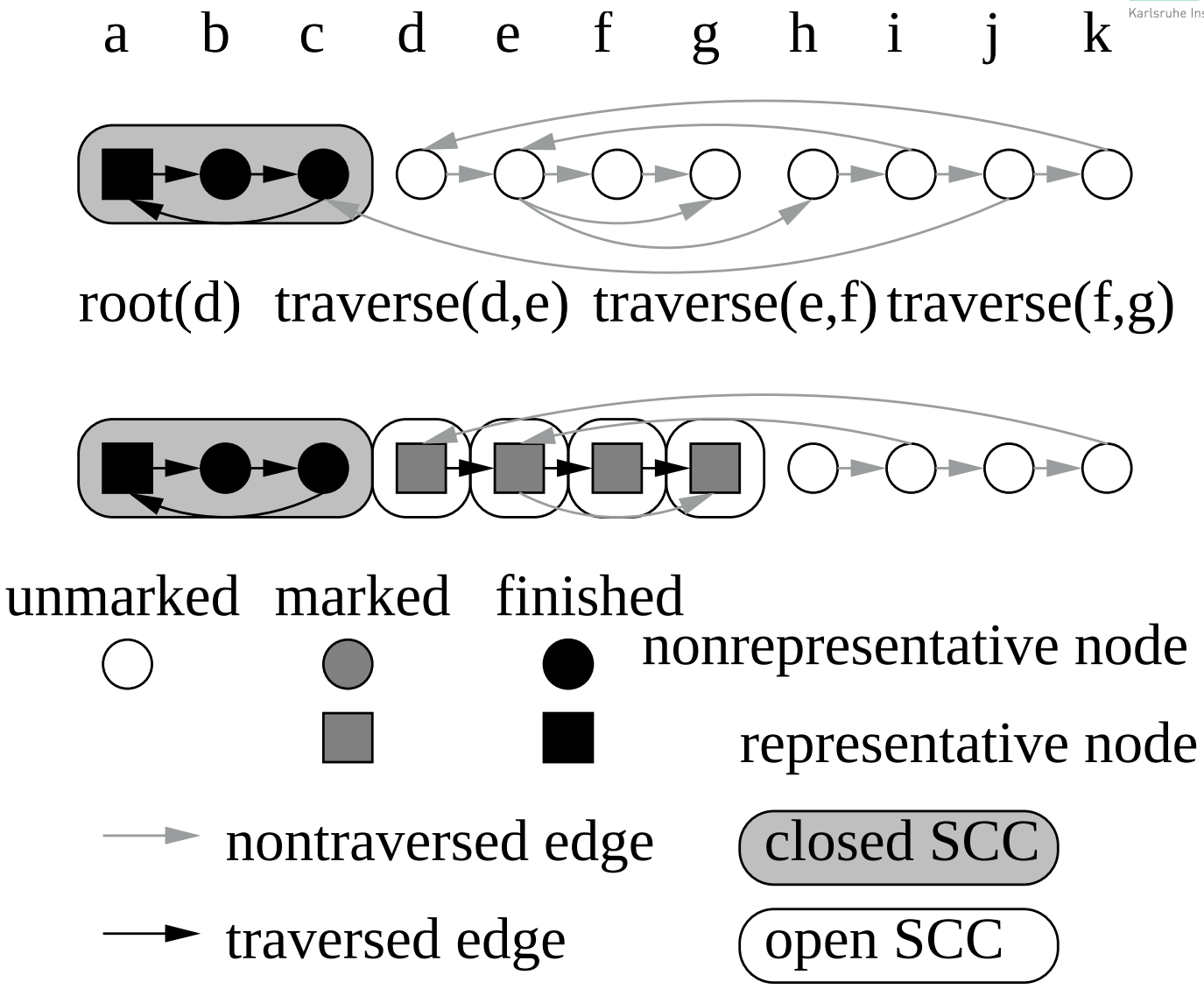

a b c d e f g h i j k
root(a) traverse(a,b) traverse(b,c)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

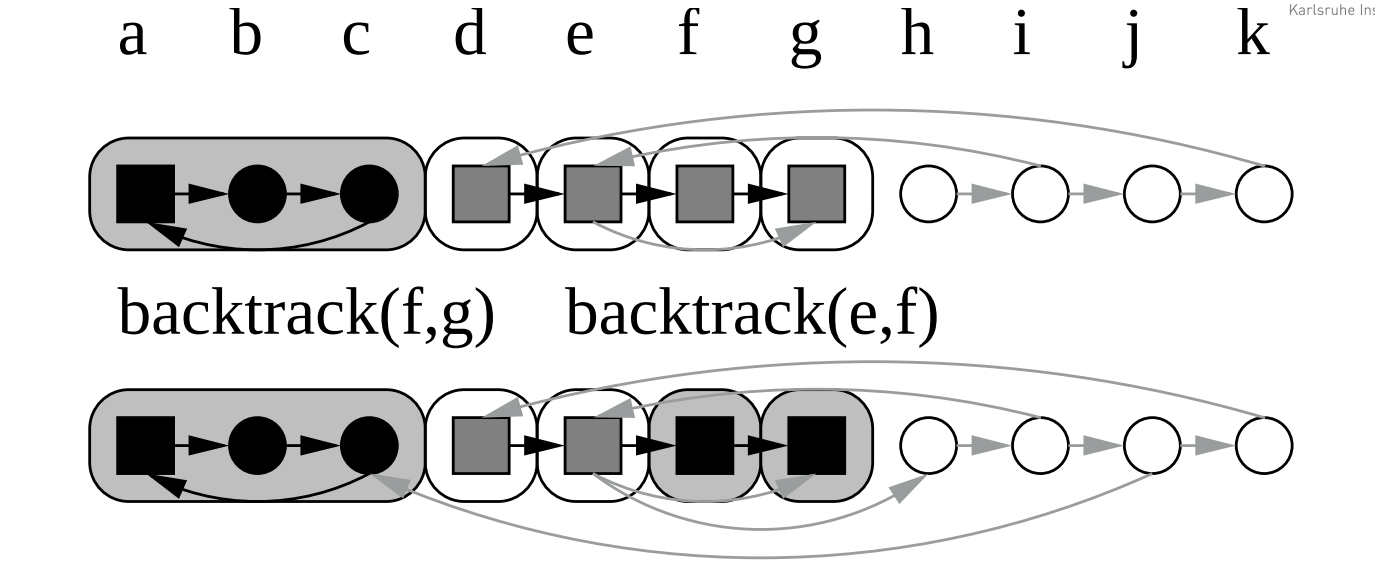

a b c d e f g h i j k
root(a)
traverse(a,b)
traverse(b,c)
traverse(c,a)

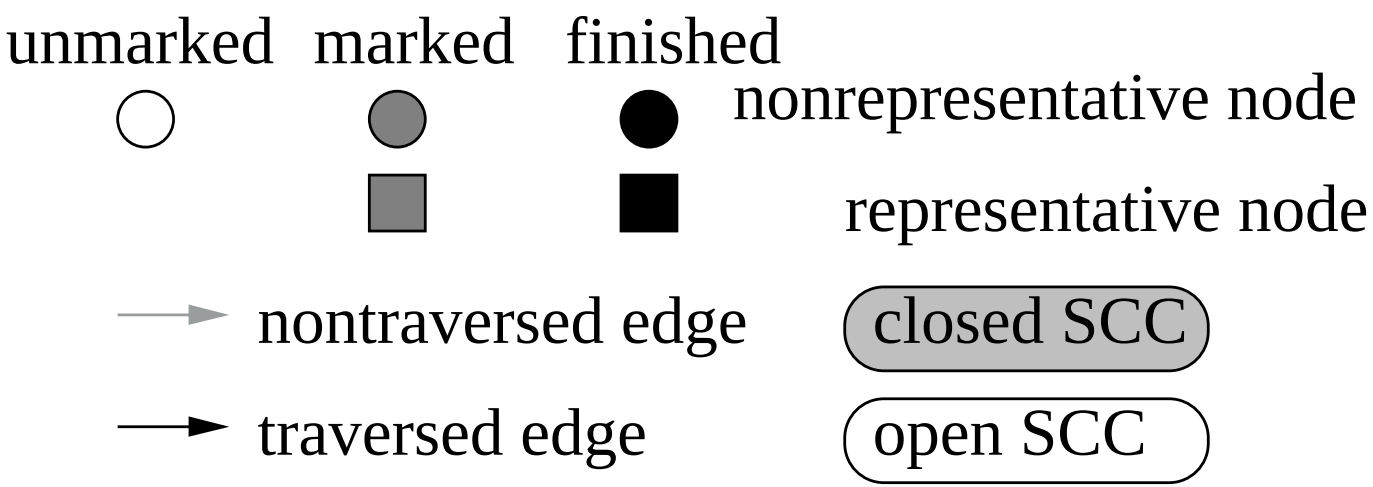

unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

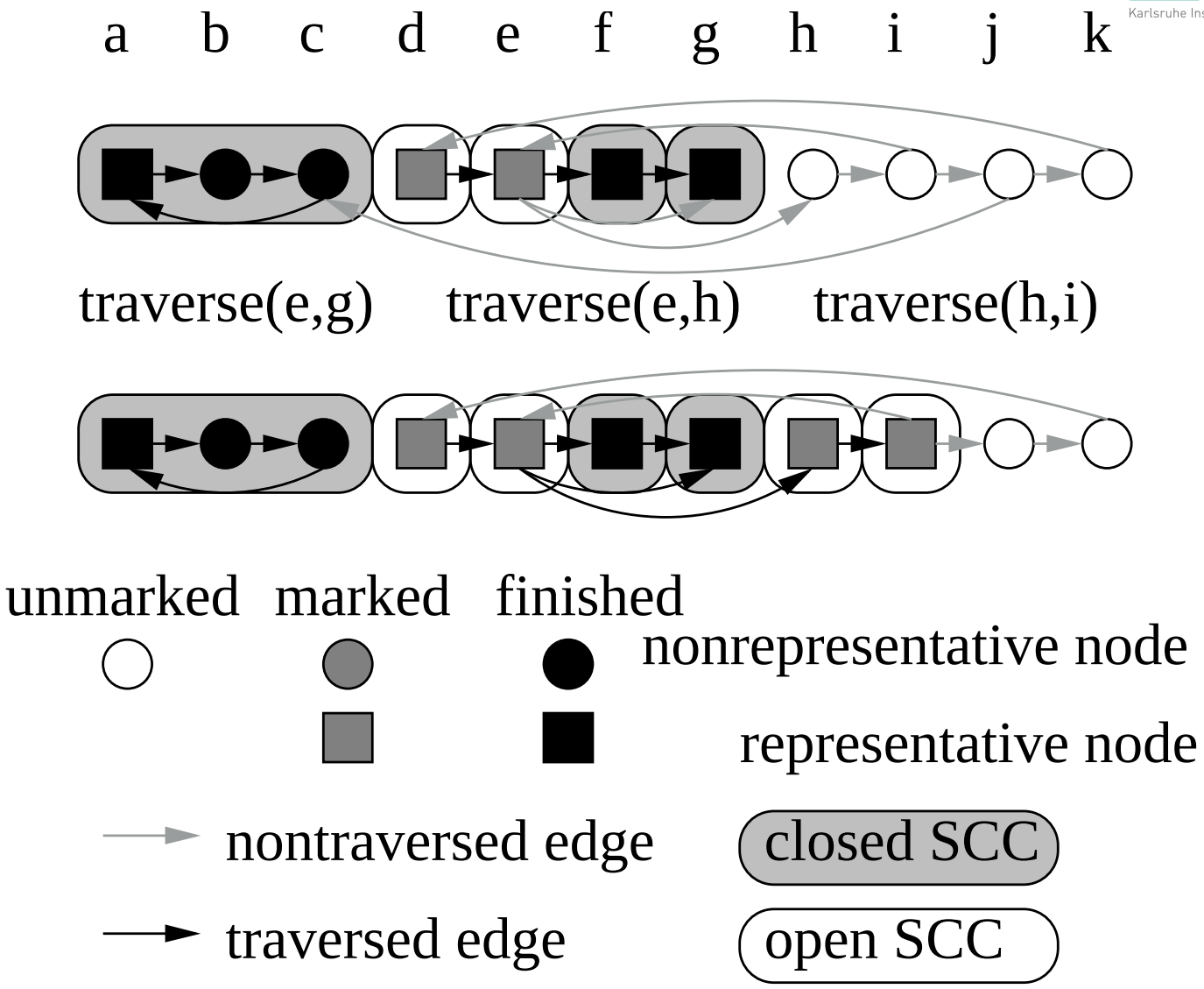

a b c d e f g h i j k
backtrack(b,c)
backtrack(a,b)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

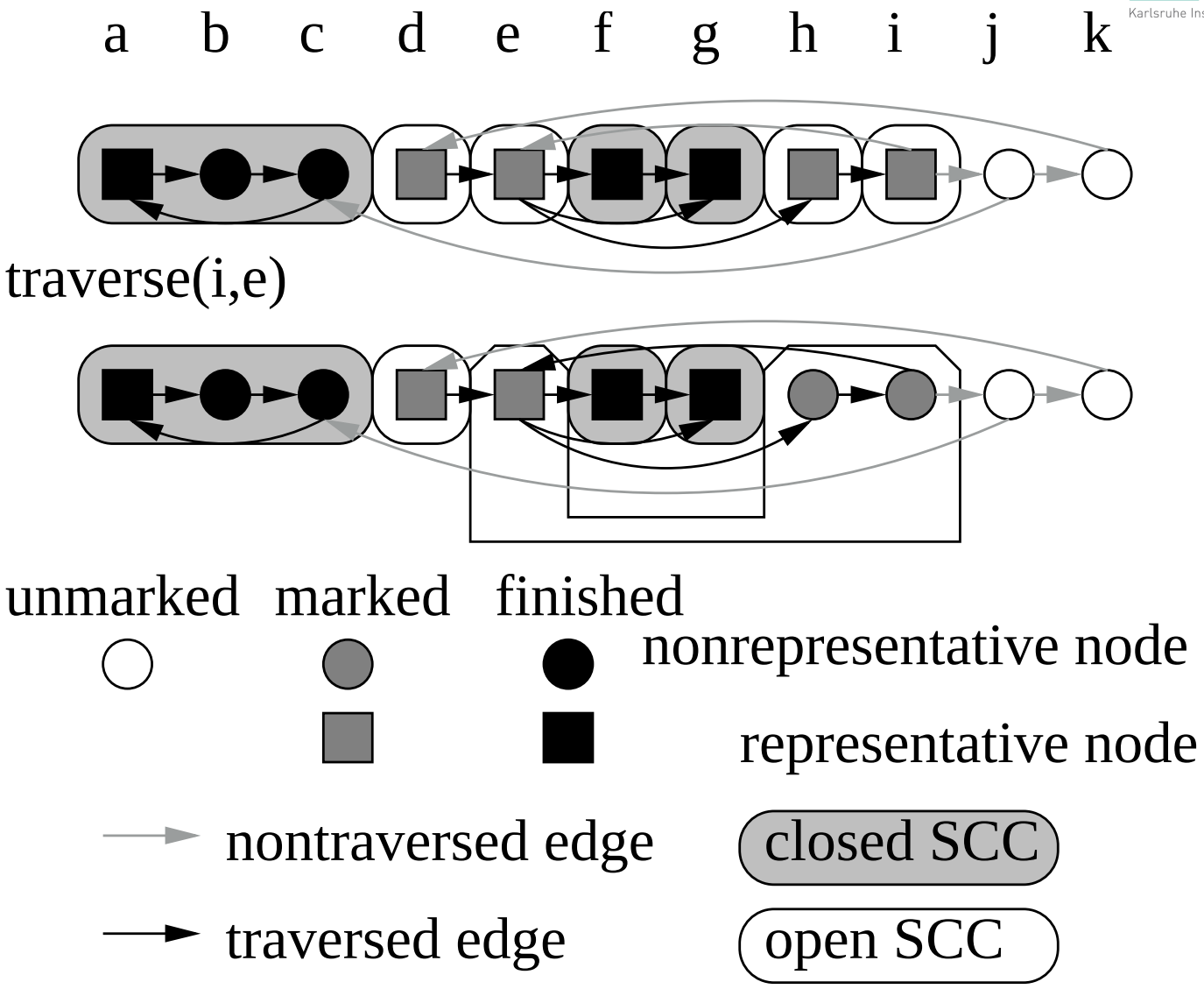

a b c d e f g h i j k
backtrack(a,a)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

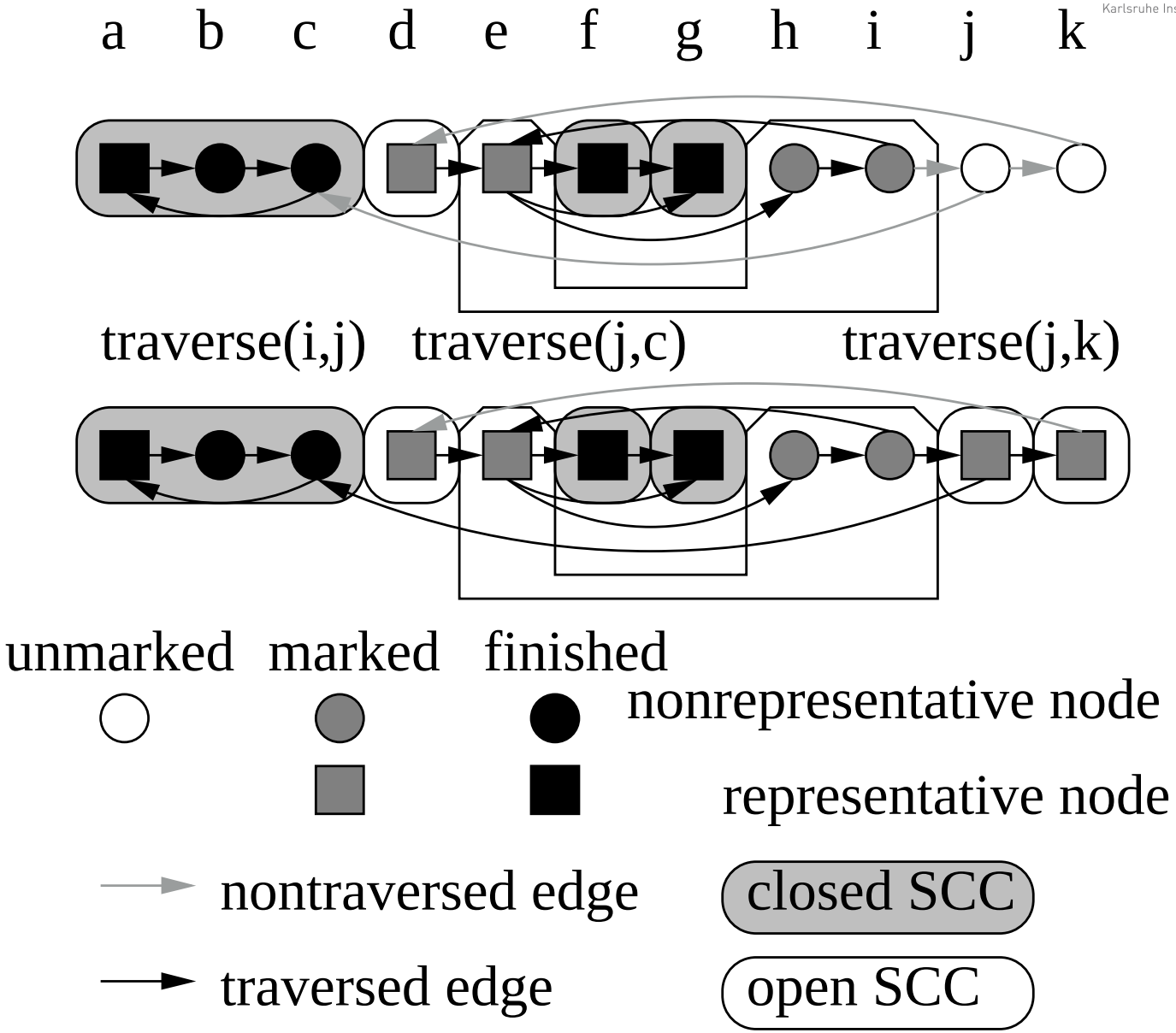

a b c d e f g h i j k
root(d)
traverse(d,e)
traverse(e,f)
traverse(f,g)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

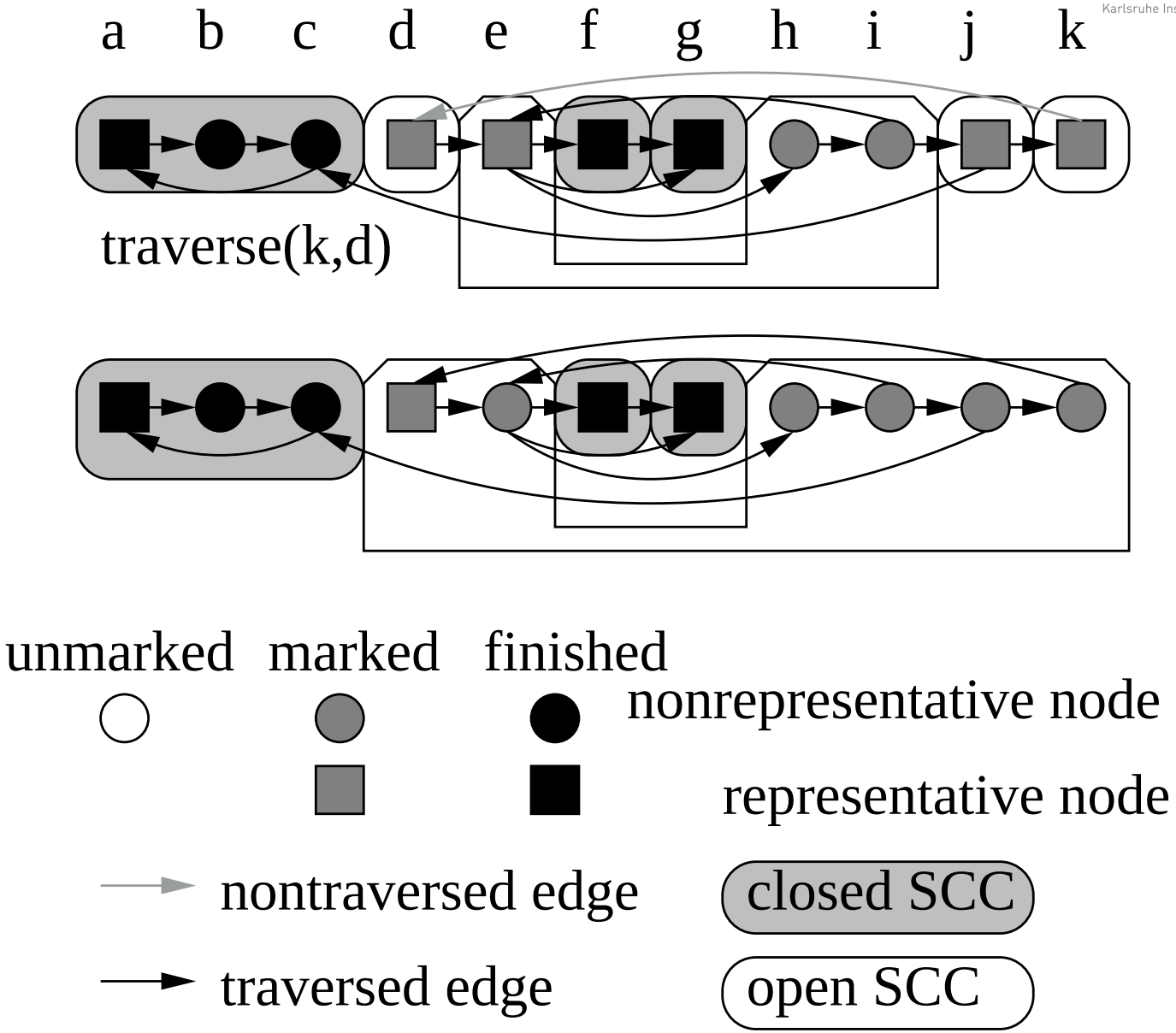

a b c d e f g h i j k
backtrack(f,g)
backtrack(e,f)

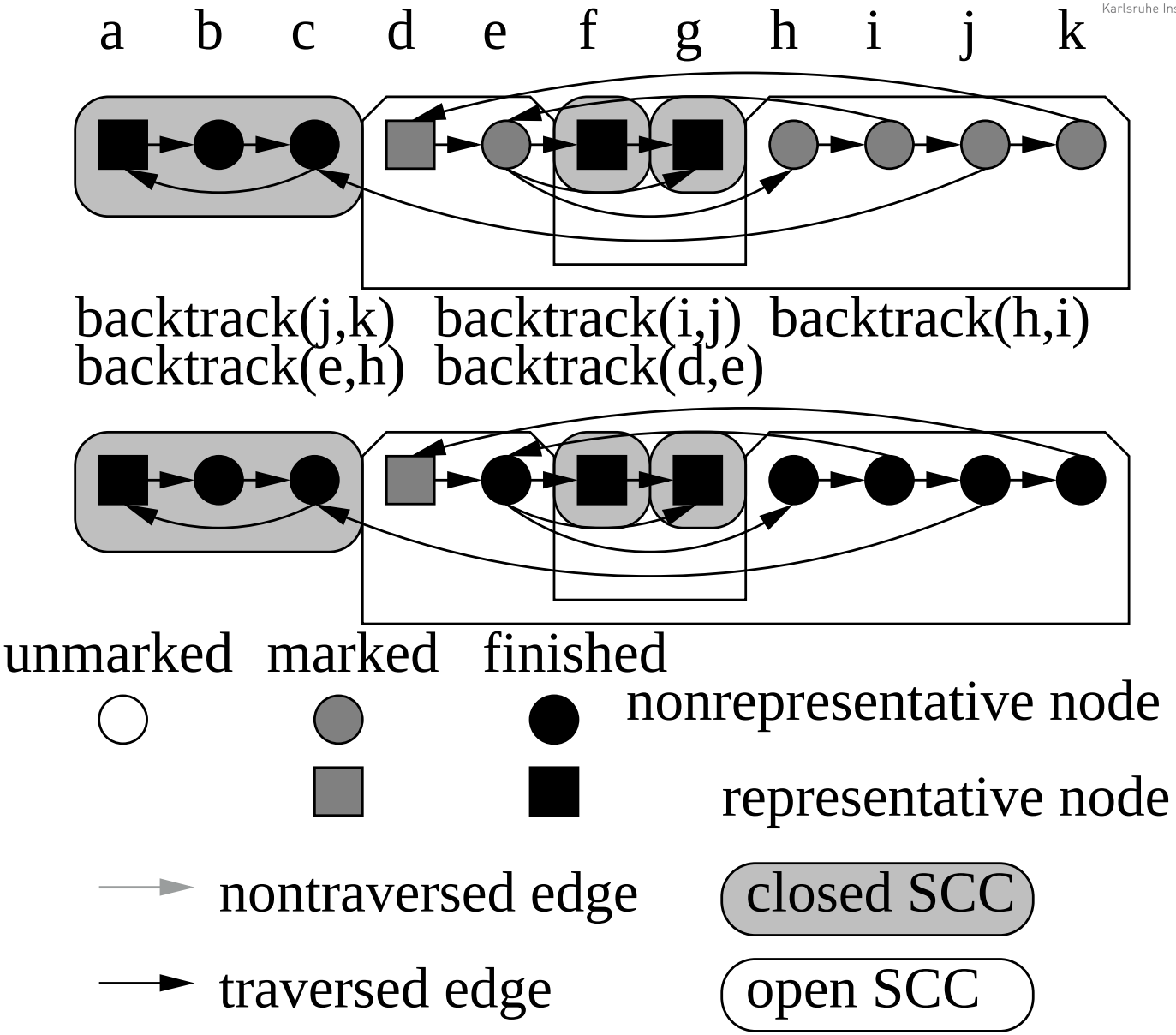

unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

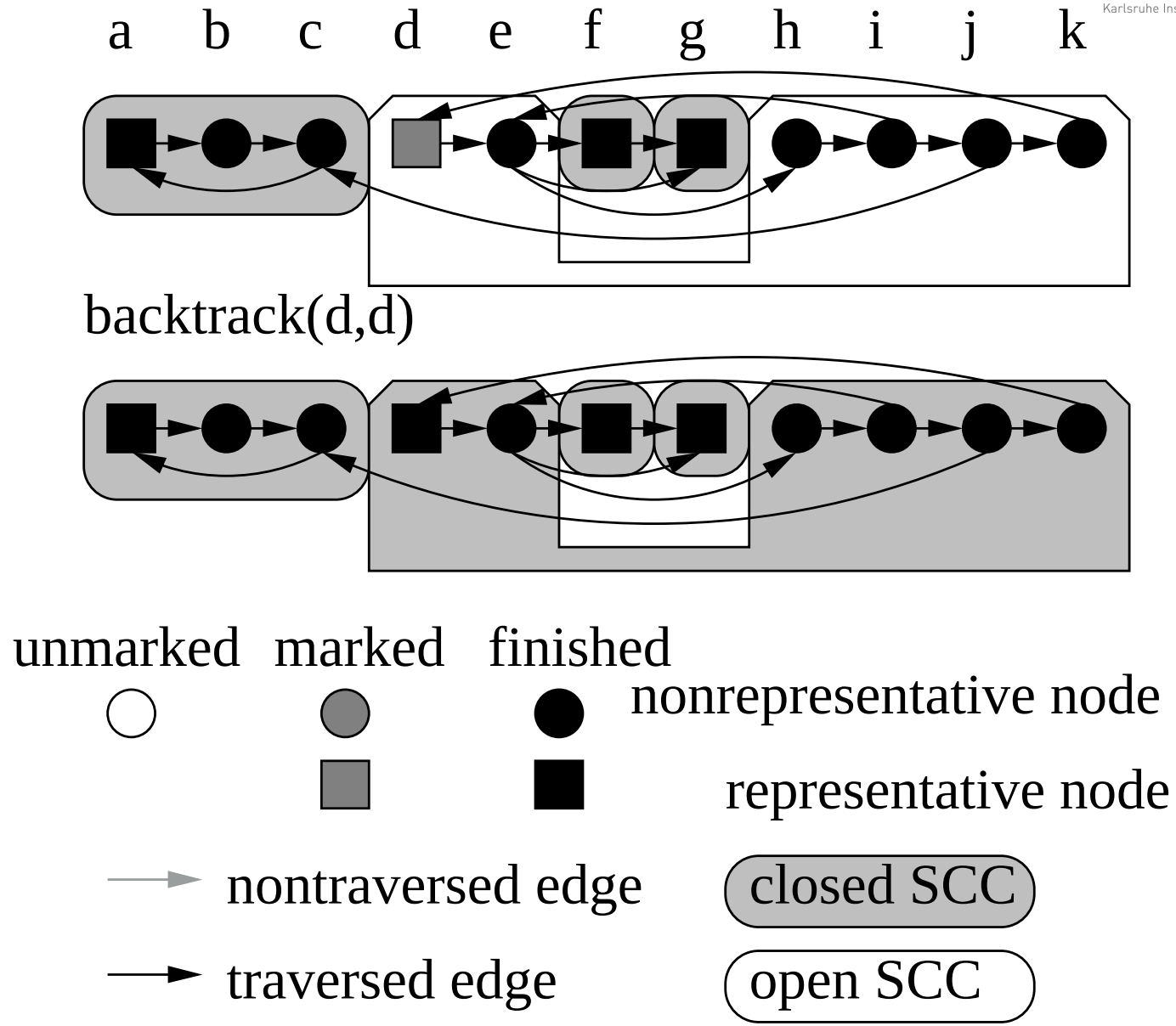

a b c d e f g h i j k
traverse(e,g)
traverse(e,h)
traverse(h,i)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC


a b c d e f g h i j k
traverse(i,e)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC


a b c d e f g h i j k
traverse(i,j)
traverse(j,c)
traverse(j,k)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC


a b c d e f g h i j k
traverse(k,d)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC


a b c d e f g h i j k
backtrack(j,k) backtrack(i,j) backtrack(h,i)
backtrack(e,h) backtrack(d,e)
unmarked marked finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC


a b c d e f g h i j k
backtrack(d,d)
unmarked
marked
finished
nonrepresentative node
representative node
nontraversed edge
traversed edge
closed SCC
open SCC

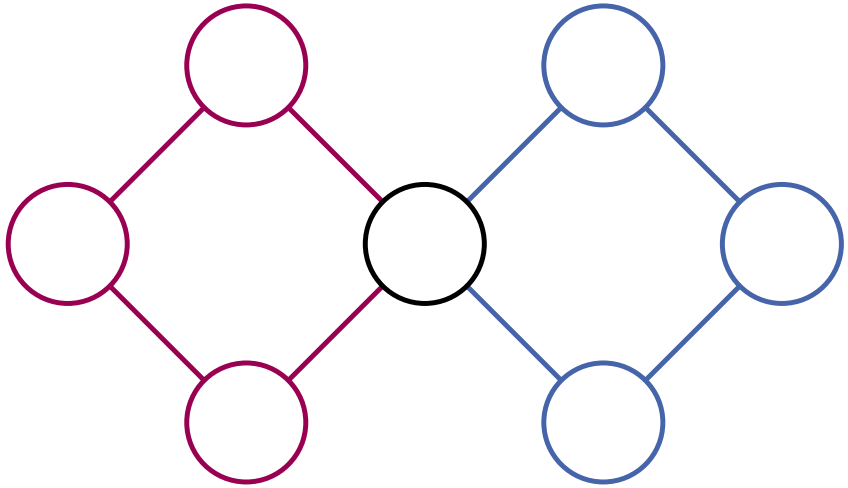
Zusammenfassung: SCC Berechnung

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

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

4 Kürzeste Wege

Folien teilweise von Rob van Stee

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

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

Anfangsknoten s.

Ausgabe: für alle $v \in V$

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

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

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

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

Kante (u,v) relaxieren

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

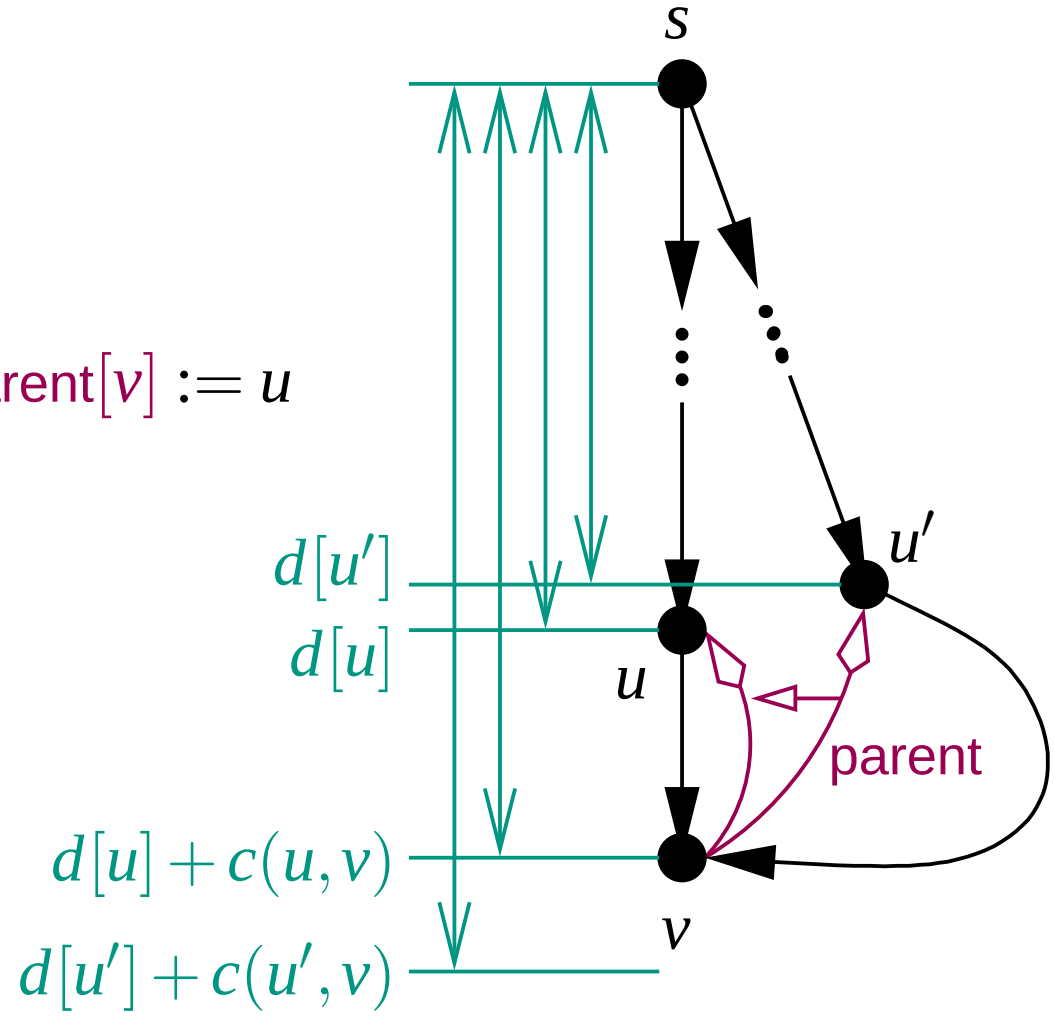
vielleicht $d[v]=\infty$

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

Invarianten bleiben erhalten!

Beobachtung:

$d[v]$ Kann sich mehrmals ändern!



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 $d[v]$

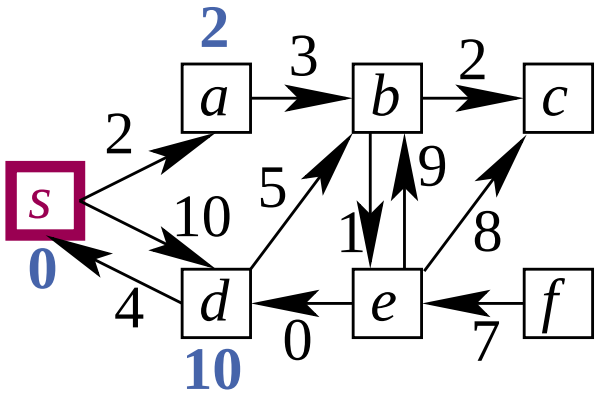
relax all edges (u, v) out of u

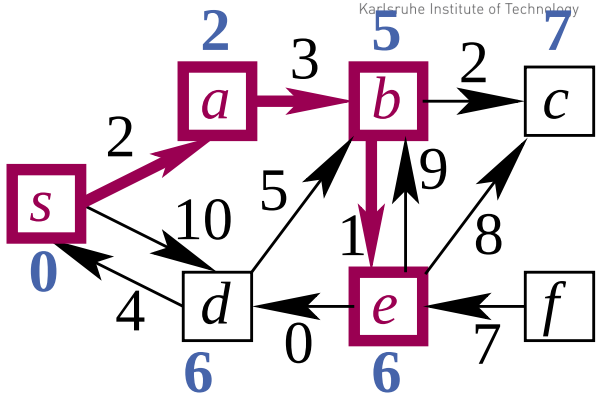
u is scanned now

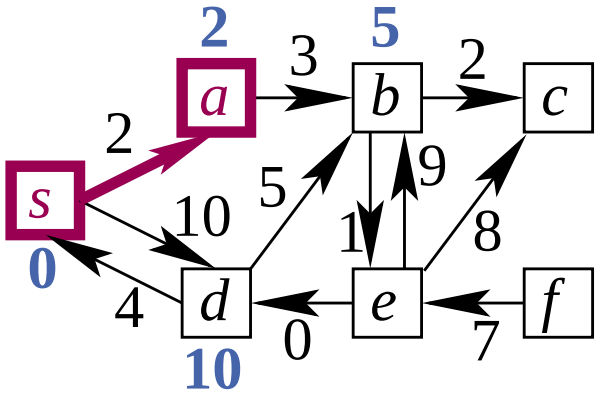
Behauptung: Am Ende definiert d die optimalen Entfernungen

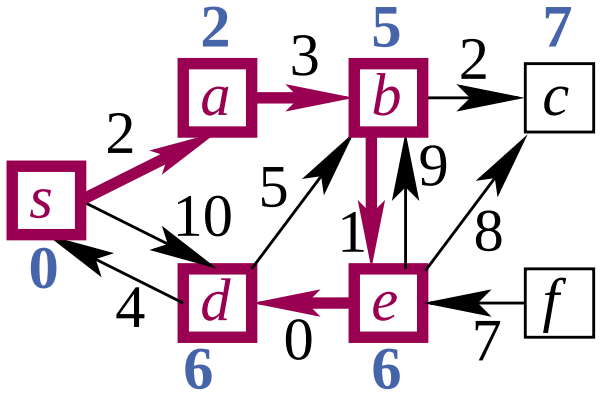
und parent die zugehörigen Wege

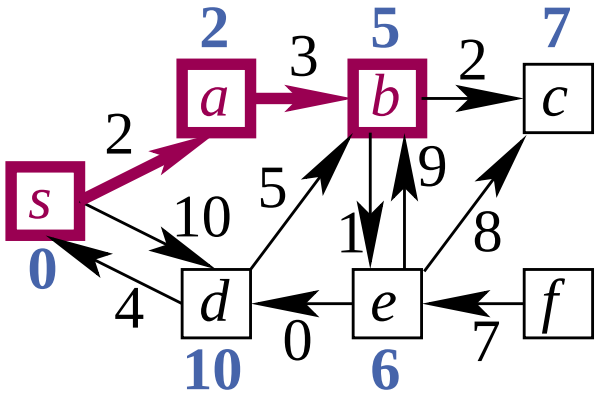
Beispiel

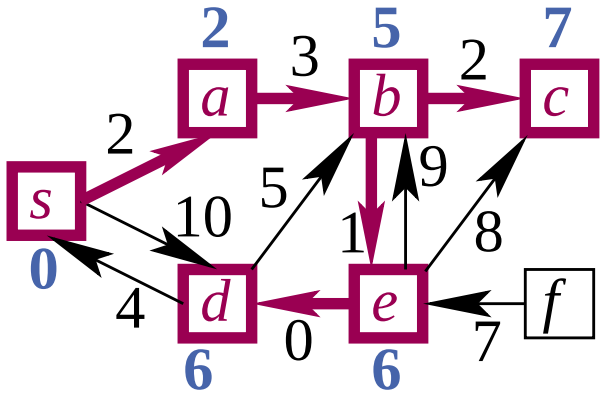

2
a
3
b
2
c
2
s
10
5
9
1
8
0
d
e
f
4
0
7
10


2
5
7
a
3
b
2
c
2
s
10
5
9
1
8
0
d
e
f
4
0
7
6
6


2
5
a
3
b
2
c
2
s
10
5
9
1
8
0
d
e
f
4
0
7
10


2
5
7
a
3
b
2
c
2
s
10
5
9
1
8
0
d
e
f
4
0
7
6
6


2
5
7
a
3
b
2
c
2
s
10
5
9
1
8
0
d
e
f
4
0
7
10
6


2
5
7
a
3
b
2
c
2
s
10
5
9
1
8
0
d
e
f
4
0
7
6
6

Laufzeit

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

Mit Fibonacci-Heapprioritätslisten:

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

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

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

Laufzeit im Durchschnitt

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

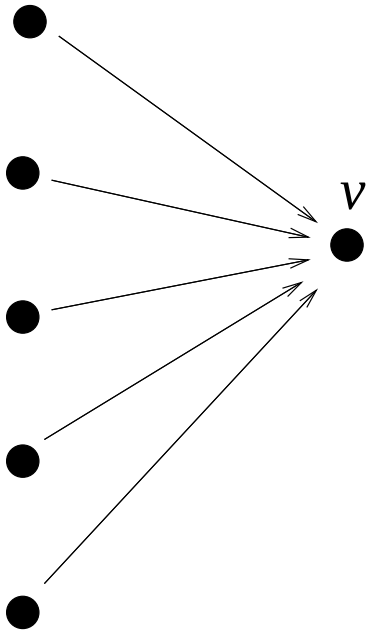
Wieviel decreaseKeys im Durchschnitt?

Modell:

- Beliebiger Graph G
- Beliebiger Anfangsknoten s
- Beliebige Mengen $C(v)$

  von Kantengewichten für

  eingehende Kanten von Knoten v



indegree(v)=5
C(v)={c_1,..., c_5}

Gemittelt wird über alle Zuteilungen

$C(v) \rightarrow$ eingehende Kanten von v

Beispiel: alle Kosten unabhängig identisch verteilt

Laufzeit im Durchschnitt

Probabilistische Sichtweise: Zufällige gleichverteilte Auswahl einer der zu mittelnden Eingaben.

$\Longrightarrow$

wir suchen den Erwartungswert der Laufzeit

Frage: Unterschied zu erwarteter Laufzeit bei randomisierten Algorithmen ?

Laufzeit im Durchschnitt

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

Dann

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

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

($T_{\text{DijkstraFib}} = \mathrm{O}(m + n\log n)$ 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. $\mathbb{E}[\#\textit{decreaseKey-Operationen}] = \mathrm{O}\left(n \log \frac{m}{n}\right)$

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

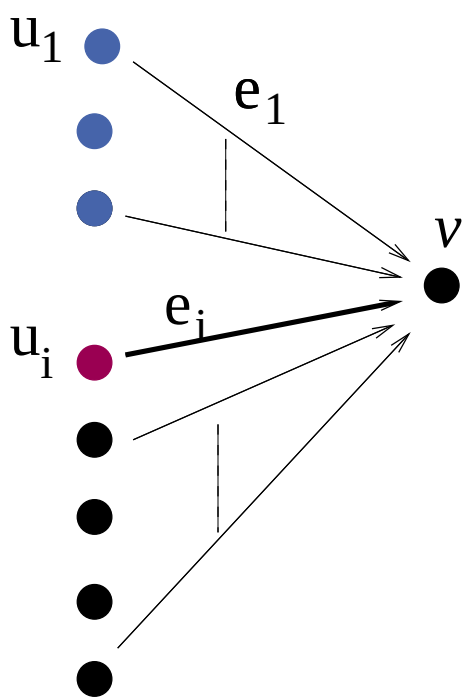
decreaseKey bei Bearbeitung von e_i nur wenn

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

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. $\mathbb{E}[\#\textit{decreaseKey-Operationen}] = \mathrm{O}\left(n \log \frac{m}{n}\right)$

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 insert(v).

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

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

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.

$$\mathrm{E}[M_n] = \mathrm{E}[\sum_{i=1}^{n} I_i] \overset{\text{Lin. } \mathrm{E}[\cdot]}{=} \sum_{i=1}^{n} \mathrm{E}[I_i]$$

$$= \sum_{i=1}^{n} \frac{1}{i} = H_n \quad \text{wegen } \mathbb{P}[I_i = 1] = \frac{1}{i}$$

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

Monotone ganzzahlige Prioritätslisten

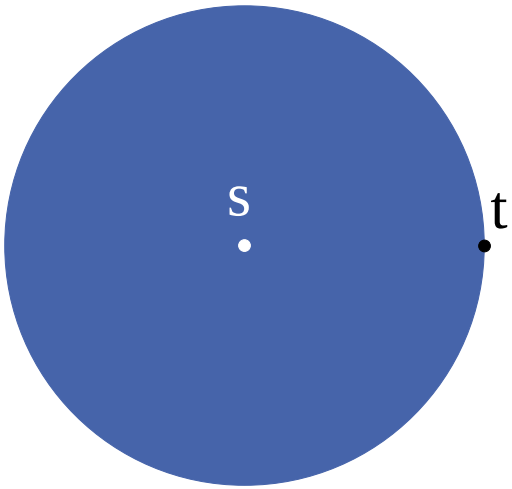
Grundidee: Datenstruktur auf Anwendung zuschneiden

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 Kosten sind ganzzahlig und im Interval $[0, C]$

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

Es gilt sogar:

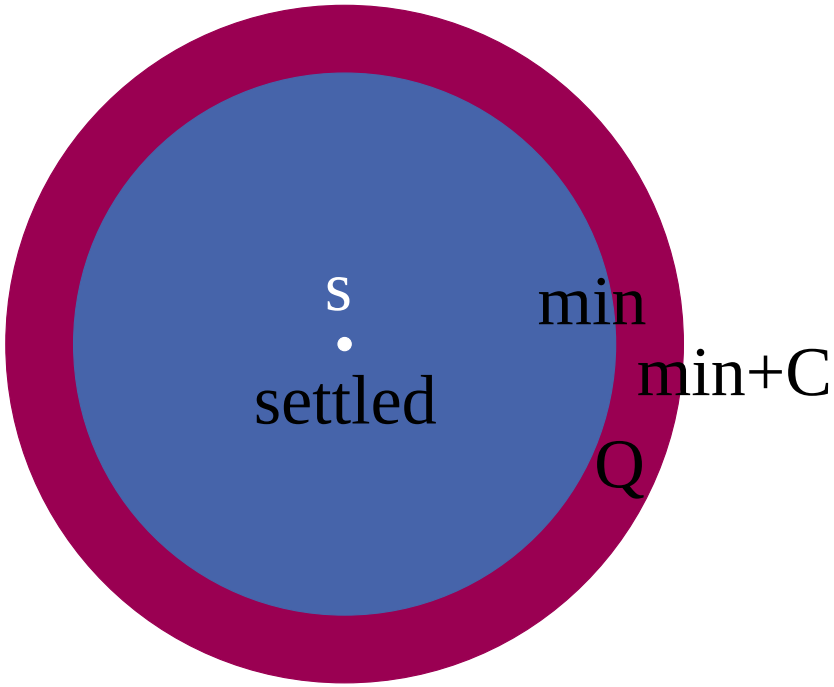
Sei min der letzte Wert,

der aus Q entfernt wurde.

In Q sind immer

nur Knoten mit Distanzen im

Interval $[min, min + C]$.

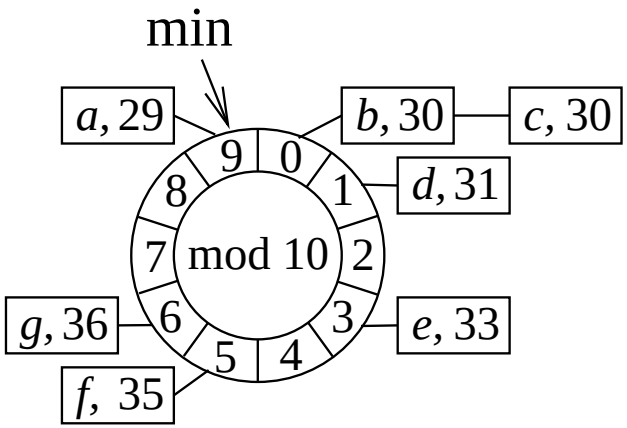


Bucket-Queue

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

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

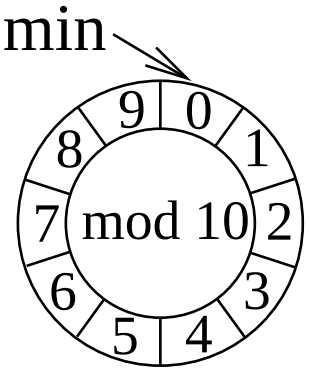
Bucket queue with C = 9



Content=

<(a,29), (b,30), (c,30), (d,31)
(e,33), (f,35), (g,36)>

Operationen



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

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

decreaseKey(v): entfernt v aus seiner Liste und fügt es ein in $B[d[v] \mod (C+1)]$ $O(1)$

deleteMin: fängt an bei Bucket $B[min \mod (C+1)]$. 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

$$
\begin{aligned}
T_{\text{Dijkstra}} &= O(m \cdot T_{\text{decreaseKey}}(n) \\
&\quad + \text{Kosten deleteMin-Operationen} \\
&\quad + n \cdot T_{\text{insert}}(n))) \\
T_{\text{DijkstraBQ}} &= O(m \cdot 1 + nC + n \cdot 1)) \\
&= O(m + nC) \text{ oder auch} \\
&= O(m + n + \text{maxPathLength})
\end{aligned}
$$

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

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 $d[v] \in [min, \ldots, min + C]$.

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 $i$ten 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	$1100\mathbf{1}010$	$1010\mathbf{1}00$	1110110
b	$1100\mathbf{0}101$	$1010\mathbf{1}10$	1110110
$\text{msd}(a,b)$	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

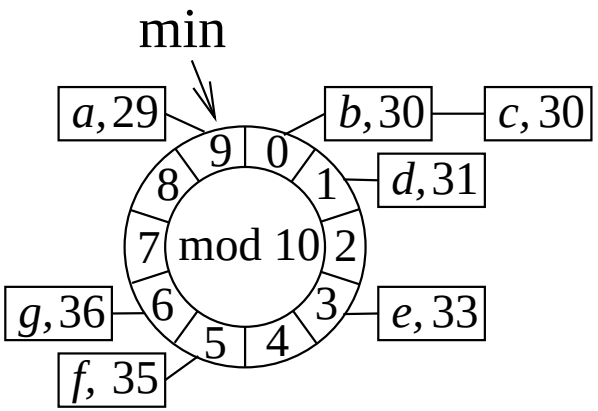
v ist gespeichert in Bucket $B[i]$ wo $i = \min(msd(min, d[v]), K)$.

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

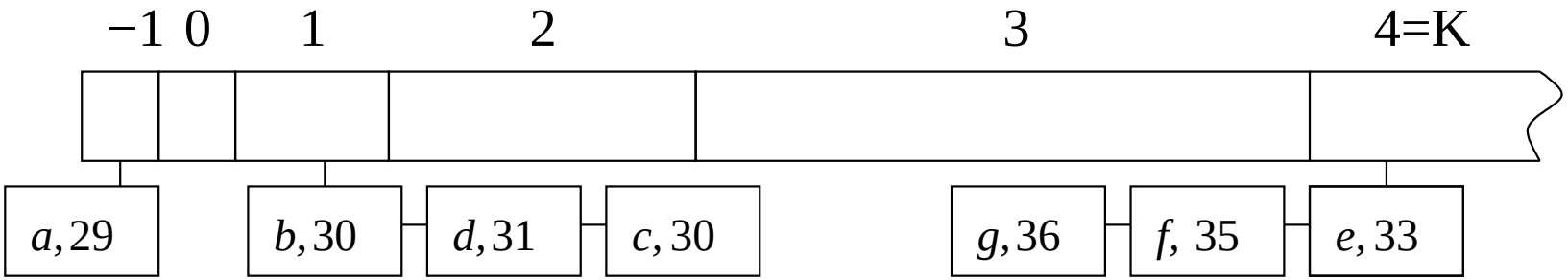
Falls $d[v] \geq 32$, dann $msd(min, d[v]) > 4$!

Bucket-Queues und Radix-Heaps



Bucket queue with C = 9

Content=
<(a,29), (b,30), (c,30), (d,31), (e,33), (f,35), (g,36)>



Binary Radix Heap

Radix Heap: deleteMin

Function deleteMin: Element

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

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

return result

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

$B[i+1], \ldots, B[K]$: 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

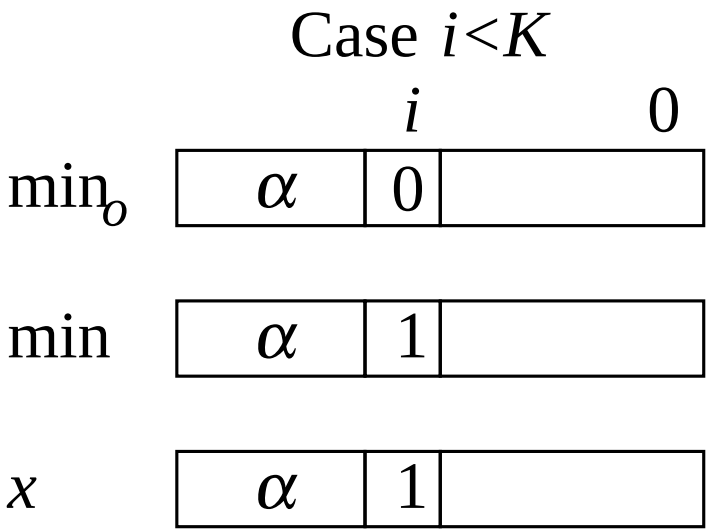
	$min = 10000$		$min = 10010$	
Bucket	$d[v]$ binär	$d[v]$	$d[v]$ binär	$d[v]$
-1	10000	16	10010	18
0	10001	17	10011	19
1	1001*	18,19	-	-
2	101**	20–23	101**	20-23
3	11***	24–25	11***	24-27
4	-	-	-	-

Bucket $B[i]$ bei Änderung von min

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

Wir zeigen nur den Fall $i < K$.

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



Kosten der deleteMin-Operationen

Bucket $B[i]$ finden: $\mathrm{O}(i)$

Elemente aus $B[i]$ verschieben: $\mathrm{O}(|B[i]|)$

Insgesamt $\mathrm{O}(K + |B[i]|)$ falls $i \geq 0$, $\mathrm{O}(1)$ 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

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

$$
\begin{aligned}
T_{\text{Dijkstra}} &= \mathrm{O}(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))) \\
T_{\text{DijkstraRadix}} &= \mathrm{O}(m + n \cdot (K + K)) = \mathrm{O}(m + n \cdot \log C)
\end{aligned}
$$

Lineare Laufzeit für zufällige Kantengewichte

Vorher gesehen: Dijkstra mit Bucket-Queues hat lineare Laufzeit für dichte Graphen ($m > n \log n \log\log n$)

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

Jetzt: Dijkstra mit Radix-Heaps hat lineare Laufzeit ($\mathrm{O}(m+n)$) falls Kantenkosten identisch uniform verteilt in $0..C$

–wir brauchen nur eine kleine Änderung im Algorithmus

Änderung im Algorithmus für zufällige Kantengewichte

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

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

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

$\Longrightarrow$ 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 $i < \log c^{in}_{\min}(v)$

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

Kosten von Verschiebungen sind dann insgesamt höchstens

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

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

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

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

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 $O(n^2 m)$

Lösung 2: Knotenpotentiale

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

Knotenpotentiale

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

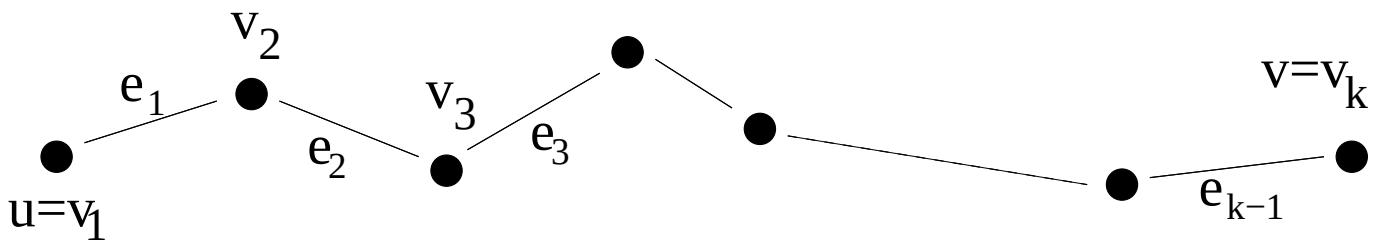
Mit Hilfe der Potentiale definieren wir reduzierte Kosten $\overline{c}(e)$ für Kante $e = (u, v)$ als

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

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

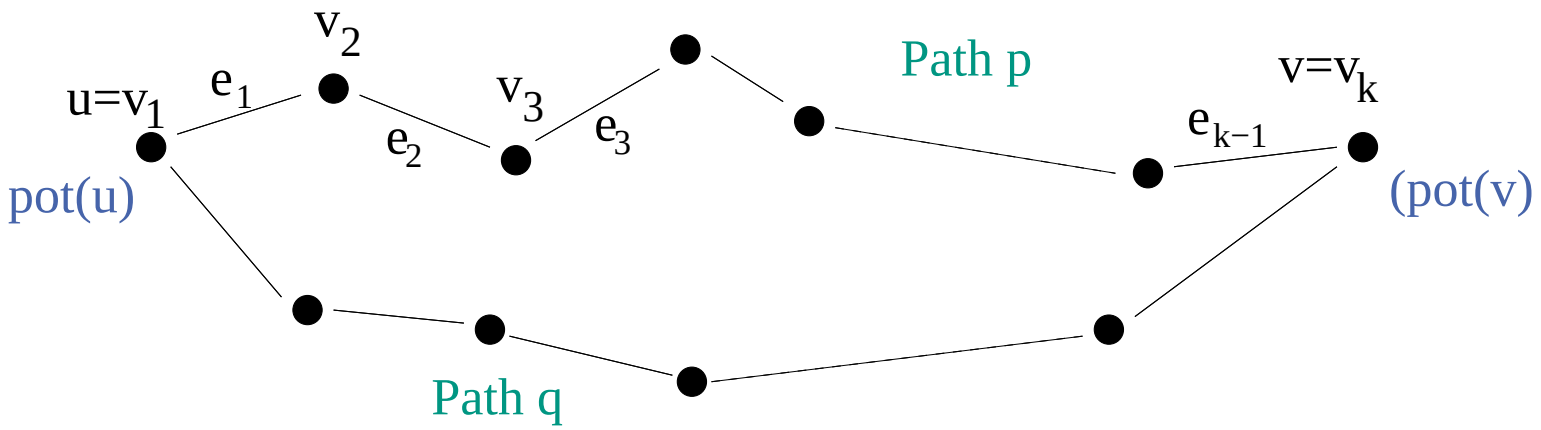
$$\begin{aligned}
\overline{c}(p) &= \sum_{i=1}^{k-1} \overline{c}(e_i) = \sum_{i=1}^{k-1} (\mathrm{pot}(v_i) + c(e_i) - \mathrm{pot}(v_{i+1})) \\
&= \mathrm{pot}(v_1) + \sum_{i=1}^{k-1} c(e_i) - \mathrm{pot}(v_k) \\
&= \mathrm{pot}(v_1) + c(p) - \mathrm{pot}(v_k).
\end{aligned}$$

Knotenpotentiale

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

$$\overline{c}(p) = \text{pot}(v_0) + c(p) - \text{pot}(v_k).$$

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



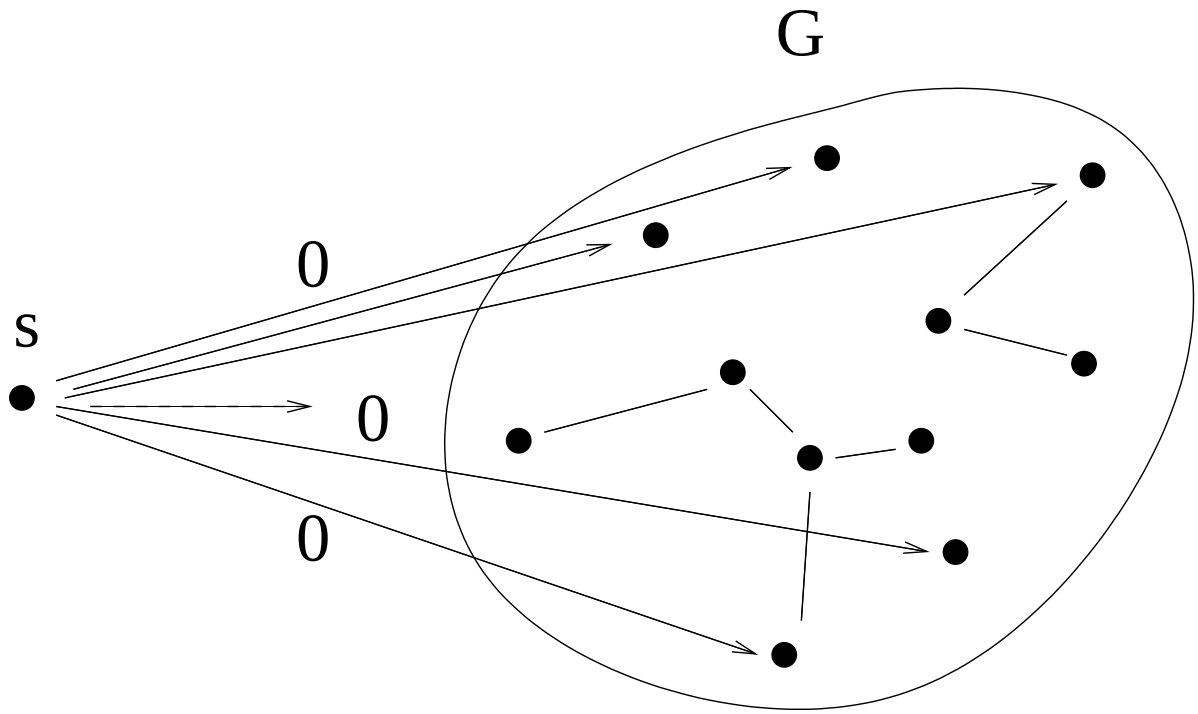
Definition: $\mu(u,v) =$ kürzeste Distanz von u nach v

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

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

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) // $O(n)$

pot:= μ:= BellmanFordSSSP(s,c) // $O(nm)$

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

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

// zurück zur ursprünglichen Kostenfunktion

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

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

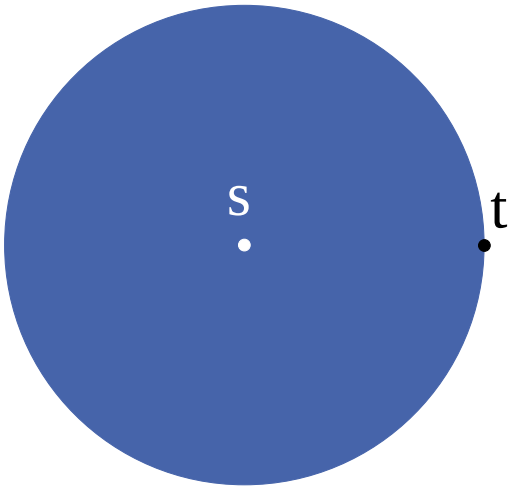
Laufzeit

- s hinzufügen: $\mathrm{O}(n)$
- Postprocessing: $\mathrm{O}(m)$ (zurück zu den ursprünglichen Kosten)
- n mal Dijkstra dominiert

Laufzeit $\mathrm{O}(n(m + n\log n)) = \mathrm{O}\left(nm + n^2 \log n\right)$

Distanz zu einem Zielknoten t

Was machen wir, wenn wir nur die Distanz von s zu einem bestimmten Knoten t wissen wollen?



