Introduction

Algorithm Theory
WS 2018/19

Fabian Kuhn

About the Course

Design and analysis techniques for algorithms

- Topics of the course:
  - divide and conquer
  - greedy
  - dynamic programming
  - advanced data structures
  - amortized analysis
  - graph algorithms
  - randomization
  - approximation algorithms
  - online algorithms
  - parallel algorithms

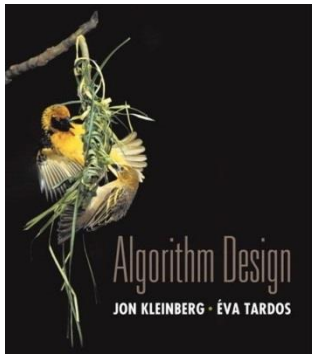
Requirements

- I assume that you have basic algorithms and data structures knowledge as well as some mathematical maturity
  - E.g., from the Bachelor course Informatik 2 and basic math courses

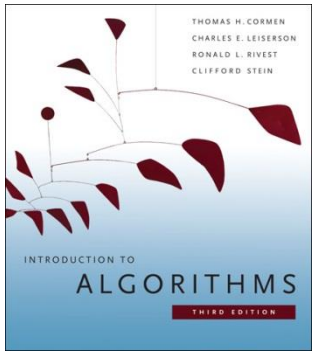
- In particular, you should be (at least partly) familiar with
  - math. induction, basic combinatorics & (discrete) probability theory, …
  - Big-O notation and Landau notation more generally
  - searching and sorting (binary search, mergesort, quicksort)
  - binary search trees, balanced binary search trees
  - priority queues (heaps)
  - hash tables
  - basic graph-theoretic definitions
  - representations of graphs
  - basic graph algorithms: traversal (depth-first, breadth-first), minimum spanning trees, shortest paths

Literature

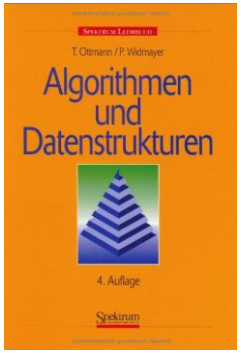
- J. Kleinberg, E. Tardos
  Algorithm Design
  Addison Wesley, 2005



- T. Cormen, C. Leiserson, R. Rivest, C. Stein
  Introduction to Algorithms, Third Edition,
  MIT Press, 2009



- T. Ottmann, P. Widmayer
  Algorithmen und Datenstrukturen
  4th Edition, Spektrum Akademischer Verlag,
  Heidelberg, 2002



- Original literature

Lecture

Lecture (101-00-026)

- Monday 14:15 – 15:45
- Thursday 10:15 – 11:45 (≈ every 2nd week)

Exercise Tutorials (101-00-026 + 051-00-034)

- Thursday 10:15 – 12:00 (English) + 16:15 – 18:00 (German)
- First exercise tutorial on Thu, Nov. 8

Language

- Lectures will be in English

General Remarks

- Theory lecture (there will be math)

Recordings

- Most lectures will be recorded
- **No guarantee that there's always a recording!**

Web Page

http://ac.informatik.uni-freiburg.de

→ Teaching → Winter Term 2018/19 → Algorithm Theory

- We will publish all important information there!
  - Check in the next days for additional information on the exercises.
- Check the web page regularly!
- Recordings will be put online
  - Sometimes possibly with some delay…

Forum

- In addition to the web page, we will also use a forum
  - The forum is provided through the Daphne system
  - You need to sign up on Daphne for this course to use the forum

- The link to the forum and for signing up will be published on the web page

- If you have a question to the lecture or the exercises, please use the forum instead of writing an email to one of us!
  - Like this, all of us and also your colleagues see the question and can answer to it
  - We can directly answer a question for everybody
  - Of course feel free to also use the forum to discuss anything related to the topics and organization of the lecture

Exercises

General Information

- There will be (theoretical) exercises to practice the material
  - We try to provide **sample solutions** (not always guaranteed)
- $\approx$ 1 problem set every 2 weeks
- You need to do the exercises either alone or in groups of 2. We encourage you to team up and do them in groups of 2!
  - send email to Philipp Schneider (philipp.schneider@cs.uni-freiburg.de)
  - write with whom you'd like to do the exercises
  - indicate whether you'd prefer to have a German tutorial group
- 50% of all exercise points are needed to be admitted to the exam

Tutorials

- Thursdays (every 2nd week, first on Nov. 8)

Exercises

Exercise groups

- 2 exercise groups (1 in English, 1 in German)
  - English group: Johannes Kalmbach (Thu, 10:15 – 12:00)
  - German group: Pascal Bachor (Thu, 16:15 – 18:00)
  - sign up by email to Philipp Schneider (see last slide)

Assistants

- Mohamad Ahmadi, Philipp Schneider
- Pascal Bachor, Johannes Kalmbach

Handing in solutions

- Solutions are always due on Mondays at 14:15 (before lecture)
- Hand in by email to your tutor or on paper
  (either in the lecture or the letter box in building 51)

Exam

Final Exam

- Final exam will take place after the semester
  - As soon as we know the date, we will publish it on the web page
- 50% of the exercise points are required to be admitted
  - Solving exercises is the best exam preparation!
- You will be allowed to bring 5 A4 pages of handwritten notes to the exam. No other material will be allowed
  - 5 A4 pages $\widehat{=}$ 5 singly-sided A4 sheets

Chapter 1
Divide and Conquer

Algorithm Theory
WS 2016/17

Fabian Kuhn

Divide-And-Conquer Principle

- Important algorithm design method

- Examples from basic alg. & data structures class (Informatik 2):
  - Sorting: Mergesort, Quicksort
  - Binary search

- Further examples
  - Median
  - Compairing orders
  - Convex hull / Delaunay triangulation / Voronoi diagram
  - Closest pairs
  - Line intersections
  - Polynomial multiplication / FFT
  - ...

Example 1: Quicksort

S	v

$S_\ell \leq v$	v	$S_r \geq v$

function Quick (S: sequence): sequence;

{returns the sorted sequence S}

begin

if $\#S \leq 1$ then **return** S

else { choose pivot element v in S;

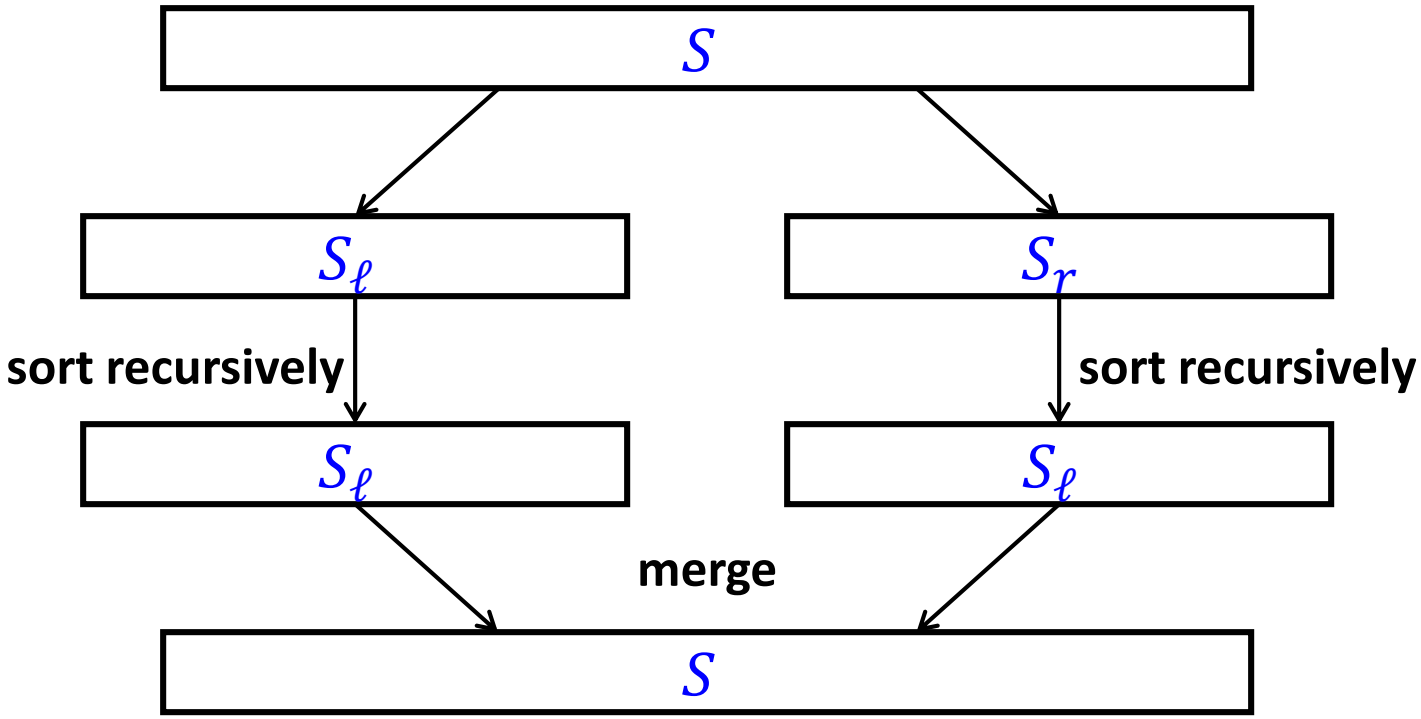
partition S into S_ℓ with elements $\geq v$,

and S_r with elements $\geq v$

return | Quick(S_ℓ) | v | Quick(S_r) |

end;

Example 2: Mergesort



Formulation of the D&C principle

Divide-and-conquer method for solving a problem instance of size n:

1. Divide

$n \leq c$: Solve the problem directly.

$n > c$: Divide the problem into k subproblems of sizes $n_1, \dots, n_k < n$ ($k \geq 2$).

2. Conquer

Solve the k subproblems in the same way (recursively).

3. Combine

Combine the partial solutions to generate a solution for the original instance.

Analysis

Recurrence relation:

- $\boldsymbol{T(n)}$: max. number of steps necessary for solving an instance of size n
- $\boldsymbol{T(n)} = \begin{cases} \boldsymbol{a} & \textbf{if } \boldsymbol{n \leq c} \\ \boldsymbol{T(n_1) + \cdots + T(n_k)} & \textbf{if } \boldsymbol{n > c} \\ \boldsymbol{+ \textbf{cost for divide and combine}} & \end{cases}$

Special case: $\boldsymbol{k = 2, n_1 = n_2 = {}^{n}/_{2}}$

- cost for divide and combine: $\mathrm{DC}(n)$
- $T(1) = a$
- $T(n) = 2T(n/2) + \mathrm{DC}(n)$

Comparing Orders

- Many web systems maintain user preferences / rankings on things like books, movies, restaurants, …

- Collaborative filtering:
  - Predict user taste by comparing rankings of different users.
  - If the system finds users with similar tastes, it can make recommendations (e.g., Amazon)

- Core issue: Compare two rankings
  - Intuitively, two rankings (of movies) are more similar, the more pairs are ordered in the same way
  - Label the first user's movies from 1 to n according to ranking
  - Order labels according to second user's ranking
  - How far is this from the ascending order (of the first user)?

Number of Inversions

Formal problem:

- **Given**: array $A = [a_1, a_2, a_3, \dots, a_n]$ of distinct elements

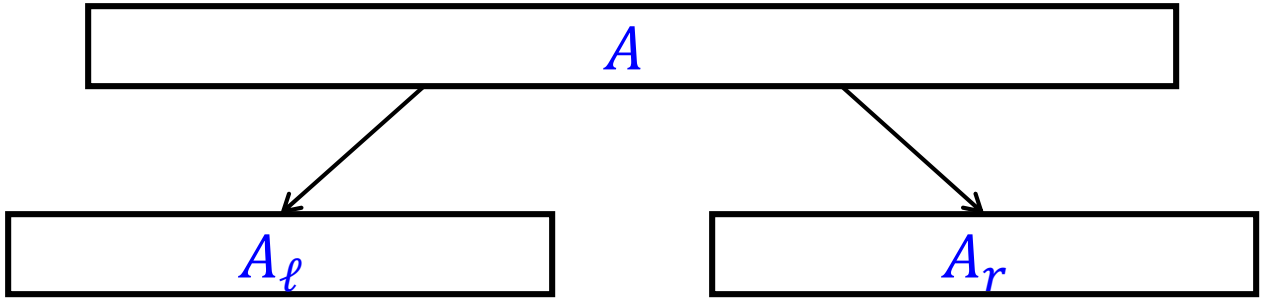
- **Objective**: Compute number of inversions I

$$I := \left|\left\{0 \leq i < j \leq n \,\middle|\, a_i > a_j)\right\}\right|$$

- **Example**: $A = [\ 4\ ,\ 1\ ,\ 5\ ,\ 2\ ,\ 7\ ,\ 10\ ,\ 6\ ]$

- **Naïve solution**:

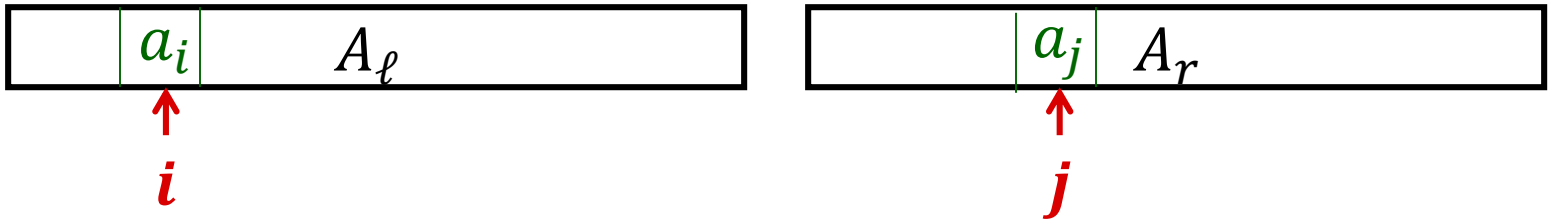
Divide and conquer



1. Divide array into 2 equal parts A_ℓ and A_r
2. Recursively compute #inversions in A_ℓ and A_r
3. Combine: add #pairs $a_i \in A_\ell$, $a_j \in A_r$ such that $a_i > a_j$

Combine Step

Assume A_ℓ and A_r are sorted



Idea:

- Maintain pointers i and j to go through the sorted parts
- While going through the sorted parts, we merge the two parts into one sorted part (like in MergeSort)

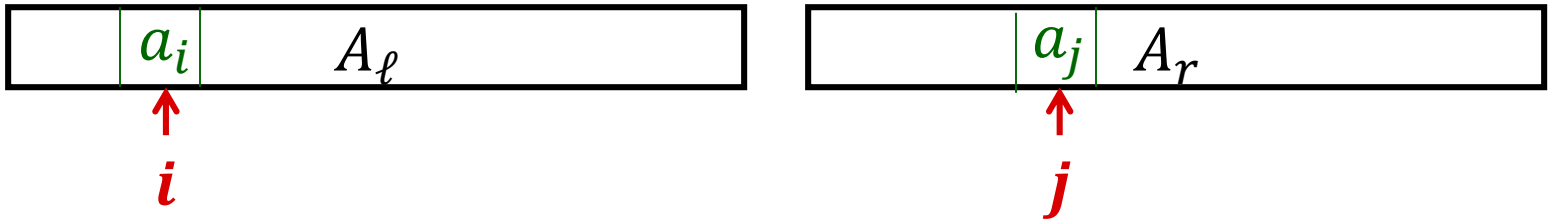
  and we count the number of inversions between the parts

Invariant:

- At each point in time, all inversions involving some element left of i (in A_ℓ) or left of j (in A_r) are counted
  - and all others still have to be counted...

Combine Step

Assume A_ℓ and A_r are sorted



- Pointers i and j, initially pointing to first elements of A_ℓ and A_r
- If $a_i < a_j$:
  - a_i is smallest among the remaining elements
  - No inversion of a_i and one of the remaining elements
  - Do not change count
- If $a_i > a_j$:
  - a_j is smallest among the remaining elements
  - a_j is smaller than all remaining elements in A_ℓ
  - Add number of remaining elements in A_ℓ to count
- Increment point, pointing to smaller element

Combine Step

- Need sub-sequences in sorted order
- Then, combine step is like merging in merge sort

- **Idea**: Solve sorting and #inversions at the same time!
  1. Partition A into two equal parts A_ℓ and A_r
  2. Recursively compute #inversions and sort A_ℓ and A_r
  3. Merge A_ℓ and A_r to sorted sequence, at the same time, compute number of inversions between elements a_i in A_ℓ and a_j in A_r

Combine Step: Example

- Assume A_ℓ and A_r are sorted

3	5	8	13	14	18	24	25	30

6	7	9	19	21	23	28	32	33

Number of Inversion: Analysis

Recurrence relation:

$$T(n) \leq 2 \cdot T(n/2) + cn, \qquad T(1) \leq a$$

Guess the solution by repeated substitution:

Number of Inversions: Analysis

Recurrence relation:

$$T(n) \leq 2 \cdot T(n/2) + cn, \qquad T(1) \leq a$$

Verify by induction:

Number of Inversions: Analysis

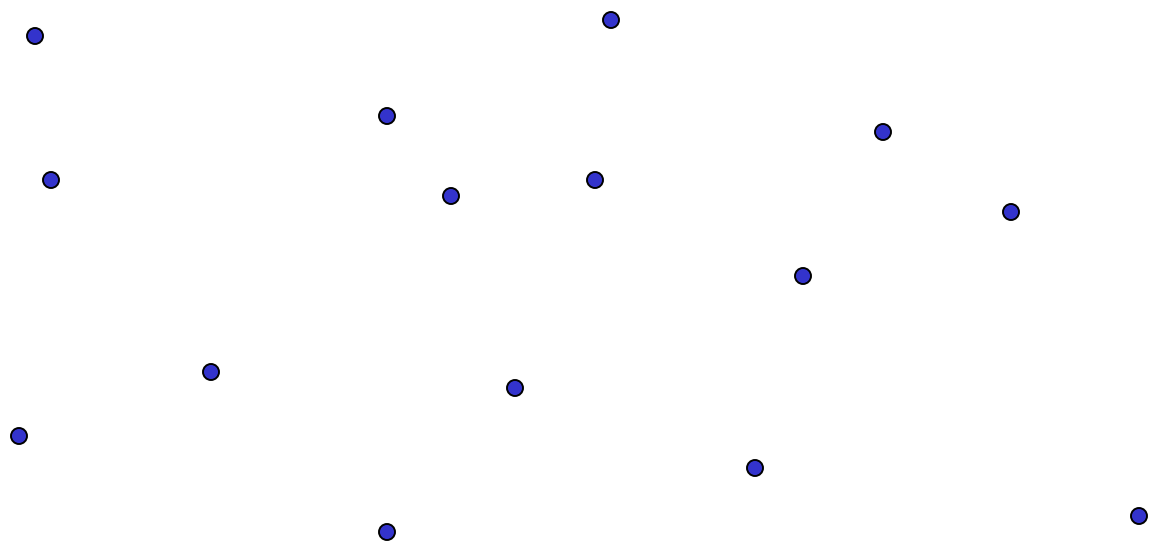
Recurrence relation:

$$T(n) \leq 2 \cdot T(n/2) + cn, \qquad T(1) \leq a$$

Guess the solution by drawing the recursion tree:

Geometric divide-and-conquer

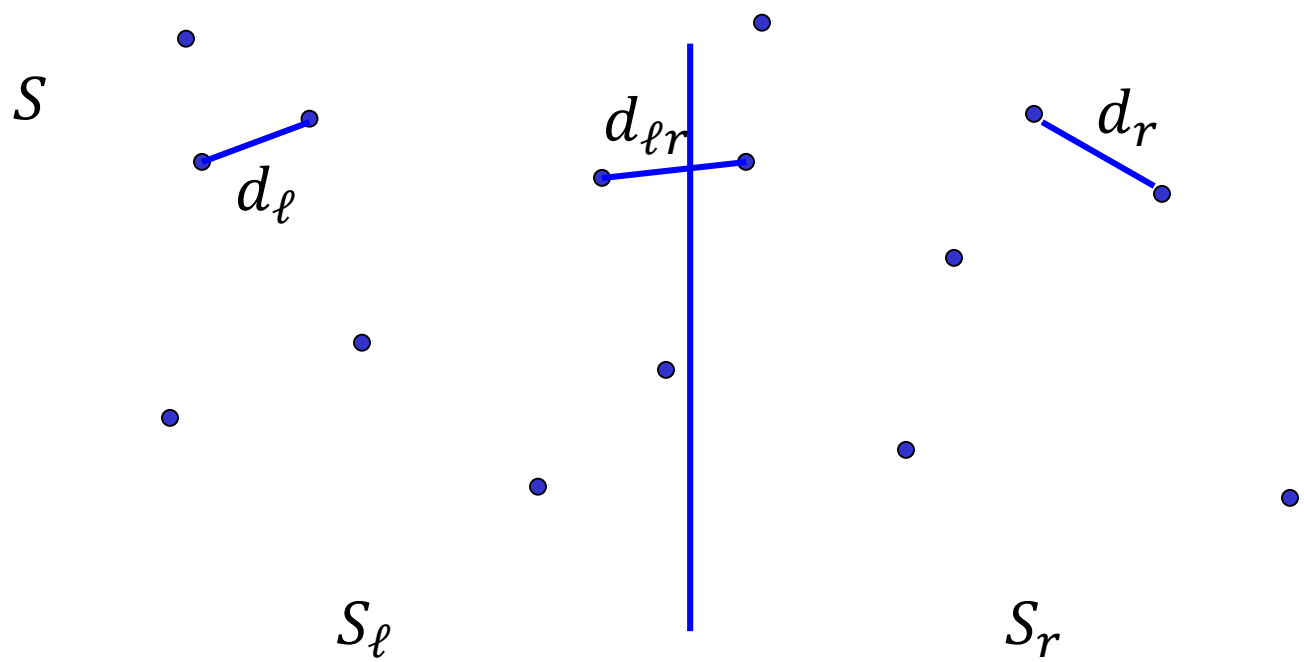
Closest Pair Problem: Given a set S of n points, find a pair of points with the smallest distance.



Naïve solution:

Divide-and-conquer solution

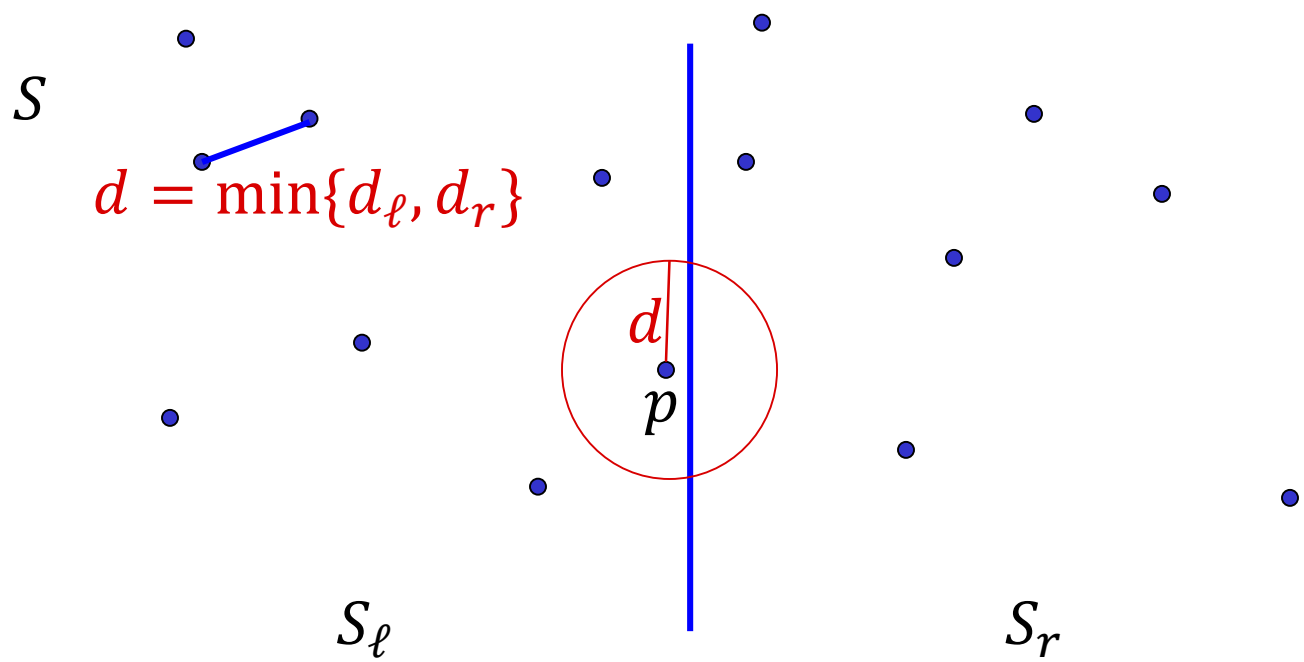
1. **Divide:** Divide S into two equal sized sets S_ℓ und S_r.
2. **Conquer:** $d_\ell = \text{mindist}(S_\ell)$ $\quad d_r = \text{mindist}(S_r)$
3. **Combine:** $d_{\ell r} = \min\{d(p_\ell, p_r) \mid p_\ell \in S_\ell, p_r \in S_r\}$
   return $\min\{d_\ell, d_r, d_{\ell r}\}$



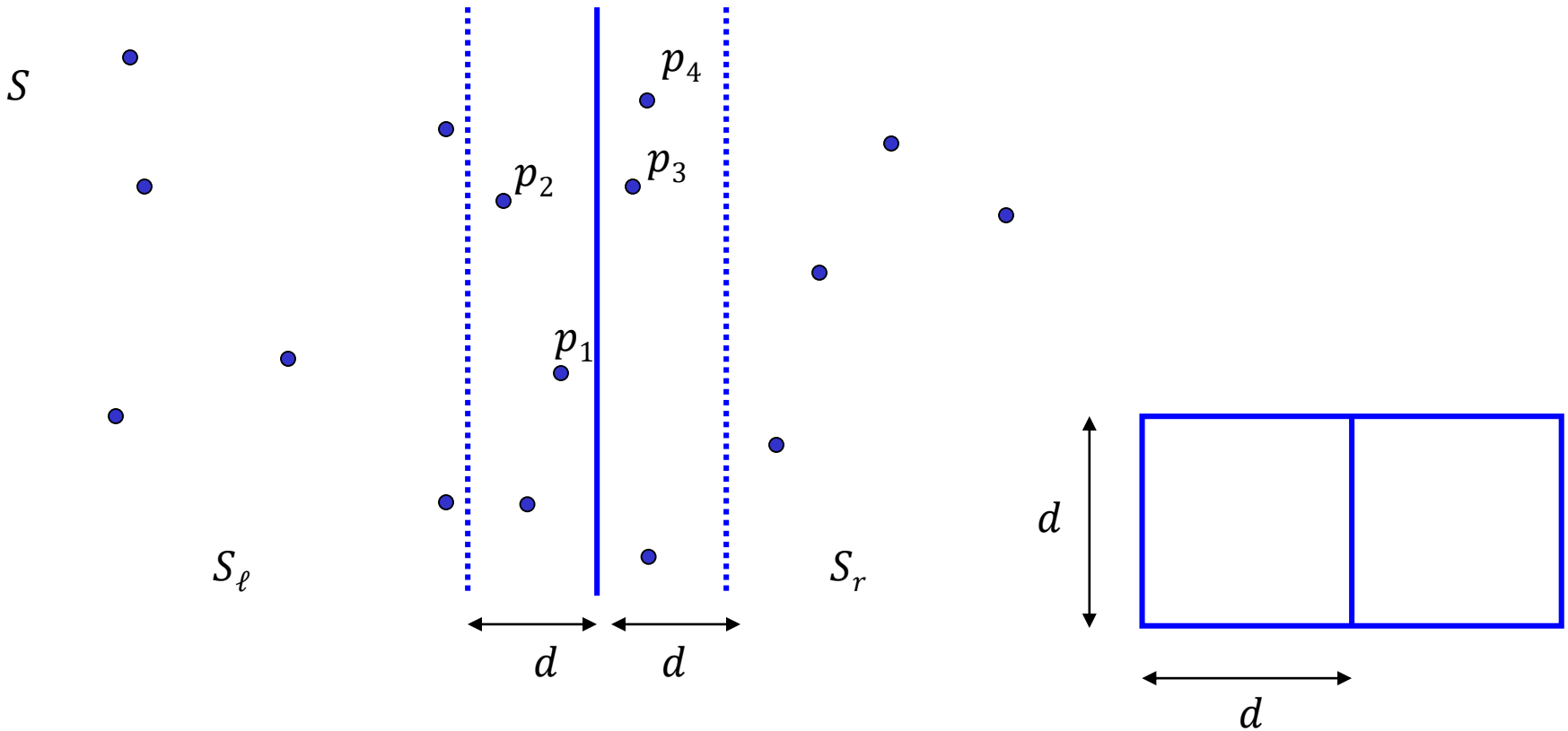
Divide-and-conquer solution

1. **Divide:** Divide S into two equal sized sets S_ℓ und S_r.
2. **Conquer:** $d_\ell = \text{mindist}(S_\ell)$ $\quad d_r = \text{mindist}(Sr)$
3. **Combine:** $d_{\ell r} = \min\{d(p_\ell, p_r) \mid p_\ell \in S_\ell, p_r \in S_r\}$
   return $\min\{d_\ell, d_r, d_{\ell r}\}$

Computation of $d_{\ell r}$:



Combine step



$$d = \min\{d_\ell, dr\}$$

Combine step

1. Consider only points within distance $\leq d$ of the bisection line, in the order of increasing y-coordinates.

2. For each point p consider all points q on the other side which are within y-distance less than d

3. There are at most 4 such points.

Implementation

- Initially sort the points in S in order of increasing x-coordinates

- While computing closest pair, also sort S according to y-coord.
  - Partition S into S_ℓ and S_r, solve and sort sub-problems recursively
  - Merge to get sorted S according to y-coordinates
  - Center points: points within x-distance $d = \min\{d_\ell, d_r\}$ of center
  - Go through center points in S in order of incr. y-coordinates

Running Time

Recurrence relation:

$$T(n) = 2 \cdot T(n/2) + c \cdot n, \qquad T(1) = a$$

Solution:

- Same as for computing number of number of inversions, merge sort (and many others…)

$$T(n) = O(n \cdot \log n)$$

Chapter 1
Divide and Conquer

Algorithm Theory
WS 2016/17

Fabian Kuhn

Formulation of the D&C principle

Divide-and-conquer method for solving a problem instance of size n:

1. Divide

$n \leq c$: Solve the problem directly.

$n > c$: Divide the problem into k subproblems of sizes $n_1, \dots, n_k < n$ ($k \geq 2$).

2. Conquer

Solve the k subproblems in the same way (recursively).

3. Combine

Combine the partial solutions to generate a solution for the original instance.

Recurrence Relations: Master Theorem

Recurrence relation

$$\boldsymbol{T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \qquad T(n) = O(1) \text{ for } n \leq n_0}$$

Cases

- $f(n) = O(n^c),\ c < \log_b a$

$$\boldsymbol{T(n) = \Theta\left(n^{\log_b a}\right)}$$

- $f(n) = \Omega(n^c),\ c > \log_b a$

$$\boldsymbol{T(n) = \Theta\big(f(n)\big)}$$

- $f(n) = \Theta\left(n^c \cdot \log^k n\right),\ c = \log_b a$

$$\boldsymbol{T(n) = \Theta\left(n^c \cdot \log^{k+1} n\right)}$$

Recurrence Relations: Master Theorem

Recurrence relation

$$\boldsymbol{T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \qquad T(n) = O(1) \text{ for } n \leq n_0}$$

Polynomials

Real polynomial p in one variable x:

$$p(x) = a_{n-1}x^{n-1} + \dots + a_1x^1 + a_0$$

Coefficients of p: $a_0, a_1, \dots, a_n \in \mathbb{R}$

Degree of p: largest power of x in p ($n-1$ in the above case)

Example:

$$p(x) = 3x^3 - 15x^2 + 18x$$

Set of all real-valued polynomials in x: $\mathbb{R}[x]$ (polynomial ring)

Operations: Evaluation

- Given: Polynomial $p \in \mathbb{R}[x]$ of degree $n-1$

$$p(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0$$

Operations: Evaluation

- Given: Polynomial $p \in \mathbb{R}[x]$ of degree $n-1$

$$p(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0$$

- **Horner's method** for evaluation at specific value x_0:

$$p(x_0) = \left(\ldots\left((a_{n-1}x_0 + a_{n-2})x_0 + a_{n-3}\right)x_0 + \cdots + a_1\right)x_0 + a_0$$

- Pseudo-code:

$p := a_{n-1}; i := n-1;$
while $(i > 0)$ **do**
    $i := i - 1;$
    $p := p \cdot x_0 + a_i$

- Running time: $O(n)$

Operations: Addition

- Given: Polynomials $p, q \in \mathbb{R}[x]$ of degree $n-1$

$$p(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0$$
$$q(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \cdots + b_1 x + b_0$$

- Compute sum $p(x) + q(x)$:

$$\begin{aligned} p(x) + q(x) \\ &= (a_{n-1}x^{n-1} + \cdots + a_0) + (b_{n-1}x^{n-1} + \cdots + b_0) \\ &= (a_{n-1} + b_{n-1})x^{n-1} + \cdots + (a_1 + b_1)x + (a_0 + b_0) \end{aligned}$$

Operations: Multiplication

- Given: Polynomials $p, q \in \mathbb{R}[x]$ of degree $n-1$

$$p(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$
$$q(x) = b_{n-1}x^{n-1} + \cdots + b_1 x + b_0$$

- Product $p(x) \cdot q(x)$:

$$\begin{aligned} p(x) \cdot q(x) &= (a_{n-1}x^{n-1} + \cdots + a_0) \cdot (b_{n-1}x^{n-1} + \cdots + b_0) \\ &= c_{2n-2}x^{2n-2} + c_{2n-3}x^{2n-3} + \cdots + c_1 x + c_0 \end{aligned}$$

- Obtaining c_i: what products of monomials have degree i?

$$\text{For } 0 \leq i \leq 2n-2\text{: } c_i = \sum_{j=0}^{i} a_j b_{i-j}$$

where $a_i = b_i = 0$ for $i \geq n$.

- Running time naïve algorithm:

Faster Multiplication?

- Multiplication is slow $\left(\Theta(n^2)\right)$ when using the standard coefficient representation

- Try divide-and-conquer to get a faster algorithm

- Assume: degree is $n-1$, n is even
- Divide polynomial $p(x) = a_{n-1}x^{n-1} + \cdots + a_0$ into 2 polynomials of degree ${}^{n}/_{2} - 1$:

$$p_0(x) = a_{{}^{n}/_{2}-1}x^{{}^{n}/_{2}-1} + \cdots + a_0$$
$$p_1(x) = a_{n-1}x^{{}^{n}/_{2}-1} + \cdots + a_{{}^{n}/_{2}}$$
$$p(x) = p_1(x) \cdot x^{{}^{n}/_{2}} + p_0(x)$$

- Similarly: $q(x) = q_1(x) \cdot x^{{}^{n}/_{2}} + q_0(x)$

Use Divide-And-Conquer

- **Divide:**
  $p(x) = p_1(x) \cdot x^{n/2} + p_0(x), \qquad q(x) = q_1(x) \cdot x^{n/2} + q_0(x)$

- **Multiplication:**
  $$p(x)q(x) = p_1(x)q_1(x) \cdot x^n + (p_0(x)q_1(x) + p_1(x)q_0(x)) \cdot x^{n/2} + p_0(x)q_0(x)$$

- 4 multiplications of degree $n/2 - 1$ polynomials:

$$T(n) = 4T\left(n/2\right) + O(n)$$

- Leads to $T(n) = \Theta(n^2)$ like the naive algorithm…
  - follows immediately by using the master theorem

More Clever Recursive Solution

- **Recall that**

$$p(x)q(x) = p_1(x)q_1(x) \cdot x^n + (p_0(x)q_1(x) + p_1(x)q_0(x)) \cdot x^{n/2} + p_0(x)q_0(x)$$

- Compute $r(x) = \big(p_0(x) + p_1(x)\big) \cdot \big(q_0(x) + q_1(x)\big)$:

Karatsuba Algorithm

- Recursive multiplication:

$$r(x) = \big(p_0(x) + p_1(x)\big) \cdot \big(q_0(x) + q_1(x)\big)$$

$$\begin{aligned} p(x)q(x) = {} & p_1(x)q_1(x) \cdot x^n \\ & + \big(r(x) - p_0(x)q_0(x) + p_1(x)q_1(x)\big) \cdot x^{n/2} \\ & + p_0(x)q_0(x) \end{aligned}$$

- Recursively do 3 multiplications of degr. $(n/2 - 1)$-polynomials

$$T(n) = 3T\left(n/2\right) + O(n)$$

- Gives: $T(n) = O(n^{1.58496\ldots})$ (see Master theorem)

Representation of Polynomials

Coefficient representation:

- Polynomial $p(x) \in \mathbb{R}[x]$ of degree $n-1$ is given by its n coefficients $a_0, \dots, a_{n-1}$:

$$p(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

- Coefficient vector $\boldsymbol{a} = (a_0, a_1, \dots a_{n-1})$

- Example:

$$p(x) = 3x^3 - 15x^2 + 18x$$

- The most typical (and probably most natural) representation of polynomials

Representation of Polynomials

Product of linear factors:

- Polynomial $p(x) \in \mathbb{C}[x]$ of degr. $n-1$ is given by its $n-1$ roots

$$p(x) = a_{n-1} \cdot (x - x_1) \cdot (x - x_2) \cdot \ldots \cdot (x - x_{n-1})$$

- Example:

$$p(x) = 3x(x-2)(x-3)$$

- Every polynomial has exactly $n-1$ roots $x_i \in \mathbb{C}$ (s.t. $p(x_i) = 0$)
  - Polynomial is uniquely defined by the $n-1$ roots and a_{n-1}

- We will not use this representation…

Representation of Polynomials

Point-value representation:

- Polynomial $p(x) \in \mathbb{R}[x]$ of degree $n-1$ is given by n point-value pairs:

$$p = \{(x_0, p(x_0)), (x_1, p(x_1)), \dots, (x_{n-1}, p(x_{n-1}))\}$$

  where $x_i \neq x_j$ for $i \neq j$.

- Example: The polynomial

$$p(x) = 3x(x-2)(x-3)$$

  is uniquely defined by the four point-value pairs $(0,0), (1,6), (2,0), (3,0)$.

Operations: Coefficient Representation

$$\boldsymbol{p(x) = a_{n-1}x^{n-1} + \cdots + a_0}, \qquad \boldsymbol{q(x) = b_{n-1}x^{n-1} + \cdots + b_0}$$

Evaluation: Horner's method: Time $O(n)$

Addition:

$$p(x) + q(x) = (a_{n-1} + b_{n-1})x^{n-1} + \cdots + (a_0 + b_0)$$

- Time: $O(n)$

Multiplication:

$$p(x) \cdot q(x) = c_{2n-2}x^{2n-2} + \cdots + c_0, \qquad \text{where } c_i = \sum_{j=0}^{i} a_j b_{i-j}$$

- Naive solution: Need to compute product $a_i b_j$ for all $0 \leq i, j \leq n$
- Time: $O(n^2)$

Operations: Linear Factors (Roots)

$$\boldsymbol{p(x) = a_{n-1} \cdot (x - x_1) \cdot \ldots \cdot (x - x_{n-1})}$$
$$\boldsymbol{q(x) = b_{n-1} \cdot (x - x_1) \cdot \ldots \cdot (x - x_{n-1})}$$

Evaluation:

- Just plug in the value where the poly. is evaluated: Time $O(n)$

Multiplication:

- Concatenate the two representations: Time $O(n)$

Addition:

- Need to find the roots of $p(x) + q(x)$
- For polynomials of degree > 4, this is not possible by using basic arithmetic operations $(+, -, \cdot, /, \sqrt[a]{b})$
- In the usual computational model impossible
  - Numerically, the roots can be computed to arbitrary precision

Operations: Point-Value Representation

$$\boldsymbol{p = \{(x_0, p(x_0)), \dots, (x_{n-1}, p(x_{n-1}))\}}$$
$$\boldsymbol{q = \{(x_0, q(x_0)), \dots, (x_{n-1}, q(x_{n-1}))\}}$$

- Note: we use the **same points $\boldsymbol{x_0, \dots, x_n}$** for both polynomials

Addition:

$$p + q = \{(x_0, p(x_0) + q(x_0)), \dots, (x_{n-1}, p(x_{n-1}) + q(x_{n-1}))\}$$

- Time: $O(n)$

Multiplication:

$$p \cdot q = \{(x_0, p(x_0) \cdot q(x_0)), \dots, (x_{2n-2}, p(x_{2n-2}) \cdot q(x_{2n-2}))\}$$

- Time: $O(n)$

Evaluation: Polynomial interpolation can be done in $O(n^2)$

Operations on Polynomials

Cost depending on representation:

	Coefficient	Roots	Point-Value
Evaluation	$O(n)$	$O(n)$	$O(n^2)$
Addition	$O(n)$	∞	$O(n)$
Multiplication	$O(n^{1.58})$	$O(n)$	$O(n)$

Faster Polynomial Multiplication?

Multiplication is fast when using the point-value representation

Idea to compute $p(x) \cdot q(x)$ (for polynomials of degree $< n$):

p, q of degree $n-1$, n coefficients

↓ **Evaluation** at points $x_0, x_1, \dots, x_{2n-2}$

$2 \times 2n$ point-value pairs $\big(x_i, p(x_i)\big)$ and $\big(x_i, q(x_i)\big)$

↓ **Point-wise multiplication**

$2n$ point-value pairs $\big(x_i, p(x_i)q(x_i)\big)$

↓ **Interpolation**

$p(x)q(x)$ of degree $2n-2$, $2n-1$ coefficients

Coefficients to Point-Value Representation

Given: Polynomial $p(x)$ by the coefficient vector $(a_0, a_1, \dots, a_{N-1})$

Goal: Compute $p(x)$ for all x in a given set X

- Where X is of size $|X| = N$
- Assume that N is a power of 2

Divide and Conquer Approach

- Divide $p(x)$ of degree $N-1$ (N is even) into 2 polynomials of degree ${}^{N}/_{2} - 1$ differently than in Karatsuba's algorithm

- $p_0(y) = a_0 + a_2 y + a_4 y^2 + \cdots + a_{N-2} y^{N/2-1}$ (even coeff.)
  $p_1(y) = a_1 + a_3 y + a_5 y^2 + \cdots + a_{N-1} y^{N/2-1}$ (odd coeff.)

Coefficients to Point-Value Representation

Goal: Compute $p(x)$ for all x in a given set X of size $|X| = N$

- Divide $p(x)$ of degr. $N-1$ into 2 polynomials of degr. ${}^{N}/_{2} - 1$

$$p_0(y) = a_0 + a_2 y + a_4 y^2 + \cdots + a_{N-2} y^{N/2-1} \quad \text{(even coeff.)}$$
$$p_1(y) = a_1 + a_3 y + a_5 y^2 + \cdots + a_{N-1} y^{N/2-1} \quad \text{(odd coeff.)}$$

Let's first look at the "combine" step:

- We need to compute $p(x)$ for all $x \in X$ after recursive calls for polynomials p_0 and p_1:

Coefficients to Point-Value Representation

Goal: Compute $p(x)$ for all x in a given set X of size $|X| = N$

- Divide $p(x)$ of degr. $N-1$ into 2 polynomials of degr. ${}^{N}/_{2} - 1$

$$p_0(y) = a_0 + a_2 y + a_4 y^2 + \cdots + a_{N-2} y^{N/2-1} \quad \text{(even coeff.)}$$
$$p_1(y) = a_1 + a_3 y + a_5 y^2 + \cdots + a_{N-1} y^{N/2-1} \quad \text{(odd coeff.)}$$

Let's first look at the "combine" step:

$$\forall x \in X: \quad p(x) = p_0(x^2) + x \cdot p_1(x^2)$$

- Recursively compute $p_0(y)$ and $p_1(y)$ for all $y \in X^2$
  - Where $X^2 := \{x^2 : x \in X\}$

- Generally, we have $|X^2| = |X|$

Analysis

Recurrence formula for the given algorithm:

Faster Algorithm?

- In order to have a faster algorithm, we need $|X^2| < |X|$

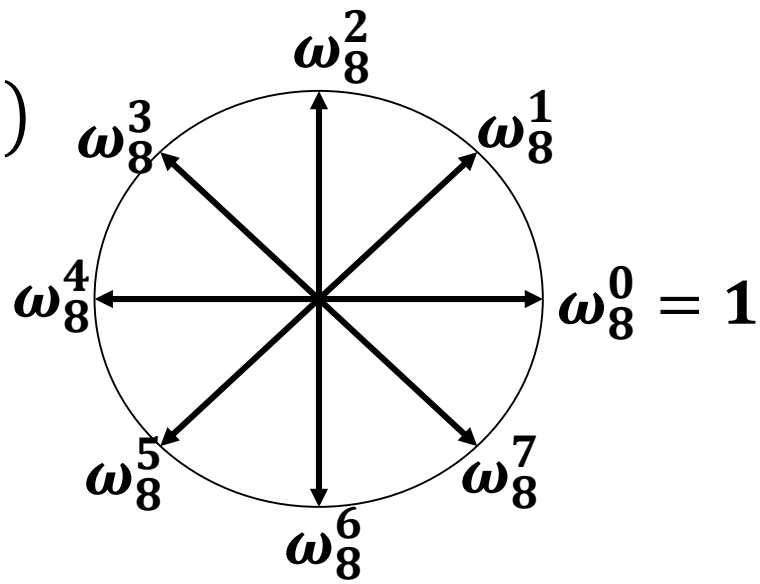
Choice of X

- Select points $x_0, x_1, \dots, x_{N-1}$ to evaluate p and q in a clever way

Consider the N complex roots of unity:

Principle root of unity: $\omega_N = e^{2\pi i/N}$

$$\left(i = \sqrt{-1}, \qquad e^{2\pi i} = 1\right)$$



Powers of ω_n (roots of unity):

$$1 = \omega_N^0, \omega_N^1, \dots, \omega_N^{N-1}$$

Note: $\omega_N^k = e^{2\pi i k/N} = \cos\frac{2\pi k}{N} + i \cdot \sin\frac{2\pi k}{N}$

Properties of the Roots of Unity

- **Cancellation Lemma:**

  For all integers $n > 0$, $k \geq 0$, and $d > 0$, we have:

$$\boldsymbol{\omega_{dn}^{dk} = \omega_n^k}\,, \qquad \boldsymbol{\omega_n^{k+n} = \omega_n^k}$$

- **Proof:**

Chapter 1
Divide and Conquer

Algorithm Theory
WS 2018/19

Fabian Kuhn

Divide-And-Conquer Principle

- Important algorithm design method

- Examples from basic alg. & data structures class (Informatik 2):
  - Sorting: Mergesort, Quicksort
  - Binary search

- Further examples
  - Median
  - Compairing orders
  - Convex hull / Delaunay triangulation / Voronoi diagram
  - Closest pairs
  - Line intersections
  - Polynomial multiplication / FFT
  - ...

Formulation of the D&C principle

Divide-and-conquer method for solving a problem instance of size n:

1. Divide

$n \leq c$: Solve the problem directly.

$n > c$: Divide the problem into k subproblems of sizes $n_1, \dots, n_k < n$ ($k \geq 2$).

2. Conquer

Solve the k subproblems in the same way (recursively).

3. Combine

Combine the partial solutions to generate a solution for the original instance.

Running Time

Recurrence relation:

$$T(n) = 2 \cdot T(n/2) + c \cdot n, \qquad T(1) = a$$

Solution:

- Same as for computing number of number of inversions, merge sort (and many others…)

$$T(n) = O(n \cdot \log n)$$

Recurrence Relations: Master Theorem

Recurrence relation

$$\boldsymbol{T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \qquad T(n) = O(1) \text{ for } n \leq n_0}$$

Cases

- $f(n) = O(n^c),\ c < \log_b a$

$$\boldsymbol{T(n) = \Theta\left(n^{\log_b a}\right)}$$

- $f(n) = \Omega(n^c),\ c > \log_b a$

$$\boldsymbol{T(n) = \Theta\big(f(n)\big)}$$

- $f(n) = \Theta\left(n^c \cdot \log^k n\right),\ c = \log_b a$

$$\boldsymbol{T(n) = \Theta\left(n^c \cdot \log^{k+1} n\right)}$$

Polynomials

Real polynomial p in one variable x:

$$p(x) = a_{n-1}x^{n-1} + \dots + a_1x^1 + a_0$$

Coefficients of p: $a_0, a_1, \dots, a_n \in \mathbb{R}$

Degree of p: largest power of x in p ($n-1$ in the above case)

Example:

$$p(x) = 3x^3 - 15x^2 + 18x$$

Set of all real-valued polynomials in x: $\mathbb{R}[x]$ (polynomial ring)

Divide-&-Conquer Polynomial Multiplication

- Multiplication is slow $\left(\Theta(n^2)\right)$ when using the standard coefficient representation

- Try divide-and-conquer to get a faster algorithm

- Assume: degree is $n-1$, n is even
- Divide polynomial $p(x) = a_{n-1}x^{n-1} + \cdots + a_0$ into 2 polynomials of degree ${}^{n}/_{2} - 1$:

$$p_0(x) = a_{{}^{n}/_{2}-1}x^{{}^{n}/_{2}-1} + \cdots + a_0$$
$$p_1(x) = a_{n-1}x^{{}^{n}/_{2}-1} + \cdots + a_{{}^{n}/_{2}}$$
$$p(x) = p_1(x) \cdot x^{{}^{n}/_{2}} + p_0(x)$$

- Similarly: $q(x) = q_1(x) \cdot x^{{}^{n}/_{2}} + q_0(x)$

Divide-&-Conquer Polynomial Multiplication

- **Divide:**
  $p(x) = p_1(x) \cdot x^{n/2} + p_0(x), \qquad q(x) = q_1(x) \cdot x^{n/2} + q_0(x)$

- **Multiplication:**
  $$p(x)q(x) = p_1(x)q_1(x) \cdot x^n + (p_0(x)q_1(x) + p_1(x)q_0(x)) \cdot x^{n/2} + p_0(x)q_0(x)$$

- 4 multiplications of degree $n/2 - 1$ polynomials:

$$T(n) = 4T\left(n/2\right) + O(n)$$

- Leads to $T(n) = \Theta(n^2)$ like the naive algorithm…
  - follows immediately by using the master theorem

Karatsuba Algorithm

- Recursive multiplication:

$$r(x) = \big(p_0(x) + p_1(x)\big) \cdot \big(q_0(x) + q_1(x)\big)$$

$$\begin{aligned} p(x)q(x) = {} & p_1(x)q_1(x) \cdot x^n \\ & + \big(r(x) - p_0(x)q_0(x) + p_1(x)q_1(x)\big) \cdot x^{n/2} \\ & + p_0(x)q_0(x) \end{aligned}$$

- Recursively do 3 multiplications of degr. $(n/2 - 1)$-polynomials

$$T(n) = 3T\left(n/2\right) + O(n)$$

- Gives: $T(n) = O(n^{1.58496\ldots})$ (see Master theorem)

Representation of Polynomials

Coefficient representation:

- Polynomial $p(x) \in \mathbb{R}[x]$ of degree $n-1$ is given by its n coefficients $a_0, \dots, a_{n-1}$:

$$p(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

- Coefficient vector $\boldsymbol{a} = (a_0, a_1, \dots a_{n-1})$

- Example:

$$p(x) = 3x^3 - 15x^2 + 18x$$

- The most typical (and probably most natural) representation of polynomials

Representation of Polynomials

Point-value representation:

- Polynomial $p(x) \in \mathbb{R}[x]$ of degree $n-1$ is given by n point-value pairs:

$$p = \{(x_0, p(x_0)), (x_1, p(x_1)), \dots, (x_{n-1}, p(x_{n-1}))\}$$

  where $x_i \neq x_j$ for $i \neq j$.

- Example: The polynomial

$$p(x) = 3x(x-2)(x-3)$$

  is uniquely defined by the four point-value pairs $(0,0), (1,6), (2,0), (3,0)$.

Operations: Coefficient Representation

$$\boldsymbol{p(x) = a_{n-1}x^{n-1} + \cdots + a_0}, \qquad \boldsymbol{q(x) = b_{n-1}x^{n-1} + \cdots + b_0}$$

Evaluation: Horner's method: Time $O(n)$

Addition:

$$p(x) + q(x) = (a_{n-1} + b_{n-1})x^{n-1} + \cdots + (a_0 + b_0)$$

- Time: $O(n)$

Multiplication:

$$p(x) \cdot q(x) = c_{2n-2}x^{2n-2} + \cdots + c_0, \qquad \text{where } c_i = \sum_{j=0}^{i} a_j b_{i-j}$$

- Naive solution: Need to compute product $a_i b_j$ for all $0 \leq i, j \leq n$
- Time: $O(n^2)$

Operations: Point-Value Representation

$$\boldsymbol{p = \{(x_0, p(x_0)), \dots, (x_{n-1}, p(x_{n-1}))\}}$$
$$\boldsymbol{q = \{(x_0, q(x_0)), \dots, (x_{n-1}, q(x_{n-1}))\}}$$

- Note: we use the **same points $\boldsymbol{x_0, \dots, x_n}$** for both polynomials

Addition:

$$p + q = \{(x_0, p(x_0) + q(x_0)), \dots, (x_{n-1}, p(x_{n-1}) + q(x_{n-1}))\}$$

- Time: $O(n)$

Multiplication:

$$p \cdot q = \{(x_0, p(x_0) \cdot q(x_0)), \dots, (x_{2n-2}, p(x_{2n-2}) \cdot q(x_{2n-2}))\}$$

- Time: $O(n)$

Evaluation: Polynomial interpolation can be done in $O(n^2)$

Operations on Polynomials

Cost depending on representation:

	Coefficient	**Roots**	**Point-Value**
Evaluation	$O(n)$	$O(n)$	$O(n^2)$
Addition	$O(n)$	∞	$O(n)$
Multiplication	$O(n^{1.58})$	$O(n)$	$O(n)$

Faster Polynomial Multiplication?