Trick 0:

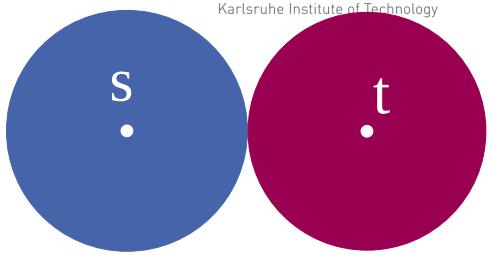
Dijkstra hört auf, wenn t aus Q entfernt wird

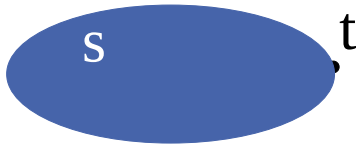
Spart “im Durchschnitt” Hälfte der Scans

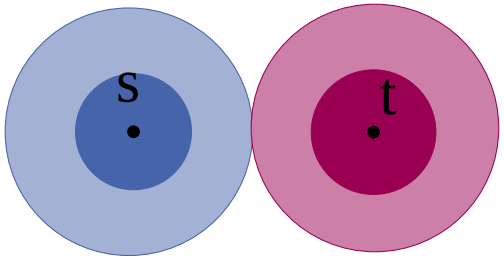
Frage: Wieviel spart es (meist) beim Europa-Navi?

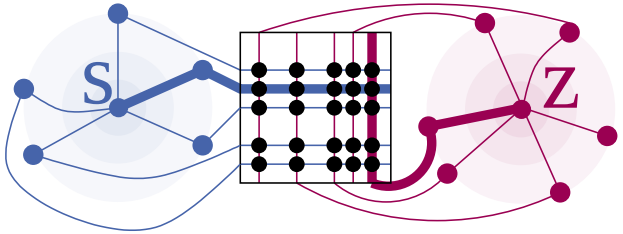
Ideen für Routenplanung

- Vorwärts + Rückwärtssuche
- **Zielgerichtete Suche**
- Hierarchien ausnutzen
- Teilabschnitte tabellieren



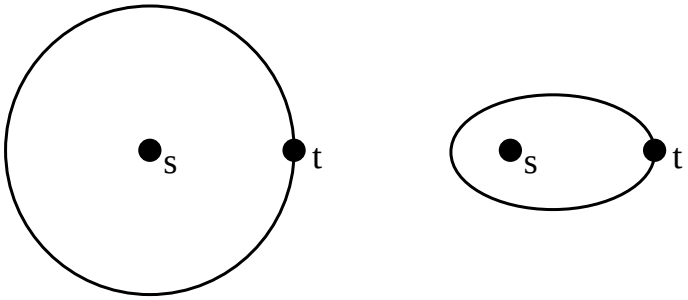






A^*-Suche

Idee: suche “in die Richtung von t”

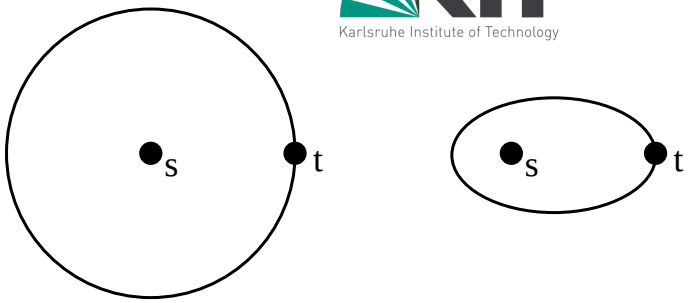


Annahme: Wir kennen eine Funktion $f(v)$ die $\mu(v,t)$ schätzt $\forall v$

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

[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 pot$(v) = f(v)$ und $\overline{c}(u,v) = c(u,v) + f(v) - f(u)$

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

Dann gilt: $\overline{c}(u,v) = c(u,v) + \mu(v,t) - \mu(u,t) = 0$ falls (u,v) auf

dem kürzesten Pfad von s nach t liegt.

$$s \to \cdots \to \underbrace{u \xrightarrow{c(u,v)} \overbrace{v \to \cdots \to t}^{\mu(v,t)}}_{\mu(u,t)}$$

Also scannt Dijkstra nur die Knoten auf diesem Pfad!

Benötigte Eigenschaften von $f(v)$

- Konsistenz (reduzierte Kosten nicht negativ):
  $c(e) + f(v) \geq f(u)\ \forall e = (u,v)$
- $f(v) \leq \mu(v,t)\ \forall v \in V$. Dann gilt $f(t) = 0$ und wir können 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: $f(v) =$ euklidischer Abstand $||s-t||_2$

bringt deutliche aber nicht überragende Beschleunigung

Fahrzeit: $\dfrac{||s-t||_2}{\text{Höchstgeschwindigkeit}}$

praktisch nutzlos

Noch besser aber mit Vorberechnung: Landmarks

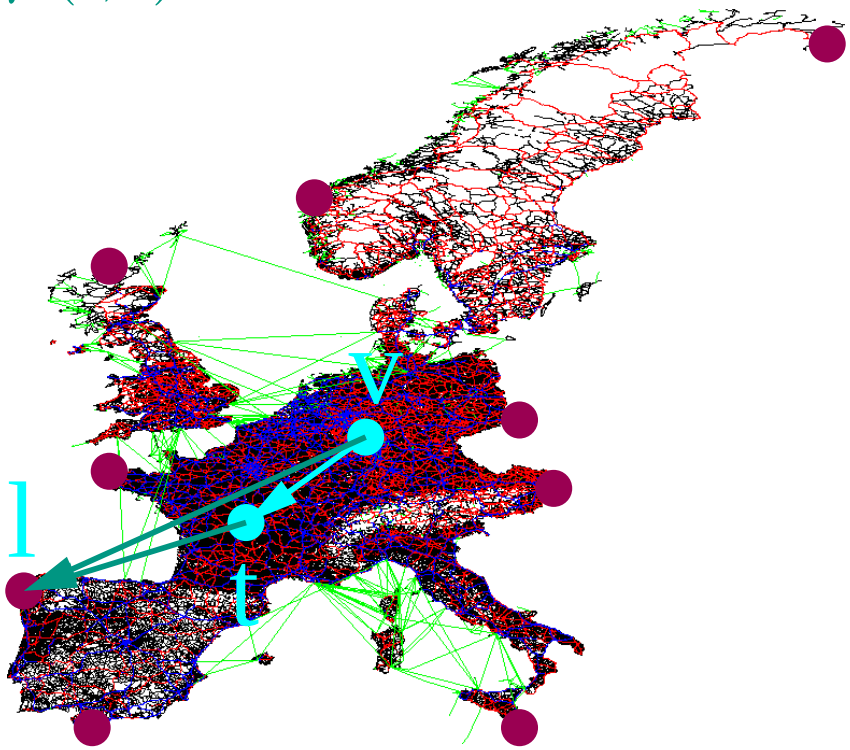
Landmarks [Goldberg Harrelson 2003]

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

5 Maximum Flows and Matchings

[mit Kurz Mehlhorn, Rob van Stee]

Folien auf Englisch

Literatur:

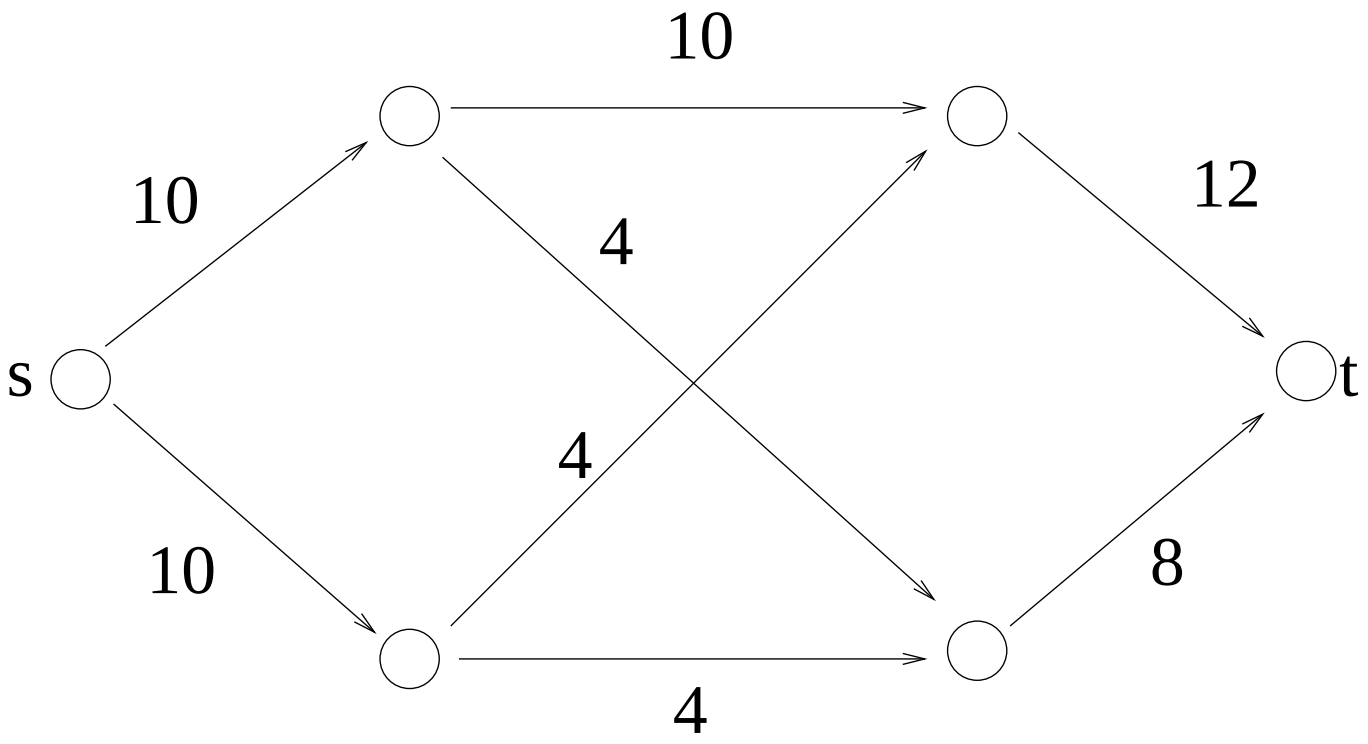
[Mehlhorn / Näher, The LEDA Platform of Combinatorial and Geometric Computing, Cambridge University Press, 1999]

`http://www.mpi-inf.mpg.de/~mehlhorn/ftp/LEDAbook/Graph_alg.ps`

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

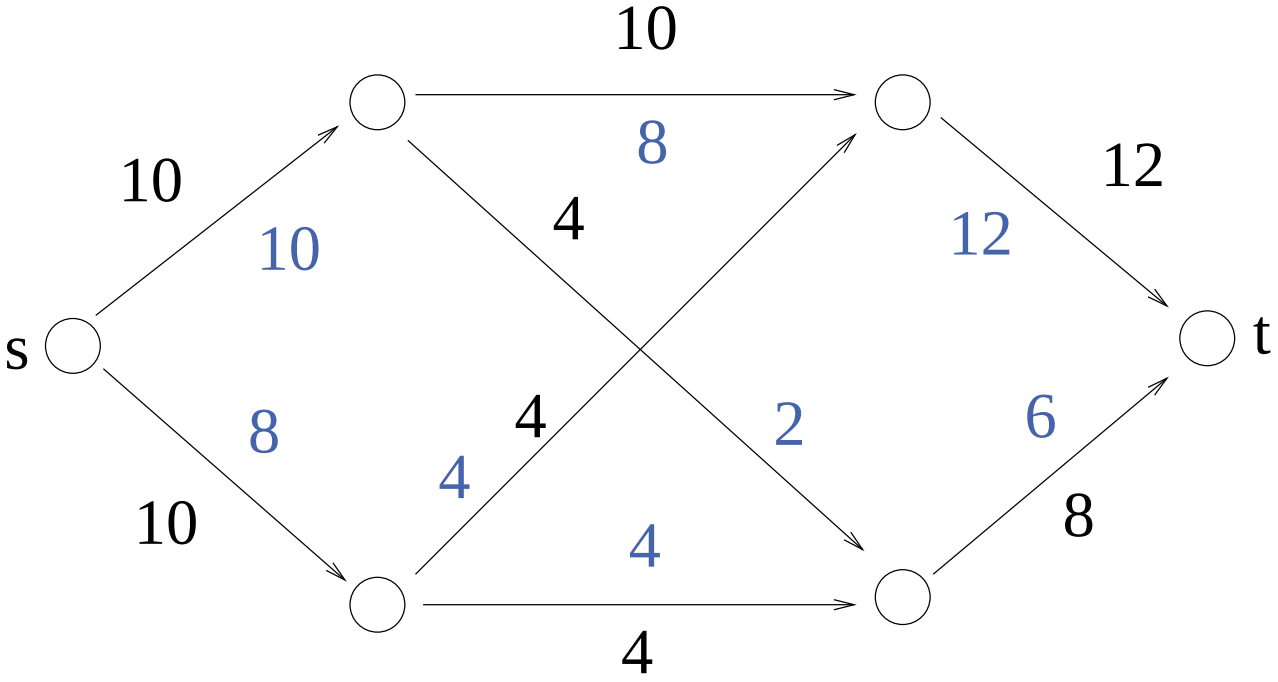
Definitions: Network

- 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 $\mathbf{val}(f) =$ total outgoing flow from $s =$ total flow going into t
- Goal: find a flow with maximum value

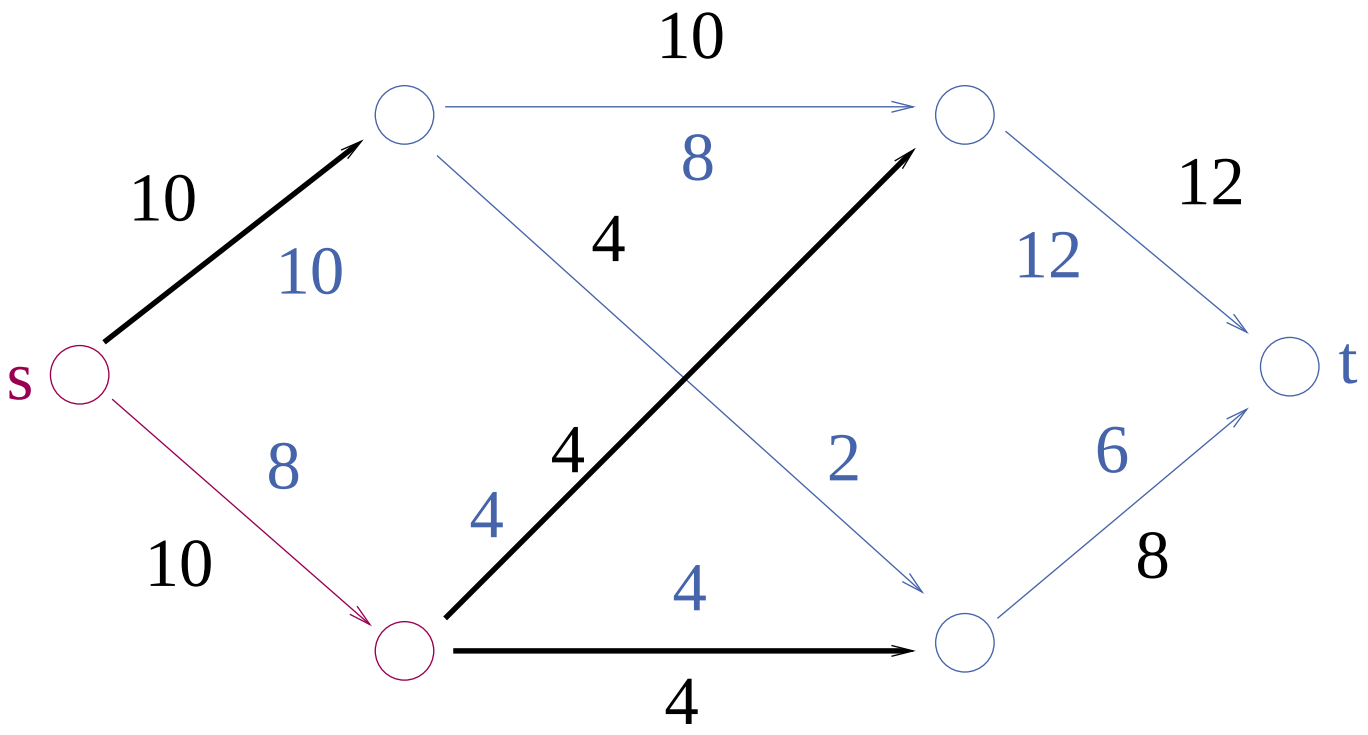


Definitions: (Minimum) s-t Cuts

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:

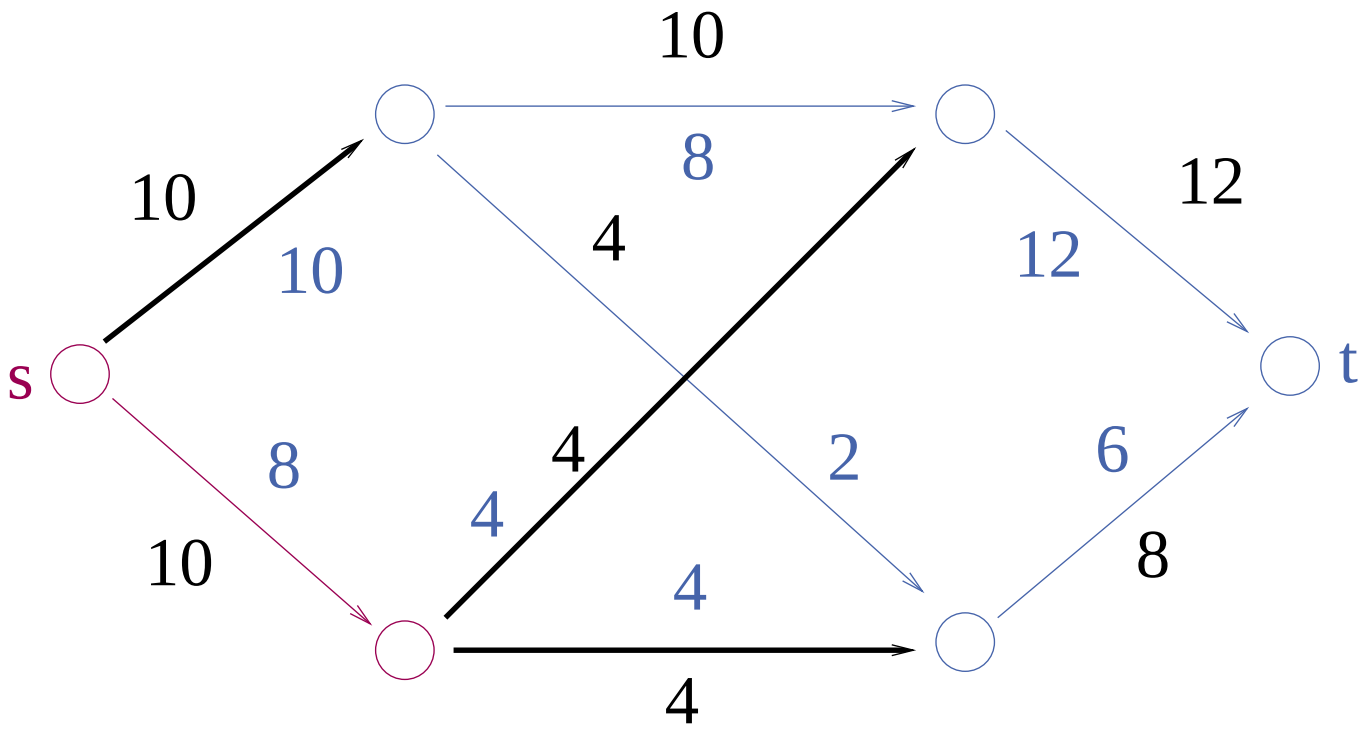
$$\sum \{c_{(u,v)} : u \in S, v \in T\}$$



Duality Between Flows and Cuts

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

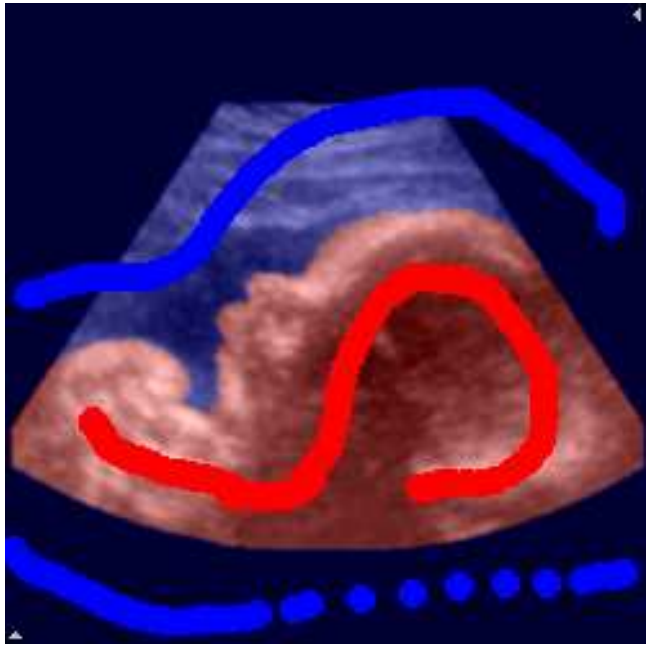
Value of an s-t max-flow $=$ minimum capacity of an s-t cut.



Proof: later

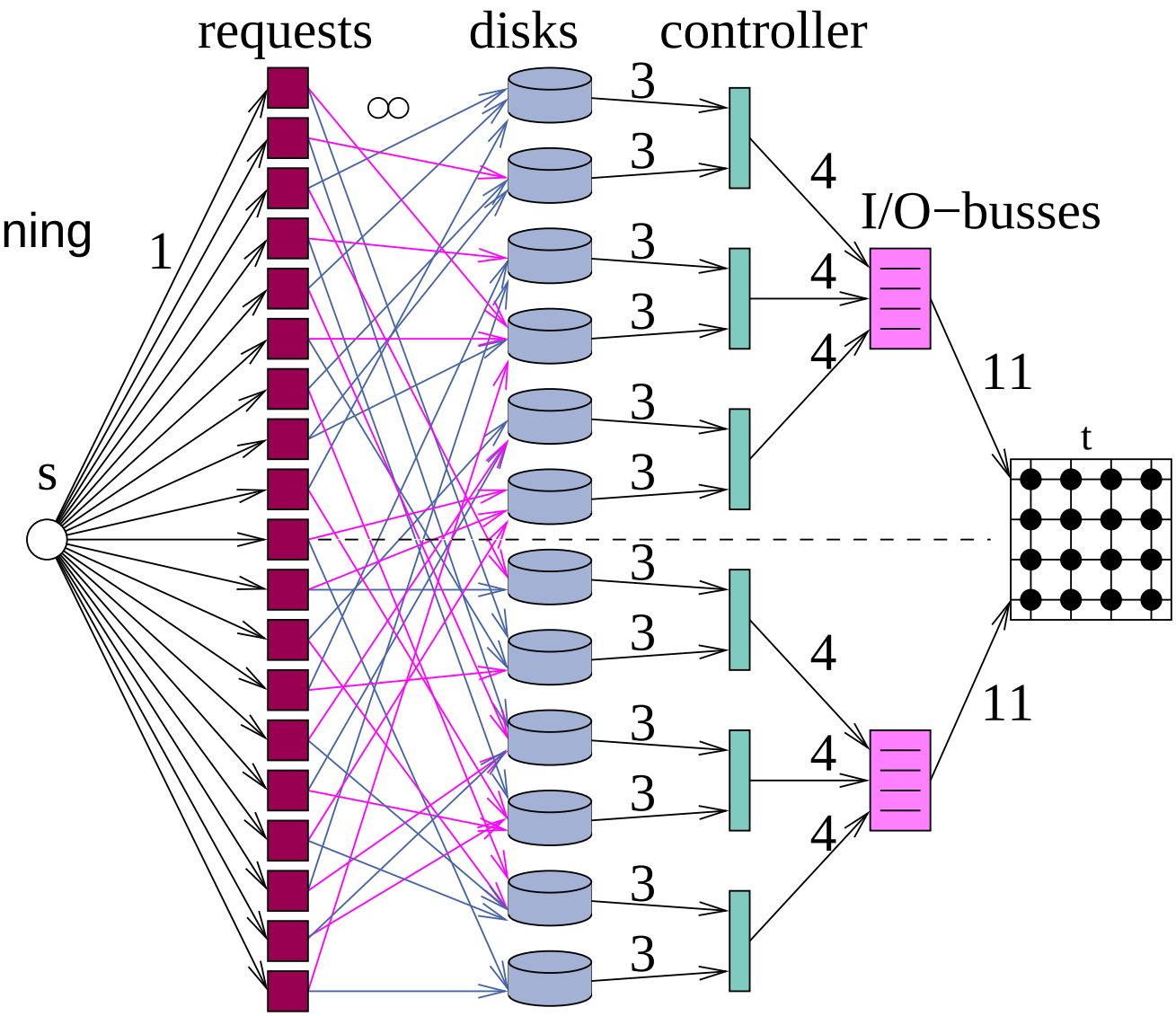
Applications

- 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	$O(m^2 n)$
1970	Dinic	$O(mn^2)$
1973	Dinic-Gabow	$O(mn \log U)$
1974	Karzanov	$O(n^3)$
1977	Cherkassky	$O(n^2\sqrt{m})$
1980	Galil-Naamad	$O(mn \log^2 n)$
1983	Sleator-Tarjan	$O(mn \log n)$

$n =$ number of nodes

$m =$ number of arcs

$U =$ largest capacity

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	$O(mn+n^{2+e})$
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	$O(\min\{m^{1/2},n^{2/3}\}m\log(n^2/m)\log U)$

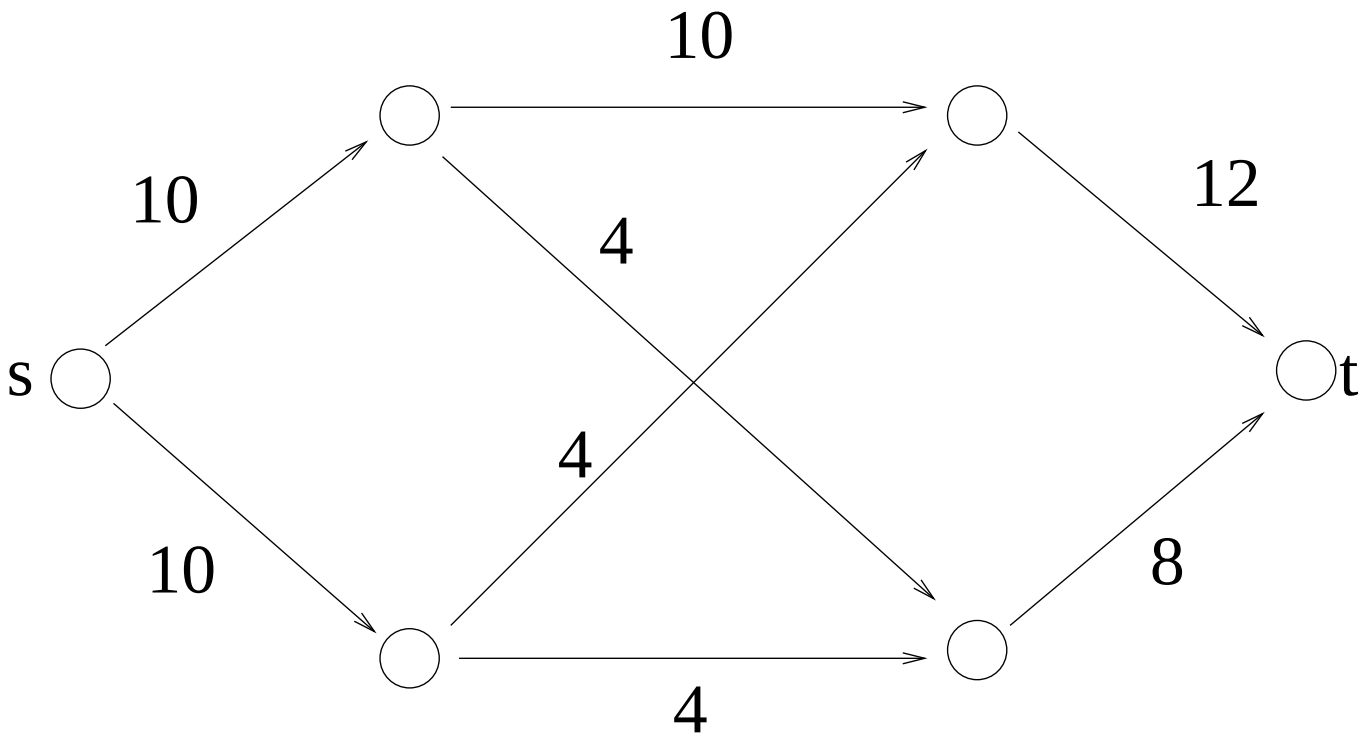
Augmenting Paths (Rough Idea)

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

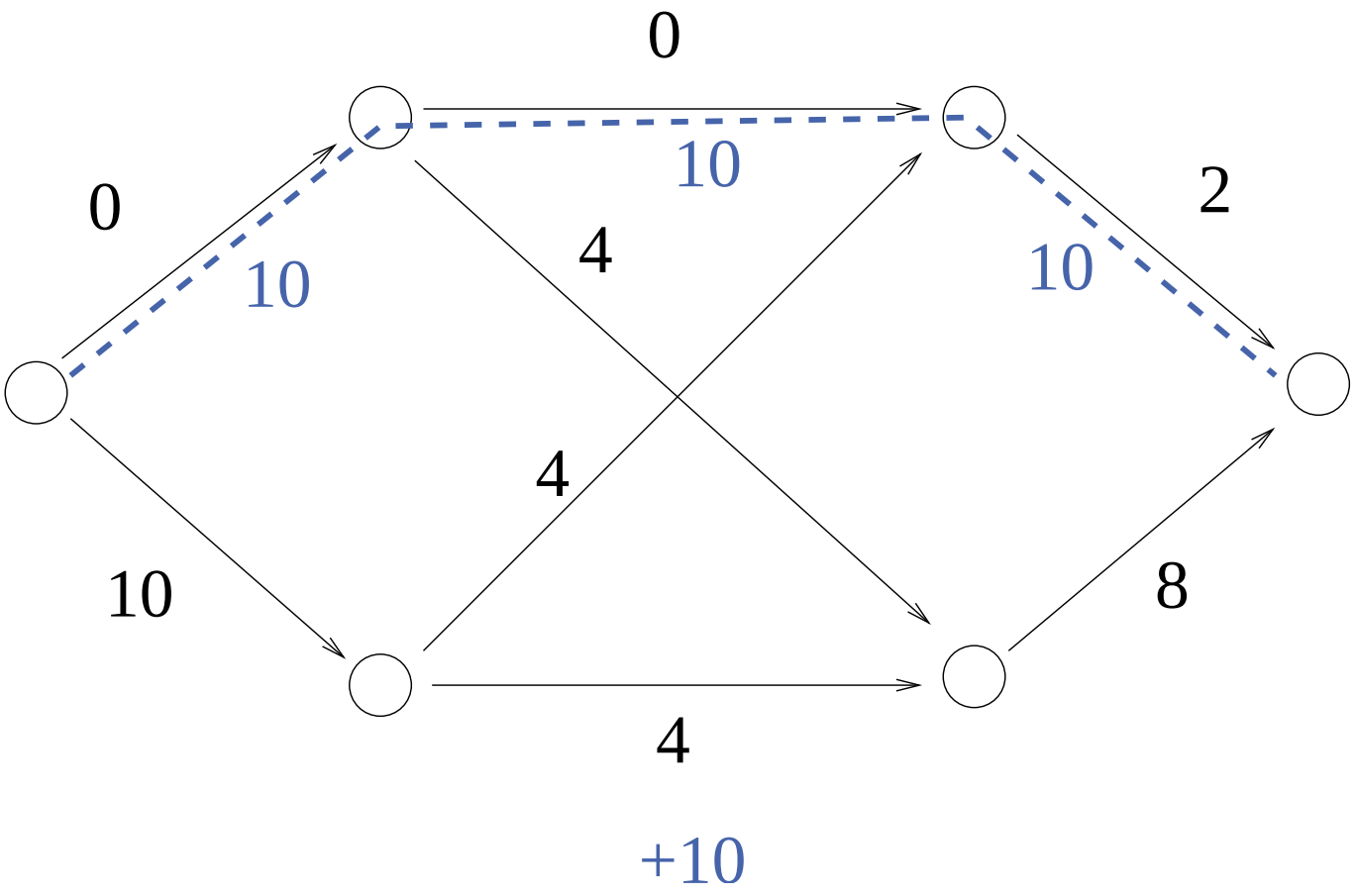
On this path, saturate the edge with the smallest spare capacity

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

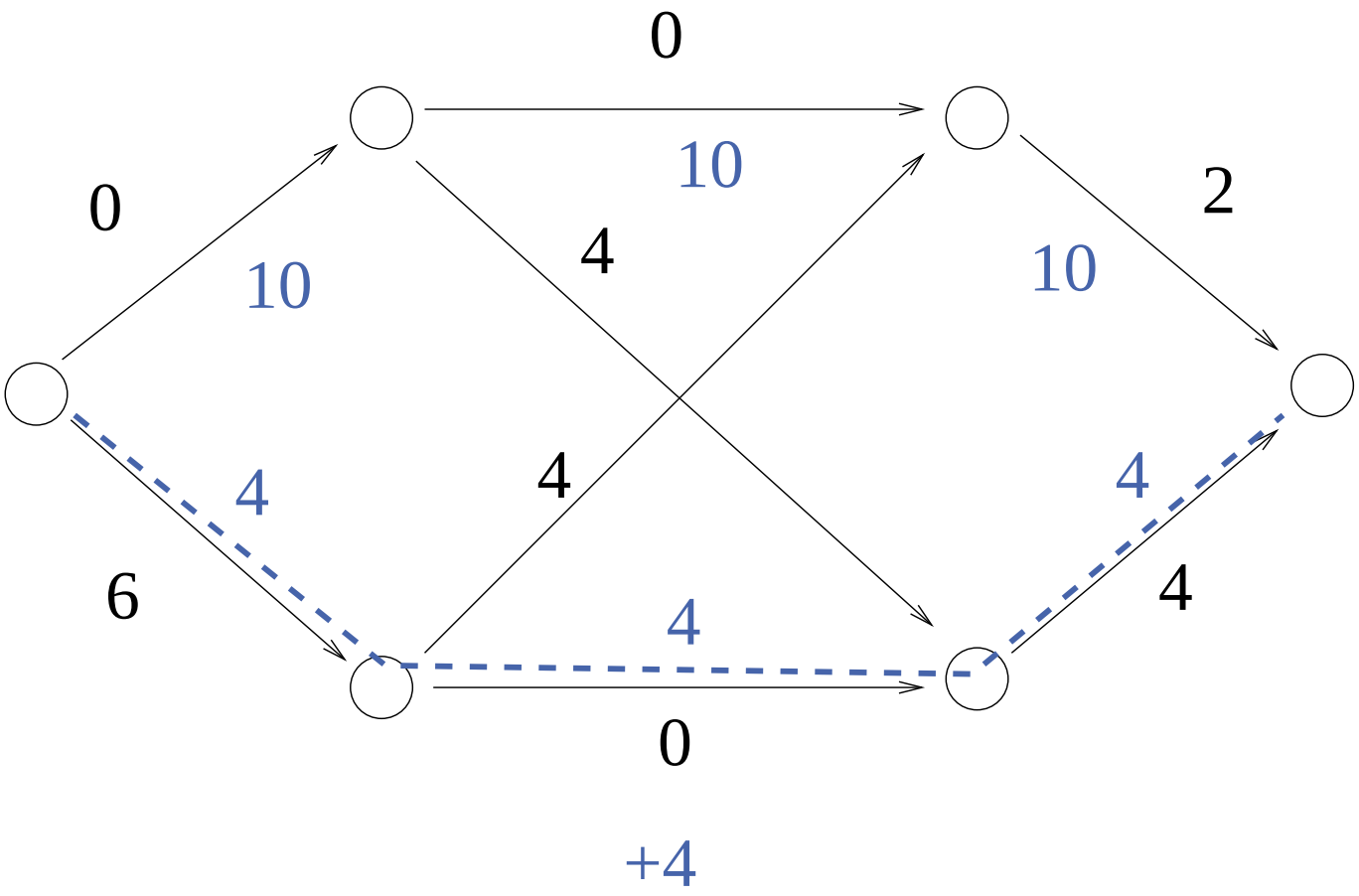
Example



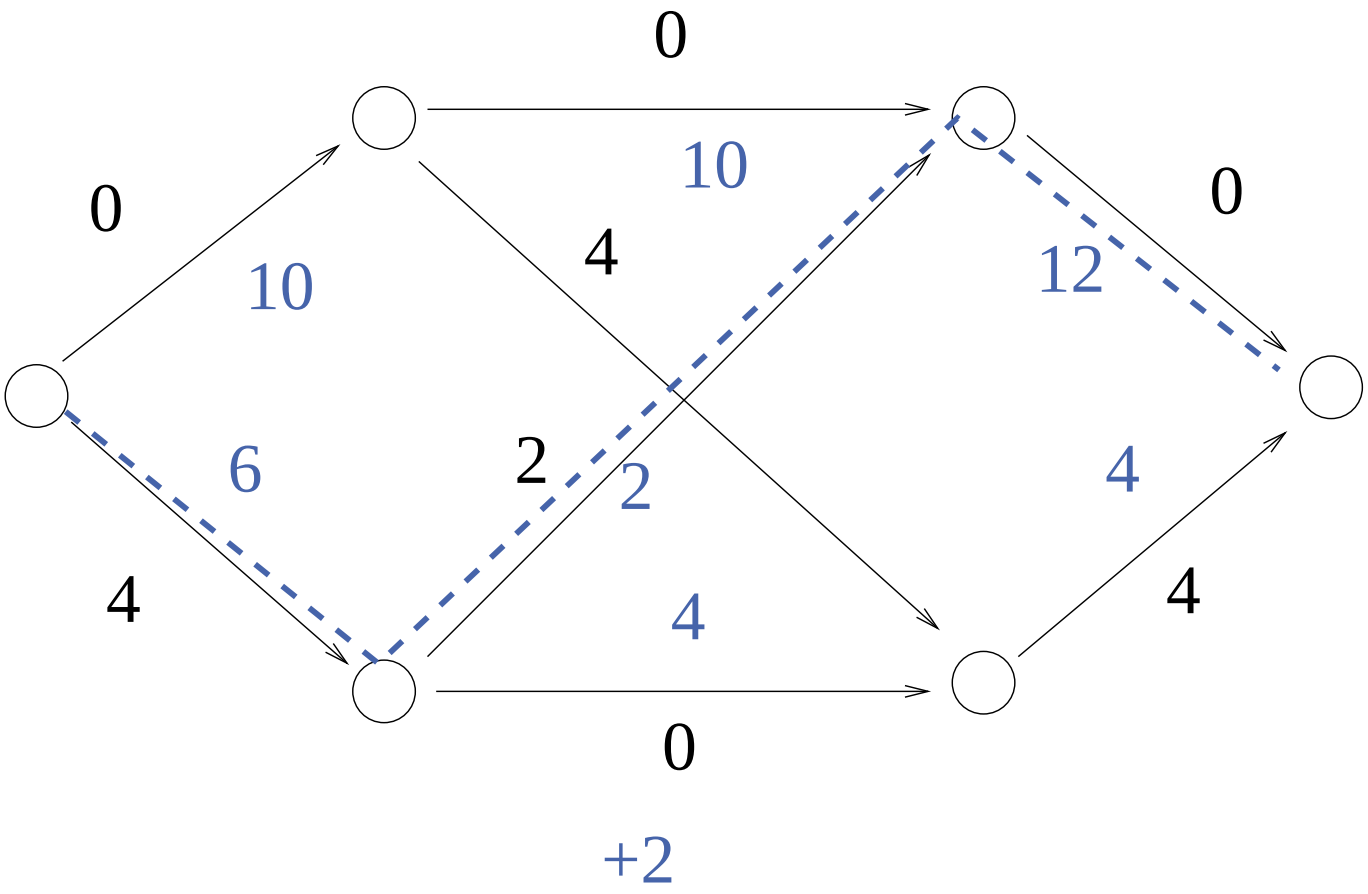
Example



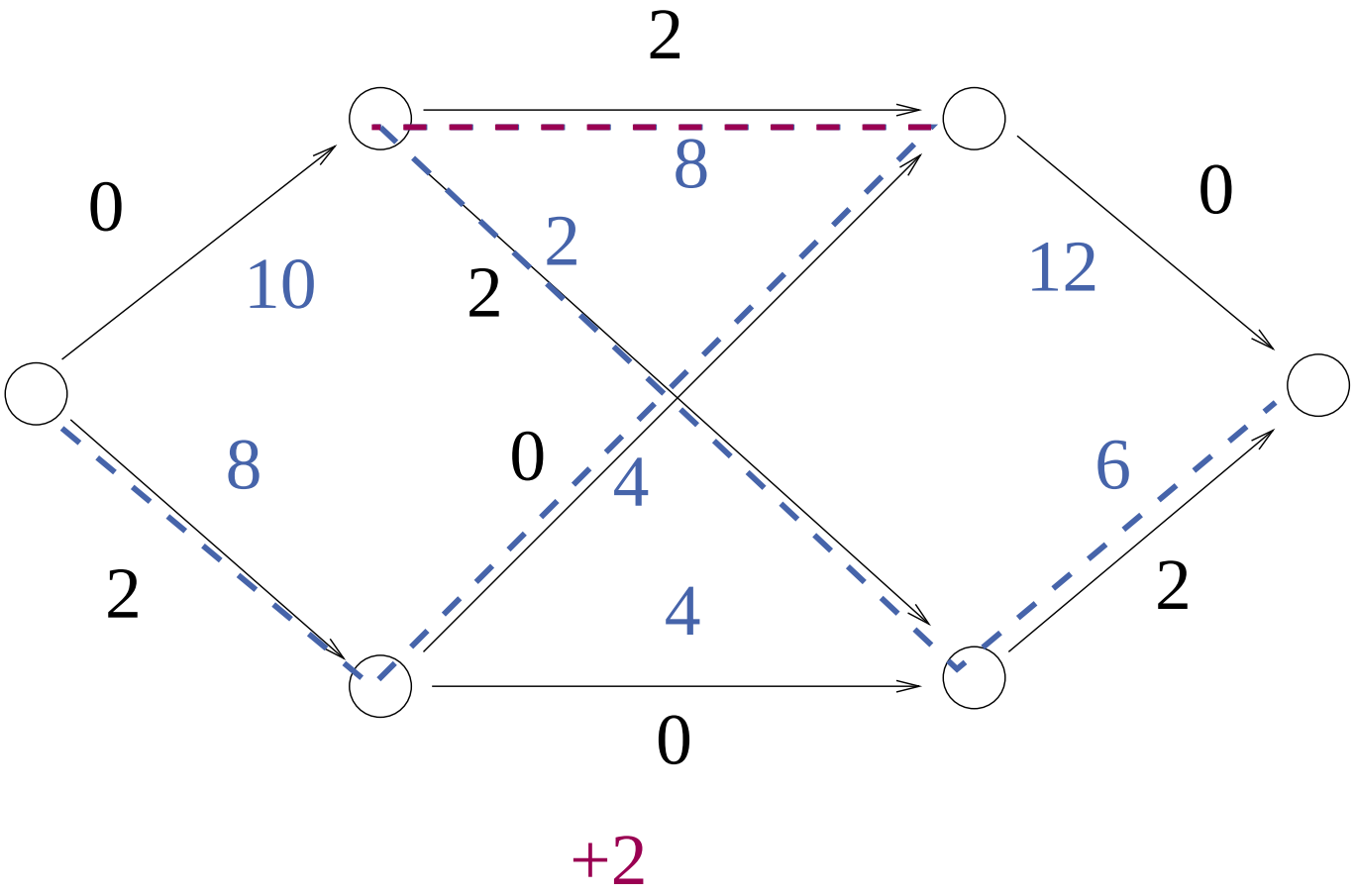
Example



Example



Example

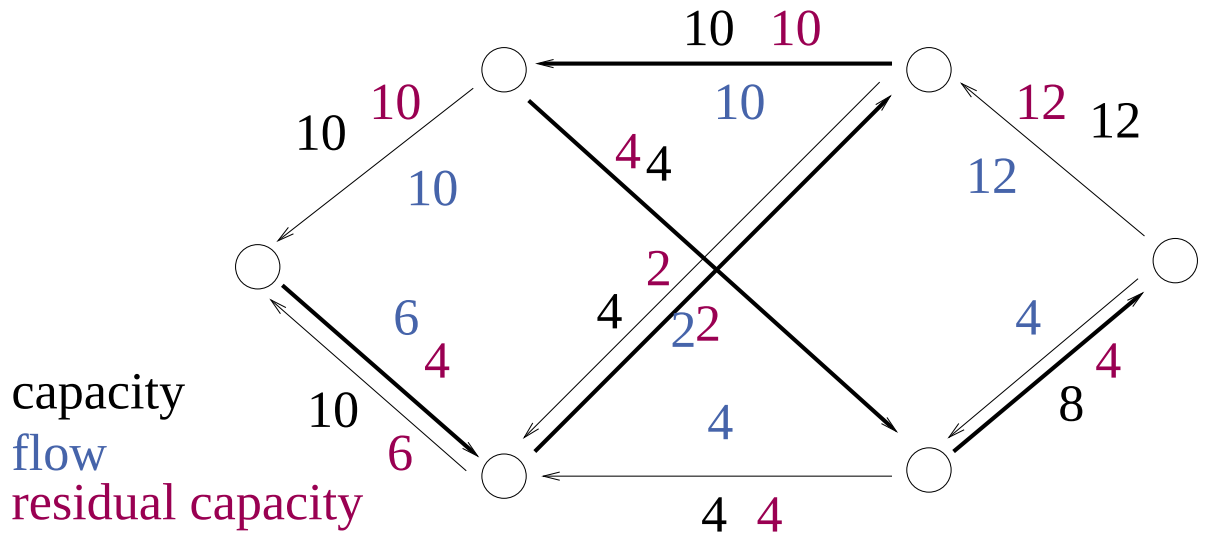


Residual Graph

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($G = (V,E), s, t, \mathrm{c} : E \to \mathbb{N}) : E \to \mathbb{N}$

$f := 0$

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

augment f along p

return f

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

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 $(S, V \setminus S)$ with

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

Some Basic Observations

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

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

Lemma 2: $\forall (u,v) \in E : c^f_{(u,v)} = 0 \Rightarrow f_{(v,u)} = 0$

Ford Fulkerson – Correctness

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

Consider cut $(S, V \setminus S)$ with S:= $\{v \in V : v \text{ reachable from } s \text{ in } G_f\}$.

Observation: $\forall (u,v) \in E \cap S \times T : c_e^f = 0$ and hence $f_{(v,u)} = 0$

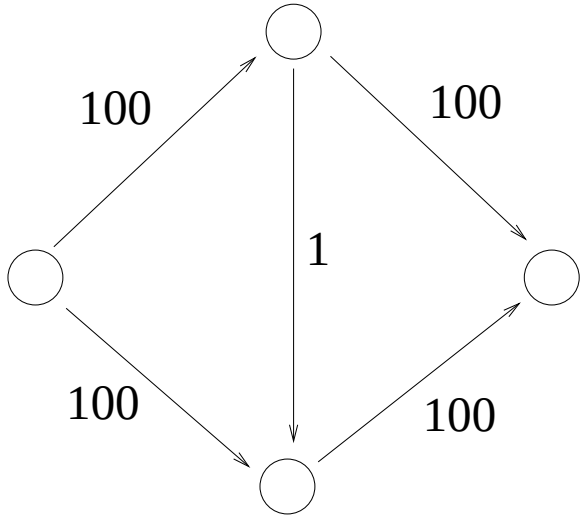
Lemma 2.

Now, by Lemma 1,

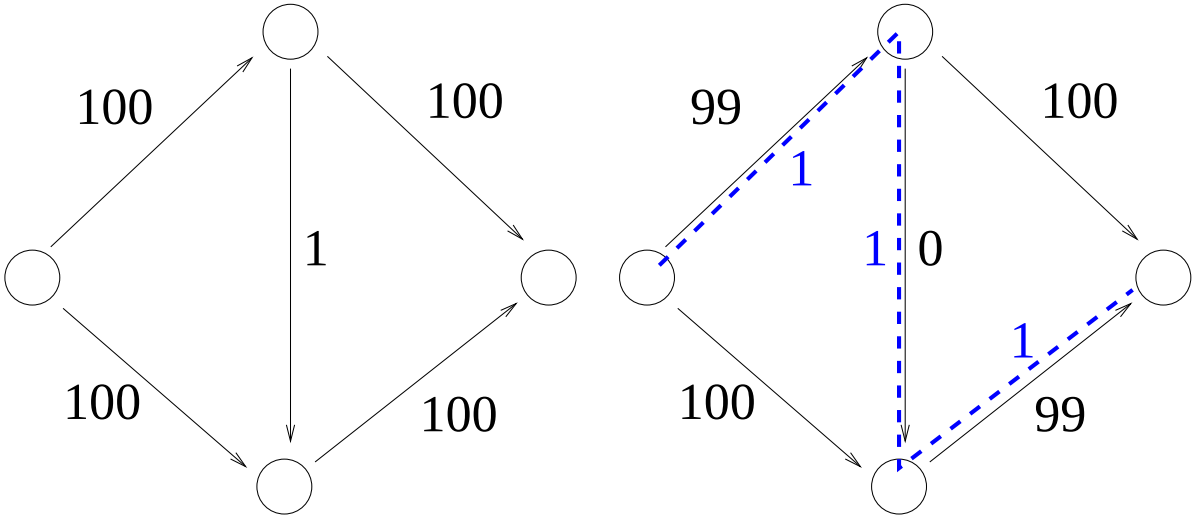
$$\begin{aligned}\mathbf{val}(f) &= \sum_{e \in E \cap S \times T} f_e - \sum_{e \in E \cap T \times S} f_e \\ &= \sum_{e \in E \cap S \times T} f_e = \text{cut capacity} \\ &\geq \text{max flow}\end{aligned}$$

Corollary: max flow = min cut

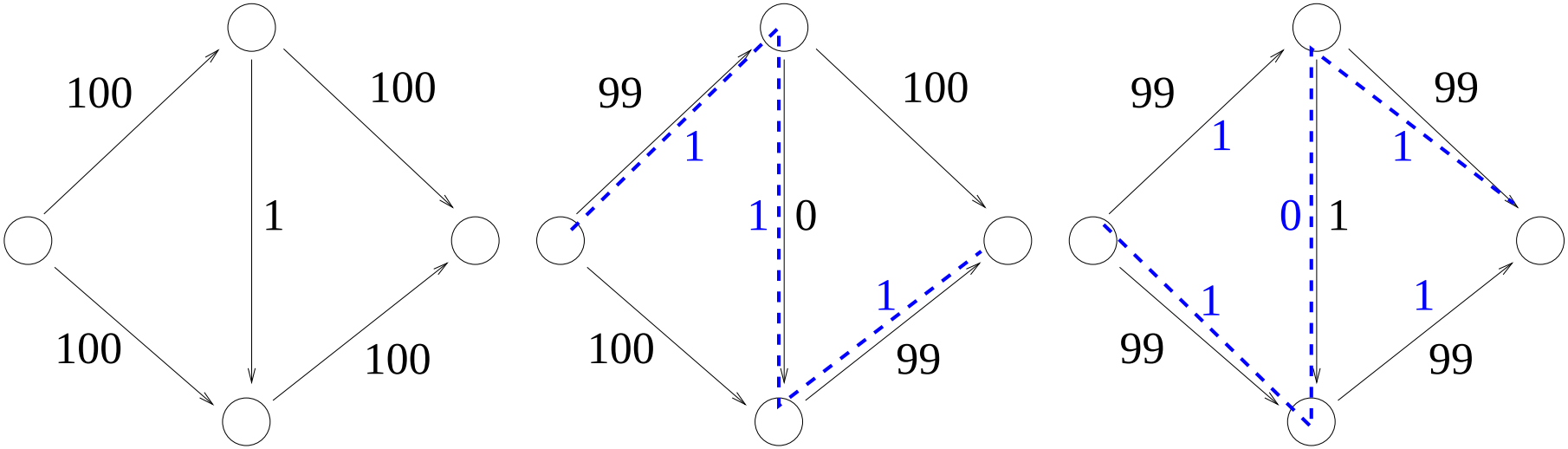
A Bad Example for Ford Fulkerson



A Bad Example for Ford Fulkerson



A Bad Example for Ford Fulkerson

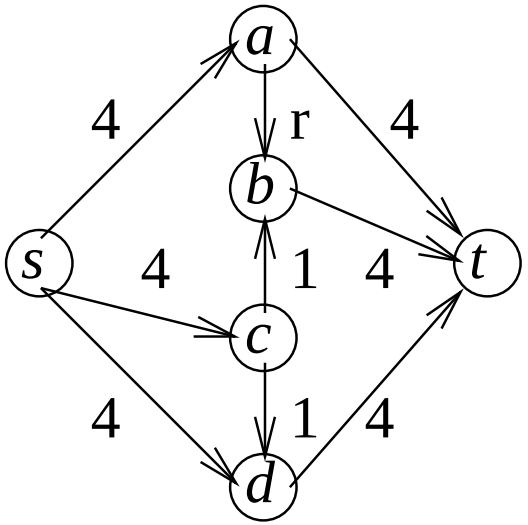


An Even Worse Example for Ford Fulkerson

[U. Zwick, TCS 148, p. 165–170, 1995]

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

Consider the graph



And the augmenting paths

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

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

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

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

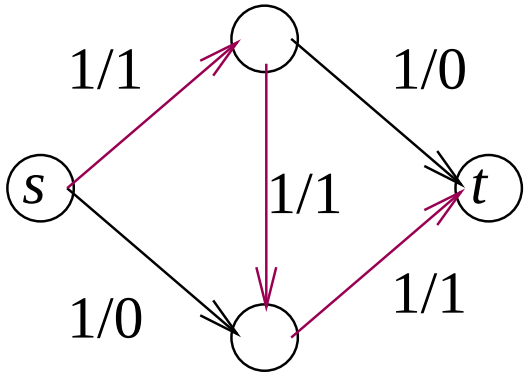
The sequence of augmenting paths $p_0(p_1, p_2, p_1, p_3)^*$ is an infinite sequence of positive flow augmentations.

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

Blocking Flows

f_b is a blocking flow in H if

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



Dinitz Algorithm

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

$f := 0$

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

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

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

find a blocking flow f_b in L_f

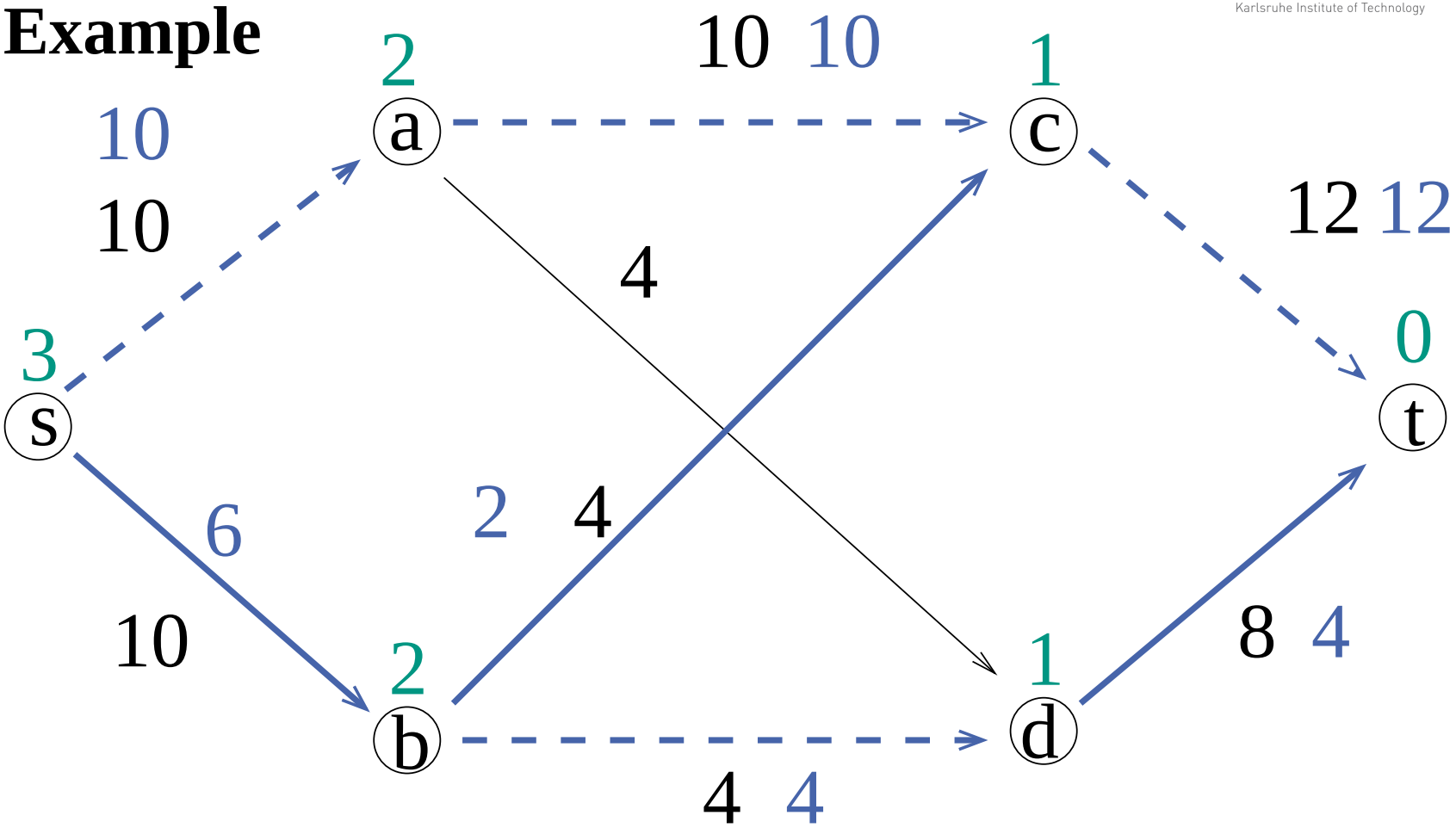
augment f += f_b

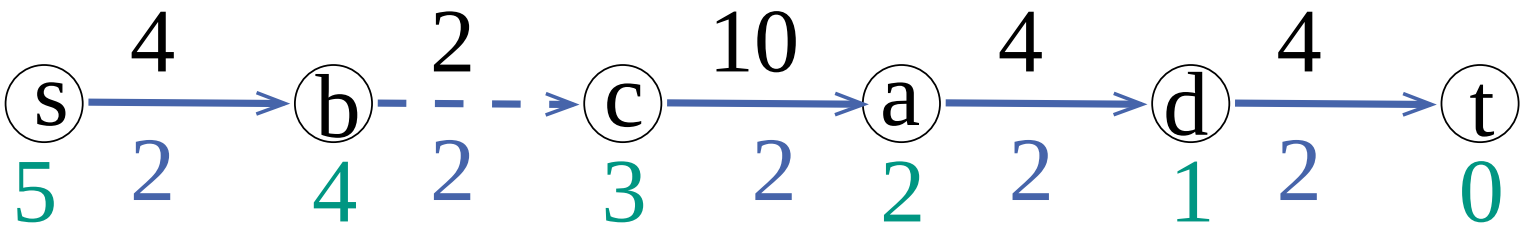
return f

Dinitz – Correctness

analogous to Ford-Fulkerson

Example


2
10 10
1
10
10
12 12
4
3
0
s
t
6
2 4
10
2
1
8 4
b
d
4 4


4
2
10
4
4
s
b
c
a
d
t
5
2
4
2
3
2
2
2
1
2
0

Computing Blocking Flows

Idee: wiederholte DFS nach augmentierenden Pfaden

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

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

loop // Round

v:= p.last()

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

δ:= $\min\{c(e) - f_b(e) : e \in p\}$

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

$f_b(e)$+=δ

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

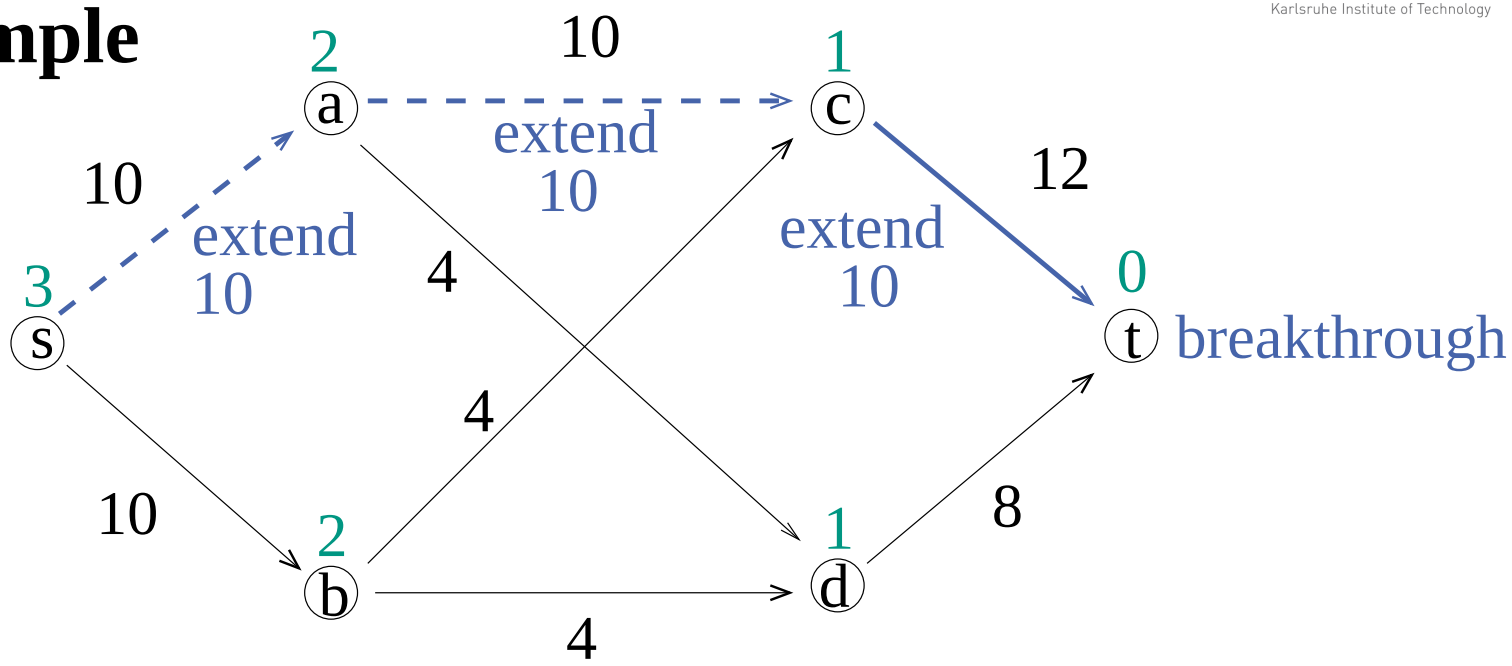
p:= $\langle s \rangle$

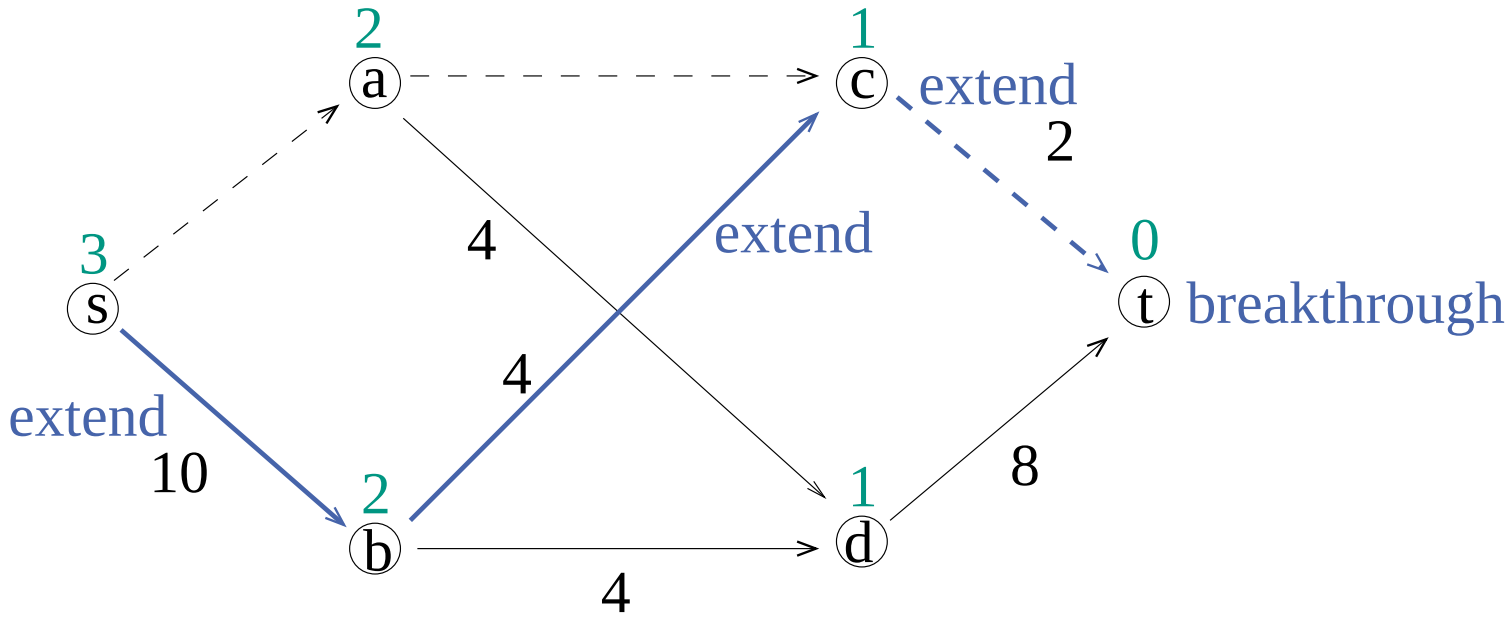
else if $\exists e = (v,w) \in E$ **then** p.pushBack(w) // extend