Multiplication is fast when using the point-value representation

Idea to compute $p(x) \cdot q(x)$ (for polynomials of degree $< n$):

p, q of degree $n-1$, n coefficients

↓ **Evaluation** at points $x_0, x_1, \dots, x_{2n-2}$

$2 \times 2n$ point-value pairs $\big(x_i, p(x_i)\big)$ and $\big(x_i, q(x_i)\big)$

↓ **Point-wise multiplication**

$2n$ point-value pairs $\big(x_i, p(x_i)q(x_i)\big)$

↓ **Interpolation**

$p(x)q(x)$ of degree $2n-2$, $2n-1$ coefficients

Coefficients to Point-Value Representation

Given: Polynomial $p(x)$ by the coefficient vector $(a_0, a_1, \dots, a_{N-1})$

Goal: Compute $p(x)$ for all x in a given set X

- Where X is of size $|X| = N$
- Assume that N is a power of 2

Divide and Conquer Approach

- Divide $p(x)$ of degree $N-1$ (N is even) into 2 polynomials of degree ${}^{N}/_{2} - 1$ differently than in Karatsuba's algorithm

- $p_0(y) = a_0 + a_2 y + a_4 y^2 + \cdots + a_{N-2} y^{N/2-1}$ (even coeff.)
  $p_1(y) = a_1 + a_3 y + a_5 y^2 + \cdots + a_{N-1} y^{N/2-1}$ (odd coeff.)

Coefficients to Point-Value Representation

Goal: Compute $p(x)$ for all x in a given set X of size $|X| = N$

- Divide $p(x)$ of degr. $N-1$ into 2 polynomials of degr. ${}^{N}/_{2} - 1$

$$p_0(y) = a_0 + a_2 y + a_4 y^2 + \cdots + a_{N-2} y^{N/2-1} \quad \text{(even coeff.)}$$
$$p_1(y) = a_1 + a_3 y + a_5 y^2 + \cdots + a_{N-1} y^{N/2-1} \quad \text{(odd coeff.)}$$

Let's first look at the "combine" step:

$$\forall x \in X: \quad p(x) = p_0(x^2) + x \cdot p_1(x^2)$$

- Recursively compute $p_0(y)$ and $p_1(y)$ for all $y \in X^2$
  - Where $X^2 := \{x^2 : x \in X\}$

- Generally, we have $|X^2| = |X|$

Analysis

Recurrence formula for the given algorithm:

Faster Algorithm?

- In order to have a faster algorithm, we need $|X^2| < |X|$

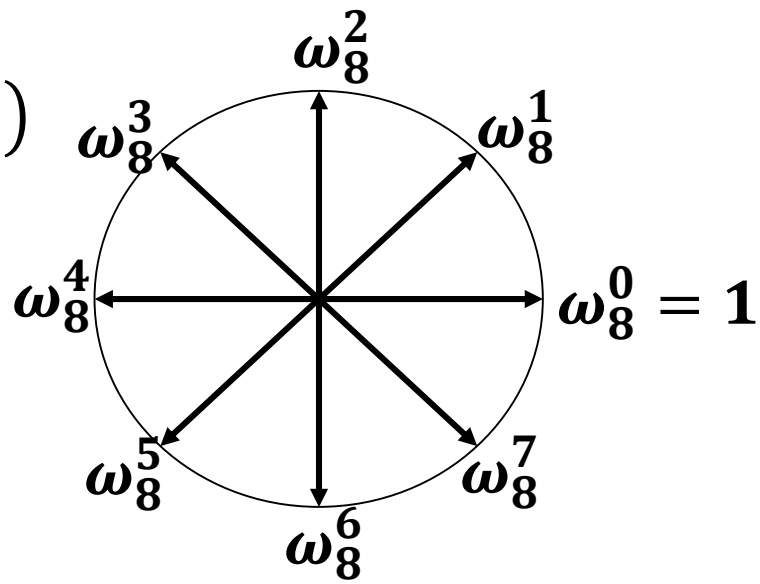
Choice of X

- Select points $x_0, x_1, \dots, x_{N-1}$ to evaluate p and q in a clever way

Consider the N complex roots of unity:

Principle root of unity: $\omega_N = e^{2\pi i/N}$

$$\left(i = \sqrt{-1}, \qquad e^{2\pi i} = 1\right)$$



Powers of ω_n (roots of unity):

$$1 = \omega_N^0, \omega_N^1, \dots, \omega_N^{N-1}$$

Note: $\omega_N^k = e^{2\pi i k/N} = \cos\frac{2\pi k}{N} + i \cdot \sin\frac{2\pi k}{N}$

Properties of the Roots of Unity

- **Cancellation Lemma:**

  For all integers $n > 0$, $k \geq 0$, and $d > 0$, we have:

$$\boldsymbol{\omega_{dn}^{dk} = \omega_n^k}\,, \qquad \boldsymbol{\omega_n^{k+n} = \omega_n^k}$$

- **Proof:**

Properties of the Roots of Unity

Claim: If $X = \left\{\omega_{2k}^{i} : i \in \{0, \dots, 2k-1\}\right\}$, we have

$$\boldsymbol{X^2 = \left\{\omega_k^i : i \in \{0, \dots, k-1\}\right\}, \qquad |X^2| = \frac{|X|}{2}}$$

Analysis

New recurrence formula:

$$T(N, |X|) \leq 2 \cdot T\left({}^{N}/_{2}, {}^{|X|}/_{2}\right) + O(N + |X|)$$

Chapter 1
Divide and Conquer

Algorithm Theory
WS 2018/19

Fabian Kuhn

Polynomials

Real polynomial p in one variable x:

$$p(x) = a_{n-1}x^{n-1} + \dots + a_1x^1 + a_0$$

Coefficients of p: $a_0, a_1, \dots, a_n \in \mathbb{R}$

Degree of p: largest power of x in p ($n-1$ in the above case)

Example:

$$p(x) = 3x^3 - 15x^2 + 18x$$

Set of all real-valued polynomials in x: $\mathbb{R}[x]$ (polynomial ring)

Operations on Polynomials

Cost depending on representation:

	Coefficient	**Roots**	**Point-Value**
Evaluation	$O(n)$	$O(n)$	$O(n^2)$
Addition	$O(n)$	∞	$O(n)$
Multiplication	$O(n^{1.58})$	$O(n)$	$O(n)$

Faster Polynomial Multiplication?

Multiplication is fast when using the point-value representation

Idea to compute $p(x) \cdot q(x)$ (for polynomials of degree $< n$):

p, q of degree $n-1$, n coefficients

↓ **Evaluation** at points $x_0, x_1, \dots, x_{2n-2}$

$2 \times 2n$ point-value pairs $\big(x_i, p(x_i)\big)$ and $\big(x_i, q(x_i)\big)$

↓ **Point-wise multiplication**

$2n$ point-value pairs $\big(x_i, p(x_i)q(x_i)\big)$

↓ **Interpolation**

$p(x)q(x)$ of degree $2n-2$, $2n-1$ coefficients

Coefficients to Point-Value Representation

Given: Polynomial $p(x)$ by the coefficient vector $(a_0, a_1, \dots, a_{N-1})$

Goal: Compute $p(x)$ for all x in a given set X

- Where X is of size $|X| = N$
- Assume that N is a power of 2

Divide and Conquer Approach

- Divide $p(x)$ of degree $N-1$ (N is even) into 2 polynomials of degree ${}^{N}/_{2} - 1$ differently than in Karatsuba's algorithm

- $p_0(y) = a_0 + a_2 y + a_4 y^2 + \cdots + a_{N-2} y^{N/2-1}$ (even coeff.)
  $p_1(y) = a_1 + a_3 y + a_5 y^2 + \cdots + a_{N-1} y^{N/2-1}$ (odd coeff.)

Coefficients to Point-Value Representation

Goal: Compute $p(x)$ for all x in a given set X of size $|X| = N$

- Divide $p(x)$ of degr. $N-1$ into 2 polynomials of degr. ${}^{N}/_{2} - 1$

$$p_0(y) = a_0 + a_2 y + a_4 y^2 + \cdots + a_{N-2} y^{N/2-1} \quad \text{(even coeff.)}$$
$$p_1(y) = a_1 + a_3 y + a_5 y^2 + \cdots + a_{N-1} y^{N/2-1} \quad \text{(odd coeff.)}$$

Let's first look at the "combine" step:

$$\forall x \in X: \quad p(x) = p_0(x^2) + x \cdot p_1(x^2)$$

- Recursively compute $p_0(y)$ and $p_1(y)$ for all $y \in X^2$
  - Where $X^2 := \{x^2 : x \in X\}$

- Generally, we have $|X^2| = |X|$

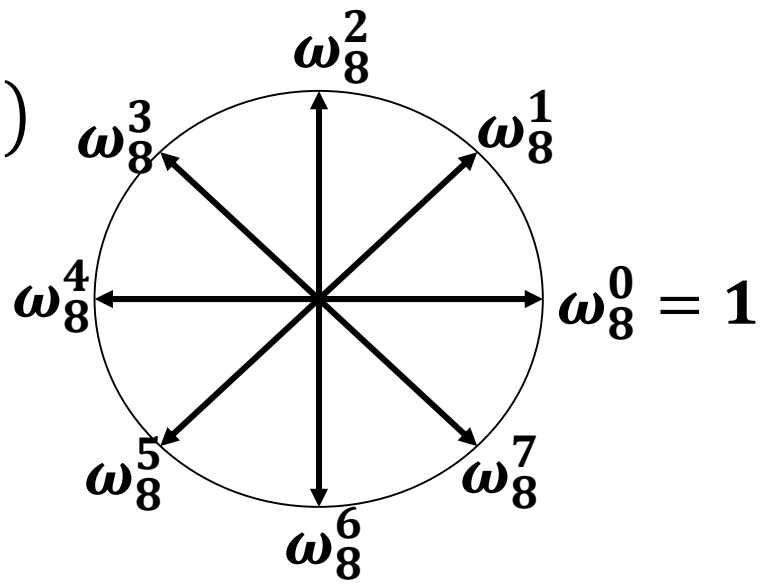
Choice of X

- Select points $x_0, x_1, \dots, x_{N-1}$ to evaluate p and q in a clever way

Consider the N complex roots of unity:

Principle root of unity: $\omega_N = e^{2\pi i/N}$

$$\left(i = \sqrt{-1}, \qquad e^{2\pi i} = 1\right)$$



Powers of ω_n (roots of unity):

$$1 = \omega_N^0, \omega_N^1, \dots, \omega_N^{N-1}$$

Note: $\omega_N^k = e^{2\pi i k/N} = \cos\frac{2\pi k}{N} + i \cdot \sin\frac{2\pi k}{N}$

Properties of the Roots of Unity

Claim: If $X = \left\{\omega_{2k}^{i} : i \in \{0, \dots, 2k-1\}\right\}$, we have

$$\boldsymbol{X^2 = \left\{\omega_k^i : i \in \{0, \dots, k-1\}\right\}, \qquad |X^2| = \frac{|X|}{2}}$$

Analysis

New recurrence formula:

$$T(N, |X|) \leq 2 \cdot T\left({}^{N}/_{2}, {}^{|X|}/_{2}\right) + O(N + |X|)$$

Faster Polynomial Multiplication?

Idea to compute $p(x) \cdot q(x)$ (for polynomials of degree $< n$):

p, q of degree $n-1$, n coefficients

↓ **Evaluation** at points $\omega_{2n}^0, \omega_{2n}^1, \dots, \omega_{2n}^{2n-1}$

$2 \times 2n$ point-value pairs $\left(\omega_{2n}^k, p\left(\omega_{2n}^k\right)\right)$ and $\left(\omega_{2n}^k, q\left(\omega_{2n}^k\right)\right)$

↓ **Point-wise multiplication**

$2n$ point-value pairs $\left(\omega_{2n}^k, p\left(\omega_{2n}^k\right)q\left(\omega_{2n}^k\right)\right)$

↓ **Interpolation**

$p(x)q(x)$ of degree $2n-2$, $2n-1$ coefficients

Discrete Fourier Transform

- The values $p\left(\omega_N^i\right)$ for $i = 0, \dots, N-1$ uniquely define a polynomial p of degree $< N$.

Discrete Fourier Transform (DFT):

- Assume $a = (a_0, \dots, a_{N-1})$ is the coefficient vector of poly. p

$$(p(x) = a_{N-1}x^{N-1} + \cdots + a_1 x + a_0)$$

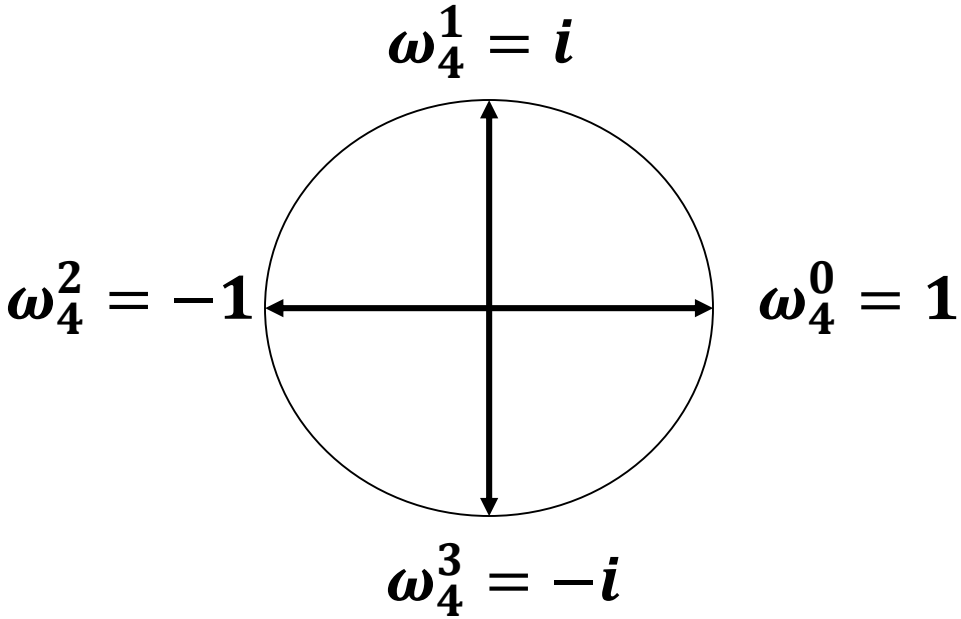
$$\mathrm{DFT}_N(a) \coloneqq \left(p\left(\omega_N^0\right), p(\omega_N^1), \dots, p(\omega_N^{N-1})\right)$$

Example

- Consider polynomial $p(x) = 3x^3 - 15x^2 + 18x$

- Choose $N = 4$

- Roots of unity:



Example

- Consider polynomial $p(x) = 3x^3 - 15x^2 + 18x$
- $N = 4$, roots of unity: $\omega_4^0 = 1, \omega_4^1 = i, \omega_4^2 = -1, \omega_4^3 = -i$
- Evaluate $p(x)$ at ω_4^k:

$$\left(\omega_4^0, p(\omega_4^0)\right) = \left(1, p(1)\right) = (1,6)$$
$$\left(\omega_4^1, p(\omega_4^1)\right) = \left(i, p(i)\right) = (i, 15 + 15i)$$
$$\left(\omega_4^2, p(\omega_4^2)\right) = \left(-1, p(-1)\right) = (-1, -36)$$
$$\left(\omega_4^3, p(\omega_4^3)\right) = \left(-i, p(-i)\right) = (-i, 15 - 15i)$$

- For $a = (0,18,-15,3)$:

$$\mathbf{DFT_4}(\boldsymbol{a}) = (\mathbf{6}, \mathbf{15} + \mathbf{15}\boldsymbol{i}, -\mathbf{36}, \mathbf{15} - \mathbf{15}\boldsymbol{i})$$

DFT: Recursive Structure

Evaluation for $k = 0, \dots, N-1$:

$$p(\omega_N^k) = p_0\left((\omega_N^k)^2\right) + \omega_N^k \cdot p_1\left((\omega_N^k)^2\right)$$

$$= \begin{cases} p_0(\omega_{N/2}^k) + \omega_N^k \cdot p_1(\omega_{N/2}^k) & \text{if } k < {}^N/_2 \\ p_0\left(\omega_{N/2}^{k-N/2}\right) + \omega_N^k \cdot p_1\left(\omega_{N/2}^{k-N/2}\right) & \text{if } k \geq {}^N/_2 \end{cases}$$

For the coefficient vector a of $p(x)$:

$$\mathrm{DFT}_N(a) = \left(p_0(\omega_{N/2}^0), \dots, p_0\left(\omega_{N/2}^{N/2-1}\right), p_0(\omega_{N/2}^0), \dots, p_0\left(\omega_{N/2}^{N/2-1}\right)\right)$$
$$+ \left(\omega_N^0 p_0(\omega_{N/2}^0), \dots, \omega_N^{N/2-1} p_0\left(\omega_{N/2}^{N/2-1}\right), \omega_N^{N/2} p_0(\omega_{N/2}^0), \dots, \omega_N^{N-1} p_0\left(\omega_{N/2}^{N/2-1}\right)\right)$$

Example

For the coefficient vector a of $p(x)$:

$$\mathrm{DFT}_N(a) = \left(p_0(\omega_{N/2}^0), \dots, p_0\left(\omega_{N/2}^{N/2-1}\right), p_0(\omega_{N/2}^0), \dots, p_0\left(\omega_{N/2}^{N/2-1}\right)\right)$$
$$+ \left(\omega_N^0 p_0(\omega_{N/2}^0), \dots, \omega_N^{N/2-1} p_0\left(\omega_{N/2}^{N/2-1}\right), \omega_N^{N/2} p_0(\omega_{N/2}^0), \dots, \omega_N^{N-1} p_0\left(\omega_{N/2}^{N/2-1}\right)\right)$$

$N = 4$:

$$p(\omega_4^0) = p_0(\omega_2^0) + \omega_4^0 p_1(\omega_2^0)$$
$$p(\omega_4^1) = p_0(\omega_2^1) + \omega_4^1 p_1(\omega_2^1)$$
$$p(\omega_4^2) = p_0(\omega_2^0) + \omega_4^2 p_1(\omega_2^0)$$
$$p(\omega_4^3) = p_0(\omega_2^1) + \omega_4^3 p_1(\omega_2^1)$$

Need: $\left(p_0(\omega_2^0), p_0(\omega_2^1)\right)$ and $\left(p_1(\omega_2^0), p_1(\omega_2^1)\right)$

(DFTs of coefficient vectors of p_0 and p_1)

Summary: Computation of DFT_N

- Divide-and-conquer algorithm for $\mathrm{DFT}_N(p)$:

1. Divide

$N \leq 1$: $\mathrm{DFT}_1(p) = a_0$

$N > 1$: Divide p into p_0 (even coeff.) and p_1 (odd coeff).

2. Conquer

Solve $\mathrm{DFT}_{N/2}(p_0)$ and $\mathrm{DFT}_{N/2}(p_1)$ recursively

3. Combine

Compute $\mathrm{DFT}_N(p)$ based on $\mathrm{DFT}_{N/2}(p_0)$ and $\mathrm{DFT}_{N/2}(p_1)$

Small Improvement

Polynomial p of degree $N-1$:

$$p(\omega_N^k) = \begin{cases} p_0(\omega_{N/2}^k) + \omega_N^k \cdot p_1(\omega_{N/2}^k) & \text{if } k < {}^N/_2 \\ p_0\left(\omega_{N/2}^{k-N/2}\right) + \omega_N^k \cdot p_1\left(\omega_{N/2}^{k-N/2}\right) & \text{if } k \geq {}^N/_2 \end{cases}$$

$$= \begin{cases} p_0(\omega_{N/2}^k) + \omega_N^k \cdot p_1(\omega_{N/2}^k) & \text{if } k < {}^N/_2 \\ p_0\left(\omega_{N/2}^{k-N/2}\right) - \omega_N^{k-N/2} \cdot p_1\left(\omega_{N/2}^{k-N/2}\right) & \text{if } k \geq {}^N/_2 \end{cases}$$

Need to compute $p_0(\omega_{N/2}^k)$ and $\omega_N^k \cdot p_1(\omega_{N/2}^k)$ for $0 \leq k < {}^N/_2$.

Example $N = 8$

$$p(\omega_8^0) = p_0(\omega_4^0) + \omega_8^0 \cdot p_1(\omega_4^0)$$
$$p(\omega_8^1) = p_0(\omega_4^1) + \omega_8^1 \cdot p_1(\omega_4^1)$$
$$p(\omega_8^2) = p_0(\omega_4^2) + \omega_8^2 \cdot p_1(\omega_4^2)$$
$$p(\omega_8^3) = p_0(\omega_4^3) + \omega_8^3 \cdot p_1(\omega_4^3)$$
$$p(\omega_8^3) = p_0(\omega_4^0) - \omega_8^0 \cdot p_1(\omega_4^0)$$
$$p(\omega_8^3) = p_0(\omega_4^1) - \omega_8^1 \cdot p_1(\omega_4^1)$$
$$p(\omega_8^3) = p_0(\omega_4^2) - \omega_8^2 \cdot p_1(\omega_4^2)$$
$$p(\omega_8^3) = p_0(\omega_4^3) - \omega_8^3 \cdot p_1(\omega_4^3)$$

Fast Fourier Transform (FFT) Algorithm

Algorithm FFT(a)

- Input: Array a of length N, where N is a power of 2
- Output: $\mathrm{DFT}_N(a)$

if $n = 1$ **then return** a_0; // $a = [a_0]$
$d^{[0]} := \mathrm{FFT}([a_0, a_2, \dots, a_{N-2}])$;
$d^{[1]} := \mathrm{FFT}([a_1, a_3, \dots, a_{N-1}])$;
$\omega_N := e^{2\pi i/N}$; $\omega := 1$;
for $k = 0$ **to** $N/2 - 1$ **do** // $\omega = \omega_N^k$

$x := \omega \cdot d_k^{[1]}$;
$d_k := d_k^{[0]} + x$; $d_{k+N/2} := d_k^{[0]} - x$;
$\omega := \omega \cdot \omega_N$

end;

return $d = [d_0, d_1, \dots, d_{N-1}]$;

Example

$$p(x) = 3x^3 - 15x^2 + 18x + 0, \qquad a = [0,18,-15,3]$$

Faster Polynomial Multiplication?

Idea to compute $p(x) \cdot q(x)$ (for polynomials of degree $< n$):

p, q of degree $n-1$, n coefficients

↓ **Evaluation** at $\omega_{2n}^0, \omega_{2n}^1, \dots, \omega_{2n}^{2n-1}$ using **FFT**

$2 \times 2n$ point-value pairs $\left(\omega_{2n}^k, p\left(\omega_{2n}^k\right)\right)$ and $\left(\omega_{2n}^k, q\left(\omega_{2n}^k\right)\right)$

↓ **Point-wise multiplication**

$2n$ point-value pairs $\left(\omega_{2n}^k, p\left(\omega_{2n}^k\right)q\left(\omega_{2n}^k\right)\right)$

↓ **Interpolation**

$p(x)q(x)$ of degree $2n-2$, $2n-1$ coefficients

Interpolation

Convert point-value representation into coefficient representation

Input: $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ with $x_i \neq x_j$ for $i \neq j$

Output:

Degree-$(n-1)$ polynomial with coefficients $a_0, \dots, a_{n-1}$ such that

$$\begin{aligned}
p(x_0) &= a_0 + a_1 x_0 + a_2 x_0^2 + \cdots + a_{n-1} x_0^{n-1} = y_0 \\
p(x_1) &= a_0 + a_1 x_1 + a_2 x_1^2 + \cdots + a_{n-1} x_1^{n-1} = y_1 \\
&\vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
p(x_{n-1}) &= a_0 + a_1 x_{n-1} + a_2 x_{n-1}^2 + \cdots + a_{n-1} x_{n-1}^{n-1} = y_{n-1}
\end{aligned}$$

→ linear system of equations for $a_0, \dots, a_{n-1}$

Interpolation

Matrix Notation:

$$\begin{pmatrix} 1 & x_0 & \cdots & x_0^{n-1} \\ 1 & x_1 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & \cdots & x_{n-1}^{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

- System of equations solvable iff $x_i \neq x_j$ for all $i \neq j$

Special Case $\boldsymbol{x_i = \omega_n^i}$:

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Interpolation

- Linear system:

$$W \cdot \boldsymbol{a} = \boldsymbol{y} \quad \Longrightarrow \quad \boldsymbol{a} = W^{-1} \cdot \boldsymbol{y}$$

$$W_{i,j} = \omega_n^{ij}, \qquad \boldsymbol{a} = \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix}, \qquad \boldsymbol{y} = \begin{pmatrix} y_0 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Claim:

$$\boldsymbol{W_{ij}^{-1} = \frac{\omega_n^{-ij}}{n}}$$

Proof: Need to show that $W^{-1}W = I_n$

DFT Matrix Inverse

$$W^{-1}W = \begin{pmatrix} \frac{1}{n} & \frac{\omega_n^{-i}}{n} & \begin{matrix} \cdots \\ \cdots \\ \vdots \\ \cdots \end{matrix} & \frac{\omega_n^{-(n-1)i}}{n} \end{pmatrix} \cdot \begin{pmatrix} \cdots & 1 & \cdots \\ \cdots & \omega_n^{j} & \cdots \\ \cdots & \omega_n^{2j} & \cdots \\ & \vdots & \\ \cdots & \omega_n^{(n-1)j} & \cdots \end{pmatrix}$$

DFT Matrix Inverse

$$(W^{-1}W)_{i,j} = \sum_{\ell=0}^{n-1} \frac{\omega_n^{\ell(j-i)}}{n}$$

Need to show $(W^{-1}W)_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$

Case $\boldsymbol{i = j}$:

DFT Matrix Inverse

$$(W^{-1}W)_{i,j} = \sum_{\ell=0}^{n-1} \frac{\omega_n^{\ell(j-i)}}{n}$$

Case $\boldsymbol{i \neq j}$:

Inverse DFT

- $W^{-1} = \begin{pmatrix} & & \cdots & \\ \frac{1}{n} & \frac{\omega_n^{-k}}{n} & \cdots & \frac{\omega_n^{-(n-1)k}}{n} \\ & & \vdots & \\ & & \cdots & \end{pmatrix}$

- We get $\boldsymbol{a} = W^{-1} \cdot \boldsymbol{y}$ and therefore

$$a_k = \begin{pmatrix} \frac{1}{n} & \frac{\omega_n^{-k}}{n} & \cdots & \frac{\omega_n^{-(n-1)k}}{n} \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

$$= \frac{1}{n} \cdot \sum_{j=0}^{n-1} \omega_n^{-kj} \cdot y_j$$

DFT and Inverse DFT

Inverse DFT:

$$a_k = \frac{1}{n} \cdot \sum_{j=0}^{n-1} \omega_n^{-kj} \cdot y_j$$

- Define polynomial $q(x) = y_0 + y_1 z + \cdots + y_{n-1} z^{n-1}$:

$$a_k = \frac{1}{n} \cdot q(\omega_n^{-k})$$

DFT:

- Polynomial $p(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$:

$$y_k = p(\omega_n^k)$$

DFT and Inverse DFT

$$q(x) = y_0 + y_1 x + \cdots + y_{n-1} x^{n-1}, \qquad a_k = \frac{1}{n} \cdot q\left(\omega_n^{-k}\right):$$

- Therefore:

$$(a_0, a_1, \dots, a_{n-1})$$
$$= \frac{1}{n} \cdot \left(q(\omega_n^{-0}), q(\omega_n^{-1}), q(\omega_n^{-2}), \dots, q\left(\omega_n^{-(n-1)}\right)\right)$$
$$= \frac{1}{n} \cdot \left(q(\omega_n^{0}), q(\omega_n^{n-1}), q(\omega_n^{n-2}), \dots, q(\omega_n^{1})\right)$$

- Recall:

$$\mathrm{DFT}_n(\boldsymbol{y}) = \left(q(\omega_n^0), q(\omega_n^1), q(\omega_n^2), \dots, q(\omega_n^{n-1})\right)$$
$$= n \cdot (a_0, a_{n-1}, a_{n-2}, \dots, a_2, a_1)$$

DFT and Inverse DFT

- We have $\mathrm{DFT}_n(\boldsymbol{y}) = n \cdot (a_0, a_{n-1}, a_{n-2}, \dots, a_2, a_1)$:

$$a_i = \begin{cases} \frac{1}{n} \cdot (\mathrm{DFT}_n(\boldsymbol{y}))_0 & \text{if } i = 0 \\ \frac{1}{n} \cdot (\mathrm{DFT}_n(\boldsymbol{y}))_{n-i} & \text{if } i \neq 0 \end{cases}$$

- DFT and inverse DFT can both be computed using FFT algorithm in $O(n \log n)$ time.

- 2 polynomials of degr. $< n$ can be multiplied in time $O(n \log n)$.

Faster Polynomial Multiplication?

Idea to compute $p(x) \cdot q(x)$ (for polynomials of degree $< n$):

p, q of degree $n-1$, n coefficients

↓ **Evaluation** at $\omega_{2n}^0, \omega_{2n}^1, \dots, \omega_{2n}^{2n-1}$ using **FFT**

$2 \times 2n$ point-value pairs $\left(\omega_{2n}^k, p\left(\omega_{2n}^k\right)\right)$ and $\left(\omega_{2n}^k, q\left(\omega_{2n}^k\right)\right)$

↓ **Point-wise multiplication**

$2n$ point-value pairs $\left(\omega_{2n}^k, p\left(\omega_{2n}^k\right)q\left(\omega_{2n}^k\right)\right)$

↓ **Interpolation** using **FFT**

$p(x)q(x)$ of degree $2n-2$, $2n-1$ coefficients

Convolution

- More generally, the polynomial multiplication algorithm computes the convolution of two vectors:

$$\boldsymbol{a} = (a_0, a_1, \dots, a_{m-1})$$
$$\boldsymbol{b} = (b_0, b_1, \dots, b_{n-1})$$

$$\boldsymbol{a} * \boldsymbol{b} = (c_0, c_1, \dots, c_{m+n-2}),$$
$$\text{where } c_k = \sum_{\substack{(i,j): i+j=k \\ i<m, j<n}} a_i b_j$$

- c_k is exactly the coefficient of x^k in the product polynomial of the polynomials defined by the coefficient vectors $\boldsymbol{a}$ and $\boldsymbol{b}$

More Applications of Convolutions

Signal Processing Example:

- Assume $\boldsymbol{a} = (a_0, \dots, a_{n-1})$ represents a sequence of measurements over time
- Measurements might be noisy and have to be smoothed out
- Replace a_i by weighted average of nearby last m and next m measurements (e.g., Gaussian smoothing):

$$a_i' = \frac{1}{Z} \cdot \sum_{j=i-m}^{i+m} a_j e^{-(i-j)^2}$$

- New vector $\boldsymbol{a}'$ is the convolution of $\boldsymbol{a}$ and the weight vector $\frac{1}{Z} \cdot \left(e^{-m^2}, e^{-(m-1)^2}, \dots, e^{-1}, 1, e^{-1}, \dots, e^{-(m-1)^2}, e^{-m^2}\right)$
- Might need to take care of boundary points…

More Applications of Convolutions

Combining Histograms:

- Vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ represent two histograms
- E.g., annual income of all men & annual income of all women

- Goal: Get new histogram $\boldsymbol{c}$ representing combined income of all possible pairs of men and women:

$$\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$$

Also, the DFT (and thus the FFT alg.) has many other applications!

DFT in Signal Processing

Assume that $y(0), y(1), y(2), \dots, y(T-1)$ are measurements of a time-dependent signal.

Inverse DFT_N of $\big(y(0), \dots, y(T-1)\big)$ is a vector $(c_0, \dots, c_{N-1})$ s.t.

$$y(t) = \sum_{k=0}^{N-1} c_k \cdot e^{\frac{2\pi i \cdot k}{N} \cdot t}$$

$$= \sum_{k=0}^{T-1} c_k \cdot \left(\cos\left(\frac{2\pi \cdot k}{N} \cdot t \right) + i \sin\left(\frac{2\pi \cdot k}{N} \cdot t \right) \right)$$

- Converts signal from time domain to frequency domain
- Signal can then be edited in the frequency domain
  - e.g., setting some $c_k = 0$ filters out some frequencies

Chapter 2
Greedy Algorithms

Algorithm Theory
WS 2018/19

Fabian Kuhn

Greedy Algorithms

- No clear definition, but essentially:

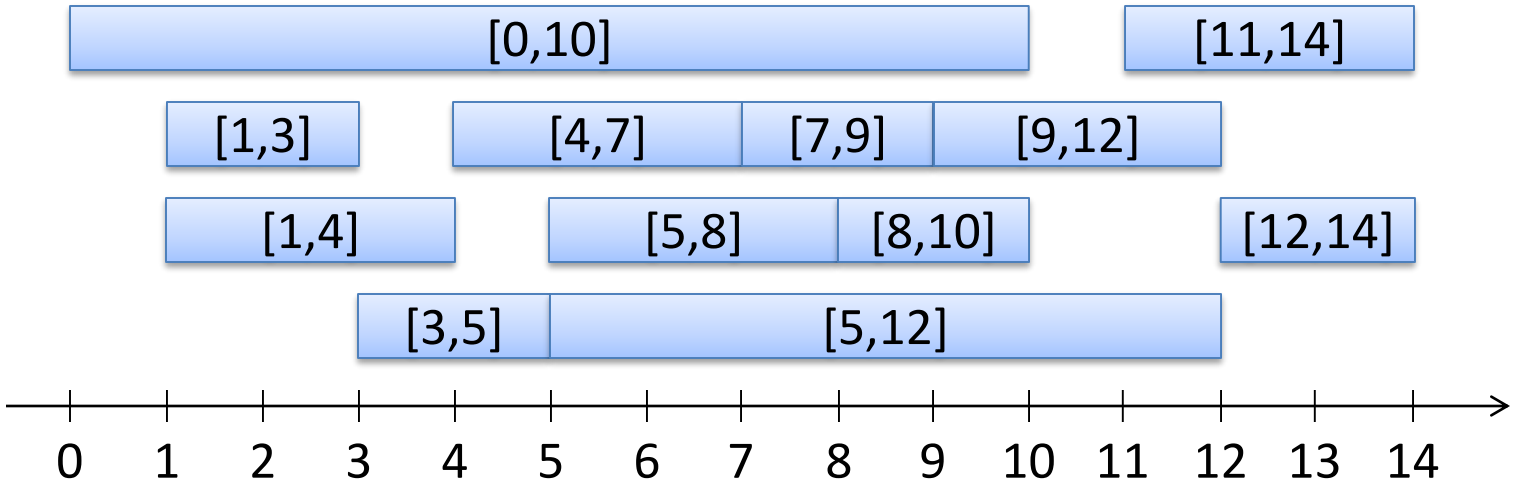
In each step make the choice that looks best at the moment!

- Depending on problem, greedy algorithms can give
  - Optimal solutions
  - Close to optimal solutions
  - No (reasonable) solutions at all
- If it works, very interesting approach!
  - And we might even learn something about the structure of the problem

Goal: Improve understanding where it works (mostly by examples)

Interval Scheduling

- **Given:** Set of intervals, e.g.
  [0,10],[1,3],[1,4],[3,5],[4,7],[5,8],[5,12],[7,9],[9,12],[8,10],[11,14],[12,14]



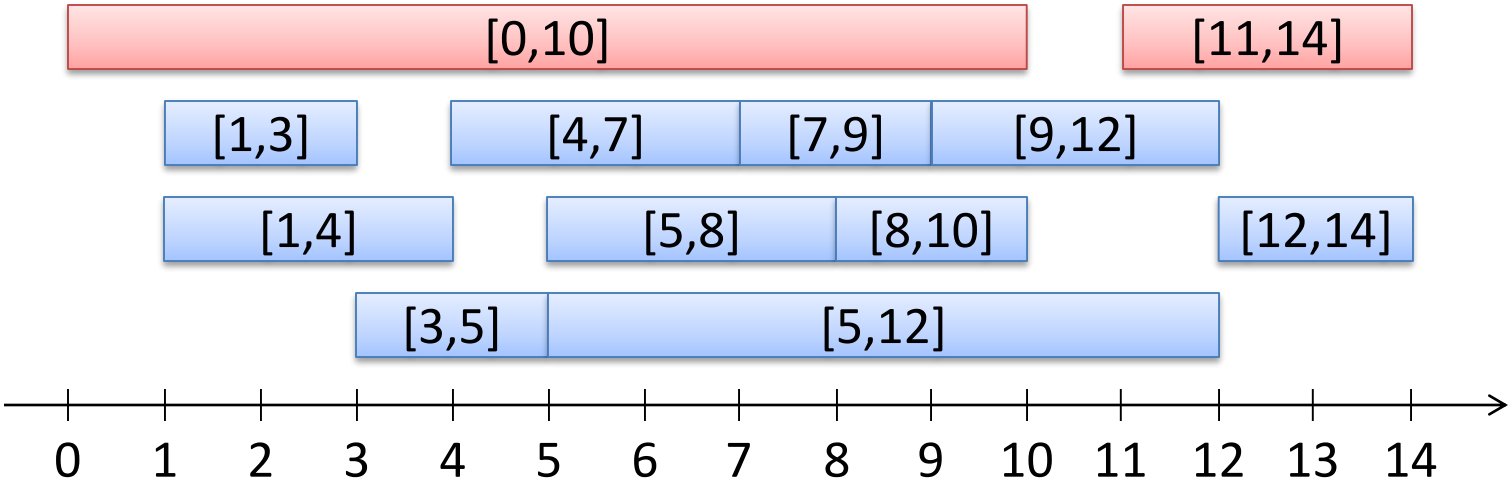
- **Goal:** Select largest possible non-overlapping set of intervals
  - For simplicity: overlap at boundary ok
    (i.e., [4,7] and [7,9] are non-overlapping)

- **Example:** Intervals are room requests; satisfy as many as possible

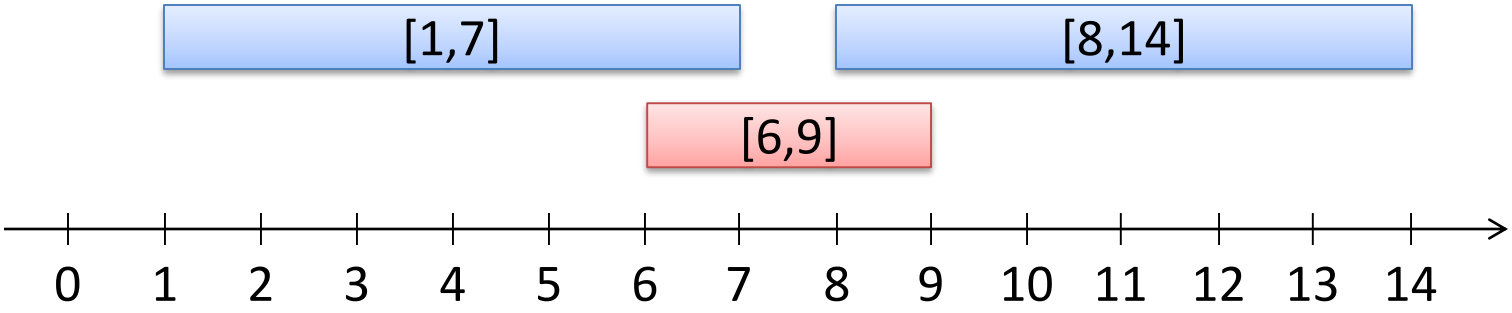
Greedy Algorithms

- Several possibilities…

Choose first available interval:

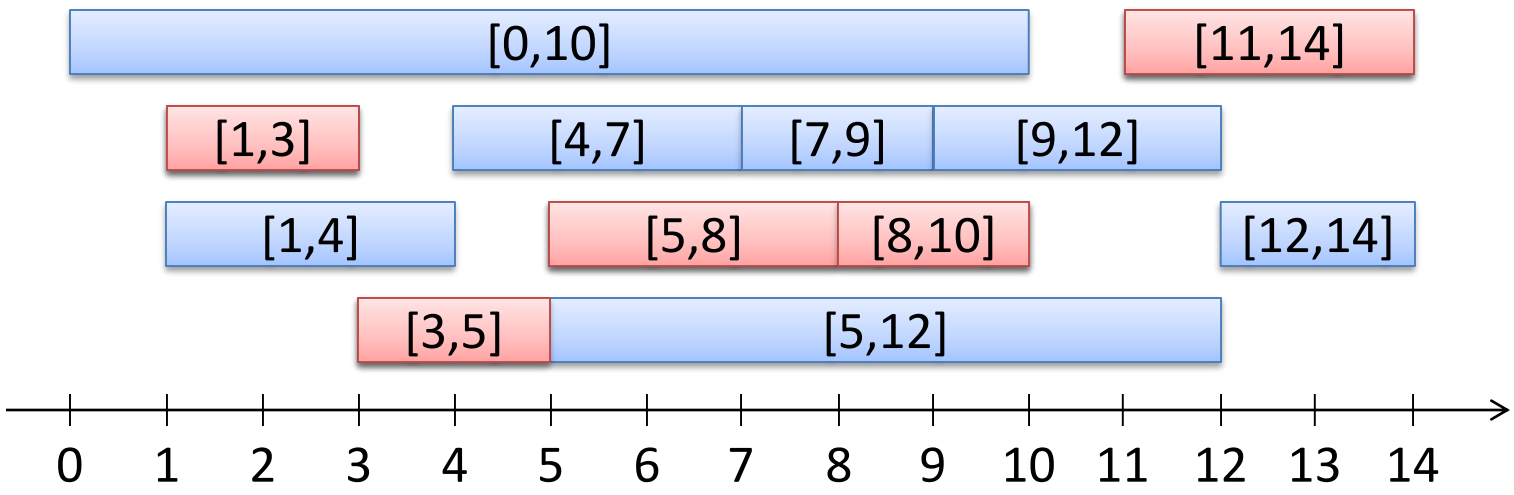


Choose shortest available interval:



Greedy Algorithms

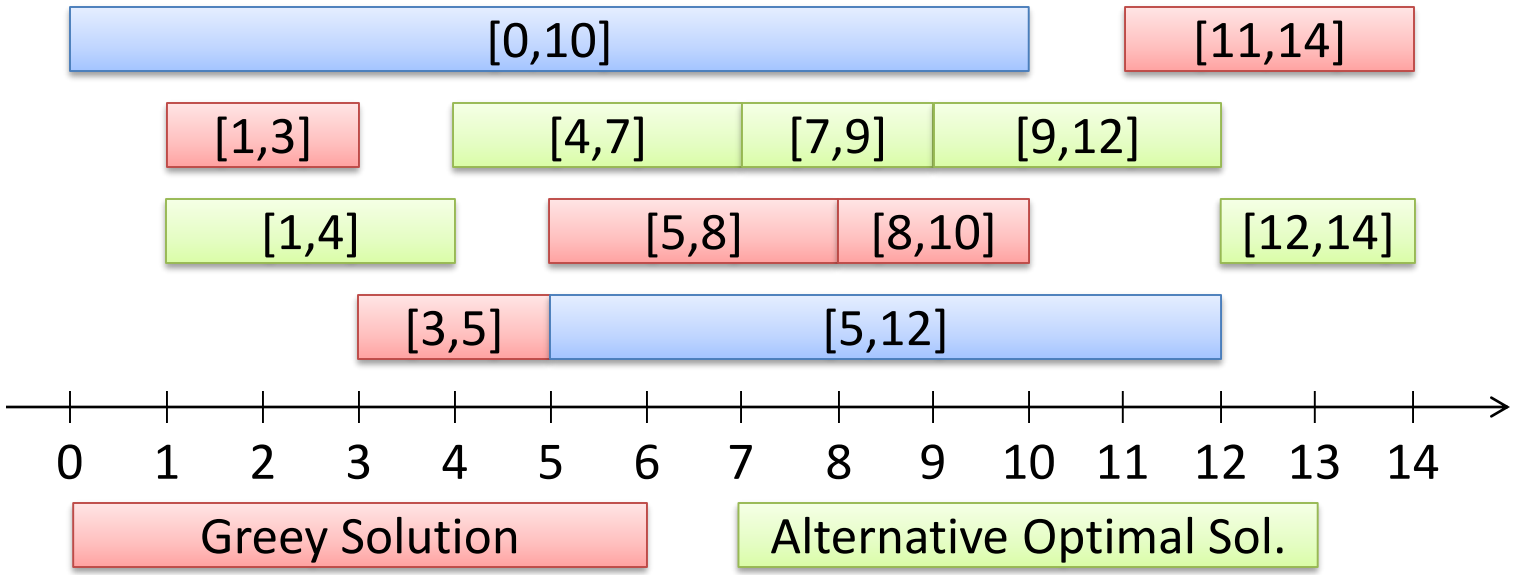
Choose available request with earliest finishing time:



R := set of all requests; S := empty set;
while R is not empty **do**
    choose $r \in R$ with smallest finishing time
    add r to S
    delete all requests from R that are not compatible with r
end // S is the solution

Earliest Finishing Time is Optimal

- Let O be the set of intervals of an optimal solution
- Can we show that $S = O$?
  - No…



- Show that $|S| = |O|$.

Greedy Stays Ahead

- Greedy solution S:

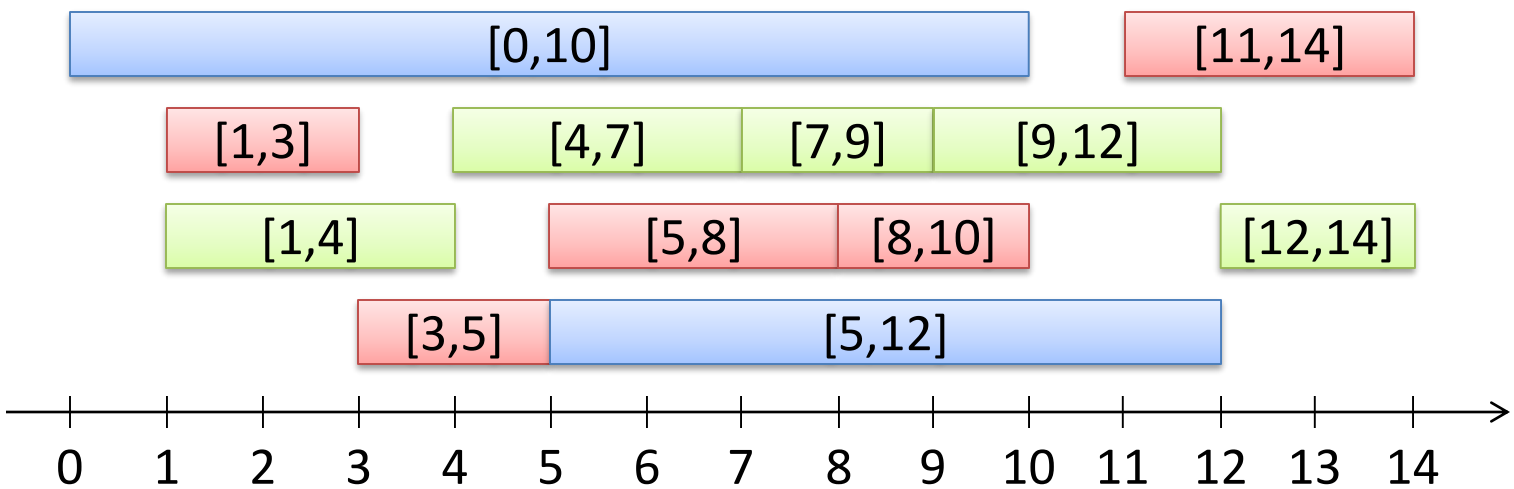
$$[a_1, b_1], [a_2, b_2], \dots, [a_{|S|}, b_{|S|}], \qquad \text{where } b_i \leq a_{i+1}$$

- Some optimal solution O:

$$[a_1^*, b_1^*], [a_2^*, b_2^*], \dots, [a_{|O|}^*, b_{|O|}^*], \qquad \text{where } b_i^* \leq a_{i+1}^*$$

- Definde $b_i := \infty$ for $i > |S|$ and $b_i^* := \infty$ for $i > |O|$

Claim: For all $i \geq 1$, $b_i \leq b_i^*$



Greedy Stays Ahead

Claim: For all $i \geq 1$, $b_i \leq b_i^*$

Proof (by induction on i):

Corollary: Earliest finishing time algorithm is optimal.

Weighted Interval Scheduling

Weighted version of the problem:

- Each interval has a weight
- Goal: Non-overlapping set with maximum total weight

Earliest finishing time greedy algorithm fails:

- Algorithm needs to look at weights
- Else, the selected sets could be the ones with smallest weight…

No simple greedy algorithm:

- We will see an algorithm using another design technique later.

Chapter 2
Greedy Algorithms

Algorithm Theory
WS 2018/19

Fabian Kuhn

Greedy Algorithms

- No clear definition, but essentially:

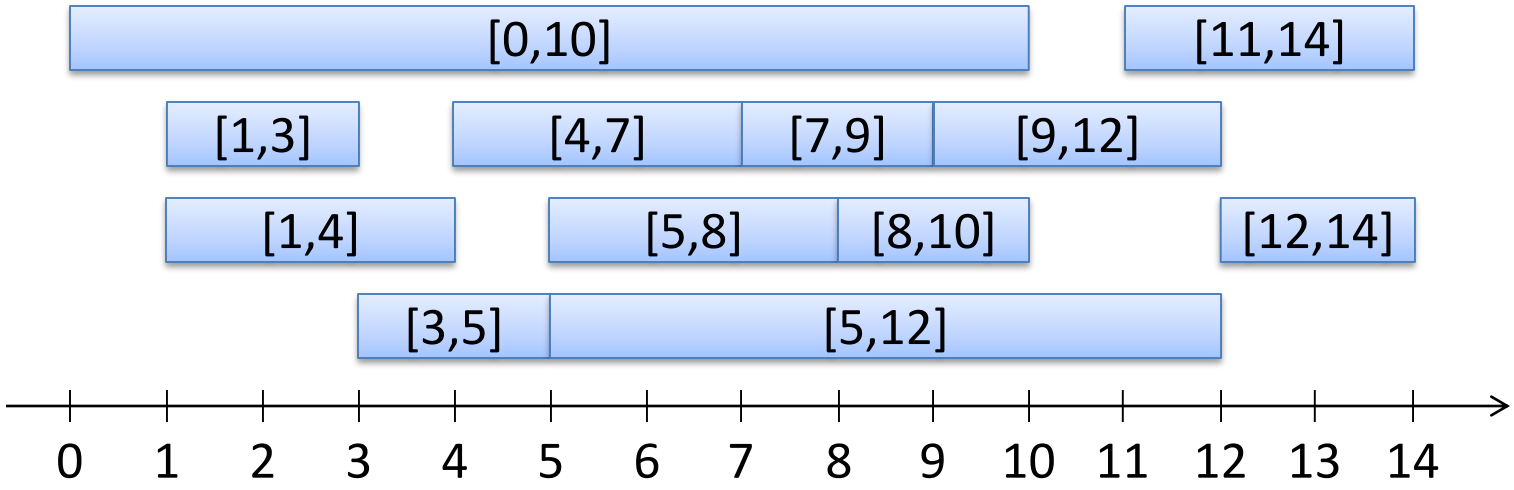
In each step make the choice that looks best at the moment!