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

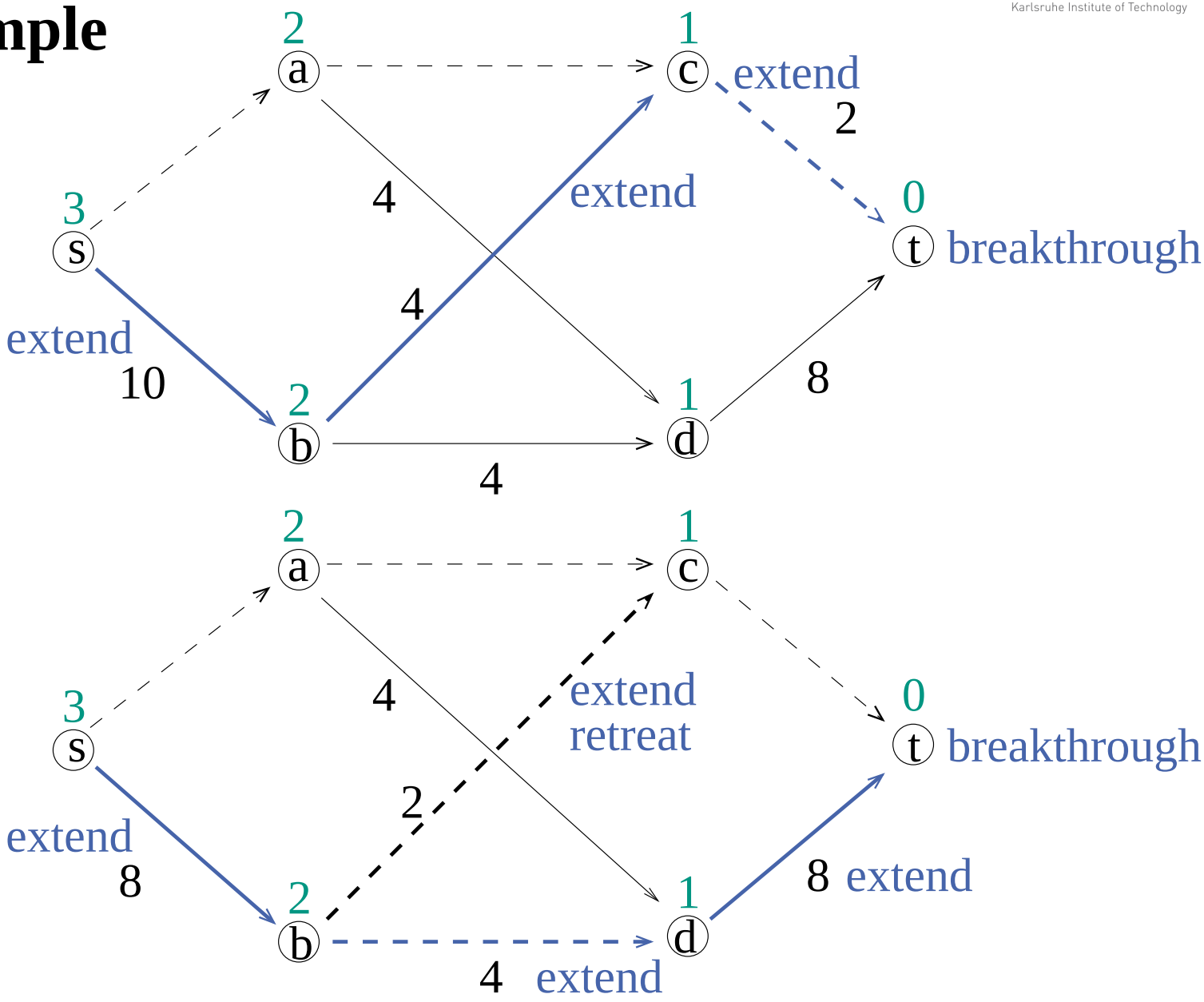
else delete the last edge from p in p and E // retreat

Example

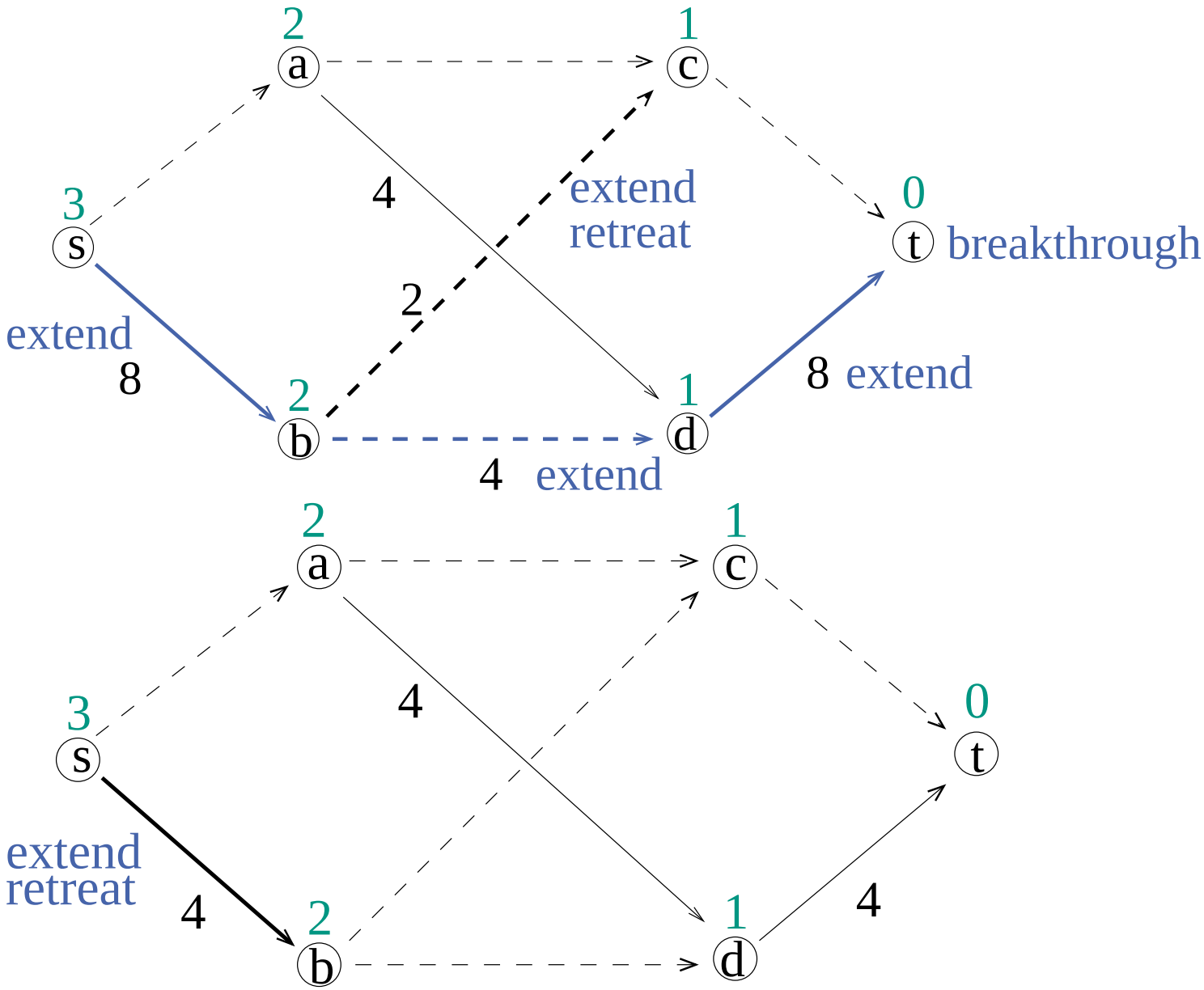

2
a
10
1
c
extend
10
10
extend
10
12
extend
10
3
s
4
0
t
breakthrough
4
10
2
b
1
d
8
4


2
a
1
c
extend
2
3
s
4
extend
0
t
breakthrough
4
extend
10
2
b
1
d
8
4

Example


2
a
1
c
extend
2
3
s
4
extend
0
t
breakthrough
4
extend
10
2
b
1
d
8
4
2
a
1
c
3
s
4
extend
retreat
0
t
breakthrough
2
extend
8
2
b
1
d
8 extend
4 extend

Example


2
a
1
c
3
s
4
extend
retreat
0
t
breakthrough
2
extend
8
2
b
1
d
8
extend
4
extend
2
a
1
c
3
s
4
0
t
extend
retreat
4
2
b
1
d
4

Blocking Flows Analysis 1

- running time $\#_{extends} + \#_{retreats} + n \cdot \#_{breakthroughs}$
- $\#_{breakthroughs} \leq m$ – ≥ 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 2. *$d(s)$ increases by at least one in each round.*

Beweis. not here □

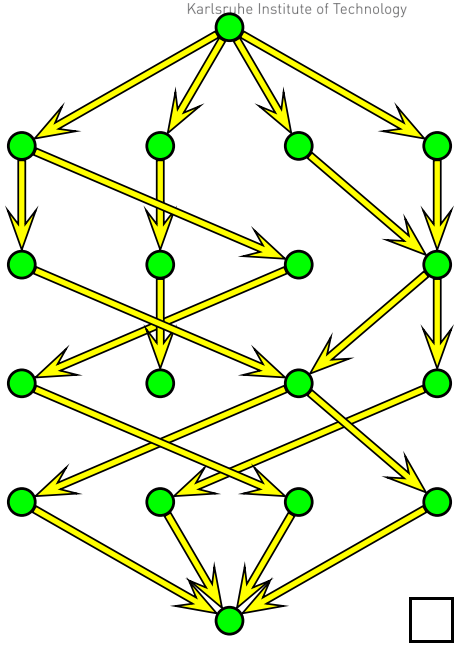
Dinitz Analysis 2

- $\leq n$ 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 3. *At most* $2\sqrt{m}$ *BF computations:*

Beweis. Consider iteration $k = \sqrt{m}$.
Any s-t path contains $\geq k$ edges
FF can find $\leq m/k = \sqrt{m}$ augmenting paths □

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

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

Dinitz Analysis 4 – Unit Networks

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

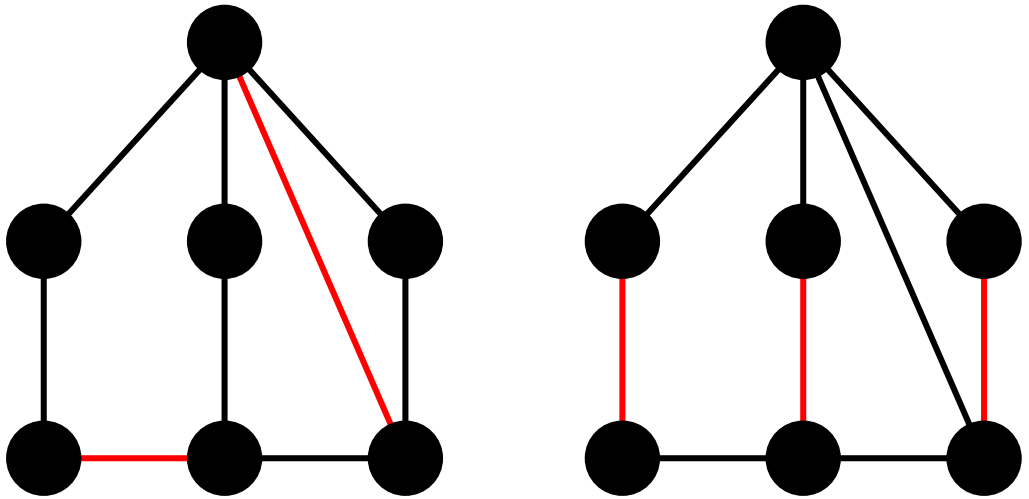
time: $O((m+n)\sqrt{n})$

Matching

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

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

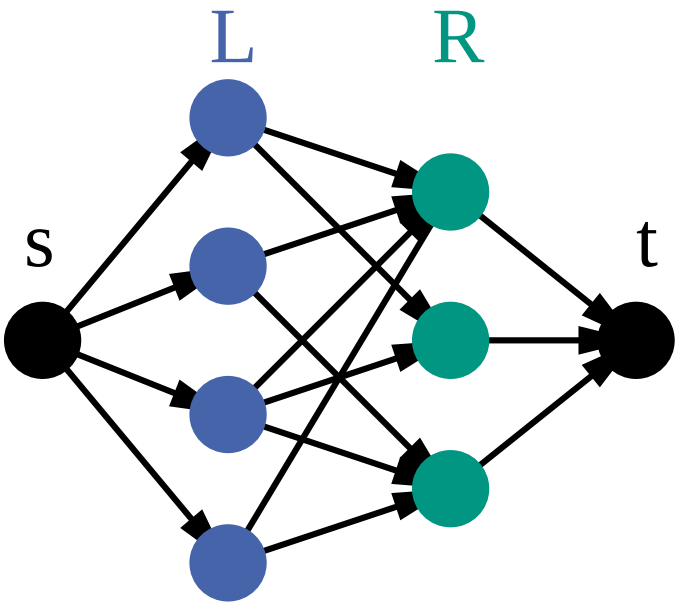
M has maximum cardinality if $\nexists$matching $M' : |M'| > |M|$



Maximum Cardinality Bipartite Matching

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

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

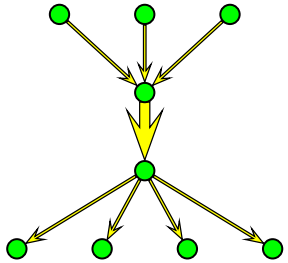


Dinitz algorithm yields $O((n+m)\sqrt{n})$ algorithm

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



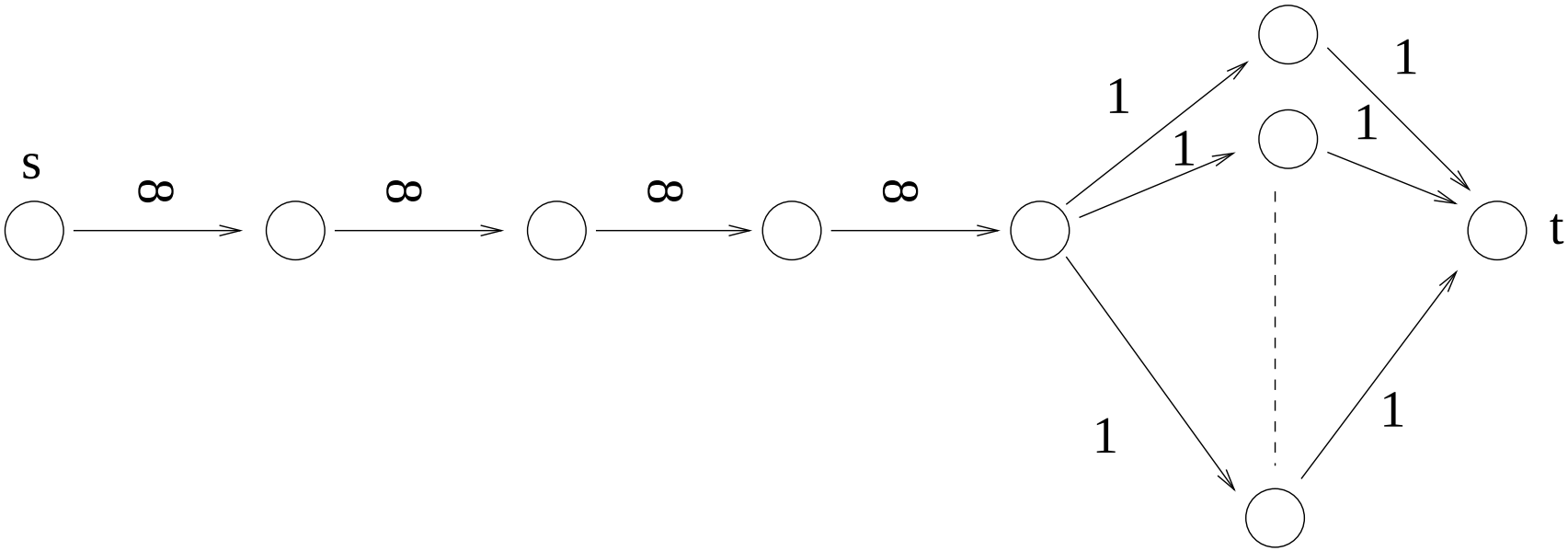
Idea: scale a parameter Δ from small to large

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 push$(e = (v,w), \delta)$

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

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** push$(e, c(e))$ // saturate

$d(s) := n$

$d(v) := 0$ for all other nodes

while $\exists v \in V \setminus \{s,t\}$: excess$(v) > 0$ **do** // active node

if $\exists e = (v,w) \in E_f : d(w) < d(v)$ **then** // eligible edge

choose some $\delta \leq \min\left\{\text{excess}(v), c_e^f\right\}$

push(e, δ) // no new steep edges

else $d(v)$++ // relabel. No new steep edges

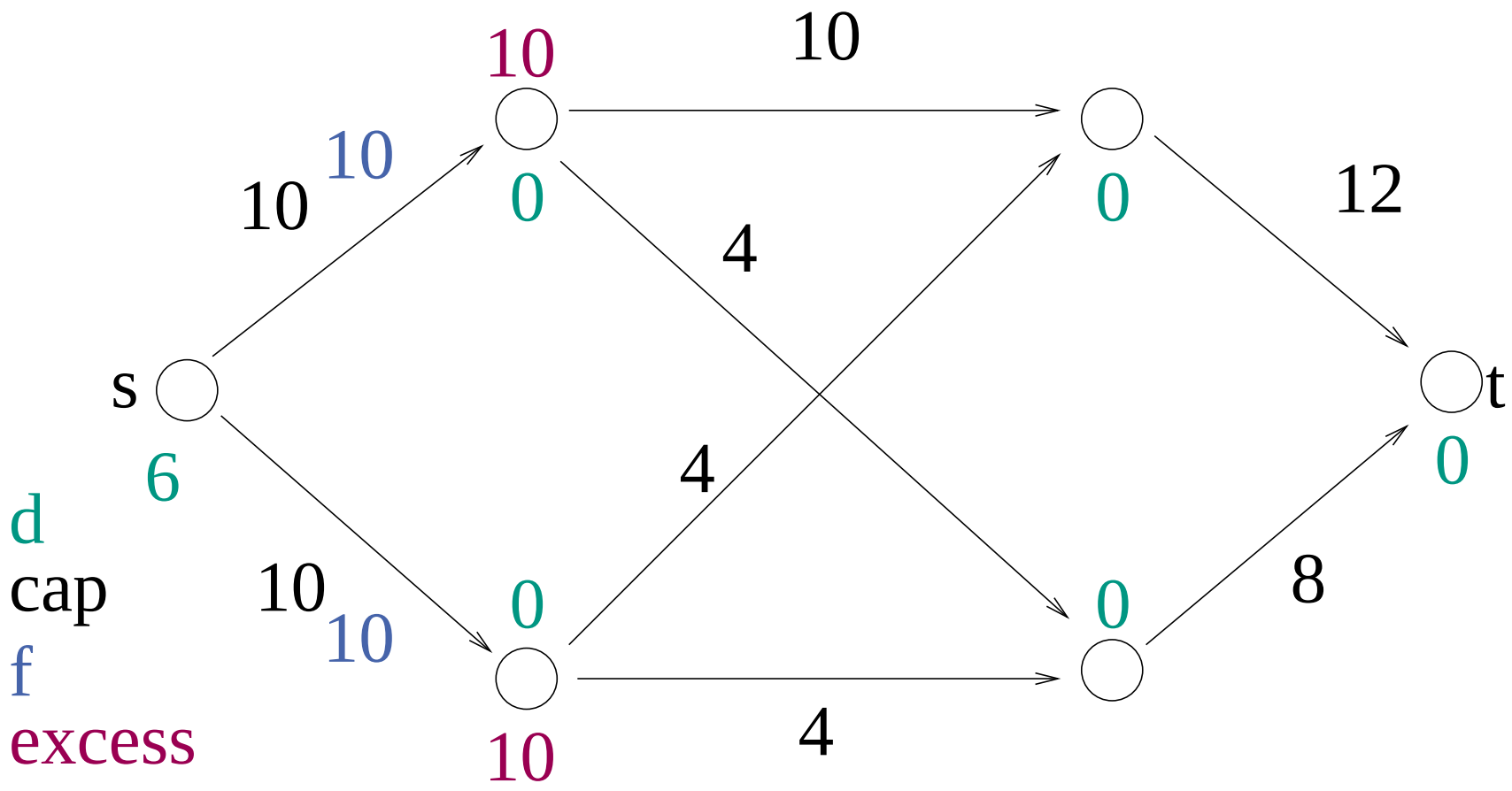
Obvious choice for δ : $\delta = \min\left\{\text{excess}(v), c_e^f\right\}$

Saturating push: $\delta = c_e^f$

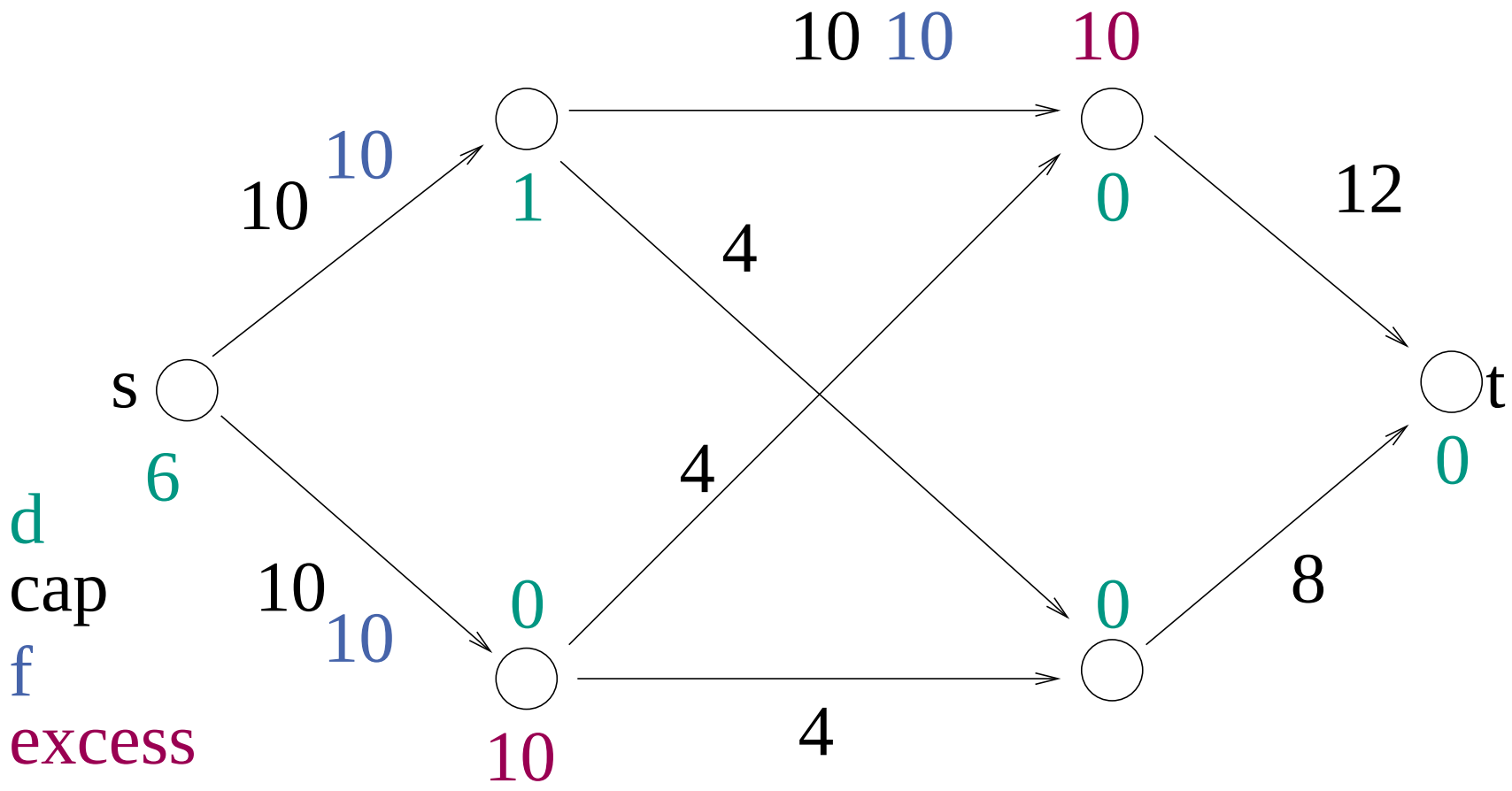
nonsaturating push: $\delta < c_e^f$

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

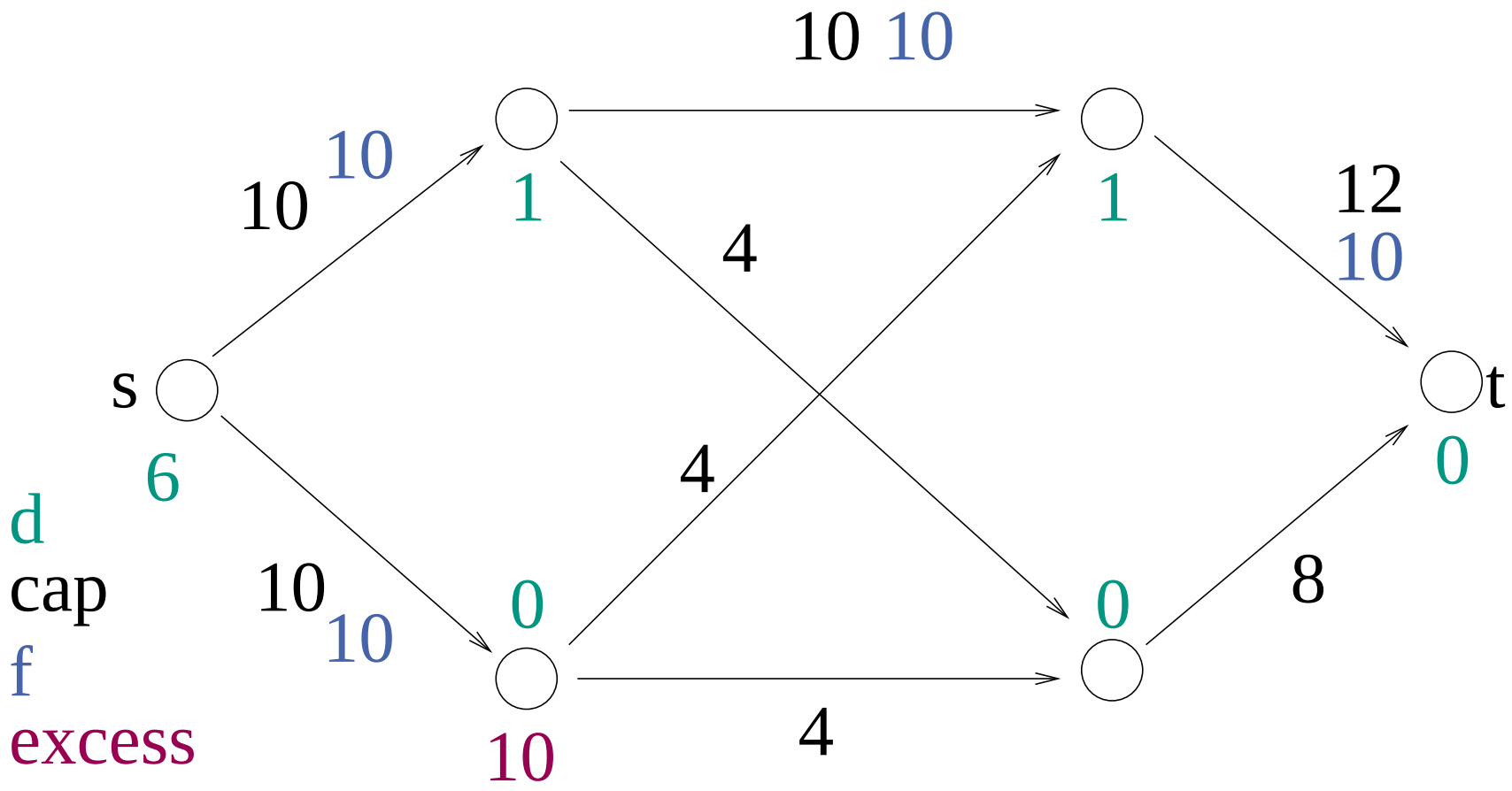
Example



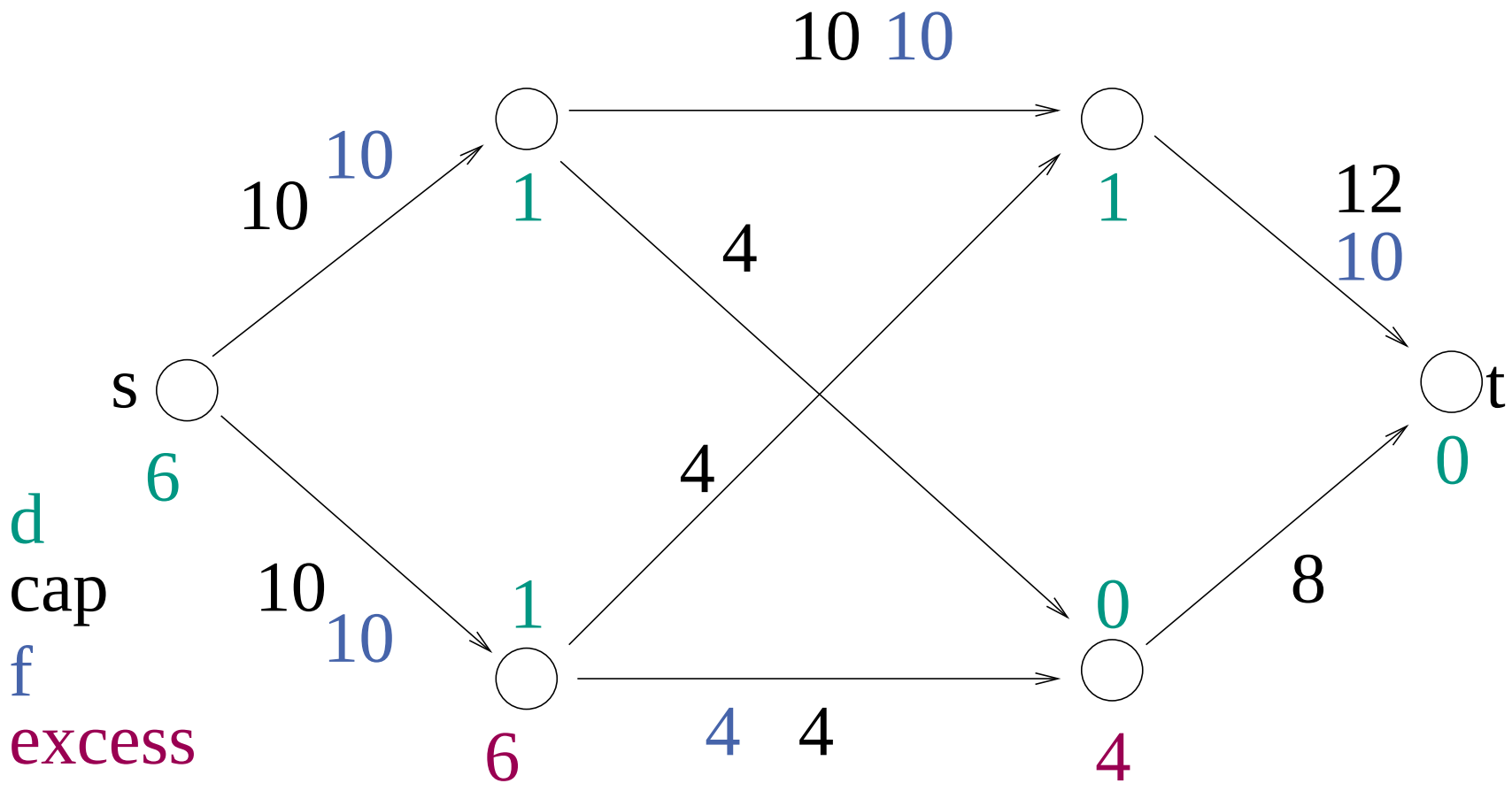
Example



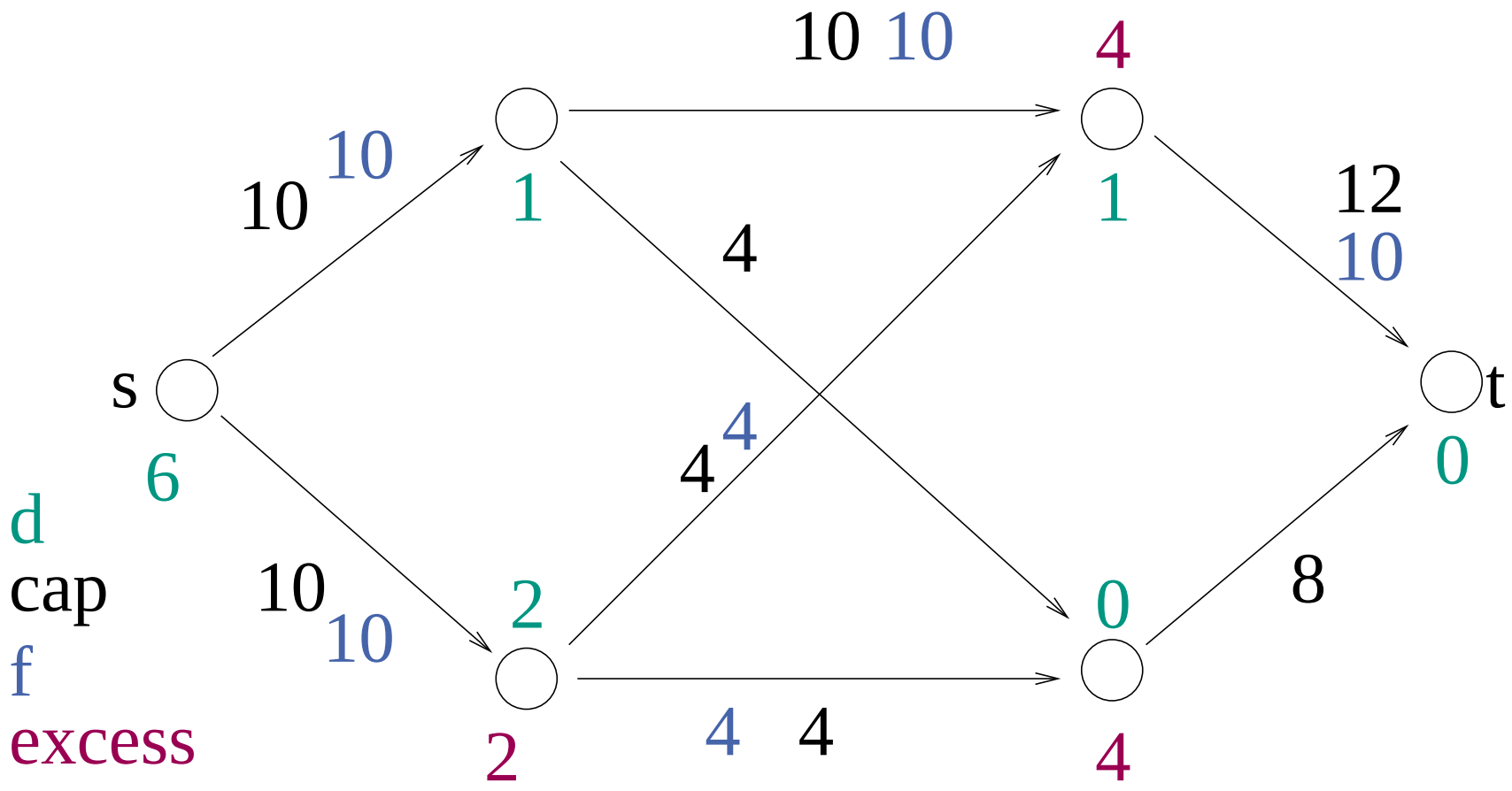
Example



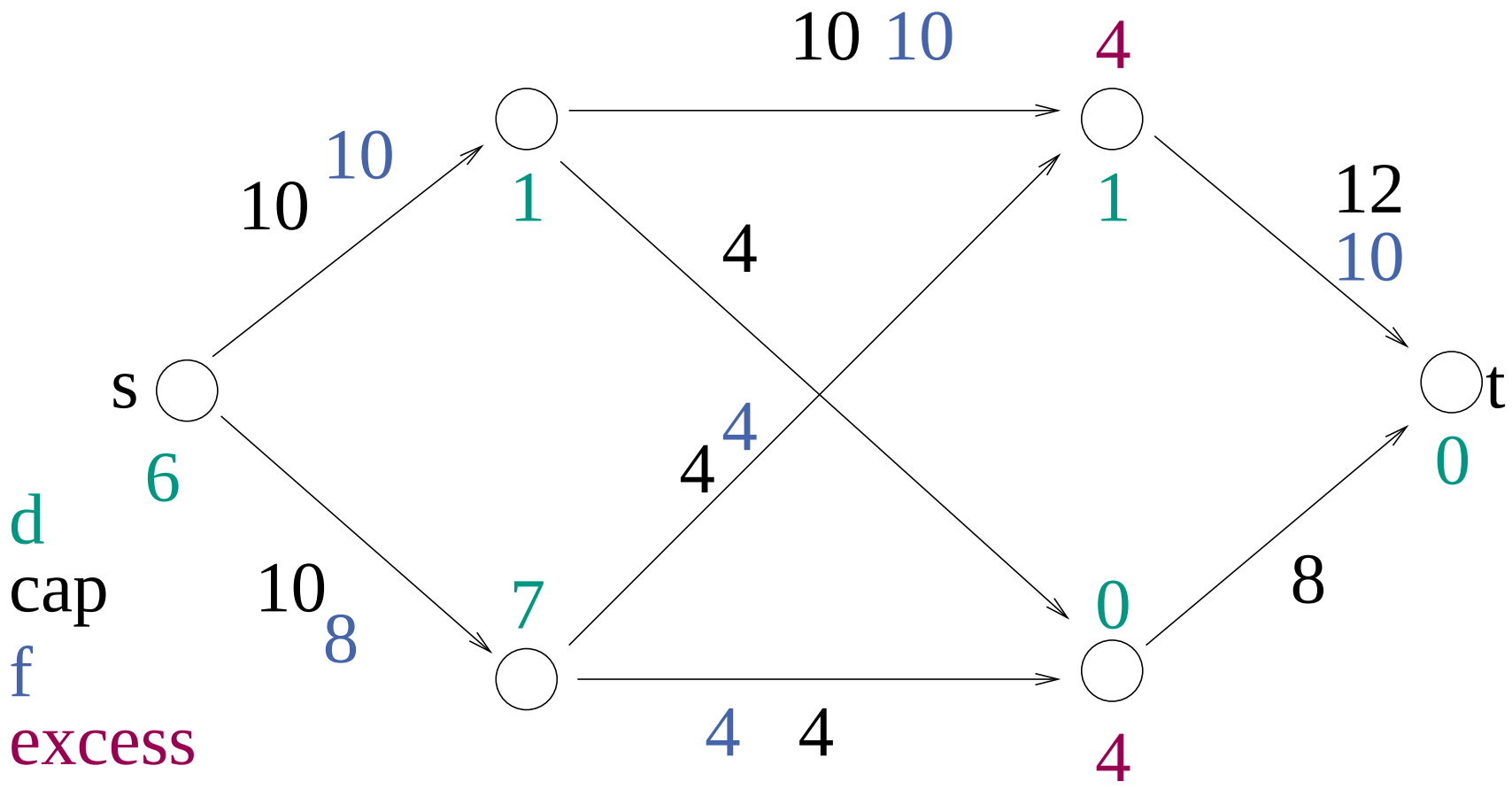
Example



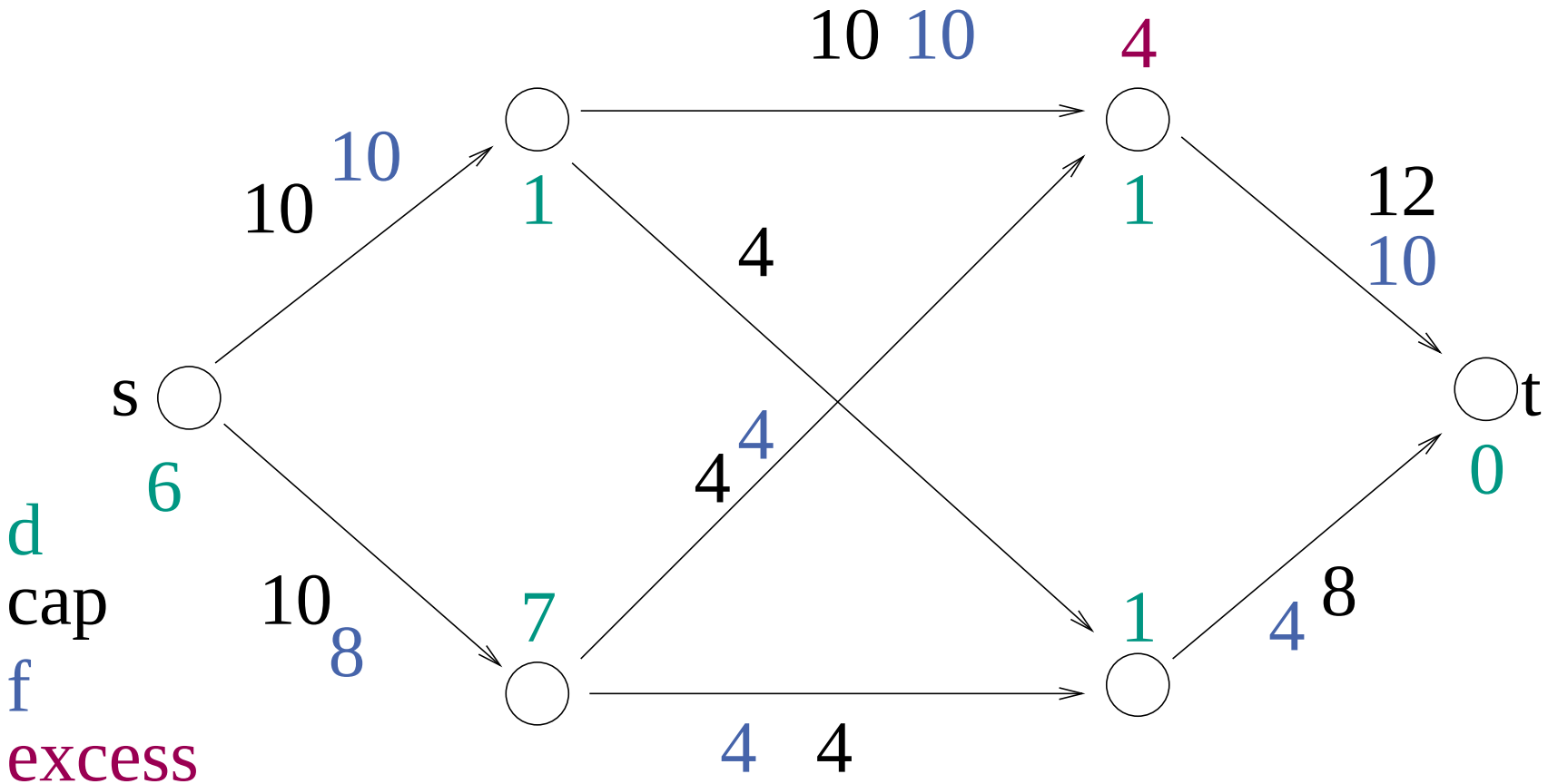
Example



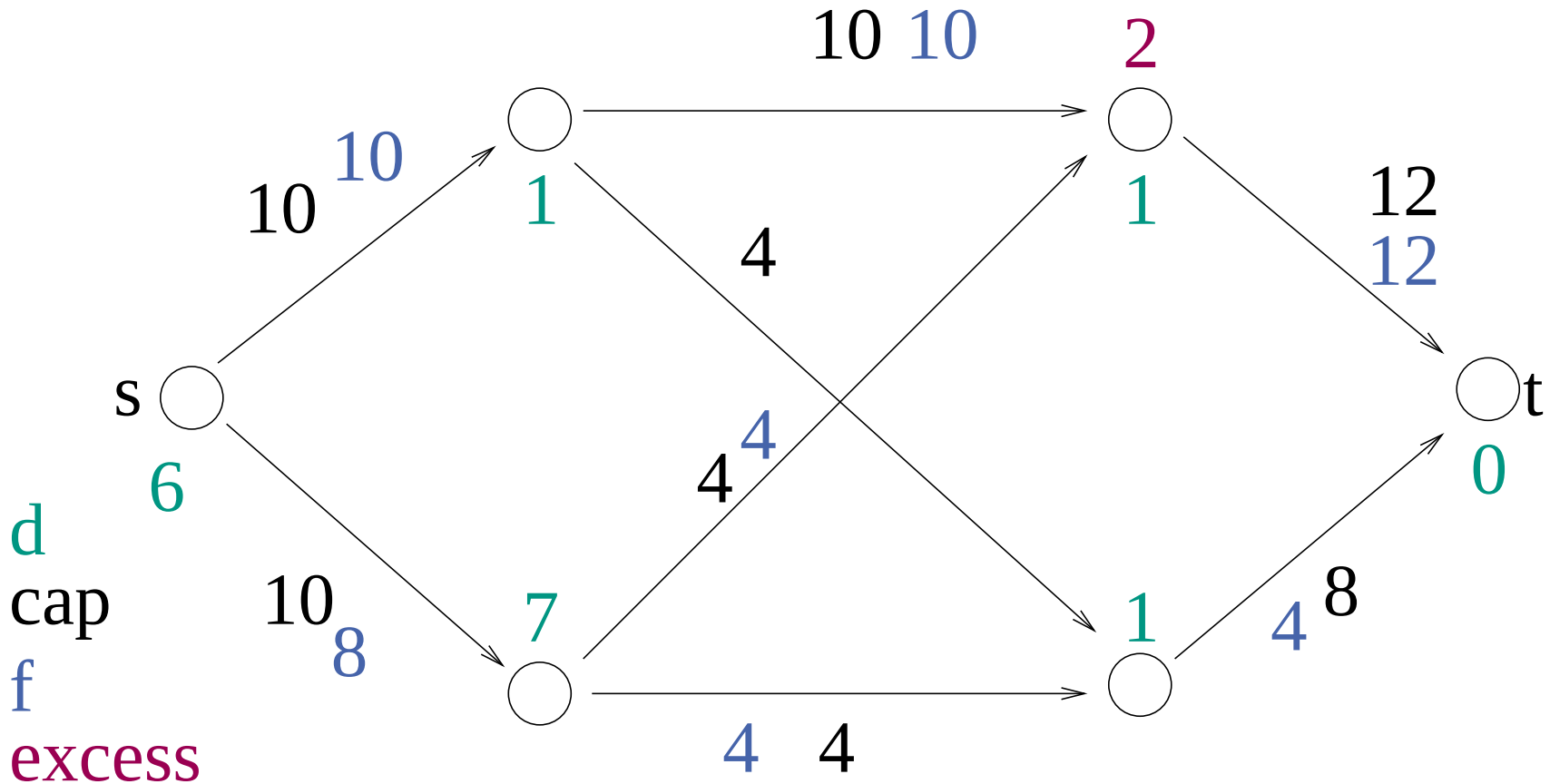
Example



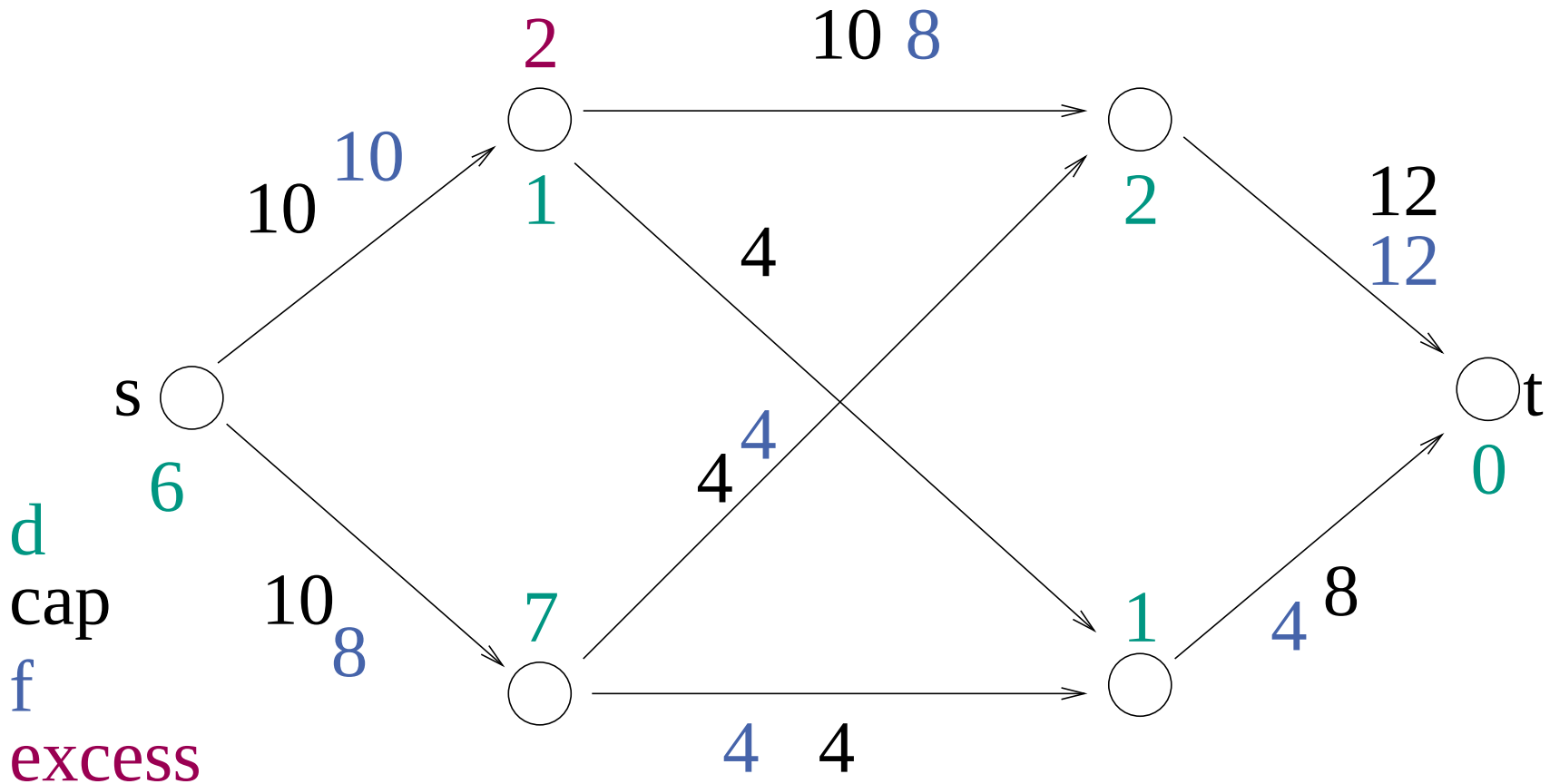
Example



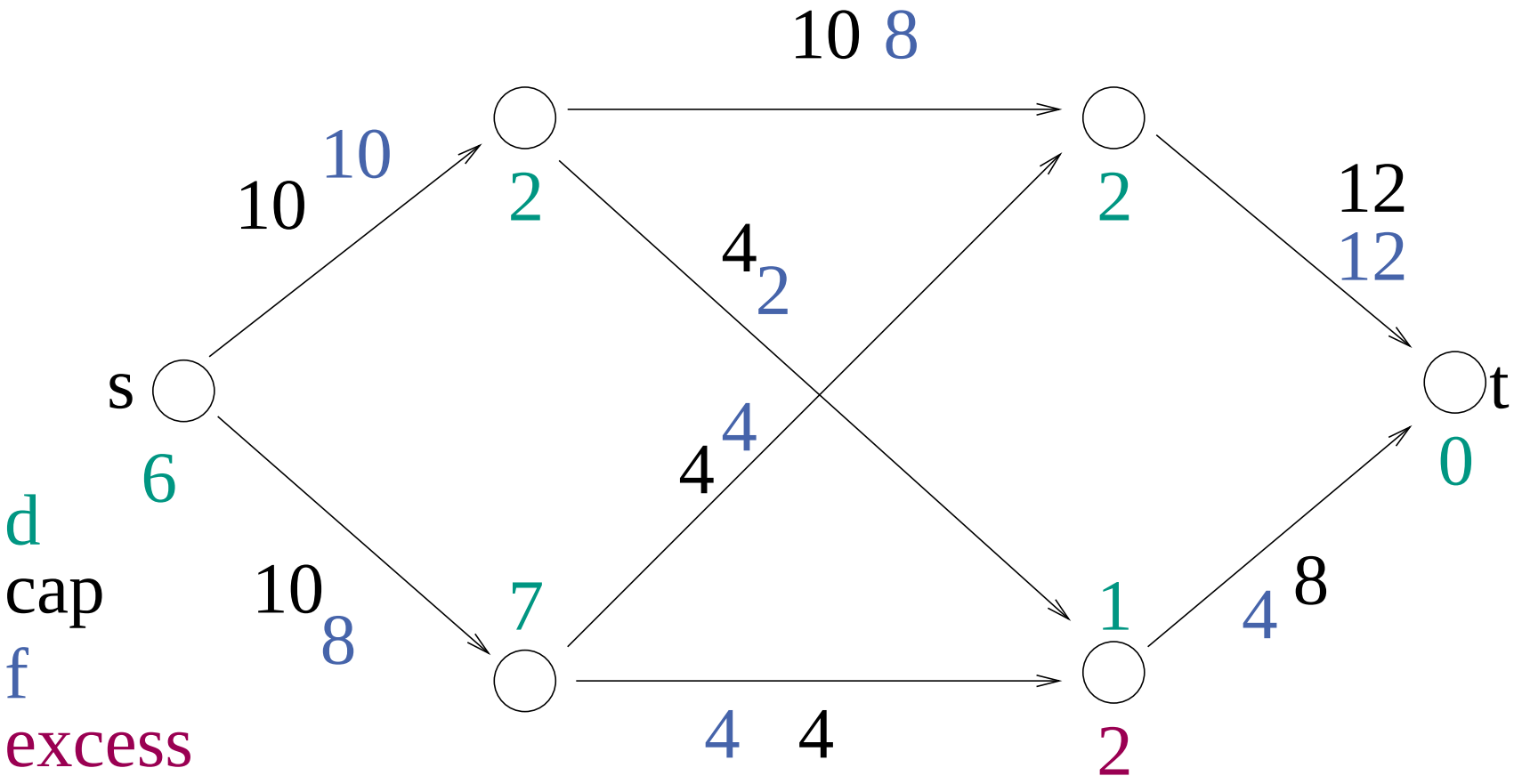
Example



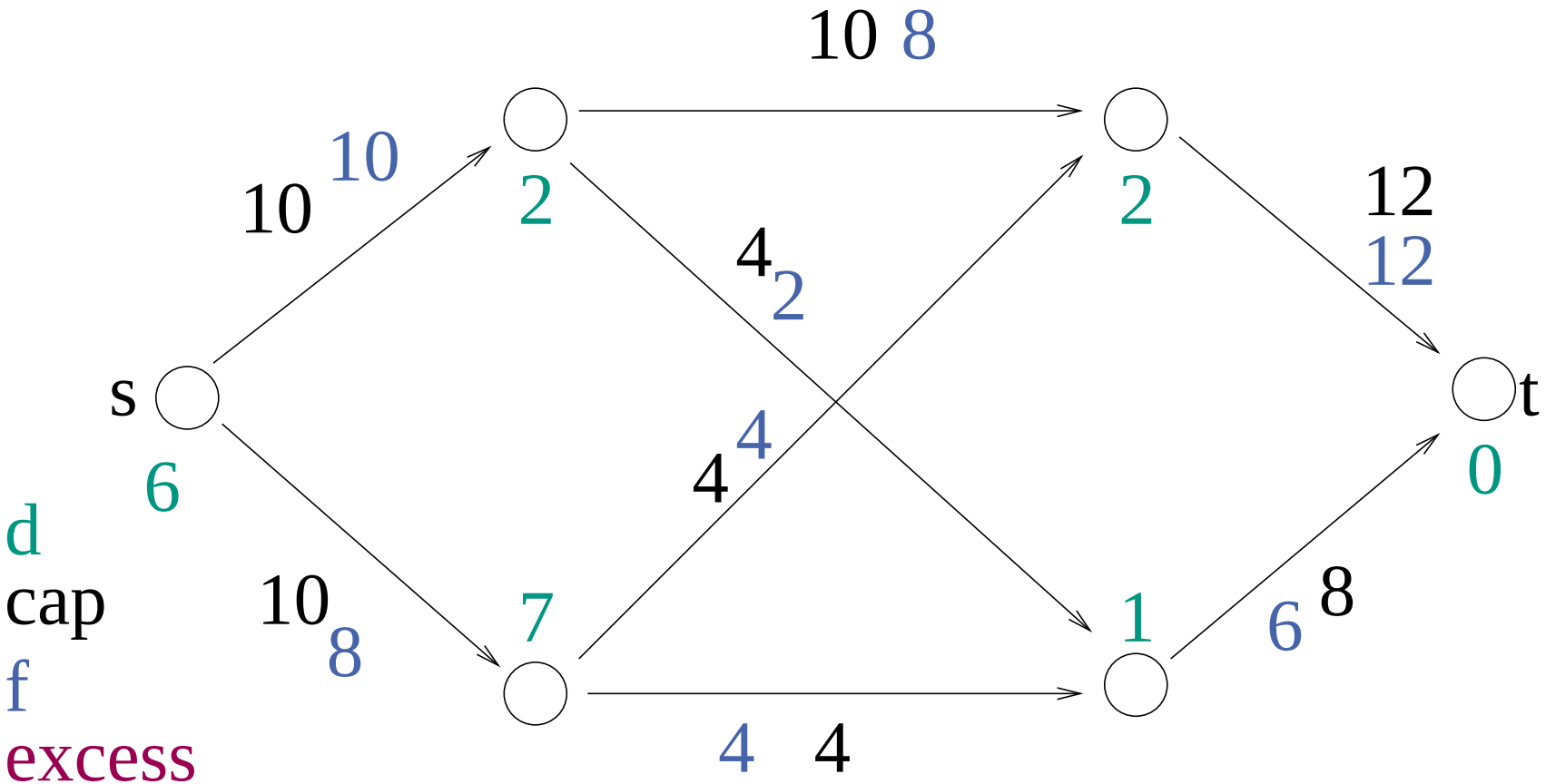
Example



Example



Example



12 pushes in total

Partial Correctness

Lemma 4. *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 $p = \langle s, \ldots, t \rangle \in G_f$ (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 5. *For any cut* (S,T),

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

Proof:

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

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 6.

$\forall$ *active nodes* v : excess$(v) > 0 \Rightarrow \exists$ *path* $\langle v, \ldots, s \rangle \in G_f$

Intuition: what got there can always go back.

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

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

$\forall (u,w) \in E_f : u \in S \Rightarrow w \in S$ 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, $\sum_{u \in S} excess(u) \leq 0$

Only the negative excess of s can outweigh excess$(v) > 0$.

Hence $s \in S$. $\square$

Lemma 7.

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

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 8. *# 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 9. *# saturating pushes* $\leq nm$

Beweis.

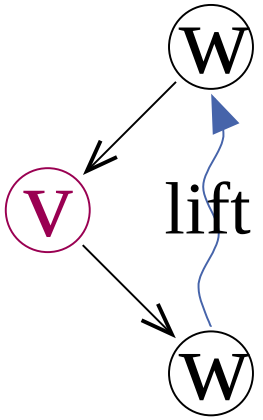
We show that there are at most n sat. pushes over any edge $e = (v, w)$.

A saturating push(e, δ) 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 10. *# nonsaturating pushes* $= O(n^2 m)$
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	$\leq 2n$	$\leq nm$	$\leq 2n^2 m$
nonsaturating push	≤ -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 11.

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

Satz 12. *Arbitrary Preflow Push finds a maximum flow in time* $O(n^2 m)$.

Beweis.

Lemma 4: 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 8: $2n^2$ relabel operations

Lemma 9: nm saturating pushes

Lemma 10: $O(n^2 m)$ nonsaturating pushes

Lemma 11: $O(nm)$ search time for eligible edges

Total time $O(n^2 m)$ □

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)

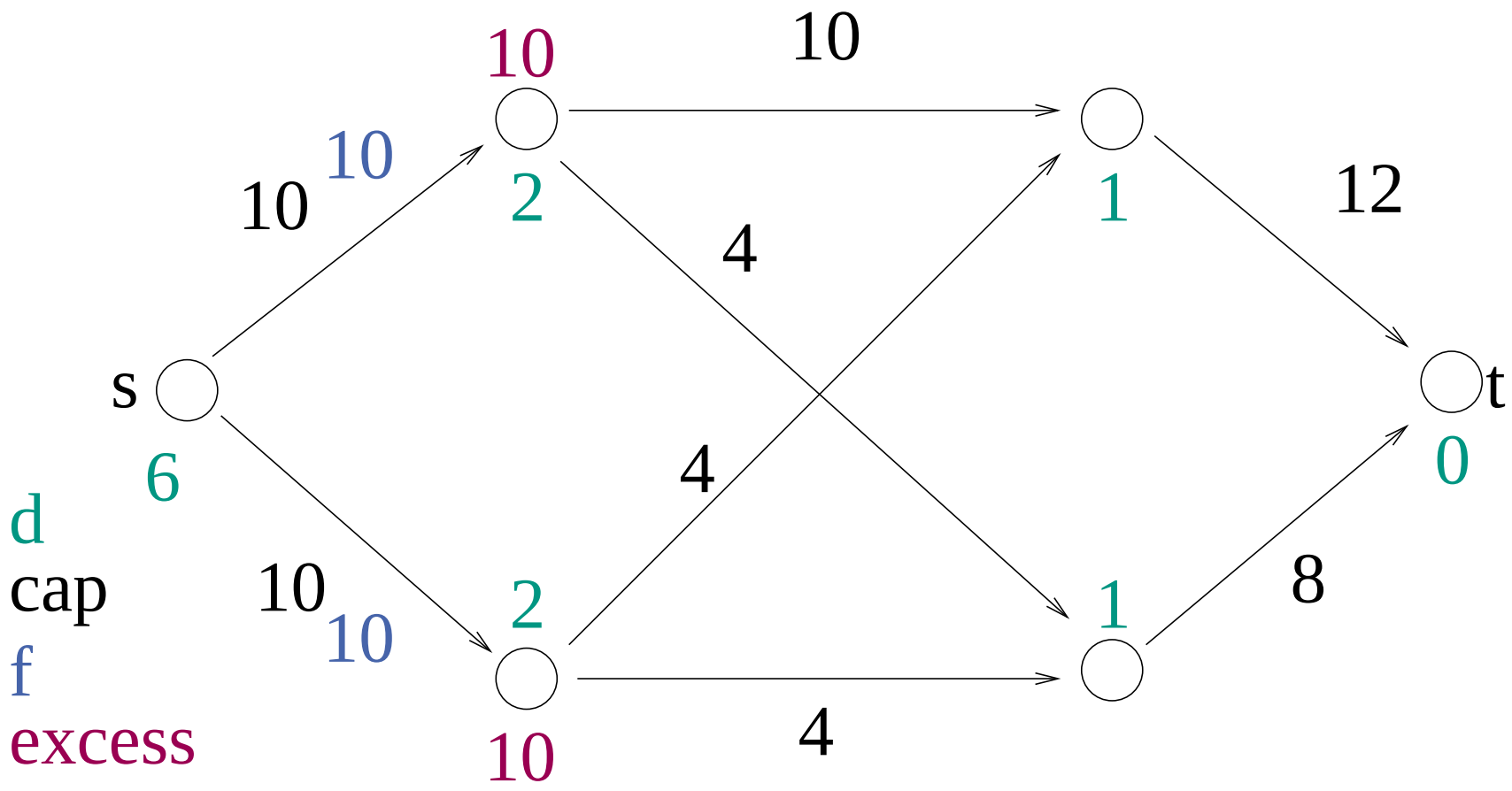
not monotone (!) but relabels "pay" for scan operations

Lemma 13. *At most $n^2\sqrt{m}$ nonsaturating pushes.*

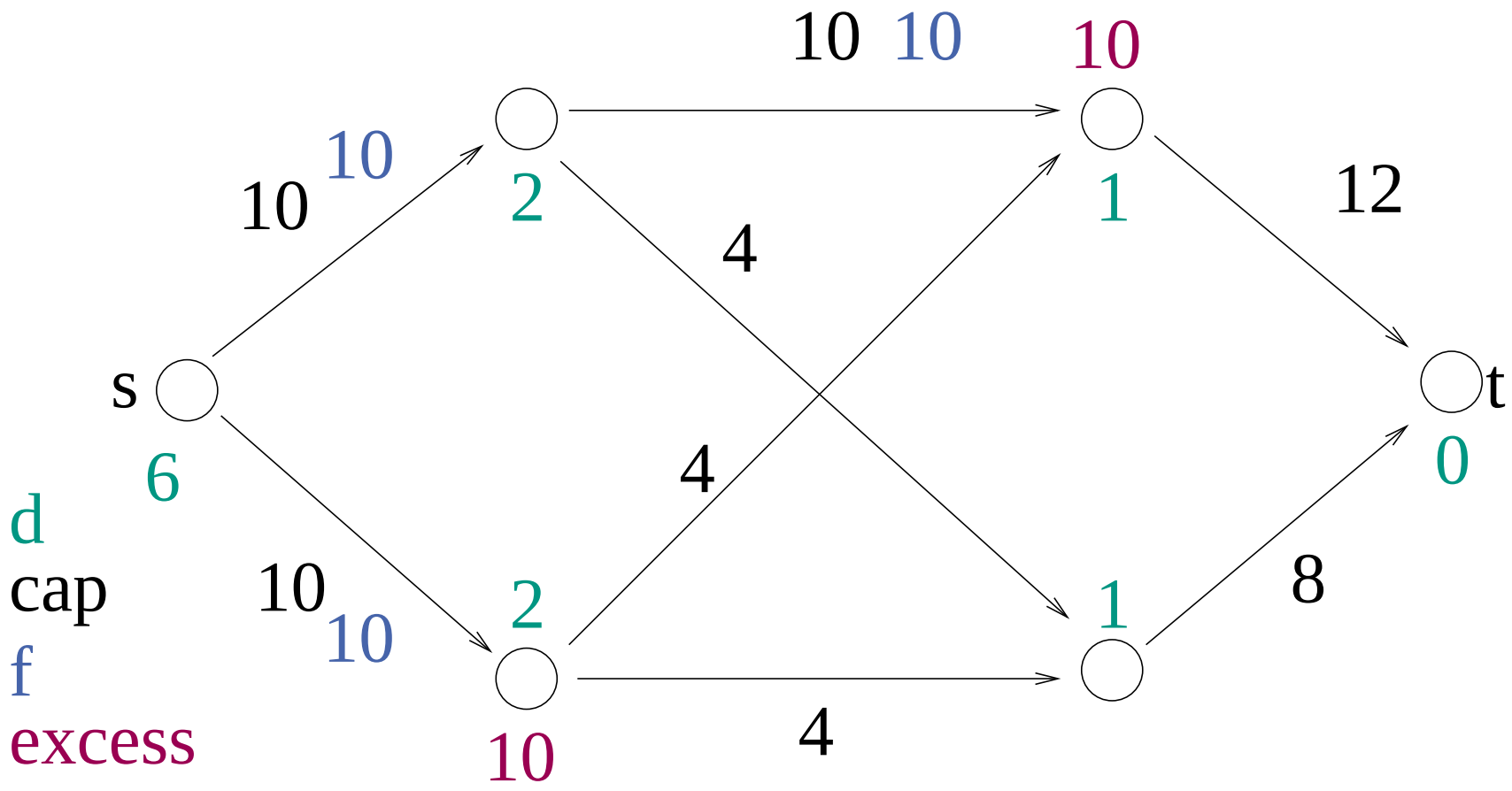
Beweis. later □

Satz 14. *Highest Level Preflow Push finds a maximum flow in time $O(n^2\sqrt{m})$.*

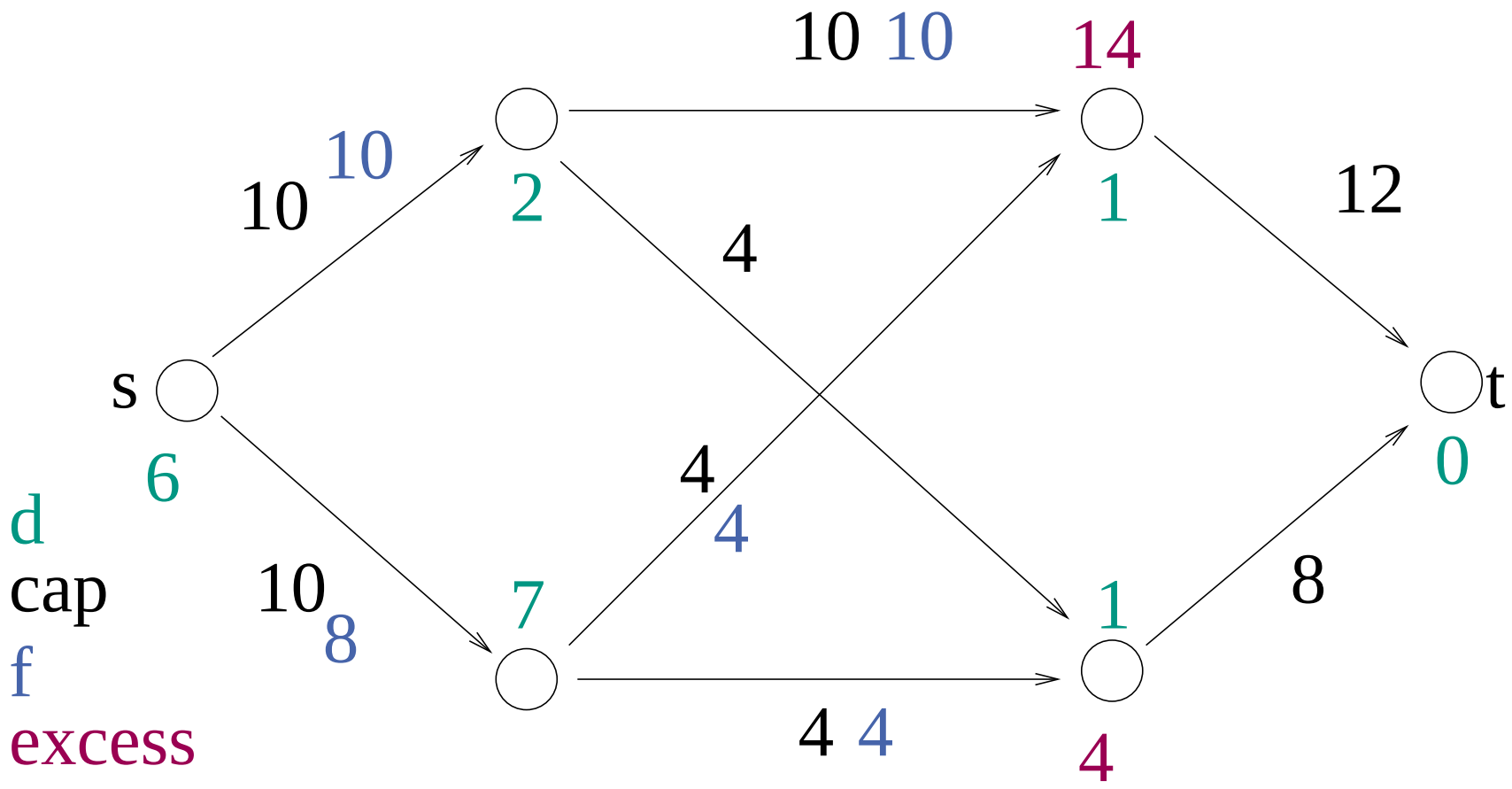
Example



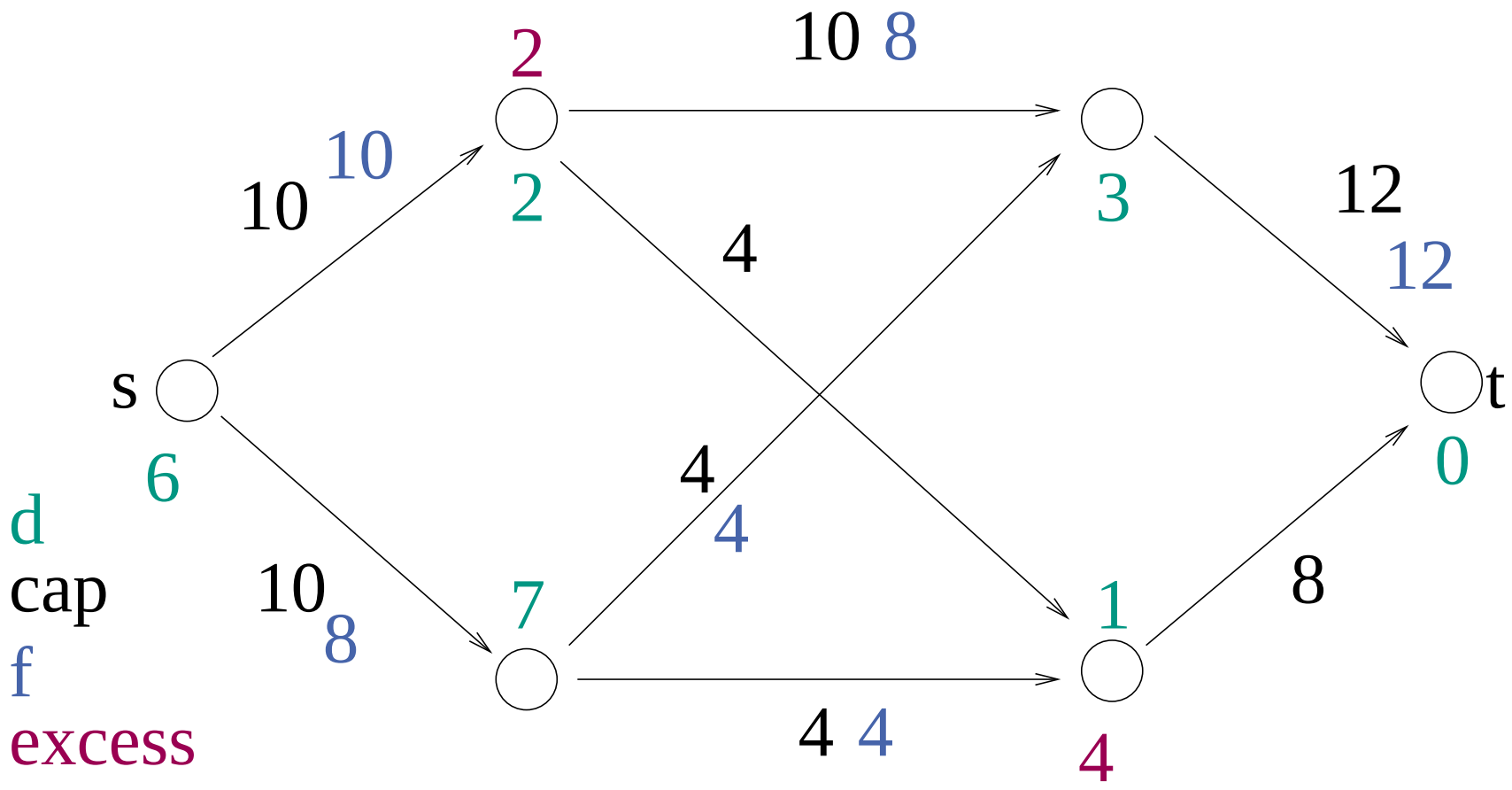
Example



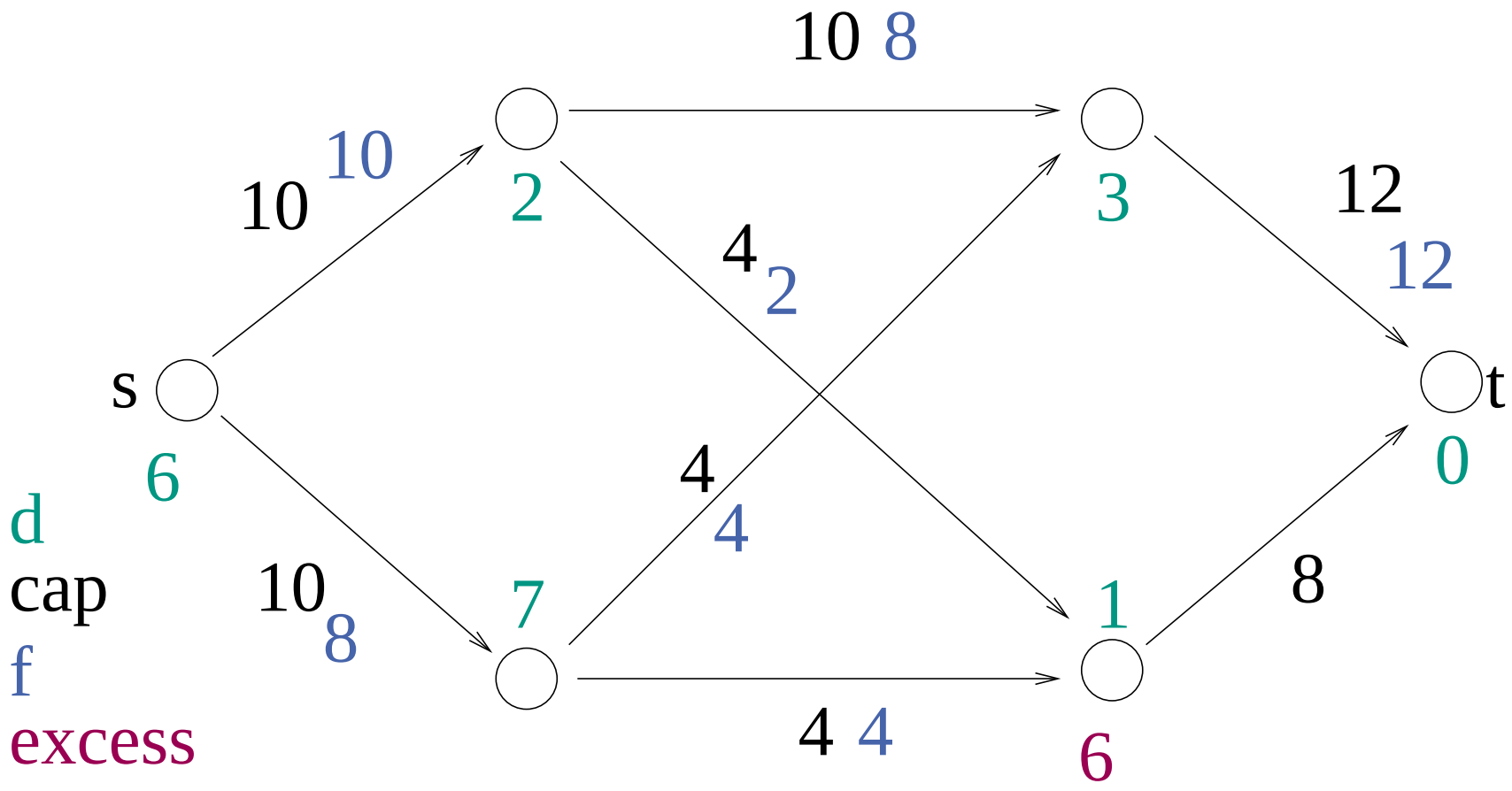
Example



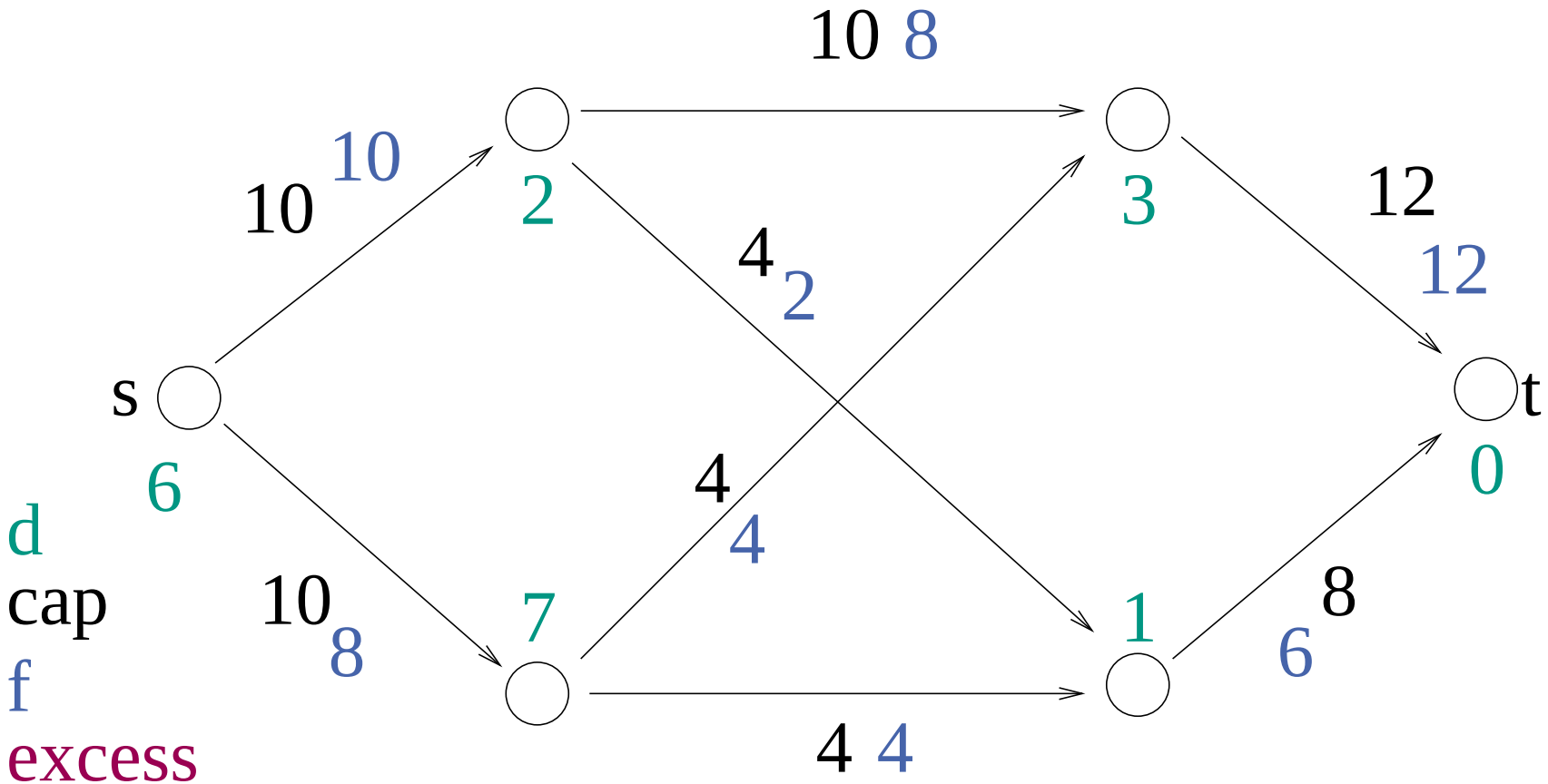
Example



Example



Example



9 pushes in total, 3 less than before

Proof of Lemma 13

$K := \sqrt{m}$ 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 Φ by at most n/K.
4. a nonsaturating push does not increase Φ.
5. an expensive phase with $Q \geq K$ nonsaturating pushes decreases Φ by at least Q.

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

Lemma 8+Lemma 9+2.+3.+4.:$\Rightarrow$

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

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 8 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 Φ 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 Φ by at most n/K.
4. a nonsaturating push does not increase Φ.

v is deactivated (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 Φ by at most n/K.
4. a nonsaturating push does not increase Φ.
5. an expensive phase with $Q \geq K$ nonsaturating pushes decreases Φ by at least Q.

During a phase d^* remains constant

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

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

Each nonsat. push across (v,w) decreases Φ 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 Φ by at most n/K.
4. a nonsaturating push does not increase Φ.
5. an expensive phase with $Q \geq K$ nonsaturating pushes decreases Φ by at least Q.

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

Lemma 8+Lemma 9+2.+3.+4.:$\Rightarrow$

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

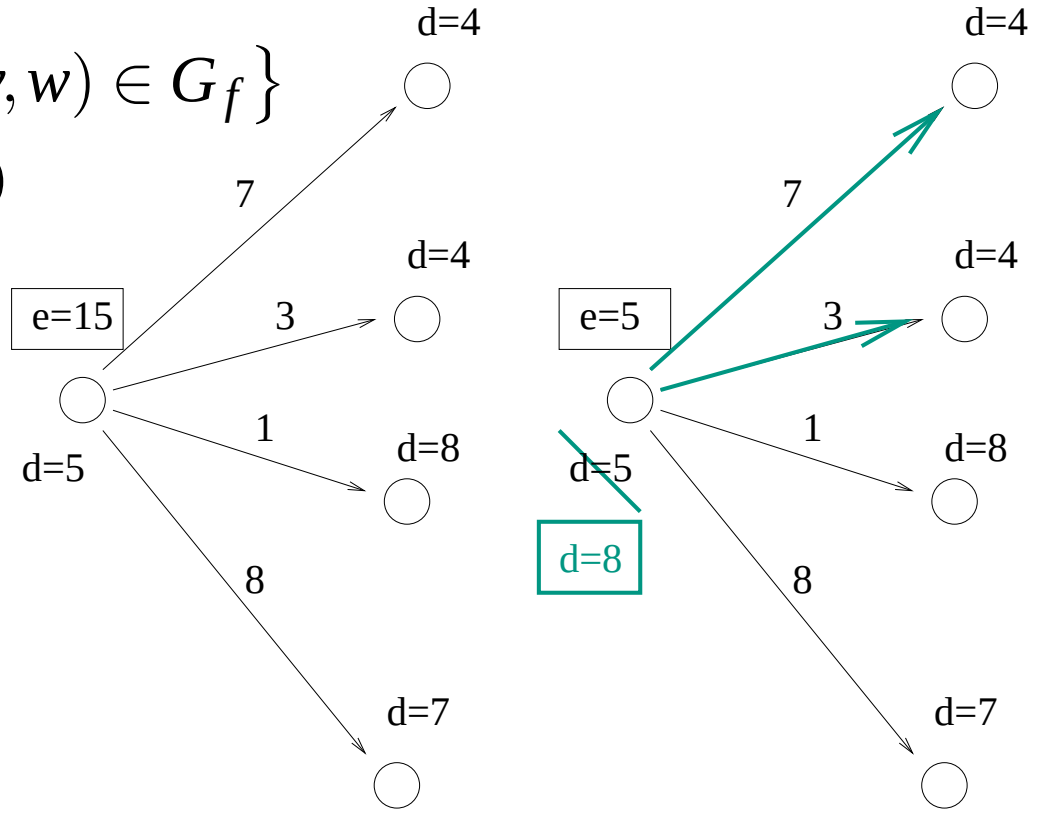
Heuristic Improvements

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

aggressive local relabeling:

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

(like a sequence of relabels)



Heuristic Improvements

Naive algorithm has best case $\Omega\left(n^2\right)$. Why?

We can do better.

aggressive local relabeling: $d(v) := 1 + \min\left\{d(w) : (v,w) \in G_f\right\}$

(like a sequence of relabels)

global relabeling: (initially and every $\mathrm{O}(m)$ edge inspections):

$d(v) := G_f$.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	HL	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=hightest level, MF=modified FIFO

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

LRH=local relabeling heuristic, GRH=global relabeling heuristics

Timings: CG1

Rule	BASIC	HL	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=hightest level, MF=modified FIFO

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

LRH=local relabeling heuristic, GRH=global relabeling heuristics

Timings: CG2

Rule	BASIC	HL	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=hightest level, MF=modified FIFO

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

LRH=local relabeling heuristic, GRH=global relabeling heuristics

Timings: AMO

Rule	BASIC	HL	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=hightest level, MF=modified FIFO

HL= $d(v) \geq n$ 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	—	—	—

Minimum Cost Flows

Define $G = (V,E)$, f, excess, and c as for maximum flows.

Let cost : $E \to \mathbb{R}$ denote the edge costs.

Consider supply : $V \to \mathbb{R}$ with $\sum_{v \in V} \text{supply}(v) = 0$. A negative supply is called a demand.

Objective: minimize $\text{cost}(f) := \sum_{e \in E} f(e)\text{cost}(e)$

subject to

$\forall v \in V : \text{excess}(v) = -\text{supply}(v)$ flow conservation constraints

$\forall e \in E : f(e) \leq c(e)$ capacity constraints

Cycle Canceling Alg. for Min-Cost Flow

Residual cost:

$\text{cost}_f(e) = \text{cost}(e)$ (exists in G_f iff $f(e) < c(e)$)

$\text{cost}_f(\text{reverse}(e)) = -\text{cost}(e)$ (exists in G_f iff $f(e) > 0$)

Lemma 15. *A feasible flow is optimal iff*

$\nexists$ *cycle* $C \in G_f : \text{cost}_f(C) < 0$

Beweis. not here □

A pseudopolynomial Algorithm:

$f :=$ any feasible flow// Exercise: solve this problem using maximum flows

invariant f is feasible

while $\exists$ cycle $C : \text{cost}_f(C) < 0$ **do** augment flow around C

Korollar 16 (Integrality Property:). *If all edge capacities are integral then there exists an integral minimum cost flow.*

Finding a Feasible Flow

set up a maximum flow network G^* starting with the min cost flow problem G:

- Add a vertex s
- $\forall v \in V$ with supply$(v) > 0$, add edge (s, v) with cap. supply(v)
- Add a vertex t
- $\forall v \in V$ with supply$(v) < 0$, add edge (v, t) with cap. $-$supply(v)
- find a maximum flow f in G^*

f saturates the edges leaving $s \Rightarrow f$ is feasible for G

otherwise there cannot be a feasible flow f' because f' could easily be converted into a flow in G^* with larger value.

Better Algorithms

Satz 17. *The min-cost flow problem can be solved in time* $O\left(mn \log n + m^2 \log \max_{e \in E} c(e)\right)$.

For details take the courses in optimization or network flows.

Special Cases of Min Cost Flows

Transportation Problem: $\forall e \in E : c(e) = \infty$

Minimum Cost Bipartite Perfect Matching:

A transportation problem in a bipartite graph $G = (A \cup B, E \subseteq A \times B)$ with

supply$(v) = 1$ for $v \in A$,

supply$(v) = -1$ for $v \in B$.

An integral flow defines a matching

Reminder: $M \subseteq E$ is a matching if (V, M) has maximum degree one.

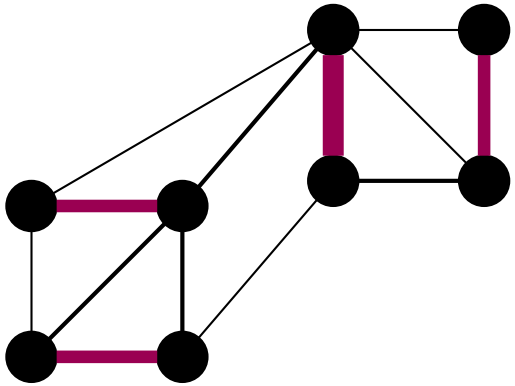
A rule of Thumb: If you have a combinatorial optimization problem. Try to formulate it as a shortest path, flow, or matching problem. If this fails its likely to be NP-hard.

Maximum Weight Matching

Generalization of maximum cardinality matching. Find a matching $M^* \subseteq E$ such that $w(M^*) := \sum_{e \in M^*} w(e)$ is maximized

Applications: Graph partitioning, selecting communication partners...

Satz 18. *A maximum weighted matching can be found in time* $O(nm + n^2 \log n)$. *[Gabow 1992]*



Approximate Weighted Matching

Satz 19. *There is an* $O(m)$ *time algorithm that finds a matching of weight at least* $\max_{matching M} w(M)/2$.
[Drake Hougardy 2002]

The algorithm is a $1/2$-approximation algorithm.

Approximate Weighted Matching Algorithm

$M' := \emptyset$

invariant M' is a set of simple paths

while $E \neq \emptyset$ **do** // find heavy simple paths

  select any $v \in V$ with degree$(v) > 0$ // select a starting node

  while degree$(v) > 0$ **do** // extend path greedily

    $(v, w) :=$ heaviest edge leaving v // **(*)**

    $M' := M' \cup \{(v, w)\}$

    remove v from the graph

    $v := w$

return any matching $M \subseteq M'$ with $w(M) \geq w(M')/2$

// one path at a time, e.g., look at the two ways to take every other edge.

Proof of Approximation Ratio

Let M^* denote a maximum weight matching.

It suffices to show that $w(M') \geq w(M^*)$.

Assign each edge to that incident node that is deleted first.

All $e^* \in M^*$ are assigned to different nodes.

Consider any edge $e^* \in M^*$ and assume it is assigned to node v.

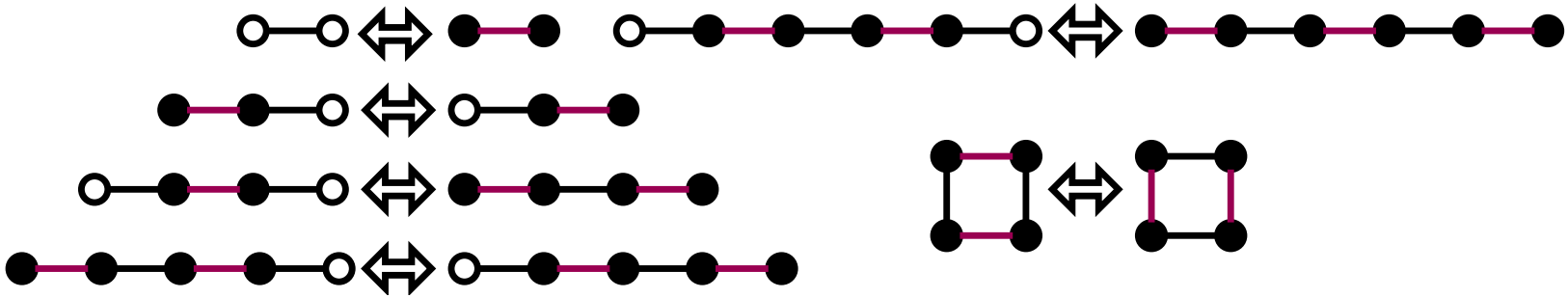
Since e^* is assigned to v, it was available in line **(*)**.

Hence, there is an edge $e \in M'$ assigned to v with $w(e) \geq w(e^*)$.

Better Approximate Weighted Matching

$\frac{2}{3} - \varepsilon$-approximation in time $O\left(m \log \frac{1}{\varepsilon}\right)$ [Pettie Sanders 2003]

0. $M := \emptyset$
1. **repeat** $\Theta\left(n \log \frac{1}{\varepsilon}\right)$ times:
2. Let $X \in V(G)$ be selected uniformly at random
3. $M := M \oplus \text{aug}(X)$
4. **return** M



Zusammenfassung Flows und Matchings

- Natürliche Verallgemeinerung von kürzesten Wegen:
  ein Pfad $\rightsquigarrow$ viele Pfade
- viele Anwendungen
- “schwierigste/allgemeinste” 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.)

$\langle e_1, \ldots, e_n \rangle$ ist Permutation von $\langle e'_1, \ldots, e'_n \rangle$ 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, field : Key $\to \mathbb{F}$ 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?

6.2 Hashing II

Perfektes Hashing

Idee: mache h injektiv.

Braucht $\Omega(n)$ bits Platz !

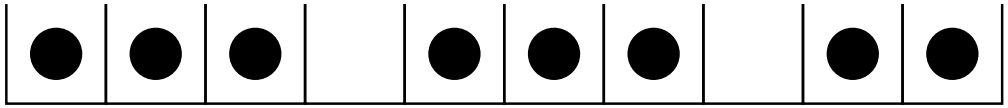
Here: Fast Space Efficient Hashing

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

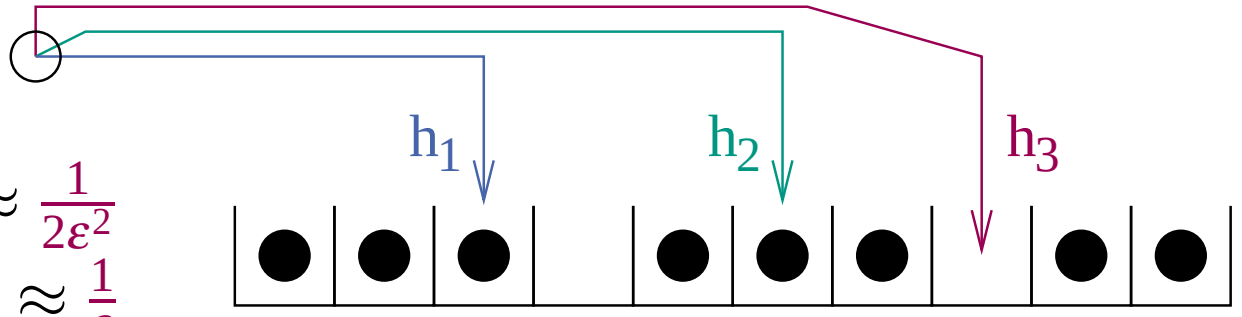
Support operations insert, delete, lookup, (doall) efficiently.

Assume a truly random hash function h

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



Related Work



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

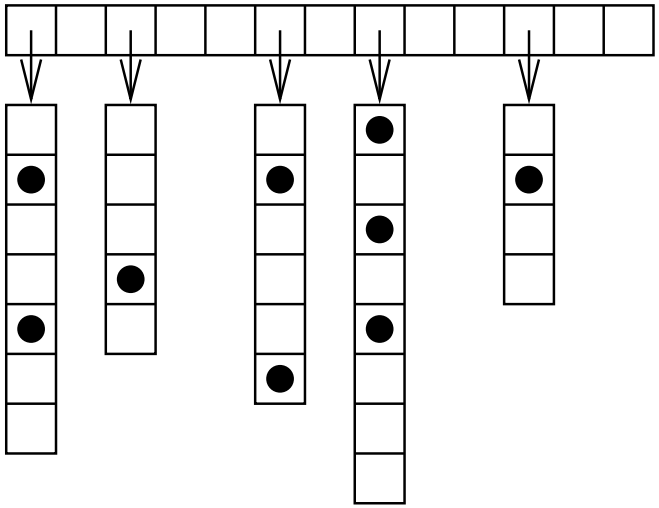
Uniform hashing: $E[T_{\text{find}}] \approx \frac{1}{\varepsilon}$

Dynamic Perfect Hashing,

[Dietzfelbinger et al. 94]

Worst case constant time

for lookup but ε 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.

Cuckoo Hashing

[Pagh Rodler 01]

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

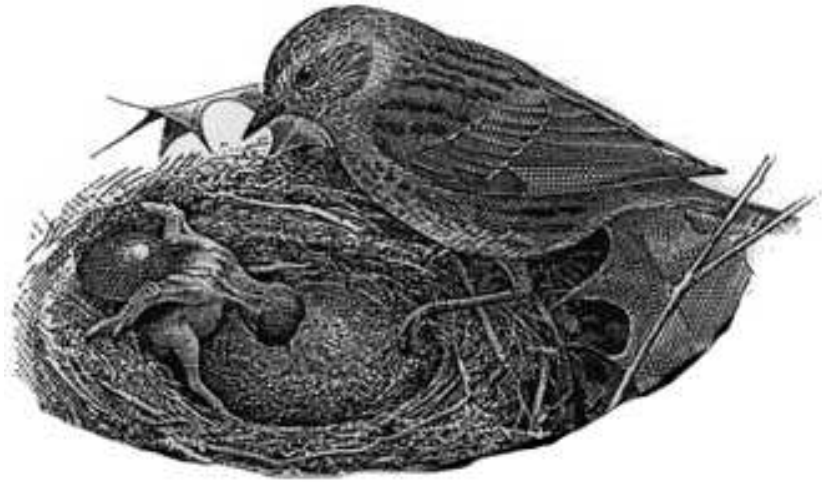
Two choices for each element.

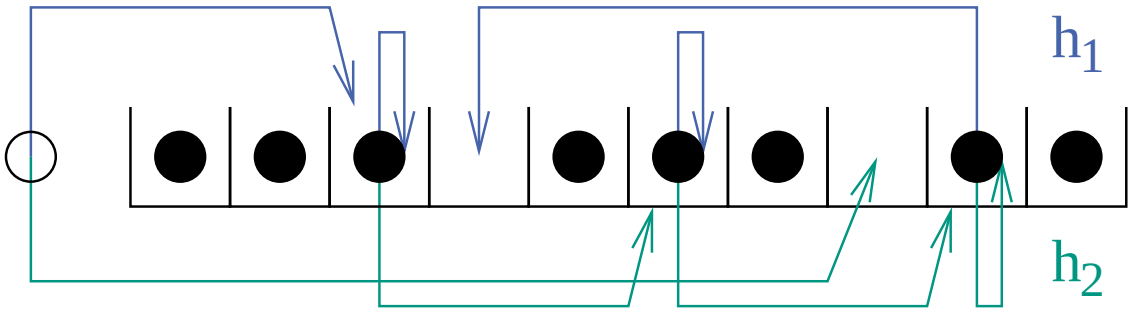
Insert moves elements;

rebuild if necessary.

Very fast lookup and insert.

Expected constant insertion time.





Cuckoo Hashing – Rebuilds

When needed ?

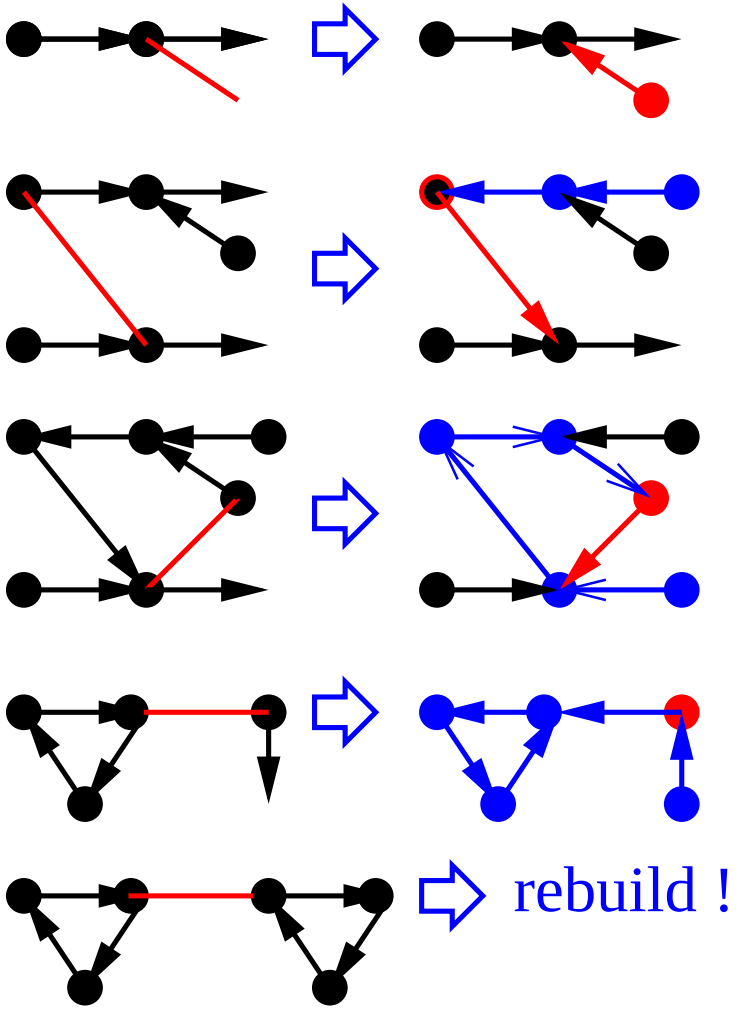
Graph model.

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: insert(x) succeeds iff the component containing $h_1(x), h_2(x)$ contains no more edges than nodes.



Cuckoo Hashing – Rebuilds

Lemma: insert(x) succeeds iff

the component containing $h_1(x), h_2(x)$

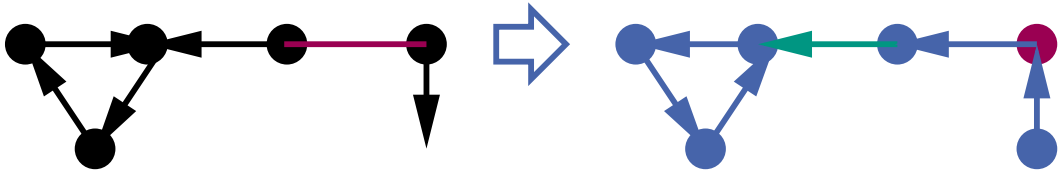
contains no more edges than nodes.

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,

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

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 $\geq 1 - \mathrm{O}(1/n)$.

Famous: The evolution of component sizes with increasing m:

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

$> (1+\varepsilon)n/2$: 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.

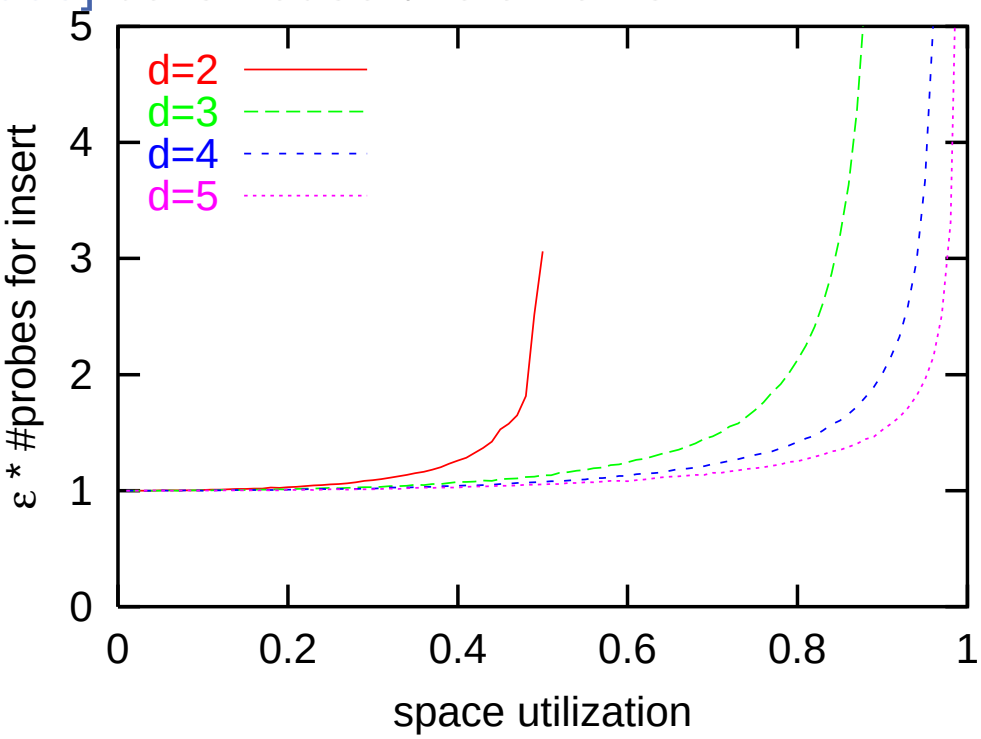
blocked: [Dietzfelbinger, Weidling 2005] cells house d elements.

Cache efficient !

access time: $O\left(\log \frac{1}{\varepsilon}\right)$

Insertion: BFS, random walk, ...

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



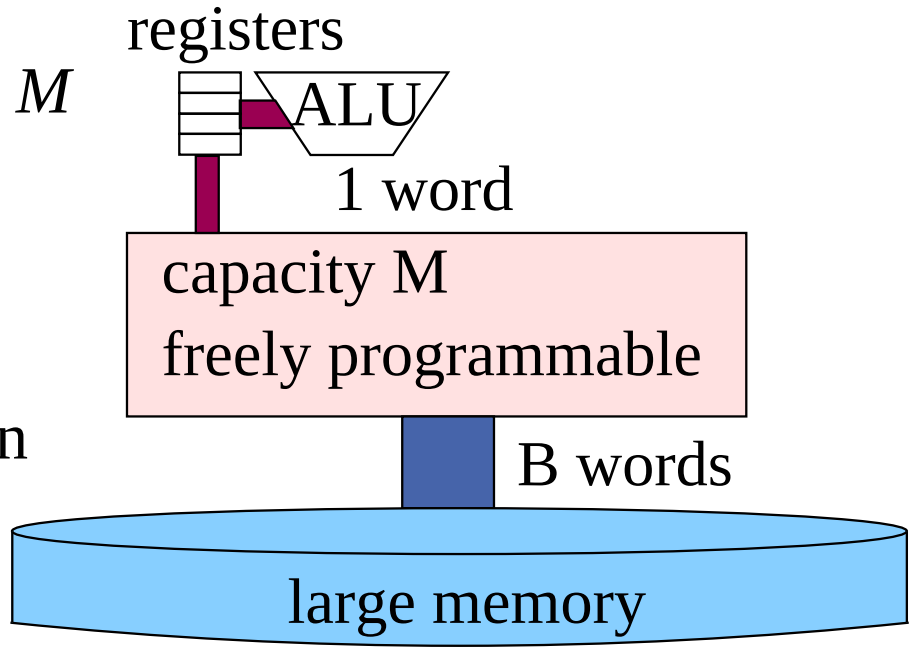
7 Externe Algorithmen

7.1 Das Sekundärspeichermodell

M: Schneller Speicher der Größe M

B: Blockgröße

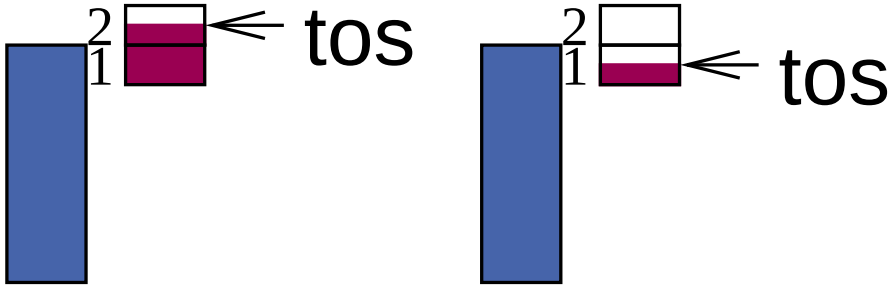
Analyse: Blockzugriffe zählen



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.

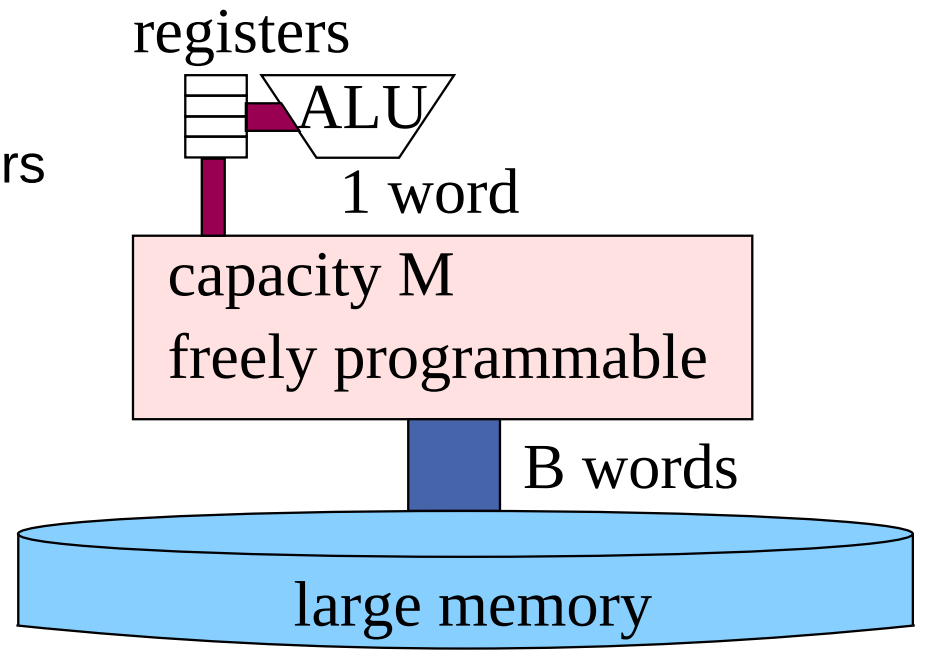
Aufgabe 2: effiziente Implementierung ohne überflüssiges Kopieren

7.3 Externes Sortieren

n: Eingabegröße

M: Größe des schnellen Speichers

B: Blockgröße



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

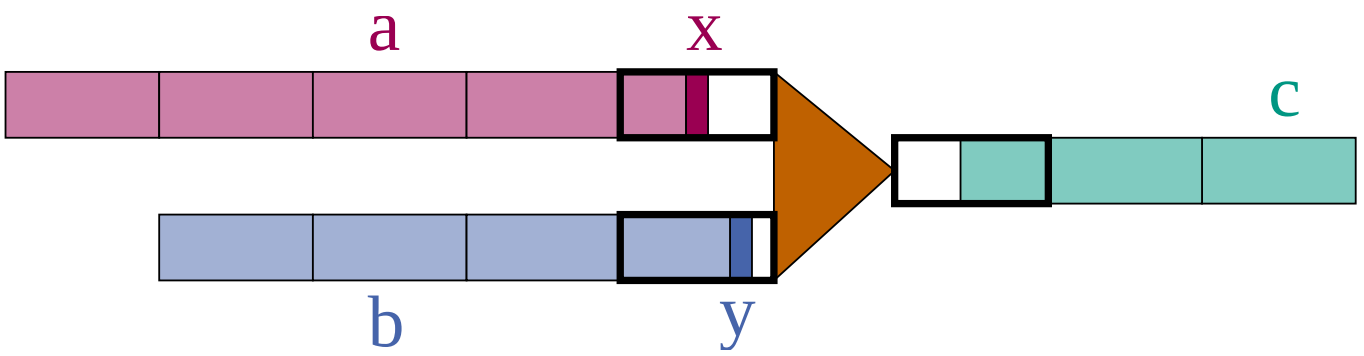
x := a.readElement // Assume emptyFile.readElement$= \infty$

y := b.readElement

for $j := 1$ **to** $|a| + |b|$ **do**

if $x \leq y$ **then** c.writeElement(x); x := a.readElement

else c.writeElement(y); y := b.readElement



Externes (binäres) Mischen – I/O-Analyse

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

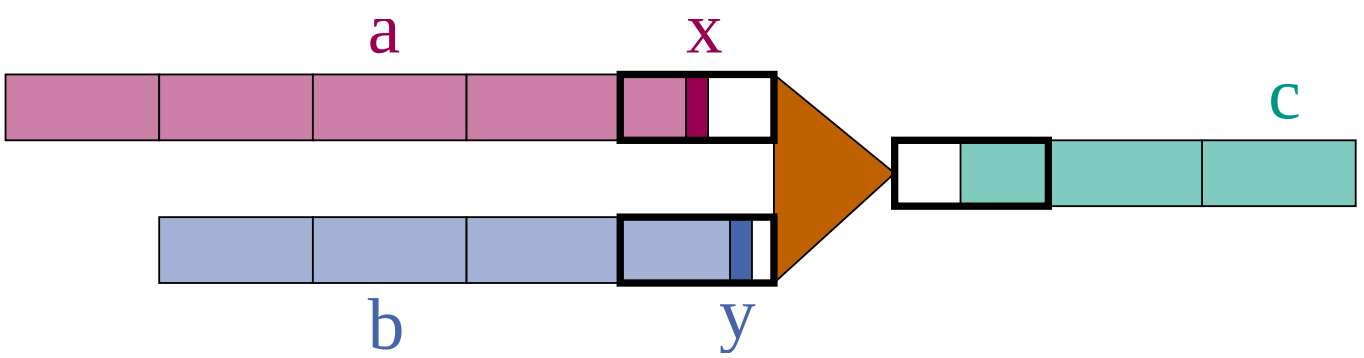
Datei b lesen: $\lceil |b|/B \rceil \leq |b|/B + 1.$

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

Insgesamt:

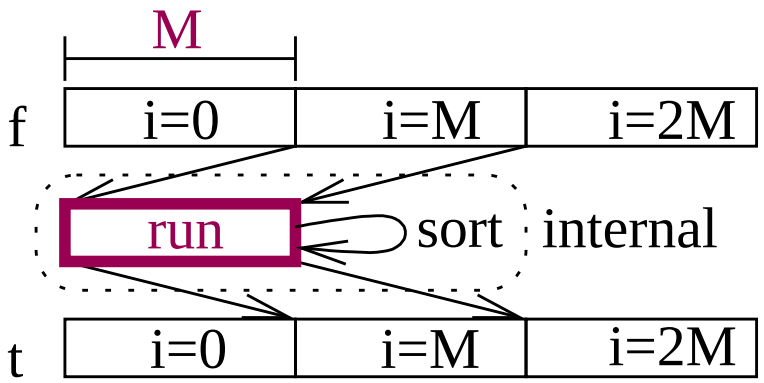
$$\leq 3 + 2\frac{|a| + |b|}{B} \approx 2\frac{|a| + |b|}{B}$$

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



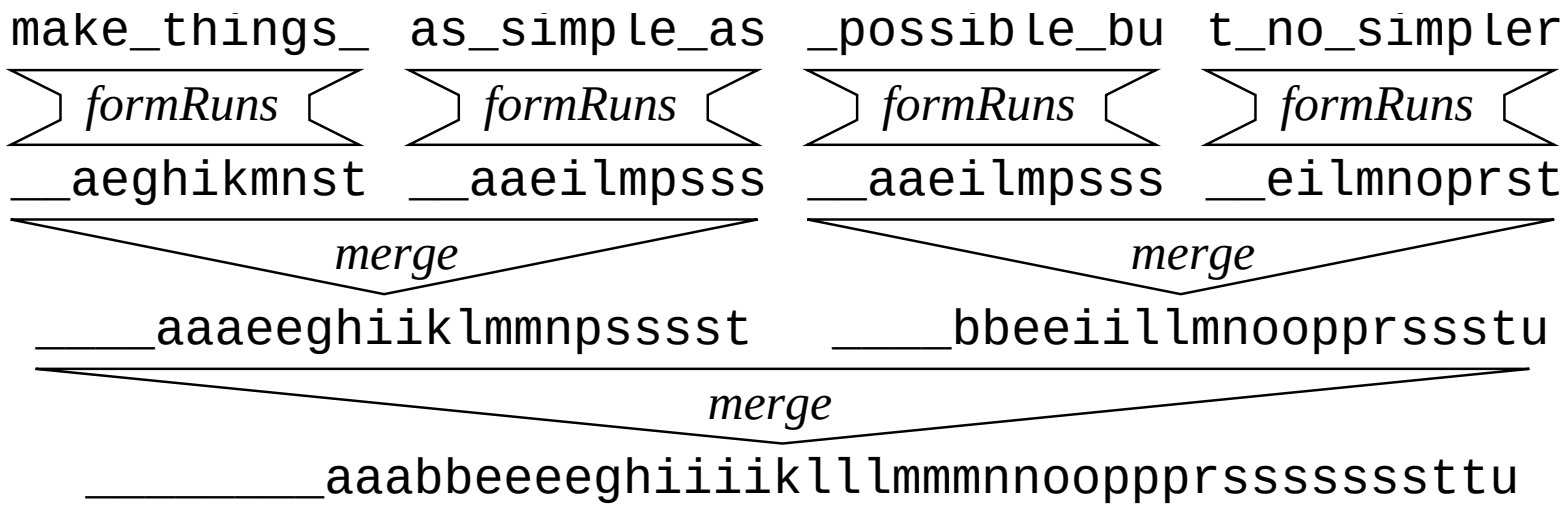
Run Formation

Sortiere Eingabeportionen der Größe M



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

Sortieren durch Externes Binäres Mischen



Procedure externalBinaryMergeSort // I/Os: $\approx$

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 // $\sum: 2\frac{n}{B}\left(1+\left\lceil \log \frac{n}{M} \right\rceil\right)$

Zahlenbeispiel: PC 2010

$n = 2^{39}$ Byte

$M = 2^{33}$ Byte

$B = 2^{22}$ Byte

I/O braucht 2^{-4} s

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

Idee: 7 Durchläufe $\rightsquigarrow$ 2 Durchläufe

Mehrwegemischen

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

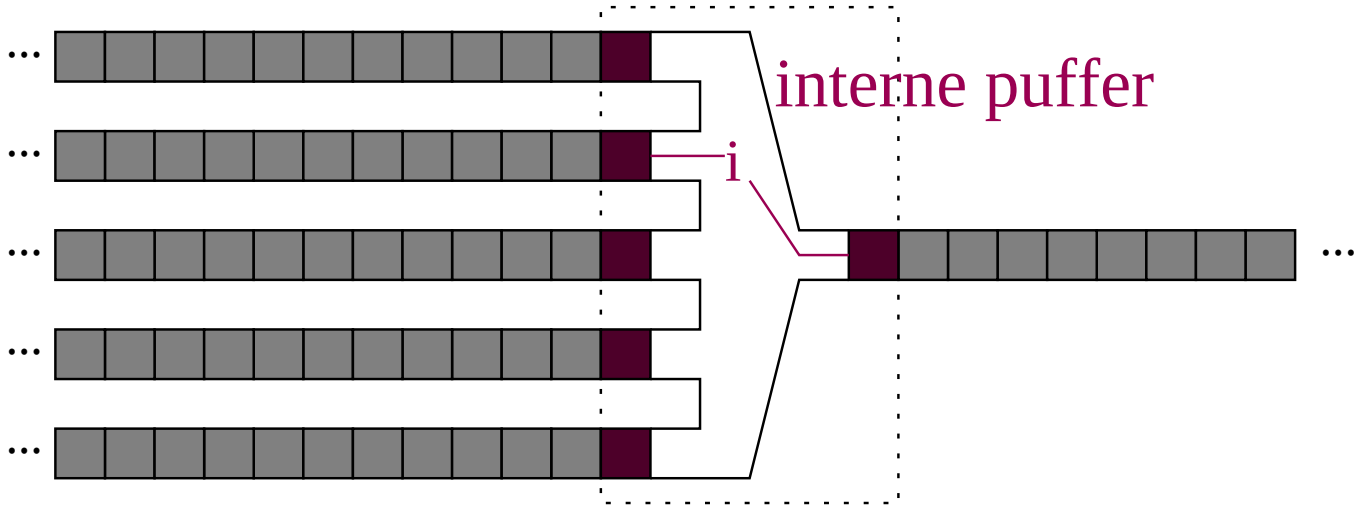
for $i := 1$ **to** k **do** $x_i := a_i$.readElement

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.writeElement(x_i)

$x_i := a_i$.readElement



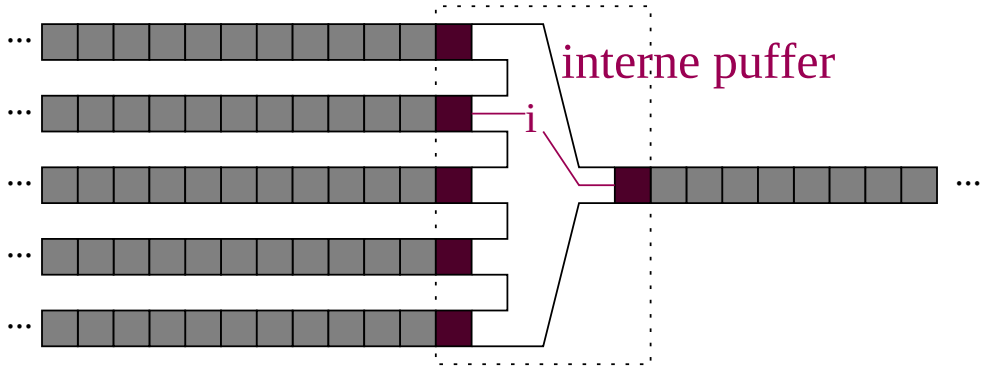
Mehrwegemischen – Analyse

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 !)

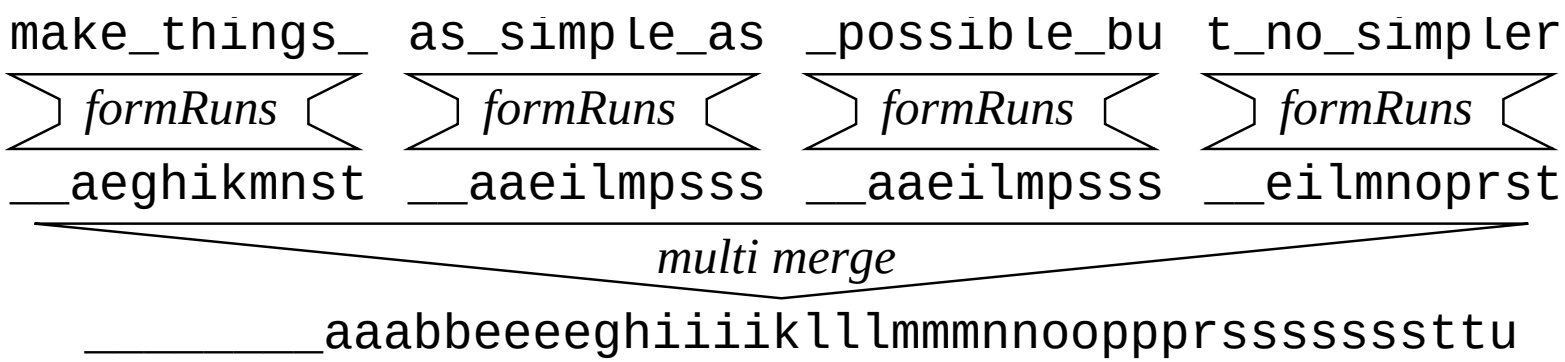
$$\mathrm{O}\left(\log k \sum_{i=1}^{k} |a_i|\right)$$

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:

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

Mehr als eine Mischphase?:

Nicht für Hierarchie Hauptspeicher, Festplatte.

Grund $\overbrace{\frac{M}{B}}^{>1000} > \overbrace{\frac{\text{RAM Euro/bit}}{\text{Platte Euro/bit}}}^{\approx 400}$

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 Sanders2003]

Offene Frage: deterministisch?

Externe Prioritätslisten

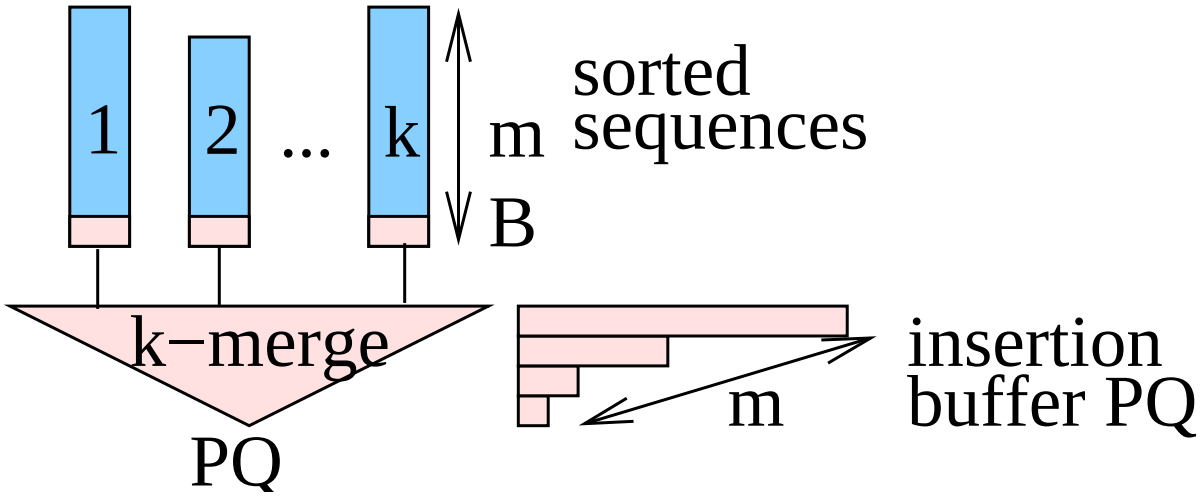
Problem: Binary heaps brauchen

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

Wir hätten gerne:

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

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



Insert: Anfangs in insertion buffer.

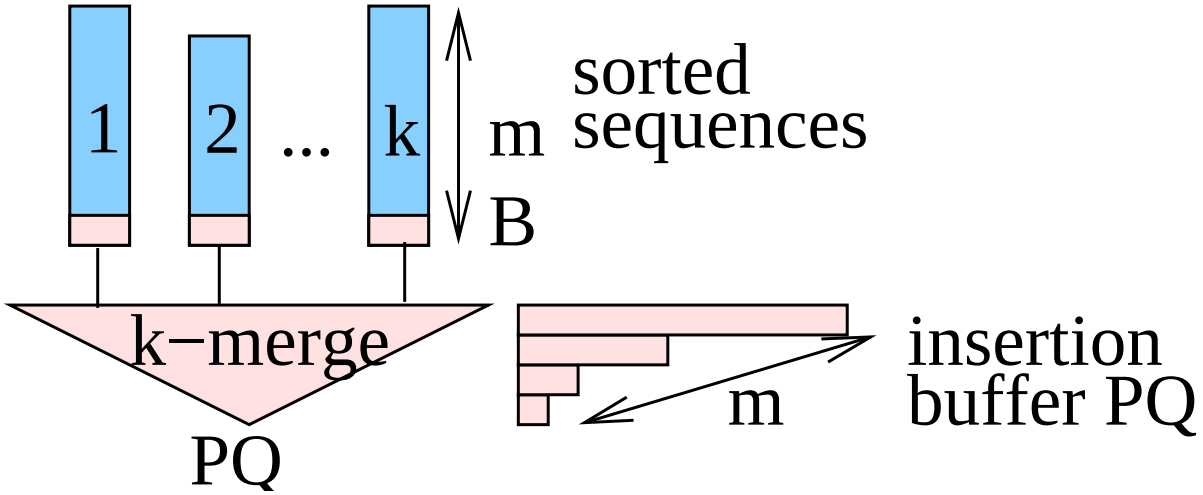
Überlauf $\longrightarrow$

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.



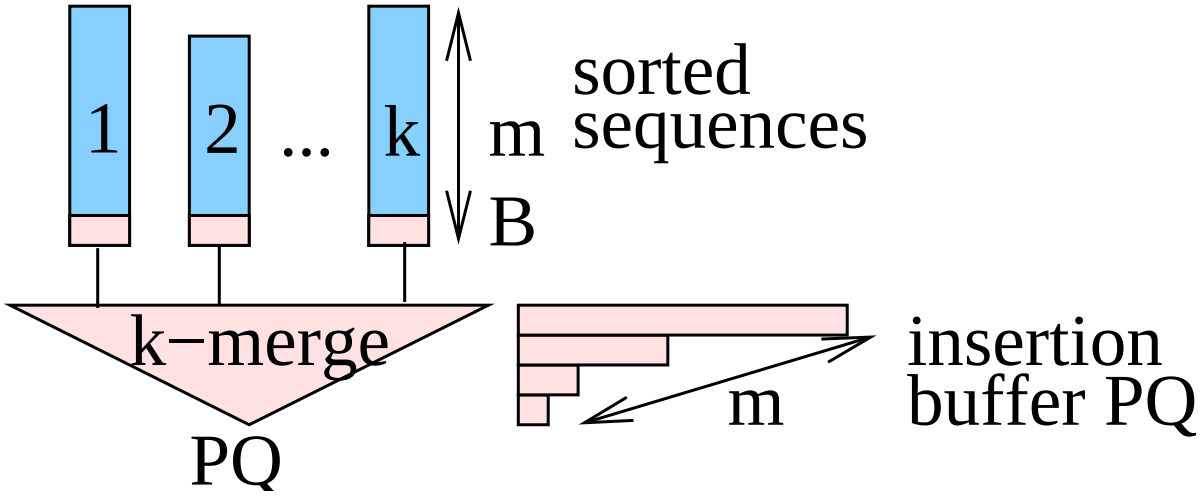
Analyse – Vergleiche (Maß für interne Arbeit)

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

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].