- Depending on problem, greedy algorithms can give
  - Optimal solutions
  - Close to optimal solutions
  - No (reasonable) solutions at all
- If it works, very interesting approach!
  - And we might even learn something about the structure of the problem

Goal: Improve understanding where it works (mostly by examples)

Interval Scheduling

- **Given:** Set of intervals, e.g.
  [0,10],[1,3],[1,4],[3,5],[4,7],[5,8],[5,12],[7,9],[9,12],[8,10],[11,14],[12,14]

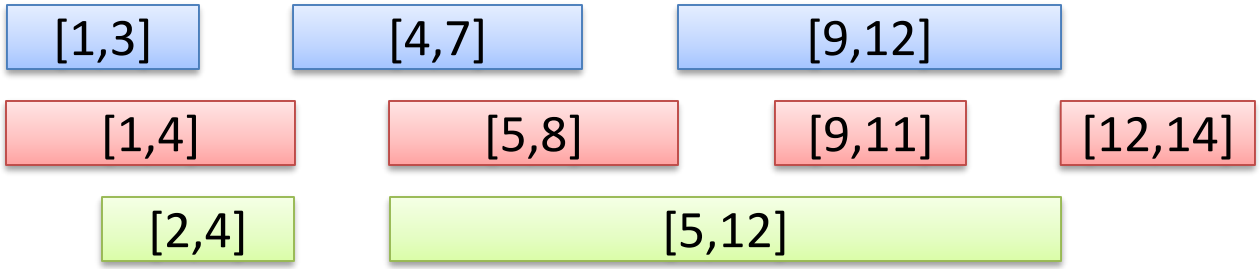


- **Goal:** Select largest possible non-overlapping set of intervals
  - For simplicity: overlap at boundary ok
    (i.e., [4,7] and [7,9] are non-overlapping)

- **Example:** Intervals are room requests; satisfy as many as possible

Interval Partitioning

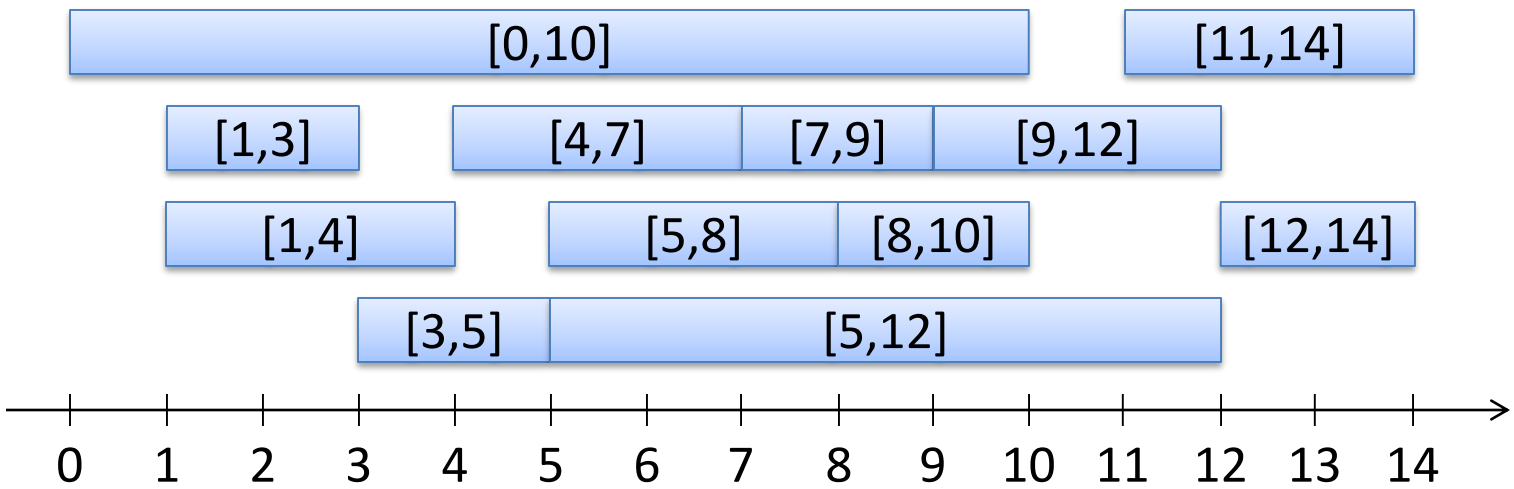
- **Schedule all intervals**: Partition intervals into as few as possible non-overlapping sets of intervals
  - Assign intervals to different resources, where each resource needs to get a non-overlapping set
- Example:
  - Intervals are requests to use some room during this time
  - Assign all requests to some room such that there are no conflicts
  - Use as few rooms as possible
- Assignment to 3 resources:



Depth

Depth of a set of intervals:

- Maximum number passing over a single point in time
- Depth of initial example is 4 (e.g., [0,10],[4,7],[5,8],[5,12]):



Lemma: Number of resources needed $\geq$ depth

Greedy Algorithm

Can we achieve a partition into "depth" non-overlapping sets?

- Would mean that the only obstacles to partitioning are local…

Algorithm:

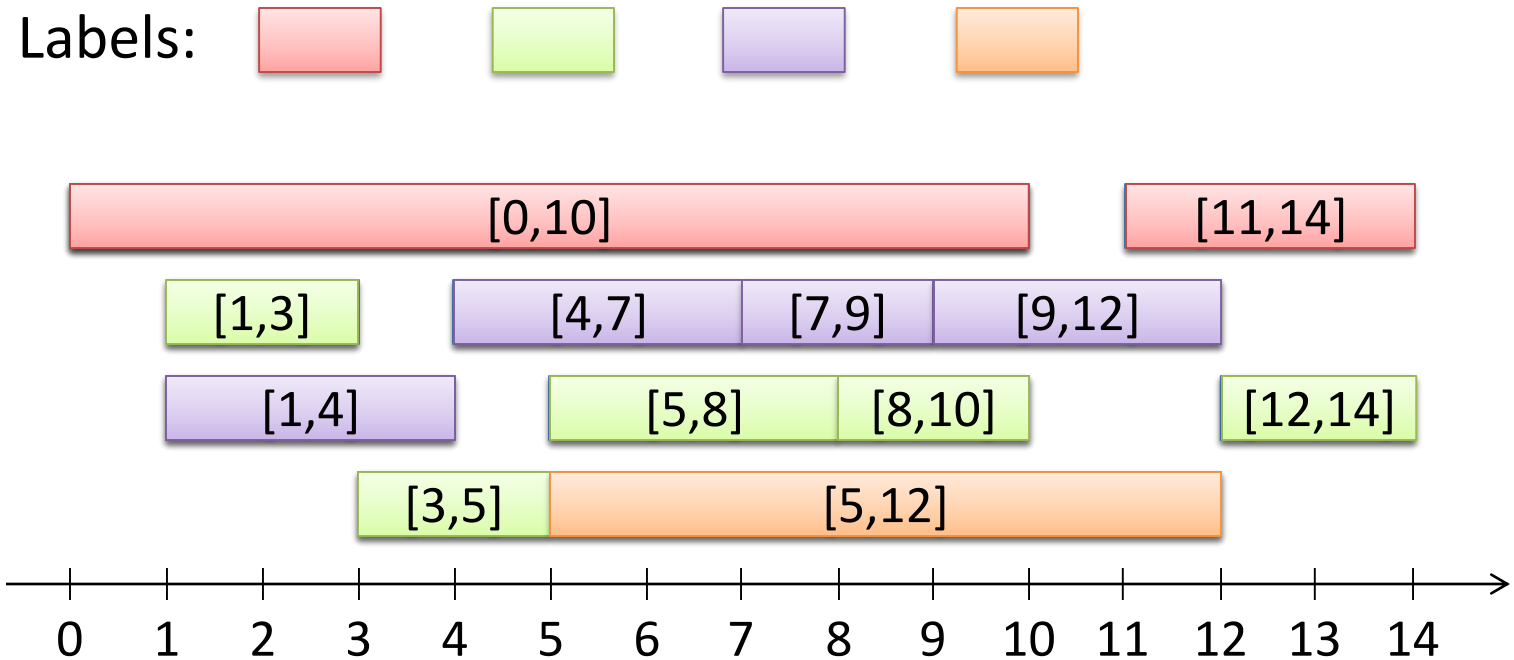
- Assign labels $1, \dots$ to the intervals; same label → non-overlapping

1. sort intervals by starting time: $I_1, I_2, \dots, I_n$
2. **for** $i = 1$ **to** n **do**
3. assign smallest possible label to I_i
   (possible label: different from conflicting intervals $I_j, j < i$)
4. **end**

Interval Partitioning Algorithm

Example:

- Labels:



- Number of labels = depth = 4

Interval Partitioning: Analysis

Theorem:

a) Let d be the depth of the given set of intervals. The algorithm assigns a label from $1, \dots, d$ to each interval.

b) Sets with the same label are non-overlapping

Proof:

- b) holds by construction
- For a):
  - All intervals I_j, $j < i$ overlapping with I_i, overlap at the beginning of I_i
  - At most $d - 1$ such intervals → some label in $\{1, \dots, d\}$ is available.

Traveling Salesperson Problem (TSP)

Input:

- Set V of n nodes (points, cities, locations, sites)
- Distance function $d: V \times V \to \mathbb{R}$, i.e., $d(u, v)$: dist. from u to v
- Distances usually symmetric, asymm. distances → asymm. TSP

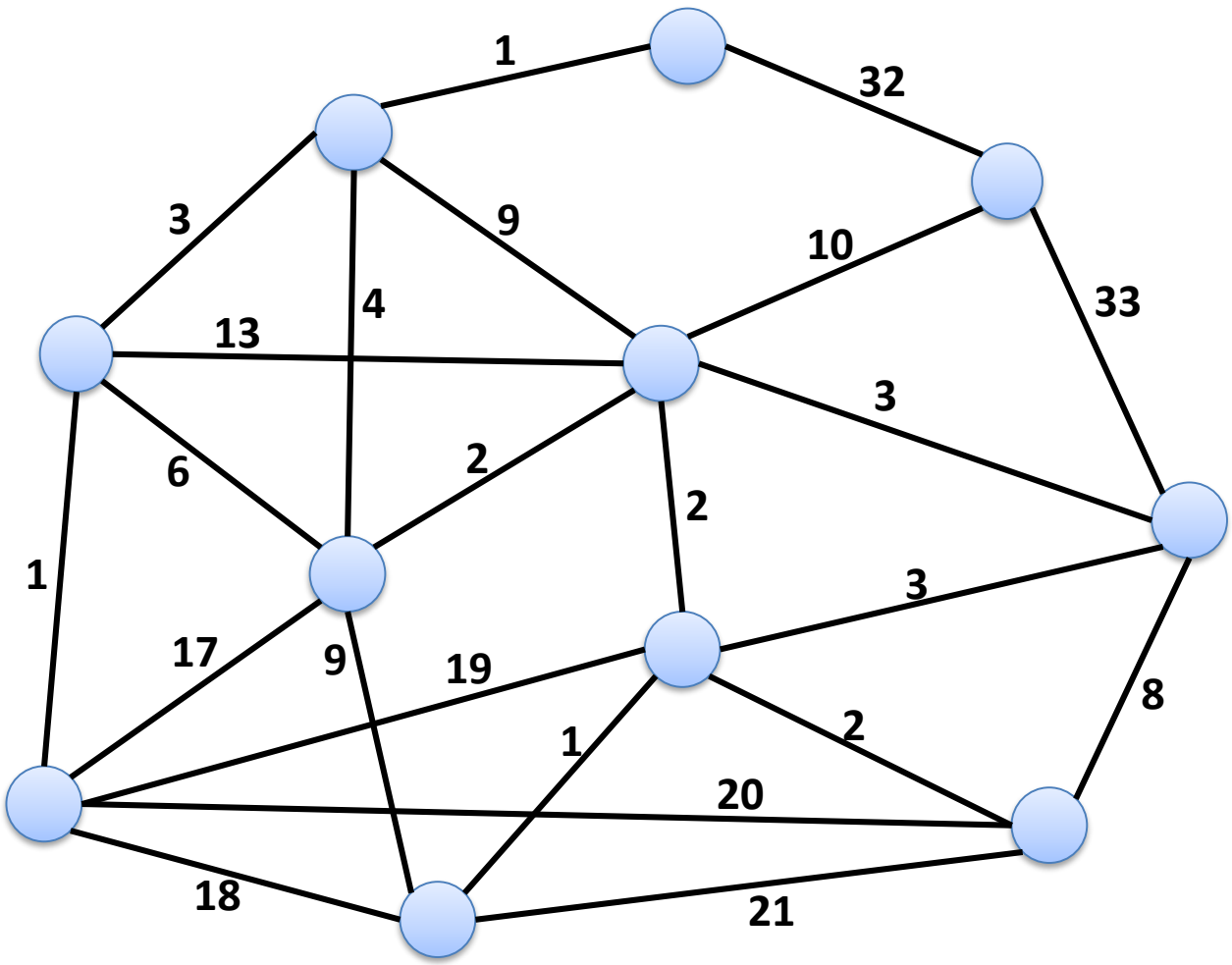
Solution:

- Ordering/permutation $v_1, v_2, \dots, v_n$ of nodes
- Length of TSP path: $\sum_{i=1}^{n-1} d(v_i, v_{i+1})$
- Length of TSP tour: $d(v_n, v_1) + \sum_{i=1}^{n-1} d(v_i, v_{i+1})$

Goal:

- Minimize length of TSP path or TSP tour

Example



Optimal Tour:

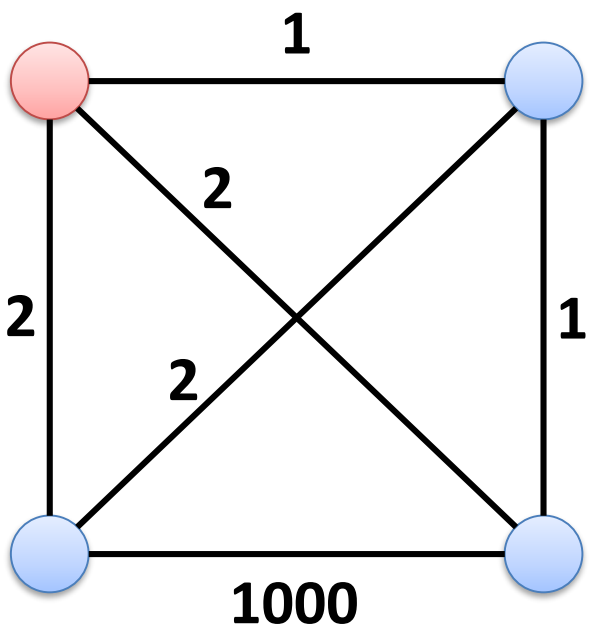
Length: 86

Greedy Algorithm?

Length: 121

Nearest Neighbor (Greedy)

- Nearest neighbor can be arbitrarily bad, even for TSP paths



TSP Variants

- Asymmetric TSP
  - arbitrary non-negative distance/cost function
  - most general, nearest neighbor arbitrarily bad
  - NP-hard to get within any bound of optimum

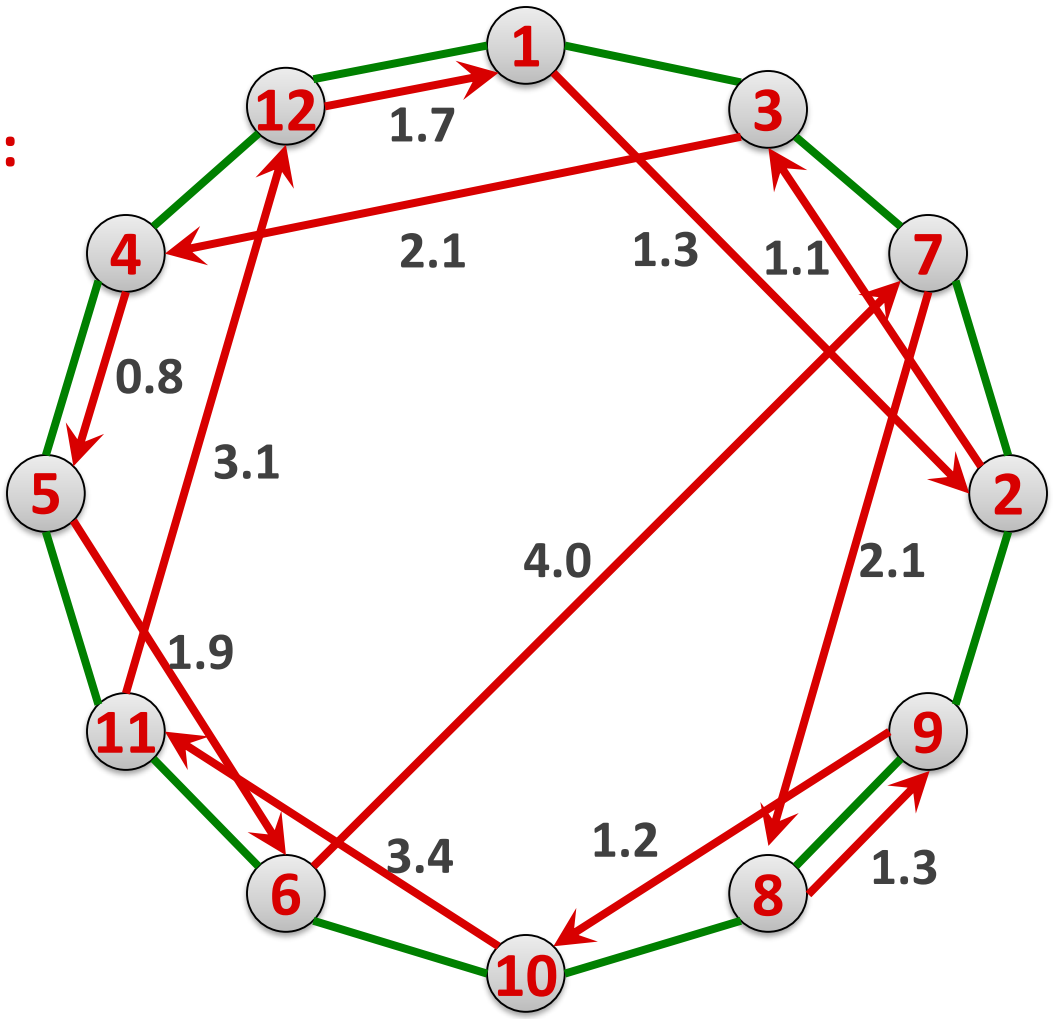
- Symmetric TSP
  - arbitrary non-negative distance/cost function
  - nearest neighbor arbitrarily bad
  - NP-hard to get within any bound of optimum

- Metric TSP
  - distance function defines metric space: symmetric, non-negative, triangle inequality: $d(u,v) \leq d(u,w) + d(w,v)$
  - possible to get close to optimum (we will later see factor $^3/_2$)
  - what about the nearest neighbor algorithm?

Metric TSP, Nearest Neighbor

Optimal TSP tour:

Nearest-Neighbor TSP tour:

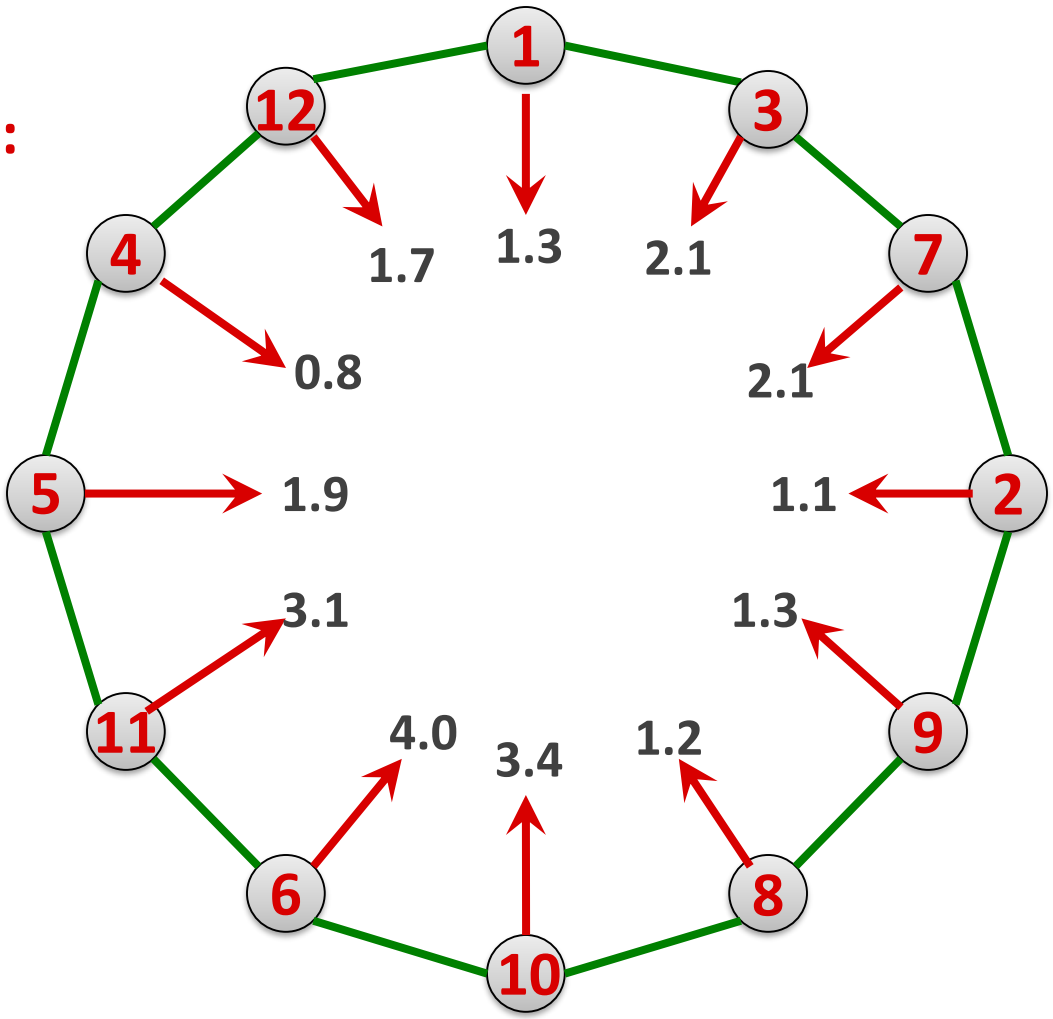


Metric TSP, Nearest Neighbor

Optimal TSP tour:

Nearest-Neighbor TSP tour:

cost = 24



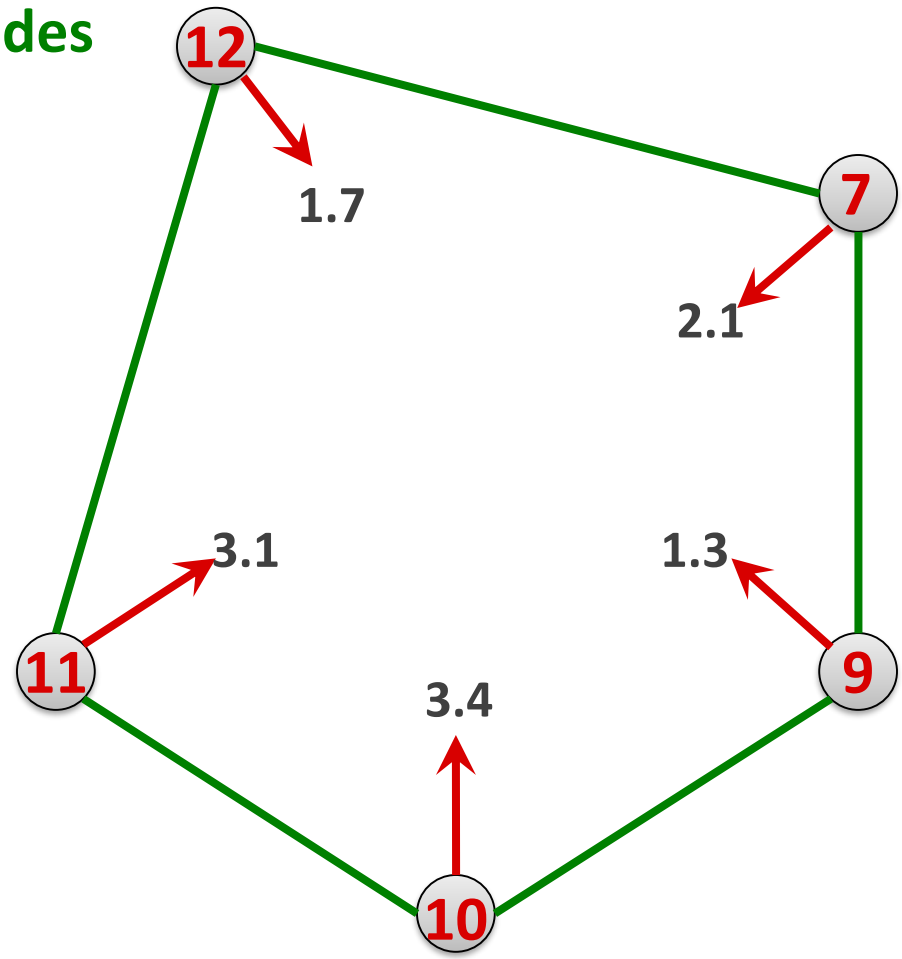
Metric TSP, Nearest Neighbor

Triangle Inequality:

optimal tour on remaining nodes

$\leq$

overall optimal tour



Metric TSP, Nearest Neighbor

Analysis works in phases:

- In each phase, assign each optimal edge to some greedy edge
  - Cost of greedy edge $\leq$ cost of optimal edge
- Each greedy edge gets assigned ≤ 2 optimal edges
  - At least half of the greedy edges get assigned
- At end of phase:

  Remove points for which greedy edge is assigned

  Consider optimal solution for remaining points
- **Triangle inequality:** remaining opt. solution $\leq$ overall opt. sol.

- Cost of greedy edges assigned in **each phase $\leq$ opt. cost**
- **Number of phases $\leq \log_2 n$**
  - +1 for last greedy edge in tour

Metric TSP, Nearest Neighbor

- Assume:

  NN: cost of greedy tour, OPT: cost of optimal tour

- We have shown:

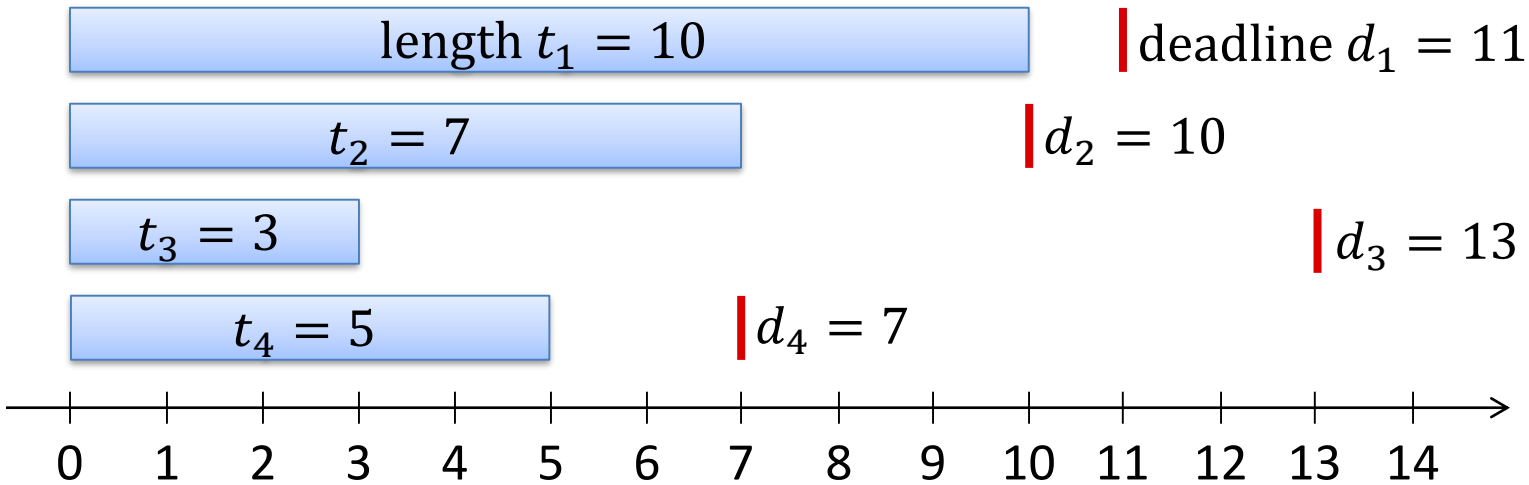
$$\frac{\mathrm{NN}}{\mathrm{OPT}} \leq 1 + \log_2 n$$

- Example of an **approximation algorithm**

- We will later see a $^3/_2$-approximation algorithm for metric TSP

Back to Scheduling

- Given: n requests / jobs with deadlines:



- Goal: schedule all jobs with minimum lateness L
  - Schedule: $s(i)$, $f(i)$: start and finishing times of request i
    Note: $f(i) = s(i) + t_i$

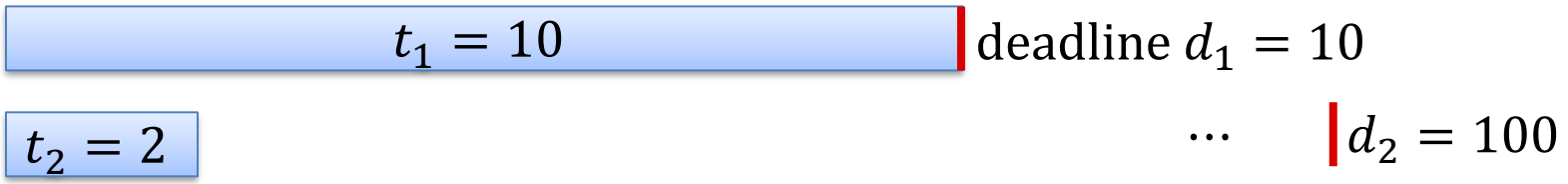
- Lateness $L := \max\left\{0, \max_i\{f(i) - d_i\}\right\}$
  - largest amount of time by which some job finishes late

- Many other natural objective functions possible…

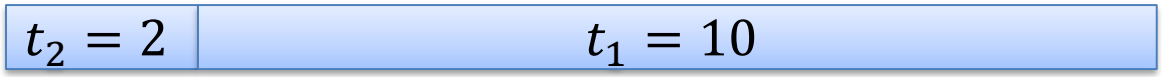
Greedy Algorithm?

Schedule jobs in order of increasing length?

- Ignores deadlines: seems too simplistic…
- E.g.:

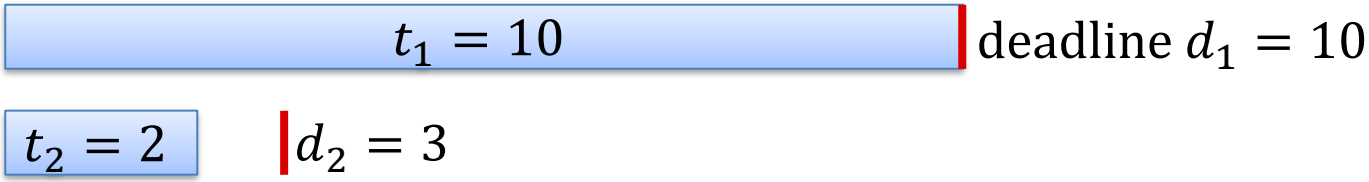


Schedule:



Schedule by increasing slack time?

- Should be concerned about slack time: $d_i - t_i$



Schedule:

Greedy Algorithm

Schedule by earliest deadline?

- Schedule in increasing order of d_i
- Ignores lengths of jobs: too simplistic?

- Earliest deadline is optimal!

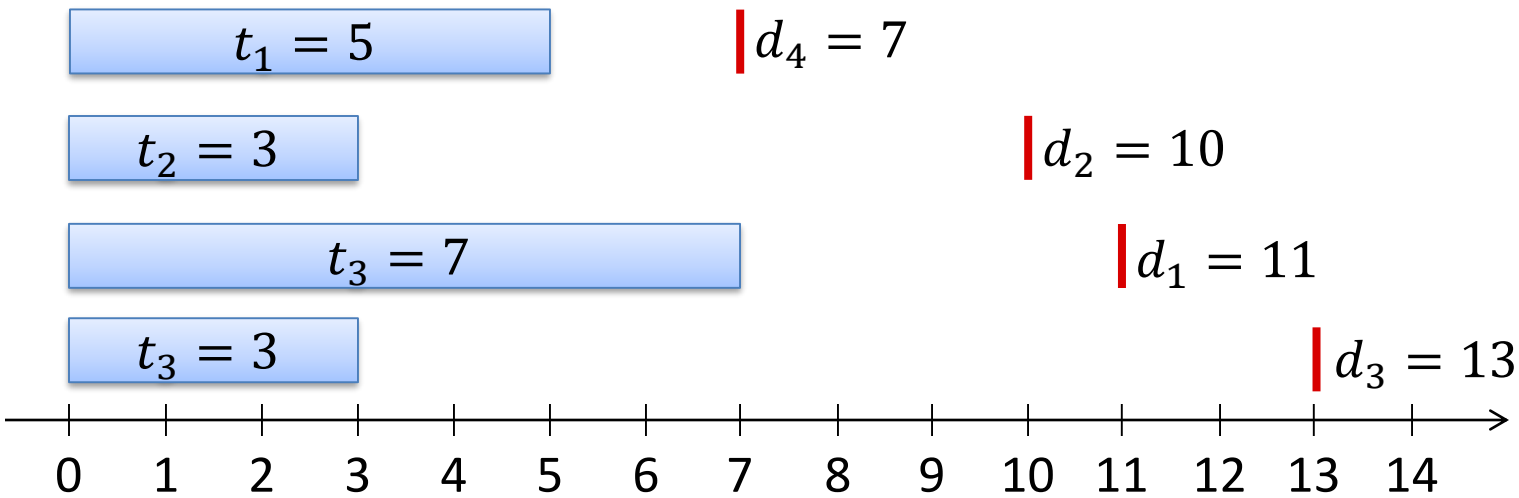
Algorithm:

- Assume jobs are reordered such that $d_1 \leq d_2 \leq \cdots \leq d_n$
- Start/finishing times:
  - First job starts at time $s(1) = 0$
  - Duration of job i is t_i: $f(i) = s(i) + t_i$
  - No gaps between jobs: $s(i+1) = f(i)$

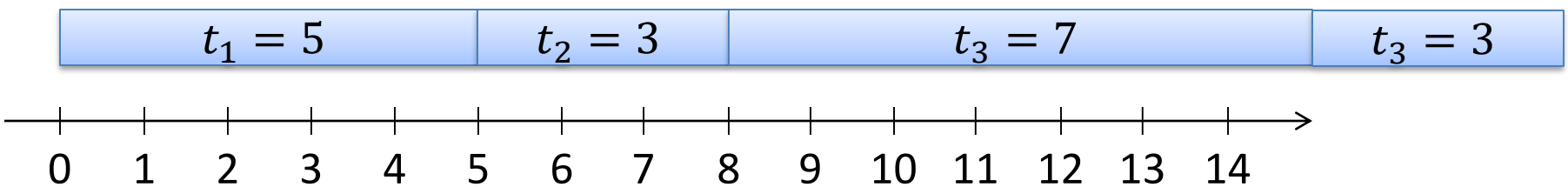
(idle time: gaps in a schedule → alg. gives schedule with no idle time)

Example

Jobs ordered by deadline:



Schedule:



Lateness: job 1: 0, job 2: 0, job 3: 4, job 4: 5

Basic Facts

1. There is an optimal schedule with no idle time
   - Can just schedule jobs earlier…

2. Inversion: Job i scheduled before job j if $d_i > d_j$
   Schedules with no inversions have the same maximum lateness

Earliest Deadline is Optimal

Theorem:
There is an optimal schedule $\mathcal{O}$ with no inversions and no idle time.

Proof:

- Consider some schedule $\mathcal{O}'$ with no idle time
- If $\mathcal{O}'$ has inversions, ∃ pair (i, j), s.t. i is scheduled immediately before j and $d_j < d_i$

- Claim: Swapping i and j gives a schedule with
  1. Fewer inversions
  2. Maximum lateness no larger than in $\mathcal{O}'$

Earliest Deadline is Optimal

Claim: Swapping i and j: maximum lateness no larger than in $\mathcal{O}'$

Exchange Argument

- General approach that often works to analyze greedy algorithms

- Start with any solution
- Define basic exchange step that allows to transform solution into a new solution that is not worse
- Show that exchange step move solution closer to the solution produced by the greedy algorithm
- Number of exchange steps to reach greedy solution should be finite…

Another Exchange Argument Example

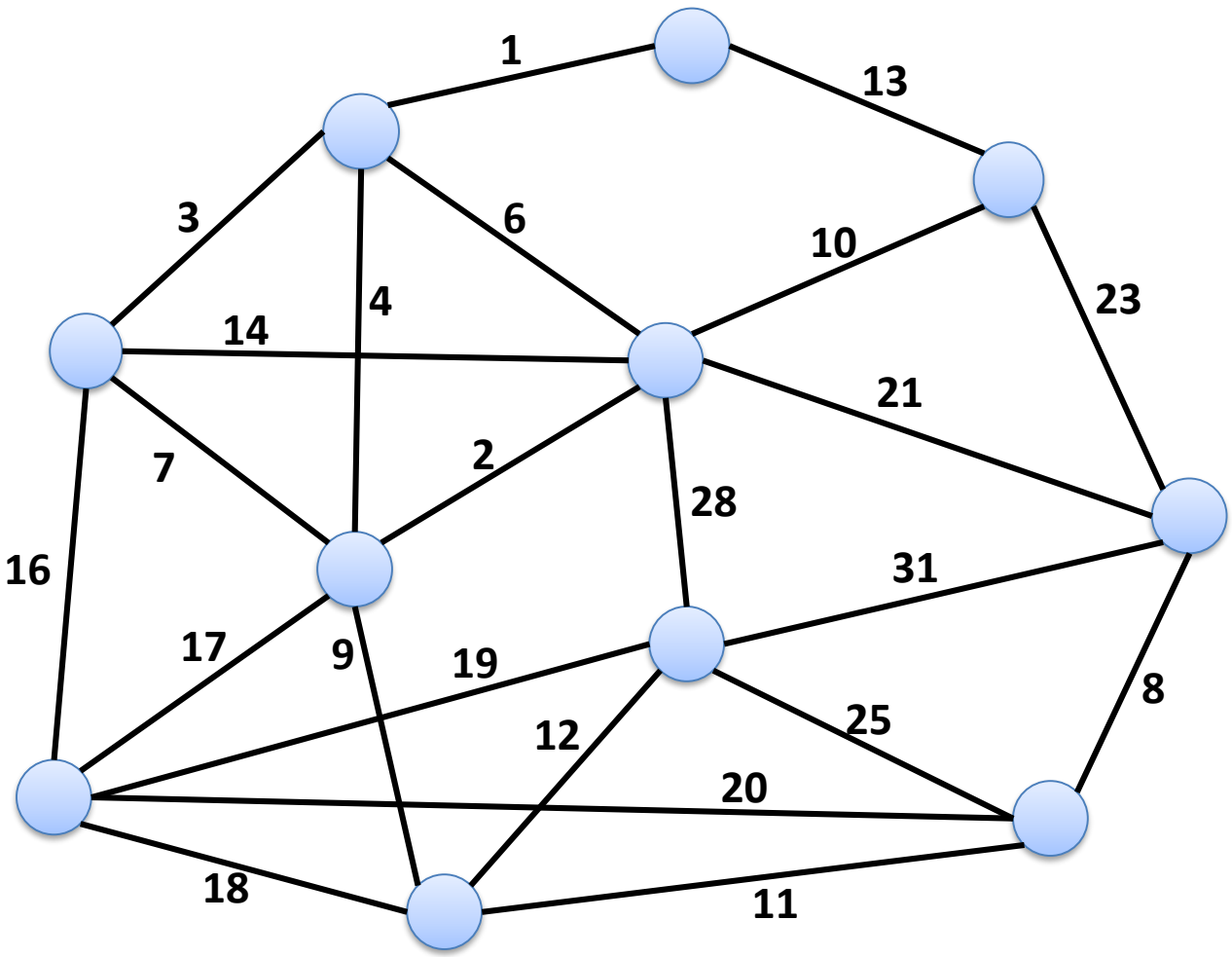
- **Minimum spanning tree (MST)** problem
  - Classic graph-theoretic optimization problem

- **Given**: weighted graph
- **Goal**: spanning tree with min. total weight

- Several greedy algorithms work

- Kruskal's algorithm:
  - Start with empty edge set
  - As long as we do not have a spanning tree:
    add minimum weight edge that doesn't close a cycle

Kruskal Algorithm: Example



Kruskal is Optimal

- Basic exchange step: swap to edges to get from tree T to tree T'
  - Swap out edge not in Kruskal tree, swap in edge in Kruskal tree
  - Swapping does not increase total weight
- For simplicity, assume, weights are unique:

Matroids

- Same, but more abstract…

Matroid: pair $(\boldsymbol{E}, \boldsymbol{I})$

- $\boldsymbol{E}$: set, called the **ground set**
- $\boldsymbol{I}$: finite family of finite subsets of E (i.e., $I \subseteq 2^E$), called **independent sets**

(E, I) needs to satisfy 3 properties:

1. Empty set is independent, i.e., $\emptyset \in \boldsymbol{I}$ (implies that $I \neq \emptyset$)
2. **Hereditary property**: For all $A \subseteq E$ and all $A' \subseteq A$,

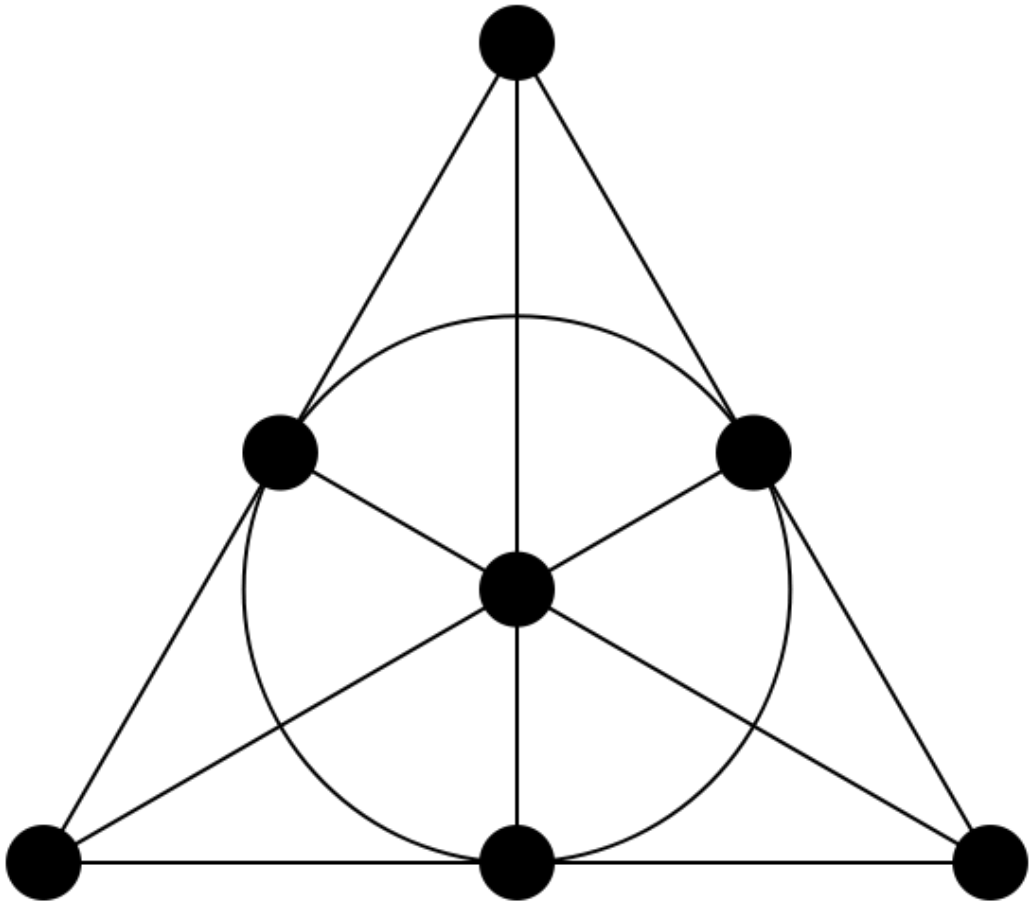
   if $\boldsymbol{A} \in \boldsymbol{I}$, then also $\boldsymbol{A}' \in \boldsymbol{I}$

3. **Augmentation / Independent set exchange property:**
   If $A, B \in I$ and $|A| > |B|$, there exists $x \in A \setminus B$ such that

$$\mathbf{B}' := \boldsymbol{B} \cup \{\boldsymbol{x}\} \in \boldsymbol{I}$$

Example

- Fano matroid:
  - Smallest finite projective plane of order 2…



Matroids and Greedy Algorithms

Weighted matroid: each $e \in E$ has a weight $w(e) > 0$

Goal: find maximum weight independent set

Greedy algorithm:

1. Start with $S = \emptyset$
2. Add max. weight $e \in E \setminus S$ to S such that $S \cup \{e\} \in I$

Claim: greedy algorithm computes optimal solution

Greedy is Optimal

- S: greedy solution A: any other solution

Matroids: Examples

Forests of a graph $\boldsymbol{G = (V, E)}$:

- forest F: subgraph with no cycles (i.e., $F \subseteq E$)
- $\mathcal{F}$: set of all forests → $(E, \mathcal{F})$ is a matroid
- Greedy algorithm gives maximum weight forest (equivalent to MST problem)

Bicircular matroid of a graph $\boldsymbol{G = (V, E)}$:

- $\mathcal{B}$: set of edges such that every connected subset has ≤ 1 cycle
- $(E, \mathcal{B})$ is a matroid → greedy gives max. weight such subgraph

Linearly independent vectors:

- Vector space V, E: finite set of vectors, I: sets of lin. indep. vect.
- Fano matroid can be defined like that

Chapter 2
Greedy Algorithms

Algorithm Theory
WS 2018/19

Fabian Kuhn

Matroids

- Same, but more abstract…

Matroid: pair $(\boldsymbol{E}, \boldsymbol{I})$

- $\boldsymbol{E}$: set, called the **ground set**
- $\boldsymbol{I}$: finite family of finite subsets of E (i.e., $I \subseteq 2^E$), called **independent sets**

(E, I) needs to satisfy 3 properties:

1. Empty set is independent, i.e., $\emptyset \in \boldsymbol{I}$ (implies that $I \neq \emptyset$)
2. **Hereditary property**: For all $A \subseteq E$ and all $A' \subseteq A$,

   if $\boldsymbol{A} \in \boldsymbol{I}$, then also $\boldsymbol{A}' \in \boldsymbol{I}$

3. **Augmentation / Independent set exchange property:** If $A, B \in I$ and $|A| > |B|$, there exists $x \in A \setminus B$ such that

$$\mathbf{B}' := \boldsymbol{B} \cup \{\boldsymbol{x}\} \in \boldsymbol{I}$$

Matroids and Greedy Algorithms

Weighted matroid: each $e \in E$ has a weight $w(e) > 0$

Goal: find maximum weight independent set

Greedy algorithm:

1. Start with $S = \emptyset$
2. Add max. weight $e \in E \setminus S$ to S such that $S \cup \{e\} \in I$

Claim: greedy algorithm computes optimal solution

Greedy is Optimal

- S: greedy solution A: any other solution

Matroids: Examples

Forests of a graph $\boldsymbol{G = (V, E)}$:

- forest F: subgraph with no cycles (i.e., $F \subseteq E$)
- $\mathcal{F}$: set of all forests → $(E, \mathcal{F})$ is a matroid
- Greedy algorithm gives maximum weight forest (equivalent to MST problem)

Bicircular matroid of a graph $\boldsymbol{G = (V, E)}$:

- $\mathcal{B}$: set of edges such that every connected subset has ≤ 1 cycle
- $(E, \mathcal{B})$ is a matroid → greedy gives max. weight such subgraph

Linearly independent vectors:

- Vector space V, E: finite set of vectors, I: sets of lin. indep. vect.
- Fano matroid can be defined like that

Forest Matroid of Graph $G = (V, E)$

Ground set: E (edges) **Independent sets:** $\mathcal{F}$ (forests of G)

Basic properties: $\emptyset \in \mathcal{F}$ + hereditary property

- Empty graph has no cycles, removing edges doesn't create cycles

Independent set exchange property:

- Given $\mathcal{F}_1, \mathcal{F}_2$ s.t. $|\mathcal{F}_1| > |\mathcal{F}_2|$
- $\mathcal{F}_1$ needs to have an edge e connecting two components of $\mathcal{F}_2$
  - Because it can only have $|\mathcal{F}_2|$ edges connecting nodes inside components

Bicircular Matroid

Bicircular Matroid

Greedoid

- Matroids can be generalized even more

- Relax hereditary property:

  Replace $A' \subseteq A \subseteq I \implies A' \in I$

  by $\emptyset \neq A \subseteq I \implies \exists a \in A, \text{ s.t. } A \setminus \{a\} \in I$

- Augmentation property holds as before

- Under certain conditions on the weights, greedy is optimal for computing the max. weight $A \in I$ of a greedoid.
  - Additional conditions automatically satisfied by hereditary property

- More general than matroids

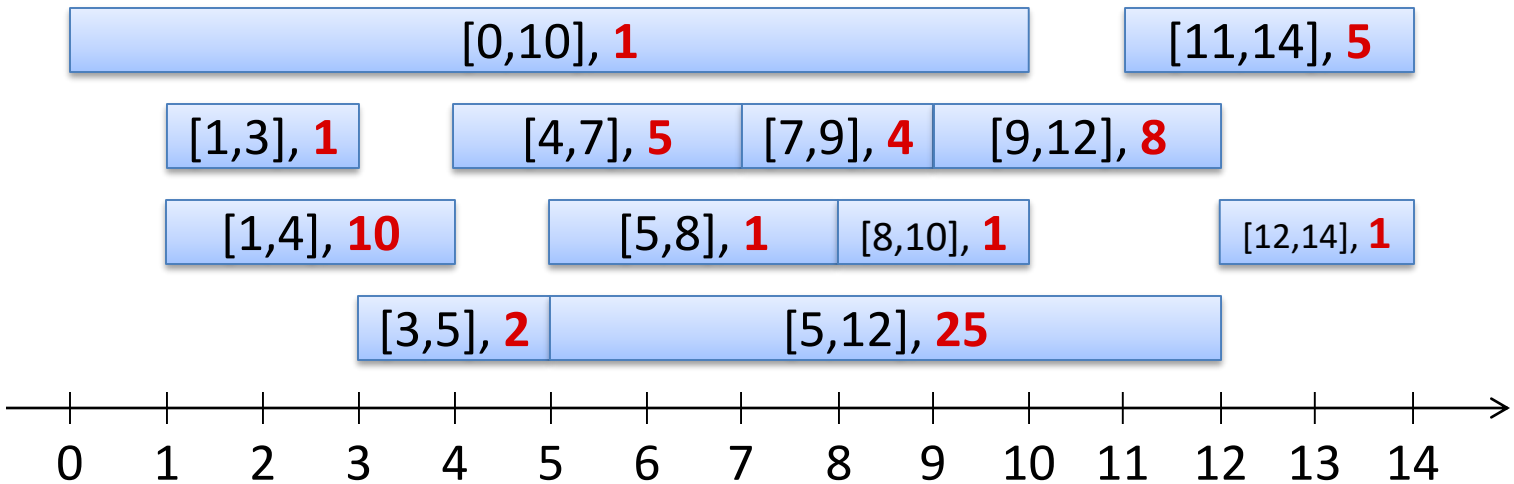
Chapter 3
Dynamic Programming

Algorithm Theory
WS 2018/19

Fabian Kuhn

Weighted Interval Scheduling

- **Given:** Set of intervals, e.g.
  [0,10],[1,3],[1,4],[3,5],[4,7],[5,8],[5,12],[7,9],[9,12],[8,10],[11,14],[12,14]
- Each interval has a **weight *w***

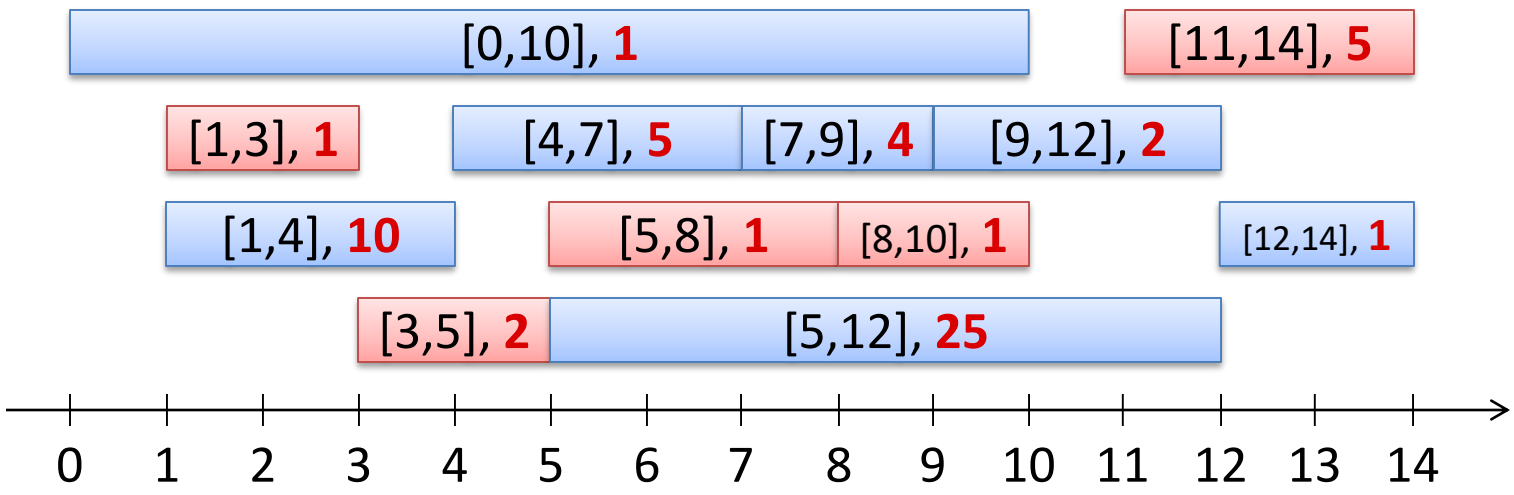


- **Goal:** Non-overlapping set of intervals of largest possible weight
  - Overlap at boundary ok, i.e., [4,7] and [7,9] are non-overlapping

- **Example:** Intervals are room requests of different importance

Greedy Algorithms

Choose available request with earliest finishing time:



- Algorithm is not optimal any more
  - It can even be arbitrarily bad…

- No greedy algorithm known that works

Solving Weighted Interval Scheduling

- Interval i: start time $s(i)$, finishing time: $f(i)$, weight: $w(i)$

- Assume intervals $1, \dots, n$ are sorted by increasing $f(i)$
  - $0 < f(1) \leq f(2) \leq \cdots \leq f(n)$, for convenience: $f(0) = 0$

- Simple observation:
  Opt. solution contains interval n or it doesn't contain interval n

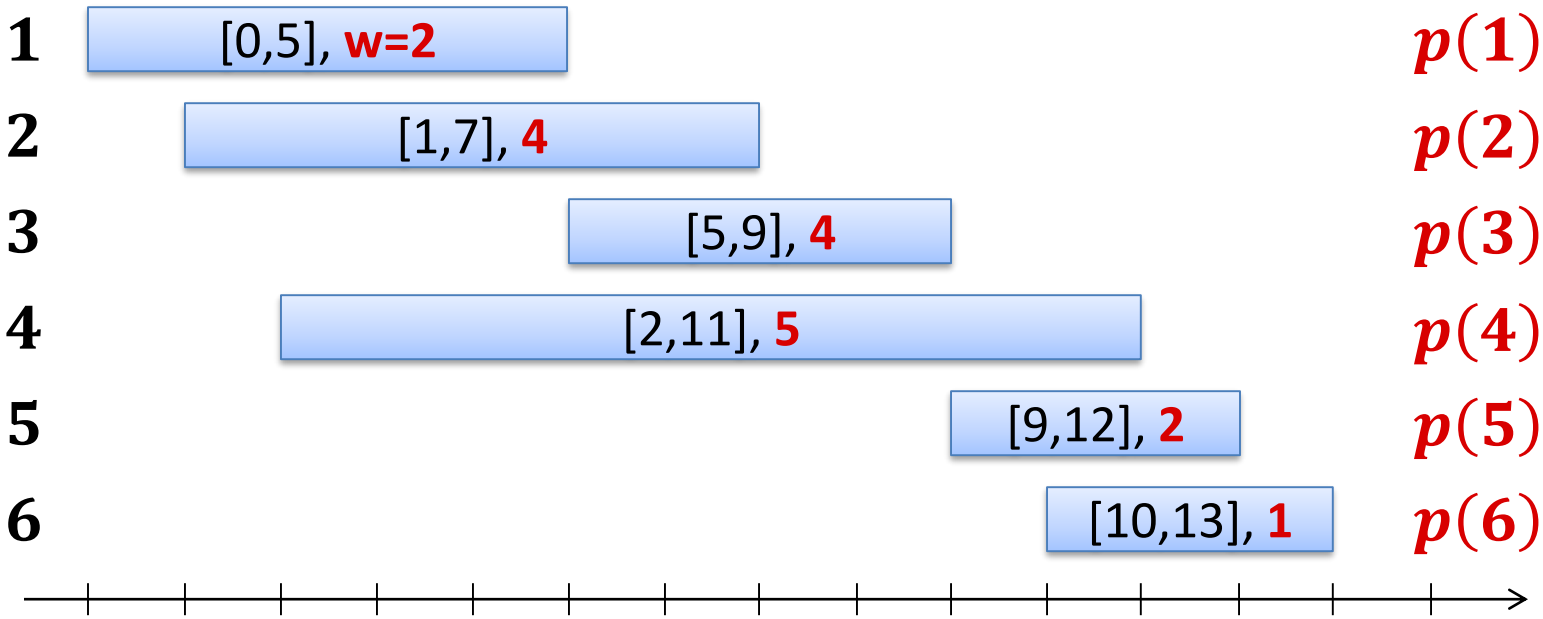
Solving Weighted Interval Scheduling

- Interval i: start time $s(i)$, finishing time: $f(i)$, weight: $w(i)$
- Assume intervals $1, \dots, n$ are sorted by increasing $f(i)$
  - $0 < f(1) \leq f(2) \leq \cdots \leq f(n)$, for convenience: $f(0) = 0$
- Simple observation:
  Opt. solution contains interval n or it doesn't contain interval n
- Weight of optimal solution for only intervals $1, \dots, k$: $W(k)$
  Define $p(k) := \max\{i \in \{0, \dots, k-1\} : f(i) \leq s(k)\}$
- Opt. solution does not contain interval n: $\boldsymbol{W(n) = W(n-1)}$

  Opt. solution contains interval n: $\boldsymbol{W(n) = w(n) + W(p(n))}$

Example

Interval:



$p(1) = 0$

$p(2) = 0$

$p(3) = 1$

$p(4) = 0$

$p(5) = 3$

$p(6) = 3$

Recursive Definition of Optimal Solution

- Recall:
  - $W(k)$: weight of optimal solution with intervals $1, \dots, k$
  - $p(k)$: last interval to finish before interval k starts

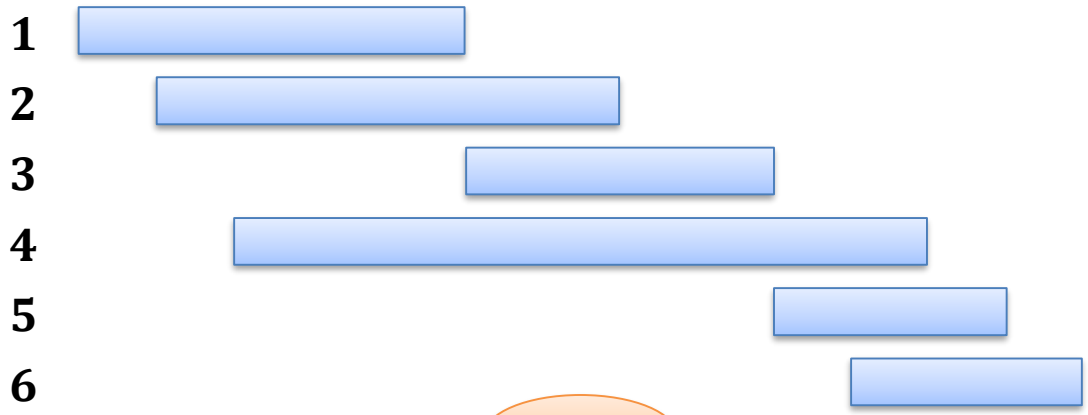
- Recursive definition of optimal weight:

$$\forall k > 1\colon\ W(k) = \max\{W(k-1), w(k) + W(p(k))\}$$
$$W(1) = w(1)$$

Immediately gives a simple, recursive algorithm

```
Compute p(k) values for all k
W(k):
    if k == 1:
        x = w(1)
    else:
        x = max{W(k-1), w(k) + W(p(k))}
    return x
```

Running Time of Recursive Algorithm



$p(1) = 0$

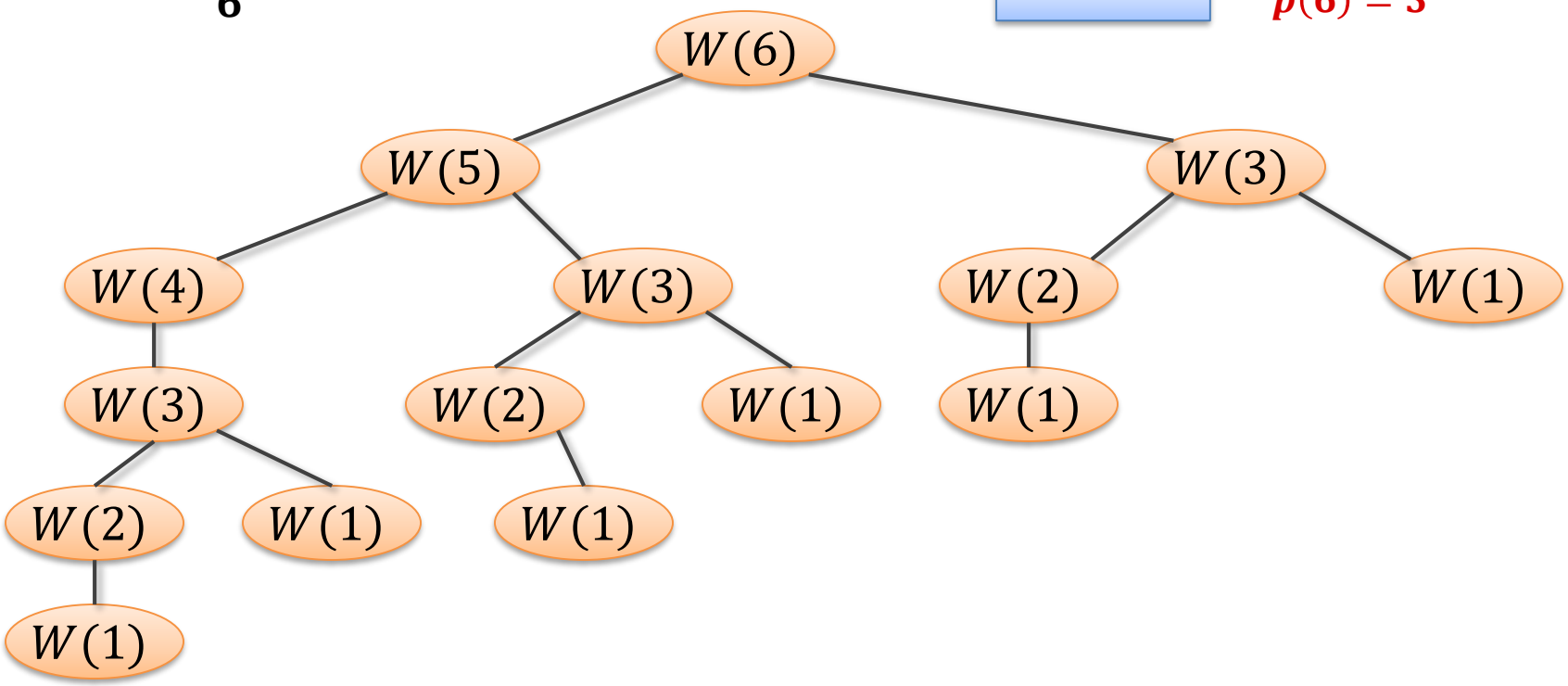
$p(2) = 0$

$p(3) = 1$

$p(4) = 0$

$p(5) = 3$

$p(6) = 3$



Memoizing the Recursion

- Running time of recursive algorithm: exponential!
- But, alg. only solves n different sub-problems: $W(1), \dots, W(n)$
- There is no need to compute them multiple times

Memoization: Store already computed values for future rec. calls

```
Compute p(k) for all k
memo = {};
W(k):
    if k in memo: return memo[k]
    if k == 1:
        x = w(1)
    else:
        x = max{W(k-1), w(k) + W(p(k))}
    memo[k] = x
    return x
```

Dynamic Programming (DP)

DP $\approx$ Recursion + Memoization

Recursion: Express problem *recursively* in terms of
(a 'small' number of) *subproblems* (of the same kind)

Memoize: *Store* solutions for *subproblems*
reuse the stored solutions if the same subproblems
has to be solved again

Weighted interval scheduling: subproblems $W(1), W(2), W(3), \ldots$

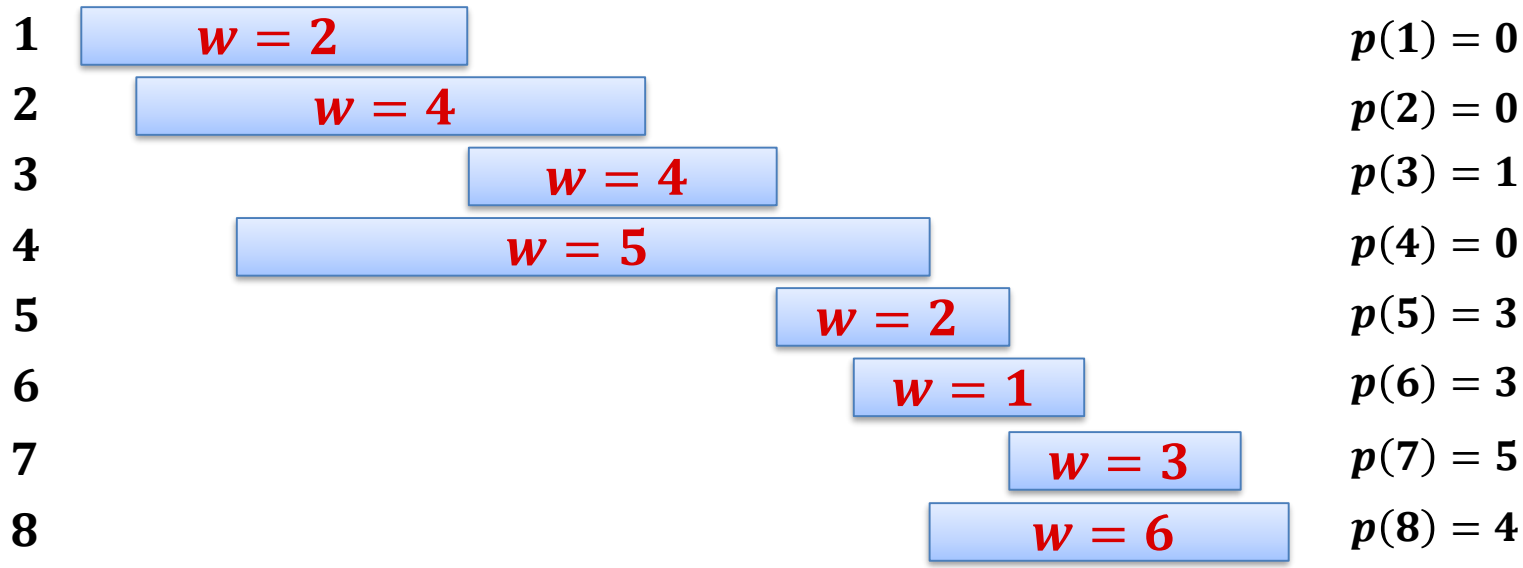
runtime $=$ #subproblems $\cdot$ time per subproblem

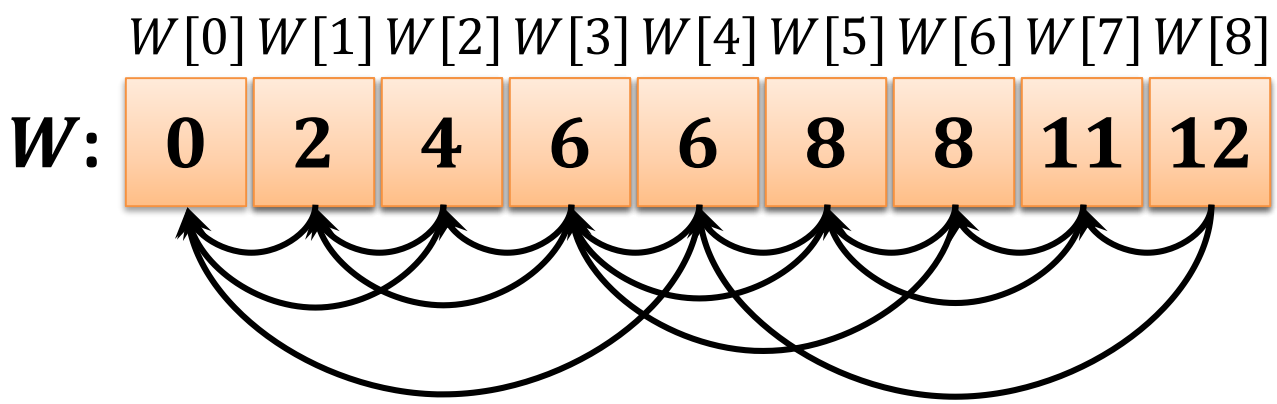
DP: Some History …

- Where das does the name come from?
- DP was developed by Richard E. Bellman in 1940s/1950s.
- In his autobiography, it says:

  "I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. … The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. … His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. … Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. … It also has a very interesting property as an adjective, and that it's impossible to use the word dynamic in a pejorative sense. … Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. …"

Example





Computing the schedule: **store where you come from!**

Dynamic Programming

„Memoization“ for increasing the efficiency of a recursive solution:

- Only the *first time* a sub-problem is encountered, its solution is computed and then stored in a table. Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned (without repeated computation!).

- *Computing the solution*: For each sub-problem, store how the value is obtained (according to which recursive rule).

Dynamic Programming

Dynamic programming / memoization can be applied if

- Optimal solution contains optimal solutions to sub-problems (recursive structure)
- Number of sub-problems that need to be considered is small

Matrix-chain multiplication

Given: sequence (chain) $\langle A_1, A_2, \dots, A_n \rangle$ of matrices

Goal: compute the product $A_1 \cdot A_2 \cdot \dots \cdot A_n$

Problem: Parenthesize the product in a way that minimizes the number of scalar multiplications.

Definition: A product of matrices is *fully parenthesized* if it is

- a single matrix
- or the product of two fully parenthesized matrix products, surrounded by parentheses.

Example

All possible fully parenthesized matrix products of the chain $\langle A_1, A_2, A_3, A_4 \rangle$:

$$(A_1 (A_2 (A_3 A_4)))$$

$$(A_1 ((A_2 A_3) A_4))$$

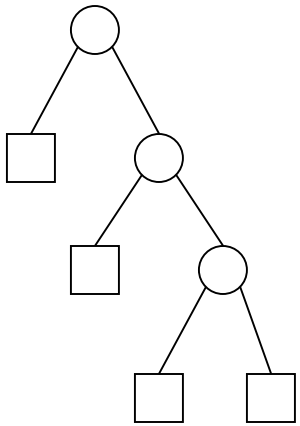
$$((A_1 A_2)(A_3 A_4))$$

$$((A_1 (A_2 A_3)) A_4)$$

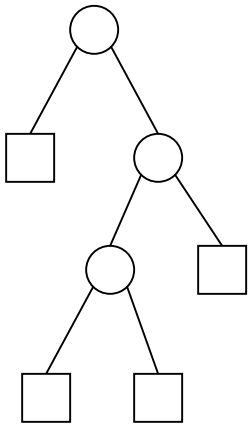
$$(((A_1 A_2) A_3) A_4)$$

Different parenthesizations

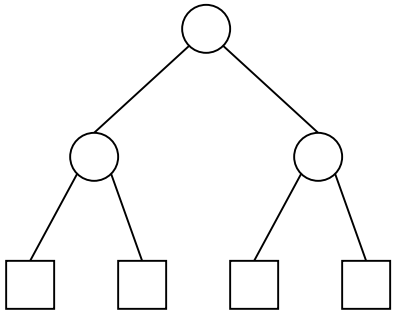
Different parenthesizations correspond to different trees:



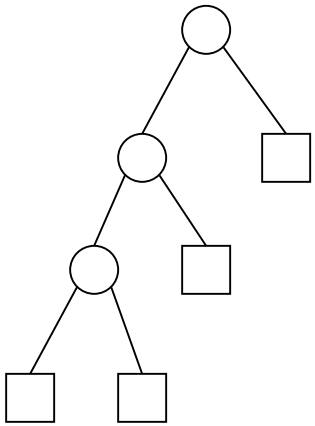
$$\big(A_1\big(A_2(A_3A_4)\big)\big)$$



$$\big(A_1\big((A_2A_3)A_4\big)\big)$$



$$\big((A_1A_2)(A_3A_4)\big)$$



$$\big(\big((A_1A_2)A_3\big)A_4\big)$$

Number of different parenthesizations

- Let $P(n)$ be the number of alternative parenthesizations of the product $A_1 \cdot \ldots \cdot A_n$:

$$P(1) = 1$$

$$P(n) = \sum_{k=1}^{n-1} P(k) \cdot P(n-k)\,, \qquad \text{for } n \geq 2$$

$$P(n+1) = \frac{1}{n+1}\binom{2n}{n} \approx \frac{4^n}{n\sqrt{\pi n}} + O\left(\frac{4^n}{\sqrt{n^5}}\right)$$

$$P(n+1) = C_n \qquad (n^{th} \text{ Catalan number})$$

- Thus: Exhaustive search needs exponential time!

Multiplying Two Matrices

$$A = (a_{ij})_{p\times q}, \qquad B = (b_{ij})_{q\times r}, \qquad A \cdot B = C = (c_{ij})_{p\times r}$$

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}$$

Algorithm *Matrix-Mult*

Input: $(p \times q)$ matrix A, $(q \times r)$ matrix B

Output: $(p \times r)$ matrix $C = A \cdot B$

```
for i := 1 to p do
    for j := 1 to r do
        C[i,j] := 0;
        for k := 1 to q do
            C[i,j] := C[i,j] + A[i,k] · B[k,j]
```

Remark:

Using this algorithm, multiplying two $(n \times n)$ matrices requires n^3 multiplications. This can also be done using $O(n^{2.373})$ multiplications.

Number of multiplications and additions: $\boldsymbol{p \cdot q \cdot r}$

Matrix-chain multiplication: Example

Computation of the product $A_1 A_2 A_3$, where

A_1 : (50×5) matrix

A_2 : (5×100) matrix

A_3 : (100×10) matrix

a) Parenthesization $((A_1 A_2)A_3)$ and $\left(A_1(A_2 A_3)\right)$ require:

$A' = (A_1 A_2)$:

$A'' = (A_2 A_3)$:

$A' A_3$:

$A_1 A''$:

Sum:

Structure of an Optimal Parenthesization

- $(A_{\ell \dots r})$: optimal parenthesization of $A_\ell \cdot \dots \cdot A_r$

  For some $1 \leq k < n$: $\boldsymbol{(A_{1\dots n}) = \big((A_{1\dots k}) \cdot (A_{k+1\dots n})\big)}$

- Any optimal solution contains optimal solutions for sub-problems

- Assume matrix A_i is a $(d_{i-1} \times d_i)$-matrix

- Cost to solve sub-problem $A_\ell \cdot \dots \cdot A_r, \ell \leq r$ optimally: $C(\ell, r)$

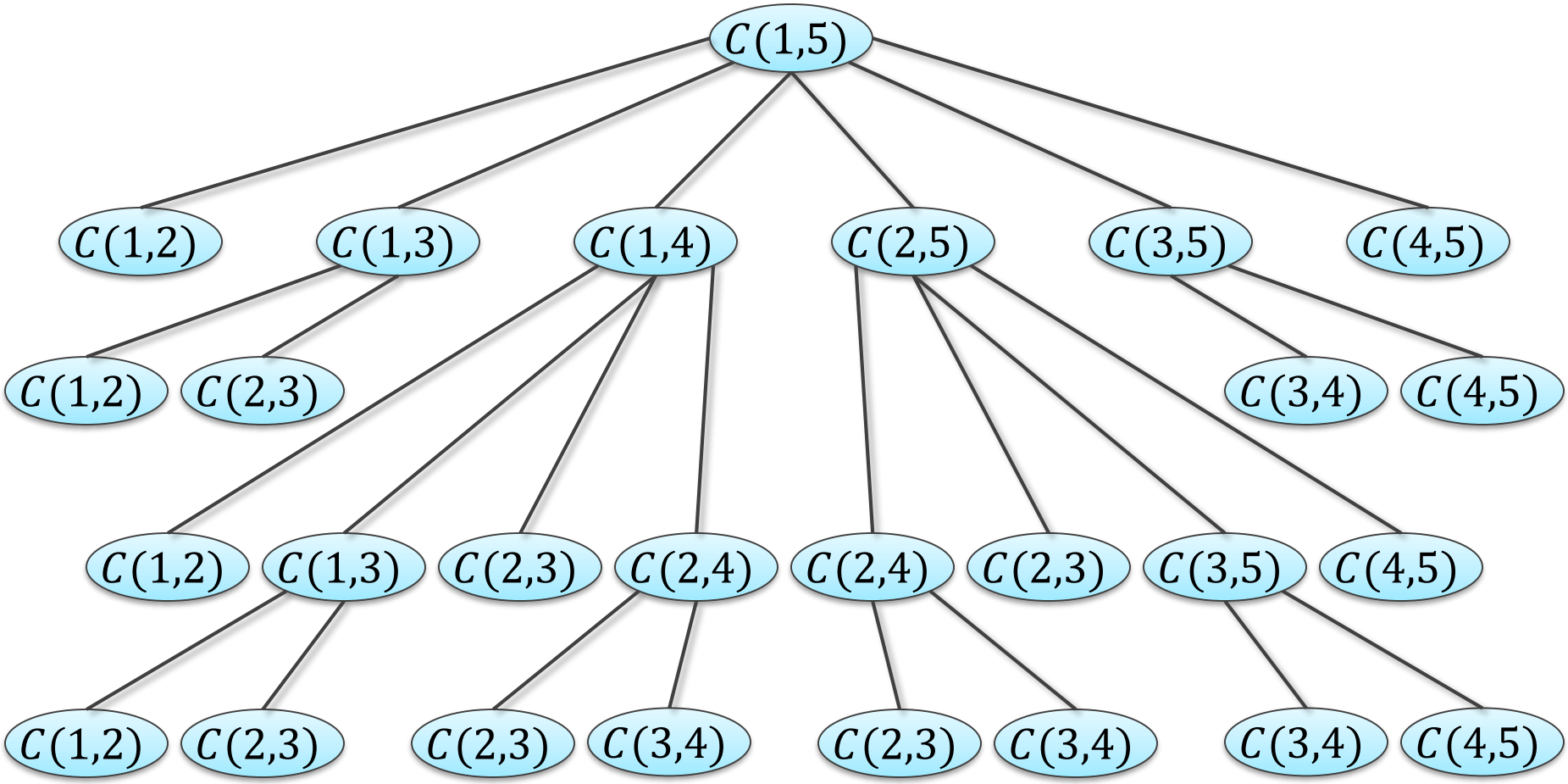
- Then:

$$\boldsymbol{C(a,b) = \min_{a \leq k < b} C(a,k) + C(k+1,b) + d_{a-1} d_k d_b}$$

$$\boldsymbol{C(a,a) = 0}$$

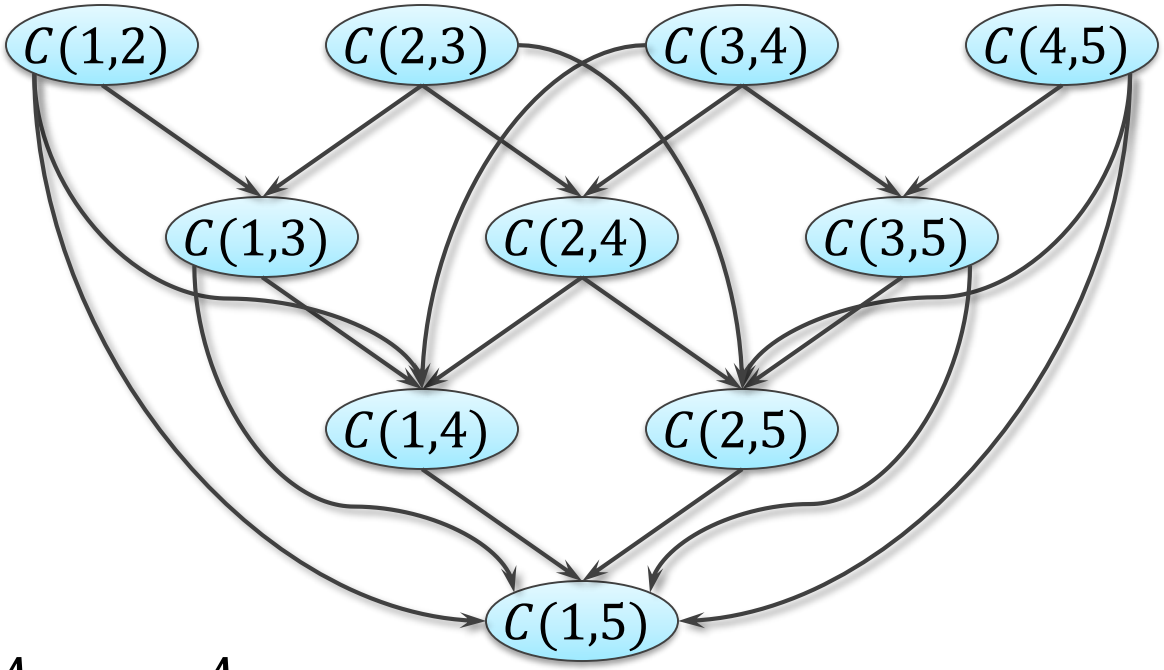
Recursive Computation of Opt. Solution

Compute $A_1 \cdot A_2 \cdot A_3 \cdot A_4 \cdot A_5$:



Using Meomization

Compute $A_1 \cdot A_2 \cdot A_3 \cdot A_4 \cdot A_5$:



Compute $A_1 \cdot \ldots \cdot A_n$:

- Each $C(i,j)$, $i < j$ is computed exactly once → $O(n^2)$ values
- Each $C(i,j)$ dir. depends on $C(i,k)$, $C(k,j)$ for $i < k < j$

  Cost for each $C(i,j)$: $O(n)$ → overall time: $\boldsymbol{O(n^3)}$

Remarks about matrix-chain multiplication

1. There is an algorithm that determines an optimal parenthesization in time

$$O(n \cdot \log n).$$

2. There is a linear time algorithm that determines a parenthesization using at most

$$1.155 \cdot C(1, n)$$

multiplications.

Chapter 3
Dynamic Programming

Algorithm Theory
WS 2018/19

Fabian Kuhn

Dynamic Programming (DP)

DP $\approx$ Recursion + Memoization

Recursion: Express problem *recursively* in terms of
(a 'small' number of) *subproblems* (of the same kind)

Memoize: *Store* solutions for *subproblems*
reuse the stored solutions if the same subproblems
has to be solved again

Weighted interval scheduling: subproblems $W(1), W(2), W(3), \ldots$

runtime $=$ #subproblems $\cdot$ time per subproblem

Dynamic Programming

„Memoization“ for increasing the efficiency of a recursive solution:

- Only the *first time* a sub-problem is encountered, its solution is computed and then stored in a table. Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned (without repeated computation!).

- *Computing the solution*: For each sub-problem, store how the value is obtained (according to which recursive rule).

Dynamic Programming

Dynamic programming / memoization can be applied if

- Optimal solution contains optimal solutions to sub-problems (recursive structure)
- Number of sub-problems that need to be considered is small

Knapsack

- n items $1, \dots, n$, each item has weight w_i and value v_i
- Knapsack (bag) of capacity W

- Goal: pack items into knapsack such that total weight is at most W and total value is maximized:

$$\max \sum_{i \in S} v_i$$

$$\text{s.t.} \quad S \subseteq \{1, \dots, n\} \text{ and } \sum_{i \in S} w_i \leq W$$

- E.g.: jobs of length w_i and value v_i, server available for W time units, try to execute a set of jobs that maximizes the total value

Recursive Structure?

- Optimal solution: $\mathcal{O}$
- If $n \notin \mathcal{O}$: $\mathrm{OPT}(n) = \mathrm{OPT}(n-1)$
- What if $n \in \mathcal{O}$?
  - Taking n gives value v_n
  - But, n also occupies space w_n in the bag (knapsack)
  - There is space for $W - w_n$ total weight left!

$$\mathrm{OPT}(n) = v_n + \text{optimal solution with first } n-1 \text{ items and knapsack of capacity } W - w_n$$

A More Complicated Recursion

$\mathbf{OPT}(\boldsymbol{k}, \boldsymbol{x})$: value of optimal solution with items $1, \dots, k$
and knapsack of capacity x

Recursion:

Dynamic Programming Algorithm

Set up table for all possible $\mathrm{OPT}(k, x)$-values

- Assume that all weights w_i are integers!

	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{2}$	$\mathbf{3}$	$\cdots$							$\boldsymbol{W}$
$\mathbf{0}$												
$\mathbf{1}$												
$\mathbf{2}$												
$\mathbf{3}$												
$\vdots$												
$\boldsymbol{n}$												

Row $\boldsymbol{i}$, column $\boldsymbol{j}$:

$$\boldsymbol{OPT(i, j)}$$

Example

- 8 items: (3,2), (2,4), (4,1), (5,6), (3,3), (4,3), (5, 4), (6,6)
  Knapsack capacity: 12

  weight value

- $OPT(k, x) = \max\{OPT(k-1, x), OPT(k-1, x - w_k) + v_k\}$

	1	2	3	4	5	6	7	8	9	10	11	12
1												
2												
3												
4												
5												
6												
7												
8												

Running Time of Knapsack Algorithm

- **Size of table:** $O(n \cdot W)$
- Time per table entry: $O(1)$ → **overall time:** $\boldsymbol{O(nW)}$
- Computing solution (set of items to pick):
  Follow $\leq n$ arrows → $O(n)$ time (after filling table)
- Note: Time depends on W → can be exponential in n…
- And it is problematic if weights are not integers.

String Matching Problems

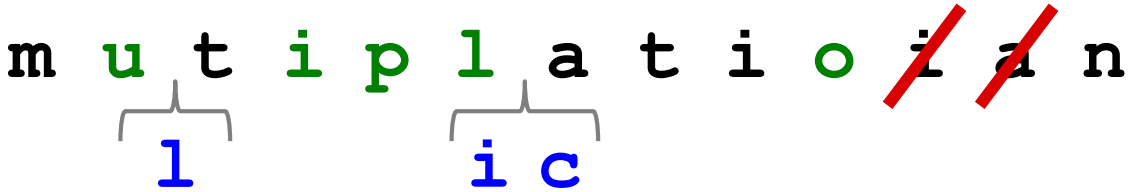
Edit distance:

- For two given strings A and B, efficiently compute the

  edit distance $D(A,B)$ (# edit operations to transform A into B)

  as well as a minimum sequence of edit operations that transform A into B.

- **Example:** mathematician → multiplication:



Edit Distance

Given: Two strings $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$

Goal: Determine the minimum number $D(A, B)$ of edit operations required to transform A into B

Edit operations:

a) **Replace** a character from string A by a character from B

b) **Delete** a character from string A

c) **Insert** a character from string B into A

```
m a - t h e m - - a t i c i a n
m u l t i p l i c a t i o - - n
```

Edit Distance – Cost Model

- Cost for **replacing** character a by b: $\boldsymbol{c(a,b) \geq 0}$

- Capture insert, delete by allowing $a = \varepsilon$ or $b = \varepsilon$:
  - Cost for **deleting** character a: $\boldsymbol{c(a,\varepsilon)}$
  - Cost for **inserting** character b: $\boldsymbol{c(\varepsilon,b)}$

- **Triangle inequality**:

$$c(a,c) \leq c(a,b) + c(b,c)$$

→ each character is changed at most once!

- **Unit cost model**: $c(a,b) = \begin{cases} 1, & \text{if } a \neq b \\ 0, & \text{if } a = b \end{cases}$

Recursive Structure

- Optimal "alignment" of strings (unit cost model)

  `bbcadfagikccm` and `abbagflrgikacc`:

```
- b b c a g f a - g i k - c c m
a b b - a d f l r g i k a c c -
```

- Consists of optimal "alignments" of sub-strings, e.g.:

```
-bbcagfa        -gik-ccm
abb-adfl   and  rgikacc-
```

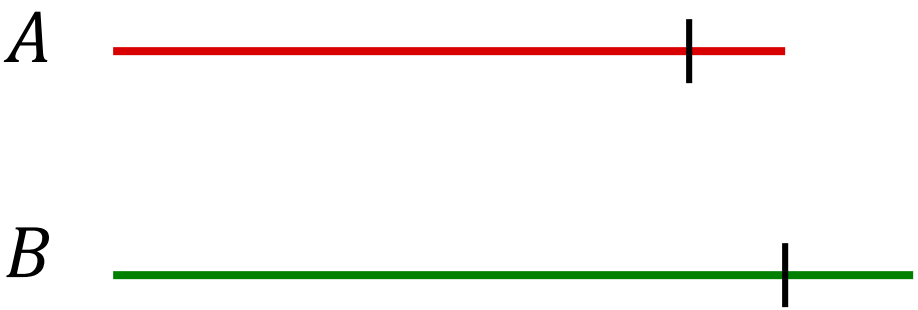
- Edit distance between $A_{1,m} = a_1 \dots a_m$ and $B_{1,n} = b_1 \dots b_n$:

$$D(A,B) = \min_{k,\ell}\left\{D\left(A_{1,k}, B_{1,\ell}\right) + D\left(A_{k+1,m}, B_{\ell+1,n}\right)\right\}$$

Computation of the Edit Distance

Let $A_k \coloneqq a_1 \dots a_k$, $B_\ell \coloneqq b_1 \dots b_\ell$, and

$$D_{k,\ell} \coloneqq D(A_k, B_\ell)$$



Computation of the Edit Distance

Three ways of ending an "alignment" between A_k and B_ℓ:

1. a_k is replaced by b_ℓ:

   $$D_{k,\ell} = D_{k-1,\ell-1} + c(a_k, b_\ell)$$

2. a_k is deleted:

   $$D_{k,\ell} = D_{k-1,\ell} + c(a_k, \varepsilon)$$

3. b_ℓ is inserted:

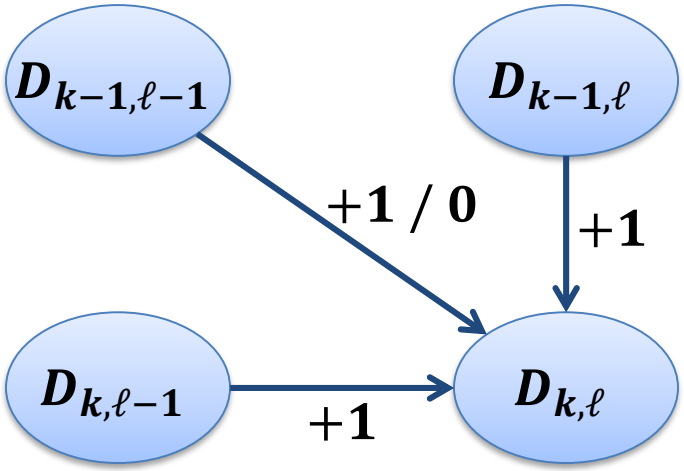
   $$D_{k,\ell} = D_{k,\ell-1} + c(\varepsilon, b_\ell)$$

Computing the Edit Distance

- Recurrence relation (for $k, \ell \geq 1$)

$$D_{k,\ell} = \min \begin{Bmatrix} D_{k-1,\ell-1} & + c(a_k, b_\ell) \\ D_{k-1,\ell} & + c(a_k, \varepsilon) \\ D_{k,\ell-1} & + c(\varepsilon, b_\ell) \end{Bmatrix} = \underbrace{\min \begin{Bmatrix} D_{k-1,\ell-1} & + 1 \,/\, 0 \\ D_{k-1,\ell} & + 1 \\ D_{k,\ell-1} & + 1 \end{Bmatrix}}_{\textbf{unit cost model}}$$

- Need to compute $D_{i,j}$ for all $0 \leq i \leq k$, $0 \leq j \leq \ell$:



Recurrence Relation for the Edit Distance

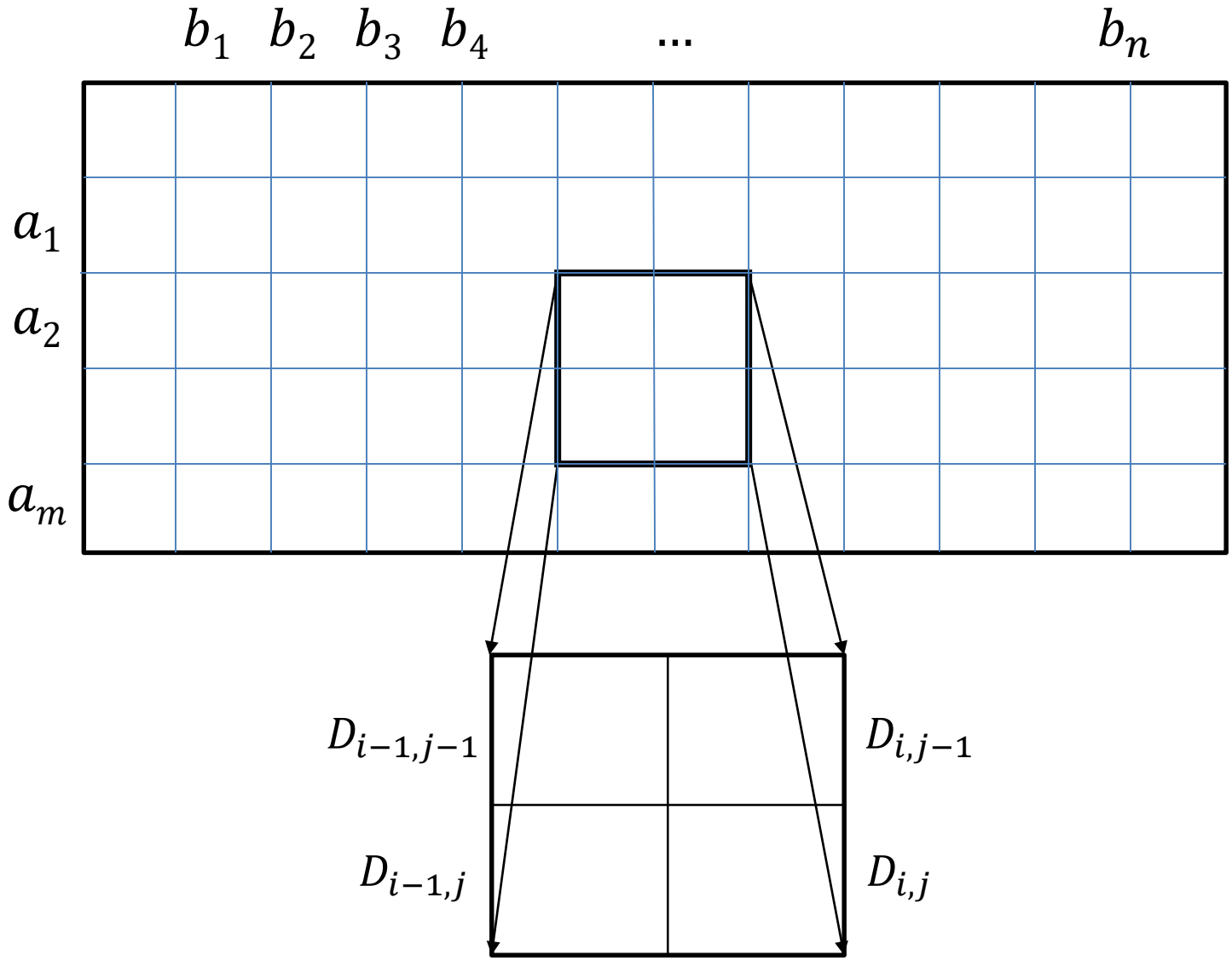
Base cases:

$$\boldsymbol{D_{0,0} = D(\varepsilon, \varepsilon) = 0}$$
$$\boldsymbol{D_{0,j} = D\left(\varepsilon, B_j\right) = D_{0,j-1} + c\left(\varepsilon, b_j\right)}$$
$$\boldsymbol{D_{i,0} = D(A_i, \varepsilon) = D_{i-1,0} + c(a_i, \varepsilon)}$$

Recurrence relation:

$$\boldsymbol{D_{i,j} = \min \begin{Bmatrix} D_{k-1,\ell-1} & + c(a_k, b_\ell) \\ D_{k-1,\ell} & + c(a_k, \varepsilon) \\ D_{k,\ell-1} & + c(\varepsilon, b_\ell) \end{Bmatrix}}$$

Order of solving the subproblems



Algorithm for Computing the Edit Distance

Algorithm *Edit-Distance*

Input: 2 strings $A = a_1 \dots a_m$ and $B = b_1 \dots b_n$

Output: matrix $D = \left(D_{ij}\right)$

$D[0,0] \coloneqq 0;$

for $i \coloneqq 1$ **to** m **do** $D[i,0] \coloneqq i;$

for $j \coloneqq 1$ **to** n **do** $D[0,j] \coloneqq j;$

for $i \coloneqq 1$ **to** m **do**

  for $j \coloneqq 1$ **to** n **do**

$$D[i,j] \coloneqq \min \begin{Bmatrix} D[i-1,j] & +1 \\ D[i,j-1] & +1 \\ D[i-1,j-1] & + c\left(a_i, b_j\right) \end{Bmatrix};$$

Example

		a	b	c	c	a
b						
a						
b						
d						
a						

Edit Operations

		a	b	c	c	a
	0	1	2	3	4	5
b	1	1	1	2	3	4
a	2	1	2	2	3	3
b	3	2	1	2	3	4
d	4	3	2	2	3	4
a	5	4	3	3	3	3

Computing the Edit Operations

Algorithm *Edit-Operations*(i, j)
Input: matrix D (already computed)
Output: list of edit operations

if $i = 0$ **and** $j = 0$ **then return** empty list

if $i \neq 0$ **and** $D[i, j] = D[i-1, j] + 1$ **then**
  return *Edit-Operations*$(i-1, j) \circ$ „delete a_i“

else if $j \neq 0$ **and** $D[i, j] = D[i, j-1] + 1$ **then**
  return *Edit-Operations*$(i, j-1) \circ$ „insert b_j“

else // $D[i, j] = D[i-1, j-1] + c(a_i, b_j)$
  if $a_i = b_i$ **then return** *Edit-Operations*$(i-1, j-1)$
  else return *Edit-Operations*$(i-1, j-1) \circ$ „replace a_i by b_j“

Initial call: *Edit-Operations*(*m*,*n*)

Edit Operations

		a	b	c	c	a
	0	1	2	3	4	5
b	1	1	1	2	3	4
a	2	1	2	2	3	3
b	3	2	1	2	3	4
d	4	3	2	2	3	4
a	5	4	3	3	3	3

Edit Distance: Summary

- Edit distance between two strings of length m and n can be computed in $O(mn)$ time.

- Obtain the edit operations:
  - for each cell, store which rule(s) apply to fill the cell
  - track path backwards from cell (m, n)
  - can also be used to get all optimal "alignments"

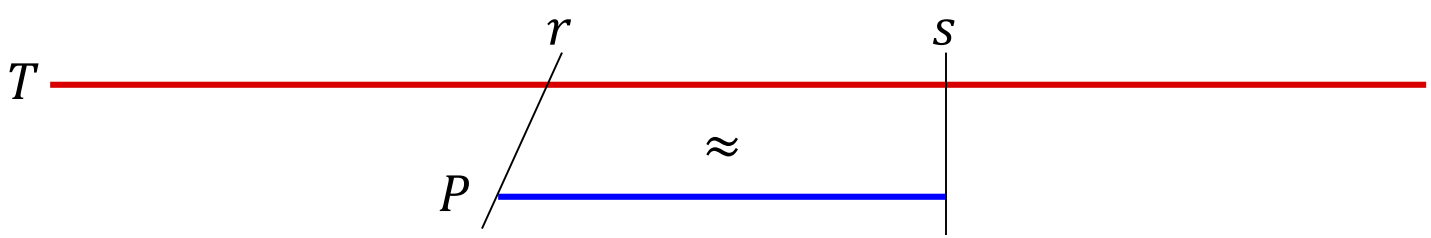
- Unit cost model:
  - interesting special case
  - each edit operation costs 1

Approximate String Matching

Given: strings $T = t_1 t_2 \dots t_n$ (text) and $P = p_1 p_2 \dots p_m$ (pattern).

Goal: Find an interval $[r, s]$, $1 \leq r \leq s \leq n$ such that the sub-string $T_{r,s} := t_r \dots t_s$ is the one with highest similarity to the pattern P:

$$\arg\min_{1 \leq r \leq s \leq n} D\left(T_{r,s}, P\right)$$



Approximate String Matching

Naive Solution:

for all $1 \leq r \leq s \leq n$ **do**

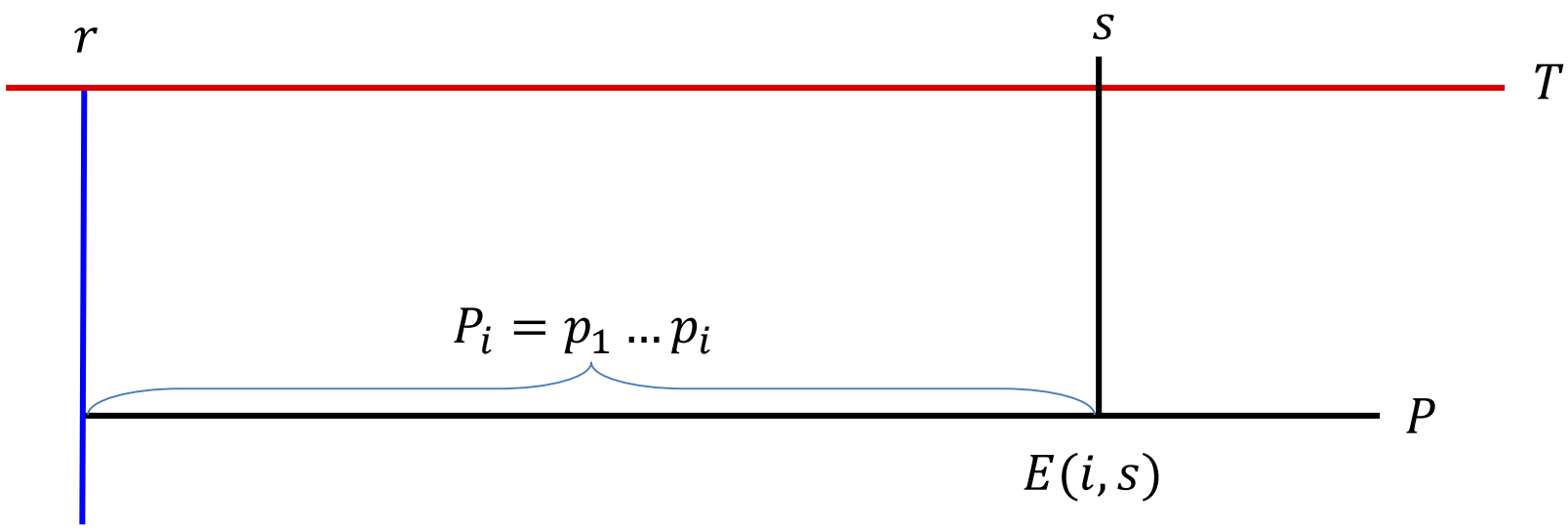
compute $D(T_{r,s}, P)$

choose the minimum

Approximate String Matching

A related problem:

- For each position s in the text and each position i in the pattern compute the minimum edit distance $E(i,s)$ between $P_i = p_1 \dots p_i$ and any substring $T_{r,s}$ of T that ends at position s.



Approximate String Matching

Three ways of ending optimal alignment between T_b and P_i:

1. t_b is replaced by p_i:

   $$E_{b,i} = E_{b-1,i-1} + c(t_b, p_i)$$

2. t_b is deleted:

   $$E_{b,i} = E_{b-1,i} + c(t_b, \varepsilon)$$

3. p_i is inserted:

   $$E_{b,i} = E_{b,i-1} + c(\varepsilon, p_i\ )$$

Approximate String Matching

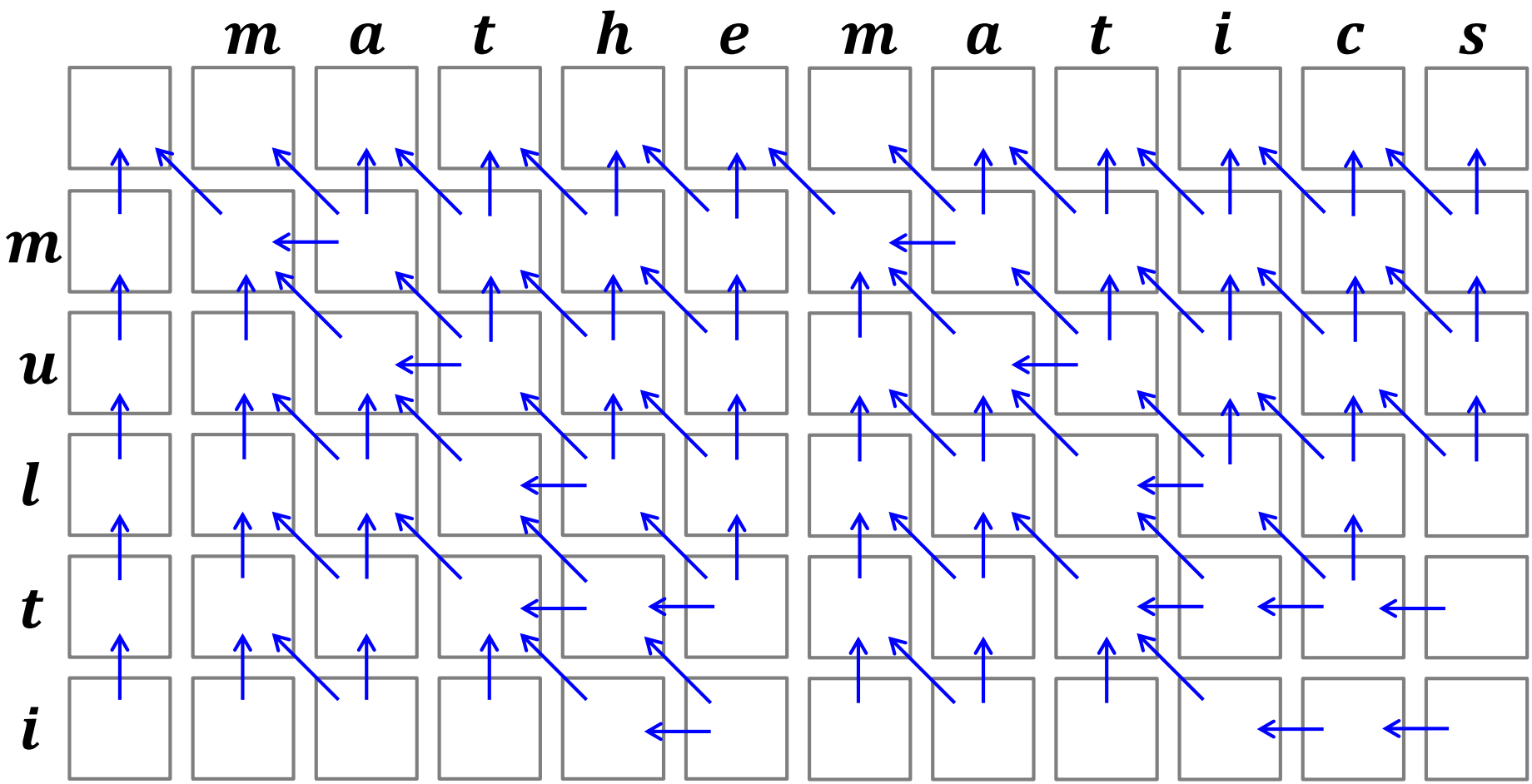
Recurrence relation (unit cost model):

$$E_{b,i} = \min \begin{Bmatrix} E_{b-1,i-1} & +1 \,/\, 0 \\ E_{b-1,i} & +1 \\ E_{b,i-1} & +1 \end{Bmatrix}$$

Base cases:

$$E_{0,0} = 0$$
$$E_{0,i} = i$$
$$E_{i,0} = 0$$

Example



Approximate String Matching

- Optimal matching consists of optimal sub-matchings
- Optimal matching can be computed in $O(mn)$ time
- Get matching(s):
  - Start from minimum entry/entries in bottom row
  - Follow path(s) to top row
- Algorithm to compute $E(b, i)$ identical to edit distance algorithm, except for the initialization of $E(b, 0)$

Related Problems in Bioinformatics

Sequence Alignment:

Find optimal alignment of two given DNA, RNA, or amino acid sequences.

```
G A - C G G A T T A G
G A T C G G A A T - G
```

Global vs. Local Alignment:

- *Global alignment*: find optimal alignment of 2 sequences
- *Local alignment*: find optimal alignment of sequence 1 (patter) with sub-sequence of sequence 2 (text)

Chapter 4
Amortized Analysis

Algorithm Theory
WS 2018/19

Fabian Kuhn

Amortization

- Consider sequence $o_1, o_2, \dots, o_n$ of n operations (typically performed on some data structure D)

- $\boldsymbol{t_i}$: execution time of operation o_i
- $\boldsymbol{T := t_1 + t_2 + \cdots + t_n}$: total execution time

- The execution time of a single operation might

  vary within a large range (e.g., $t_i \in [1, O(i)]$)

- The worst case overall execution time might still be small

  → average execution time per operation might be small in the worst case, even if single operations can be expensive

Analysis of Algorithms

- Best case
- Worst case
- Average case
- Amortized worst case

What is the average cost of an operation in a worst case sequence of operations?