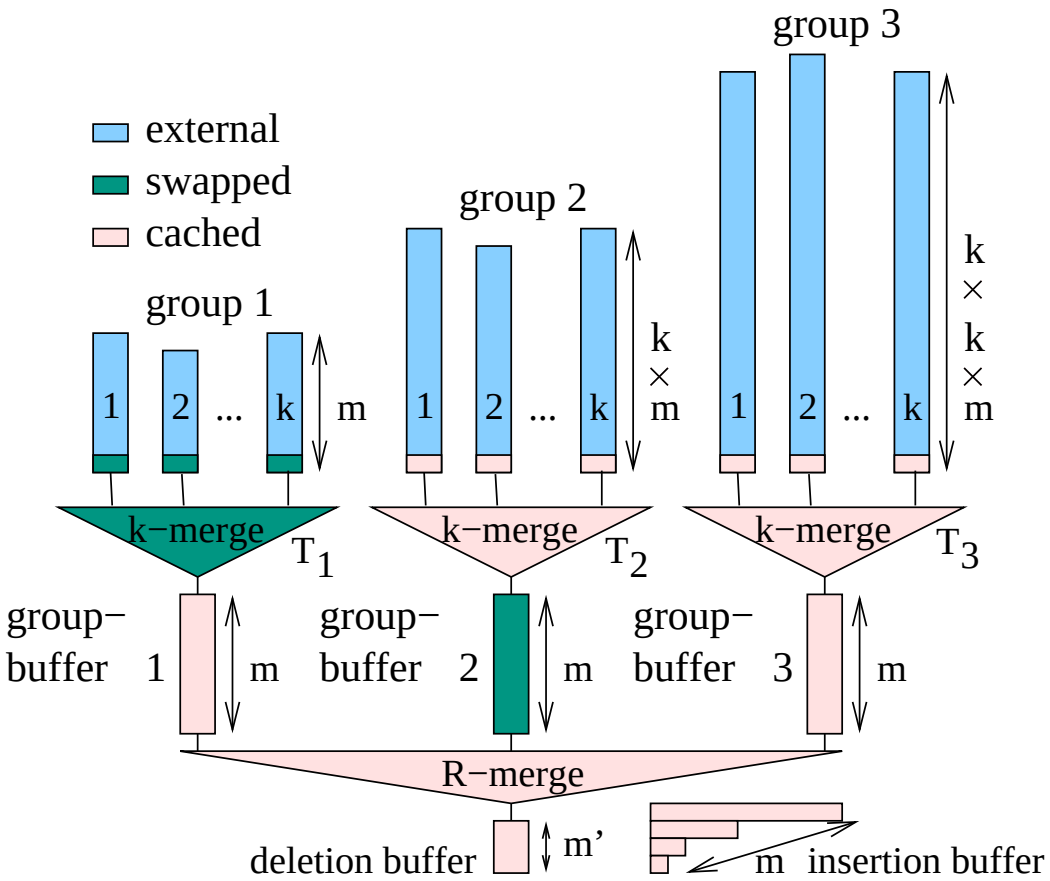
deleteMin:

“amortisiert umsonst”.

Details:

Vorlesung

Algorithm Engineering.



Experiments

Keys: random 32 bit integers

Associated information: 32 dummy bits

Deletion buffer size: 32

Group buffer size: 256

Merging degree k: 128

Near optimal

: performance on

all machines tried!

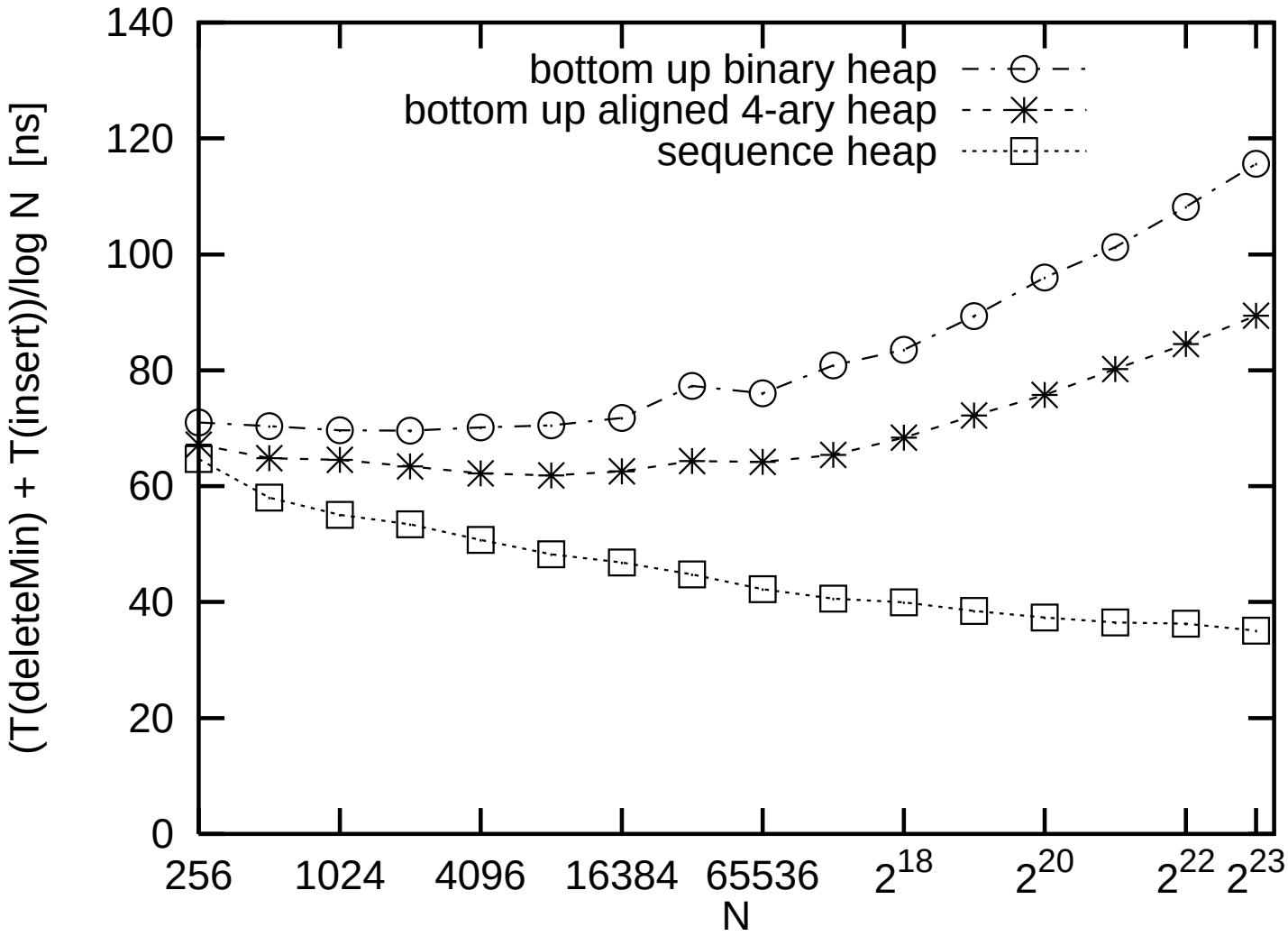
Compiler flags: Highly optimizing, nothing advanced

Operation Sequence:

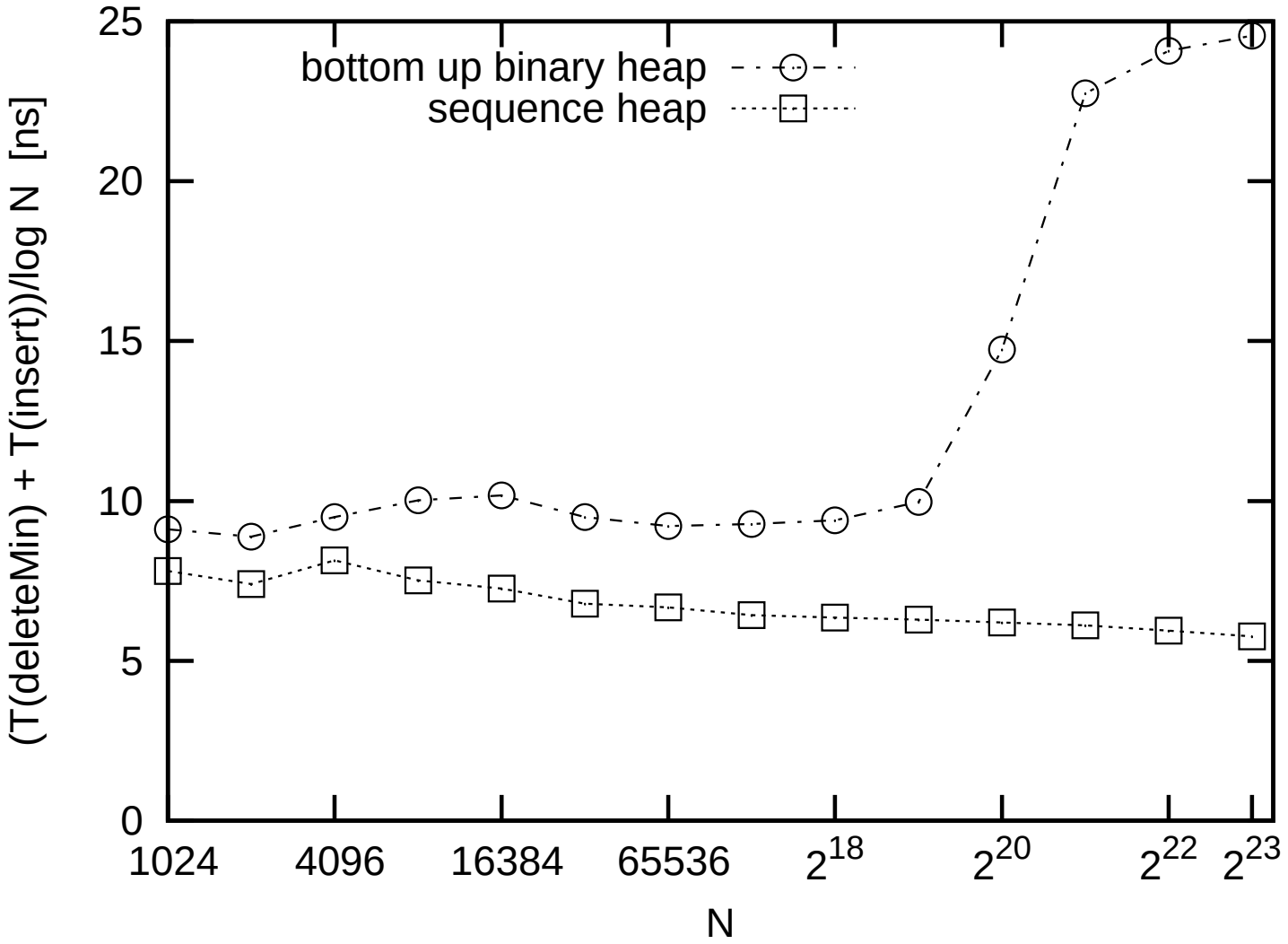
$$\left(\text{Insert-DeleteMin-Insert}\right)^N \left(\text{DeleteMin-Insert-DeleteMin}\right)^N$$

Near optimal performance on all machines tried!

Alpha-21164, 533 MHz



Core2 Duo Notebook, 1.??? GHz



Minimale Spannbäume

Semiexterner 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

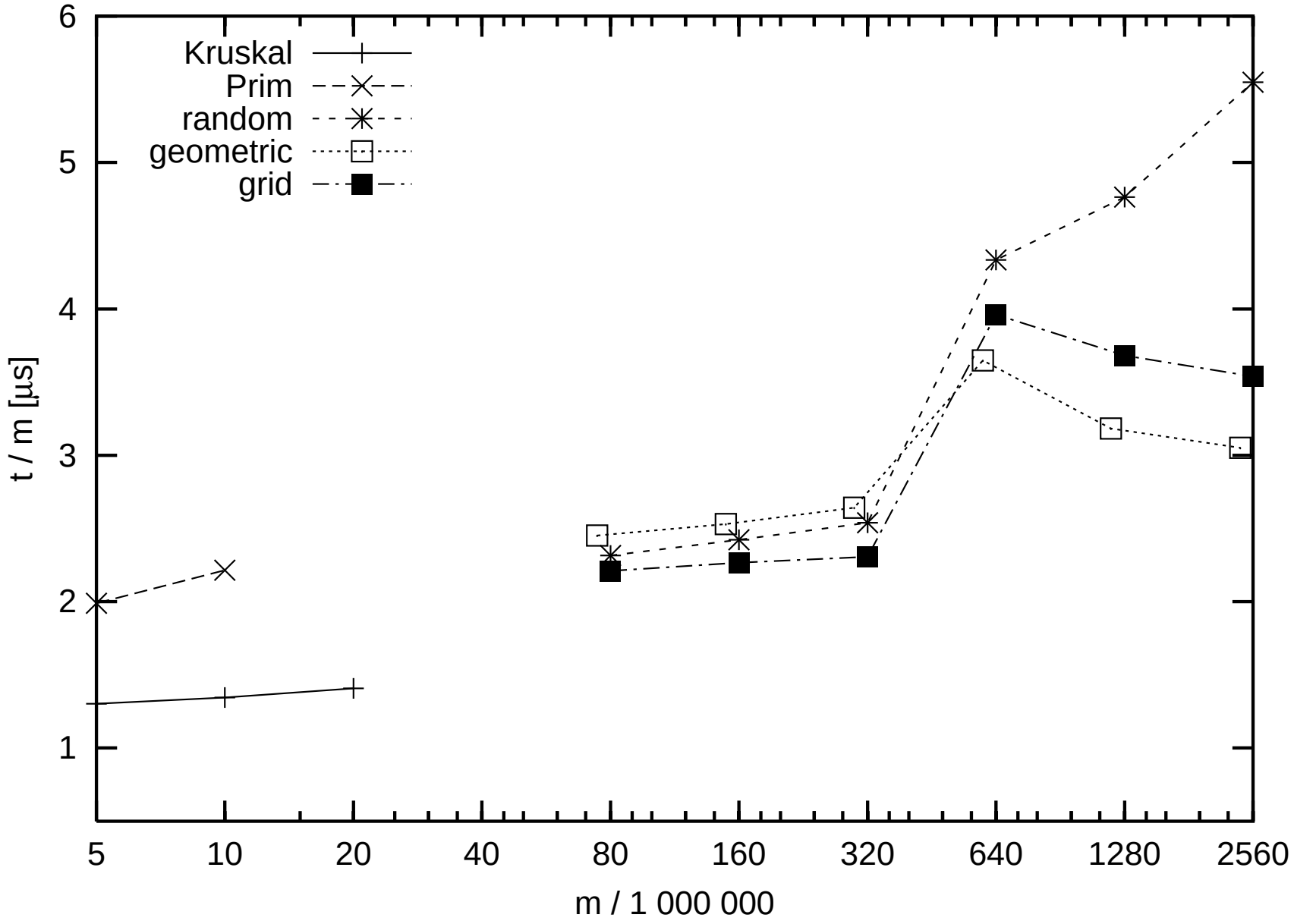
- Reduziere Knotenzahl mittels Kontraktion von MST-Kanten

  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 semiexternen Fall

kürzeste Wege: ähnlich BFS.

8 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 Algorithmik II

- Multicore Basisalgorithmen (Singler, EU-Projekt PEPPHER)

  $\rightsquigarrow$ z.B. g++ STL parallel mode

- GPU Algorithmen (Osipov, EU-Projekt PEPPHER)

  $\rightsquigarrow$ z.B. schnellster vergleichsbasierter Sortierer

- Parallele Externe Algorithmen (Daten auf Festplatte) (Osipov, Singler,. . .)

  $\rightsquigarrow$ Diverse Sortierbenchmarks

- Hauptspeicherbasierte Datenbanken (Kooperation SAP)

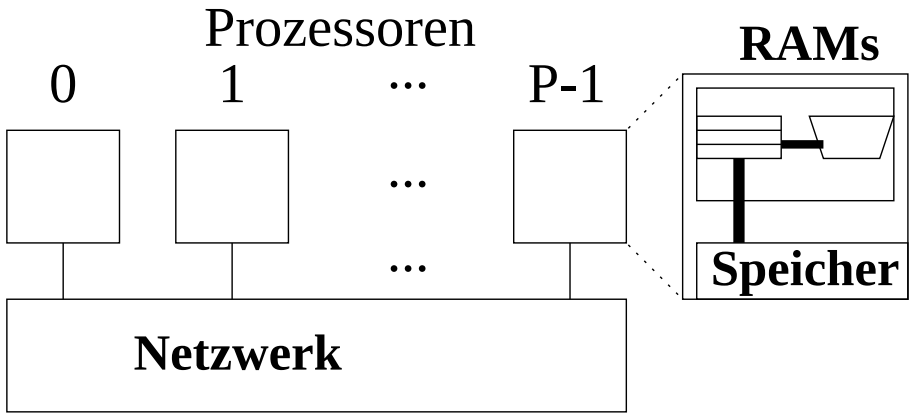
- Graphpartitionierung (Schulz, Osipov)

  $\rightsquigarrow$ z.T. beste bekannte Ergebnisse in Standardbenchmark

- Lastbalancierung, DFG TransRegio Invasic, Jochen Speck

8.1 Modell

Nachrichtengekoppelte Parallelrechner



- Prozessoren sind RAMs
- asynchrone Programmabarbeitung
- Interaktion durch Nachrichtenaustausch

Kostenmodell für Nachrichtenaustausch

Jedes PE kann

gleichzeitig maximal eine Nachricht senden und empfangen.

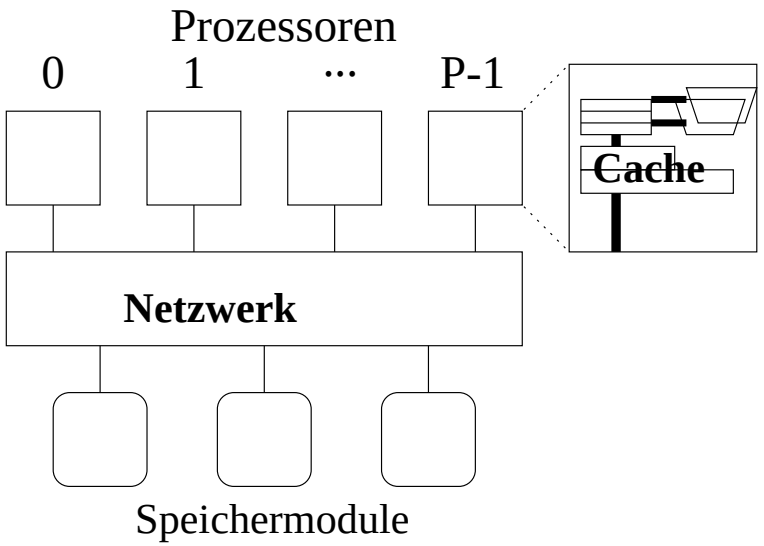
Bei Nachrichtenlänge ℓ dauert das

$$T_{\text{comm}}(\ell) = T_{\text{start}} + \ell T_{\text{byte}}$$

- 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 Parallellismus 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.

(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?).

8.2 Beispiel: Assoziative Operationen (=Reduktion)

Satz 20. *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?)

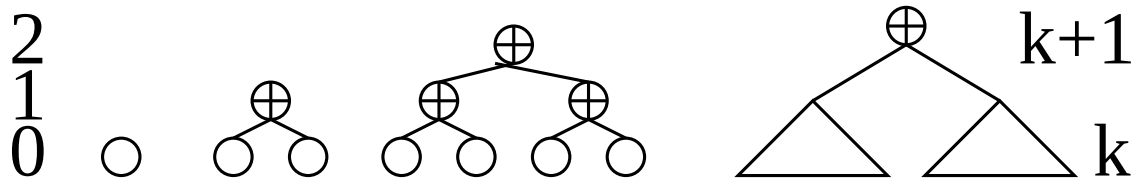
Induktion über k:

$k = 0$: trivial

$k \rightsquigarrow k+1$:

$$\bigoplus_{i<2^{k+1}} x_i = \underbrace{\overbrace{\bigoplus_{i<2^k} x_i}^{\text{Tiefe } k} \oplus \overbrace{\bigoplus_{i<2^k} x_{i+2^k}}^{\text{Tiefe } k \text{ (IA)}}}_{\text{Tiefe } k+1}$$

■



Pseudocode

PE index $i \in \{0,\ldots,p-1\}$

// 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 $i - 2^k$

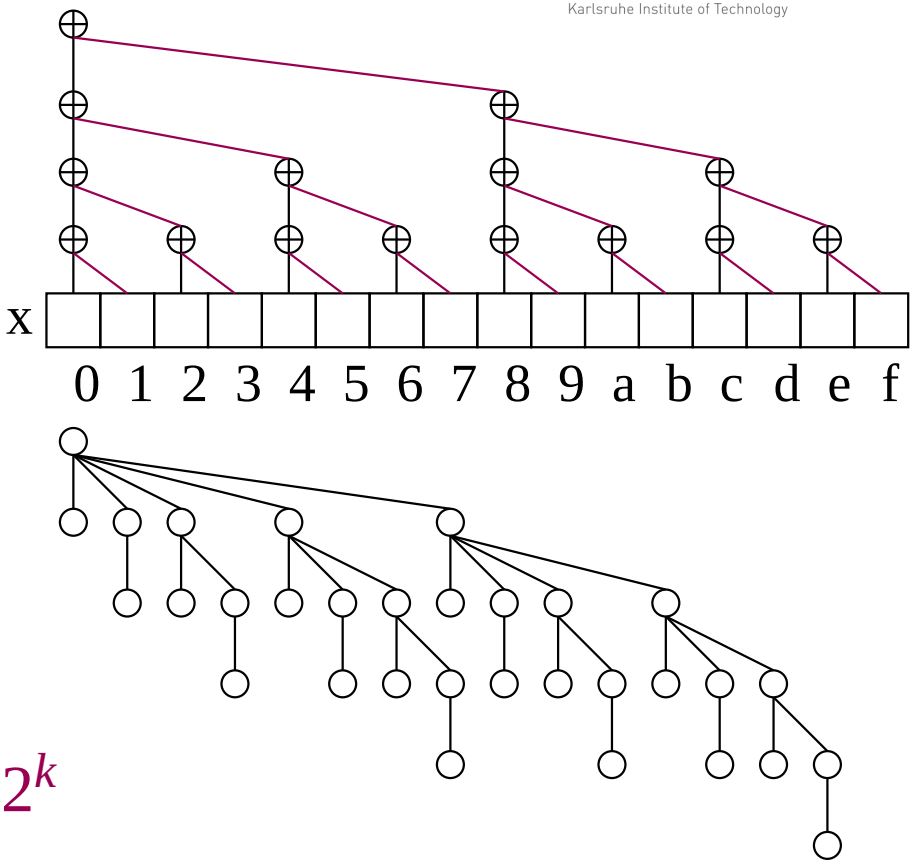
      active := 0

    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



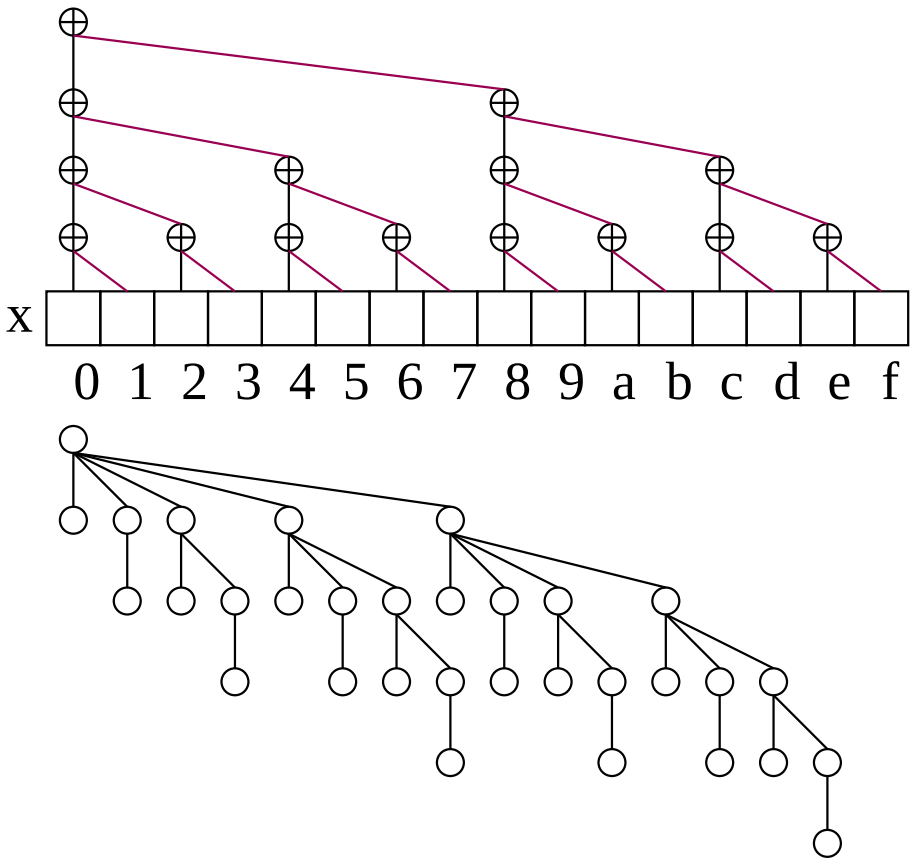
Analyse

n PEs

Zeit $O(\log n)$

Speedup $O(n/\log n)$

Effizienz $O(1/\log n)$



Weniger ist Mehr

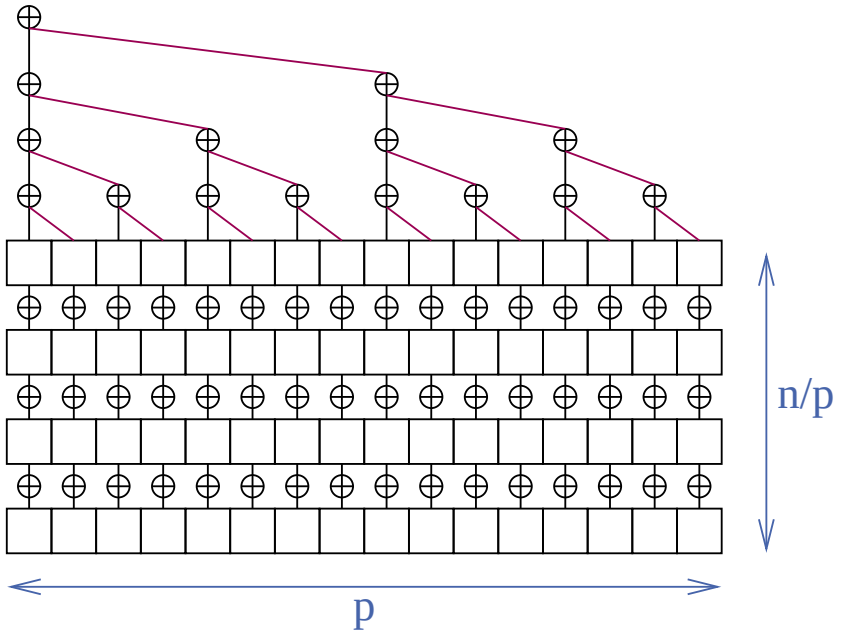
p PEs

Jedes PE addiert

n/p Elemente sequentiell

Dann parallele Summe

für p Teilsummen



Zeit $T_{\text{seq}}(n/p) + \Theta(\log p)$

Effizienz

$$\frac{T_{\text{seq}}(n)}{p(T_{\text{seq}}(n/p) + \Theta(\log p))} = \frac{1}{1 + \Theta(p\log(p))/n} = 1 - \Theta\left(\frac{p\log p}{n}\right)$$

falls $n \gg p\log p$

Diskussion Reduktionsoperation

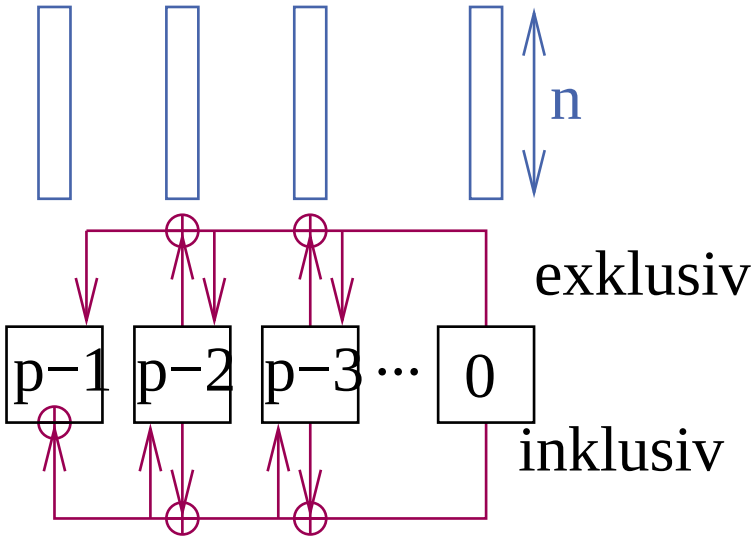
- Binärbaum führt zu logarithmischer Ausführungszeit
- Nützlich auf den meisten Modellen
- Brent's Prinzip: Ineffiziente Algorithmen werden durch Verringerung der Prozessorzahl effizient

Präfixsummen

Gesucht

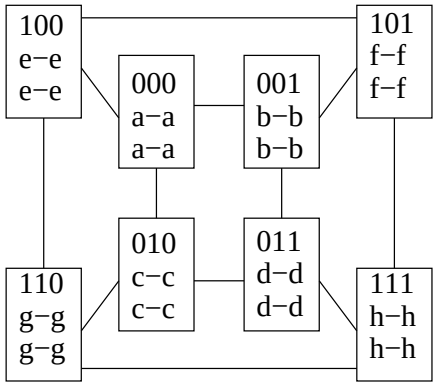
$$x@i := \bigotimes_{i' \leq i} m@i'$$

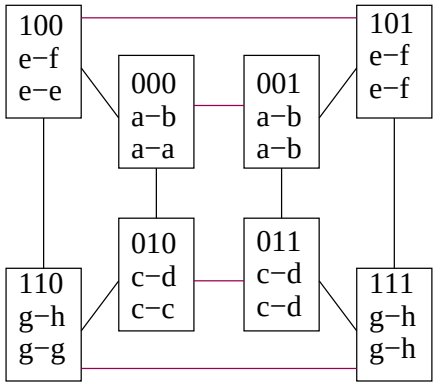
(auf PE i) Objekte der Länge ℓ



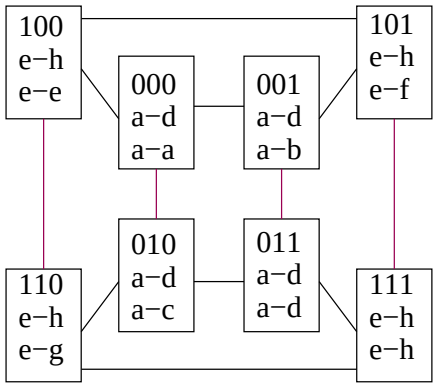
inhärent sequentiell ???

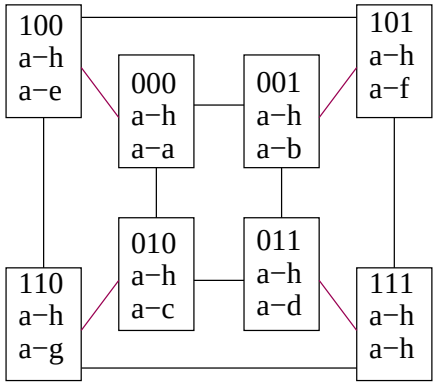
Hyperwürfelalgorithmus





i
sum
x





// view PE index i as a

// d-bit bit array

Function hcPrefix(m)

$x := \sigma := m$

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

invariant $\sigma = \bigotimes_{j=i[k..d-1]0^k}^{i[k..d-1]1^k} \text{m@j}$

invariant $x = \bigotimes_{j=i[k..d-1]0^k}^{i} \text{m@j}$

$y := \sigma@(i \oplus 2^k)$ // sendRecv

$\sigma := \sigma \otimes y$

if $i[k] = 1$ **then** $x := x \otimes y$

return x

Analyse

$$T_{\text{prefix}} = \mathrm{O}\big((T_{\text{start}} + \ell T_{\text{byte}}) \log p\big)$$

Problemchen:

Nichtoptimal bei $\ell T_{\text{byte}} > T_{\text{start}}$ (analog schon bei Reduktion)

siehe Spezialvorlesung

$\rightsquigarrow \mathrm{O}\big(T_{\text{start}} \log P + \ell T_{\text{byte}}\big)$

8.3 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

$$d_0 \leq \cdots \leq d_k = v \leq d_{k+1} \leq \cdots \leq d_{n-1}$$

qSort$([d_0, \ldots, d_{k-1}], k)$

qSort$([d_{k+1}, \ldots, d_{n-1}], n-k-1)$

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 durchnummerieren (Präfix-Summe)
5. Daten umverteilen
6. Prozessoren aufspalten
7. Parallele 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 $p' + i - j$ // $i - j = \sum_{k=0}^{i} d@k > v$

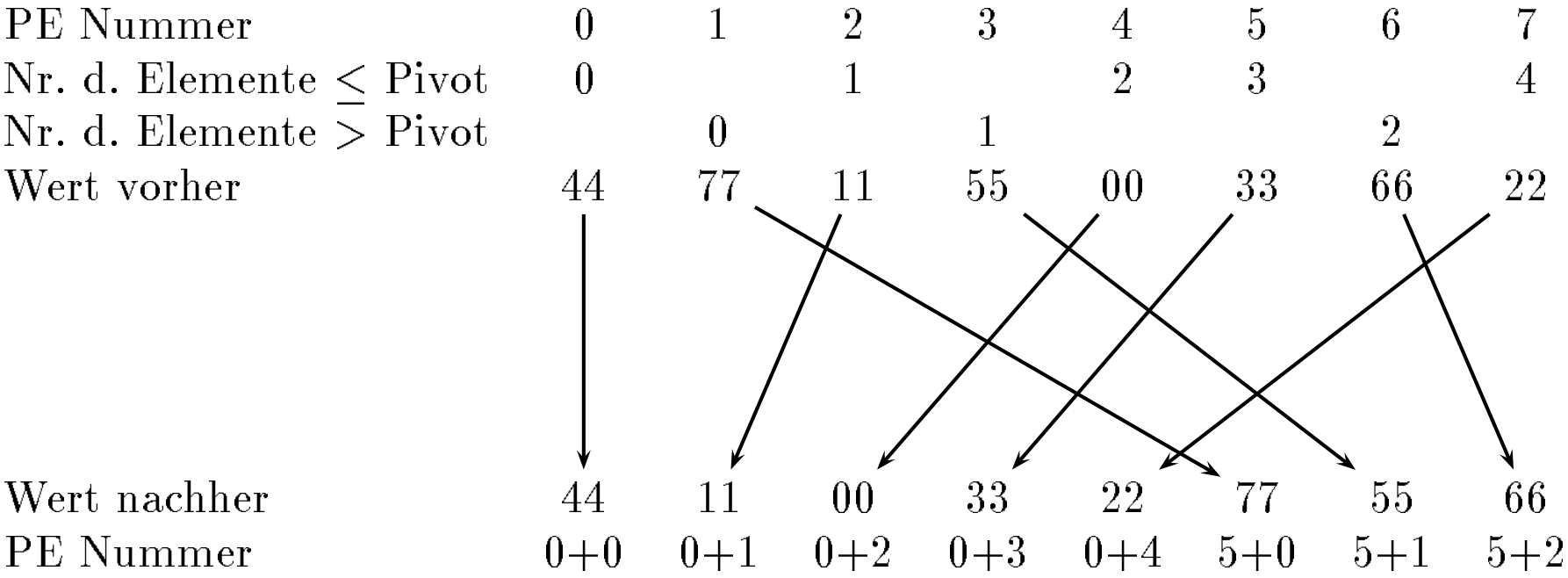
receive d

if $i < p'$ **then** join left partition; qsort(d, i, p')

else join right partition; qsort$(d, i - p', p - p')$

Beispiel

pivot $v = 44$



```
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
  - $\rightsquigarrow$ Zeit $O(T_{\text{start}} \log p)$
- erwartete Rekursionstiefe: $O(\log p)$

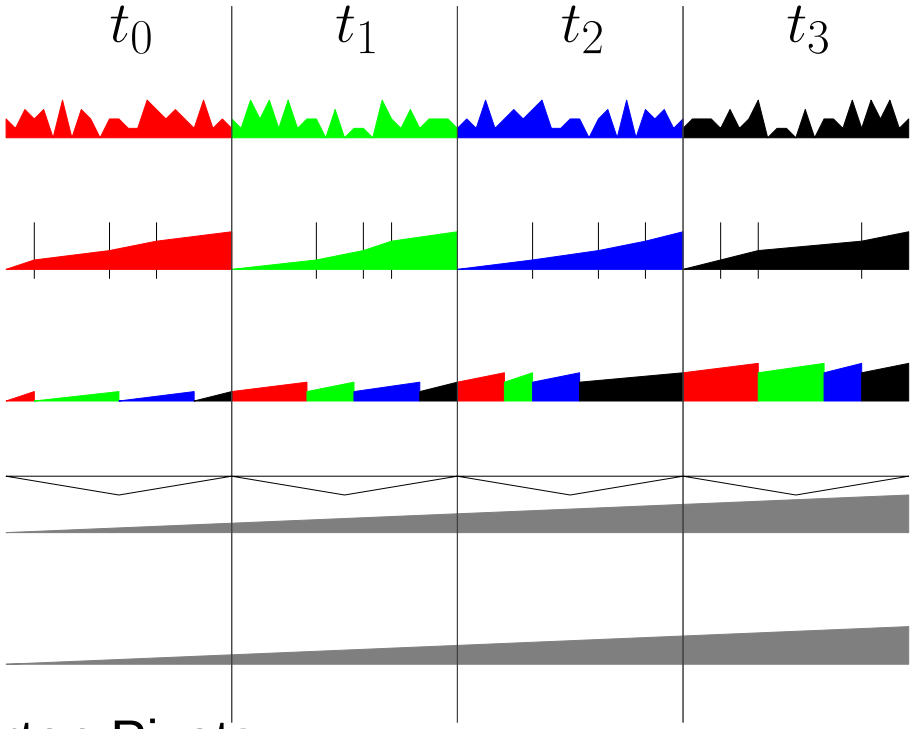
Erwartete Gesamtzeit: $O\left(T_{\text{start}} \log^2 p\right)$

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 $\mathrm{O}\left(\frac{n \log n}{p} + \log^2 p\right)$ möglich
- Bei verteiltem Speicher stört, dass jedes Element $\Omega(\log p)$ mal transportiert wird.
  $\rightsquigarrow \cdots \rightsquigarrow$ Zeit $\mathrm{O}\left(\frac{n}{p}(\log n + T_{\text{byte}} \log p) + T_{\text{start}} \log^2 p\right)$

Paralleles Sortieren durch Mehrwegemischen

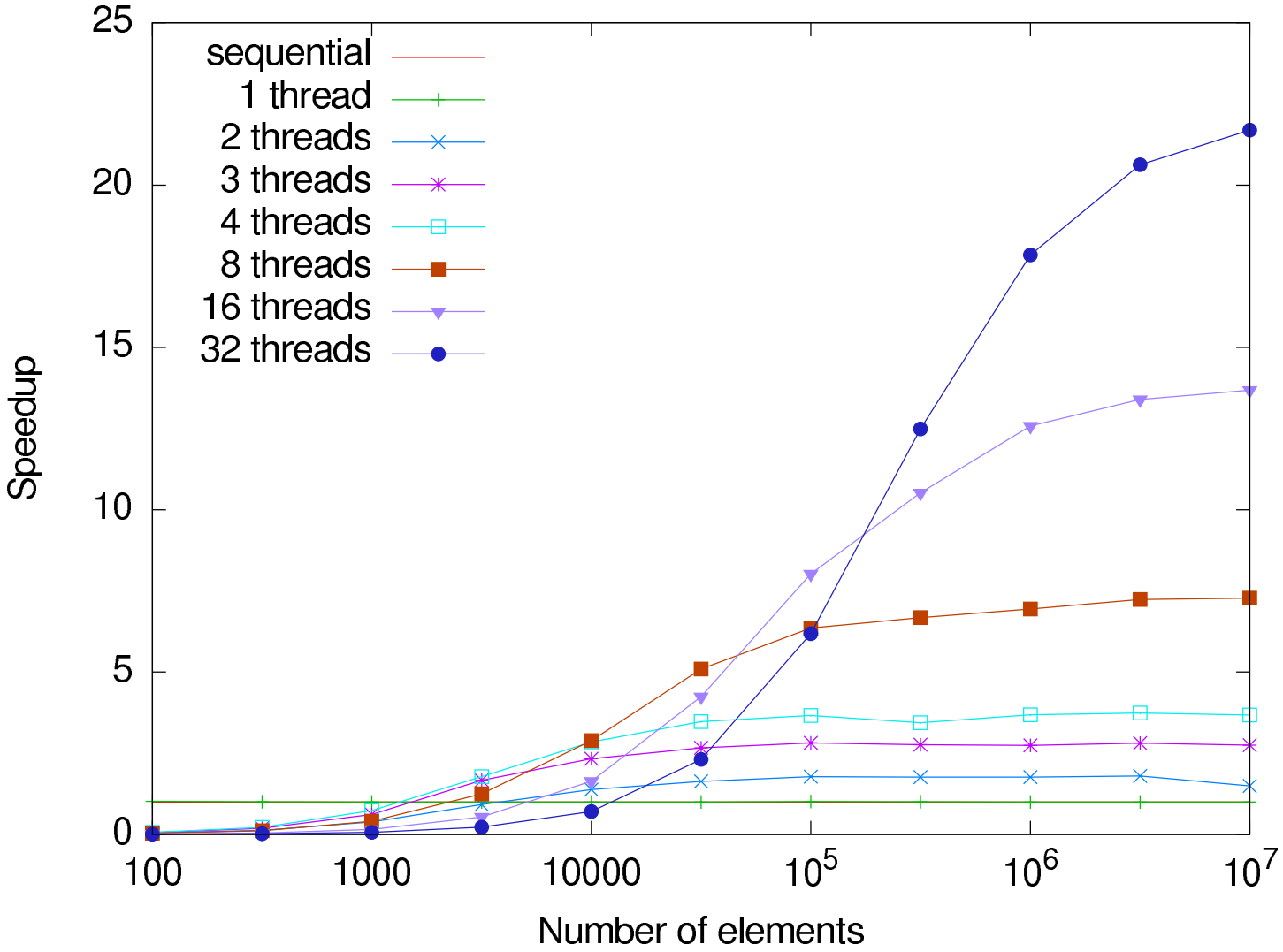
1. p Prozessoren sortieren
   je n/p Elemente lokal
2. Finde pivots so dass
   n/p Elemente zwischen
   zwei benachbarten pivots liegen
3. Jeder Prozessor macht
   Mehrwegemischen für alle
   Elemente zwischen zwei benachbarten Pivots.



Mehr in “Parallele Algorithmen”

Messungen Sparc T1 – 8 Kerne, 128 bit Elemente

[Sanders Singler 2007]



Mehr zu parallelem Sortieren

- Theoretikeralgorithmen mit Laufzeit $\mathrm{O}(\log p)$ bei $p = n$
- Praktikable Algorithmen mit Laufzeit $\mathrm{O}(\log p)$ für $n = \mathrm{O}(\sqrt{p})$ –
  wenn schnell wichtiger als Effizient ist, z.B. base case
- Sample sort: k-Wegeverallgemeinerung von quicksort
- Paralleles externes Sortieren

Mehr zu parallelen Algorithmen – Parallelisierung der Basic Toolbox ?

Verteilte Hashtabellen: geht aber teuer $\rightsquigarrow$ lieber vermeiden

Prioritätslisten, Suchbäume: ähnliches Problem. Am ehesten mit batched updates

BFS: OK bei kleinem Graphdurchmesser

DFS: $\approx$ inhärent nichtparallelisierbar

kürzeste Wege: ähnlich BFS. Aber all-to-all, Vorberechnungen OK

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

9 Stringology (Zeichenkettenalgorithmen)

- Strings sortieren
- Patterns suchen
  - Pattern vorverarbeiten
  - Text vorverarbeiten
    - Invertierte Indizes
    - Suffix Arrays
- Datenkompression

9.1 Strings Sortieren

multikey quicksort

Function mkqSort(s : Sequence **of** String, i : $\mathbb{N}$) : Sequence **of** String

assert $\forall e, e' \in s : e[1..i-1] = e'[1..i-1]$

if $|s| \leq 1$ **then return** s // base case

pick $p \in s$ uniformly at random // pivot character

return concatenation of mkqSort($\langle e \in s : e[i] < p[i] \rangle, i$),
mkqSort($\langle e \in s : e[i] = p[i] \rangle, i+1$), and
mkqSort($\langle e \in s : e[i] > p[i] \rangle, i$)

- Laufzeit: $O(|s| \log |s| + \sum_{t \in s} |t|)$
- genauer: $O(|s| \log |s| + d)$ (d: Summe der eindeutigen Präfixe)
- Tutorium: in-place!

9.2 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: $O(m \log n) \ldots O(m)$

Suffixtabellen

aus

Linear Work Suffix Array Construction

Juha Kärkkäinen, Peter Sanders, Stefan Burkhardt

Journal of the ACM

Seiten 1–19, Nummer 6, Band 53.

Etwas “Stringology”-Notation

String S: Array $S[0..n) := S[0..n-1] := [S[0], \ldots, S[n-1]]$

von Buchstaben

Suffix: $S_i := S[i..n)$

Endmarkierungen: $S[n] := S[n+1] := \cdots := 0$

0 ist kleiner als alle anderen Zeichen

Suffixe Sortieren

Sortiere die Menge $\{S_0, S_1, \ldots, S_{n-1}\}$
von Suffixen des Strings S der Länge n
(Alphabet $[1, n] = \{1, \ldots, n\}$)
in lexikographische Reihenfolge.

- suffix $S_i = S[i, n]$ für $i \in [0..n-1]$

$S =$ banana:

0	banana
1	anana
2	nana
3	ana
4	na
5	a

$\Longrightarrow$

5	a
3	ana
1	anana
0	banana
4	na
2	nana

Anwendungen

- Volltextsuche
- Burrows-Wheeler Transformation (`bzip2` Kompressor)
- Ersatz für kompliziertere Suffixbäume
- Bioinformatik: Wiederholungen suchen,. . .

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

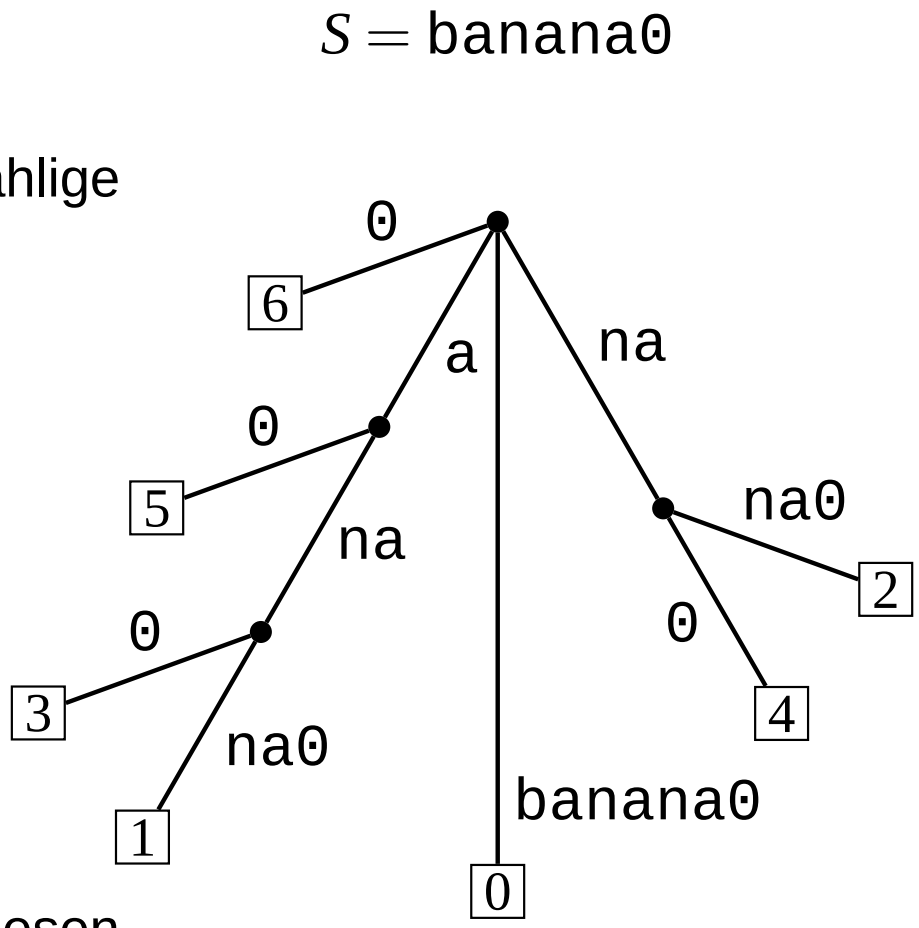
zwischen verglichenen Zeichenketten vorberechnen

Suffix-Baum: $O(m)$ kann aus SA berechnet werden

Suffix-Baum

[Weiner '73][McCreight '76]

- ☐ kompaktierter Trie der Suffixe
- \+ 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



Alphabet-Modell

Geordnetes Alphabet: Zeichen können nur verglichen werden

Konstante Alphabetgröße: endliche Menge

deren Größe nicht von n abhängt.

Ganzzahliges Alphabet: Alphabet ist $\{1,\ldots,\sigma\}$

für eine ganze Zahl $\sigma \geq 2$

Geordnetes $\rightarrow$ ganzzahliges Alphabet

Sortiere die Zeichen von S

Ersetze $S[i]$ durch seinen Rang

```
012345    135024
banana -> aaabnn
213131 <- 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

Ein erster Teile-und-Herrsche-Ansatz

1. $SA^1 =$sort $\{S_i : i$ ist ungerade$\}$ (Rekursion)
2. $SA^0 =$sort $\{S_i : i$ 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^1 berechnen

- Erstes Zeichen weglassen.

  banana $\rightarrow$ anana

- Ersetze Buchstabenpaare durch Ihre lexikographischen Namen

  $\boxed{\texttt{an}}\boxed{\texttt{an}}\boxed{\texttt{a}\$}\rightarrow 221$

- Rekursion

  $\langle 1, 21, 221\rangle$

- Rückübersetzen

  $\langle \texttt{a}, \texttt{ana}, \texttt{anana}\rangle$

Berechne SA^0 aus SA^1

1	anana
3	ana
5	a

$\Longrightarrow$

5	a
3	ana
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)
2	n 2(ana)
4	n 1(a)

$\Longrightarrow$

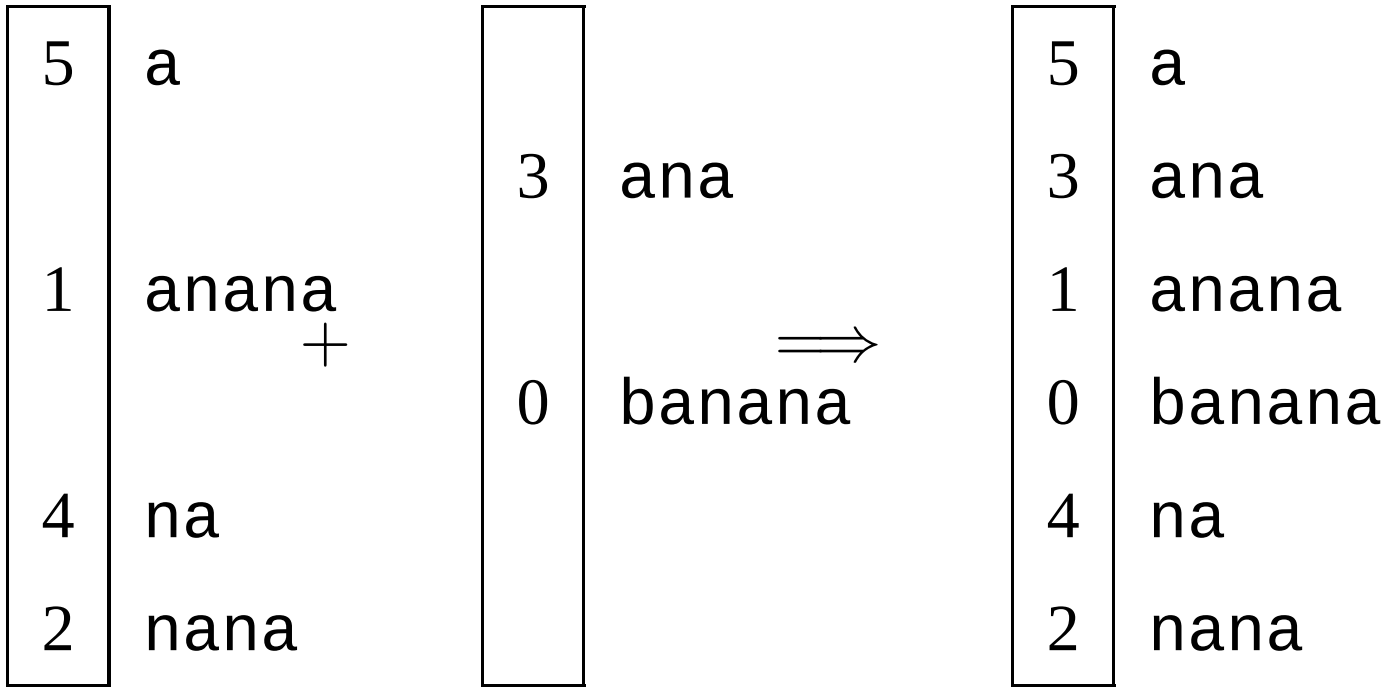
0	banana
4	na
2	nana

Radix-Sort

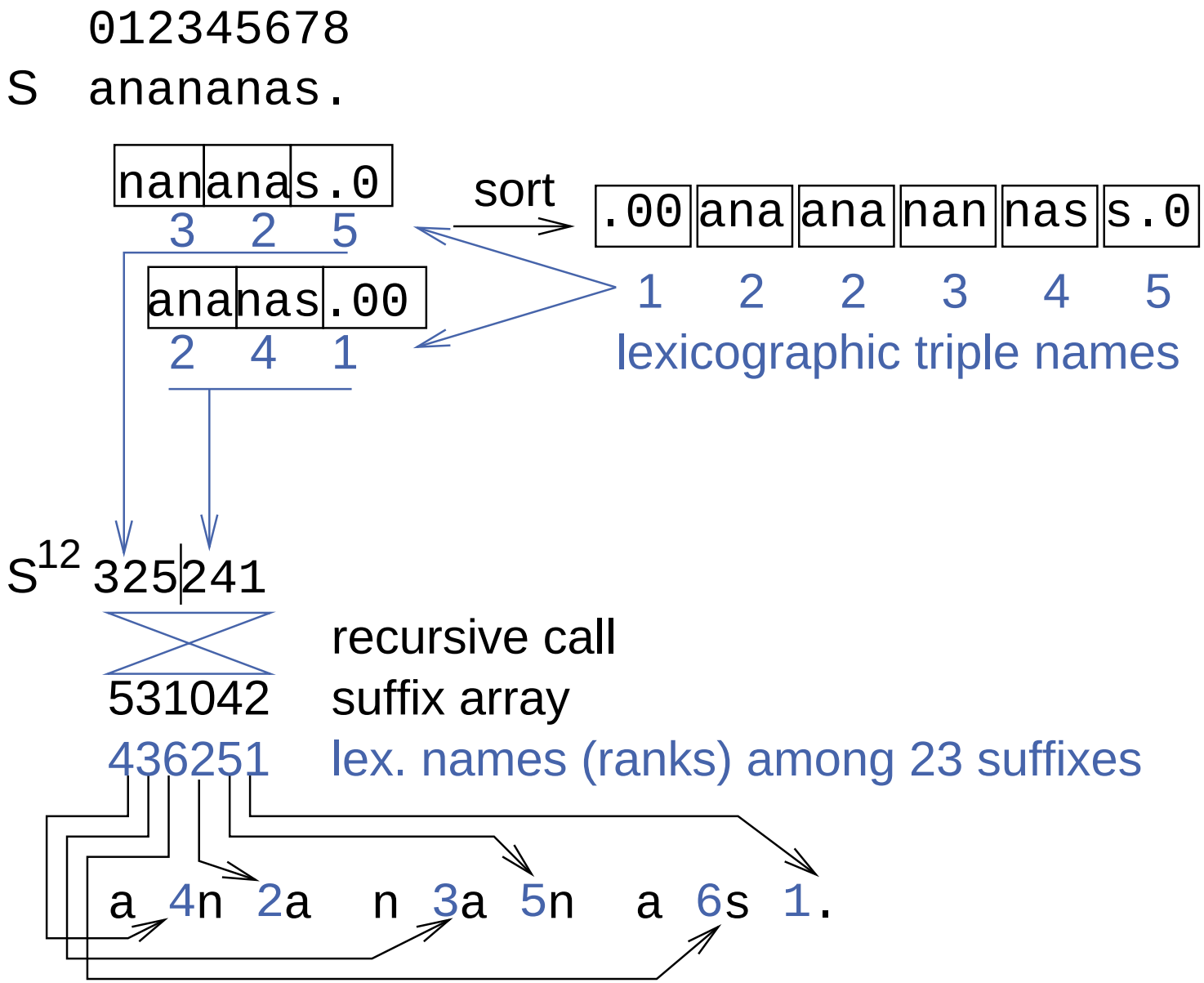
Asymmetrisches Divide-and-Conquer

1. $SA^{12} =$sort $\{S_i : i \bmod 3 \neq 0\}$ (Rekursion)
2. $SA^0 =$sort $\{S_i : i \bmod 3 = 0\}$ (einfach mittels SA^{12})
3. Mische SA^{12} und SA^0 (einfach!)

$$S = \text{banana}$$



Rekursion, Beispiel



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^{12}_i von S^{12} repräsentiert S_{3i+1}
  Suffix $S^{12}_{n/3+i}$ von S^{12} repräsentiert S_{3i+2}

- Rekursion auf (S^{12}) (Alphabetgröße $\leq 2n/3$)
- Annotiere die 23-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 $\text{key}(a[i]) \in [0..n]$

$c[0..n] := [0,\ldots,0]$ Zähler

for $i \in [0..n)$ do $c[a[i]]{+}{+}$ zähle

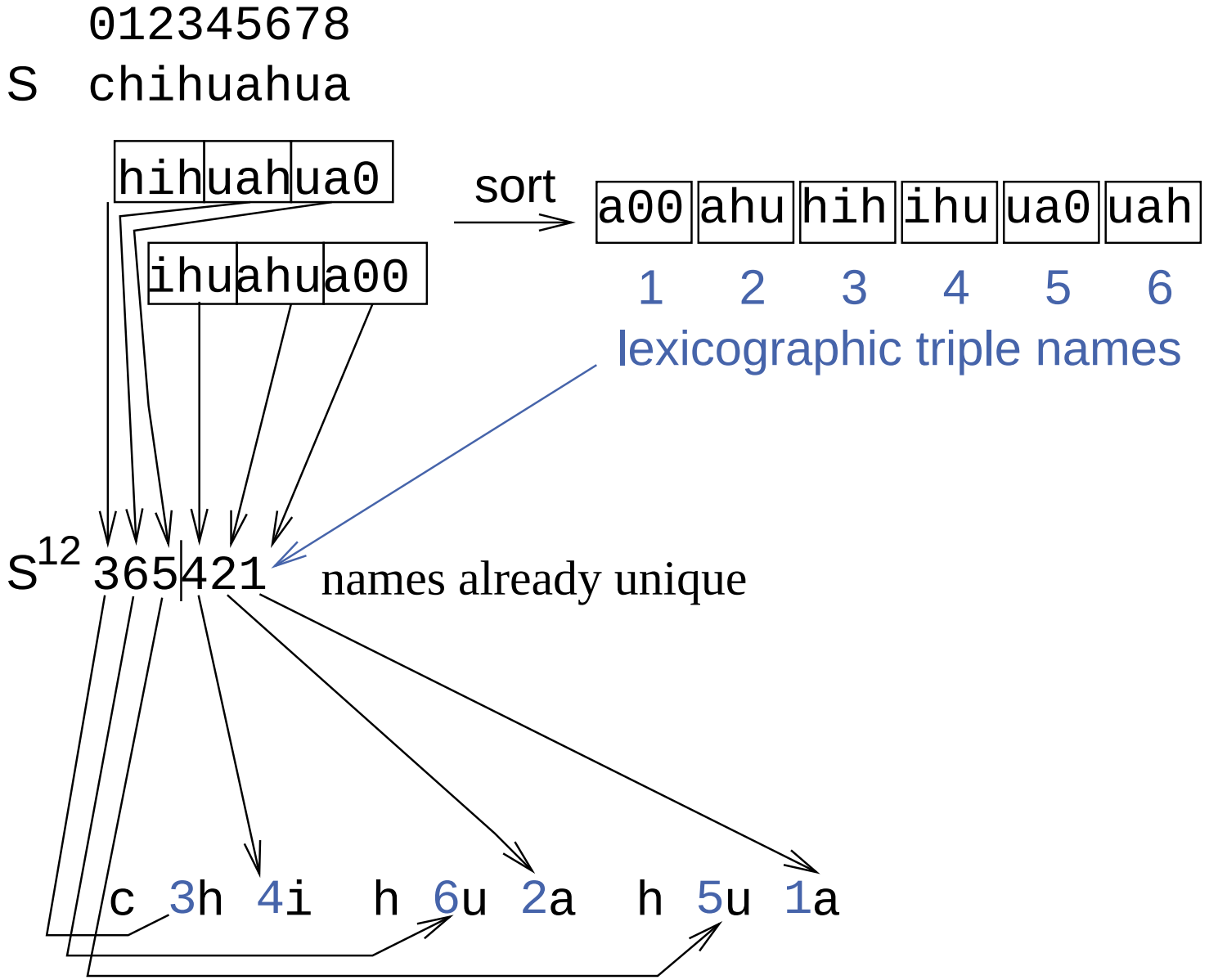
$s := 0$

for $i \in [0..n)$ do $(s, c[i]) := (s + c[i], s)$ Präfixsummen

for $i \in [0..n)$ do $b[c[a[i]]{+}{+}] := a[i]$ bucket sort

Zeit $O(n)$!

Rekursions-Beispiel: Einfacher Fall



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 6u 2(ahua)
6: h 5u 1(a)
0: c 3h 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 00 0(0)
```

$\Downarrow$

```
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 occurences
  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 y 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

LCP-Array

speichert Längen der längsten gemeinsamen Präfixe lexikographisch benachbarter Suffixe!

$S =$ banana:

0	banana
1	anana
2	nana
3	ana
4	na
5	a

SA =

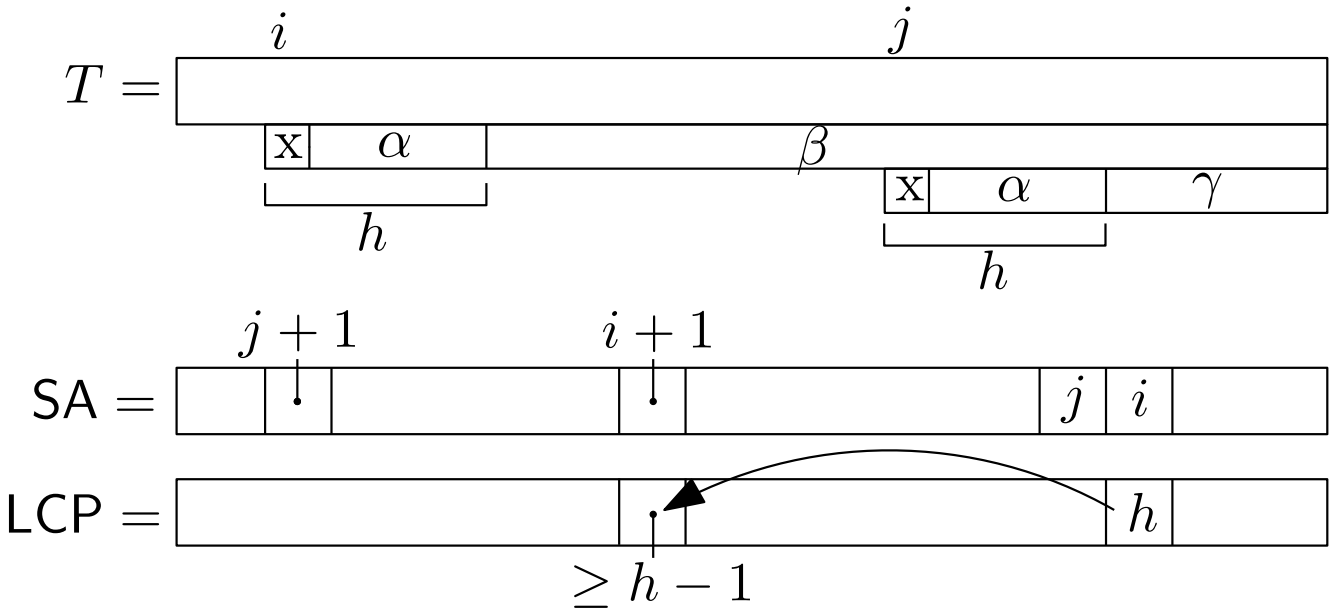
5	a
3	ana
1	anana
0	banana
4	na
2	nana

LCP =

0	a
1	**a**na
3	**ana**na
0	banana
0	na
2	**na**na

LCP-Array: Berechnung

- naiv $O(n^2)$
- inverses Suffix-Array: $\mathrm{SA}^{-1}[\mathrm{SA}[i]] = i$ (wo steht i in SA?)
- For all $1 \leq i < n$: $\mathrm{LCP}[\mathrm{SA}^{-1}[i+1]] \geq \mathrm{LCP}[\mathrm{SA}^{-1}[i]] - 1$.