Example 1: Augmented Stack

Stack Data Type: Operations

- $S.\mathrm{push}(x)$: inserts x on top of stack
- $S.\mathrm{pop}()$: removes and returns top element

Complexity of Stack Operations

- In all standard implementations: $O(1)$

Additional Operation

- $\boldsymbol{S}$**.multipop(**$\boldsymbol{k}$**)** : remove and return top k elements
- Complexity: $O(k)$

- What is the amortized complexity of these operations?

Augmented Stack: Amortized Cost

Amortized Cost

- Sequence of operations $i = 1, 2, 3, \dots, n$
- Actual cost of op. i: $\boldsymbol{t_i}$
- Amortized cost of op. i is $\boldsymbol{a_i}$ if for every possible seq. of op.,

$$T = \sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i$$

Actual Cost of Augmented Stack Operations

- $S.\mathrm{push}(x), S.\mathrm{pop}()$: actual cost $t_i = O(1)$
- $S.\mathrm{multipop}(k)$: actual cost $t_i = O(k)$
- Amortized cost of all three operations is constant
  - The total number of "popped" elements cannot be more than the total number of "pushed" elements: **cost for pop/multipop $\leq$ cost for push**

Augmented Stack: Amortized Cost

Amortized Cost

$$T = \sum_i t_i \leq \sum_i a_i$$

Actual Cost of Augmented Stack Operations

- $S.\mathrm{push}(x), S.\mathrm{pop}()$: actual cost $t_i \leq c$
- $S.\mathrm{multipop}(k)$: actual cost $t_i \leq c \cdot k$

Example 2: Binary Counter

Incrementing a binary counter: determine the bit flip cost:

Operation	Counter Value	Cost
	00000	
1	00001	1
2	00010	2
3	00011	1
4	00100	3
5	00101	1
6	00110	2
7	00111	1
8	01000	4
9	01001	1
10	01010	2
11	01011	1
12	01100	3
13	01101	1

Accounting Method

Observation:

- Each increment flips exactly one 0 into a 1

$$0010001111 \Longrightarrow 0010010000$$

Idea:

- Have a bank account (with initial amount 0)
- Paying x to the bank account costs x
- Take "money" from account to pay for expensive operations

Applied to binary counter:

- Flip from 0 to 1: pay 1 to bank account (cost: 2)
- Flip from 1 to 0: take 1 from bank account (cost: 0)
- Amount on bank account = number of ones
  → We always have enough "money" to pay!

Accounting Method

Op.	Counter	Cost	To Bank	From Bank	Net Cost	Credit
	0 0 0 0 0					
1	0 0 0 0 **1**	1				
2	0 0 0 **1** **0**	2				
3	0 0 0 1 **1**	1				
4	0 0 **1** **0** **0**	3				
5	0 0 1 0 **1**	1				
6	0 0 1 **1** **0**	2				
7	0 0 1 1 **1**	1				
8	0 **1** **0** **0** **0**	4				
9	0 1 0 0 **1**	1				
10	0 1 0 **1** **0**	2				

Chapter 4
Amortized Analysis

Algorithm Theory
WS 2018/19

Fabian Kuhn

Amortized Analysis

What is the average cost of an operation in a worst case sequence of operations?

Amortized Cost

- Sequence of operations $i = 1, 2, 3, \dots, n$
- Actual cost of op. i: $\boldsymbol{t_i}$
- Amortized cost of op. i is $\boldsymbol{a_i}$ if for every possible seq. of op.,

$$T = \sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i$$

Example 2: Binary Counter

Incrementing a binary counter: determine the bit flip cost:

Operation	Counter Value	Cost
	00000	
1	00001	1
2	00010	2
3	00011	1
4	00100	3
5	00101	1
6	00110	2
7	00111	1
8	01000	4
9	01001	1
10	01010	2
11	01011	1
12	01100	3
13	01101	1

Accounting Method

Observation:

- Each increment flips exactly one 0 into a 1

$$0010001111 \Longrightarrow 0010010000$$

Idea:

- Have a bank account (with initial amount 0)
- Paying x to the bank account costs x
- Take "money" from account to pay for expensive operations

Applied to binary counter:

- Flip from 0 to 1: pay 1 to bank account (cost: 2)
- Flip from 1 to 0: take 1 from bank account (cost: 0)
- Amount on bank account = number of ones
  → We always have enough "money" to pay!

Potential Function Method

- Most generic and elegant way to do amortized analysis!
  - But, also more abstract than the others…

- State of data structure / system: $S \in \mathcal{S}$ (state space)

  Potential function $\boldsymbol{\Phi}: \boldsymbol{\mathcal{S}} \to \mathbb{R}_{\geq 0}$

- **Operation $\boldsymbol{i}$:**
  - $\boldsymbol{t_i}$: actual cost of operation i
  - $\boldsymbol{S_i}$: state after execution of operation i (S_0: initial state)
  - $\boldsymbol{\Phi_i} := \Phi(S_i)$: potential after exec. of operation i
  - $\boldsymbol{a_i}$: amortized cost of operation i:

$$\boldsymbol{a_i := t_i + \Phi_i - \Phi_{i-1}}$$

Potential Function Method

Operation i:

actual cost: t_i amortized cost: $a_i = t_i + \Phi_i - \Phi_{i-1}$

Overall cost:

$$T := \sum_{i=1}^{n} t_i = \left(\sum_{i}^{n} a_i\right) + \Phi_0 - \Phi_n$$

Binary Counter: Potential Method

- **Potential function:**

  Φ: number of ones in current counter

- Clearly, $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all $i \geq 0$

- Actual cost t_i:
  - 1 flip from 0 to 1
  - $t_i - 1$ flips from 1 to 0

- Potential difference: $\Phi_i - \Phi_{i-1} = 1 - (t_i - 1) = 2 - t_i$

- Amortized cost: $a_i = t_i + \Phi_i - \Phi_{i-1} = 2$

Example 3: Dynamic Array

- How to create an array where the size dynamically adapts to the number of elements stored?
  - e.g., Java “ArrayList” or Python “list”

Implementation:

- Initialize with initial size N_0
- Assumptions: Array can only grow by appending new elements at the end
- If array is full, the size of the array is increased by a factor $\beta > 1$

Operations (array of size N):

- read / write: actual cost $O(1)$
- append: actual cost is $O(1)$ if array is not full, otherwise the append cost is $O(\beta \cdot N)$ (new array size)

Example 3: Dynamic Array

Notation:

- n: number of elements stored
- N: current size of array

Cost t_i of ith append operation: $t_i = \begin{cases} 1 & \text{if } n < N \\ \beta \cdot N & \text{if } n = N \end{cases}$

Claim: Amortized append cost is $O(1)$

Potential function Φ?

- should allow to pay expensive append operations by cheap ones
- when array is full, Φ has to be large
- immediately after increasing the size of the array, Φ should be small again

Dynamic Array: Potential Function

Cost t_i of ith append operation: $t_i = \begin{cases} 1 & \text{if } n < N \\ \beta \cdot N & \text{if } n = N \end{cases}$

Dynamic Array: Amortized Cost

Cost t_i of ith append operation: $t_i = \begin{cases} 1 & \text{if } n < N \\ \beta \cdot N & \text{if } n = N \end{cases}$

Chapter 5
Data Structures

Algorithm Theory
WS 2018/19

Fabian Kuhn

Examples

Dictionary:

- Operations: insert(*key,value*), delete(*key*), find(*key*)
- Implementations:
  - Linked list: all operations take $O(n)$ time (n: size of data structure)
  - Balanced binary tree: all operations take $O(\log n)$ time
  - Hash table: all operations take $O(1)$ times (with some assumptions)

Stack (LIFO Queue):

- Operations: push, pull
- Linked list: $O(1)$ for both operations

(FIFO) Queue:

- Operations: enqueue, dequeue
- Linked list: $O(1)$ time for both operations

Here: **Priority Queues (heaps)**, **Union-Find data structure**

Dijkstra's Algorithm

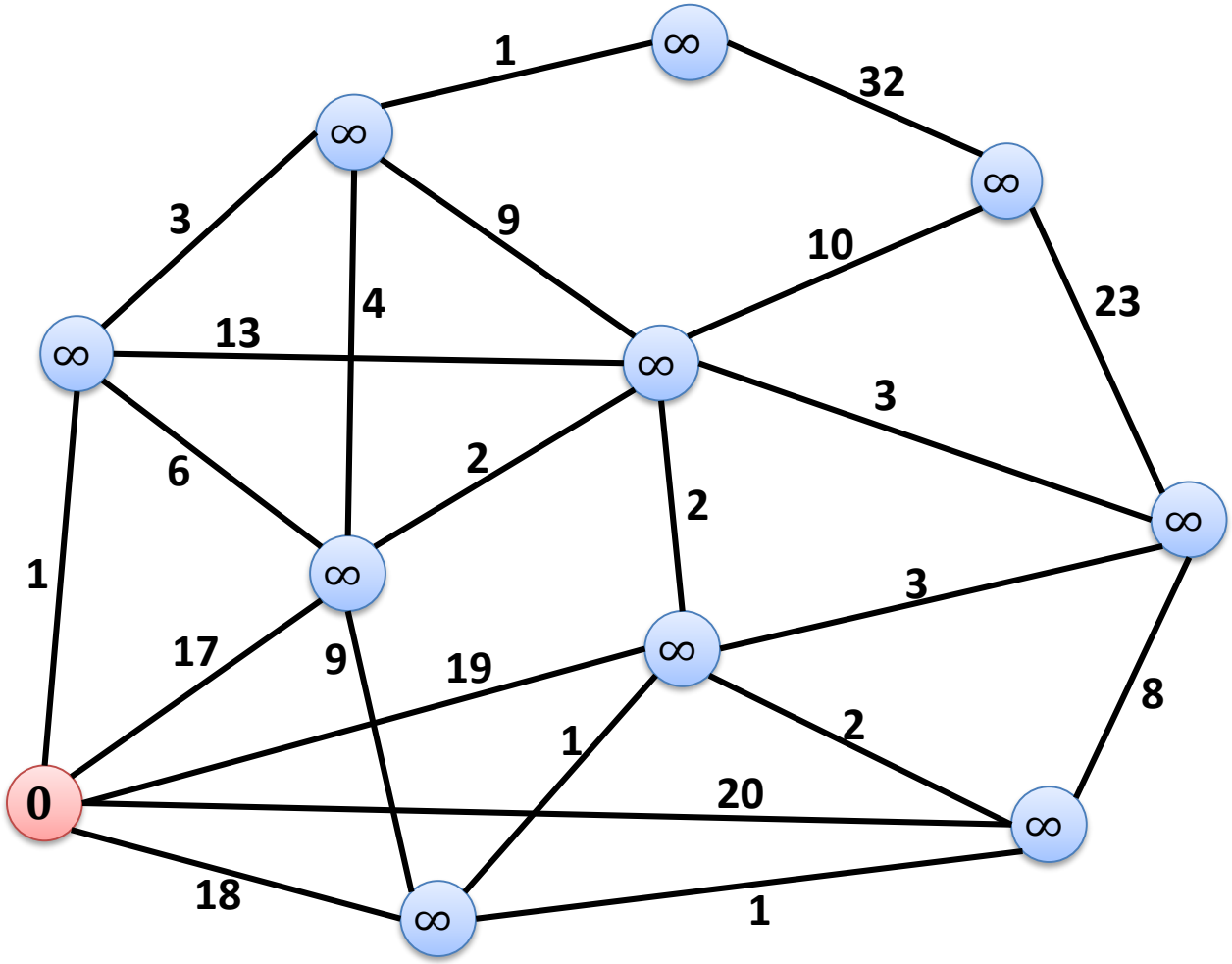
Single-Source Shortest Path Problem:

- **Given:** graph $G = (V, E)$ with edge weights $w(e) \geq 0$ for $e \in E$
  source node $s \in V$
- **Goal:** compute shortest paths from s to all $v \in V$

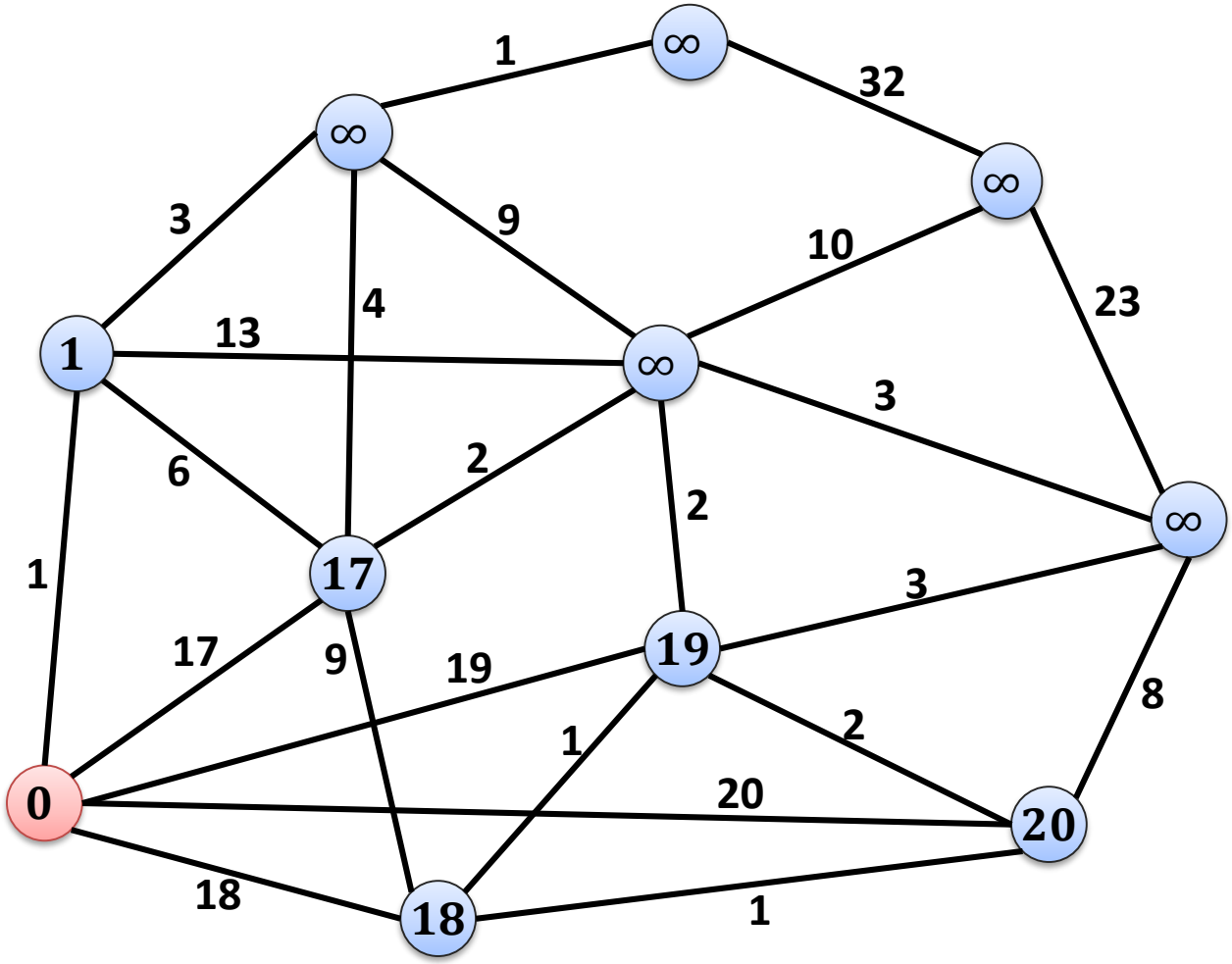
Dijkstra's Algorithm:

1. Initialize $d(s, s) = 0$ and $d(s, v) = \infty$ for all $v \neq s$
2. All nodes are unmarked
3. Get unmarked node u which minimizes $d(s, u)$:
4. For all $e = \{u, v\} \in E$, $d(s, v) = \min\{d(s, v), d(s, u) + w(e)\}$
5. mark node u
6. Until all nodes are marked

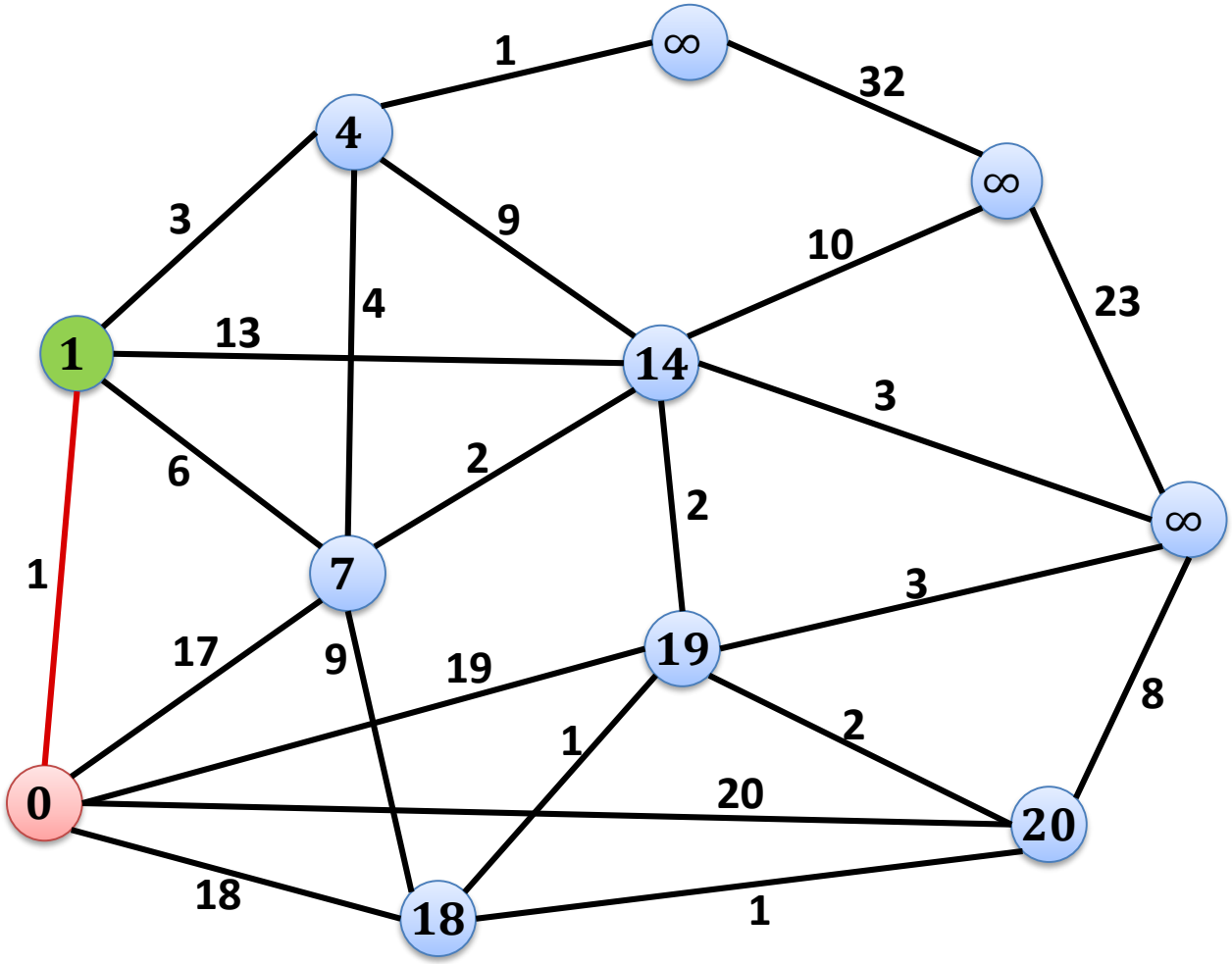
Example



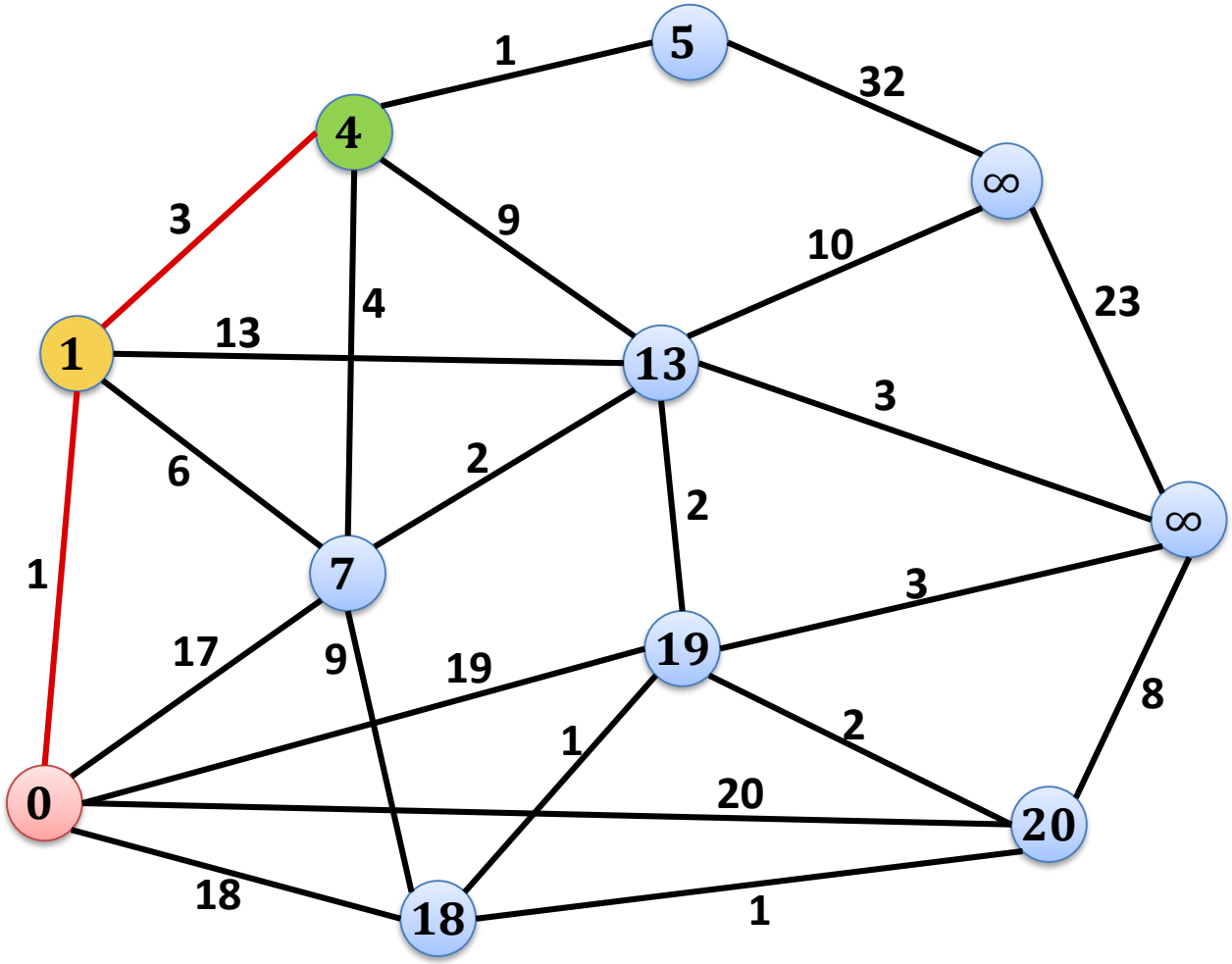
Example



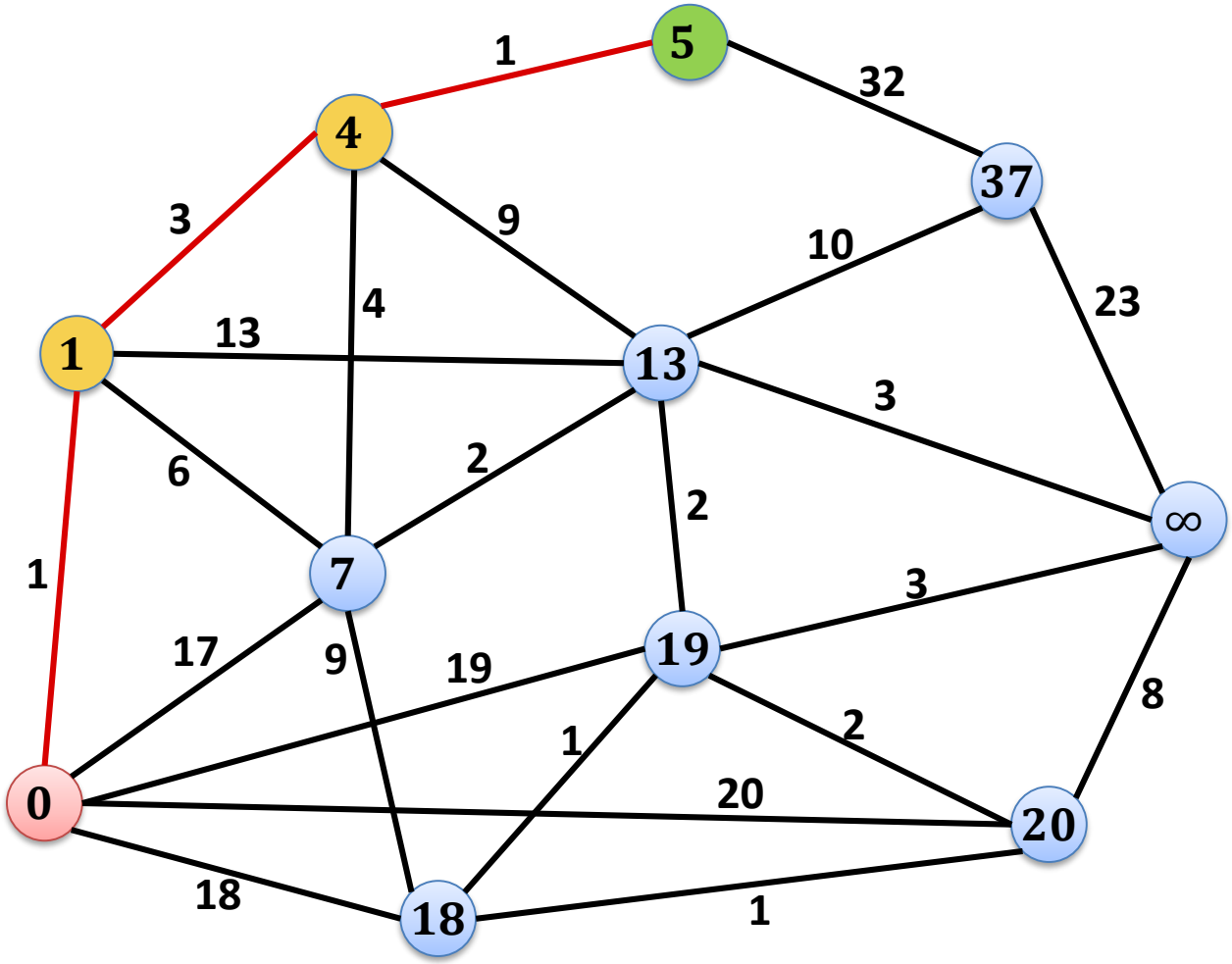
Example



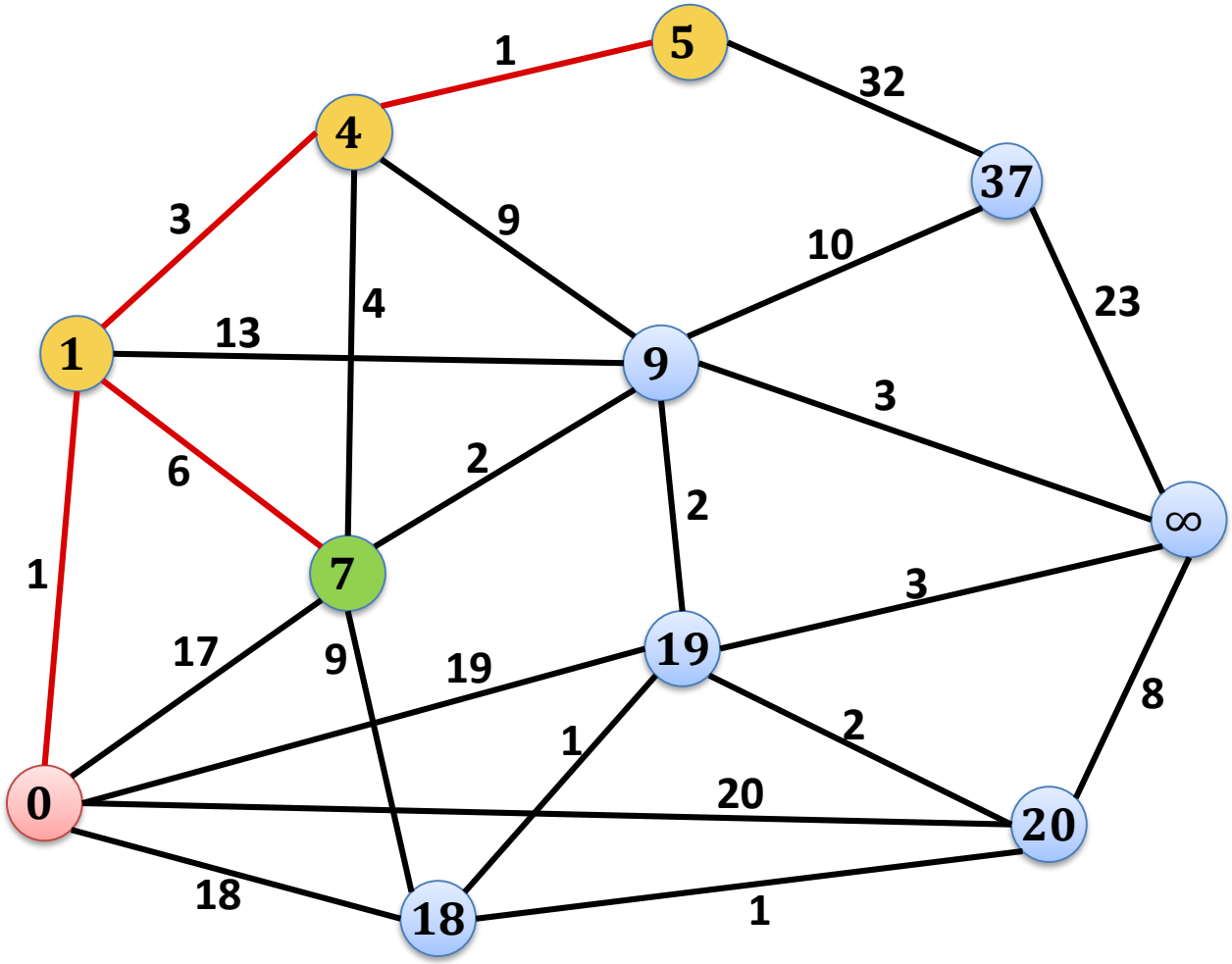
Example



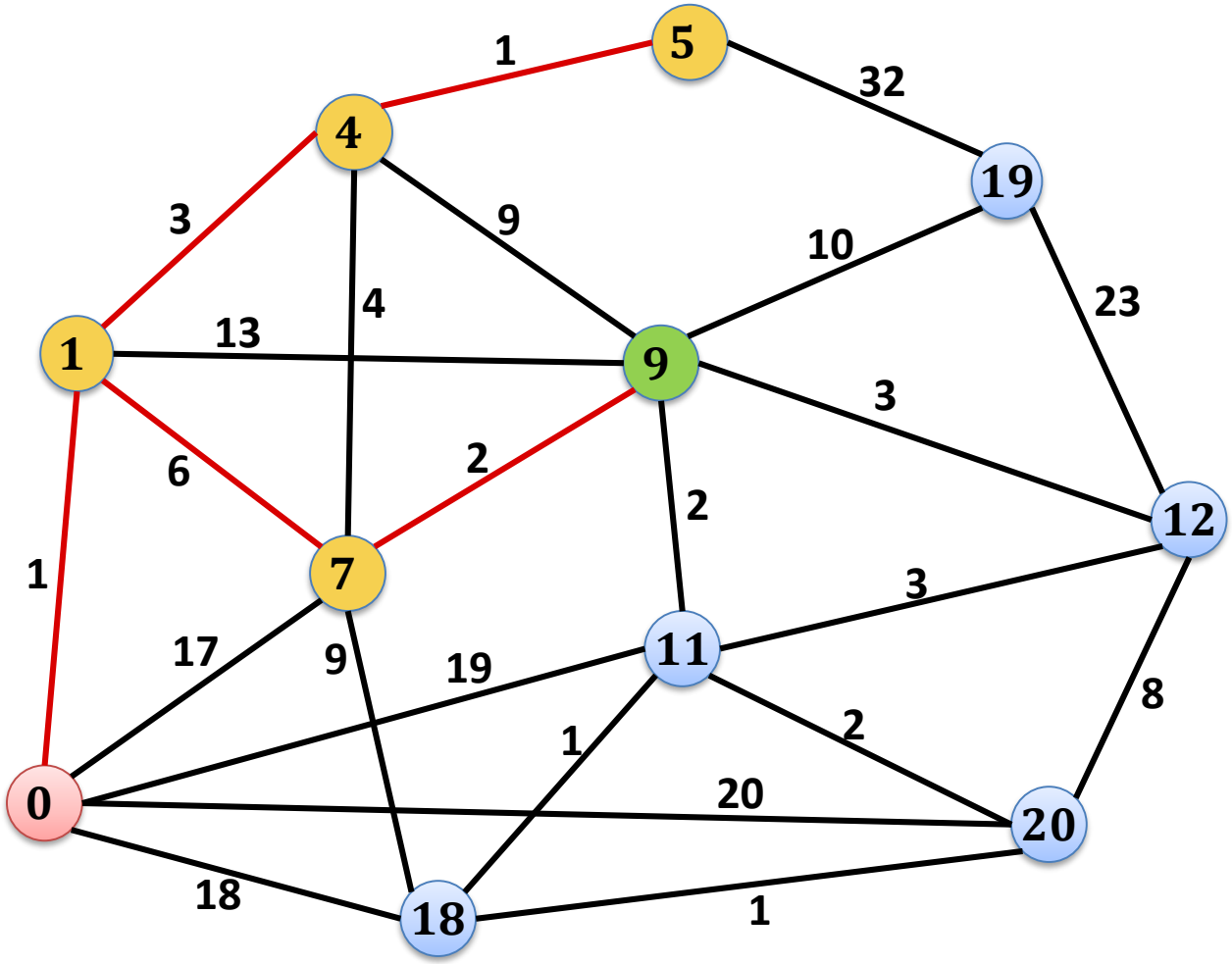
Example



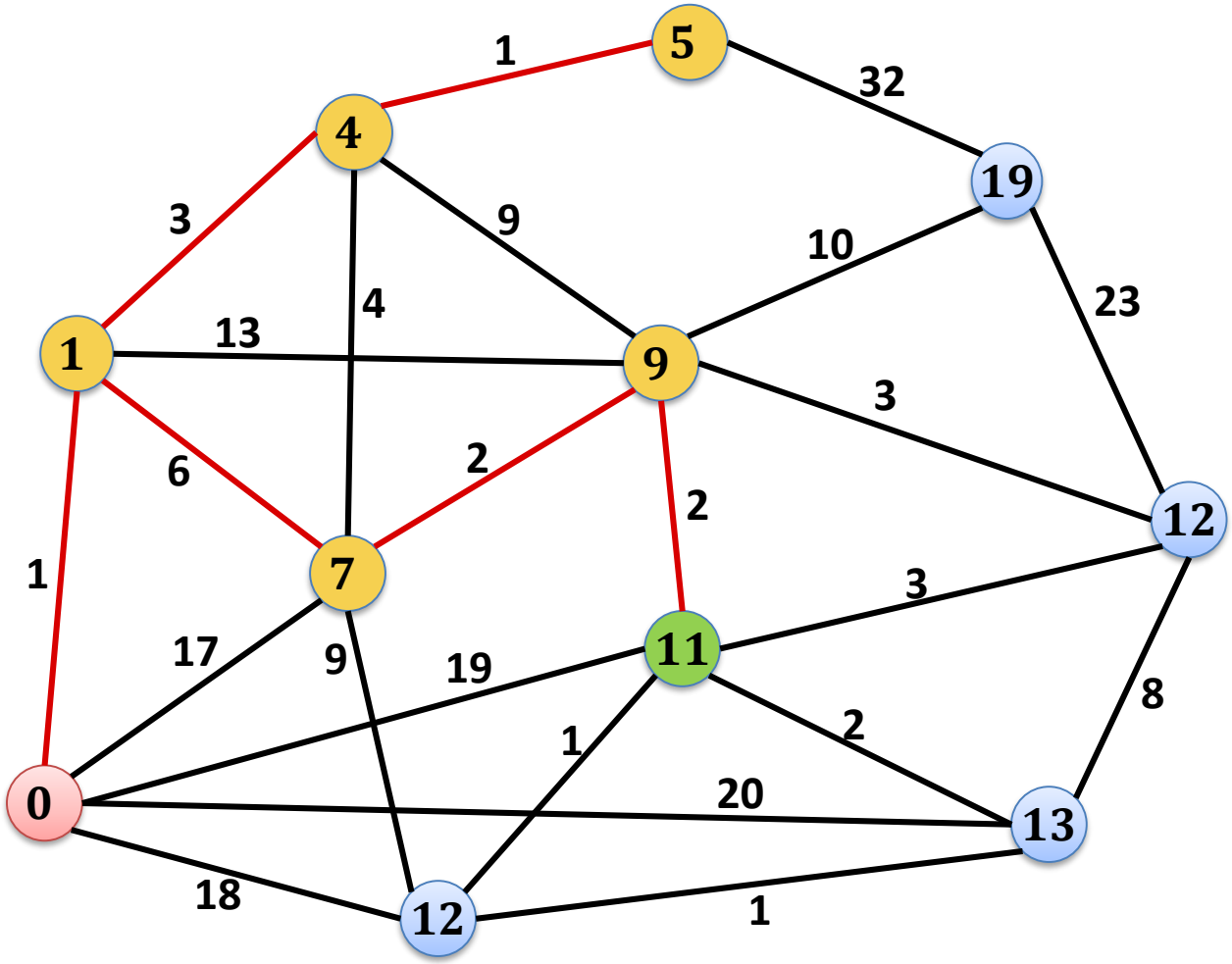
Example



Example



Example



Implementation of Dijkstra's Algorithm

Dijkstra's Algorithm:

1. Initialize $d(s,s) = 0$ and $d(s,v) = \infty$ for all $v \neq s$
2. All nodes $v \neq s$ are unmarked

3. Get unmarked node u which minimizes $d(s,u)$:

4. For all $e = \{u,v\} \in E$, $d(s,v) = \min\{d(s,v), d(s,u) + w(e)\}$

5. mark node u

6. Until all nodes are marked

Priority Queue / Heap

- Stores (*key,data*) pairs (like dictionary)
- But, different set of operations:

- **Initialize-Heap**: creates new empty heap
- **Is-Empty**: returns true if heap is empty
- **Insert**(*key,data*): inserts (*key,data*)-pair, returns pointer to entry
- **Get-Min**: returns (*key,data*)-pair with minimum *key*
- **Delete-Min**: deletes minimum (*key,data*)-pair
- **Decrease-Key**(*entry,newkey*): decreases *key* of *entry* to *newkey*
- **Merge**: merges two heaps into one

Implementation of Dijkstra's Algorithm

Store nodes in a priority queue, use $\boldsymbol{d(s,v)}$ as keys:

1. Initialize $d(s,s) = 0$ and $d(s,v) = \infty$ for all $v \neq s$
2. All nodes $v \neq s$ are unmarked

3. Get unmarked node u which minimizes $d(s,u)$:

4. mark node u

5. For all $e = \{u,v\} \in E$, $d(s,v) = \min\{d(s,v), d(s,u) + w(e)\}$

6. Until all nodes are marked

Analysis

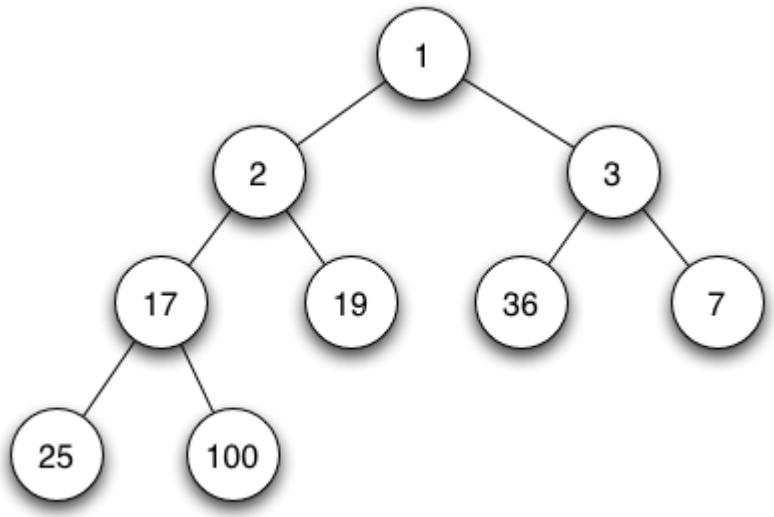
Number of priority queue operations for Dijkstra:

- **Initialize-Heap**: $\mathbf{1}$
- **Is-Empty**: $|V|$
- **Insert**: $|V|$
- **Get-Min**: $|V|$
- **Delete-Min**: $|V|$
- **Decrease-Key**: $|E|$
- **Merge**: $\mathbf{0}$

Priority Queue Implementation

Implementation as min-heap:

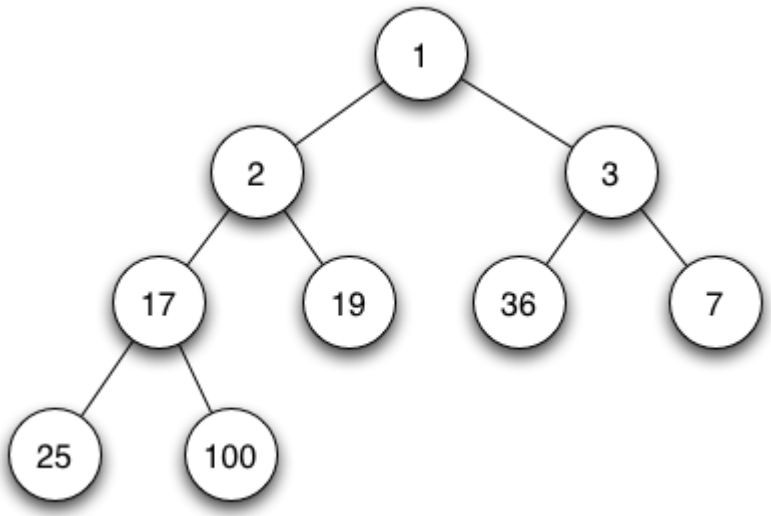
→ complete binary tree,
e.g., stored in an array



Priority Queue Implementation

Implementation as min-heap:

→ complete binary tree,
e.g., stored in an array



- **Initialize-Heap**: $\boldsymbol{O(1)}$
- **Is-Empty**: $\boldsymbol{O(1)}$
- **Insert**: $\boldsymbol{O(\log n)}$
- **Get-Min**: $\boldsymbol{O(1)}$
- **Delete-Min**: $\boldsymbol{O(\log n)}$
- **Decrease-Key**: $\boldsymbol{O(\log n)}$
- **Merge** (heaps of size m and n, $m \leq n$): $\boldsymbol{O(m \log n)}$

Can We Do Better?

- Cost of **Dijkstra** with **complete binary min-heap** implementation:

$$O(|E| \log |V|)$$

- **Binary heap:**
  insert, delete-min, and decrease-key cost $O(\log n)$
  merging two heaps is expensive

- One of the operations insert or delete-min must cost $\Omega(\log n)$:
  - Heap-Sort:
    Insert n elements into heap, then take out the minimum n times
  - (Comparison-based) sorting costs at least $\Omega(n \log n)$.

- But maybe we can improve merge, decrease-key, and one of the other two operations?

Fibonacci Heaps

Structure:

A Fibonacci heap H consists of a collection of trees satisfying the **min-heap** property.

Min-Heap Property:

Key of a node $v \leq$ keys of all nodes in any sub-tree of v

Fibonacci Heaps

Structure:

A Fibonacci heap H consists of a collection of trees satisfying the min-heap property.

Variables:

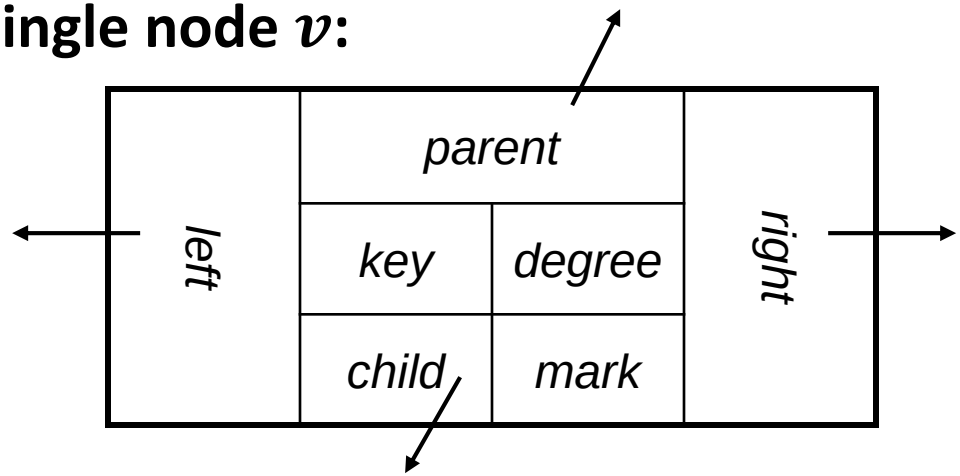
- $H.min$: root of the tree containing the (a) minimum key
- $H.rootlist$: circular, doubly linked, unordered list containing the roots of all trees
- $H.size$: number of nodes currently in H

Lazy Merging:

- To reduce the number of trees, sometimes, trees need to be merged
- Lazy merging: Do not merge as long as possible...