LCP-Array: Berechnung

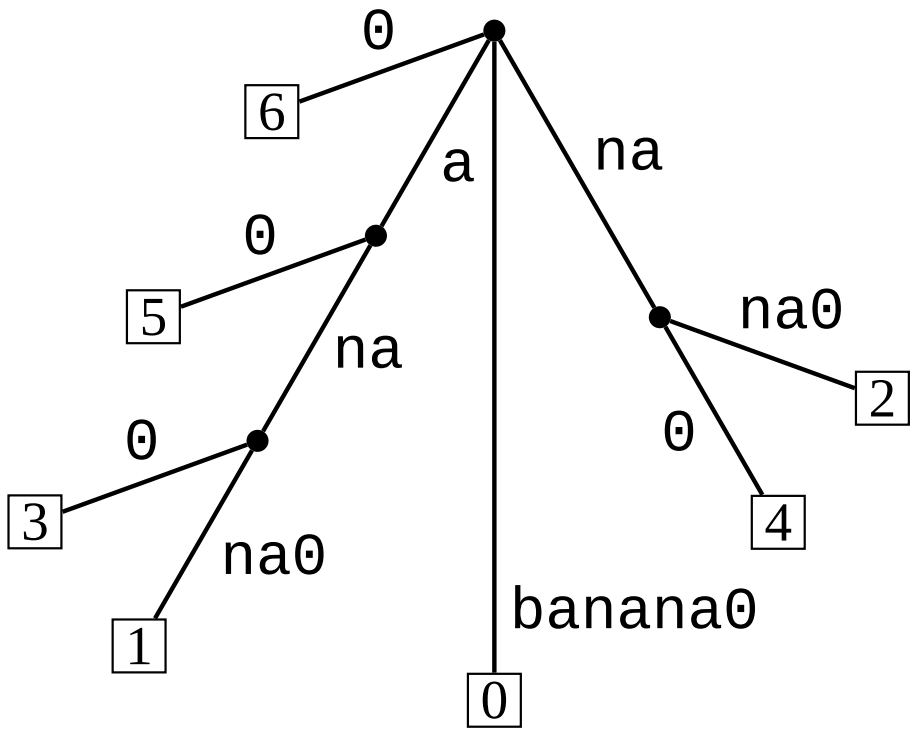
- For all $1 \leq i < n$: $\mathrm{LCP}[\mathrm{SA}^{-1}[i+1]] \geq \mathrm{LCP}[\mathrm{SA}^{-1}[i]] - 1$.

$h := 0, \mathrm{LCP}[1] := 0$
for $i = 1, \ldots, n$ **do**
  if $\mathrm{SA}^{-1}[i] \neq 1$ **then**
    while $t_{i+h} = t_{\mathrm{SA}[\mathrm{SA}^{-1}[i]-1]+h}$ **do** $h{+}{+}$
    $\mathrm{LCP}[\mathrm{SA}^{-1}[i]] := h$
    $h := \max(0, h-1)$

- Zeit: $O(n)$

Suffix-Baum aus SA und LCP

$S =$ banana0



- naiv: $O(n^2)$
- mit Suffix-Array: in lexikographischer Reihenfolge

Suffix-Baum aus SA und LCP

- LCP-Werte helfen!
- Betrachte nur rechtesten Pfad!
- Finde tiefsten Knoten mit String-Tiefe $\leq \mathrm{LCP}[i] \rightsquigarrow$ Einfügepunkt!

	0	1	2	3	4	5	6
$\mathrm{SA} =$	6	5	3	1	0	4	2
$\mathrm{LCP} =$	0	0	1	3	0	0	2

- Zeit $O(n)$

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

Suche in Suffix Arrays

$l := 1; r \leftarrow := n+1$

while $l < r$ **do** // search left index

$q := \lfloor \frac{l+r}{2} \rfloor$

if $P >_{\text{lex}} T_{\text{SA}[q]\ldots\min\{\text{SA}[q]+m-1,n\}}$

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 $P =_{\text{lex}} T_{\text{SA}[q]\ldots\min\{\text{SA}[q]+m-1,n\}}$

then $l := q$ **else** $r := q-1$

return $[s,l]$

- Zeit $O(m \log n)$ (geht auch: $O(m+\log n)$)

Burrows-Wheeler-Transformation

$T = \text{CACAACCAC\$}$

1	2	3	4	5	6	7	8	9	10
C	A	C	A	A	C	C	A	C	$
A	C	A	A	C	C	A	C	$	C
C	A	A	C	C	A	C	$	C	A
A	A	C	C	A	C	$	C	A	C
A	C	C	A	C	$	C	A	C	A
C	C	A	C	$	C	A	C	A	A
C	A	C	$	C	A	C	A	A	C
A	C	$	C	A	C	A	A	C	C
C	$	C	A	C	A	A	C	C	A
$	C	A	C	A	A	C	C	A	C

$T^{(1)}$ $T^{(6)}$

$\Rightarrow$ sort columns lexicogr.

1	2	3	4	5	6	7	8	9	10	
$	A	A	A	A	C	C	C	C	C	$\rightarrow F$ (first)
C	A	C	C	C	$	A	A	A	C	
A	C	$	A	C	C	A	C	C	A	
C	C	C	A	A	A	C	$	A	C	
A	A	A	C	C	C	C	C	A	$	
A	C	C	C	$	A	A	A	C	C	
C	$	A	C	C	A	C	C	C	A	
C	C	A	A	A	C	$	A	A	C	
A	A	C	$	C	C	C	A	C	A	
C	C	C	C	A	A	A	C	$	A	$\rightarrow T^{\text{BWT}} = L$ (last)

$T^{(1)}$

Burrows-Wheeler-Transformation

$T = \text{CACAACCAC\$}$

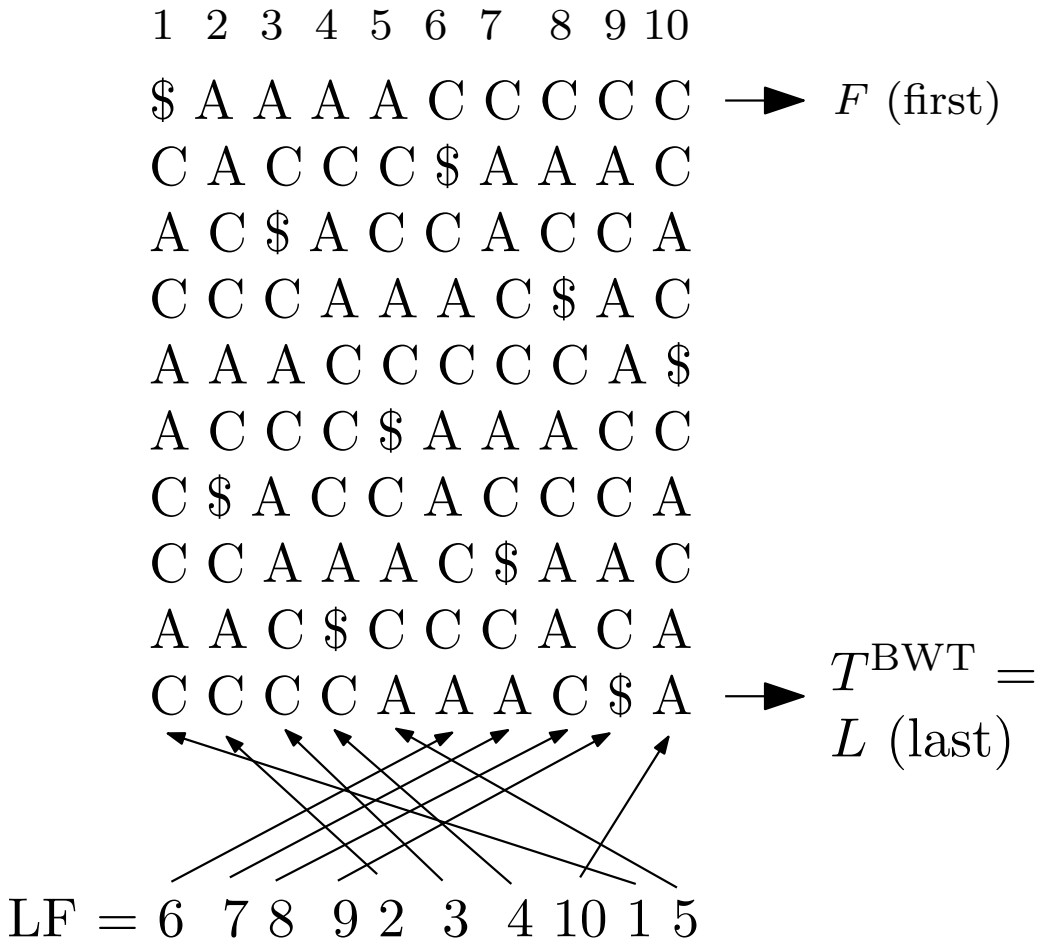
	1	2	3	4	5	6	7	8	9	10	
A=	10	4	8	2	5	9	3	7	1	6	
	$	A	A	A	A	C	C	C	C	C	→ F (first)
	C	A	C	C	C	$	A	A	A	C	
	A	C	$	A	C	C	A	C	C	A	
	C	C	C	A	A	A	C	$	A	C	
	A	A	A	C	C	C	C	C	A	$	
	A	C	C	C	$	A	A	A	C	C	
	C	$	A	C	C	A	C	C	C	A	
	C	C	A	A	A	C	$	A	A	C	
	A	A	C	$	C	C	C	A	C	A	
	C	C	C	C	A	A	A	C	$	A	→ $T^{\text{BWT}} =$ L (last)

- $L[i] = t_i^{\text{BWT}} = t_{\text{SA}[i]-1}$ $\Rightarrow$ BWT computable in $O(n)$ time

Rücktransformation

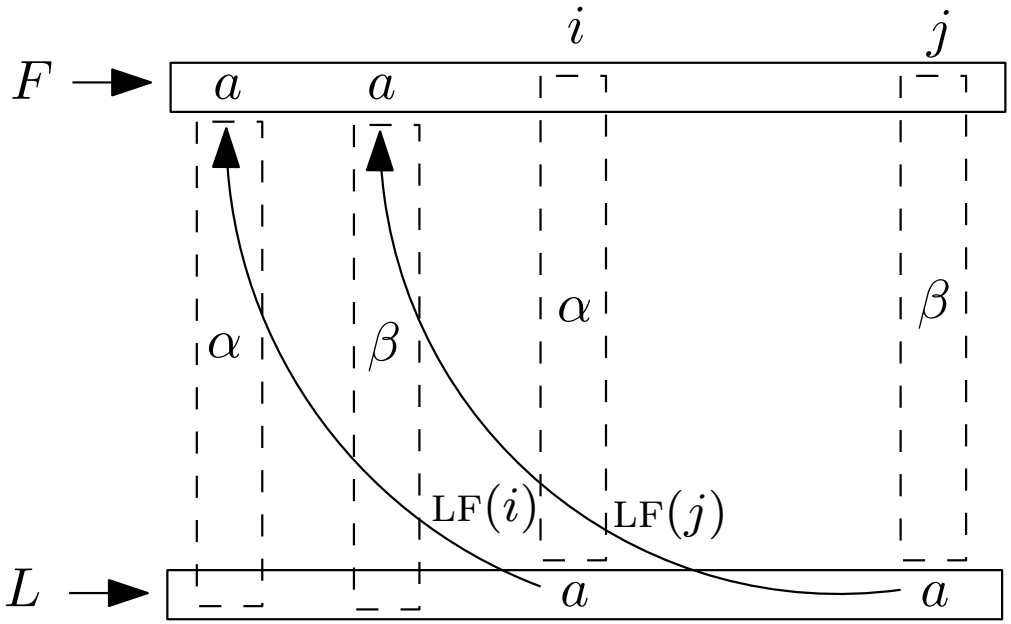
- $LF[i] =$ wo steht mein Vorgängerzeichen?

$T = \text{CACAACCAC\$}$



Rücktransformation

- Gleiche Zeichen haben die gleiche Ordnung in F und L:
  - falls $L[i] = L[j]$ und $i < j$, dann $LF[i] < LF[j]$)



Rücktransformation

- Gleiche Zeichen haben die gleiche Ordnung in F und L:
  - falls $L[i] = L[j]$ und $i < j$, dann $LF[i] < LF[j]$)
- Die LF-Funktion kann direkt aus der BWT L berechnet werden:
  - $C(a)$ = Gesamtzahl aller Zeichen kleiner als a
  - $Occ(i)$ = Anzahl der $L[i]$'s in $L[1,i]$

$\Rightarrow LF[i] = C(L[i]) + Occ(i)$

$$\overset{\$\ \ A\ \ C}{C = 0\ 1\ 5}$$

$$F = \$\ A\ A\ A\ A\ C\ C\ C\ C\ C$$

$$L = C\ C\ C\ C\ A\ A\ A\ C\ \$\ A$$

$$\text{OCC} = 1\ 2\ 3\ 4\ 1\ 2\ 3\ 5\ 1\ 4$$

$$\begin{aligned}
T_n &= \boxed{\$}\ , k = 1 \\
L[1] = C \Rightarrow T_{n-1} &= \boxed{C}\ , k = \text{LF}(1) = 6 \\
L[6] = A \Rightarrow T_{n-2} &= \boxed{A}\ , k = \text{LF}(6) = 3 \\
L[3] = C \Rightarrow T_{n-3} &= \boxed{C}\ , k = \text{LF}(3) = 8 \\
L[8] = C \Rightarrow T_{n-4} &= \boxed{C}\ , k = \text{LF}(8) = 10 \\
L[10] &\text{ etc.}
\end{aligned}$$

T reversed

BWT: Kompression

- Erster Schritt: erzeuge kleine Zahlen (Move-to-Front)

$$T^{\mathrm{BWT}} = \mathrm{C\ C\ C\ C\ A\ A\ A\ C\ \$\ A}$$

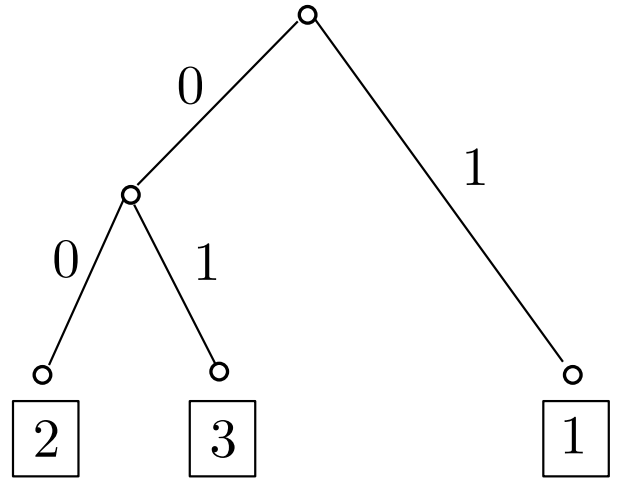
i	Y_{old}	T_i^{BWT}	$R[i]$	Y_{new}
1	\$AC	C	3	C\$A
2	C\$A	C	1	C\$A
3	C\$A	C	1	C\$A
4	C\$A	C	1	C\$A
5	C\$A	A	3	AC\$
6	AC\$	A	1	AC\$
7	AC\$	A	1	AC\$
8	AC\$	C	2	CA\$
9	CA\$	\$	3	\$CA
10	\$CA	A	3	A\$C

BWT: Kompression

- Zweiter Schritt: Variable-Längen-Kodierung
- Huffman

character	frequency
1	5
2	1
3	4

$\Rightarrow$



$R[i]$
3
1
1
1
3
1
1
2
3
3

$\Rightarrow$

Huffman-Kodierung

- Präfix-freie Codes variabler Länge
- können greedy konstruiert werden
  - erzeuge binären Baum bottom-up
  - nimm die seltensten 2 Zeichen(gruppen)
  - erzeuge neuen Knoten, der beide Zeichen(gruppen) repräsentiert
  - Häufkeit neu = Summe der beiden Häufigkeiten
- Path-Labels sind Zeichenkodierungen (li:0; re: 1)
- Operationen: extract-min & increase-key $\rightsquigarrow O(n \log n)$

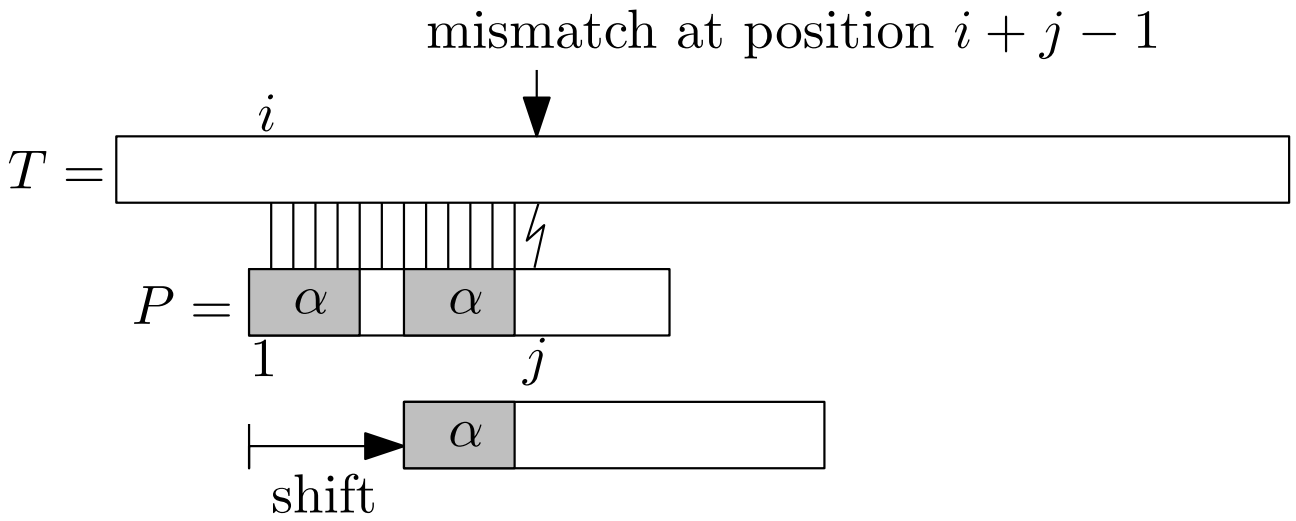
Naives Pattern Matching

- 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 $i \leq n - m + 1$
  while $j \leq m$ and $t_{i+j-1} = p_j$ **do** j++ // compare characters
  if $j > m$ **then** output "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



- $\text{border}[j] =$ längstes echtes Präfix von $P_{1\ldots j-1}$, das auch (echtes) Suffix von $P_{1\ldots j-1}$ ist

Knuth-Morris-Pratt (1977)

$i := 1$ // index in T

$j := 1$ // index in P

while $i \leq n - m + 1$

  while $j \leq m$ and $t_{i+j-1} = p_j$ **do** j++ // compare characters

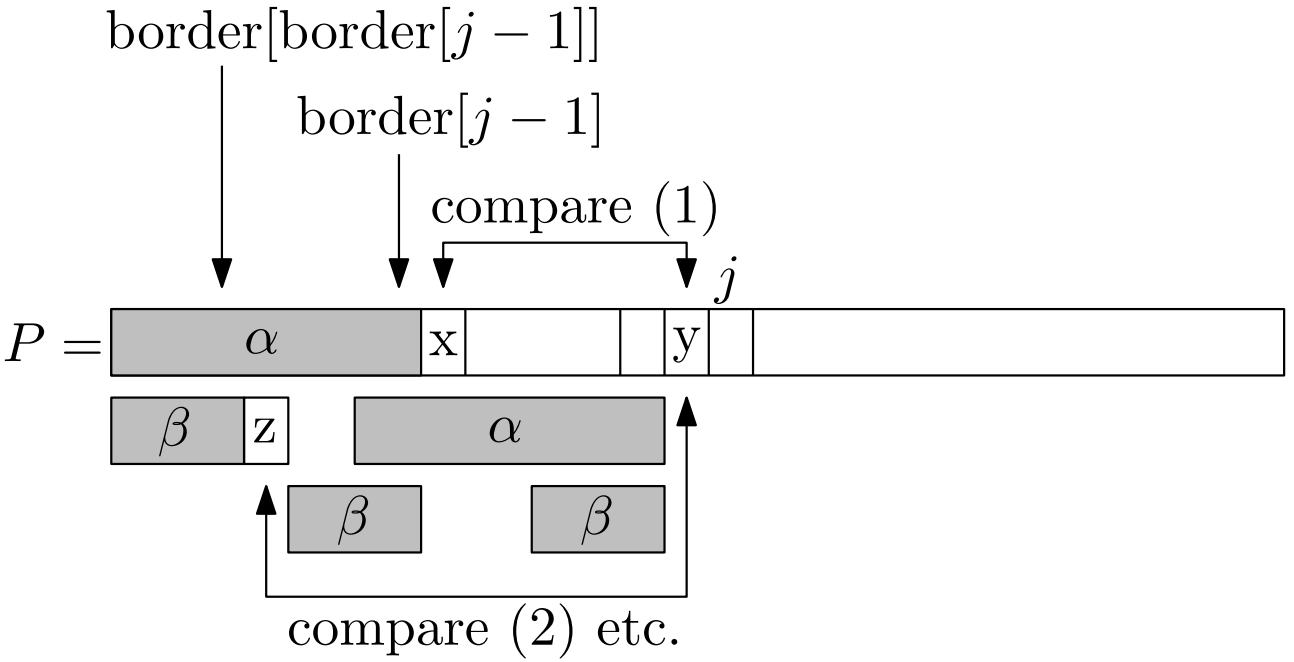
  if $j > m$ **then** output "P occurs at position i in T"