Trees in Fibonacci Heaps

Structure of a single node v:



- $v.child$: points to circular, doubly linked and unordered list of the children of v
- $v.left, v.right$: pointers to siblings (in doubly linked list)
- $v.mark$: will be used later…

Advantages of circular, doubly linked lists:

- Deleting an element takes constant time
- Concatenating two lists takes constant time

Example

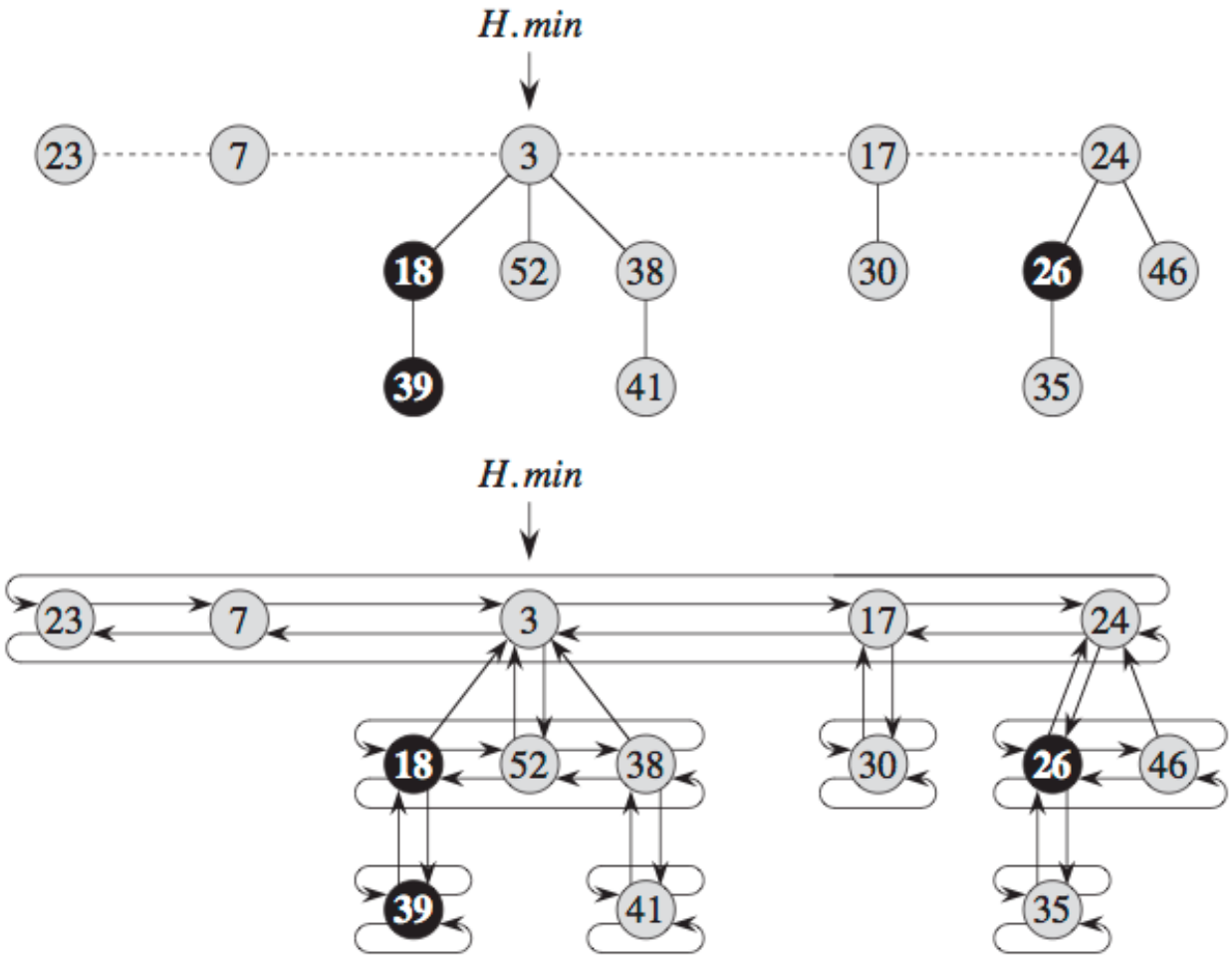


Figure: Cormen et al., Introduction to Algorithms

Simple (Lazy) Operations

Initialize-Heap H:

- $H.rootlist := H.min := null$

Merge heaps H and H':

- concatenate root lists
- update $H.min$

Insert element e into H:

- create new one-node tree containing e → H'
  - mark of root node is set to $\mathbf{false}$
- merge heaps H and H'

Get minimum element of H:

- return $H.min$

Operation Delete-Min

Delete the node with minimum key from H and return its element:

1. $m := H.min$;
2. **if** $H.size > 0$ **then**
3. remove $H.min$ from $H.rootlist$;
4. add $H.min.child$ (list) to $H.rootlist$
5. $\boldsymbol{H.Consolidate()}$;

   // Repeatedly merge nodes with equal degree in the root list
   // until degrees of nodes in the root list are distinct.
   // Determine the element with minimum key

6. **return** m

Rank and Maximum Degree

Ranks of nodes, trees, heap:

Node v:

- $rank(v)$: degree of v (number of children of v)

Tree T:

- $rank(T)$: rank (degree) of root node of T

Heap H:

- $rank(H)$: maximum degree (#children) of any node in H

Assumption (n: number of nodes in H):

$$rank(H) \leq D(n)$$

– for a known function $D(n)$

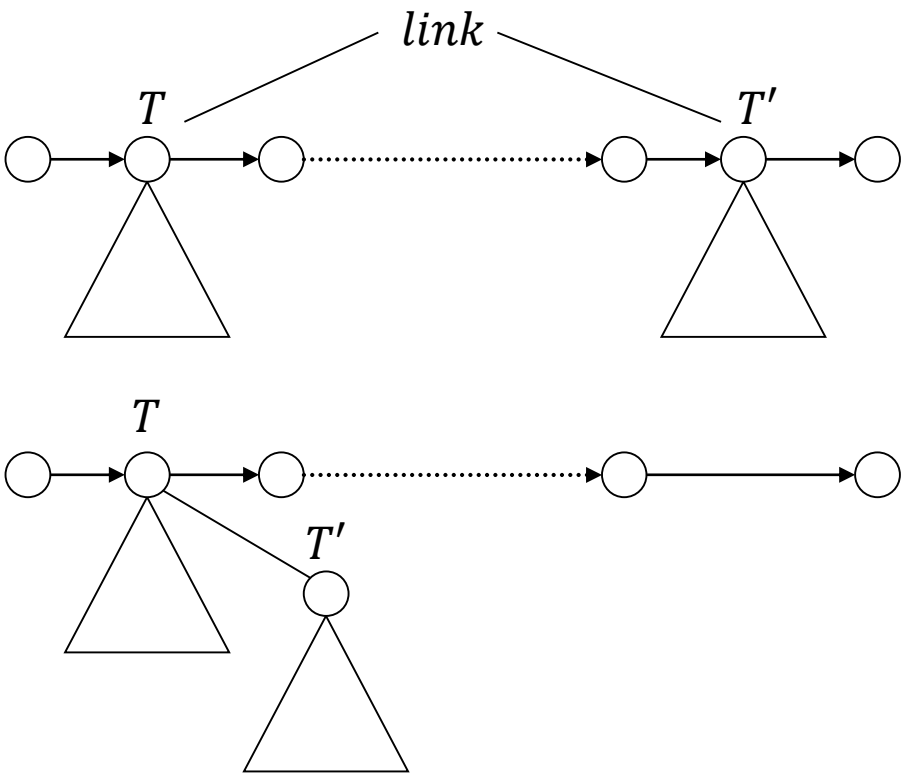
Merging Two Trees

Given: Heap-ordered trees T, T' with $rank(T) = rank(T')$

- Assume: min-key of $T <$ min-key of T'

Operation $\boldsymbol{link(T, T')}$:

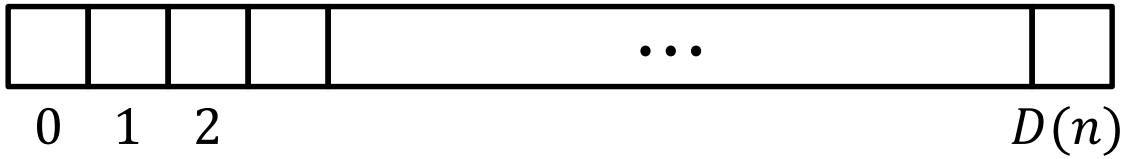
- Removes tree T' from root list and adds T' to child list of T



- $rank(T) := rank(T) + 1$
- $(T'.mark = \mathbf{false})$

Consolidation of Root List

Array A pointing to find roots with the same rank:



Consolidate:

Time:
$\boldsymbol{O(|H.rootlist| + D(n))}$

1. **for** $i := 0$ **to** $D(n)$ **do** $A[i] :=$ null;
2. **while** $H.rootlist \neq$ null **do**
3. $\quad T :=$ "delete and return first element of $H.rootlist$"
4. $\quad$ **while** $A[rank(T)] \neq$ null **do**
5. $\quad\quad T' := A[rank(T)]$;
6. $\quad\quad A[rank(T)] := null$;
7. $\quad\quad T := link(T, T')$
8. $\quad A[rank(T)] := T$
9. Create new $H.rootlist$ and $H.min$

Chapter 5
Data Structures

Algorithm Theory
WS 2018/19

Fabian Kuhn

Priority Queue / Heap

- Stores (*key,data*) pairs (like dictionary)
- But, different set of operations:

- **Initialize-Heap**: creates new empty heap
- **Is-Empty**: returns true if heap is empty
- **Insert**(*key,data*): inserts (*key,data*)-pair, returns pointer to entry
- **Get-Min**: returns (*key,data*)-pair with minimum *key*
- **Delete-Min**: deletes minimum (*key,data*)-pair
- **Decrease-Key**(*entry,newkey*): decreases *key* of *entry* to *newkey*
- **Merge**: merges two heaps into one

Analysis

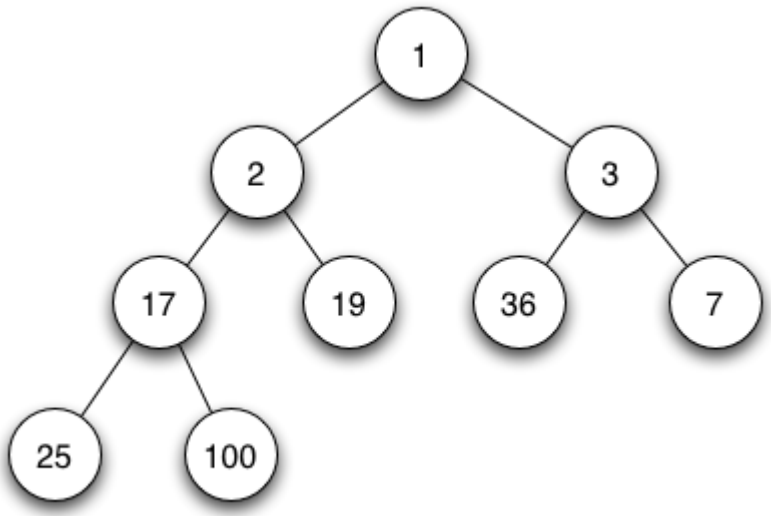
Number of priority queue operations for Dijkstra:

- **Initialize-Heap**: $\mathbf{1}$
- **Is-Empty**: $|V|$
- **Insert**: $|V|$
- **Get-Min**: $|V|$
- **Delete-Min**: $|V|$
- **Decrease-Key**: $|E|$
- **Merge**: $\mathbf{0}$

Priority Queue Implementation

Implementation as min-heap:

→ complete binary tree,
e.g., stored in an array



- **Initialize-Heap**: $\boldsymbol{O(1)}$
- **Is-Empty**: $\boldsymbol{O(1)}$
- **Insert**: $\boldsymbol{O(\log n)}$
- **Get-Min**: $\boldsymbol{O(1)}$
- **Delete-Min**: $\boldsymbol{O(\log n)}$
- **Decrease-Key**: $\boldsymbol{O(\log n)}$
- **Merge** (heaps of size m and n, $m \leq n$): $\boldsymbol{O(m \log n)}$

Can We Do Better?

- Cost of **Dijkstra** with **complete binary min-heap** implementation:

$$O(|E| \log |V|)$$

- **Binary heap:**
  insert, delete-min, and decrease-key cost $O(\log n)$
  merging two heaps is expensive

- One of the operations insert or delete-min must cost $\Omega(\log n)$:
  - Heap-Sort:
    Insert n elements into heap, then take out the minimum n times
  - (Comparison-based) sorting costs at least $\Omega(n \log n)$.

- But maybe we can improve merge, decrease-key, and one of the other two operations?

Fibonacci Heaps

Structure:

A Fibonacci heap H consists of a collection of trees satisfying the **min-heap** property.

Min-Heap Property:

Key of a node $v \leq$ keys of all nodes in any sub-tree of v

Example

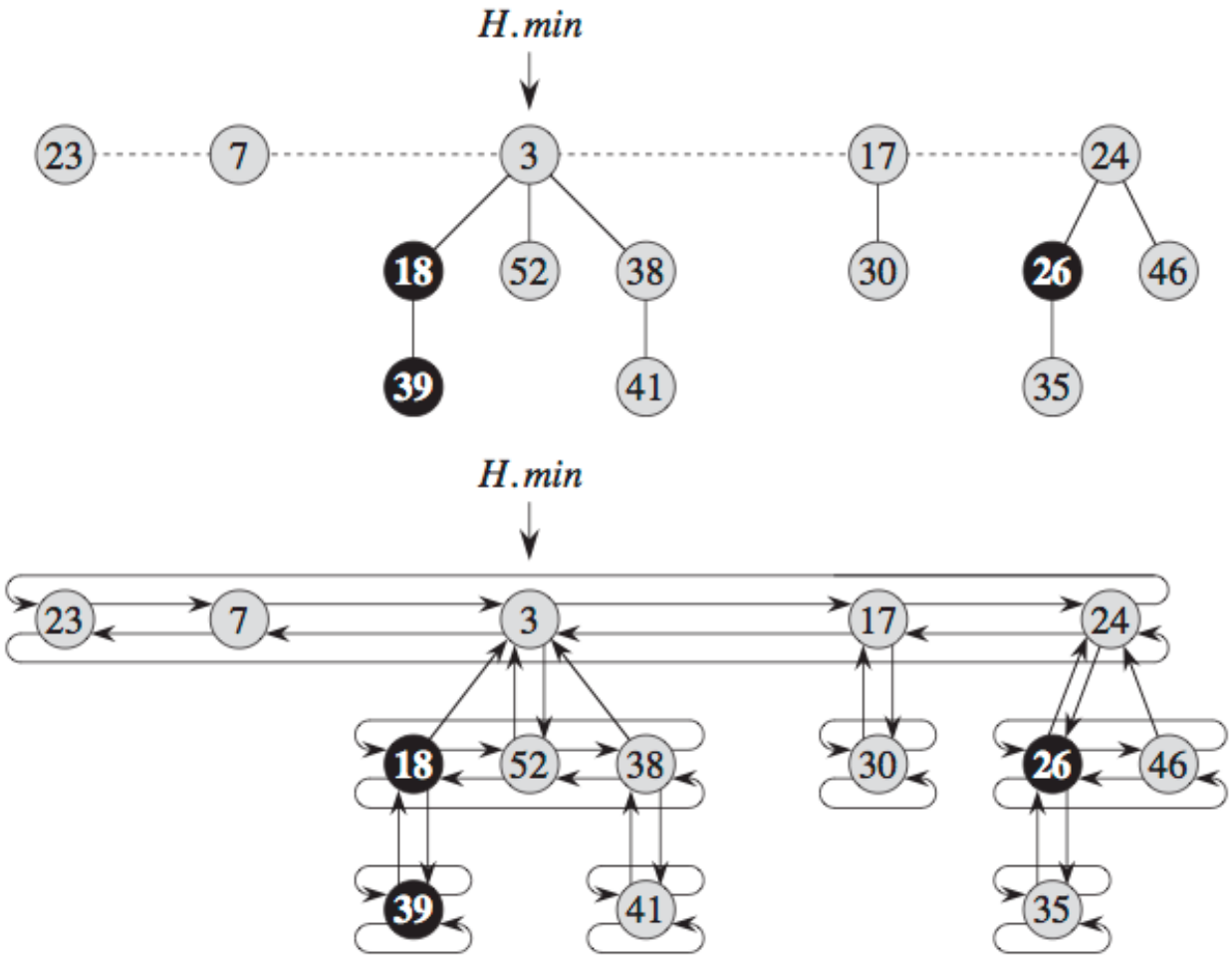


Figure: Cormen et al., Introduction to Algorithms

Simple (Lazy) Operations

Initialize-Heap H:

- $H.rootlist := H.min := null$

Merge heaps H and H':

- concatenate root lists
- update $H.min$

Insert element e into H:

- create new one-node tree containing e → H'
  - mark of root node is set to $\mathbf{false}$
- merge heaps H and H'

Get minimum element of H:

- return $H.min$

Operation Delete-Min

Delete the node with minimum key from H and return its element:

1. $m := H.min$;
2. **if** $H.size > 0$ **then**
3. remove $H.min$ from $H.rootlist$;
4. add $H.min.child$ (list) to $H.rootlist$
5. $\boldsymbol{H.Consolidate()}$;

   // Repeatedly merge nodes with equal degree in the root list
   // until degrees of nodes in the root list are distinct.
   // Determine the element with minimum key

6. **return** m

Rank and Maximum Degree

Ranks of nodes, trees, heap:

Node v:

- $rank(v)$: degree of v (number of children of v)

Tree T:

- $rank(T)$: rank (degree) of root node of T

Heap H:

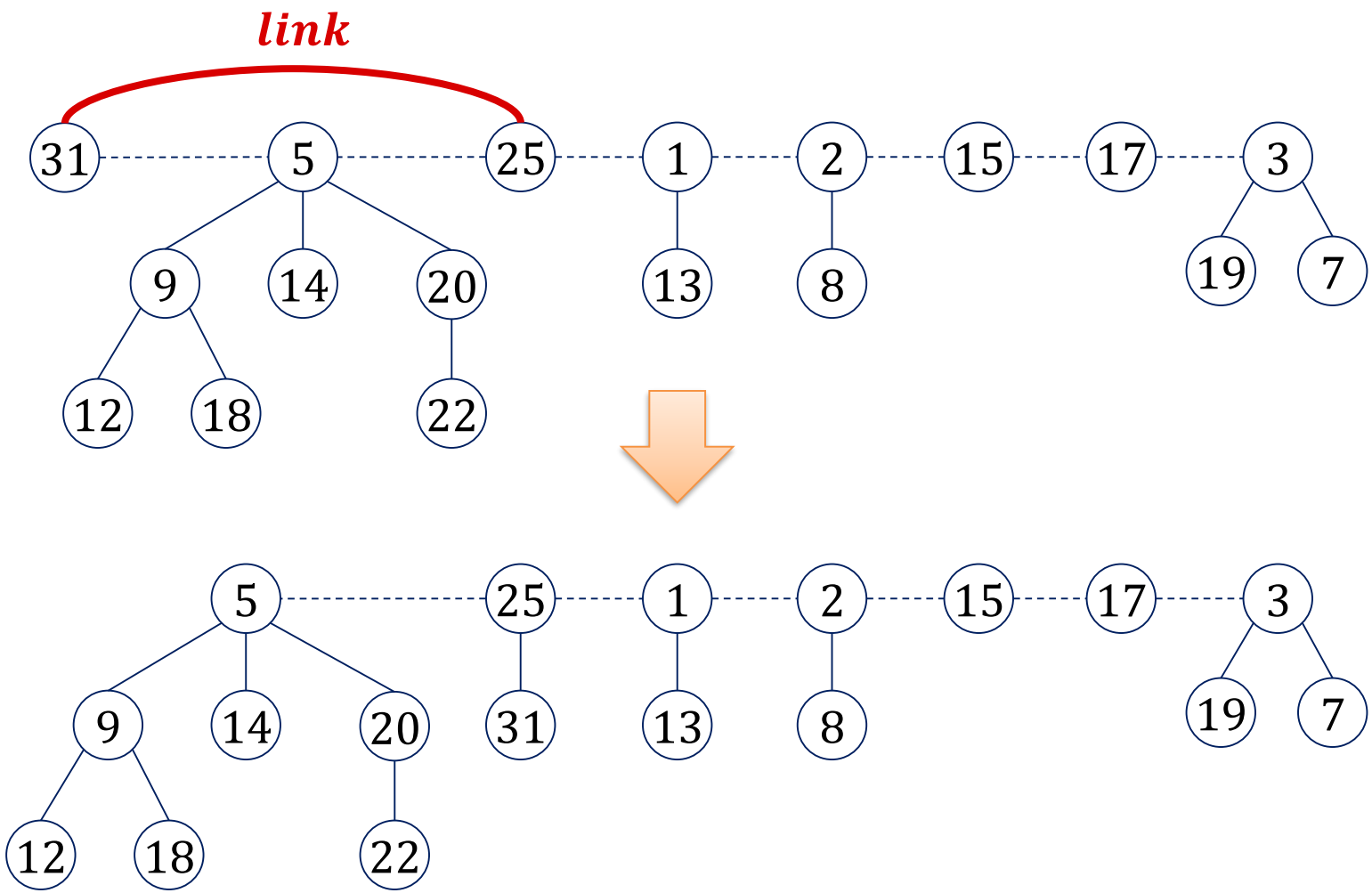
- $rank(H)$: maximum degree (#children) of any node in H

Assumption (n: number of nodes in H):

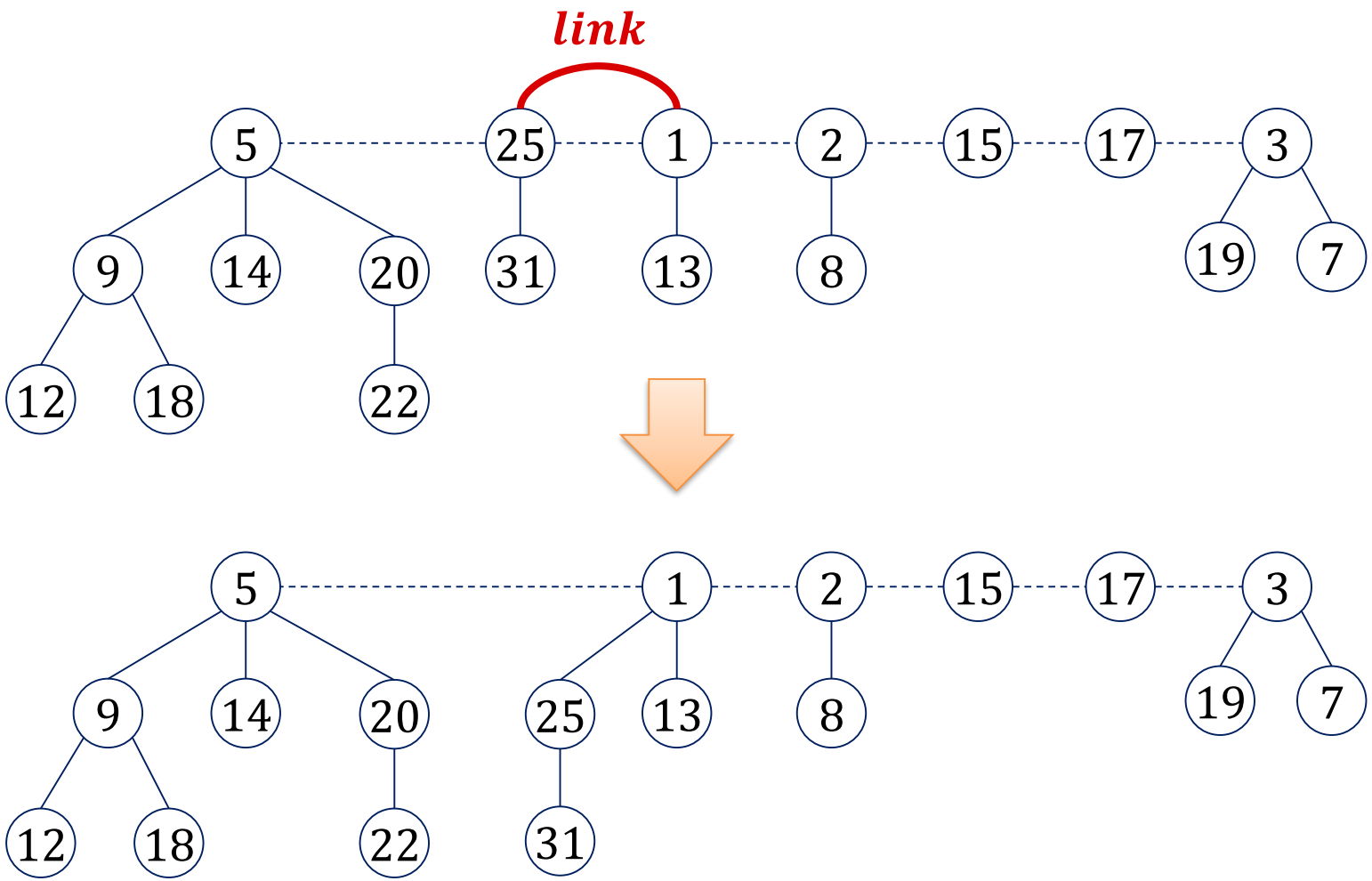
$$rank(H) \leq D(n)$$

– for a known function $D(n)$

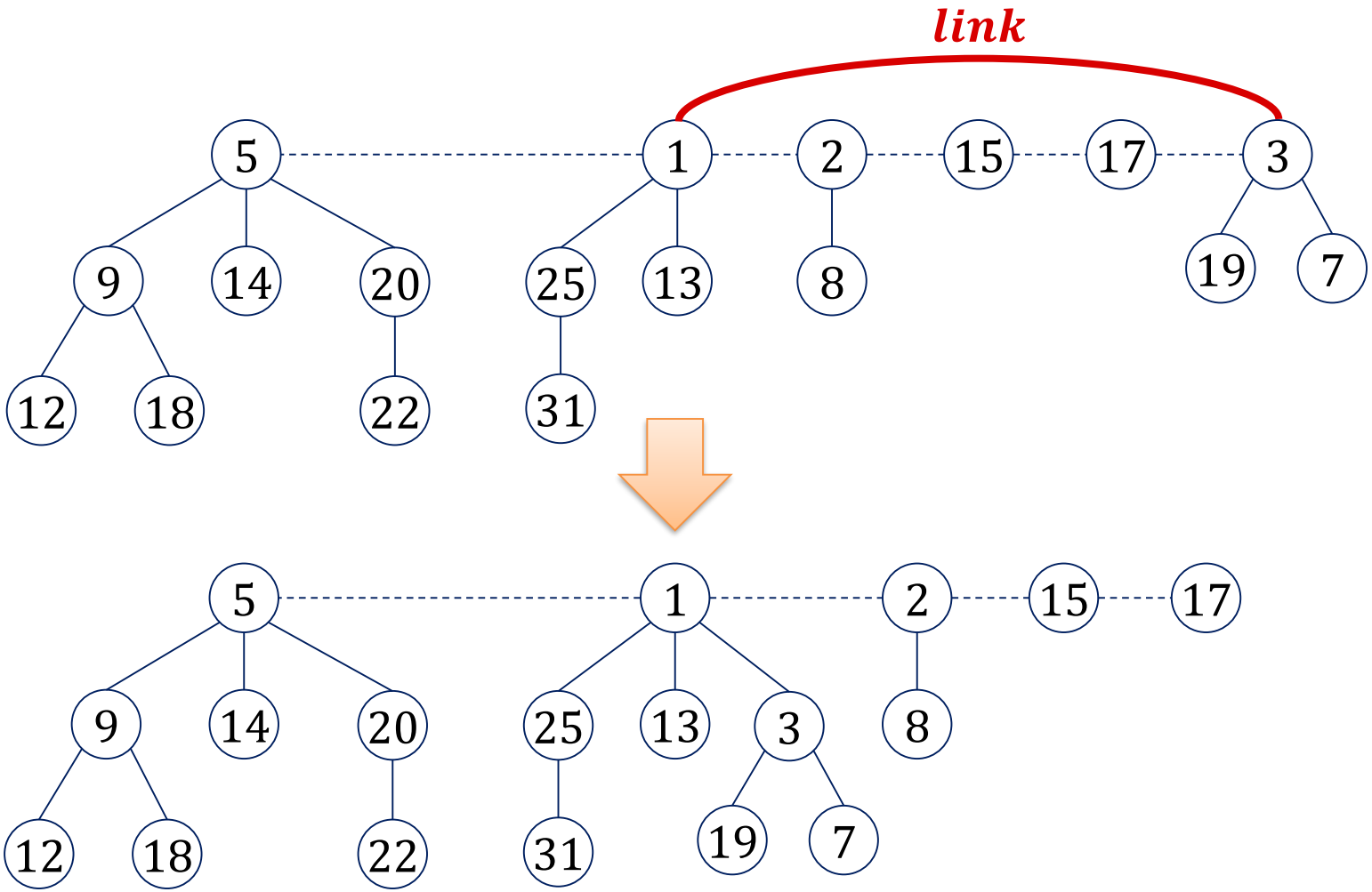
Consolidate Example



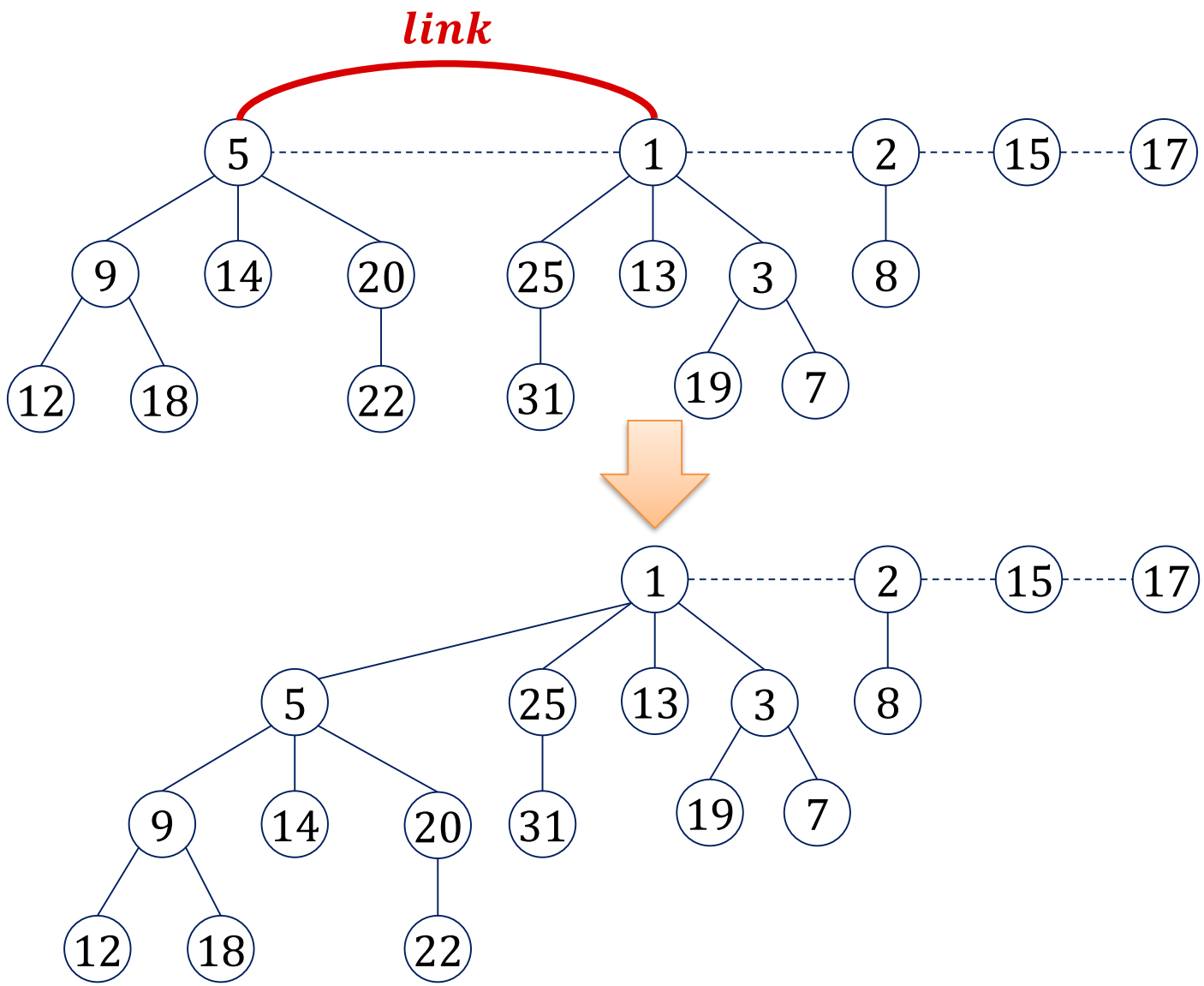
Consolidate Example



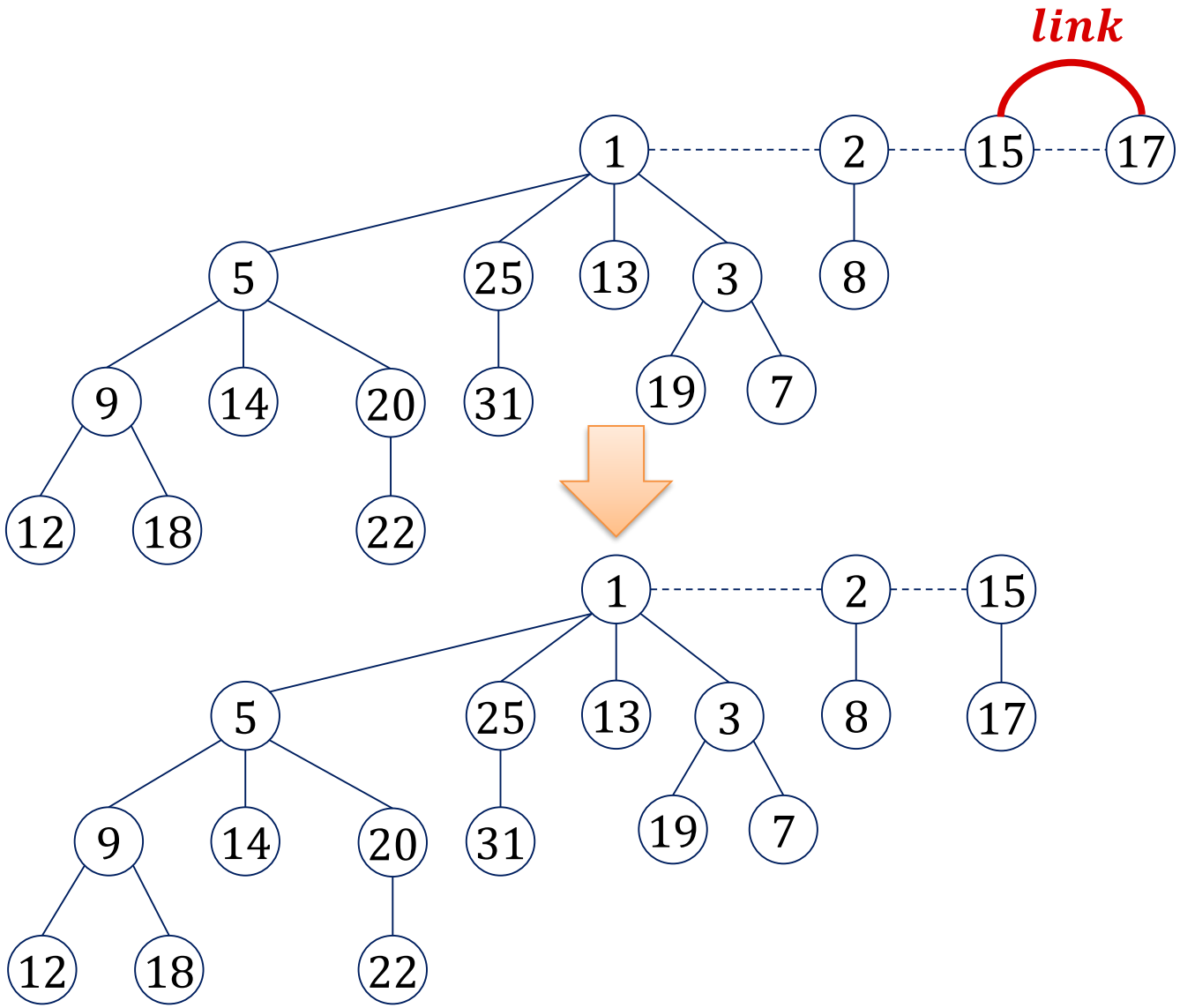
Consolidate Example



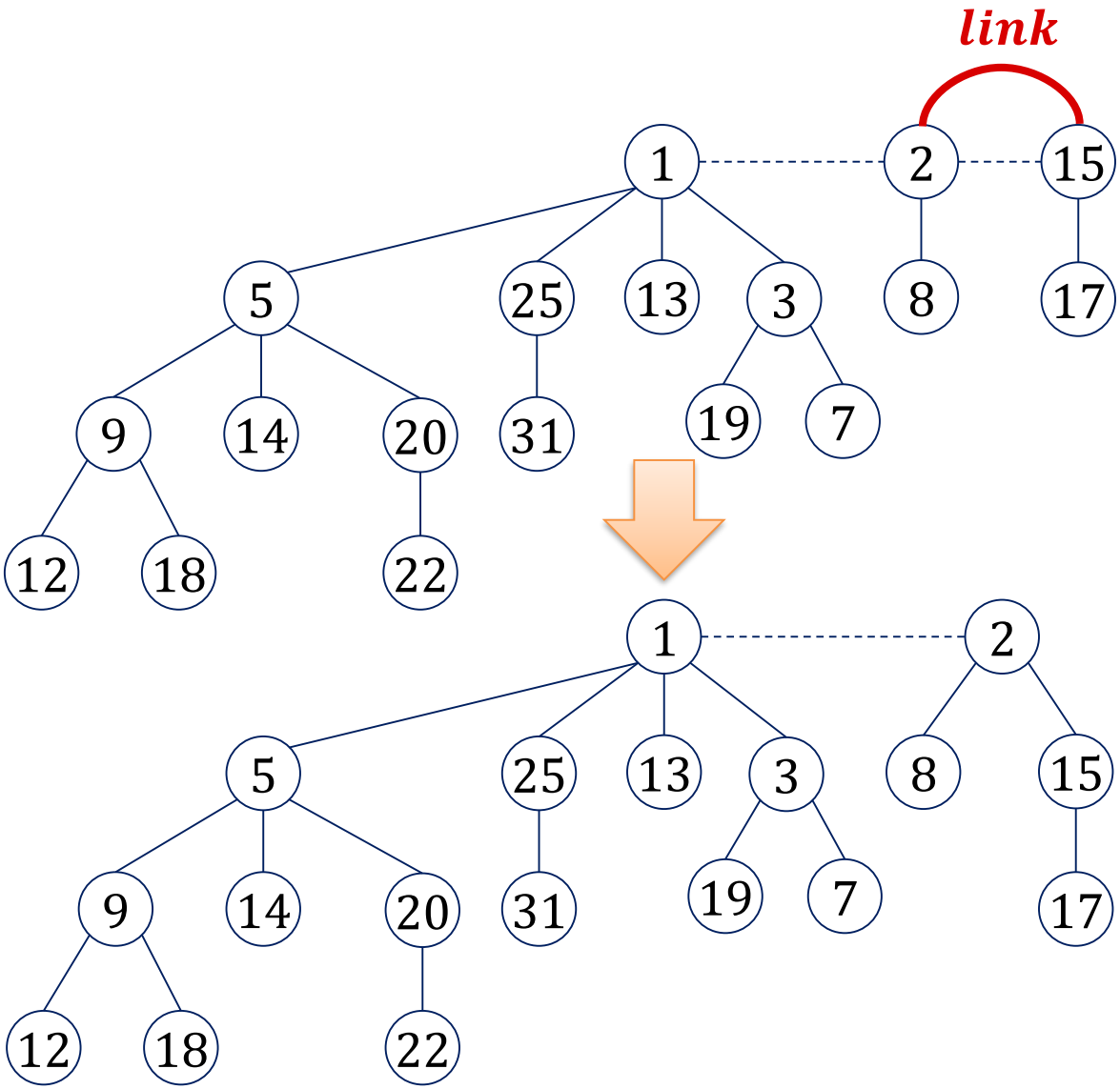
Consolidate Example



Consolidate Example



Consolidate Example



Operation Decrease-Key

Decrease-Key$(\boldsymbol{v}, \boldsymbol{x})$**:** (decrease key of node v to new value x)

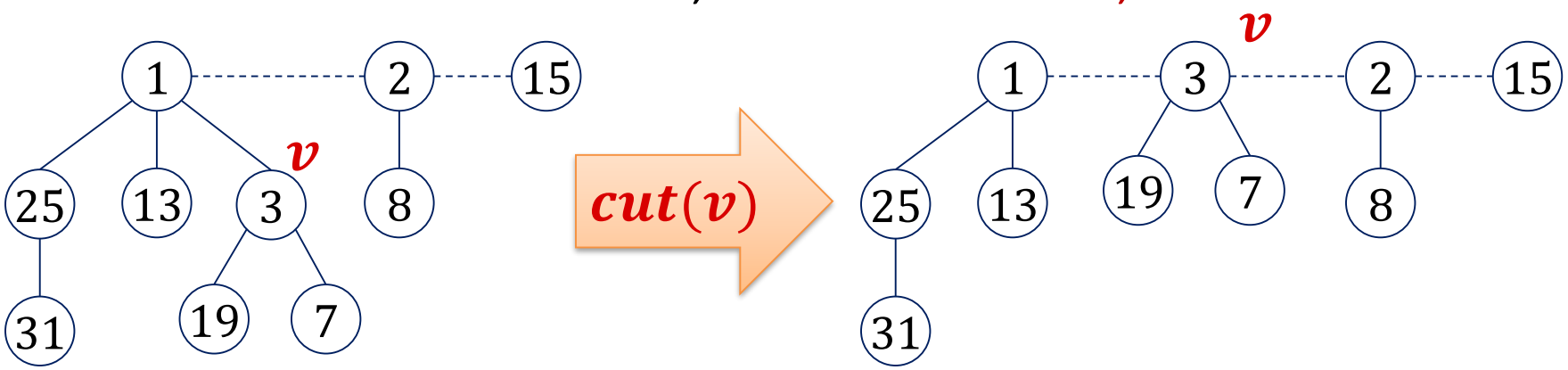
1. **if** $x \geq v.key$ **then return**;
2. $v.key := x$; update $H.min$;
3. **if** $v \in H.rootlist \ \vee \ x \geq v.parent.key$ **then return**
4. **repeat**
5. $\quad parent := v.parent$;
6. $\quad \boldsymbol{H.cut(v)}$;
7. $\quad v := parent$;
8. **until** $\neg(\boldsymbol{v.mark}) \ \vee \ v \in H.rootlist$;
9. **if** $v \notin H.rootlist$ **then** $\boldsymbol{v.mark := \mathrm{true}}$;

Operation Cut(v)

Operation $H.cut(v)$:

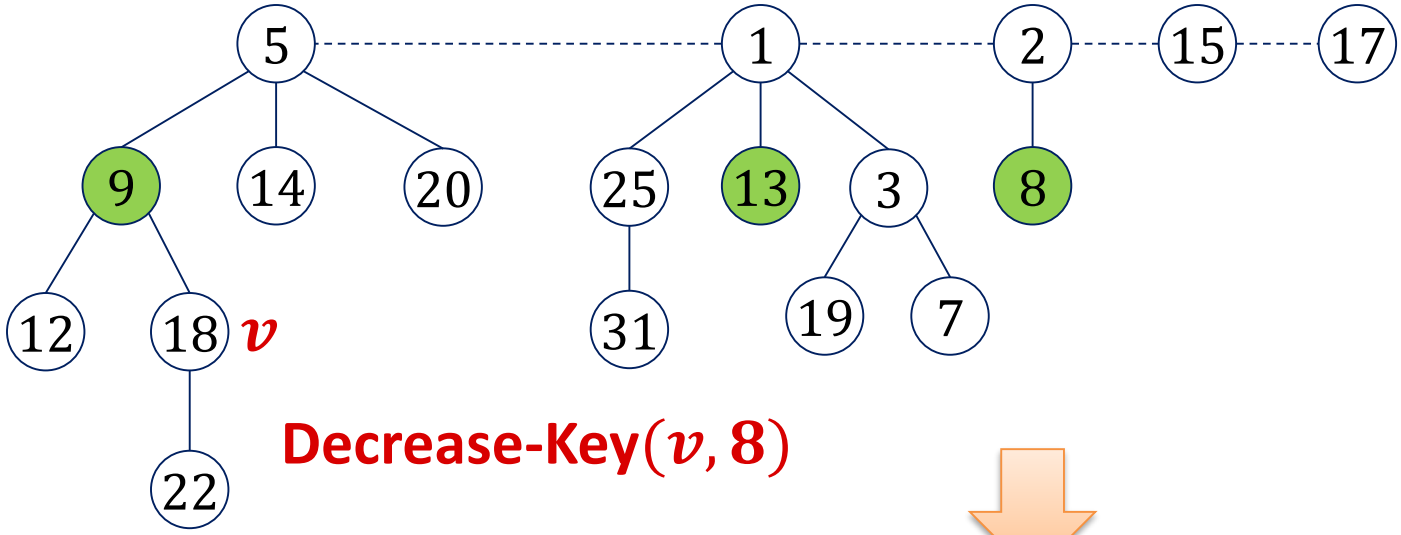
- Cuts v's sub-tree from its parent and adds v to rootlist

1. **if** $v \notin H.rootlist$ **then**
2. // cut the link between v and its parent
3. $rank(v.parent) := rank(v.parent) - 1;$
4. remove v from $v.parent.child$ (list)
5. $v.parent := \text{null};$
6. add v to $H.rootlist$; $v.mark := \text{false};$

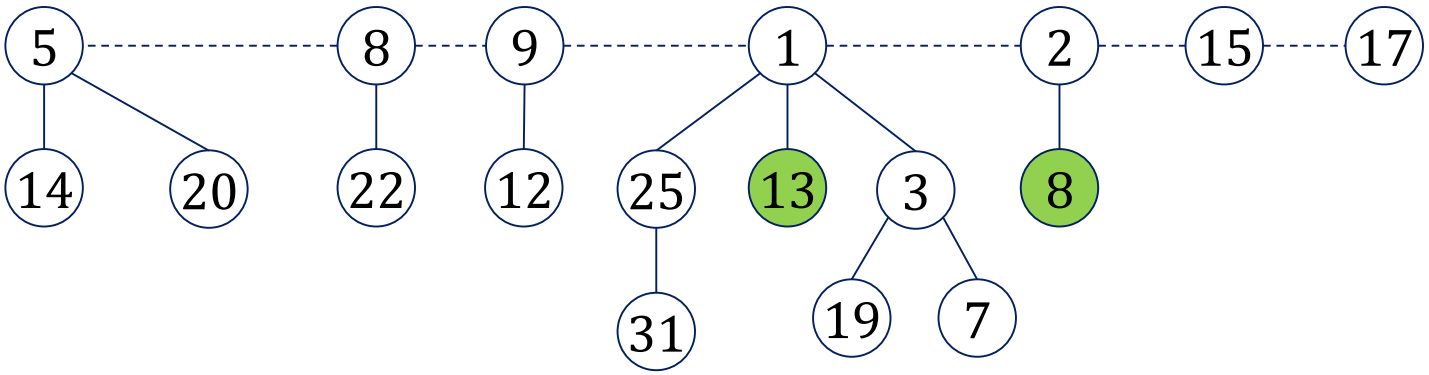


Decrease-Key Example

- Green nodes are marked



Decrease-Key$(v, 8)$



Fibonacci Heaps Marks

- Nodes in the root list (the tree roots) are always unmarked
  → If a node is added to the root list (insert, decrease-key), the mark of the node is set to false.

- Nodes not in the root list can only get marked when a subtree is cut in a decrease-key operation

- A node v is marked if and only if v is not in the root list and v has lost a child since v was attached to its current parent
  – a node can only change its parent by being moved to the root list

Fibonacci Heap Marks

History of a node v:

v is being linked to a node ⇒ $\boldsymbol{v.mark = \text{false}}$

a child of v is cut ⇒ $\boldsymbol{v.mark := \text{true}}$

a second child of v is cut ⇒ $\boldsymbol{H.cut(v);}$
$\boldsymbol{v.mark := false}$

- Hence, the boolean value $v.mark$ indicates whether node v has lost a child since the last time v was made the child of another node.
- Nodes v in the root list always have $v.mark = \text{false}$

Cost of Delete-Min & Decrease-Key

Delete-Min:

1. Delete min. root r and add $r.child$ to $H.rootlist$

   time: $O(1)$

2. Consolidate $H.rootlist$

   time: $O(\text{length of } H.rootlist + D(n))$

- Step 2 can potentially be linear in n (size of H)

Decrease-Key (at node v):

1. If new key $<$ parent key, cut sub-tree of node v

   time: $O(1)$

2. Cascading cuts up the tree as long as nodes are marked

   time: $O(\text{number of consecutive marked nodes})$

- Step 2 can potentially be linear in n

Exercise: Both operations can take $\Theta(n)$ time in the worst case!

Cost of Delete-Min & Decrease-Key

- Cost of delete-min and decrease-key can be $\Theta(n)$…
  - Seems a large price to pay to get insert and merge in $O(1)$ time

- Maybe, the operations are efficient most of the time?
  - It seems to require a lot of operations to get a long rootlist and thus, an expensive consolidate operation
  - In each decrease-key operation, at most one node gets marked: We need a lot of decrease-key operations to get an expensive decrease-key operation

- Can we show that the average cost per operation is small?

- We can → requires **amortized analysis**

Fibonacci Heaps Complexity

- Worst-case cost of a single delete-min or decrease-key operation is $\Omega(n)$

- Can we prove a small worst-case amortized cost for delete-min and decrease-key operations?

Recall:

- Data structure that allows operations $O_1, \dots, O_k$

- We say that operation O_p has amortized cost a_p if for every execution the total time is

$$T \leq \sum_{p=1}^{k} n_p \cdot a_p \,,$$

where n_p is the number of operations of type O_p

Amortized Cost of Fibonacci Heaps

- **Initialize-heap**, **is-empty**, **get-min**, **insert**, and **merge** have **worst-case cost $O(1)$**

- **Delete-min** has **amortized cost $O(\log n)$**
- **Decrease-key** has **amortized cost $O(1)$**

- Starting with an empty heap, any sequence of n operations with at most n_d delete-min operations has total cost (time)

$$T = O(n + n_d \log n).$$

- We will now need the marks…

- Cost for Dijkstra: $O(|E| + |V| \log |V|)$

Fibonacci Heaps: Marks

Cycle of a node:

1. Node v is removed from root list and linked to a node

   $\boldsymbol{v.mark = \text{false}}$

2. Child node u *of* v is cut and added to root list

   $\boldsymbol{v.mark := \text{true}}$

3. Second child of v is cut

   node v is cut as well and moved to root list

   $\boldsymbol{v.mark := \text{false}}$

The boolean value $v.mark$ indicates whether node v has lost a child since the last time v was made the child of another node.

Potential Function

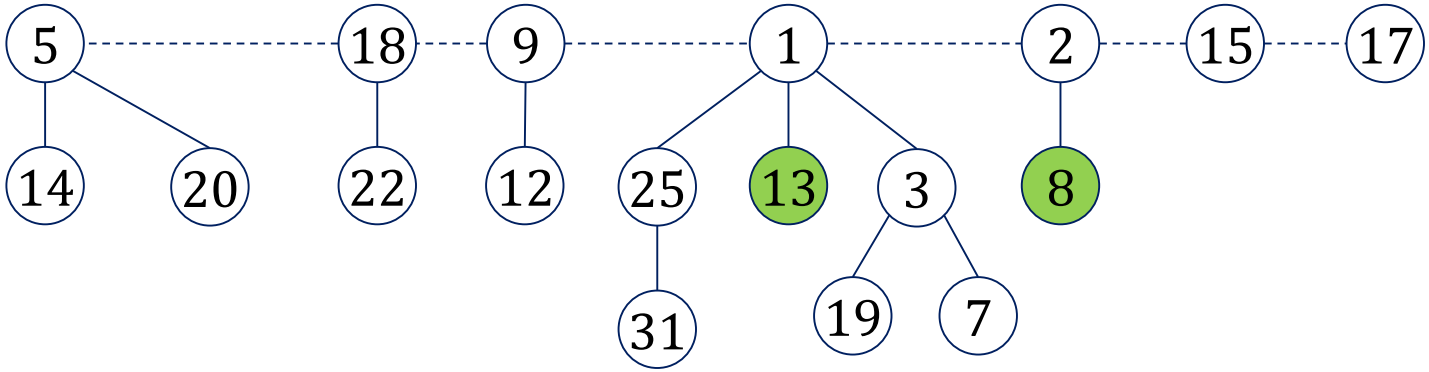
System state characterized by two parameters:

- $\boldsymbol{R}$**:** number of trees (length of $H.rootlist$)
- $\boldsymbol{M}$**:** number of marked nodes (not in the root list)

Potential function:

$$\boldsymbol{\Phi := R + 2M}$$

Example:



- $R = 7, M = 2 \rightarrow \Phi = 11$

Actual Time of Operations

- Operations: ***initialize-heap***, ***is-empty***, ***insert***, ***get-min***, ***merge***

  actual time: $O(1)$

  – Normalize unit time such that

  $$t_{init}, t_{is-empty}, t_{insert}, t_{get-min}, t_{merge} \leq 1$$

- Operation ***delete-min***:
  – Actual time: $O\big(\text{length of } H.rootlist + D(n)\big)$
  – Normalize unit time such that

  $$t_{del-min} \leq D(n) + \text{ length of } H.rootlist$$

- Operation **descrease-key**:
  – Actual time: $O(\text{length of path to next unmarked ancestor})$
  – Normalize unit time such that

  $$t_{decr-key} \leq \text{length of path to next unmarked ancestor}$$

Amortized Times

Assume operation i is of type:

- **initialize-heap:**
  - actual time: $t_i \leq 1$, potential: $\Phi_{i-1} = \Phi_i = 0$
  - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$
- **is-empty, get-min:**
  - actual time: $t_i \leq 1$, potential: $\Phi_i = \Phi_{i-1}$ (heap doesn't change)
  - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$
- **merge:**
  - Actual time: $t_i \leq 1$
  - combined potential of both heaps: $\Phi_i = \Phi_{i-1}$
  - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$

Amortized Time of Insert

Assume that operation i is an *insert* operation:

- **Actual time:** $t_i \leq 1$
- **Potential function:**
  - M remains unchanged (no nodes are marked or unmarked, no marked nodes are moved to the root list)
  - R grows by 1 (one element is added to the root list)

$$M_i = M_{i-1}, \qquad R_i = R_{i-1} + 1$$
$$\Phi_i = \Phi_{i-1} + 1$$

- **Amortized time:**

$$\boldsymbol{a_i = t_i + \Phi_i - \Phi_{i-1} \leq 2}$$

Amortized Time of Delete-Min

Assume that operation i is a *delete-min* operation:

Actual time: $t_i \leq D(n) + |H.rootlist|$

Potential function $\boldsymbol{\Phi = R + 2M}$:

- R: changes from $|H.rootlist|$ to at most $D(n) + 1$
- M: (# of marked nodes that are not in the root list)
  - Number of marks does not increase

$$M_i = M_{i-1}, \qquad R_i \leq R_{i-1} + D(n) + 1 - |H.rootlist|$$
$$\Phi_i \leq \Phi_{i-1} + D(n) + 1 - |H.rootlist|$$

Amortized Time:

$$\boldsymbol{a_i = t_i + \Phi_i - \Phi_{i-1} \leq 2D(n) + 1}$$

Amortized Time of Decrease-Key

Assume that operation i is a *decrease-key* operation at node u:

Actual time: $t_i \leq$ length of path to next unmarked ancestor v

Potential function $\boldsymbol{\Phi = R + 2M}$:

- Assume, node u and nodes $u_1, \dots, u_k$ are moved to root list
  - $u_1, \dots, u_k$ are marked and moved to root list, $v.\mathrm{mark}$ is set to true

Amortized Time of Decrease-Key

Assume that operation i is a *decrease-key* operation at node u:

Actual time: $t_i \leq$ length of path to next unmarked ancestor v

Potential function $\boldsymbol{\Phi = R + 2M}$:

- Assume, node u and nodes $u_1, \dots, u_k$ are moved to root list
  - $u_1, \dots, u_k$ are marked and moved to root list, $v.\mathrm{mark}$ is set to true
- $\geq k$ marked nodes go to root list, ≤ 1 node gets newly marked
- R grows by $\leq k + 1$, M grows by 1 and is decreased by $\geq k$

$$R_i \leq R_{i-1} + k + 1, \qquad M_i \leq M_{i-1} + 1 - k$$
$$\Phi_i \leq \Phi_{i-1} + (k+1) - 2(k-1) = \Phi_{i-1} + 3 - k$$

Amortized time:

$$\boldsymbol{a_i = t_i + \Phi_i - \Phi_{i-1} \leq k + 1 + 3 - k = 4}$$

Complexities Fibonacci Heap

- **Initialize-Heap**: $\boldsymbol{O(1)}$
- **Is-Empty**: $\boldsymbol{O(1)}$
- **Insert**: $\boldsymbol{O(1)}$
- **Get-Min**: $\boldsymbol{O(1)}$
- **Delete-Min**: $\boldsymbol{O(D(n))}$
- **Decrease-Key**: $\boldsymbol{O(1)}$

amortized

- **Merge** (heaps of size m and n, $m \leq n$): $\boldsymbol{O(1)}$

- **How large can $D(n)$ get?**

Rank of Children

Lemma:

Consider a node v of rank k and let $u_1, \dots, u_k$ be the children of v in the order in which they were linked to v. Then,

$$\boldsymbol{rank(u_i) \geq i - 2}.$$

Proof:

Size of Trees

Fibonacci Numbers:

$$F_0 = 0, \qquad F_1 = 1, \qquad \forall k \geq 2: F_k = F_{k-1} + F_{k-2}$$

Lemma:

In a Fibonacci heap, the size of the sub-tree of a node v with rank k is at least F_{k+2}.

Proof:

- S_k: minimum size of the sub-tree of a node of rank k

Size of Trees

$$S_0 = 1, \qquad S_1 = 2, \qquad \forall k \geq 2: S_k \geq 2 + \sum_{i=0}^{k-2} S_i$$

- Claim about Fibonacci numbers:

$$\forall k \geq 0: F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

Size of Trees

$$S_0 = 1, S_1 = 2, \forall k \geq 2: S_k \geq 2 + \sum_{i=0}^{k-2} S_i, \qquad F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

- Claim of lemma: $S_k \geq F_{k+2}$

Size of Trees

Lemma:
In a Fibonacci heap, the size of the sub-tree of a node v with rank k is at least F_{k+2}.

Theorem:
The maximum rank of a node in a Fibonacci heap of size n is at most

$$\boldsymbol{D(n) = O(\log n)}\,.$$

Proof:

- The Fibonacci numbers grow exponentially:

$$F_k = \frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1+\sqrt{5}}{2} \right)^k - \left(\frac{1-\sqrt{5}}{2} \right)^k \right)$$

- For $D(n) \geq k$, we need $n \geq F_{k+2}$ nodes.

Summary: Binary and Fibonacci Heaps

	Binary Heap	Fibonacci Heap
initialize	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(1)$
get-min	$O(1)$	$O(1)$
delete-min	$O(\log n)$	$O(\log n)$ *
decrease-key	$O(\log n)$	$O(1)$ *
merge	$O(m \cdot \log n)$	$O(1)$
is-empty	$O(1)$	$O(1)$

* **amortized time**

Minimum Spanning Trees

Prim Algorithm:

1. Start with any node s (v is the initial component)
2. In each step:
   Grow the current component by adding the minimum weight edge e connecting the current component with any other node

Kruskal Algorithm:

1. Start with an empty edge set
2. In each step:
   Add minimum weight edge e such that e does not close a cycle

Implementation of Prim Algorithm

Start at node s, very similar to Dijkstra's algorithm:

1. Initialize $d(s) = 0$ and $d(v) = \infty$ for all $v \neq s$
2. All nodes $s \geq v$ are unmarked

3. Get unmarked node u which minimizes $d(u)$:

4. For all $e = \{u, v\} \in E$, $d(v) = \min\{d(v), w(e)\}$

5. mark node u

6. Until all nodes are marked

Implementation of Prim Algorithm

Implementation with Fibonacci heap:

- Analysis identical to the analysis of Dijkstra's algorithm:

  $O(n)$ insert and delete-min operations

  $O(m)$ decrease-key operations

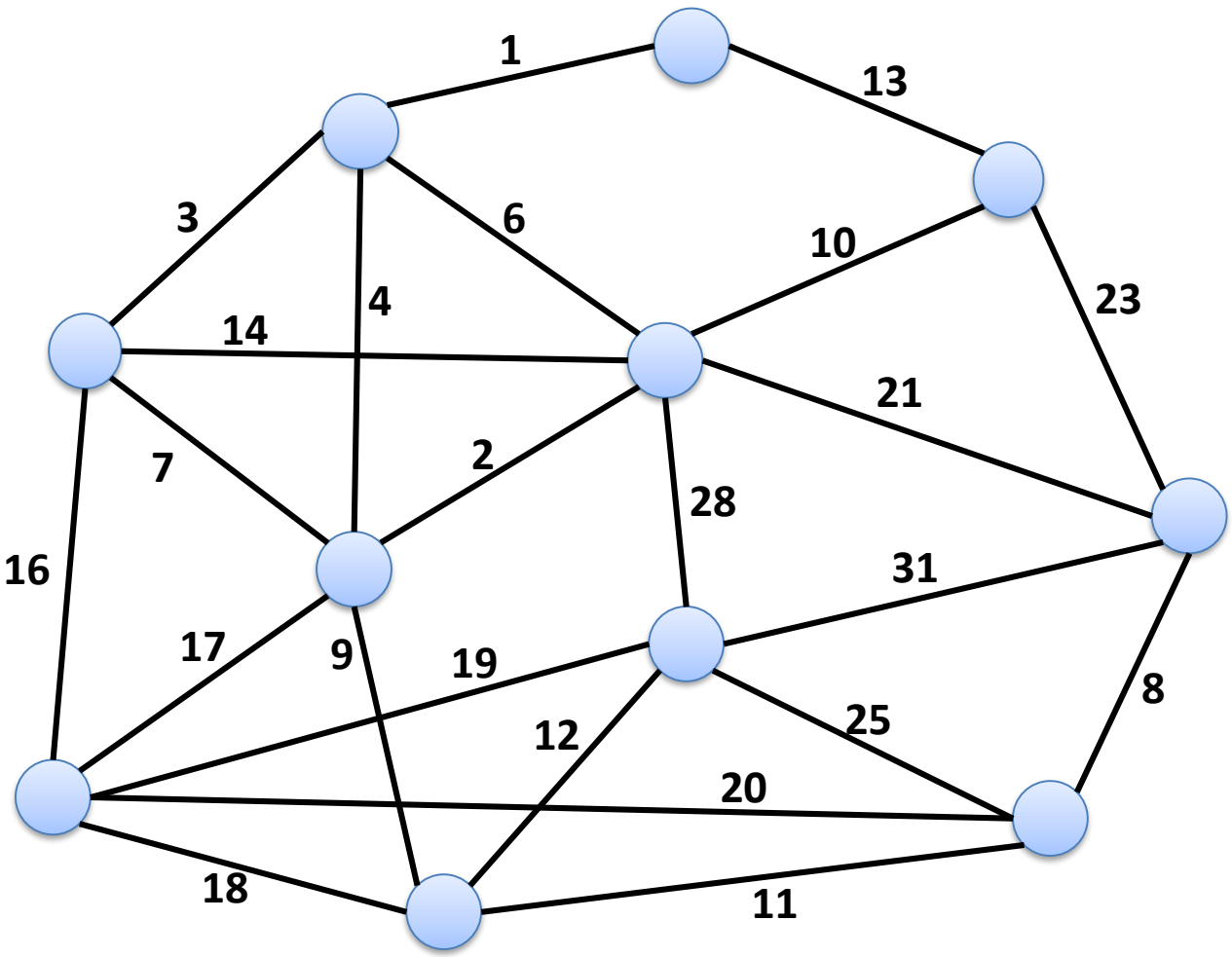
- Running time: $\boldsymbol{O(m + n \log n)}$

Chapter 5
Data Structures

Algorithm Theory
WS 2018/19

Fabian Kuhn

Kruskal Algorithm



1. Start with an empty edge set

2. In each step: Add minimum weight edge *e* such that *e* does *not* close a cycle

Implementation of Kruskal Algorithm

1. Go through edges in order of increasing weights

2. For each edge e:

   if e does not close a cycle then

   add e to the current solution

Union-Find Data Structure

Also known as **Disjoint-Set Data Structure**…

Manages partition of a set of elements

- set of disjoint sets

Operations:

- $\mathbf{make_set}(\boldsymbol{x})$**:** create a new set that only contains element x
- $\mathbf{find}(\boldsymbol{x})$**:** return the set containing x
- $\mathbf{union}(\boldsymbol{x}, \boldsymbol{y})$**:** merge the two sets containing x and y

Implementation of Kruskal Algorithm

1. Initialization:
   For each node v: $\mathrm{make_set}(v)$

2. Go through edges in order of increasing weights:
   Sort edges by edge weight

3. For each edge $e = \{u, v\}$:

   if $\mathbf{find}(\boldsymbol{u}) \neq \mathbf{find}(\boldsymbol{v})$ **then**

   add e to the current solution

   $\mathbf{union}(\boldsymbol{u}, \boldsymbol{v})$

Managing Connected Components

- Union-find data structure can be used more generally to manage the connected components of a graph

  … if edges are added incrementally

- $\text{make_set}(v)$ for every node v
- $\text{find}(v)$ returns component containing v
- $\text{union}(u, v)$ merges the components of u and v
  (when an edge is added between the components)

- Can also be used to manage biconnected components

Basic Implementation Properties

Representation of sets:

- Every set S of the partition is identified with a representative, by one of its members $x \in S$

Operations:

- $\text{make_set}(x)$: x is the representative of the new set $\{x\}$
- $\text{find}(x)$: return representative of set S_x containing x
- $\text{union}(x, y)$: unites the sets S_x and S_y containing x and y and returns the new representative of $S_x \cup S_y$

Observations

Throughout the discussion of union-find:

- n: total number of $\mathrm{make_set}$ operations
- m: total number of operations ($\mathrm{make_set}$, find, and union)

Clearly:

- $m \geq n$
- There are at most $n - 1$ union operations

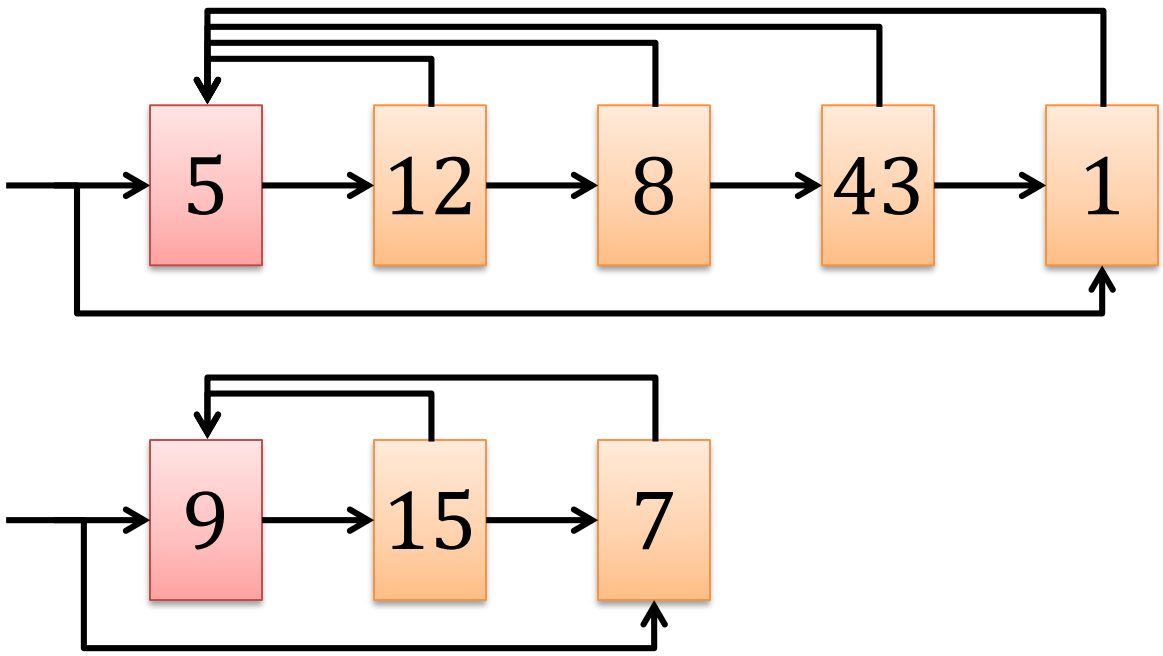
Remark:

- We assume that the n $\mathrm{make_set}$ operations are the first n operations
  - Does not really matter…

Linked List Implementation

Each set is implemented as a linked list:

- representative: first list element (all nodes point to first elem.)
  in addition: pointer to first and last element



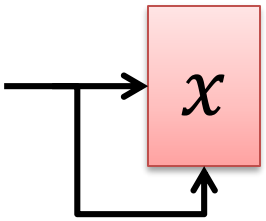
- sets: $\{1,5,8,12,43\}, \{7,9,15\}$; representatives: $5, 9$

Linked List Implementation

make_set(x):

- Create list with one element:

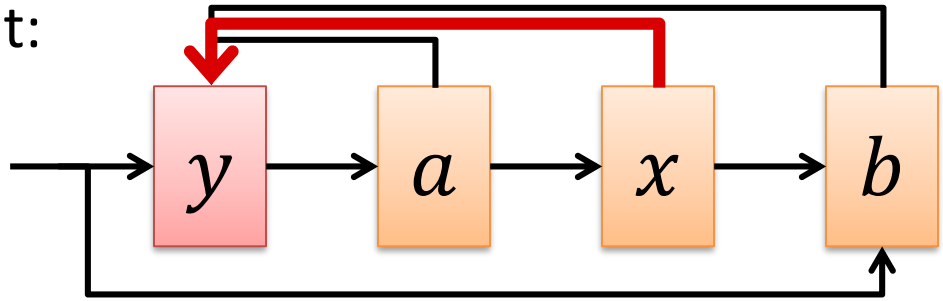
  time: $O(1)$



find(x):

- Return first list element:

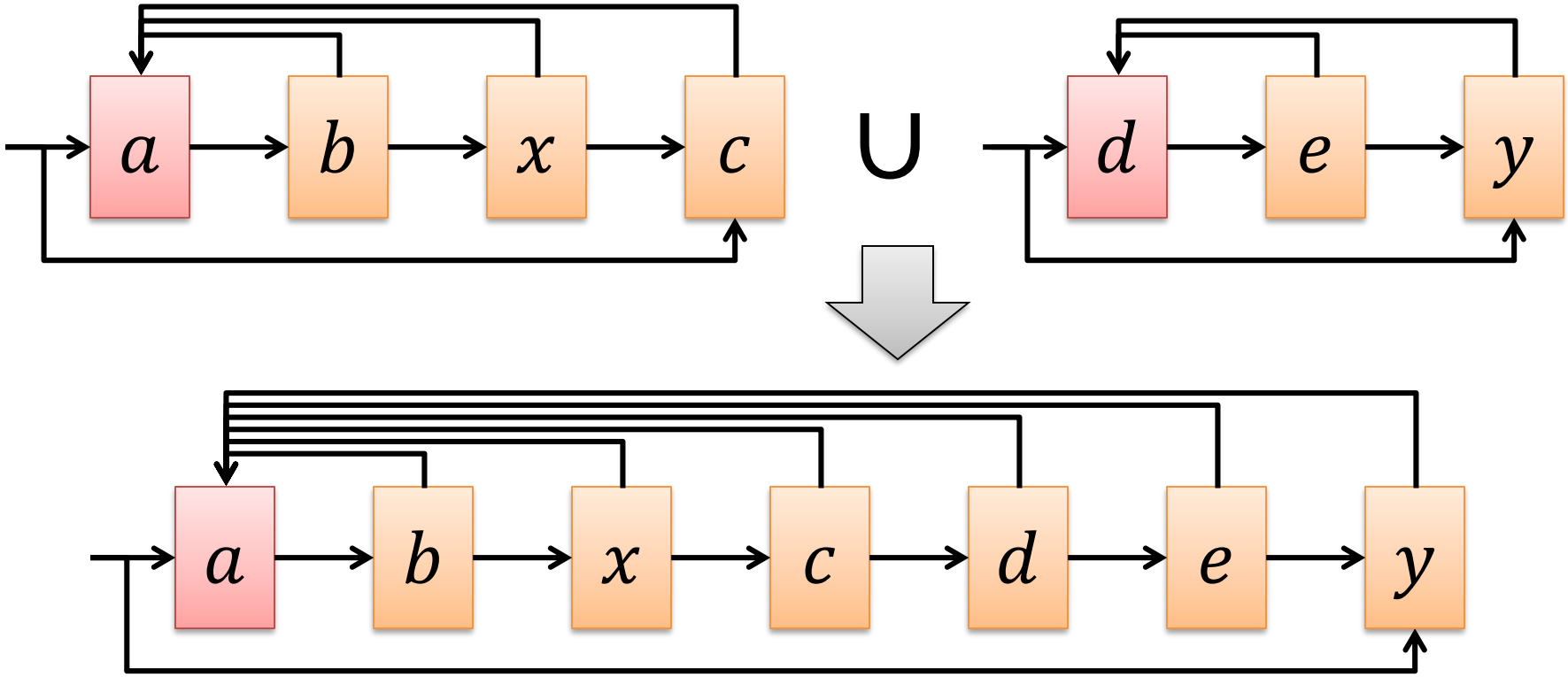
  time: $O(1)$



Linked List Implementation

union(x, y):

- Append list of y to list of x:



Time: O(length of list of y)

Cost of Union (Linked List Implementation)

Total cost for $n-1$ union operations can be $\Theta(n^2)$:

- $\text{make_set}(x_1), \text{make_set}(x_2), \dots, \text{make_set}(x_n),$
  $\text{union}(x_{n-1}, x_n), \text{union}(x_{n-2}, x_{n-1}), \dots, \text{union}(x_1, x_2)$

Weighted-Union Heuristic

- In a bad execution, average cost per union can be $\Theta(n)$
- Problem: The longer list is always appended to the shorter one

Idea:

- In each union operation, append shorter list to longer one!

Cost for union of sets S_x and S_y: $O\left(\min\{|S_x|, |S_y|\}\right)$

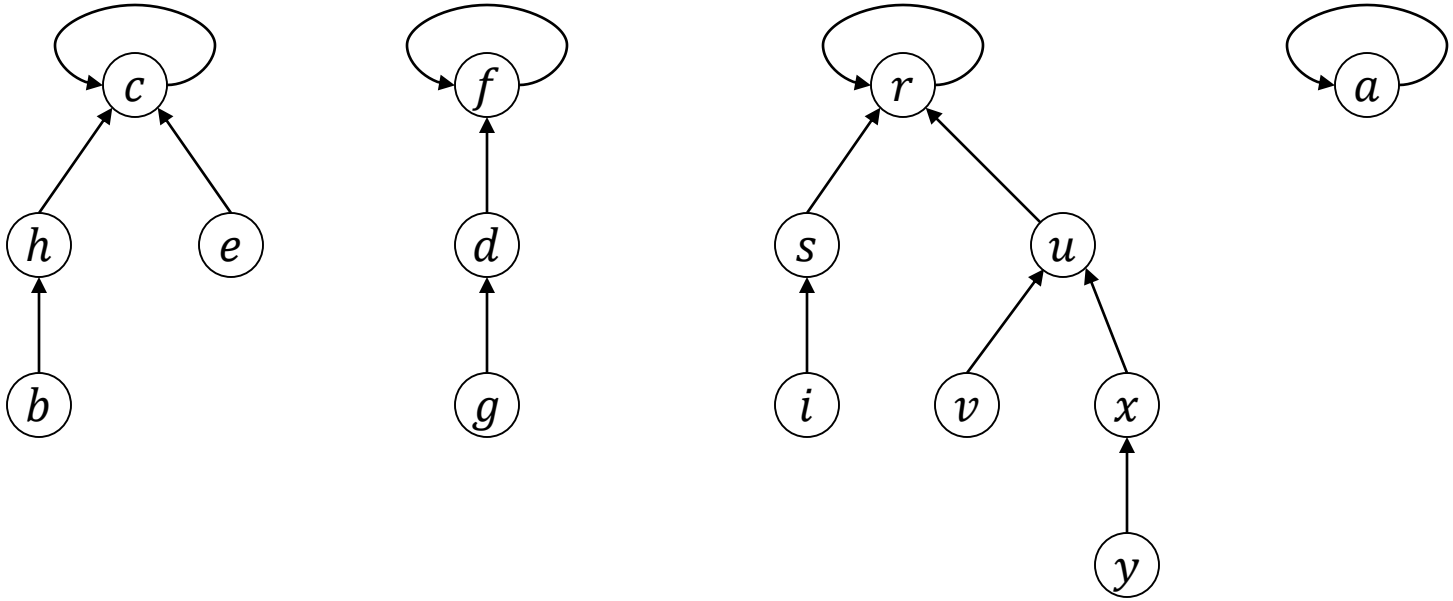
Theorem: The overall cost of m operations of which at most n are $\mathrm{make_set}$ operations is $\boldsymbol{O(m + n \log n)}$.

Weighted-Union Heuristic

Theorem: The overall cost of m operations of which at most n are $\mathrm{make_set}$ operations is $\boldsymbol{O(m + n \log n)}$.

Proof:

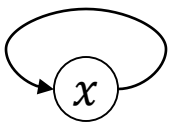
Disjoint-Set Forests



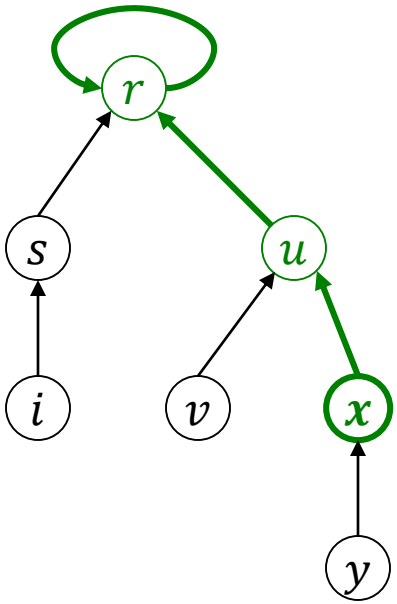
- Represent each set by a tree
- Representative of a set is the root of the tree

Disjoint-Set Forests

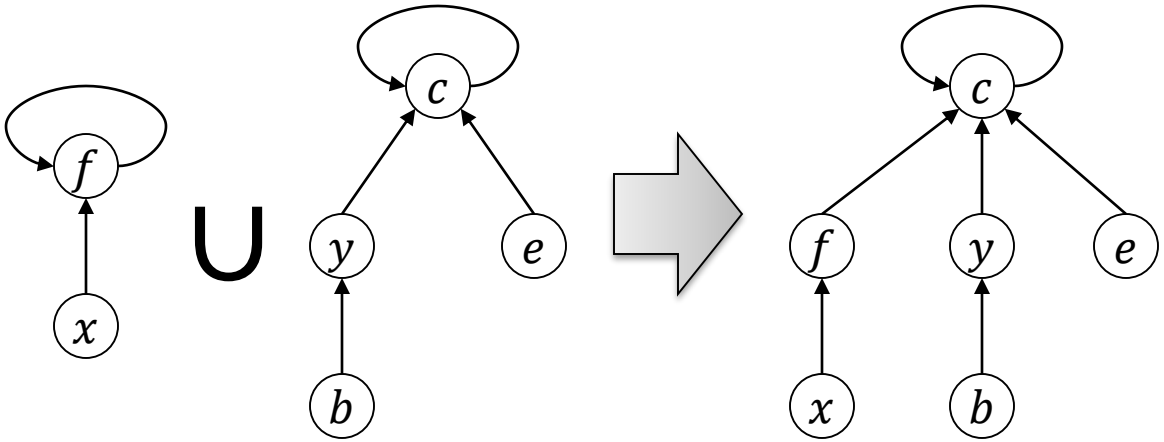
make_set(x): create new one-node tree



find(x): follow parent point to root
(parent pointer to itself)



union(x, y): attach tree of x to tree of y



Bad Sequence

Bad sequence leads to tree(s) of depth $\Theta(n)$

- $\text{make_set}(x_1), \text{make_set}(x_2), \dots, \text{make_set}(x_n),$
  $\text{union}(x_1, x_2), \text{union}(x_1, x_3), \dots, \text{union}(x_1, x_n)$

Union-By-Size Heuristic

Union of sets S_1 and S_2:

- Root of trees representing S_1 and S_2: r_1 and r_2
- W.l.o.g., assume that $|S_1| \geq |S_2|$
- Root of $S_1 \cup S_2$: r_1 (r_2 is attached to r_1 as a new child)

Theorem: If the union-by-size heuristic is used, the **worst-case cost of a find-operation is $O(\log n)$**

Proof:

Similar Strategy: union-by-rank

- rank: essentially the depth of a tree

Union-Find Algorithms

Recall: m operations, n of the operations are make_set-operations

Linked List with Weighted Union Heuristic:

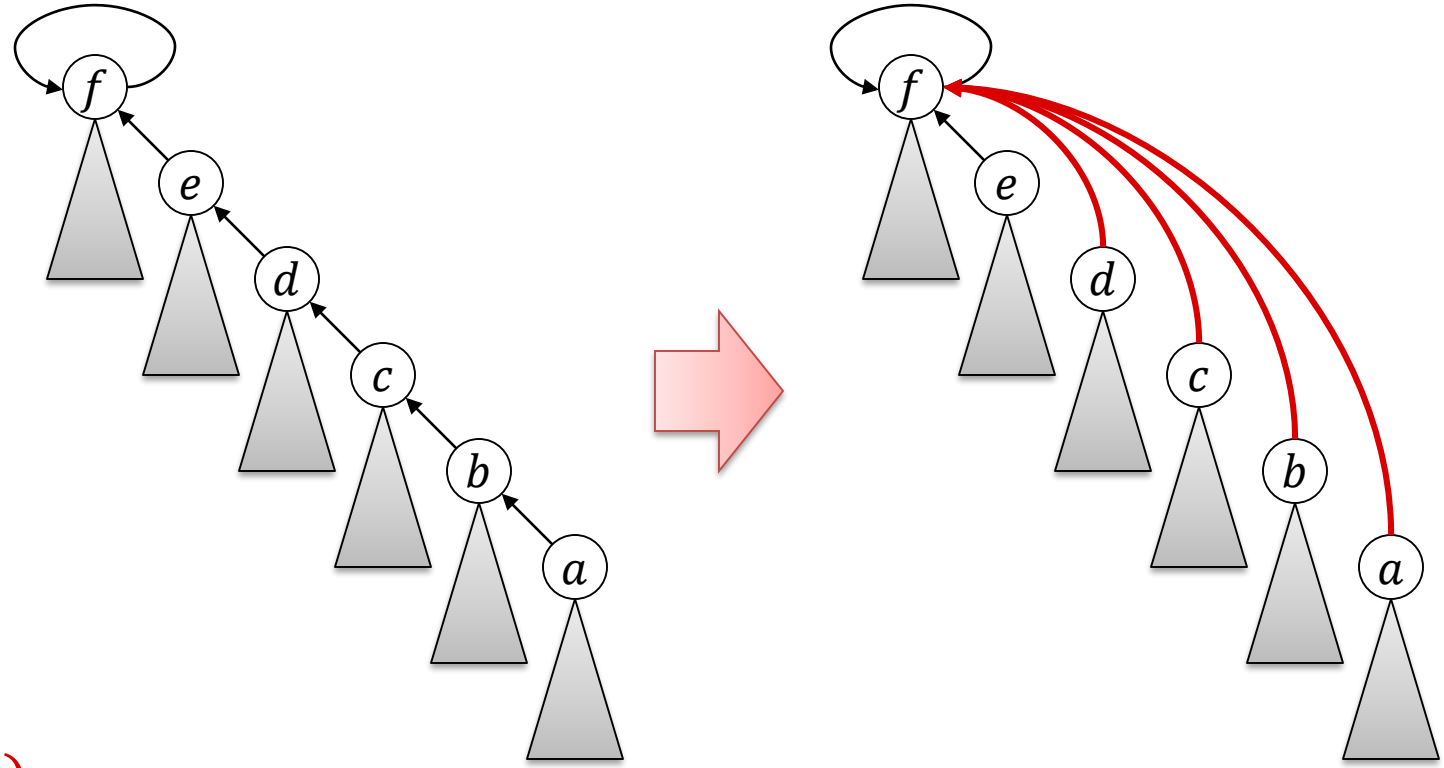
- make_set: worst-case cost $O(1)$
- find : worst-case cost $O(1)$
- union : amortized worst-case cost $O(\log n)$

Disjoint-Set Forest with Union-By-Size Heuristic:

- make_set: worst-case cost $O(1)$
- find : worst-case cost $O(\log n)$
- union : worst-case cost $O(\log n)$

Can we make this faster?

Path Compression During Find Operation



find(a):

1. **if** $a \neq a.parent$ **then**
2. $\quad a.parent := \text{find}(a.parent)$
3. **return** $a.parent$

Complexity With Path Compression

When using only path compression (without union-by-rank):

m: total number of operations

- f of which are find-operations
- n of which are make_set-operations
  → at most $n-1$ are union-operations

Total cost: $\mathbf{O}\left(\boldsymbol{m}+\boldsymbol{f}\cdot\left\lceil\mathbf{log}_{2+f/n}\,\boldsymbol{n}\right\rceil\right)=\boldsymbol{O}\left(\boldsymbol{m}+\boldsymbol{f}\cdot\mathbf{log}_{2+m/n}\boldsymbol{n}\right)$

Union-By-Size and Path Compression

Theorem:

Using the combined union-by-rank and path compression heuristic, the running time of m disjoint-set (union-find) operations on n elements (at most n $\mathrm{make_set}$-operations) is

$$\boldsymbol{\Theta\big(m \cdot \alpha(m,n)\big)},$$

Where $\alpha(m,n)$ is the inverse of the Ackermann function.

Ackermann Function and its Inverse

Ackermann Function:

For $k, \ell \geq 1$,

$$A(k,\ell) := \begin{cases} 2^{\ell}, & \text{if } k = 1, \ell \geq 1 \\ A(k-1, 2), & \text{if } k > 1, \ell = 1 \\ A\big(k-1, A(k, \ell-1)\big), & \text{if } k > 1, \ell > 1 \end{cases}$$

Inverse of Ackermann Function:

$$\alpha(m,n) := \min\{k \geq 1 \mid A(k, \lfloor m/n \rfloor) > \log_2 n\}$$

Inverse of Ackermann Function

- $\alpha(m,n) \coloneqq \min\{k \geq 1 \mid A(k, \lfloor {}^{m}/_{n} \rfloor) > \log_2 n\}$

  $m \geq n \Longrightarrow A(k, \lfloor {}^{m}/_{n} \rfloor) \geq A(k,1) \Longrightarrow \alpha(m,n) \leq \min\{k \geq 1 | A(k,1) > \log n\}$

- $A(1,\ell) = 2^{\ell}, \quad A(k,1) = A(k-1,2),$
  $A(k,\ell) = A\big(k-1, A(k,\ell-1)\big)$

Chapter 6
Graph Algorithms

Algorithm Theory
WS 2018/19

Fabian Kuhn

Graphs

Extremely important concept in computer science

Graph $\boldsymbol{G = (V, E)}$

- V: node (or vertex) set
- $E \subseteq V^2$: edge set
  - Simple graph: no self-loops, no multiple edges
  - Undirected graph: we often think of edges as sets of size 2 (e.g., $\{u, v\}$)
  - Directed graph: edges are sometimes also called arcs
  - Weighted graph: (positive) weight on edges (or nodes)
- (simple) path: sequence $v_0, \dots, v_k$ of nodes such that
  $$(v_i, v_{i+1}) \in E \text{ for all } i \in \{0, \dots, k-1\}$$
- …

Many real-world problems can be formulated as optimization problems on graphs

Graph Optimization: Examples

Minimum spanning tree (MST):

- Compute min. weight spanning tree of a weighted undir. Graph

Shortest paths:

- Compute (length) of shortest paths (single source, all pairs, …)

Traveling salesperson (TSP):

- Compute shortest TSP path/tour in weighted graph

Vertex coloring:

- Color the nodes such that neighbors get different colors
- Goal: minimize the number of colors

Maximum matching:

- Matching: set of pair-wise non-adjacent edges
- Goal: maximize the size of the matching

Network Flow

Flow Network:

- Directed graph $G = (V, E)$, $E \subseteq V^2$
- Each (directed) edge e has a capacity $c_e \geq 0$
  - Amount of flow (traffic) that the edge can carry
- A single source node $s \in V$ and a single sink node $t \in V$

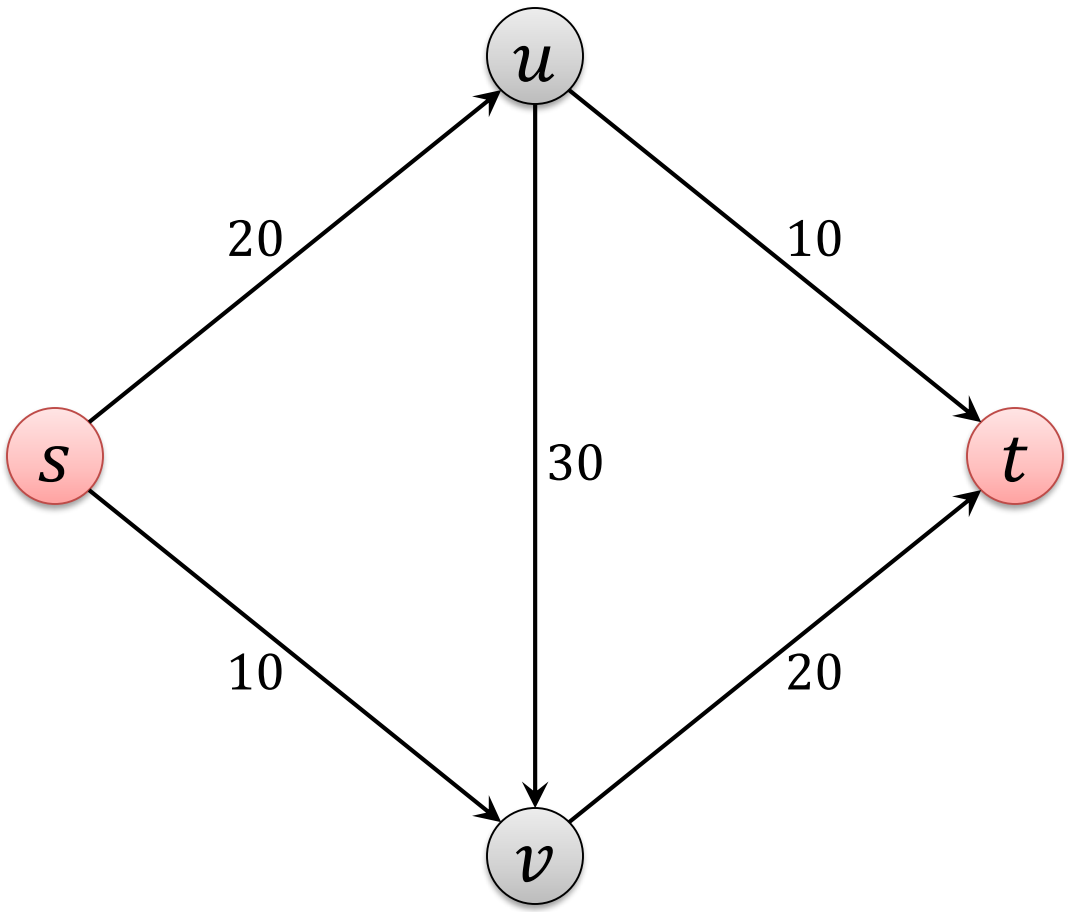
Flow: (informally)

- Traffic from s to t such that each edge carries at most its capacity

Examples:

- Highway system: edges are highways, flow is the traffic
- Computer network: edges are network links that can carry packets, nodes are switches
- Fluid network: edges are pipes that carry liquid

Example: Flow Network



Network Flow: Definition

Flow: function $\boldsymbol{f}: \boldsymbol{E} \to \mathbb{R}_{\geq 0}$

- $f(e)$ is the amount of flow carried by edge e

Capacity Constraints:

- For each edge $e \in E$, $f(e) \leq c_e$

Flow Conservation:

- For each node $v \in V \setminus \{s, t\}$,

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$$

Flow Value:

$$|f| := \sum_{e \text{ out of } s} f\big((s,u)\big) = \sum_{e \text{ into } t} f\big((v,t)\big)$$

Notation

We define:

$$f^{\text{in}}(v) := \sum_{e \text{ into } v} f(e)\,, \qquad f^{\text{out}}(v) := \sum_{e \text{ out of } v} f(e)$$

For a set $S \subseteq V$:

$$f^{\text{in}}(S) := \sum_{e \text{ into } S} f(e)\,, \qquad f^{\text{out}}(S) := \sum_{e \text{ out of } S} f(e)$$

Flow conservation: $\forall v \in V \setminus \{s,t\}$: $f^{\text{in}}(v) = f^{\text{out}}(v)$

Flow value: $|f| = f^{\text{out}}(s) = f^{\text{in}}(t)$

For simplicity: Assume that all capacities are positive integers

The Maximum-Flow Problem

Maximum Flow:

Given a flow network, find a flow of maximum possible value

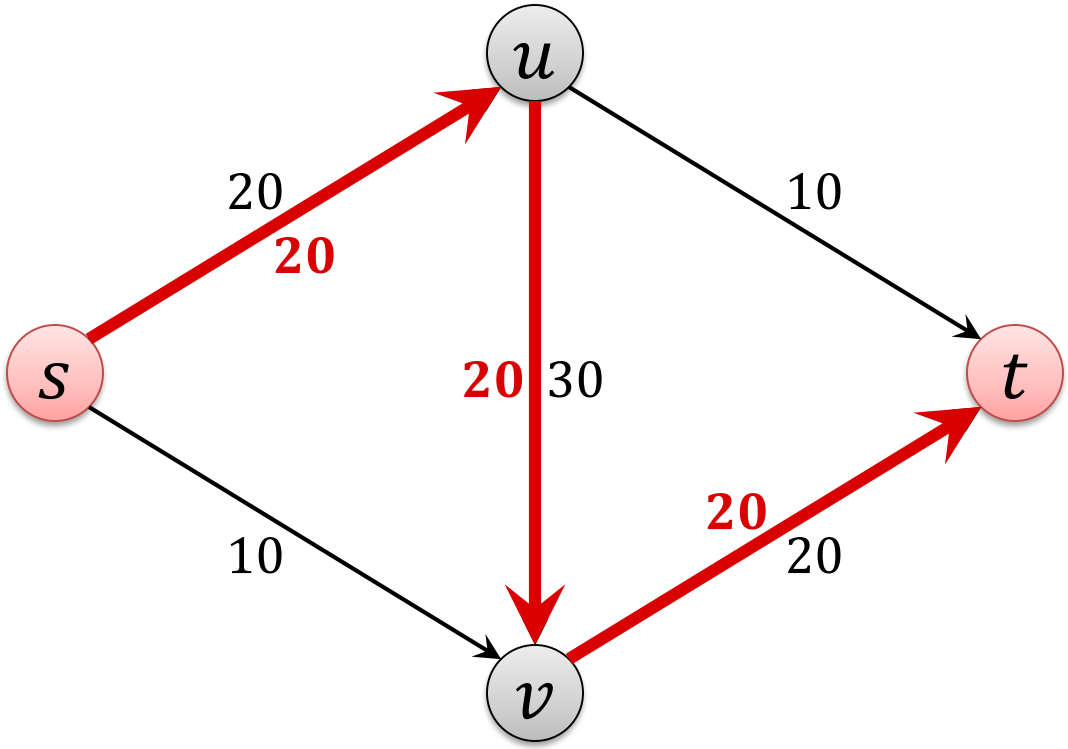
- Classical graph optimization problem
- Many applications (also beyond the obvious ones)
- Requires new algorithmic techniques

Maximum Flow: Greedy?

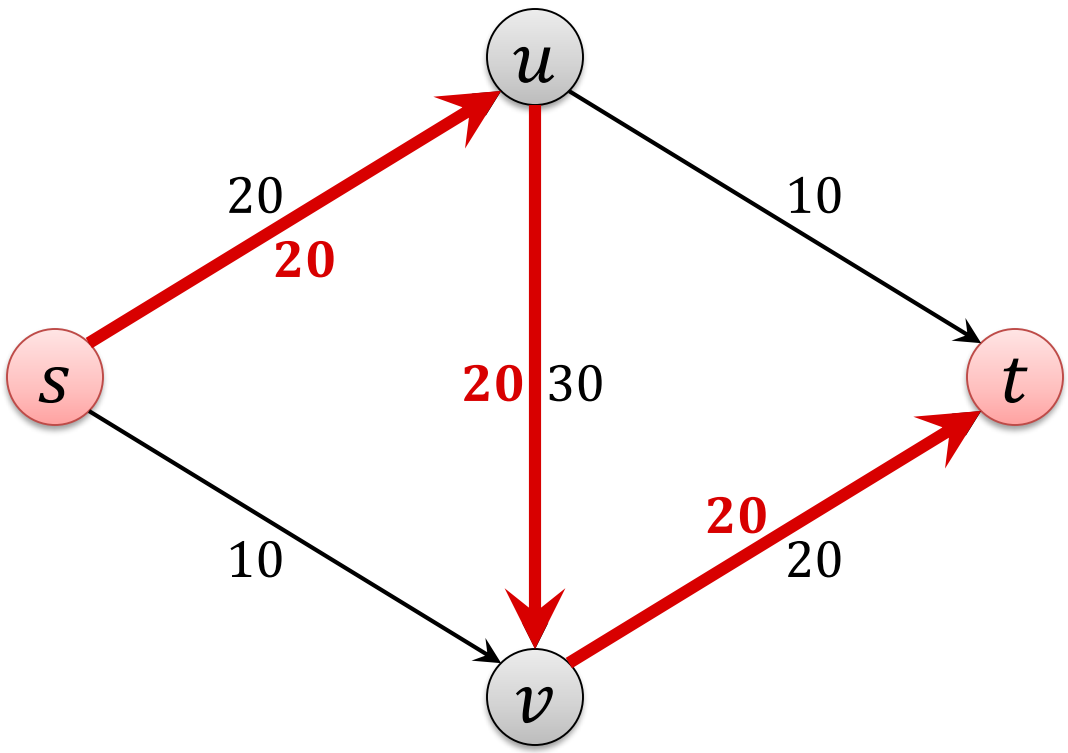
Does greedy work?

A natural greedy algorithm:

- As long as possible, find an s-t-path with free capacity and add as much flow as possible to the path



Improving the Greedy Solution



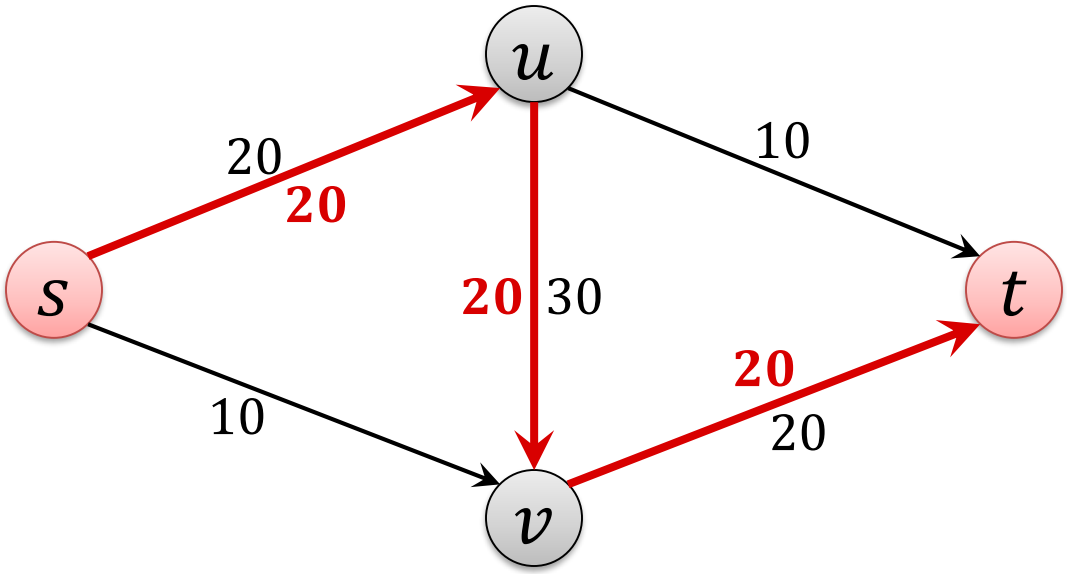
- Try to push 10 units of flow on edge (s, v)
- Too much incoming flow at v: reduce flow on edge (u, v)
- Add that flow on edge (u, t)

Residual Graph

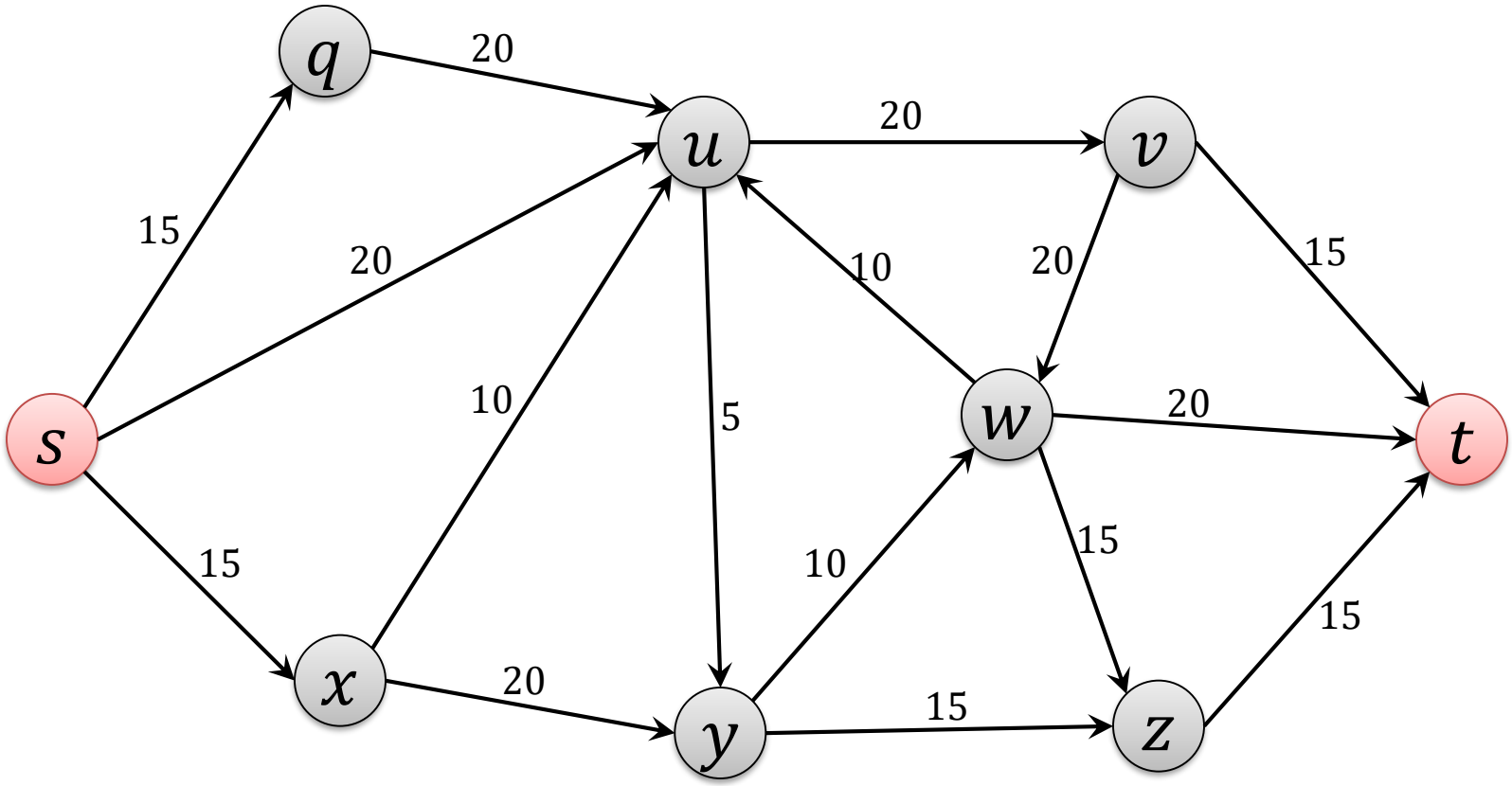
Given a flow network $G = (V, E)$ with capacities c_e (for $e \in E$)

For a flow f on G, define directed graph $G_f = (V_f, E_f)$ as follows:

- Node set $V_f = V$
- For each edge $e = (u, v)$ in E, there are two edges in E_f:
  - forward edge $e = (u, v)$ with residual capacity $c_e - f(e)$
  - backward edge $e' = (v, u)$ with residual capacity $f(e)$