  $i := i + j - \text{border}[j] - 1$ // advance in T

  $j := \max\{1, border[j] + 1\}$ // skip first $\text{border}[j]$ 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,...,m
    while i ≥ 1 and p_{i+1} ≠ p_j do i = border[i]
    i++
    border[j] := i
```

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 Σ

String $S = \langle s_1, \ldots, s_n \rangle \in \Sigma^*$

Textkompressionsalgorithms $f : S \to f(S)$ mit $|f(S)|$ (z.B. gemessen in bits) möglichst klein.

Theorie Verlustfreier Textkompression

Informationstheorie. Zum Beispiel

Entropie: $H(S) = -\sum_i \log(p(s_i))$ wobei $p(x) = |\{s_i : s_i = x\}|/n$ die relative Häufigkeit von s_i ist.

untere Schranke für # bits pro Zeichen falls Text einer Zufallsquelle entspränge.

$\rightsquigarrow$ Huffman-Coding ist annähernd optimal! (Entropiecodierung) ????

Schon eher:

Entropie höherer Ordnung betrachte Teilstrings fester Länge

“Ultimativ”: Kolmogorov Komplexität. Leider nicht berechenbar.

Wörterbuchbasierte Textkompression

Grundidee: wähle $\Sigma' \subseteq \Sigma^*$ und ersetze $S \in \Sigma^*$ durch $S' = \langle s'_1, \ldots, s'_k \rangle \in \Sigma'^*$, 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 Σ' sogar kEntropie(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,...`

$\rightarrow$ 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$(\langle s_1, \ldots, s_n \rangle, \Sigma)$

$D := \Sigma$ // 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$(\langle c_1, \ldots, c_k \rangle)$

$D := \Sigma$

$p := \varepsilon$ // current string

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

$D := D \cup p \cdot \text{decode}(c_i)[1]$

$p := c_i$

output decode(p)

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. $|D| \leq 4096 \rightsquigarrow$ 12bit 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:

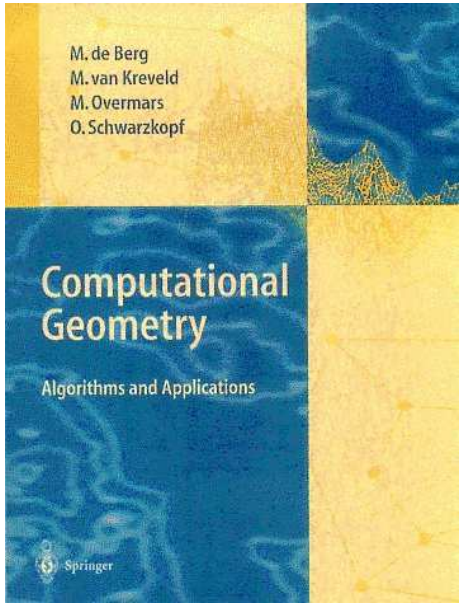
  (universelles) hashing

10 Geometrische Algorithmen

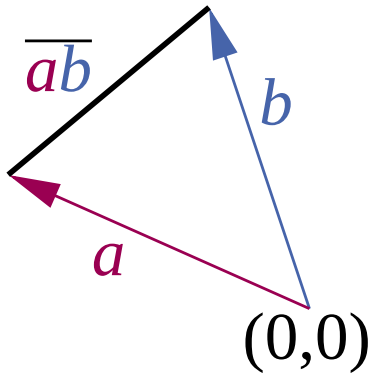
- Womit beschäftigen sich geom. Algorithmen?
- Schnitt von Strecken: Bentley-Ottmann-Algorithmus
- Konvexe Hüllen
- Kleinste einschließende Kugel
- Range Search

Quelle:

[Computational Geometry – Algorithms and Applications
de Berg, van Kreveld, Overmars, Schwartzkopf
Springer, 1997]



Elementare Geometrische Objekte



Punkte: $x \in \mathbb{R}^d$

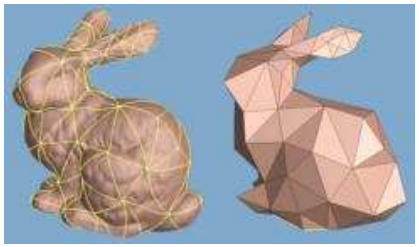
Strecken: $\overline{ab} := \{\alpha a + (1-\alpha)b : \alpha \in [0,1]\}$

uvam: Halbräume, Ebenen, Kurven,...

Dimension d:

1: Oft trivial. Gilt i. allg. nicht als geometrisches Problem

2: Geogr. Informationssysteme (GIS), Bildverarbeitung,...



3: Computergrafik, Simulationen,...

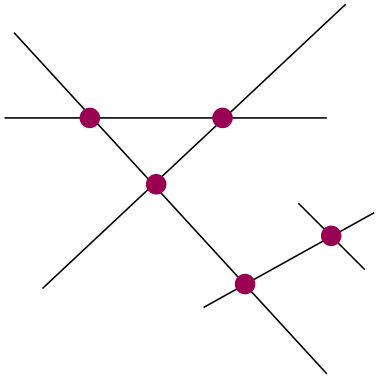
≥ 4: Optimierung, Datenbanken, maschinelles Lernen,...

curse of dimensionality!

n: Anzahl vorliegender Objekte

Typische Fragestellungen

- Schnittpunkte zwischen n Strecken



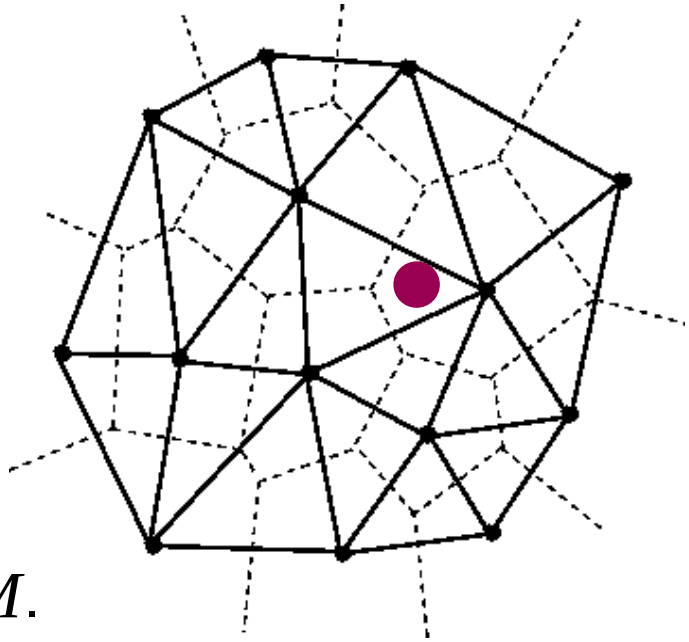
Typische Fragestellungen

- Schnittpunkte zwischen n Strecken
- Konvexe Hülle
- Triangulation von Punktmengen

  (2D, verallgemeinerbar)

  z.B. Delaunaytriangulierung:

  Kein Dreieck enthält weiteren Punkt



- Voronoi-Diagramme: Sei $x \in M \subseteq \mathbb{R}^d$.

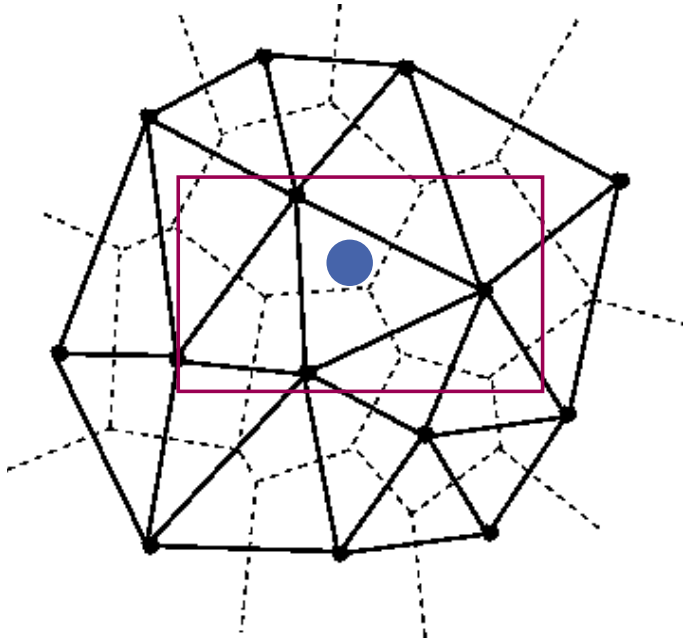
  $\forall y \in \mathbb{R}^d$ bestimme nächstes Element aus M.

  (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
- ...



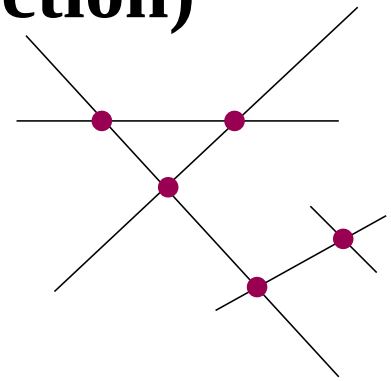
Mehr Fragestellungen

- Sichtbarkeitsberechnungen
- Lineare Programmierung
- Geometrische Versionen von Optimierungsproblemen
  - Kürzeste Wege, z.B.

    energieffiziente Kommunikation in Radionetzwerken
  - minimale Spannbäume

    reduzierbar auf Delaunay-Triangulierung + Graphalgorithmus
  - Matchings
  - Handlungsreisendenproblem
  - ...
- ...

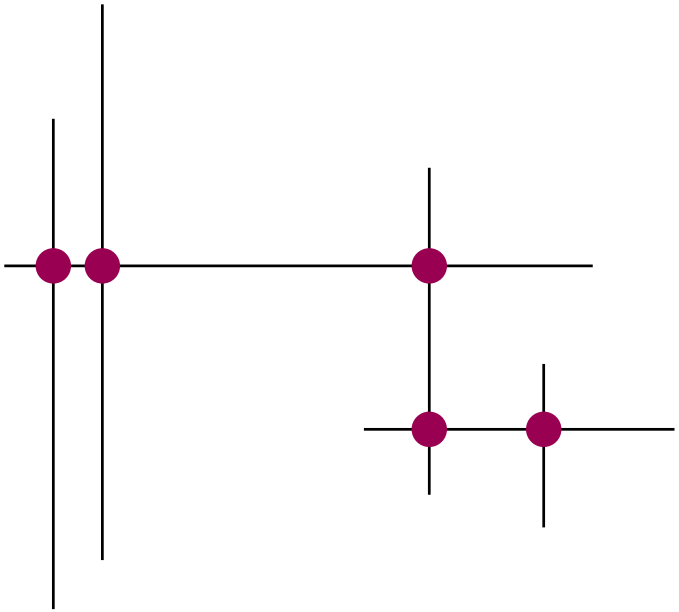
10.1 Streckenschnitt (line segment intersection)



Gegeben: $S = \{s_1, \ldots, s_n\}$, n Strecken

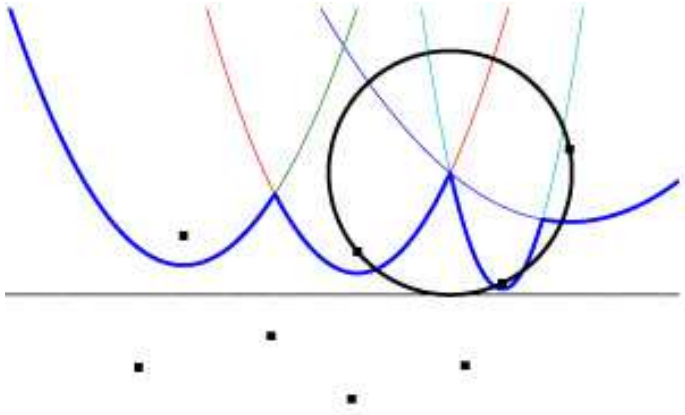
Gesucht: Schnittpunkte $\bigcup_{s,t \in S} s \cap t$

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 Voronoi-diagrammen)



Streckenschnitt: Naiver Algorithmus

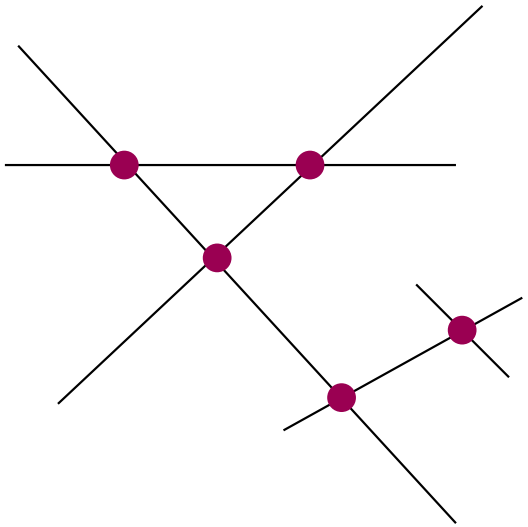
foreach $\{s,t\} \subseteq S$ **do**

  if $s \cap t \neq \emptyset$ **then**

    output $\{s,t\}$

Problem: Laufzeit $\Theta\left(n^2\right)$.

Zu langsam für große Datenmengen

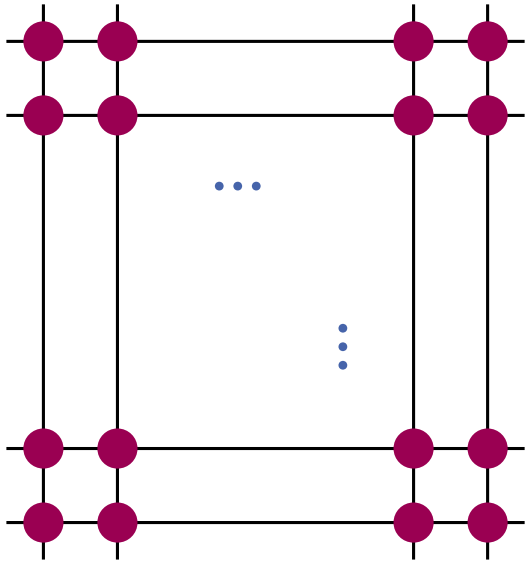


Streckenschnitt: Untere Schranke

$\Omega(n+k)$ 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 $k = \mathrm{O}(n)$.

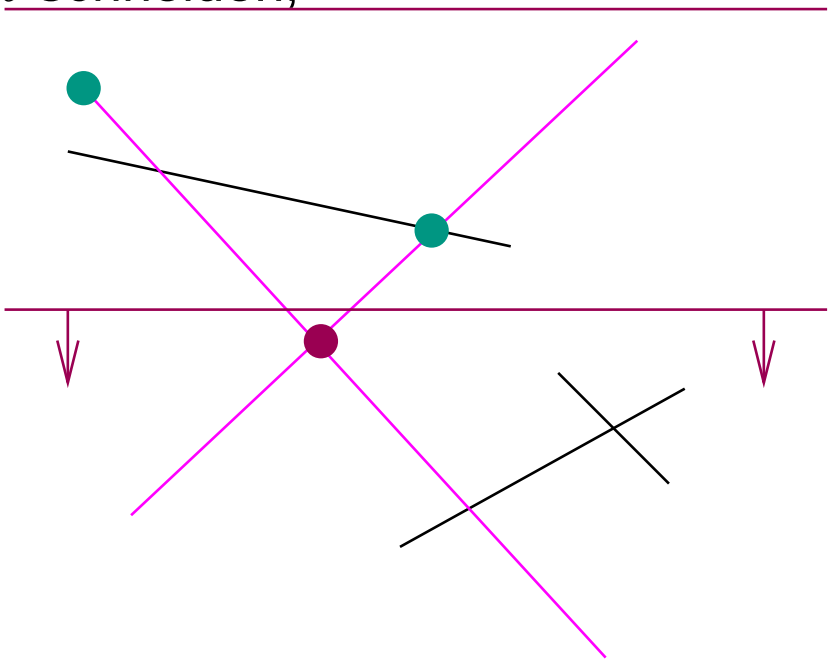


Idee: Plane-Sweep-Algorithmen

(Waagereche) Sweep-Line ℓ läuft von oben nach unten.

Invariante: Schnittpunkte oberhalb von ℓ wurden korrekt ausgegeben

Beobachtung: Nur Segmente, die sich mit ℓ schneiden, müssen neu ausgegeben werden.



Plane-Sweep für orth. Streckenschnitt

$T = \langle\rangle$: SortedSequence **of** Segment

invariant T stores the vertical segments intersecting ℓ

Q:= sort($\langle (y,s) : \exists$hor. seg. s at y or $\exists$vert. seg. s 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.insert(s)

    else if s is a vertical segment and ends at y **then** T.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 $\{s,t\}$

        handle horizontal segments on ℓ// interval intersection problem

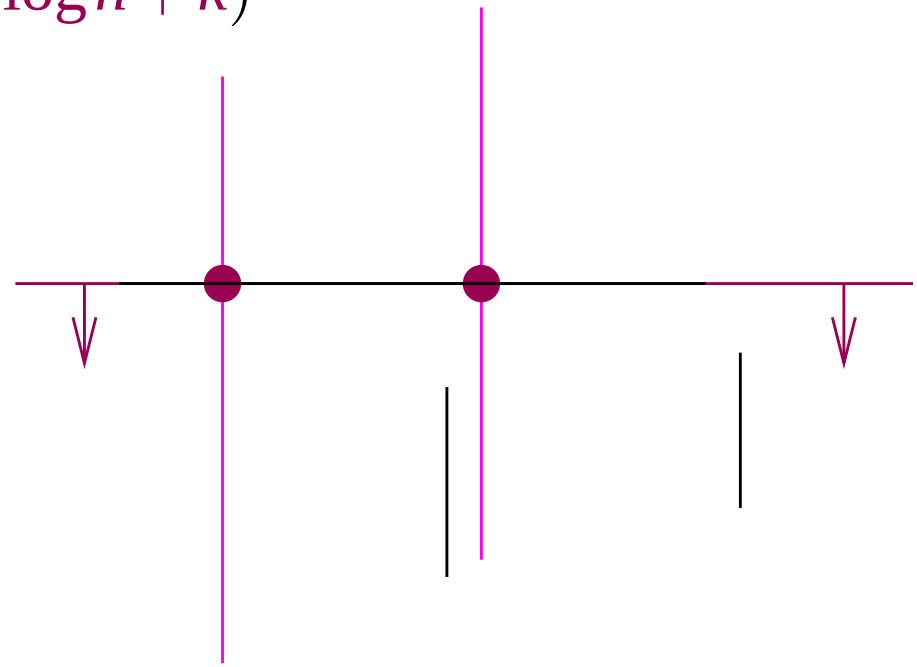
Analyse orth. Streckenschnitt

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: $O(n\log n + \sum_s k_s) = O(n\log n + k)$



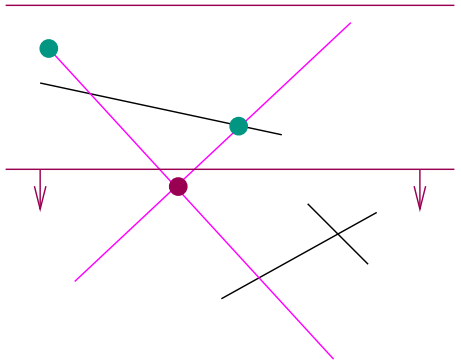
Verallgemeinerung – aber erstmal “nicht ganz”

Annahme zunächst:

Allgemeine Lage, d.h. hier

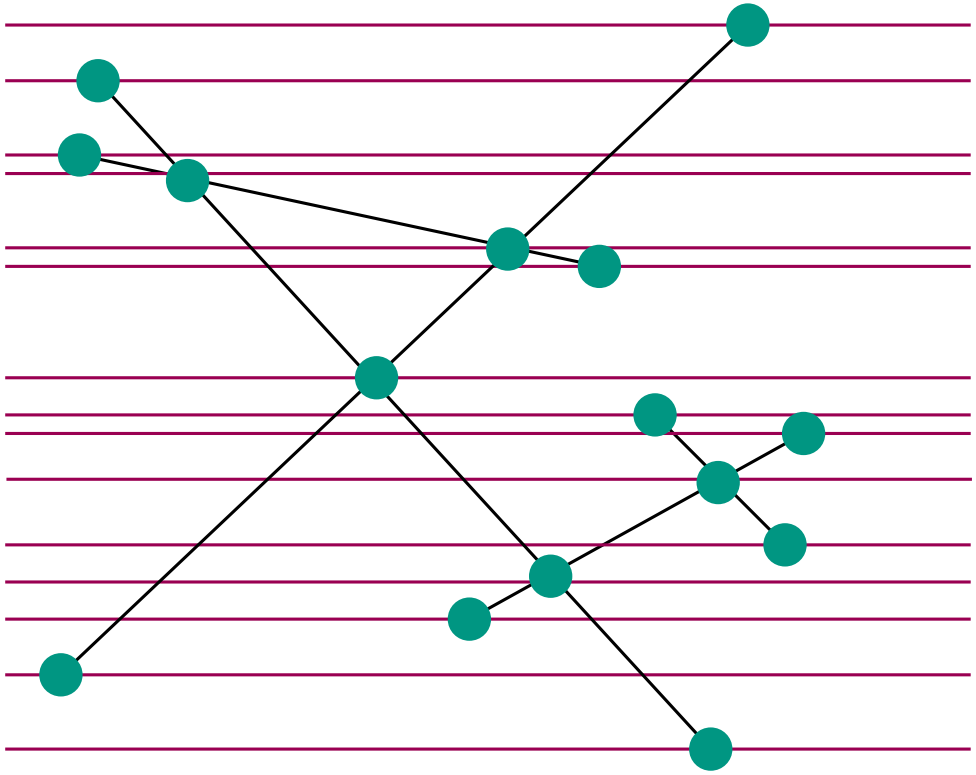
- Keine horizontalen Strecken
- Keine Überlappungen
- Schnittpunkte jeweils im Inneren von genau zwei Strecken

Beobachtung: kleine zuf. Perturbationen produzieren allg. Lage.



Verallgemeinerung – Grundidee

- Plane-Sweep mit Sweep-Line ℓ
- Status T := nach x geordnete Folge der ℓ schneidenden Strecken
- Ereignis:= Statusänderung
  - Startpunkte
  - Endpunkte
  - Schnittpunkte
- Schnitttest nur für Segmente, die an einem Ereignispunkt in T benachbart sind.



Verallgemeinerung – Korrektheit

Lemma:

$s \cap t = \{(x,y)\} \longrightarrow \exists$ Ereignis : s,t werden Nachbarn auf ℓ

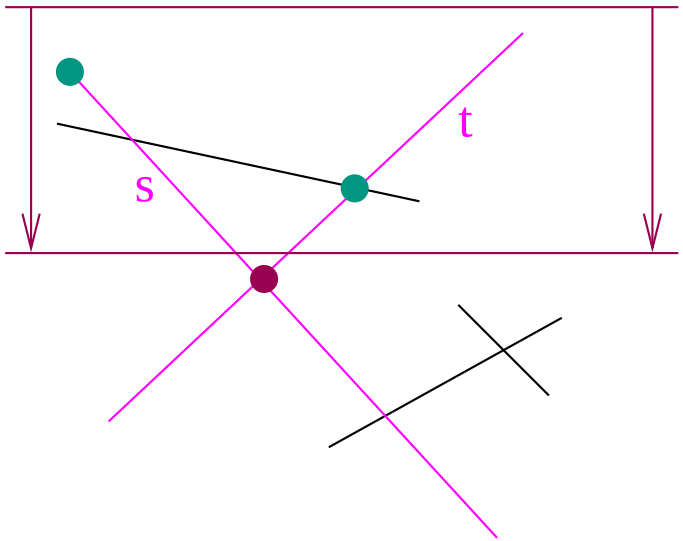
Beweis:

Anfangs : $T = \langle\rangle \longrightarrow s,t$ sind nicht in ℓ benachbart.

@$y+\varepsilon$: s,t sind in ℓ benachbart.

$\longrightarrow$

$\exists$ Ereignis bei dem s und t Nachbarn werden.



Verallgemeinerung – Implementierung

$T = \langle\rangle$: SortedSequence **of** Segment

invariant T stores the relative order of the segments intersecting ℓ

Q : MaxPriorityQueue

$Q := Q \cup \left\{ (y, \text{start}, s) : s = \overline{(x,y)(x',y')} \in S, y \geq y' \right\}$ // $O(n \log n)$

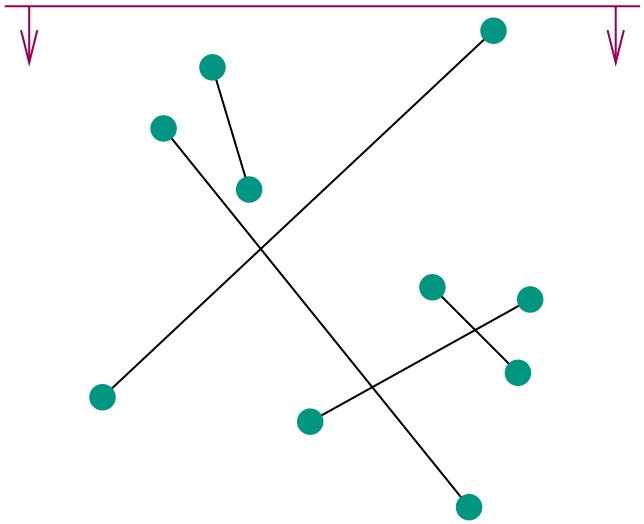
$Q := Q \cup \left\{ (y, \text{finish}, s) : s = \overline{(x,y)(x',y')} \in S, y \leq y' \right\}$ // $O(n \log n)$

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.insert(s) // $O(\log n)$

  prev:= pred(h) // $O(1)$

  next:= succ(h) // $O(1)$

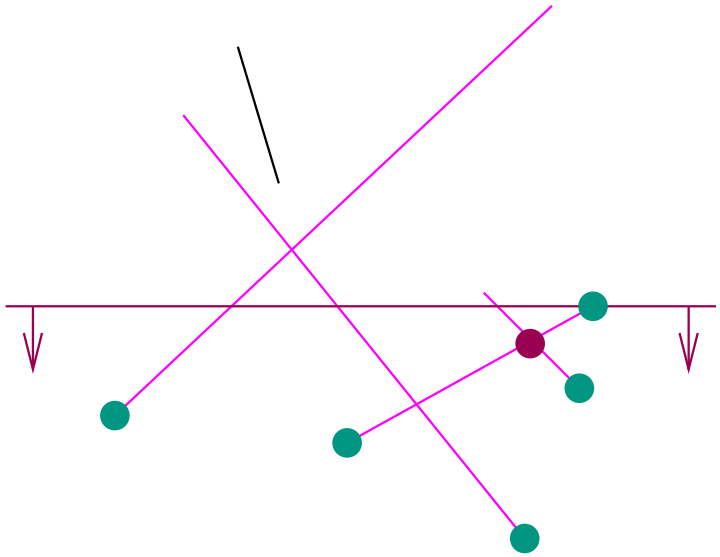
  findNewEvent(prev, h)

  findNewEvent(h, next)

Procedure findNewEvent(s,t) // $O(1+\log n)$

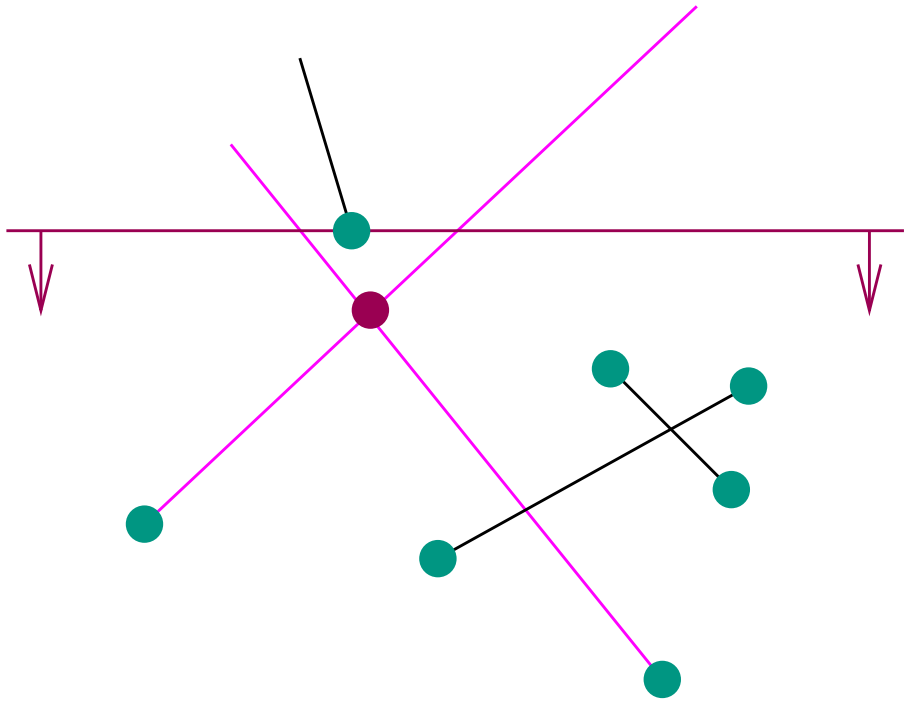
  if $*s$ and $*t$ intersect at $y' < y$ **then**

    Q.insert$((y', \text{intersection}, (s,t)))$



handleEvent$(y, \text{finish}, s, T, Q)$ // $n\times$

- h:= T.locate(s) // $O(\log n)$
- prev:= pred(h) // $O(1)$
- next:= succ(h) // $O(1)$
- T.remove(s) // $O(\log n)$
- findNewEvent(prev, next)



handleEvent$(y,$ intersection$,(a,b),T,Q)$ // $k\times$

output $(*s \cap *t)$ // $O(1)$

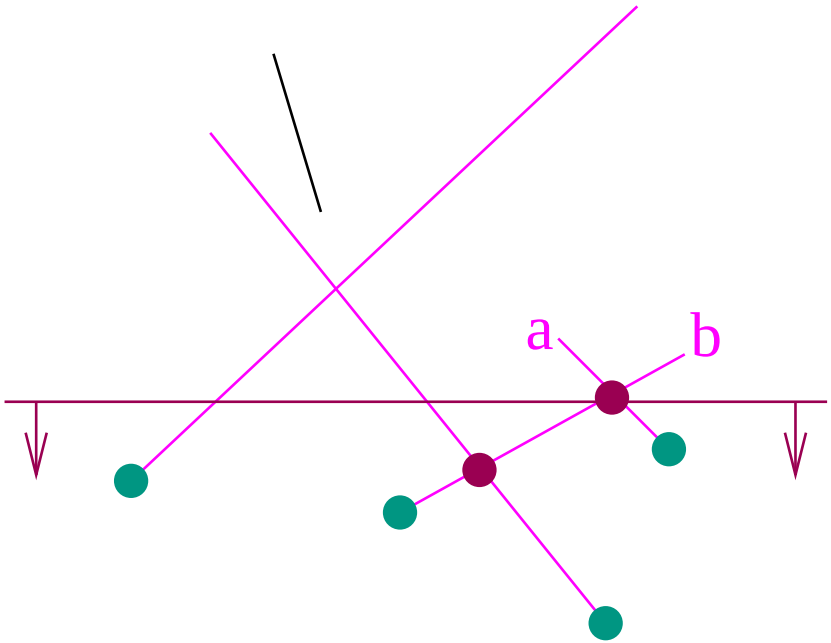
T.swap(a,b) // $O(\log n)$

prev:= pred(b) // $O(1)$

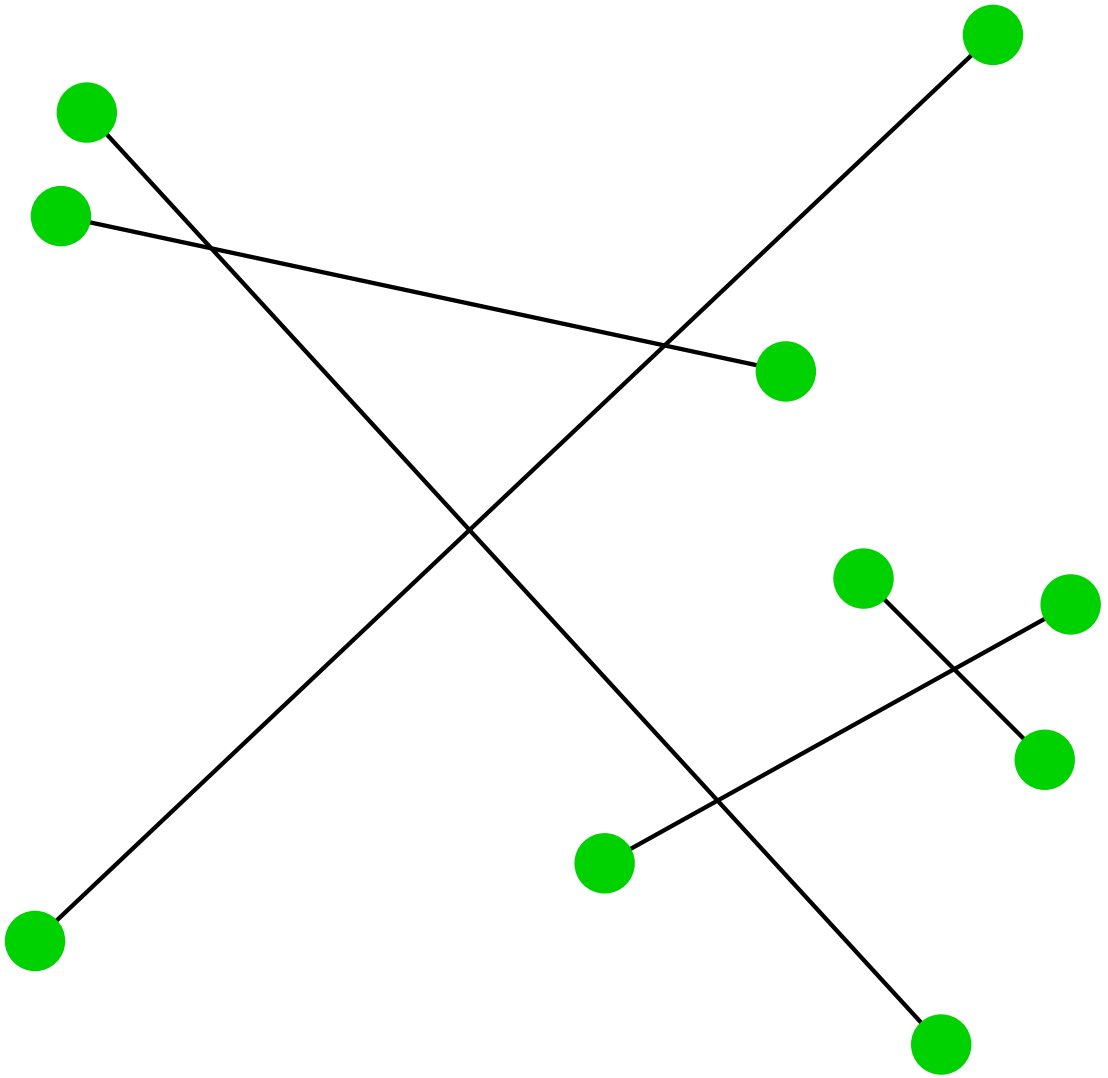
next:= succ(a) // $O(1)$

findNewEvent(prev, b)

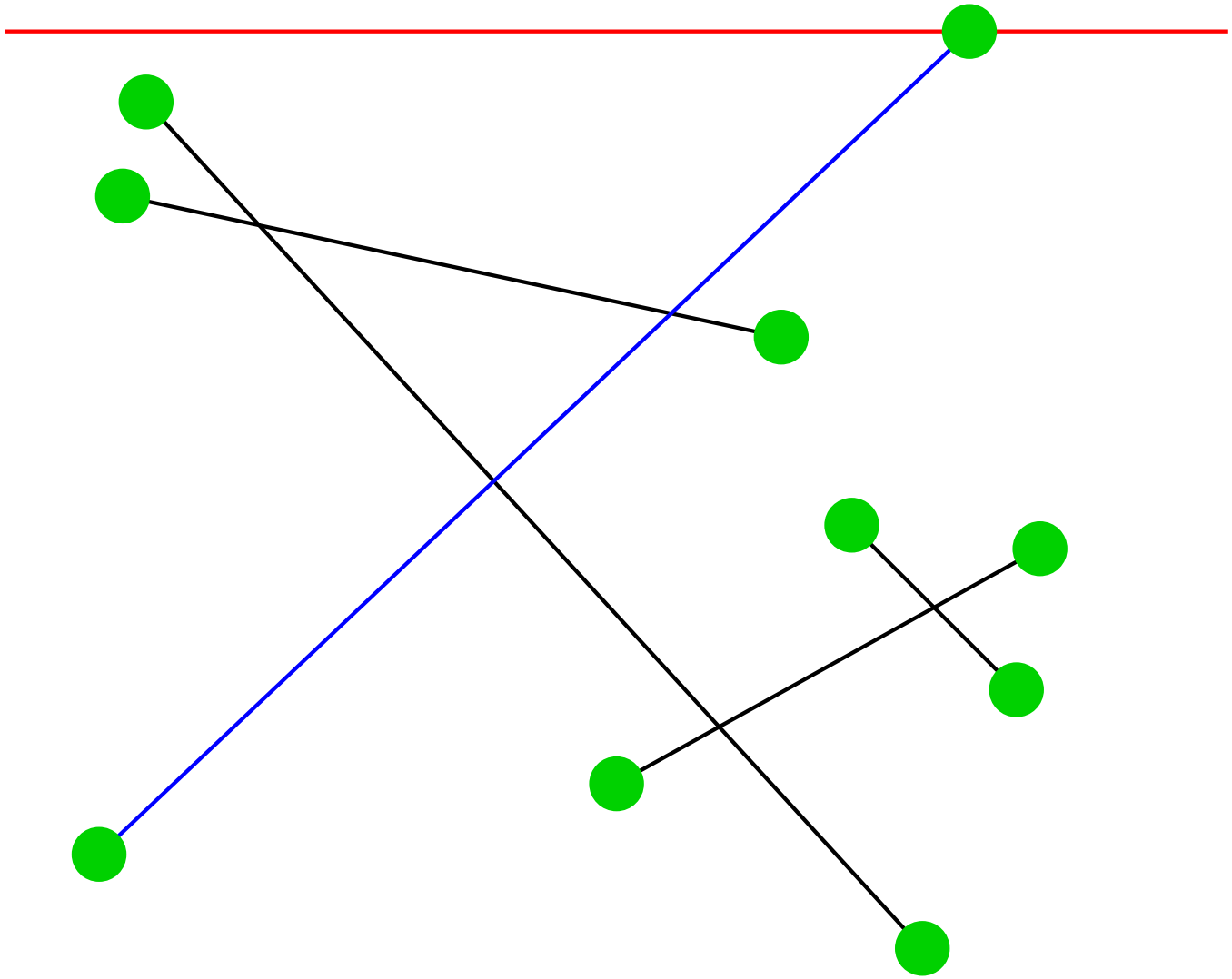
findNewEvent(a, next)



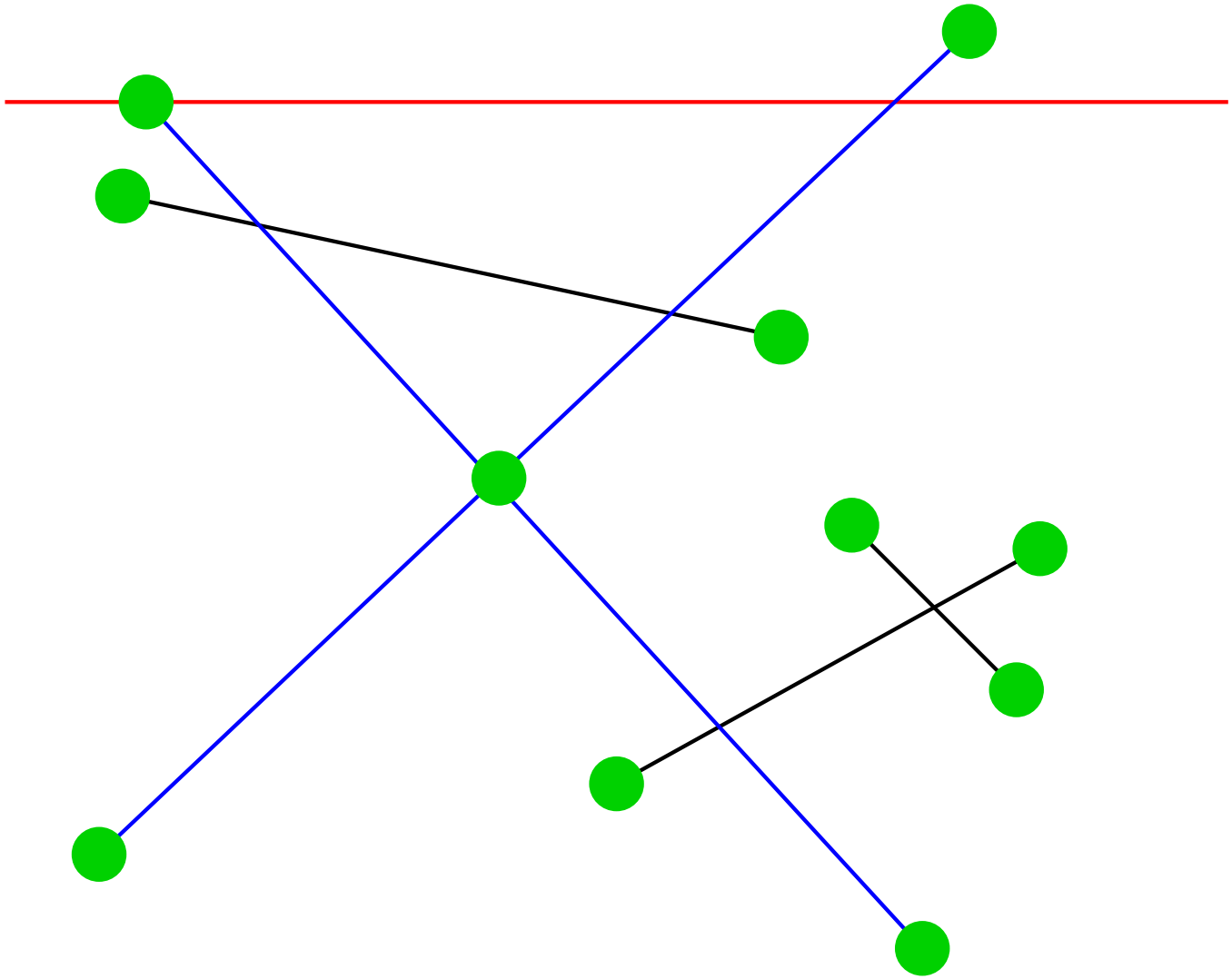
Verallgemeinerung – Beispiel



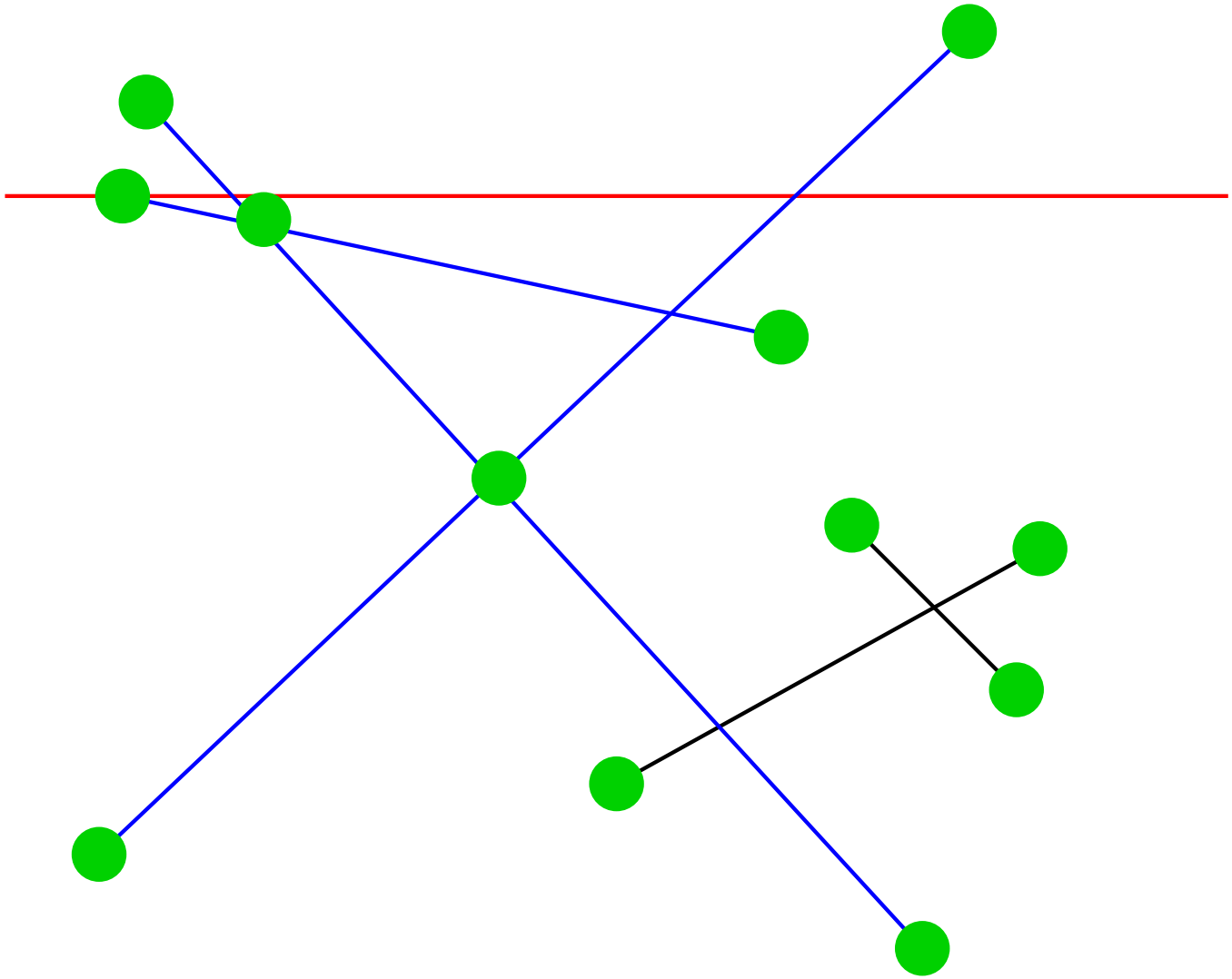
Verallgemeinerung – Beispiel



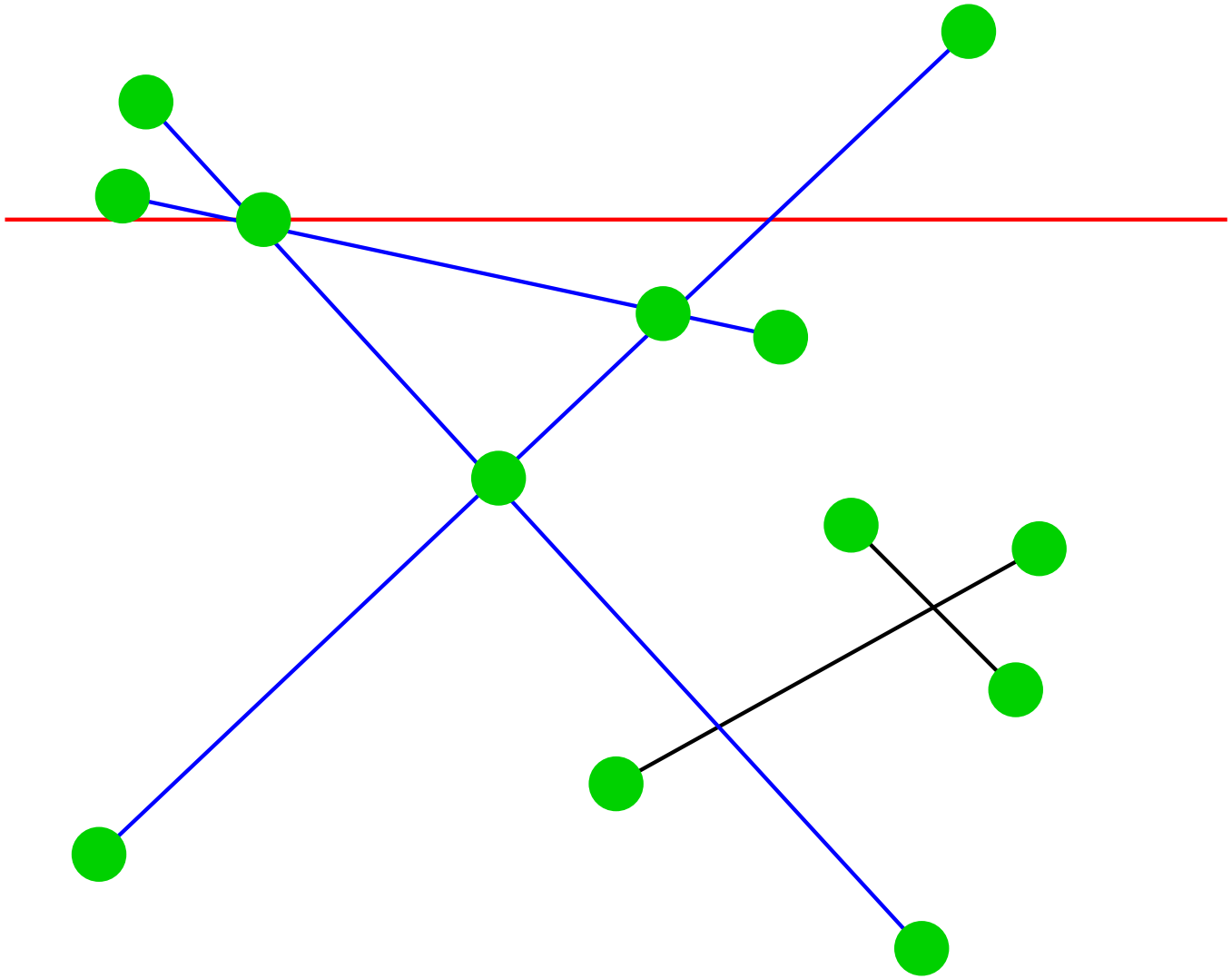
Verallgemeinerung – Beispiel



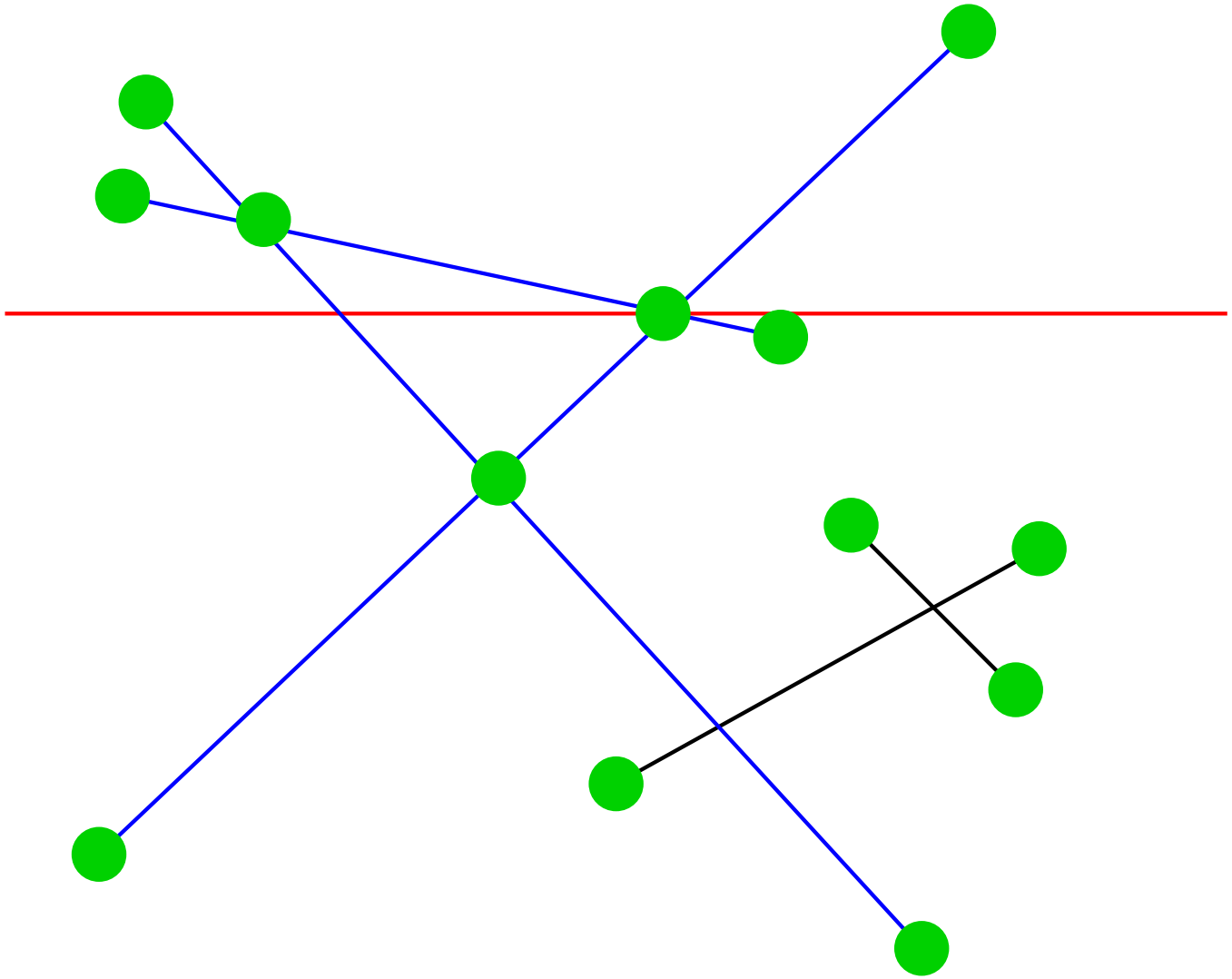
Verallgemeinerung – Beispiel



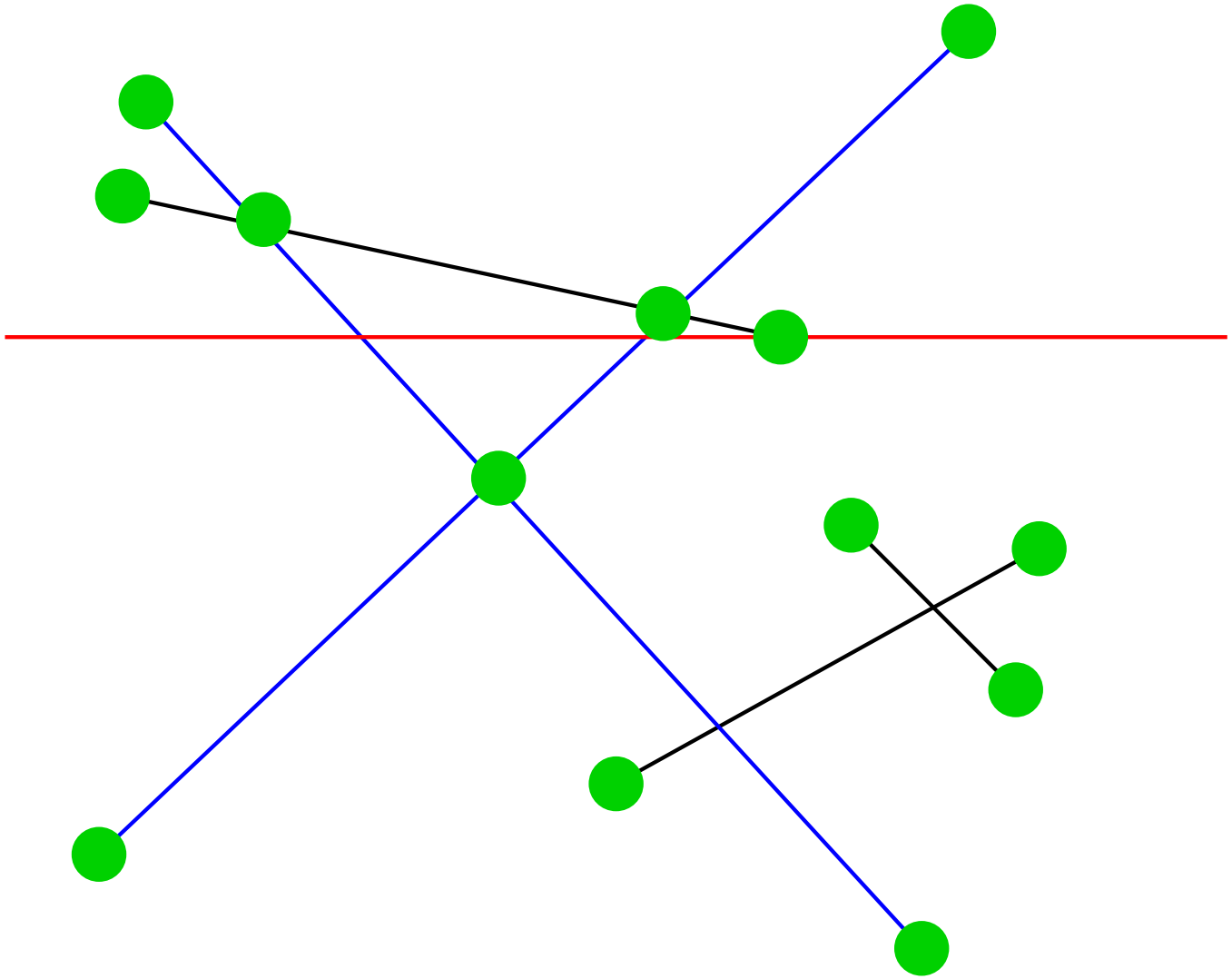
Verallgemeinerung – Beispiel



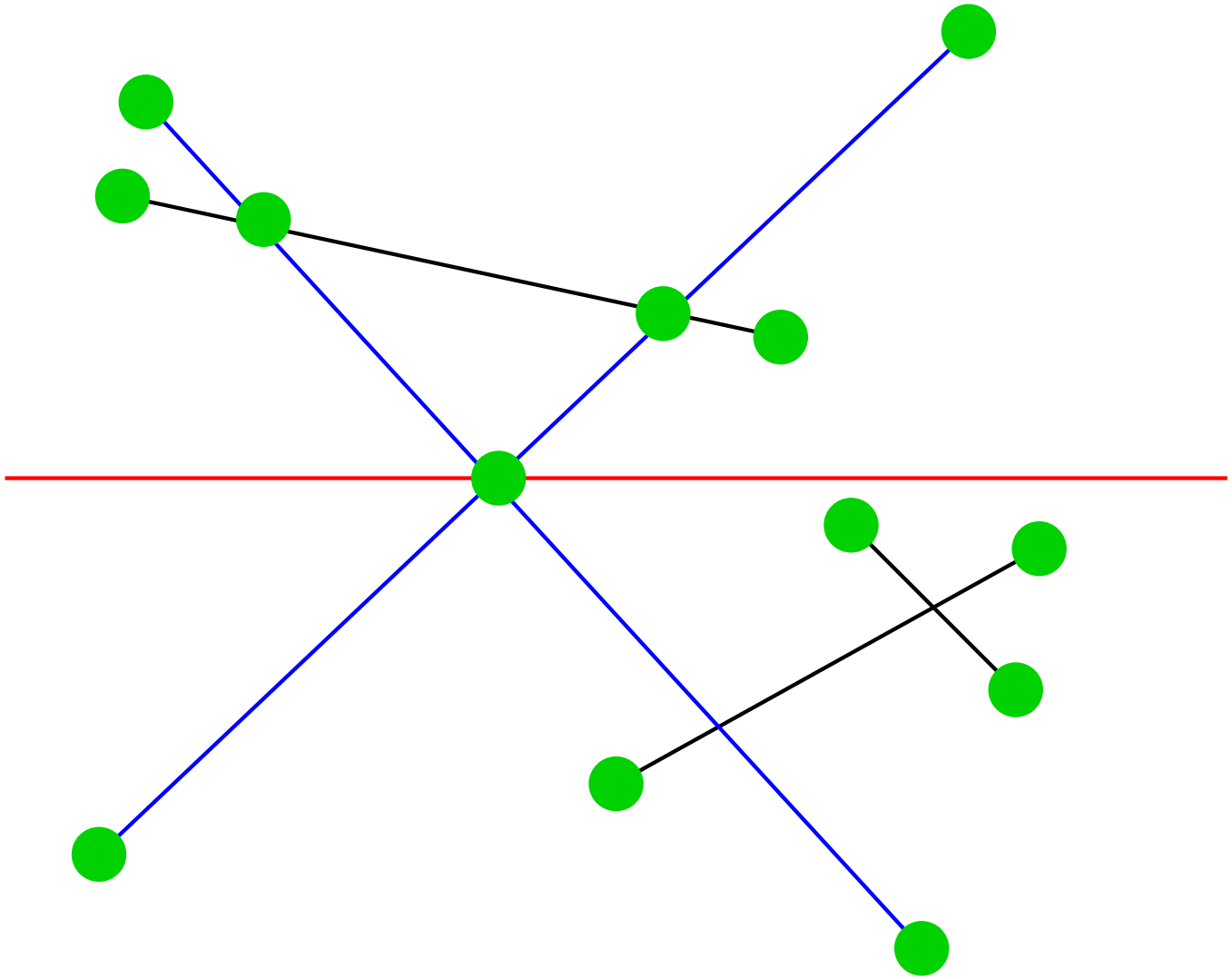
Verallgemeinerung – Beispiel



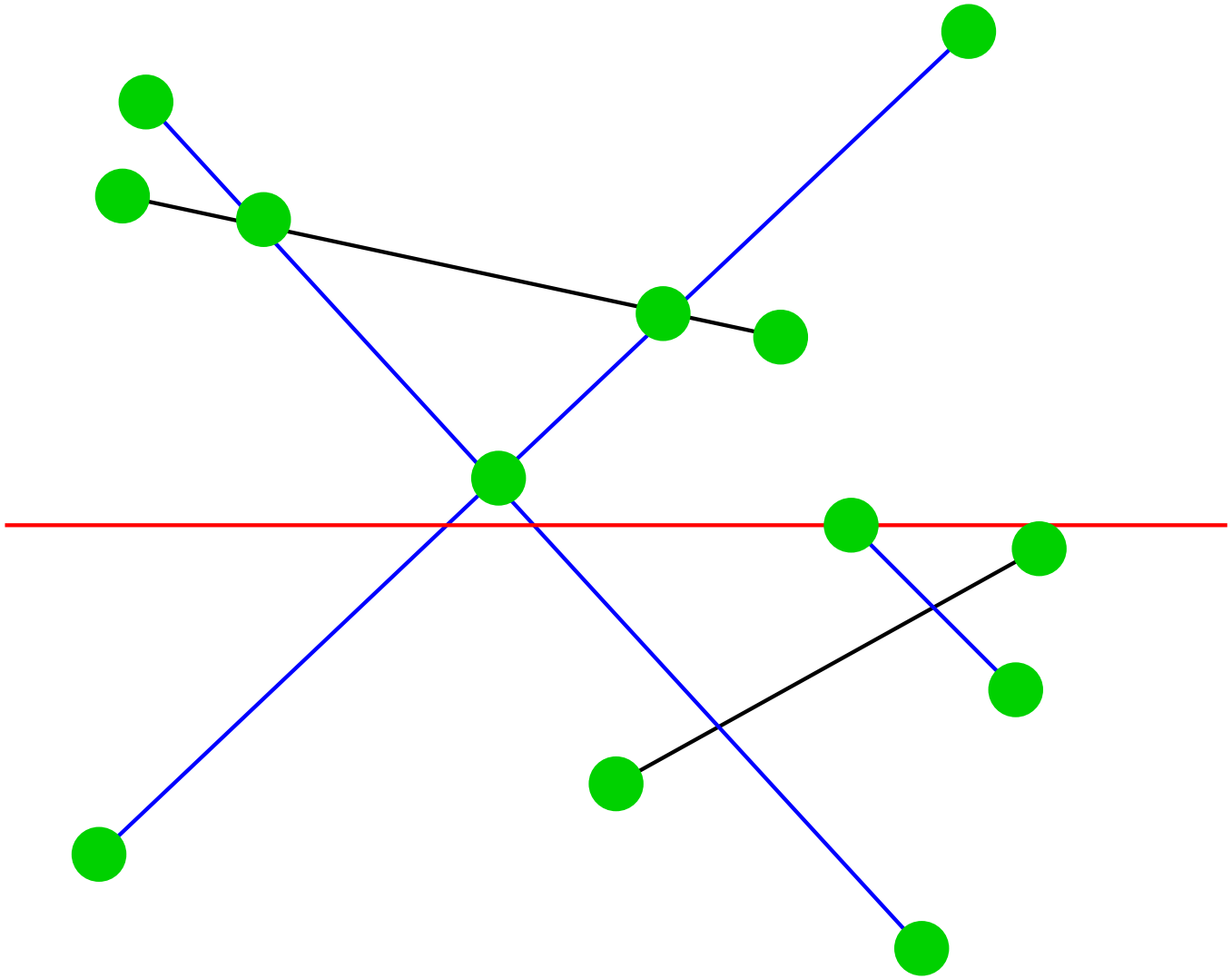
Verallgemeinerung – Beispiel



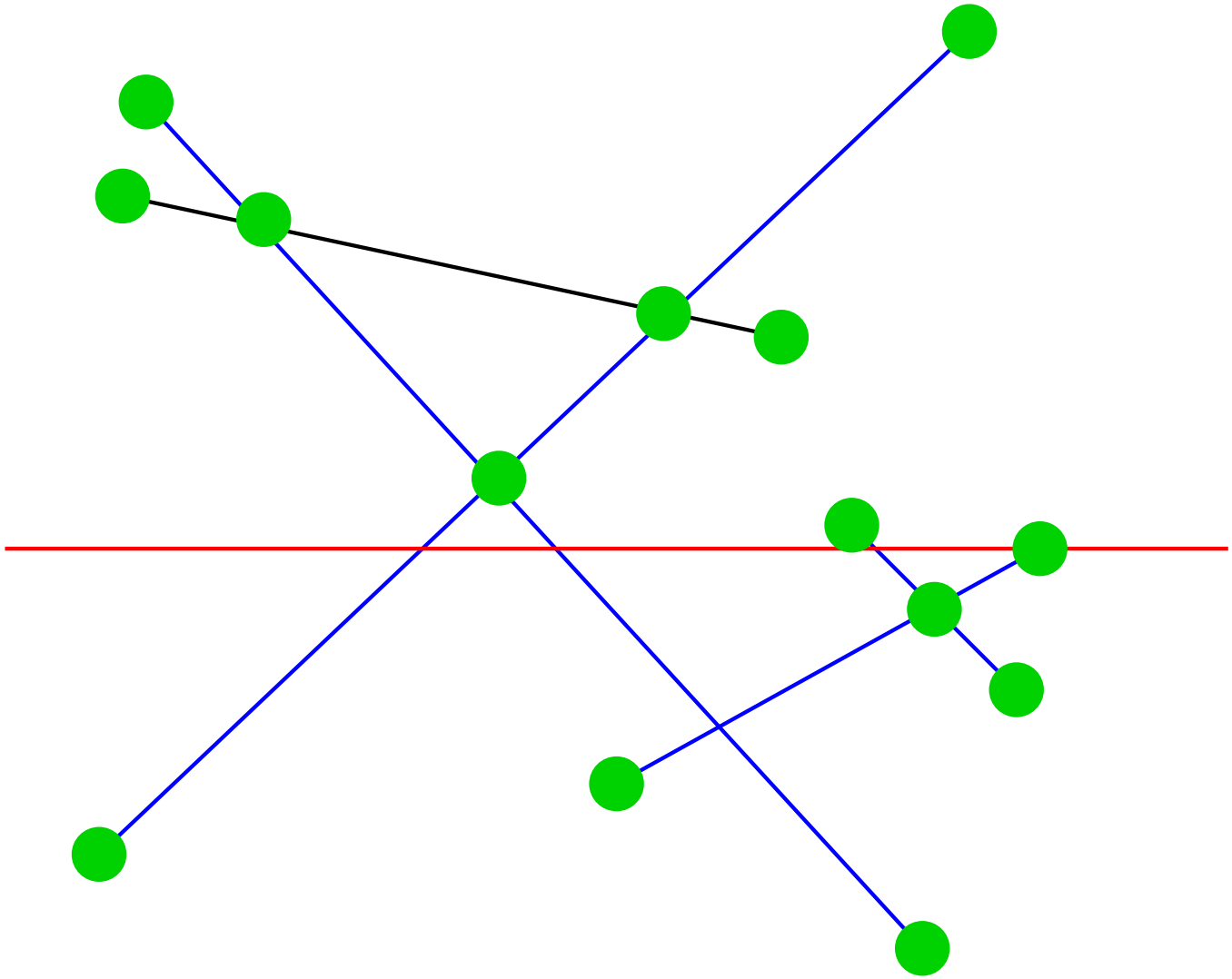
Verallgemeinerung – Beispiel



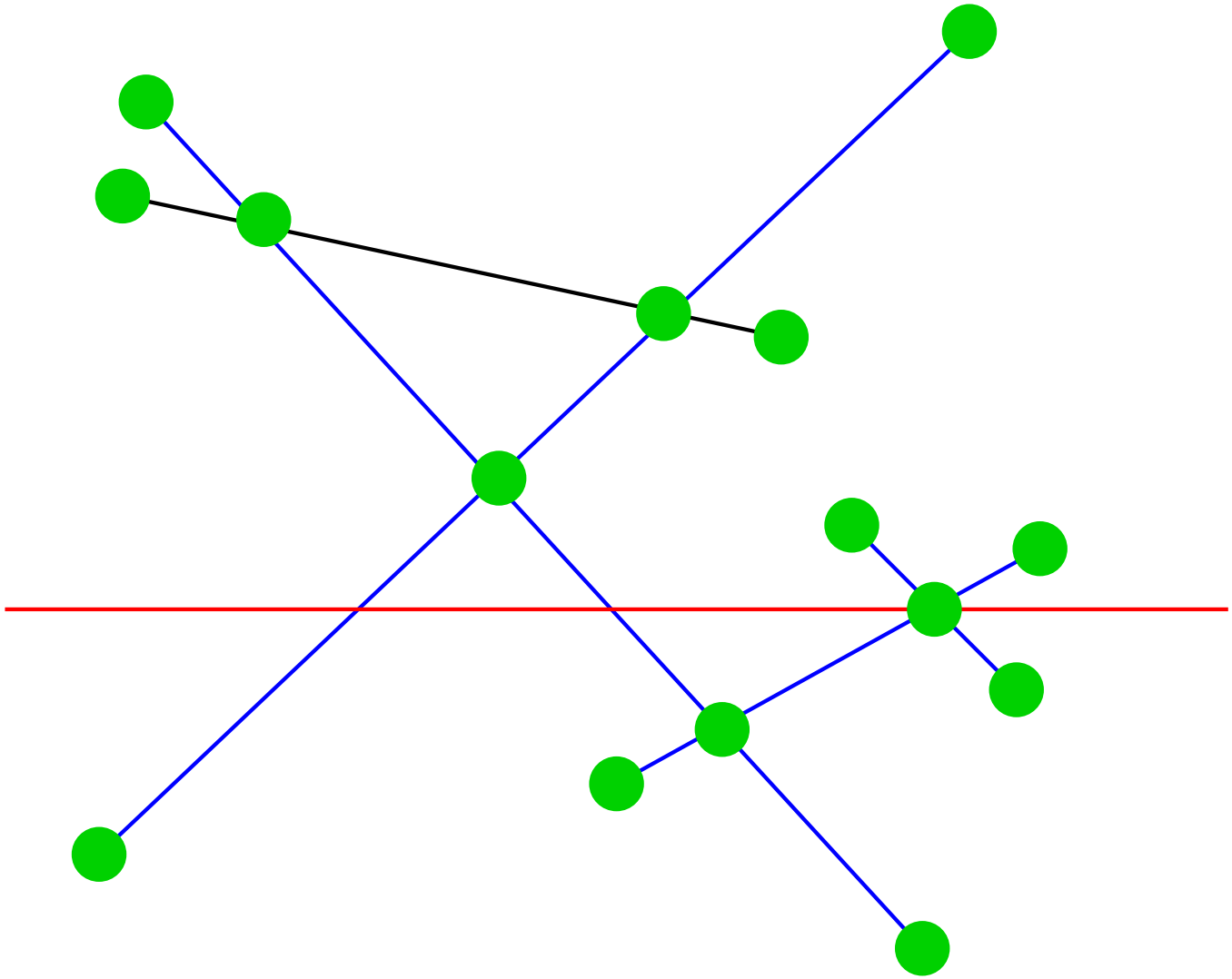
Verallgemeinerung – Beispiel



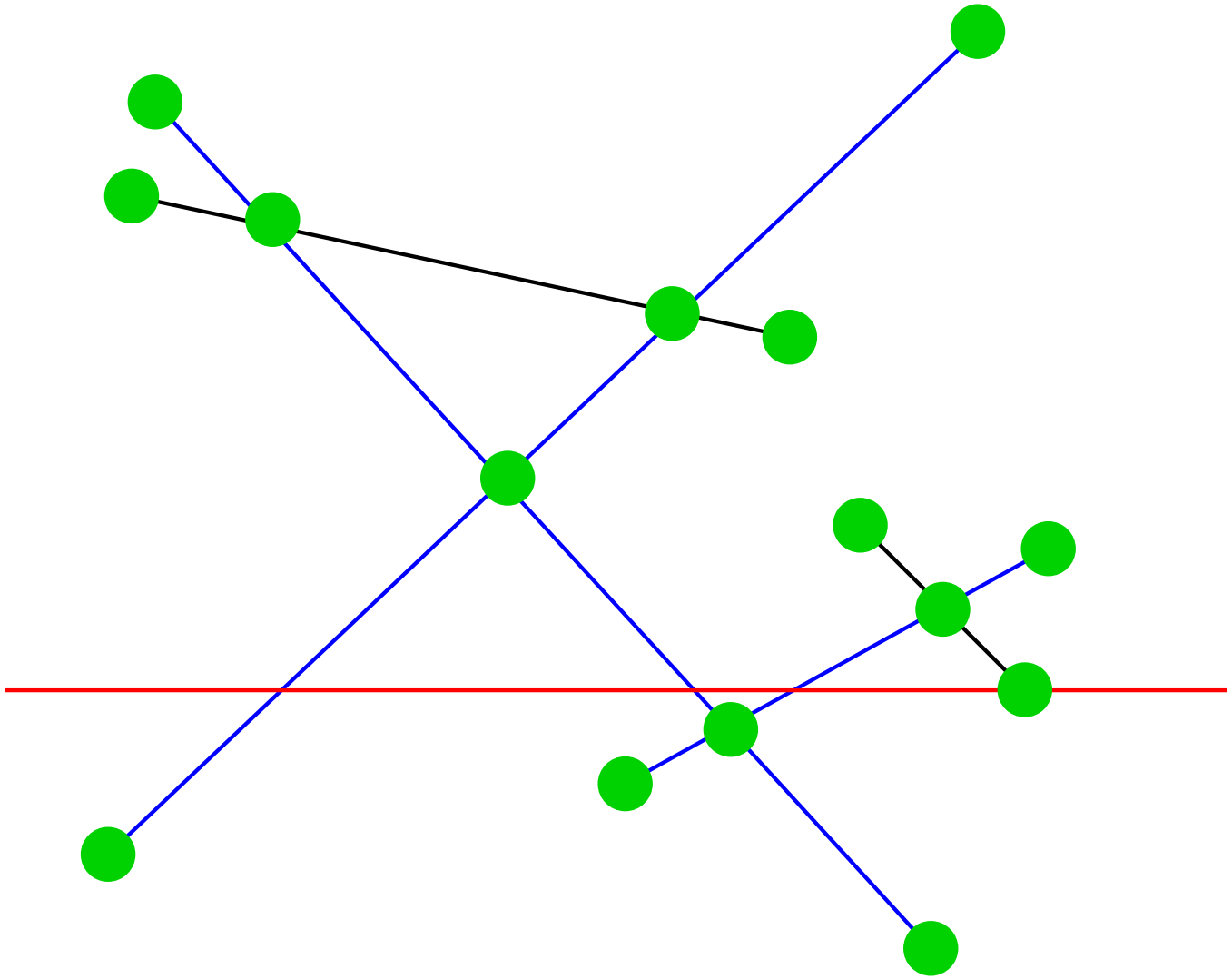
Verallgemeinerung – Beispiel



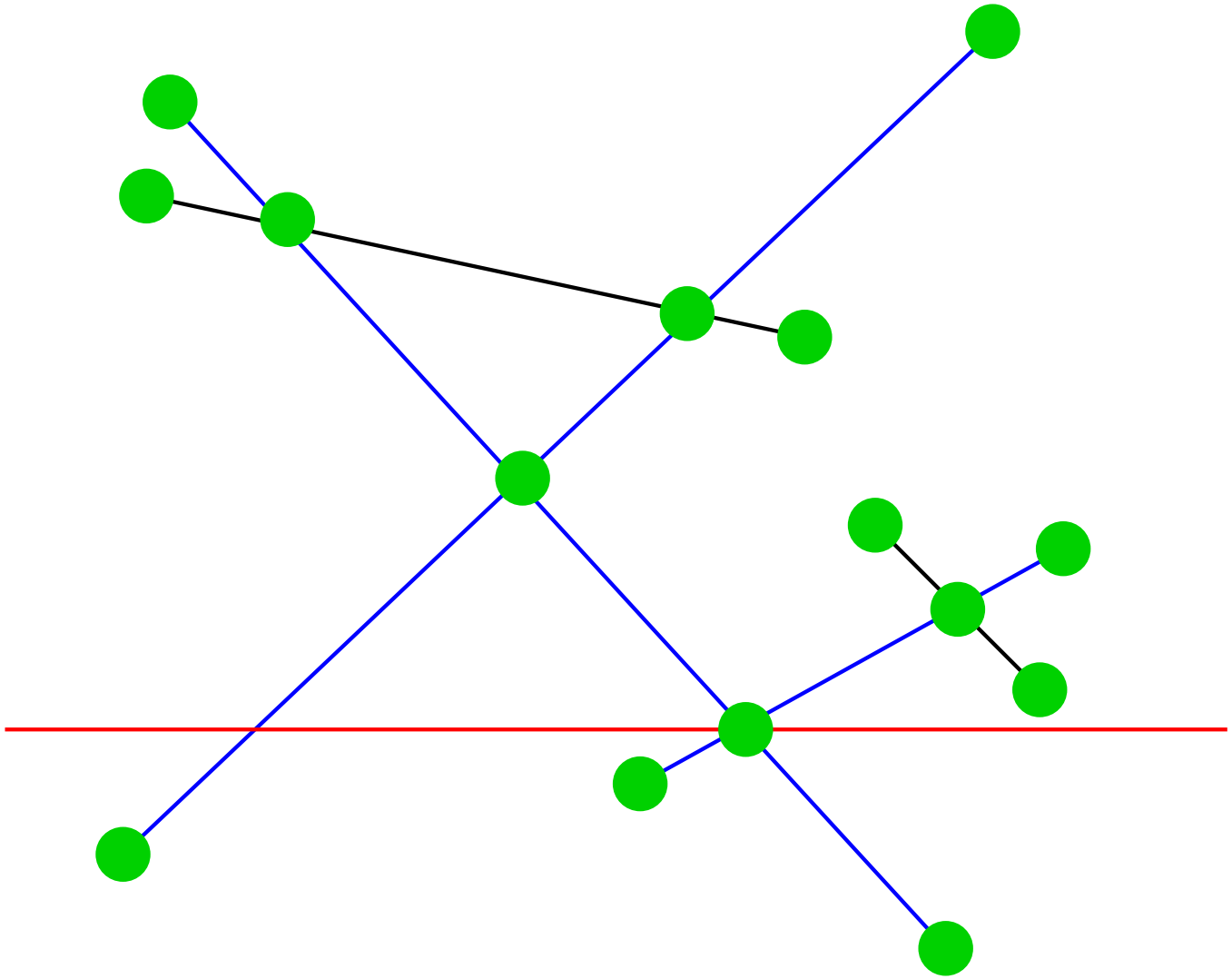
Verallgemeinerung – Beispiel



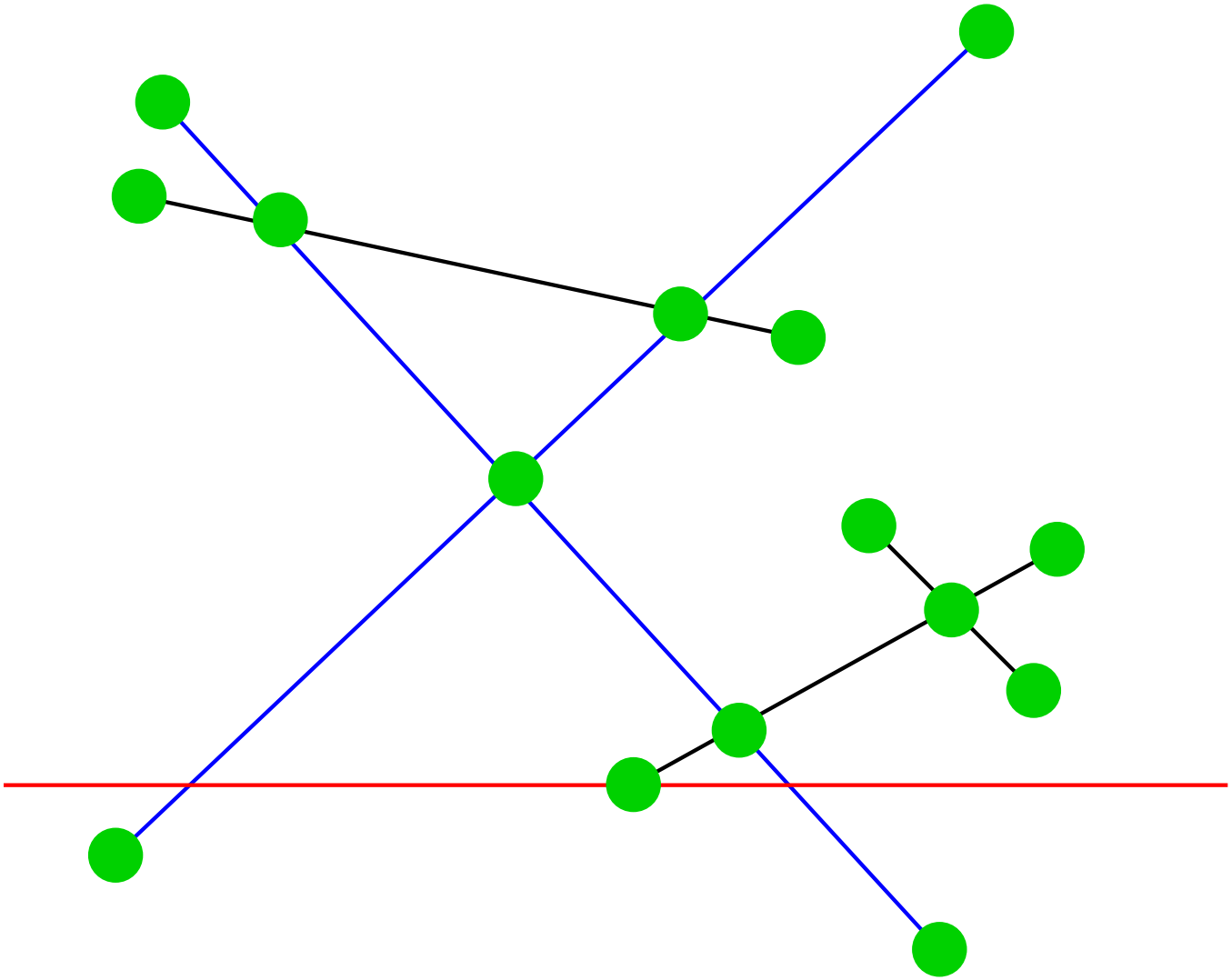
Verallgemeinerung – Beispiel



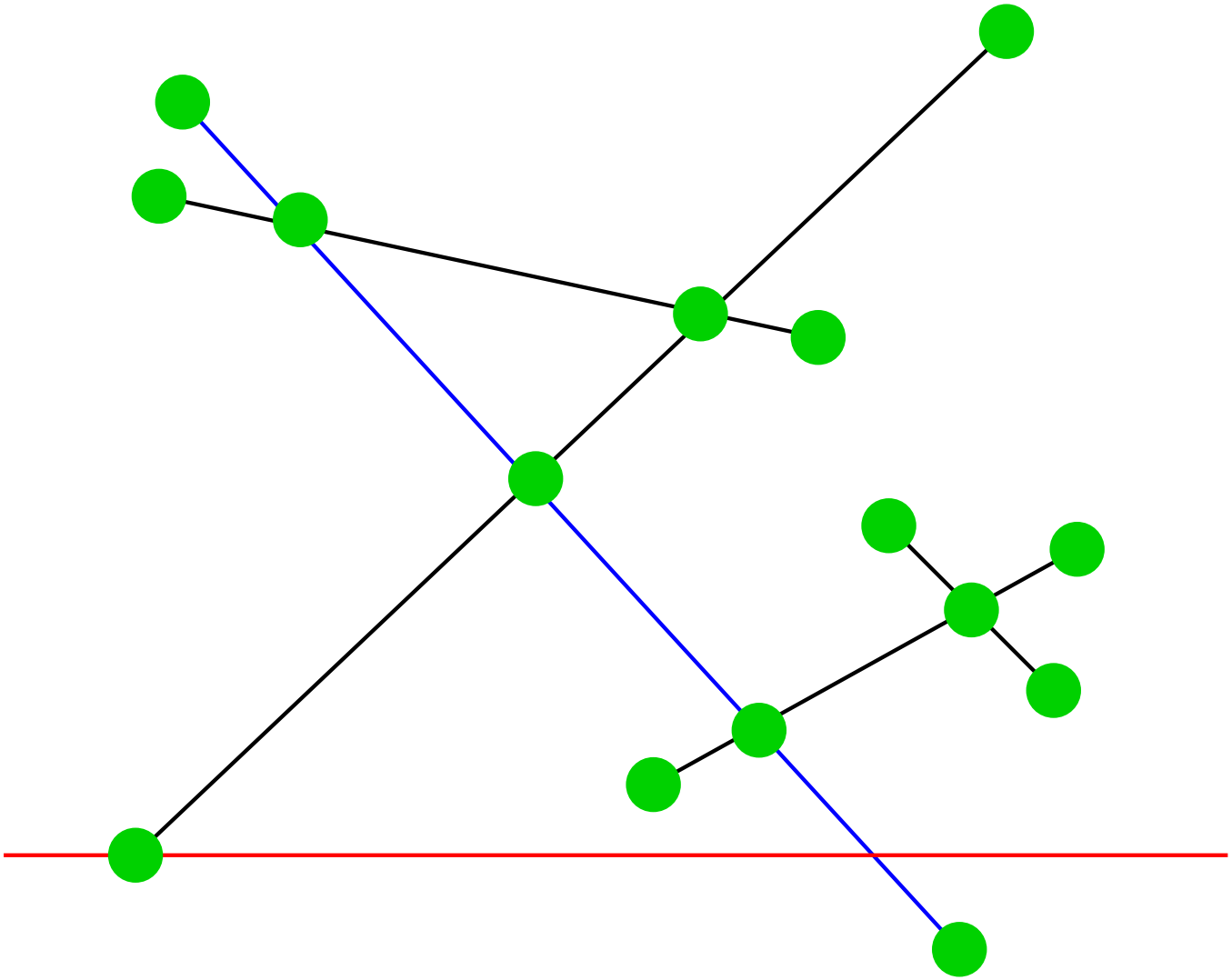
Verallgemeinerung – Beispiel



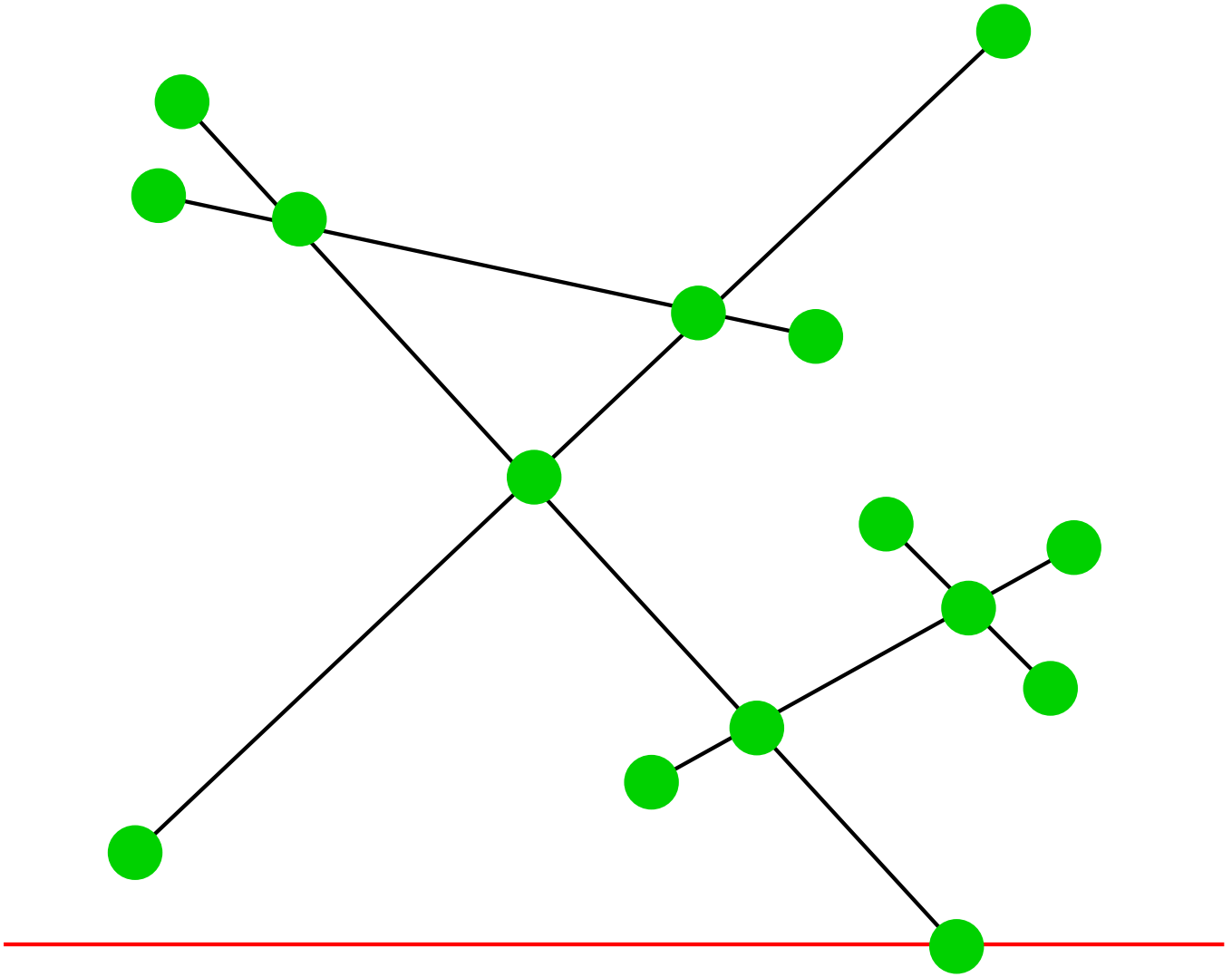
Verallgemeinerung – Beispiel



Verallgemeinerung – Beispiel



Verallgemeinerung – Beispiel



Verallgemeinerung – Analyse

Insgesamt: $\mathrm{O}((n+k)\log n)$

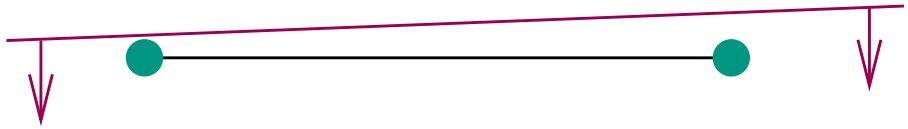
Verallgemeinerung – jetzt (fast) wirklich

Verbleibende Annahme: Keine Überlappungen

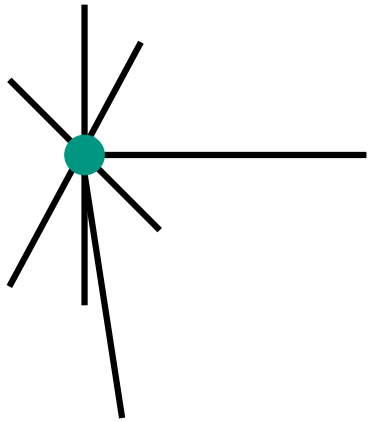
Ordnung für Q : $(x,y) \prec (x',y') \Leftrightarrow y > y' \vee y = y' \wedge x < x'$

(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.remove$(L \cup C)$

T.insert$(C \cup U)$ such that order just below p is correct

if $U \cup C = \emptyset$ **then**

findNewEvent$(T$.findPred$(p), T$.findSucc$(p), p)$

else

findNewEvent$(T$.findPred$(p), T$.findLeftmost$(p), p)$

findNewEvent$(T$.findRightmost$(p), T$.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.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 $\mathrm{O}(n)$:

lösche Schnittpunkte zwischen nicht benachbarten Strecken aus T.

Die werden ohnehin wieder eingefügt wenn sie wieder benachbart werden.

Mehr Linienschnitt

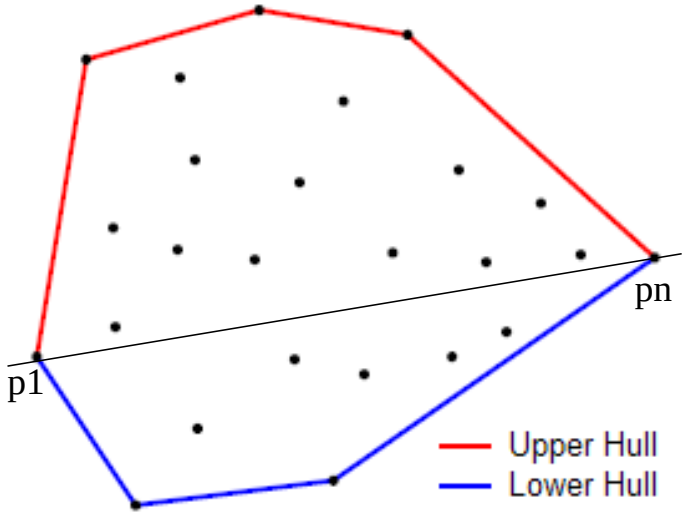
- [Bentley Ottmann 1979] Zeit $O((n+k)\log n)$
- [Chazelle Edelsbrunner 1988] Zeit $O(n\log n+k)$
- [Pach Sharir 1991] Zeit $O((n+k)\log n)$, Platz $O(n)$
- [Mulmuley 1988] erwartete Zeit $O(n\log n+k)$, Platz $O(n)$
- [Balaban 1995] Zeit $O(n\log n+k)$, Platz $O(n)$

10.2 2D Konvexe Hülle

Gegeben: Menge $P = \{p_1, \ldots, p_n\}$ 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.



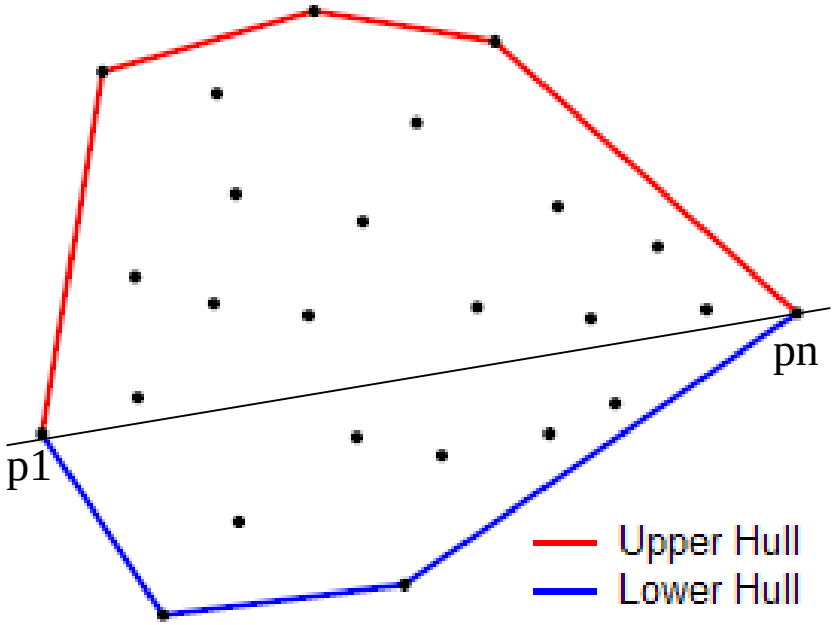
Konvexe Hülle

sortiere P lexikographisch, d.h., ab jetzt

$p_1 < p_2 < \cdots < p_n$

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$(p_1, \ldots, p_n)$

$L = \langle p_n, p_1, p_2 \rangle$: Stack **of** Point

invariant L is the upper hull of $\langle p_n, p_1, \ldots, p_i \rangle$

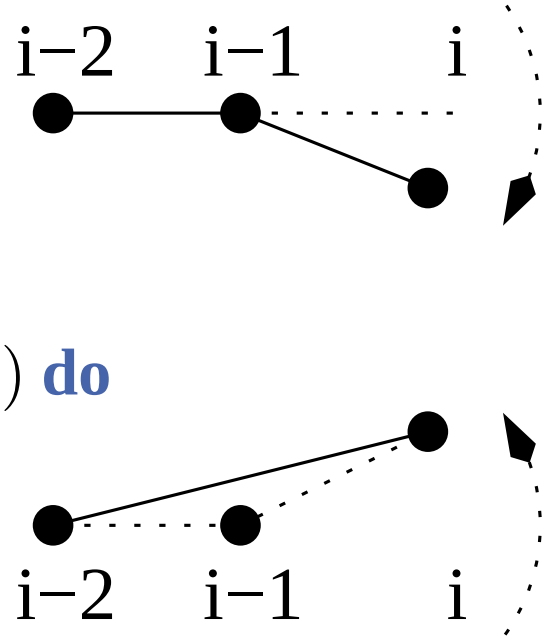
for $i := 3$ **to** n **do**

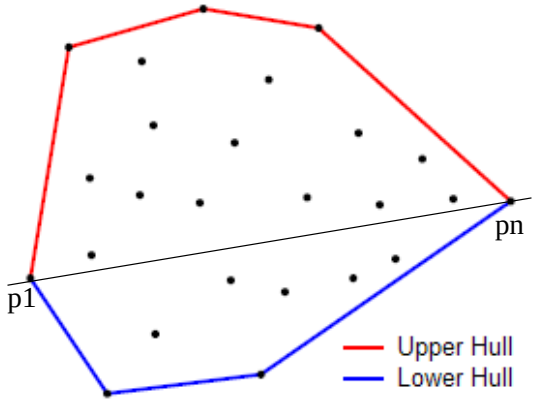
while $\neg$rightTurn$(L.\text{secondButlast}, L.\text{last}, p_i)$ **do**

L.pop

$L := L \circ \langle p_i \rangle$

return L





Graham's Scan – Beispiel

Function upperHull$(p_1, \ldots, p_n)$

$L = \langle p_n, p_1, p_2 \rangle$: Stack **of** Point

invariant L is the upper hull of $\langle p_n, p_1, \ldots, p_i \rangle$

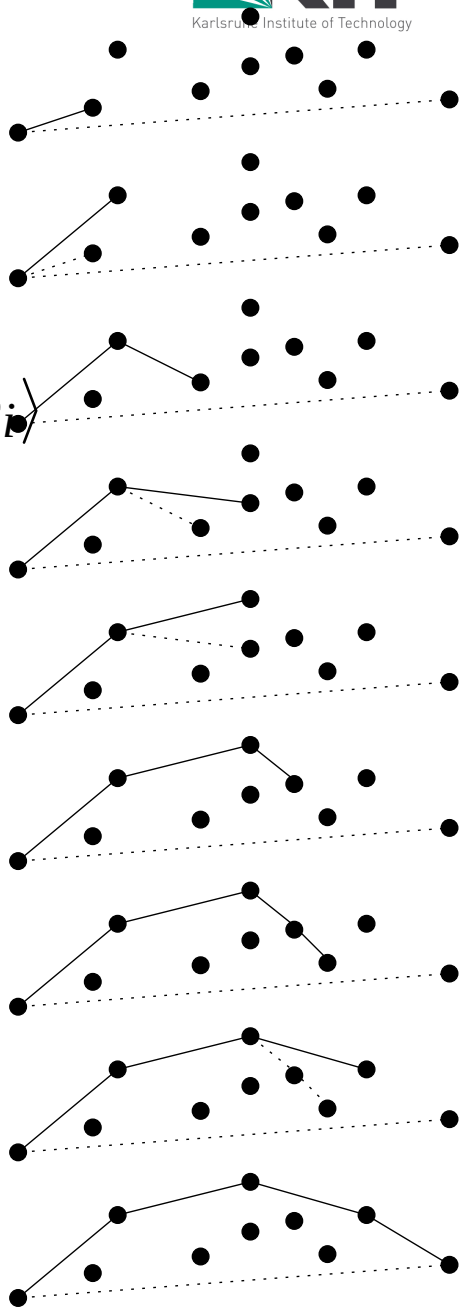
for $i := 3$ **to** n **do**

while $\neg$rightTurn$(L$.secondButlast, L.last, $p_i)$ **do**

L.pop

$L := L \circ \langle p_i \rangle$

return L



Graham's Scan – Analyse

Function upperHull$(p_1, \ldots, p_n)$

$L = \langle p_n, p_1, p_2 \rangle$: Stack **of** Point

invariant L is the upper hull of $\langle p_n, p_1, \ldots, p_i \rangle$

for $i := 3$ **to** n **do**

while $\neg$rightTurn$(L.\text{secondButlast}, L.\text{last}, p_i)$ **do**

L.pop

$L := L \circ \langle p_i \rangle$

return L

Sortieren $+\mathrm{O}(n)$

Wieviele Iterationen der While-Schleife insgesamt?

3D Konvexe Hülle

Geht in Zeit $O(n \log n)$ [Preparata Hong 1977]

Konvexe Hülle, $d \geq 4$

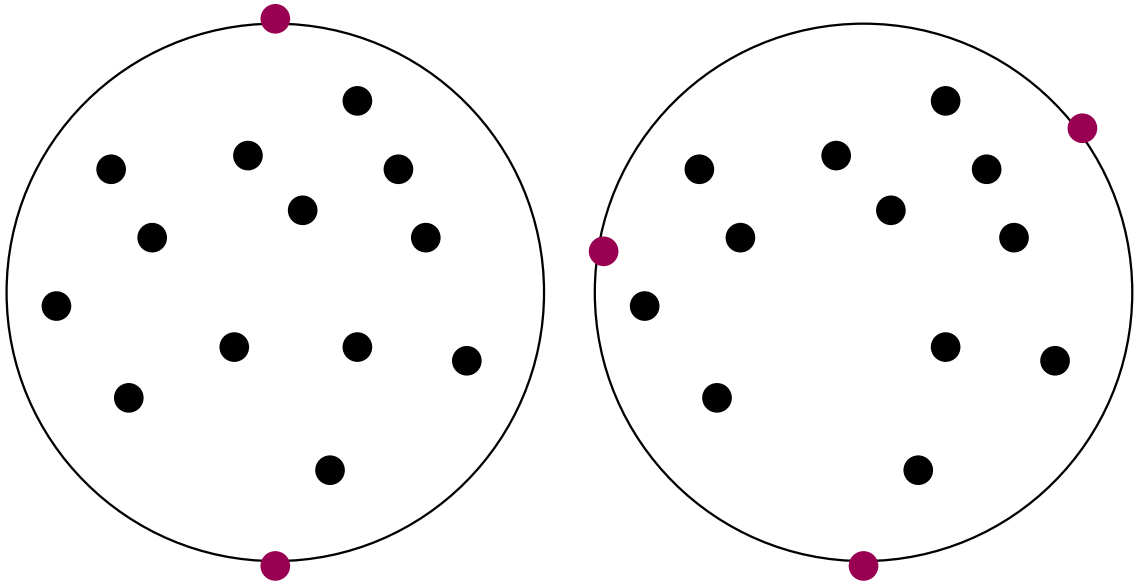
Ausgabekomplexität $O\left(n^{\lfloor d/2 \rfloor}\right)$

10.3 Kleinste einschließende Kugel

Gegeben: Menge $P = \{p_1, \ldots, p_n\}$ 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 smallestEnclosingBallWithPoints(P, Q)

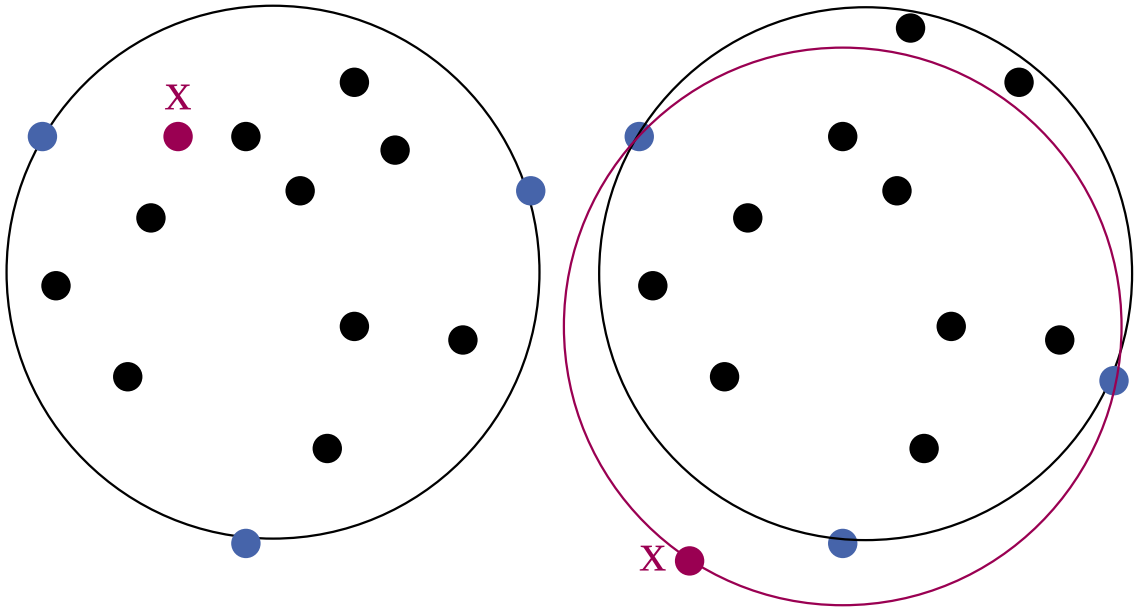
if $|P| = 1 \vee |Q| = d+1$ **then return** ball(Q)

pick random $x \in P$

B:= smallestEnclosingBallWithPoints$(P \setminus \{x\}, Q)$

if $x \in B$ **then return** B

return smallestEnclosingBallWithPoints$(P \setminus \{x\}, Q \cup \{x\})$



Kleinste einschließende Kugel – Korrektheit

Function smallestEnclosingBallWithPoints(P, Q)

  if $|P| = 1 \vee |Q| = d+1$ **then return** ball(Q)

  pick random $x \in P$

  B:= smallestEnclosingBallWithPoints$(P \setminus \{x\}, Q)$

  if $x \in B$ **then return** B

  return smallestEnclosingBallWithPoints$(P \setminus \{x\}, Q \cup \{x\})$

z.Z.: $x \notin B \rightarrow x$ ist auf dem Rand von sEB(P)

Wir zeigen Kontraposition:

x nicht auf dem Rand von sEB(P)

$\longrightarrow$ sEB(P) = sEB$(P \setminus \{x\}) = B$

z.Z.: sEBs sind eindeutig!

Also $x \in B$

Lemma: sEB(P) ist eindeutig bestimmt.

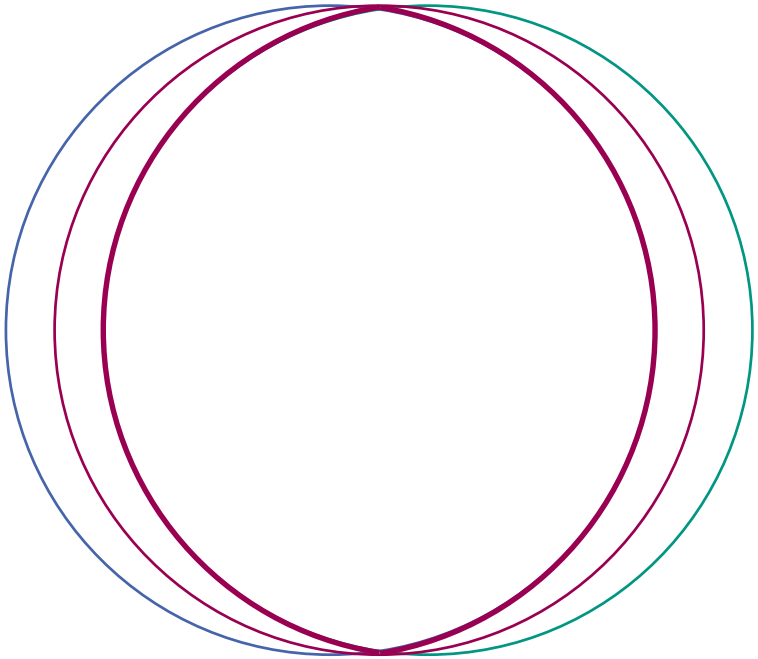
Beweis: Annahme, $\exists$sEBs $B_1 \neq B_2$

$\longrightarrow P \subseteq B_1 \wedge P \subseteq B_2$

$\longrightarrow P \subseteq B_1 \cap B_2 \subseteq \text{sEB}(B_1 \cap B_2) =: B$

Aber dann ist radius$(B) <$ radius(B_1)

Widerspruch zur Annahme, dass B_1 eine sEB ist. $\square$



Kleinste einschließende Kugel – Analyse

Wir zählen die erwartete Anzahl der Tests $x \in B$, $T(p,q)$.

$$T(p,d+1) = T(1,p) = 0 \qquad \text{Basis der Rekurrenz}$$

$$\begin{aligned} T(p,q) &\leq 1 + T(p-1,q) + \mathbb{P}[x \notin B]\, T(p,q+1) \\ &\leq 1 + T(p-1,q) + \frac{d+1-q}{p} T(p,q+1) \end{aligned}$$

Kleinste einschließende Kugel – Analyse, $d = 2$

$T(p, d+1) = T(1, p) = 0$

$$T(p,q) \leq 1 + T(p-1,q) + \frac{d+1-q}{p} T(p,q+1)$$

$$T(p,2) \leq 1 + T(p-1,2) + \frac{1}{p} T(p,3) \leq 1 + T(p-1,2) \leq p$$

$$T(p,1) \leq 1 + T(p-1,1) + \frac{2}{p} T(p,2)$$

$$\leq 1 + T(p-1,1) + \frac{2}{p} p = 3 + T(p-1,1) \leq 3p$$

$$T(p,0) \leq 1 + T(p-1,0) + \frac{3}{p} T(p,1)$$

$$\leq 1 + T(p-1,0) + \frac{3}{p} 3p = 10 + T(p-1,0) \leq 10p$$

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

    $\longrightarrow$ Lösung bleibt gleich oder ist an Lösungsdef. beteiligt

10.4 2D Bereichssuche (range search)

Daten: $P = \{p_1, \ldots, p_n\} \subseteq \mathbb{R}^2$

Anfragen: achsenparallele Rechtecke $Q = [x, x'] \times [y, y']$

finde $P \cap Q$ (range reporting)

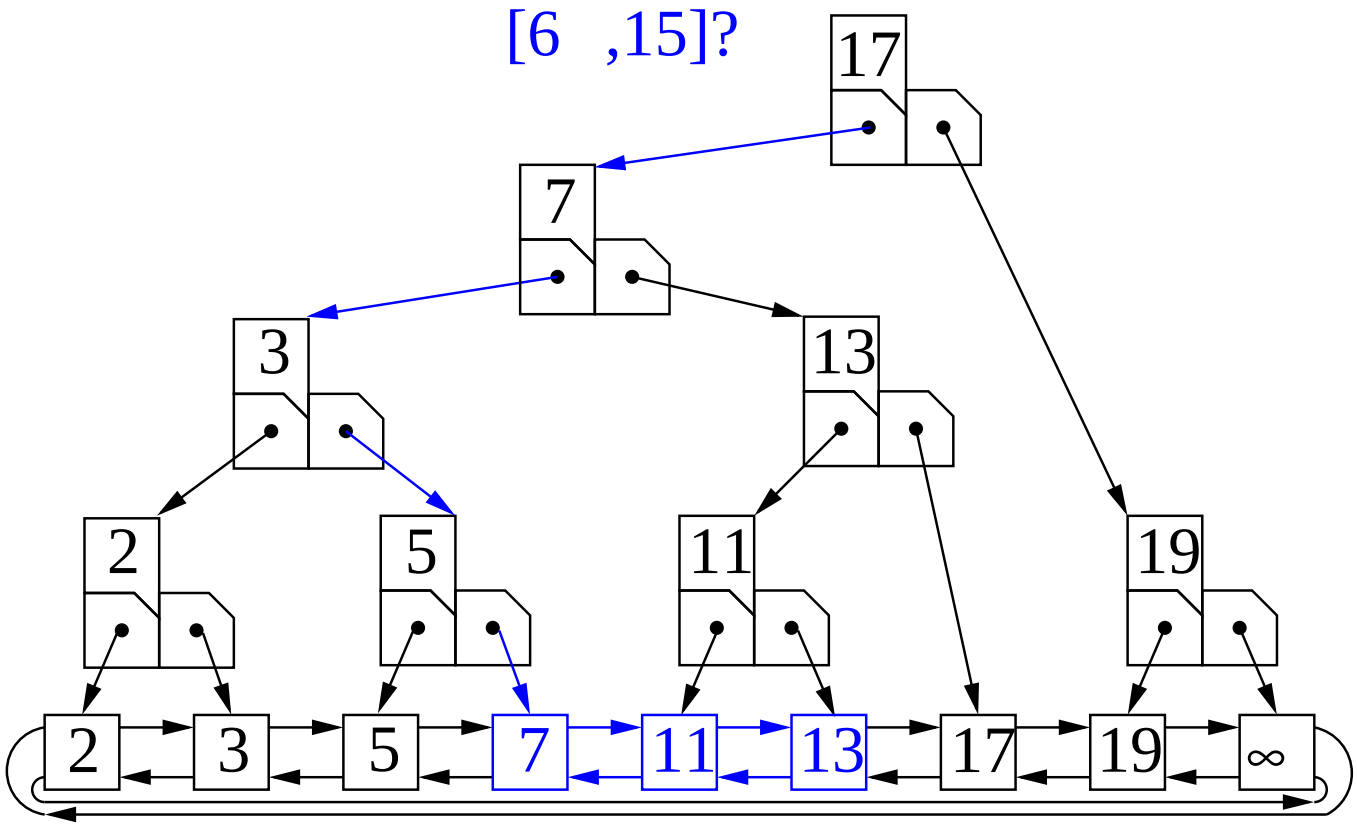
oder $k = |P \cap Q|$ (range counting)

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 $P = \{(x_1,y_1),\ldots,(x_n,y_n)\}$ durch ihren Rang:

$x_i \rightarrow$ Rang von x_i in $\{x_1,\ldots,x_n\}$

$y_i \rightarrow$ Rang von y_i in $\{y_1,\ldots,y_n\}$

Reduktion auf $1..n \times 1..n$

Ersetze Koordinaten $P = \{(x_1,y_1),\ldots,(x_n,y_n)\}$ durch ihren Rang:

P_x:= sort($\langle x_1,\ldots,x_n\rangle$); P_y:= sort($\langle y_1,\ldots,y_n\rangle$)

P:= $\{(\text{binarySearch}(x,P_x),\text{binarySearch}(y,P_y)) : (x,y) \in P\}$

Function rangeQuery($[x,x'] \times [y,y']$)

x:= binarySearchSucc(x,P_x); x':= binarySearchPred(x',P_x)

y:= binarySearchSucc(y,P_y); y':= binarySearchPred(y',P_y)

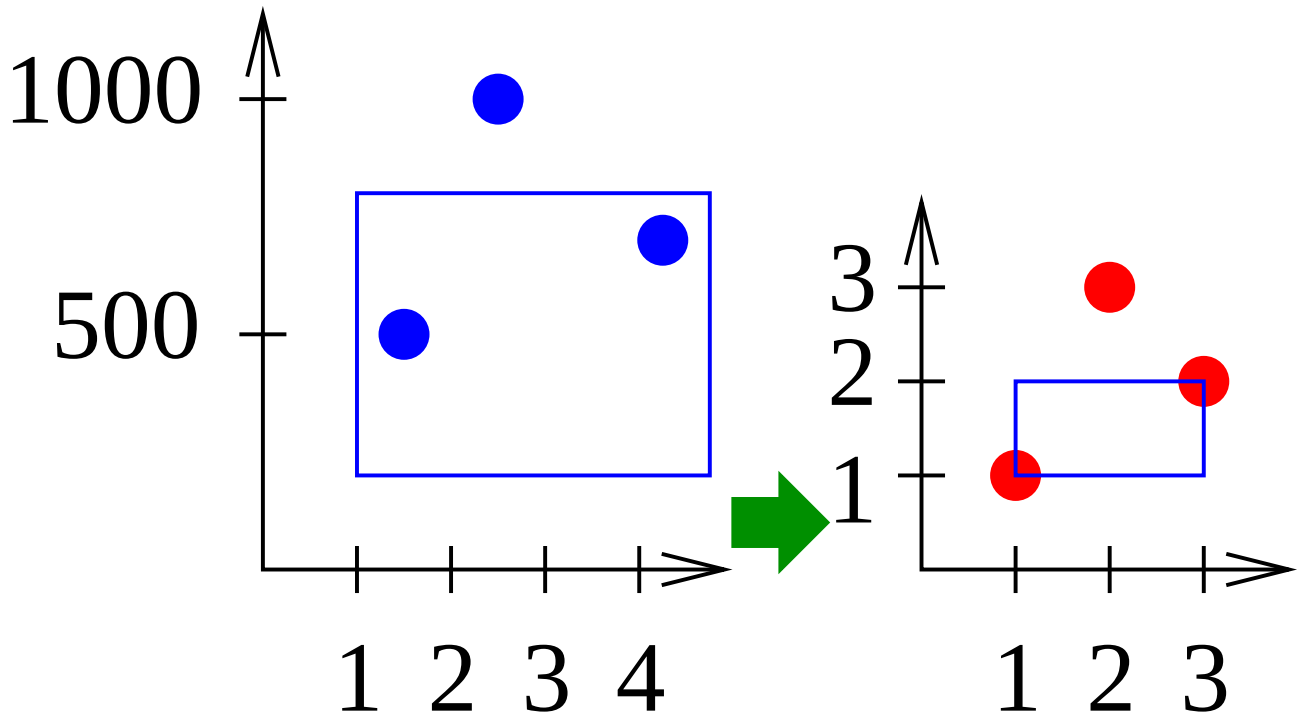
R:= intRangeQuery($[x,x'] \times [y,y']$)

return $\{(P_x[x],P_y[y]) : (x,y) \in R\}$

Zeit $O(n \log n)$

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$(X = \langle x_1, \ldots, x_n \rangle)$// represents $(x_1, 1), \ldots, (x_n, n)$

//Constructor:

if $n < n_0$ **then** store X directly

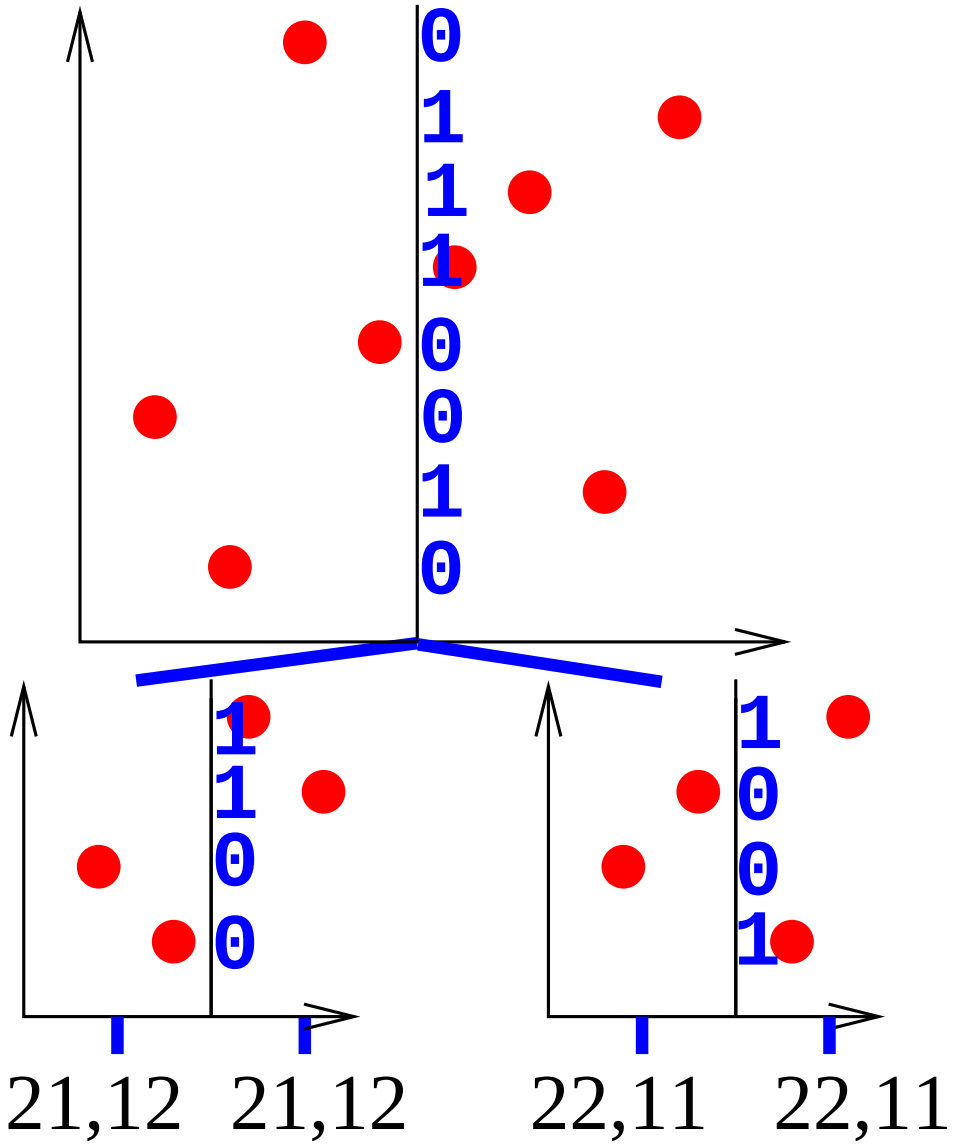
else

store bitvector b with $b[i] = 1$ iff $x_i > \lfloor n/2 \rfloor$

ℓ:= 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$([x,x'] \times [y,y'])$

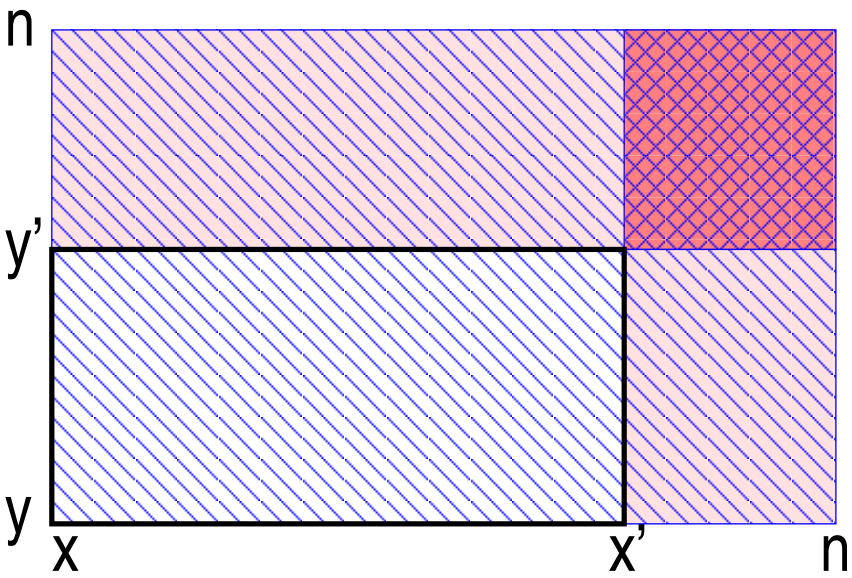
return

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 $n \le n_0$ **then return** $|[x,n]\times[y,n]\cap P|$ // brute force

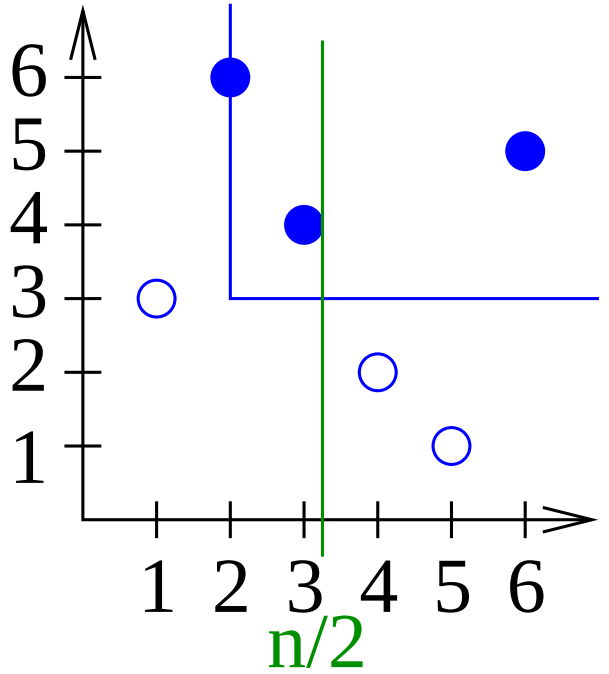
  y_r:= b.rank(y) // Number of els $\le y$ in right half

  if $x \le \lfloor n/2 \rfloor$ **then**

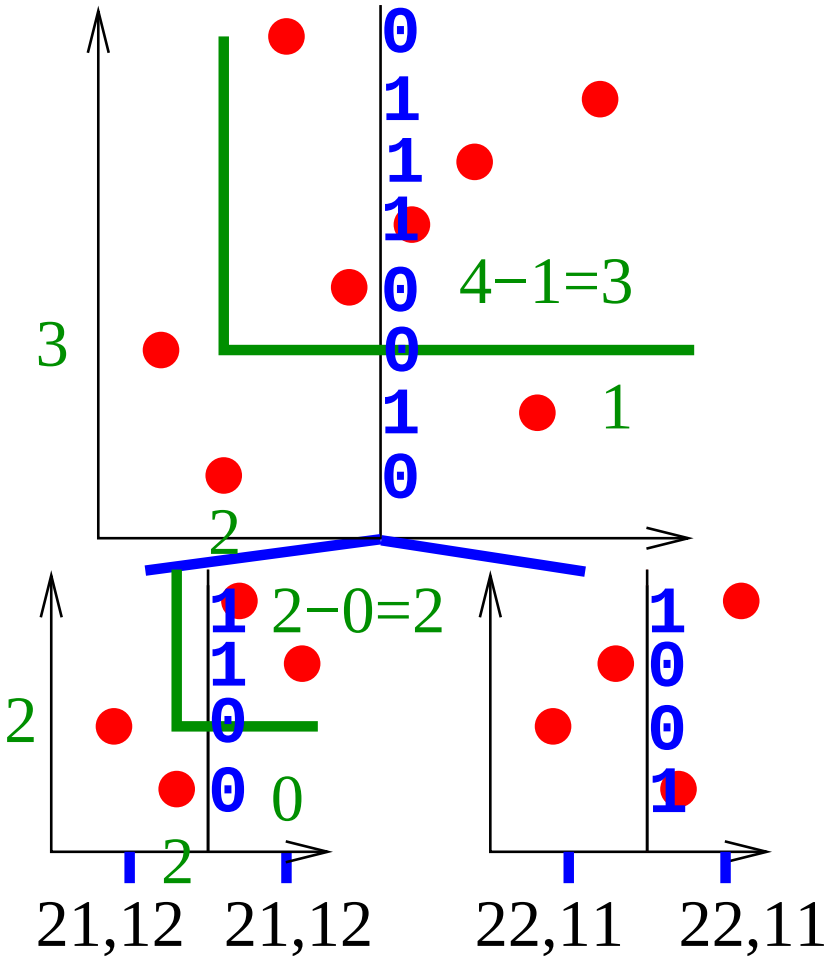
    return ℓ.intDominanceCount$(x, y-y_r) + \lceil n/2 \rceil - y_r$

  else

    return r.intDominanceCount$(x-\lfloor n/2 \rfloor, y_r)$



Beispiel



Analyse

Nur ein rekursiver Aufruf.

Rekursionstiefe $\mathrm{O}(\log n)$.

rank in konstanter Zeit (s.u.)

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

Wavelet Tree Dominance Reporting Query

Function intDominanceReporting(x,y) // $[x,n]\times[y,n]\cap P$

  if $n \le n_0$ **then return** $[x,n]\times[y,n]\cap P$ // brute force

  $R := \emptyset$ // Result

  $y_r := b$.rank(y) // Number of els $\le y$ in right half

  if $x \le \lfloor n/2 \rfloor$ **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** $R := R \cup$ 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

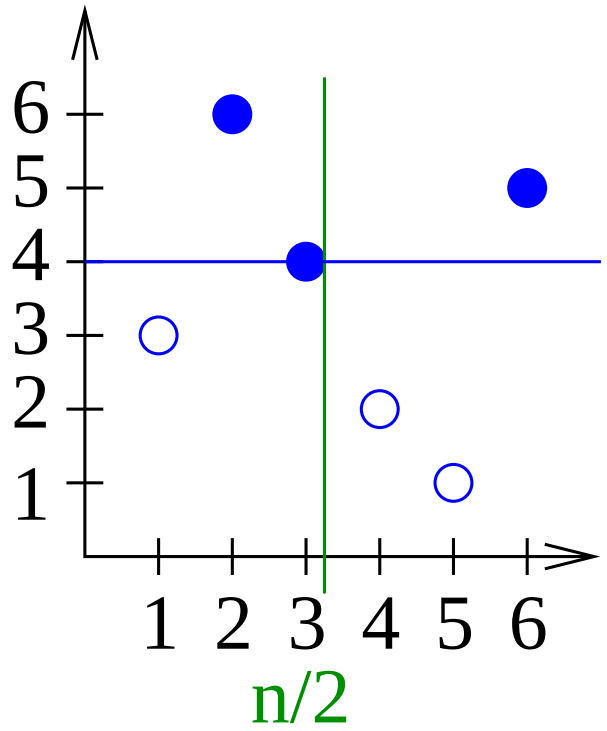
  y_r:= b.rank(y) // Number of els $\leq y$ in right half

  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

$T(n_0,0) = \mathrm{O}(1)$

$T(n,0) = T(n/2,0) + O(1) \Longrightarrow T(n,0) = \mathrm{O}(\log n)$

$T(n,k) = T(n/2,k') + T(n/2,k-k') + O(1)$ also

$T(n,k) \leq ck' \log n + c(k-k')\log n + c = c(1+k\log n) = \mathrm{O}(k\log n)$

Zeit $\mathrm{O}(k+\log n)$ brauch zusätzlichen Faktor $\log n$ Platz.

Z.B. komplette Listen auf allen Ebenen speichern

Übungsaufgabe?

Allgemeine Reporting Query

4-seitig

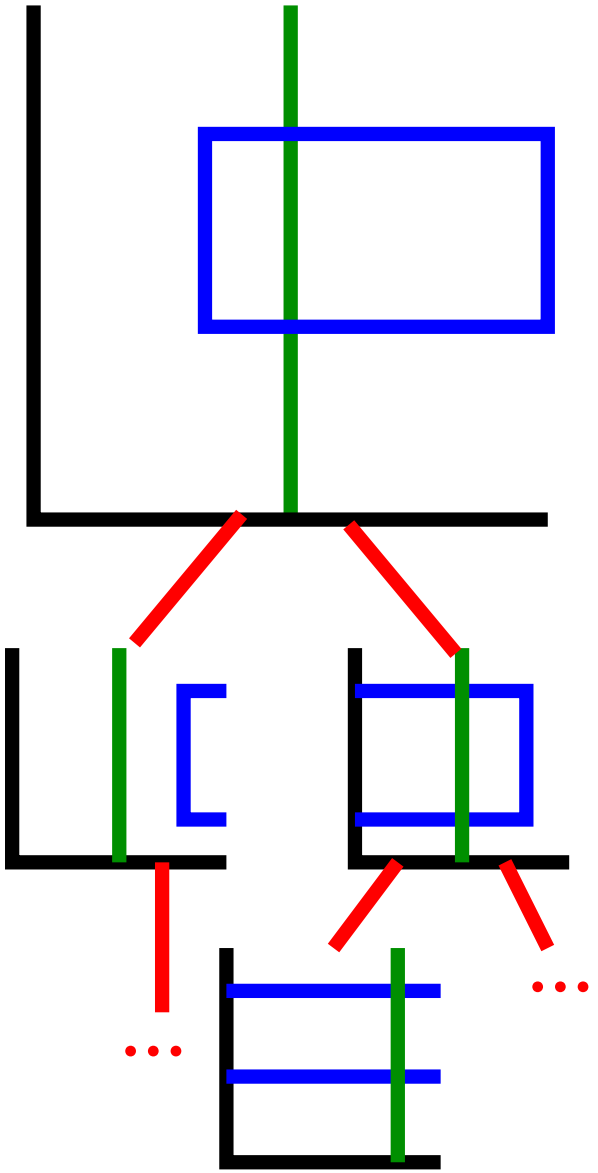
$\rightsquigarrow$

3-seitig (2 Varianten)

$\rightsquigarrow$

y-range

Analog oneSidedReporting (zwei Ranks statt einem)



Bitvektoren v mit rank in $O(1)$

Wähle $B = \Theta(\log n)$.

Vorberechnung bRank$[i]$:= v.rank(iB)

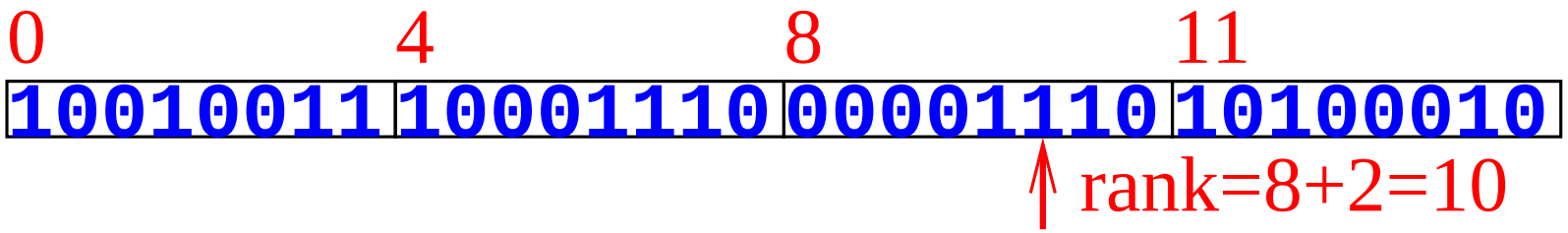
Zeit $O(n)$, Platz $O(n)$ bits

Reduktion auf logarithmische Eingabegröße:

Function rank(j) **return** bRank$[j$ **div** $B] +$ rank$(v[B(j$ **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
- Bespiel: Baum mit Platz $2n + o(n)$ bits und Navigation in konstanter Zeit.

11 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 so umdefinieren, dass es polynomial lösbar wird.

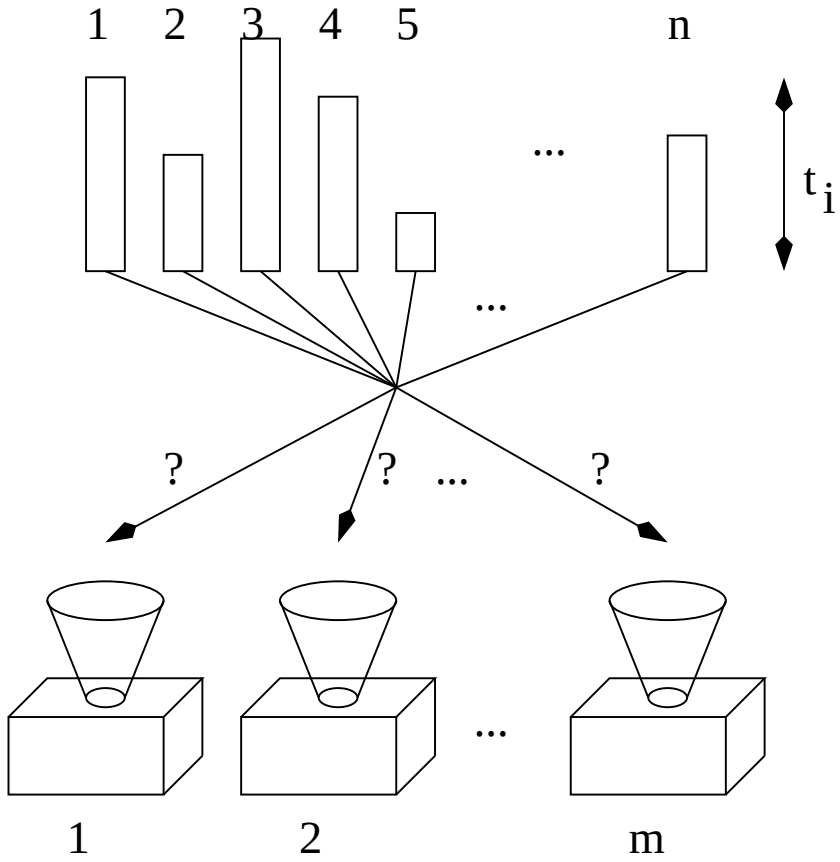
Scheduling unabhängiger gewichteter Jobs auf parallelen Machinen

$\mathbf{x}(j)$: Maschhine auf der Job j ausgeführt wird

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

Zielfunktion: Minimiere Makespan

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



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

List Scheduling

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

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

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

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 $\mathbf{x}$

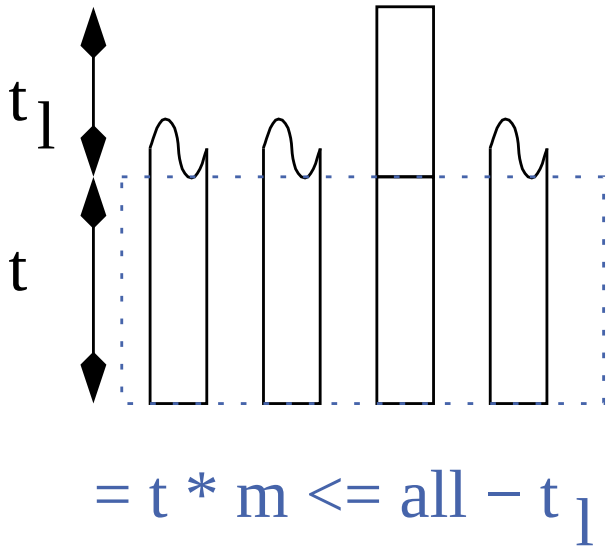
Viele Kleine Jobs

Lemma 21. *Falls ℓ 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 22. $L_{\max} \geq \sum_j \frac{t_j}{m}$

Lemma 23. $L_{\max} \geq \max_j t_j$

Der Approximationsfaktor

Definition:

Ein Minimierungsalgorithmus erzielt Approximationsfaktor ρ 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:

$$\frac{f(\mathbf{x})}{f(\mathbf{x}^*)} \qquad \text{(obere Schranke Lemma 21)}$$

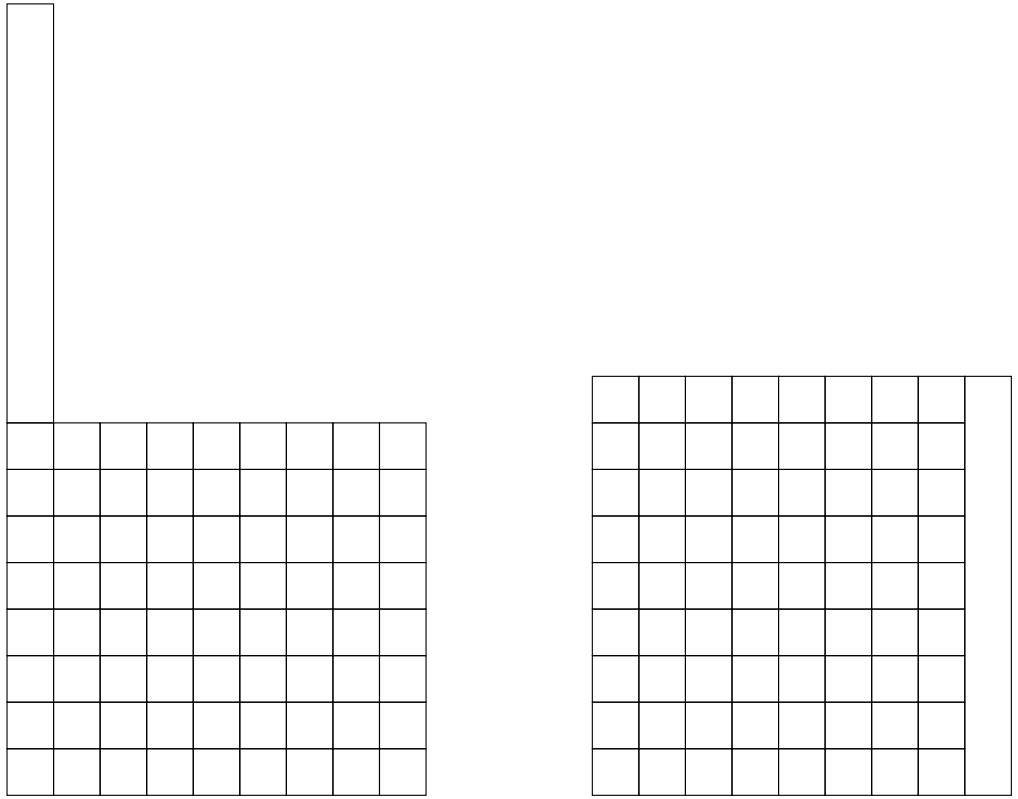
$$\leq \frac{\sum_j t_j/m}{f(\mathbf{x}^*)} + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} \qquad \text{(untere Schranke Lemma 22)}$$

$$\leq 1 + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} \qquad \text{(unterre Schranke Lemma 23)}$$

$$\leq 1 + \frac{m-1}{m} = 2 - \frac{1}{m}$$

Diese Schranke is bestmöglich

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



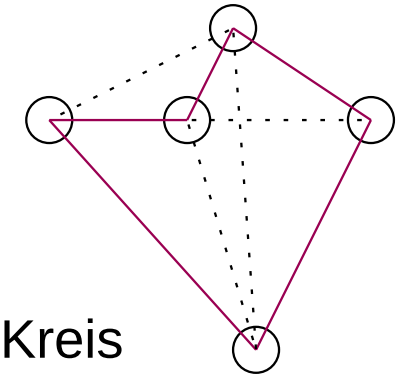
List Scheduling: 2m−1 OPT: m

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, . . .
- z.B. Unser Projekt: Jobs ihrerseits parallel

Nichtapproximierbarkeit des Handlungsreisendenproblems (TSP)



Gegeben ein Graph $G = (V, V \times V)$, finde einen einfachen Kreis $C = (v_1, v_2, \ldots, v_n, v_1)$ 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

HamiltonCycle$\leq_p$ a-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

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

$\longrightarrow \exists$ Tour mit Gewicht $n \longrightarrow \exists$ 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 unger. Graph $G = (V,E)$ mit Gewichtsfunktion $c : E \to \mathbb{R}_+$. Definiere

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

Beispiel: (ungerichteter) Strassengraph $\longrightarrow$ Abstandstabelle

Euler-Touren/-Kreise

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

Ein Pfad $P = \langle e_1, \ldots, e_m \rangle$ ist eine Euler-Tour falls $\{e_1, \ldots, e_m\} = E$.

(Jede Kante wird genau einmal besucht)

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

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

2-Approximation durch minimalen Spannbaum

Lemma 24.

Gesamtgewicht eines MST $\leq$

Gesamtgewicht jeder TSP-Tour

Algorithmus:

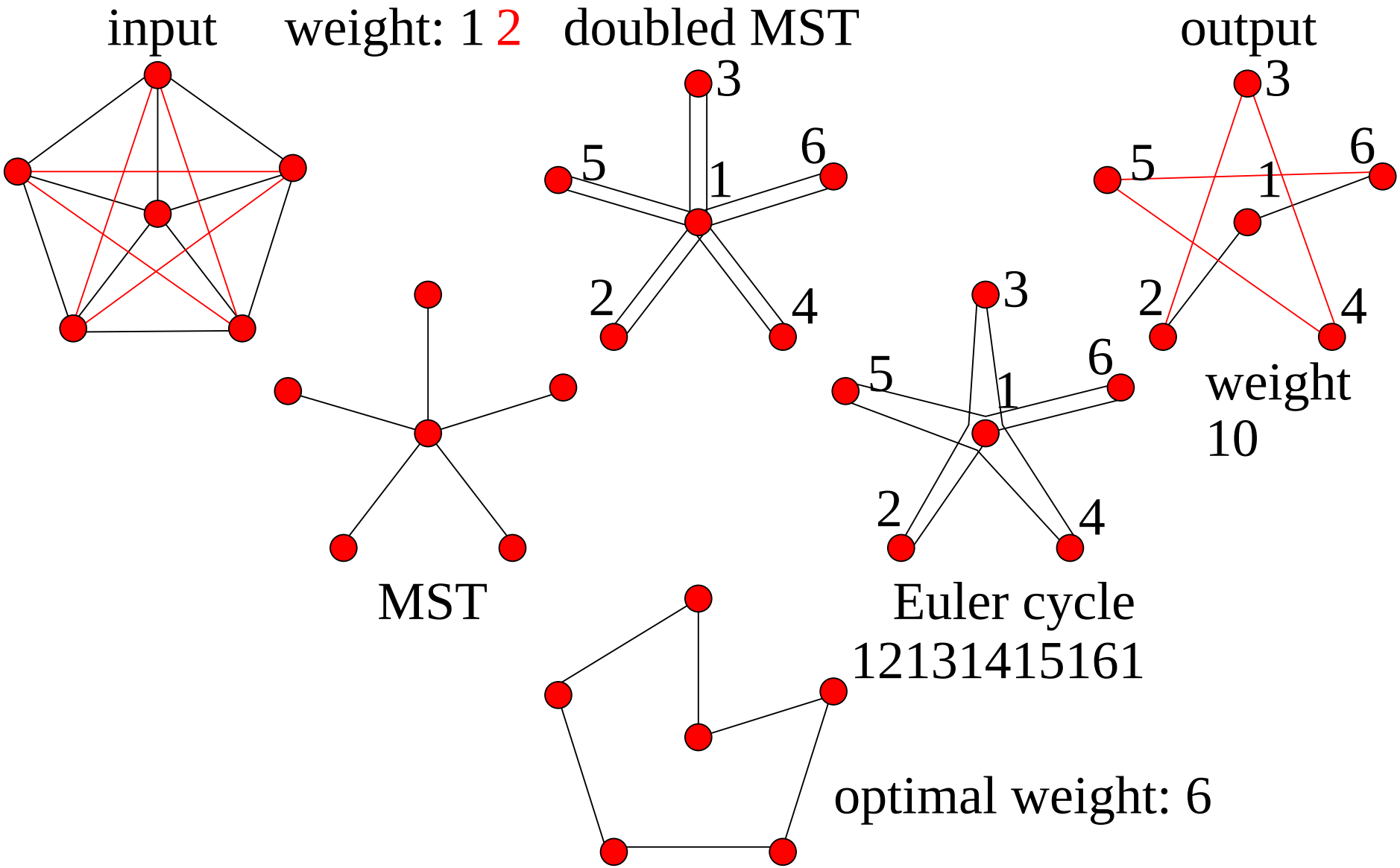
$T :=$ MST(G) // weight$(T) \leq$ opt

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

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

output removeDuplicates(T'') // shortcutting

Beispiel


input
weight: 1 2
doubled MST
output
3
5
1
6
2
4
weight
10
MST
Euler cycle
12131415161
optimal weight: 6

Beweis von Gewicht MST$\leq$ 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 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

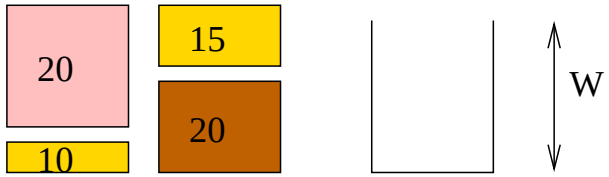
$\mathscr{A}$ ist pseudopolynomieller Algorithmus falls

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

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 $1,\ldots,i$ die Profit $\geq P$ ergeben.

Lemma 25.

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

Dynamische Prgrammierung 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 $\mathscr{A}$ ist ein

(Fully) Polynomial Time Approximation Scheme

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

Eingabe: Instanz I, Fehlerparameter ε

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

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

Beispielschranken

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/ε
n^{42/ε^3}	$\vdots$
$n + 2^{2^{1000/\varepsilon}}$	$\vdots$
$\vdots$	$\vdots$

FPTAS für Knapsack

$P := \max_i p_i$ // maximaler Einzelprofit

$K := \frac{\varepsilon P}{n}$ // Skalierungsfaktor

$p'_i := \left\lfloor \frac{p_i}{K} \right\rfloor$ // skaliere Profite

$\mathbf{x}' :=$ dynamicProgrammingByProfit$(\mathbf{p}', \mathbf{w}, C)$

gib $\mathbf{x}'$ aus

Lemma 26. $\mathbf{p}\cdot\mathbf{x}' \geq (1-\varepsilon)\text{opt}$.

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

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

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

$$K\mathbf{p}'\cdot\mathbf{x}^* \leq K\mathbf{p}'\cdot\mathbf{x}' = \sum_{i\in x'} K\left\lfloor\frac{p_i}{K}\right\rfloor \leq \sum_{i\in x'} K\frac{p_i}{K} = \mathbf{p}\cdot\mathbf{x}'.$$ 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 27. *Laufzeit* $O\left(n^3/\varepsilon\right)$.

Beweis. Die Laufzeit $O\left(n\hat{P}'\right)$ 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]

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

- Weniger buckets C_j (nichtuniform)
- Ausgefeilte dynamische Programmierung

Optimale Algorithmem 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.]

12 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: $k =$ Ausgabegröße

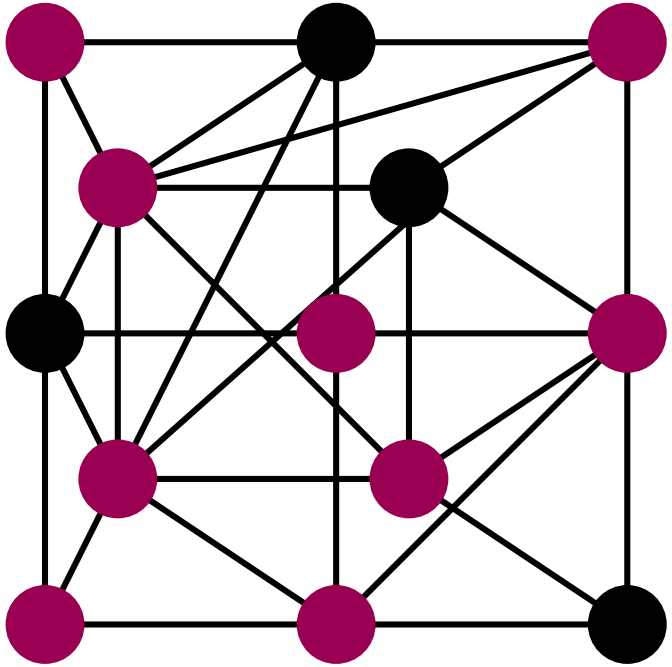
[Niedermeier, Invitation to Fixed Parameter Algorithms, Oxford U. Press, 2006]

Beispiel: VERTEX COVER (Knotenüberdeckung)

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

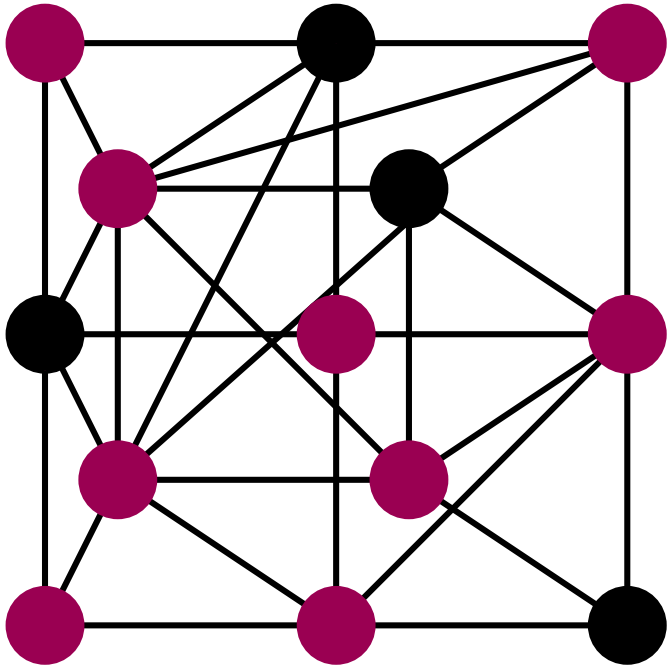
Parameter $k \in \mathbb{N}$.

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 $O\left(n^{k+1}\right)$ Brute-Force-Algorithmus



Fixed parameter tractable

Eine formale Sprache $L \in$ 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: $2^k n^2$, $k^{k!} n^{333}$, $n + 1.1^k$

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

Parametrisierte Komplexitätstheorie

Definition: Parametrisierte Reduktion von L auf L':
$(\mathscr{I},k) \to (f(\mathscr{I}),g(k))$

- $(f(\mathscr{I}),g(k))$ ist Ja-Instanz für L' gdw $(\mathscr{I},k)$ Ja-Instanz für L
- f hat Laufzeit $O(h(k)\cdot \text{poly}(n))$ für berechenbare Funktion h

$$\text{FPT} \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq NP$$

$W[i]$ abgeschlossen bzgl. parametrisierter Reduktion.

$\rightsquigarrow$ Vollständigkeitsbegriff analog NP-Vollständigkeit

Satz: Independent Set ist $W[1]$-vollständig

Satz: Dominating Set ist $W[2]$-vollständig

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$(G-v,k-1)\vee$ // 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

[Buss 1993]

Beobachtung: $\forall v \in V$: degree$(v) > k \Longrightarrow v \in$ Lösung $\vee$ unlösbar

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

  while $\exists v \in V$: 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: $\forall v \in V$: degree$(v) > k \Longrightarrow v \in$ Lösung $\vee$ unlösbar

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

  while $\exists v \in V$: 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 $\exists v \in V$: 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 $O(n^2)$

  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 $\exists v \in V$: degree$(v) > k$ **do**

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

    $G := G - v$

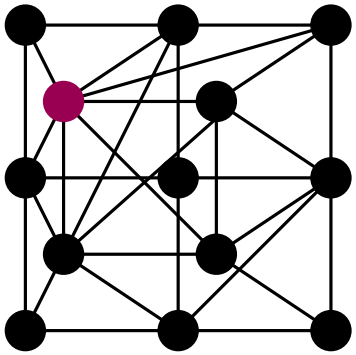
    $k := k - 1$

  remove isolated nodes

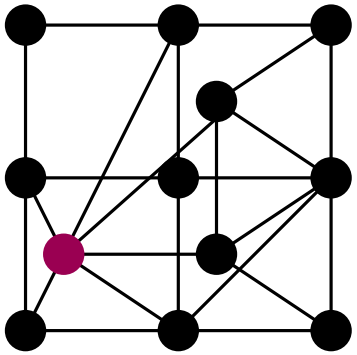
  if $|E| > k^2$ **then return** false

  return vertexCover(G, k)

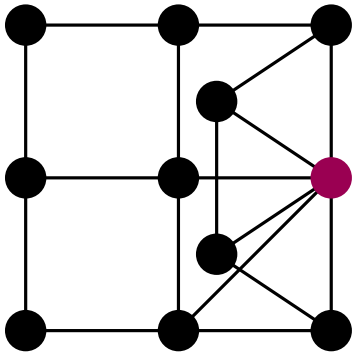
k=6?



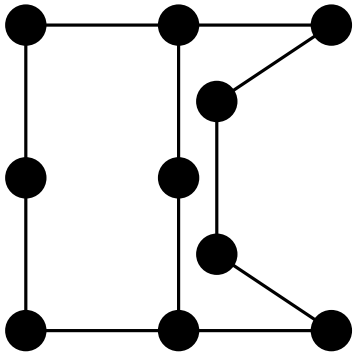
k=5?



k=4?



k=3?



11 Kanten
>
9=3*3
=>
no

Reduktionsregeln

0: nicht im cover

1: OBdA Nachbar im cover

Aufgabe: vertex cover für Bäume?

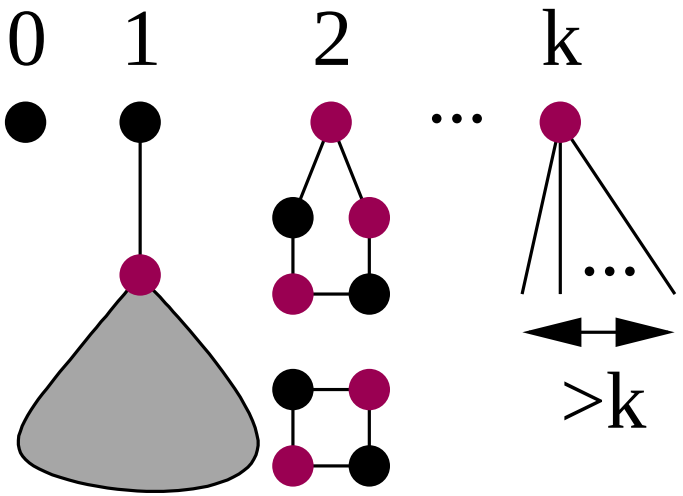
2: geht auch aber komplizierter. Aber,

trivial wenn alle Knoten Grad zwei haben

“Nimm jeden zweiten Knoten”

$> k$: muss ins cover

Mehr Regeln ?



Crown Reductions

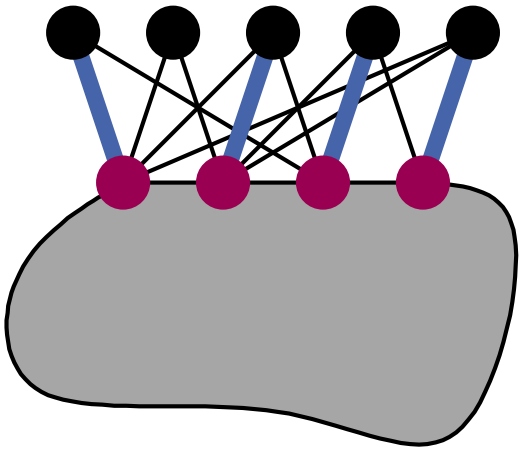
[Chor Fellows Juedes 2004]

[Linear Kernels in Linear Time, or How to Save k Colors in O(n^2) Steps, WG 2004, LNCS 3353]

Sei I independent set und $\mathcal{N}(I) = \{v \in V : \exists u \in I : \{u,v\} \in E\}$ die Nachbarschaft von I. Sei M ein Matching maximaler Größe im bipartiten Graph $(I \cup \mathcal{N}(I), \{\{u,v\} : u \in I, v \in \mathcal{N}(I)\})$.

$|M| = |\mathcal{N}(I)| \Rightarrow \exists$ min. vertex cover, das $\mathcal{N}(I)$ enthält.

Berechnungsheuristiken mittels max. cardinality matching.



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 $\exists v \in V$: degree$(v)=1$ **then**

return vertexCover2$(G-$ neighbor$(v),k-1)$

if $\exists v \in V$: 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

$$T(k)=(n+m)+T(k-1)+T(k-3)=O\left((n+m)1.4656^k\right)$$

$\rightsquigarrow$ benutze erzeugende Funktionen

Weitere Verbesserungen

- Kerne der Größe $2k$ (wieder mittels Matching-Algorithmen)
- Reduziere Zeit pro rekursivem Aufruf auf $\mathrm{O}(1)$
- Detaillierte Fallunterscheidungen $\leadsto$

  kleinere Konstante im exponentiellen Teil

$$\leadsto \mathrm{O}\left(1.2738^k + kn\right)$$

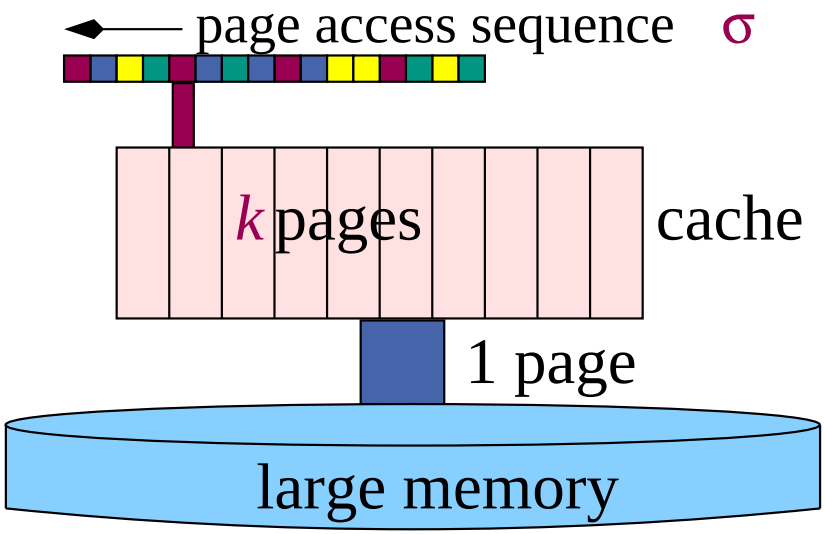
[Chen Kanj Xia 2006]

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:

$$C_{ALG} = \sup_{\sigma} \frac{\mathrm{ALG}(\sigma)}{\mathrm{OPT}(\sigma)}$$

  (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 $y > 6$.

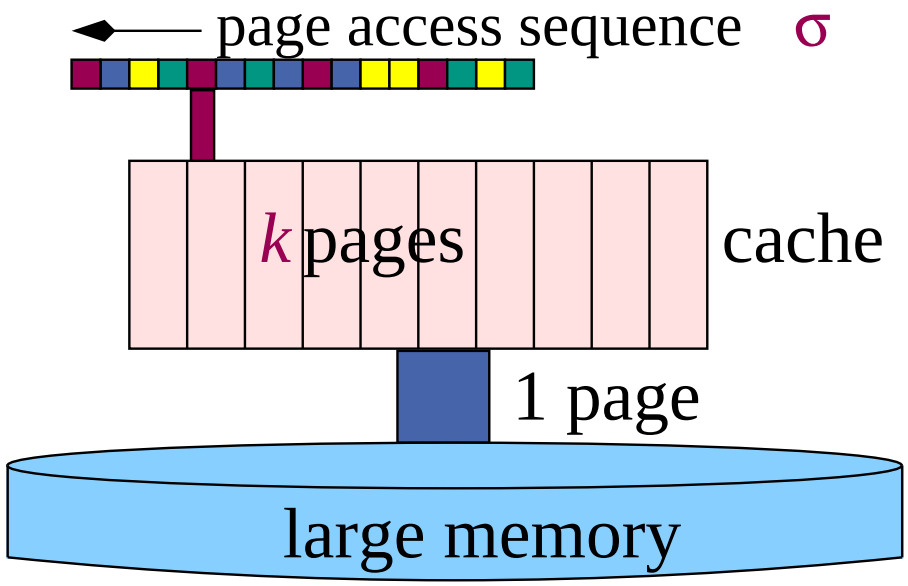
  You pay $300 + 50(y-1)$, OPT pays only 300

$$\frac{250+50y}{300} = \frac{5+y}{6} \geq 2.$$

- Idea: do not pay the large cost (buy skis) until you would have paid the same amount in small costs (rent)

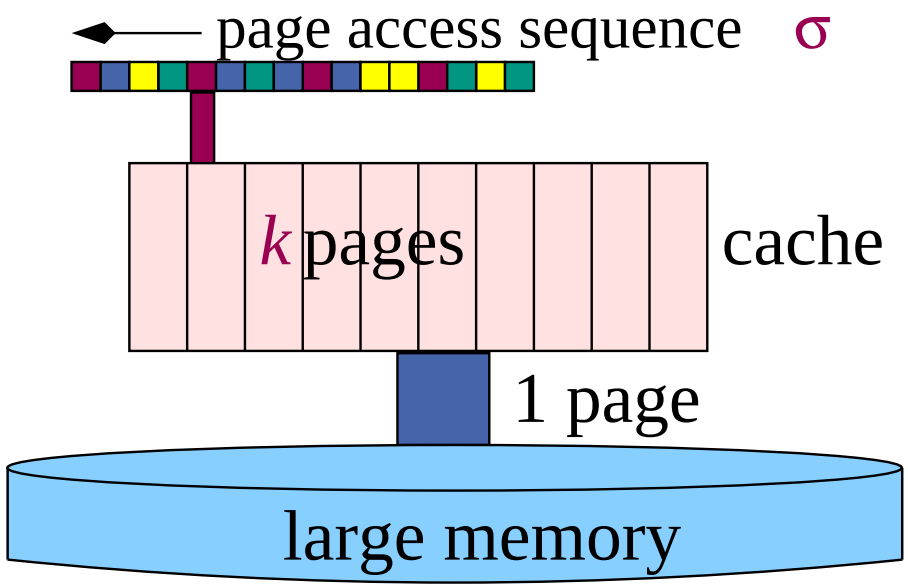
Paging

- 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)?



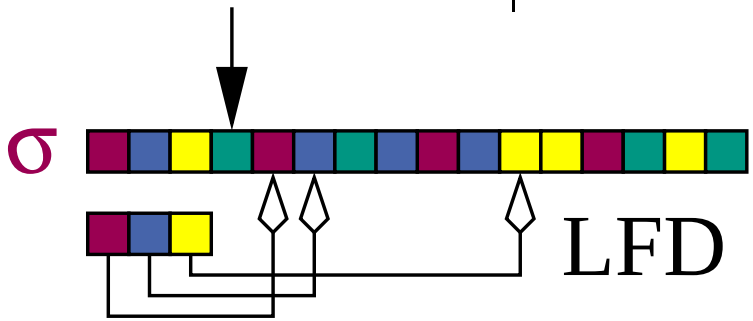
Definitions

- 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

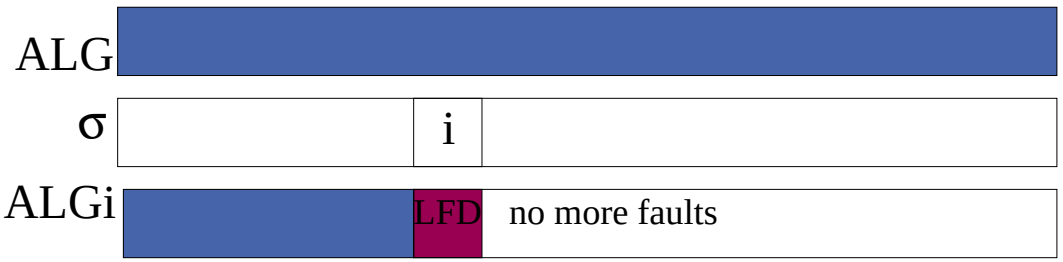


Longest Forward Distance is optimal

We show: any optimal offline algorithm can be changed to act like LFD without increasing the number of page faults.

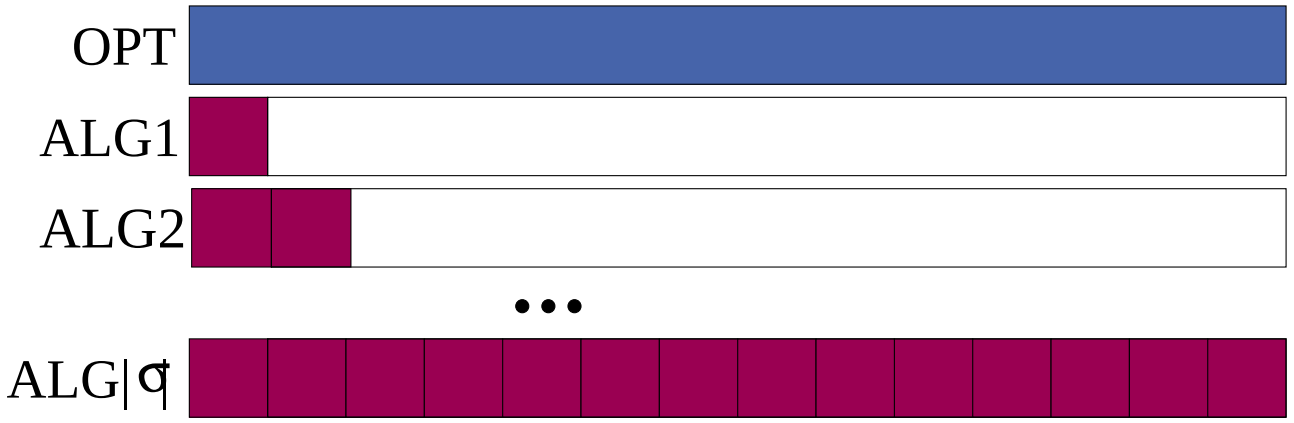
Inductive claim: given an algorithm ALG, we can create ALG_i such that

- ALG and ALG_i act identically on the first $i-1$ requests
- If request i causes a fault (for both algorithms), ALG_i evicts page with longest forward distance
- $ALG_i(\sigma) \leq ALG(\sigma)$



Using the claim

- Start with a given request sequence σ and an optimal offline algorithm ALG
- Use the claim for $i = 1$ on ALG to get ALG_1, which evicts the LFD page on the first request (if needed)
- Use the claim for $i = 2$ on ALG_1 to get ALG_2
- ...
- Final algorithm $\text{ALG}_{|\sigma|}$ is equal to OPT



s

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

$$p_1, p_2, \ldots, p_{k-1}, p_k, p_{k+1}, p_k, p_{k+1}, \cdots$$

- LFU is also not competitive: consider

$$p_1^m, p_2^m, \ldots, p_{k-1}^m, (p_k, p_{k+1})^{m-1}$$

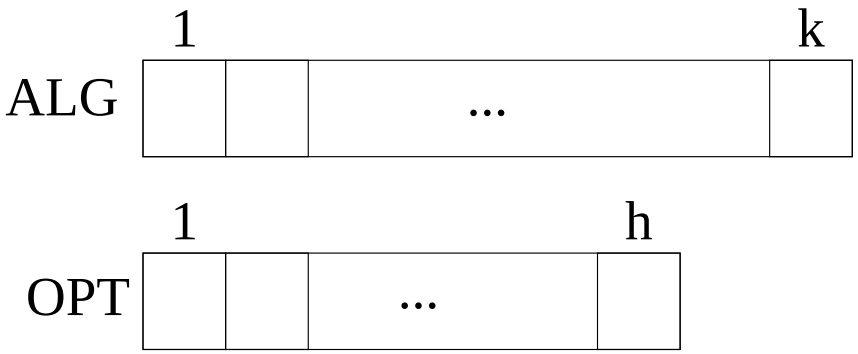
A general lower bound

- 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

  $\Rightarrow$ 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

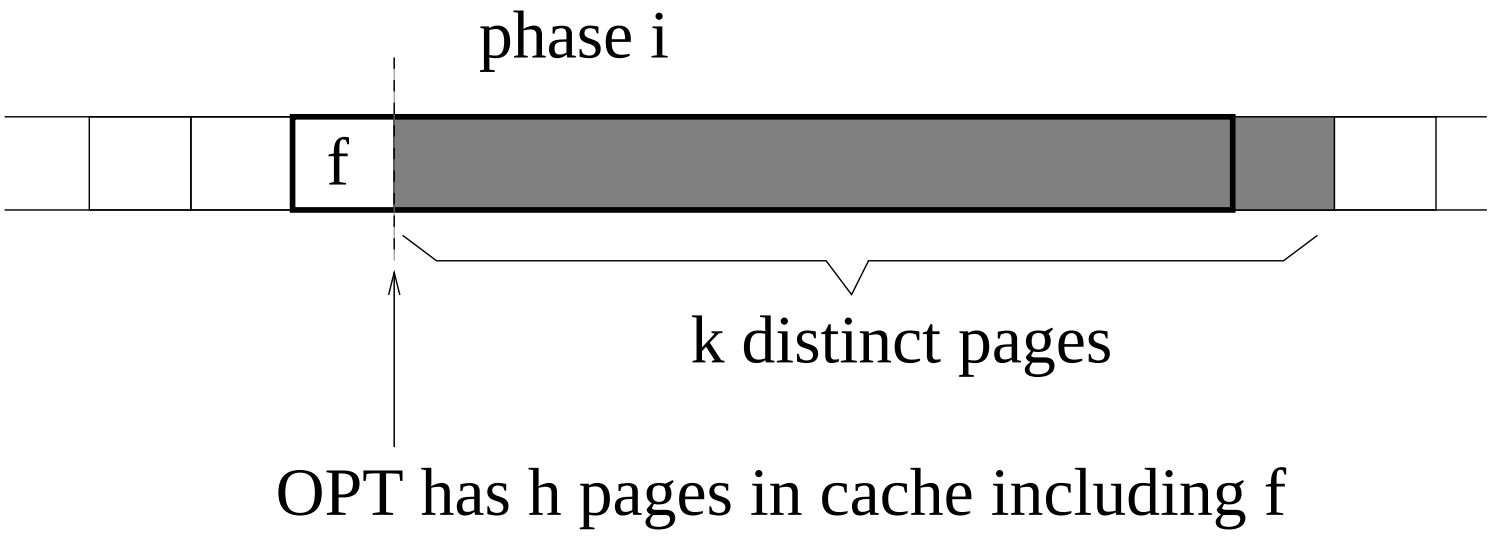
Proof: divide request sequence σ into **phases**.

- 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 choosen 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)

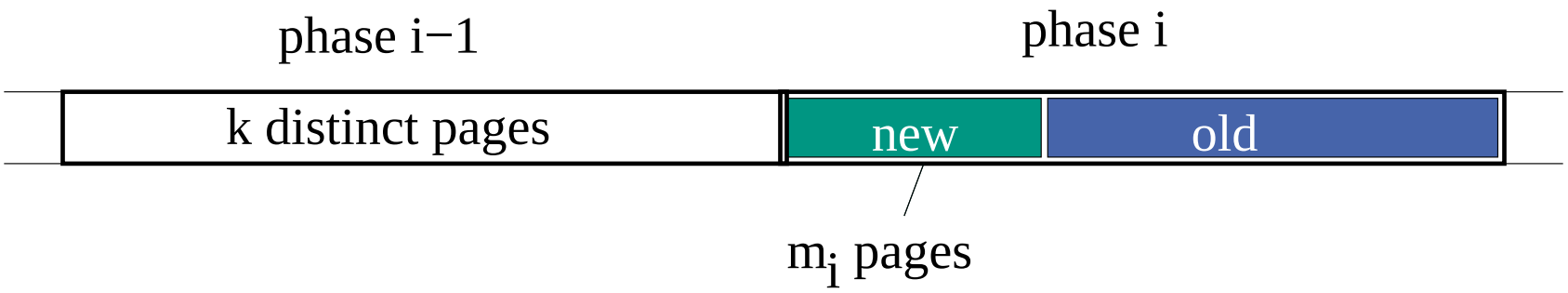
Miss probability when $j+1$st old page becomes marked

$$1 - \frac{\#\text{ old unmarked cached pages}}{\#\text{ old unmarked pages}} \leq 1 - \frac{k-m-j}{k-j} = \frac{m}{k-j}$$

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-competetive 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 competetive
- Open question (?): competetiveness of RMARK with ressource augmentation?

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