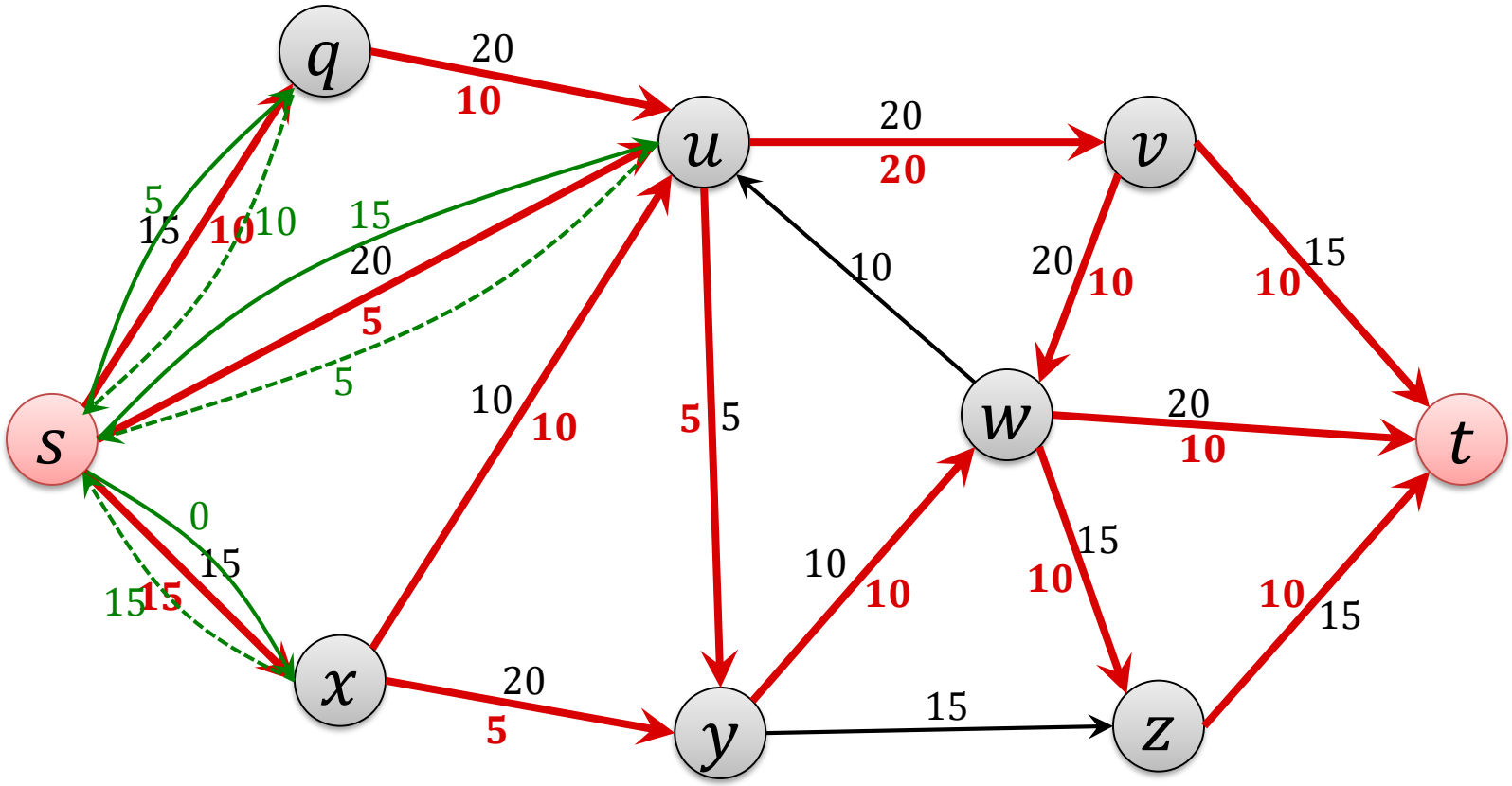


Residual Graph: Example



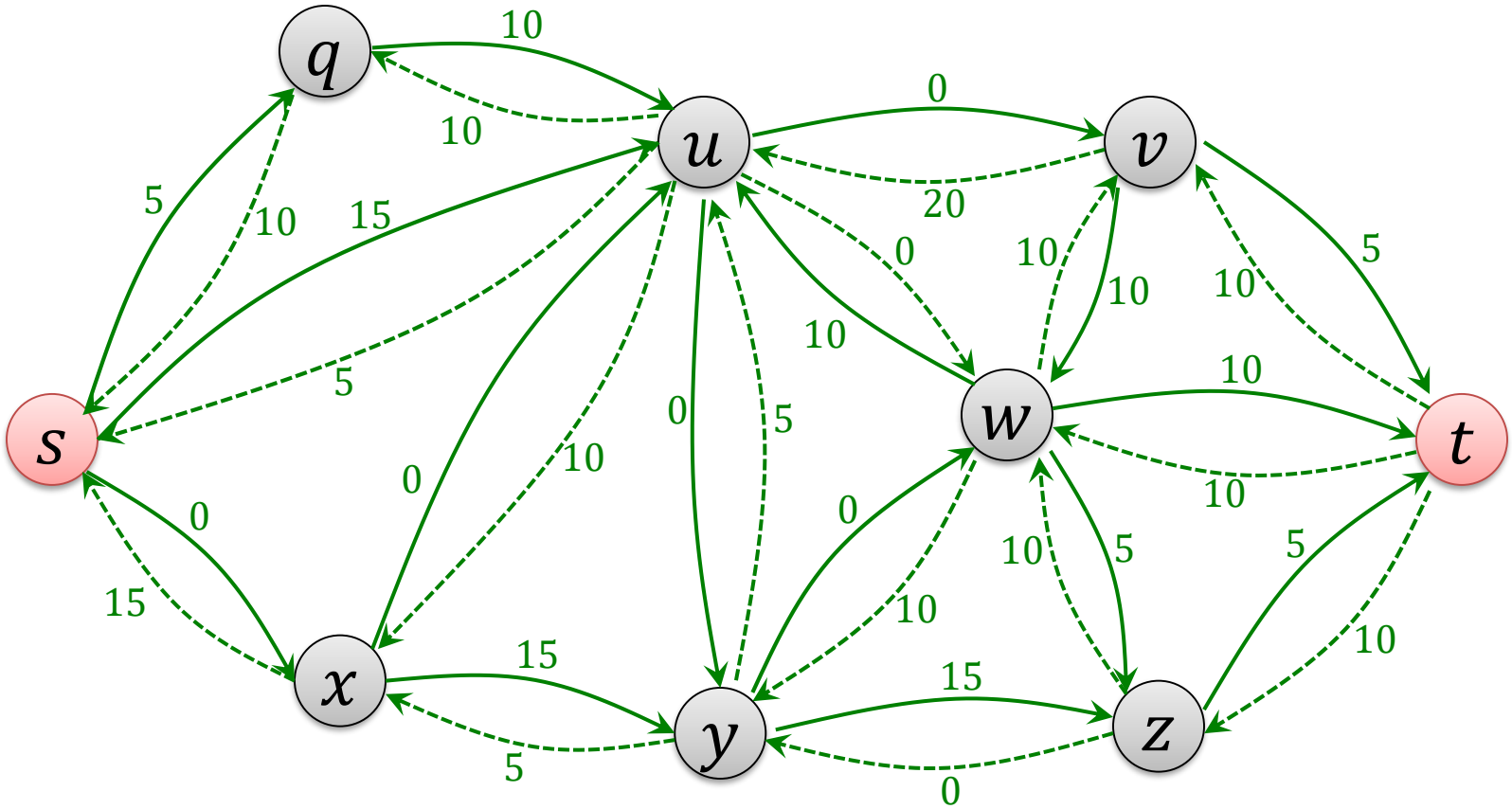
Residual Graph: Example

Flow f



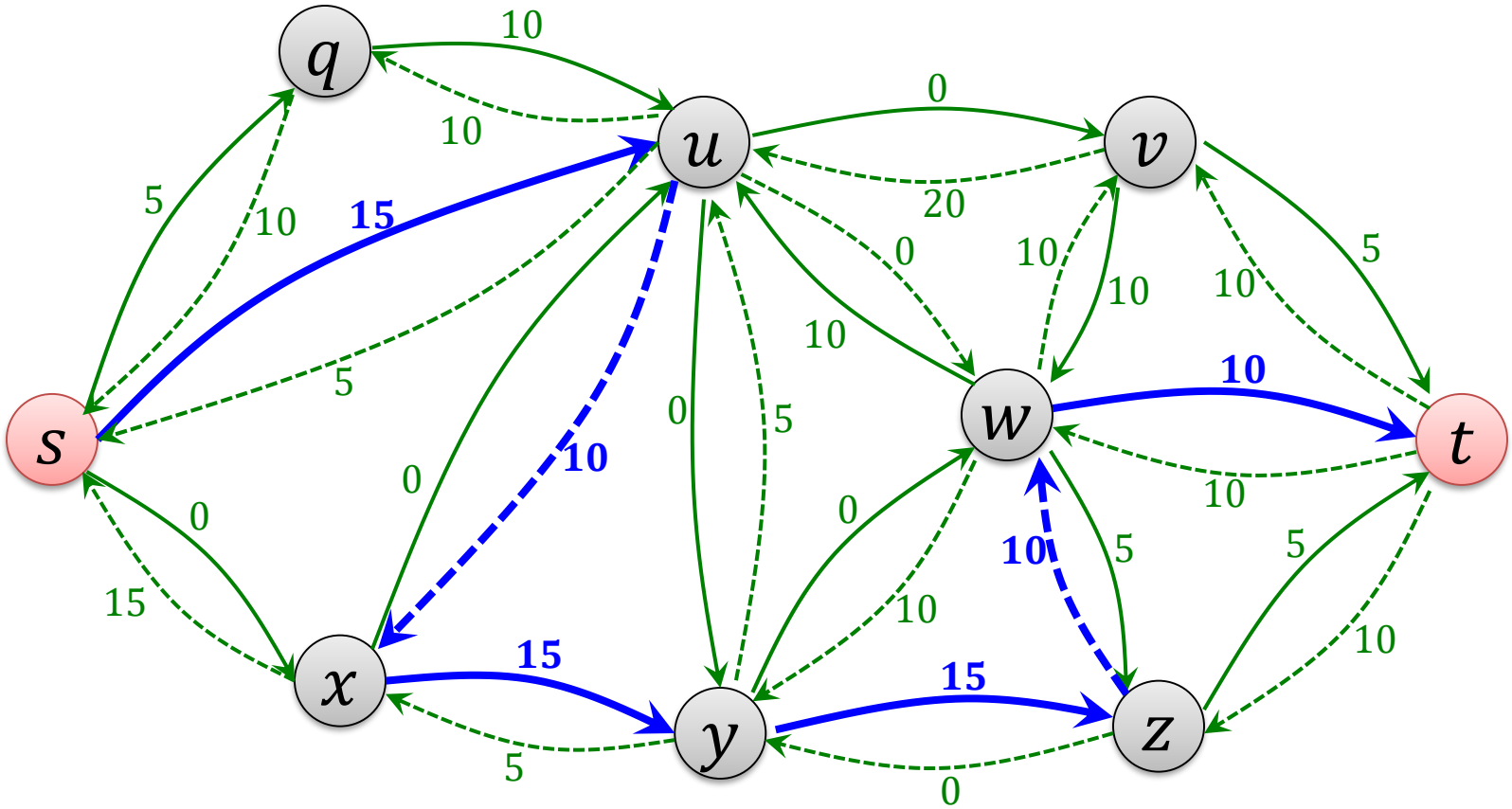
Residual Graph: Example

Residual Graph G_f

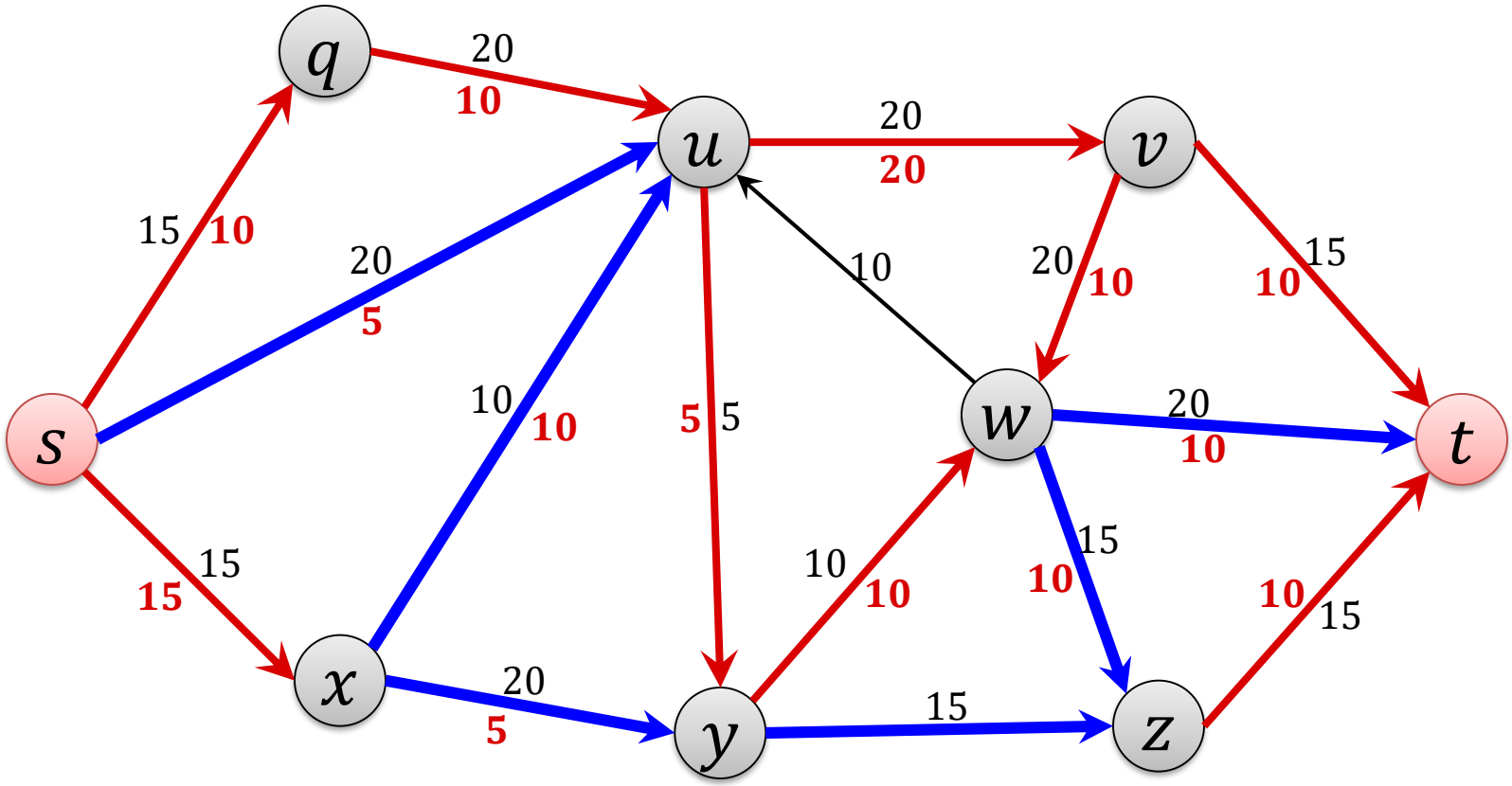


Augmenting Path

Residual Graph G_f

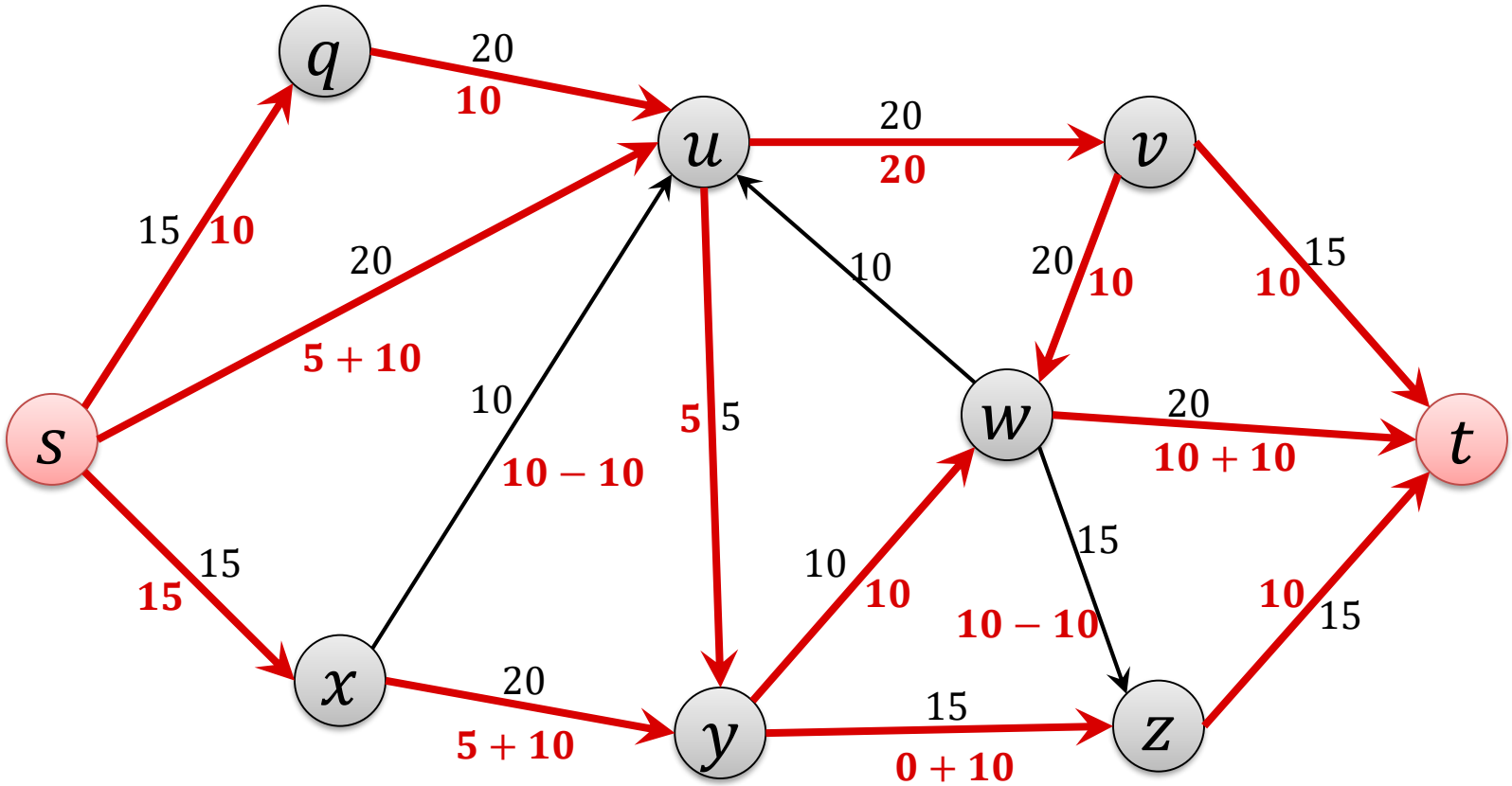


Augmenting Path



Augmenting Path

New Flow

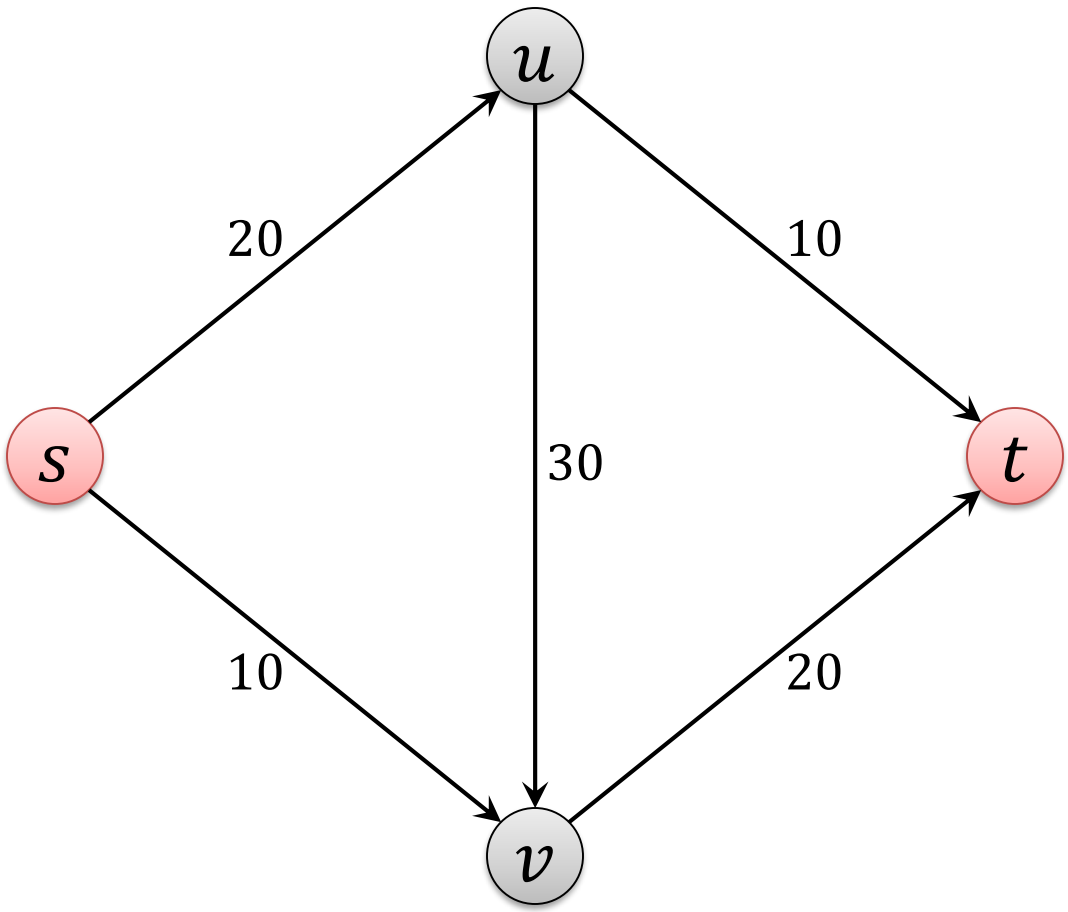


Chapter 6
Graph Algorithms

Algorithm Theory
WS 2018/19

Fabian Kuhn

Example: Flow Network



Network Flow: Definition

Flow: function $\boldsymbol{f}: \boldsymbol{E} \to \mathbb{R}_{\geq 0}$

- $f(e)$ is the amount of flow carried by edge e

Capacity Constraints:

- For each edge $e \in E$, $f(e) \leq c_e$

Flow Conservation:

- For each node $v \in V \setminus \{s, t\}$,

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$$

Flow Value:

$$|f| := \sum_{e \text{ out of } s} f\big((s,u)\big) = \sum_{e \text{ into } t} f\big((v,t)\big)$$

The Maximum-Flow Problem

Maximum Flow:

Given a flow network, find a flow of maximum possible value

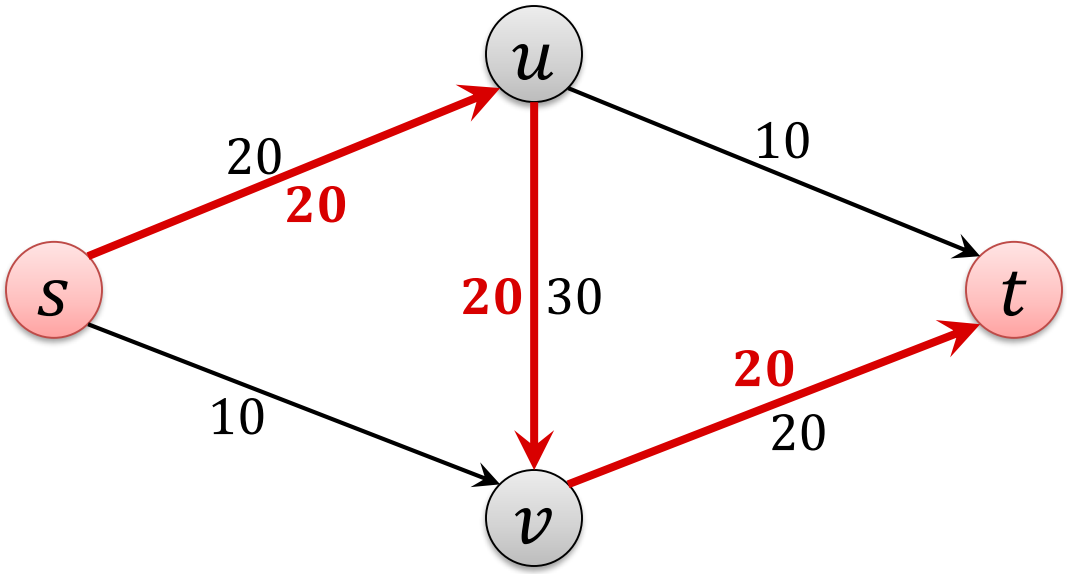
- Classical graph optimization problem
- Many applications (also beyond the obvious ones)
- Requires new algorithmic techniques

Residual Graph

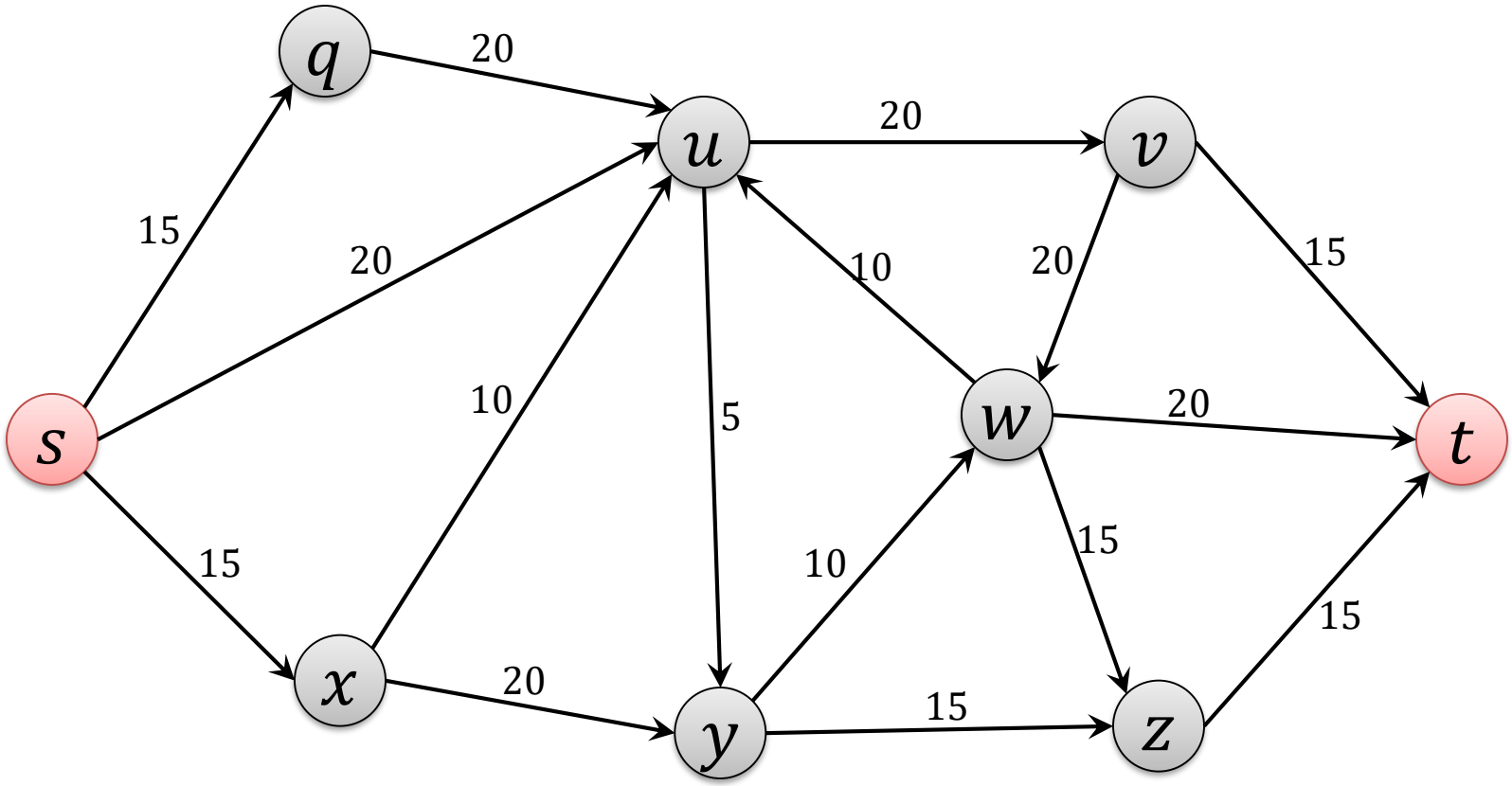
Given a flow network $G = (V, E)$ with capacities c_e (for $e \in E$)

For a flow f on G, define directed graph $G_f = (V_f, E_f)$ as follows:

- Node set $V_f = V$
- For each edge $e = (u, v)$ in E, there are two edges in E_f:
  - forward edge $e = (u, v)$ with residual capacity $c_e - f(e)$
  - backward edge $e' = (v, u)$ with residual capacity $f(e)$

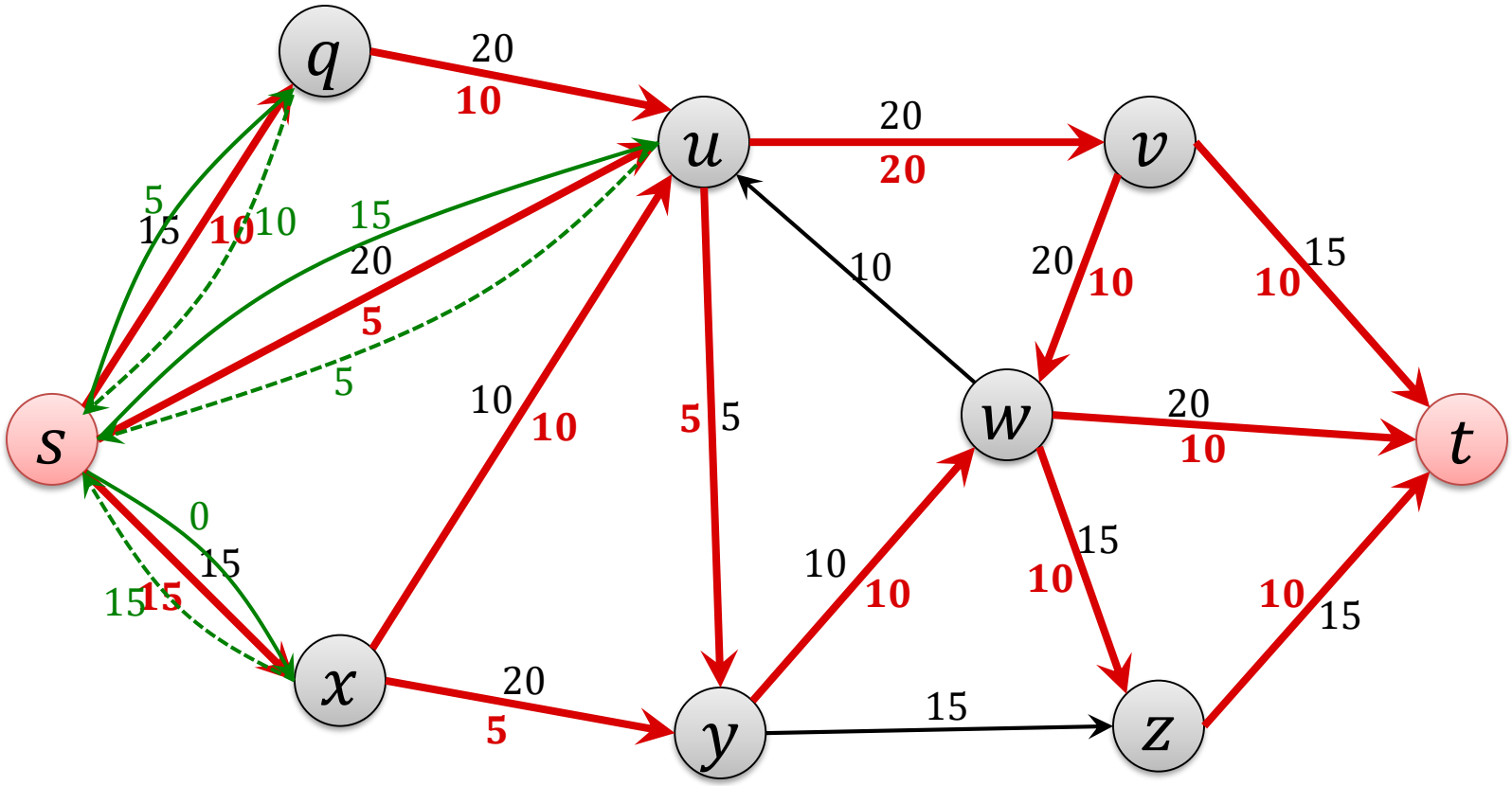


Residual Graph: Example



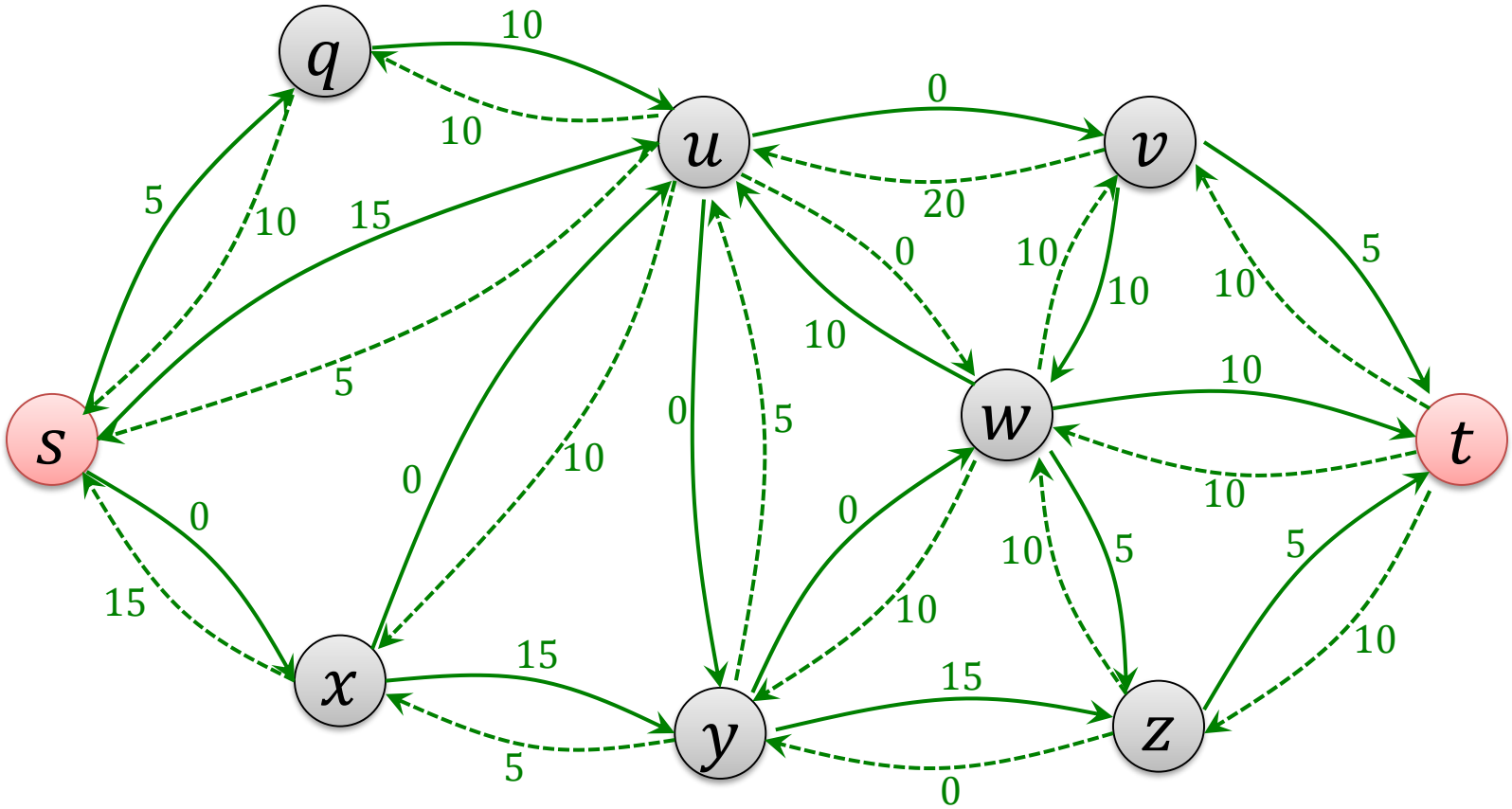
Residual Graph: Example

Flow f



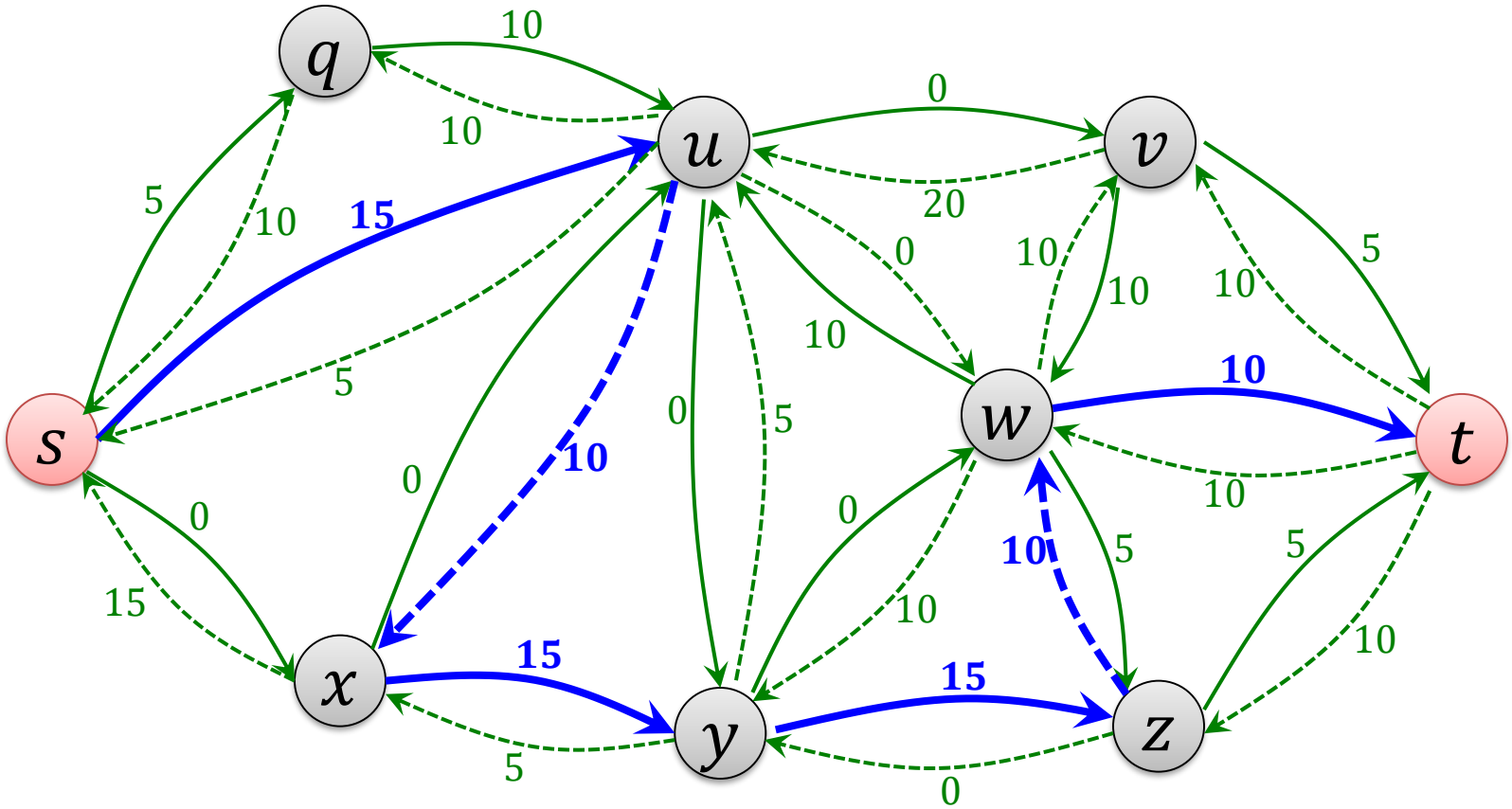
Residual Graph: Example

Residual Graph G_f

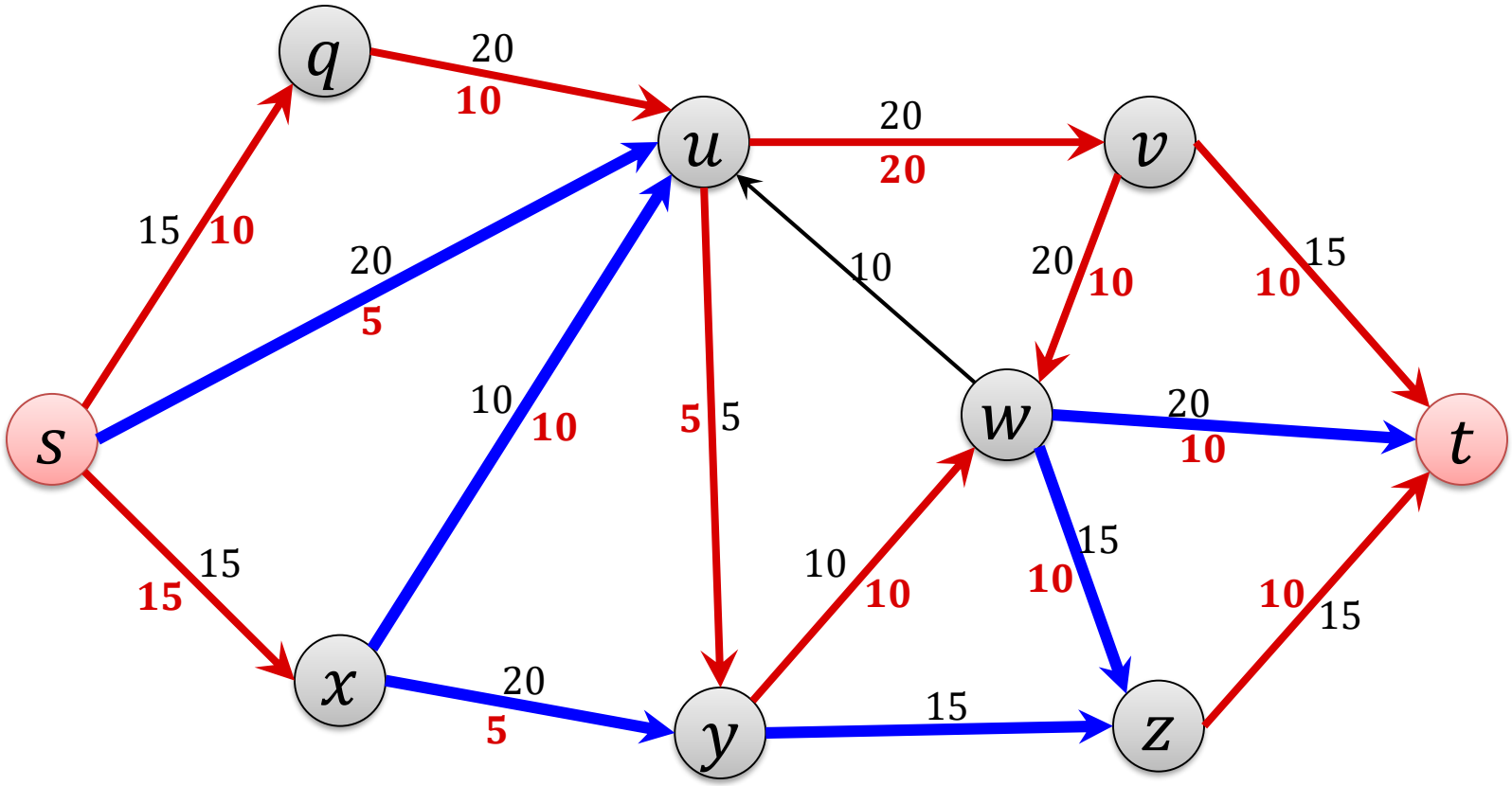


Augmenting Path

Residual Graph G_f

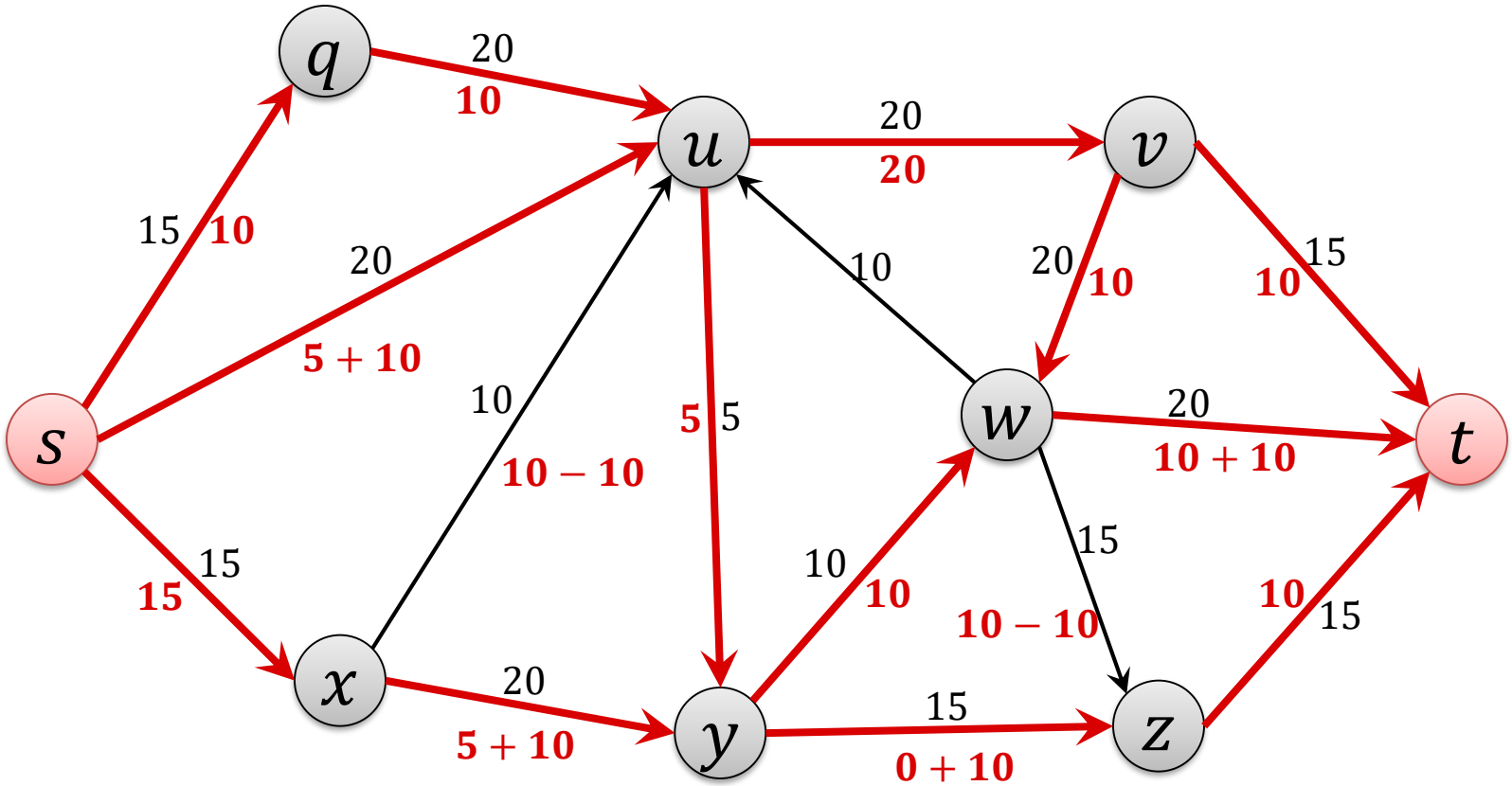


Augmenting Path



Augmenting Path

New Flow



Augmenting Path

Definition:

An augmenting path P is a (simple) s-t-path on the residual graph G_f on which each edge has residual capacity > 0.

$\text{bottleneck}(P, f)$: minimum residual capacity on any edge of the augmenting path P

Augment flow f to get flow f':

- For every forward edge (u, v) on P:

$$\boldsymbol{f'((u,v)) := f((u,v)) + \text{bottleneck}(P, f)}$$

- For every backward edge (u, v) on P:

$$\boldsymbol{f'((v,u)) := f((v,u)) - \text{bottleneck}(P, f)}$$

Augmented Flow

Lemma: Given a flow f and an augmenting path P, the resulting augmented flow f' is legal and its value is

$$|\boldsymbol{f}'| = |\boldsymbol{f}| + \mathbf{bottleneck}(\boldsymbol{P}, \boldsymbol{f}).$$

Proof:

Augmented Flow

Lemma: Given a flow f and an augmenting path P, the resulting augmented flow f' is legal and its value is

$$|\boldsymbol{f}'| = |\boldsymbol{f}| + \mathbf{bottleneck}(\boldsymbol{P}, \boldsymbol{f}).$$

Proof:

Ford-Fulkerson Algorithm

- Improve flow using an augmenting path as long as possible:

1. Initially, $f(e) = 0$ for all edges $e \in E$, $G_f = G$
2. **while** there is an augmenting s-t-path P in G_f **do**
3. Let P be an augmenting s-t-path in G_f;
4. $f' := \text{augment}(f, P)$;
5. update f to be f';
6. update the residual graph G_f
7. **end**;

Ford-Fulkerson Running Time

Theorem: If all edge capacities are integers, the Ford-Fulkerson algorithm terminates after at most C iterations, where

$$C = \text{"max flow value"} \leq \sum_{e \text{ out of } s} c_e \,.$$

Proof:

Ford-Fulkerson Running Time

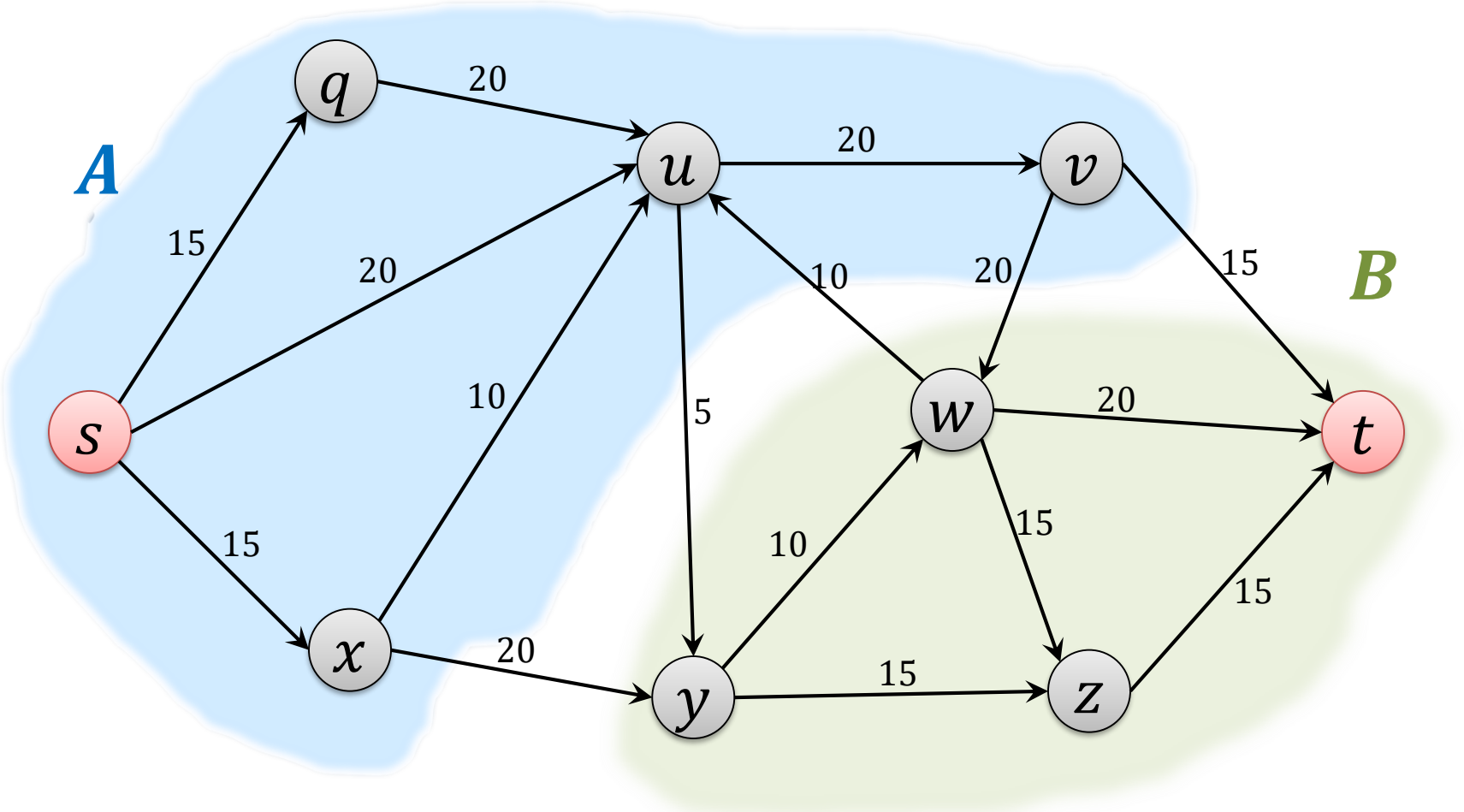
Theorem: If all edge capacities are integers, the Ford-Fulkerson algorithm can be implemented to run in $O(mC)$ time.

Proof:

s-t Cuts

Definition:

An s-t cut is a partition (A, B) of the vertex set such that $s \in A$ and $t \in B$

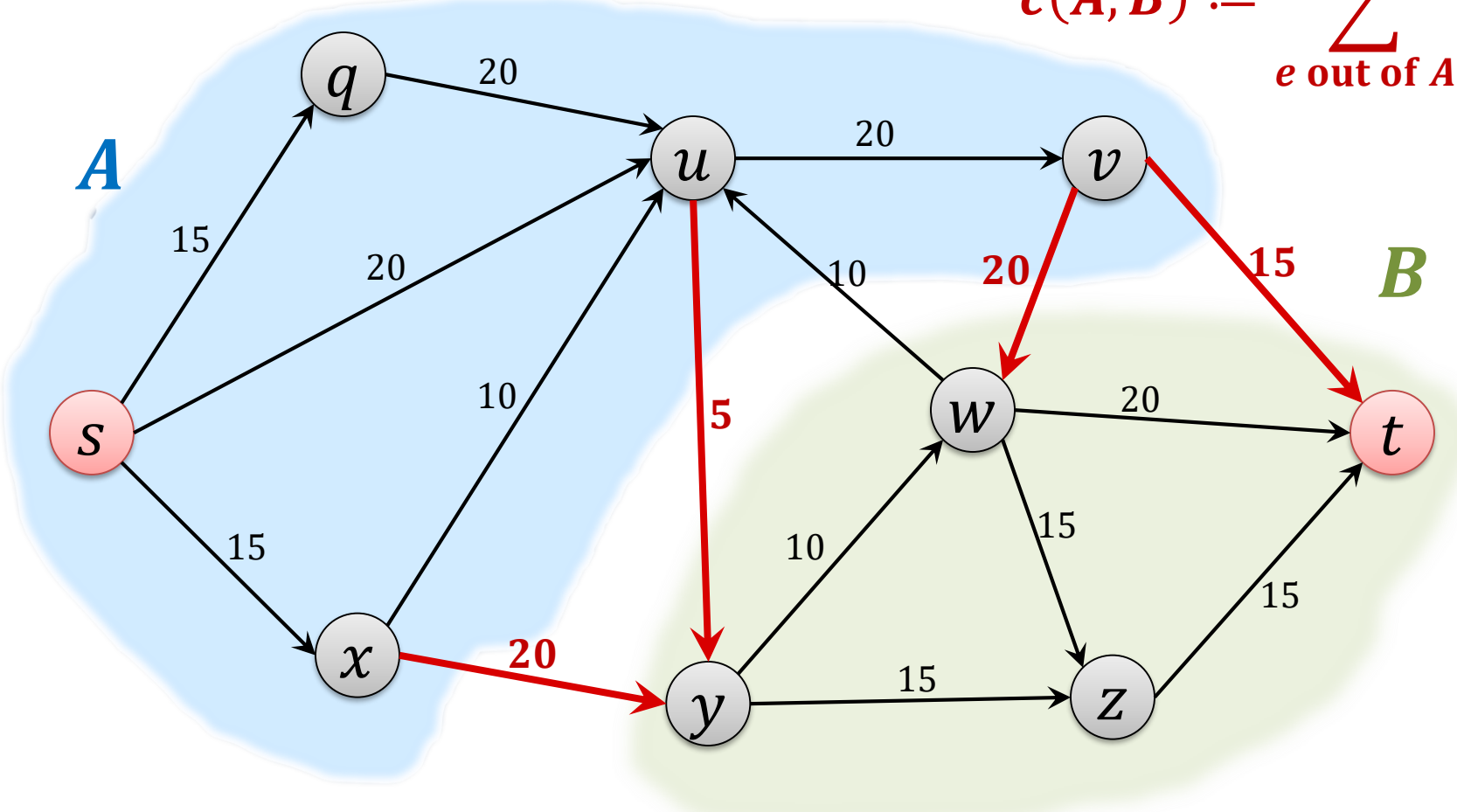


Cut Capacity

Definition:

The capacity $c(A,B)$ of an s-t-cut (A,B) is defined as

$$c(A,B) := \sum_{e \text{ out of } A} c_e \,.$$



Cuts and Flow Value

Lemma: Let f be any s-t flow, and (A, B) any s-t cut. Then,

$$|\boldsymbol{f}| = \boldsymbol{f}^{\mathbf{out}}(\boldsymbol{A}) - \boldsymbol{f}^{\mathbf{in}}(\boldsymbol{A}).$$

Proof:

Cuts and Flow Value

Lemma: Let f be any s-t flow, and (A, B) any s-t cut. Then,

$$|\boldsymbol{f}| = \boldsymbol{f}^{\mathbf{out}}(\boldsymbol{A}) - \boldsymbol{f}^{\mathbf{in}}(\boldsymbol{A}).$$

Lemma: Let f be any s-t flow, and (A, B) any s-t cut. Then,

$$|\boldsymbol{f}| = \boldsymbol{f}^{\mathbf{in}}(\boldsymbol{B}) - \boldsymbol{f}^{\mathbf{out}}(\boldsymbol{B}).$$

Proof:

Upper Bound on Flow Value

Lemma:

Let f be any s-t flow and (A, B) any s-t cut. Then $\boldsymbol{|f| \leq c(A, B)}$.

Proof:

Ford-Fulkerson Gives Optimal Solution

Lemma: If f is an s-t flow such that there is no augmenting path in G_f, then there is an s-t cut (A^*, B^*) in G for which

$$|\boldsymbol{f}| = \boldsymbol{c}(\boldsymbol{A}^*, \boldsymbol{B}^*).$$

Proof:

- Define $\boldsymbol{A}^*$: set of nodes that can be reached from s on a path with positive residual capacities in G_f:

- For $B^* = V \setminus A^*$, (A^*, B^*) is an s-t cut
  - By definition $s \in A^*$ and $t \notin A^*$

Ford-Fulkerson Gives Optimal Solution

Lemma: If f is an s-t flow such that there is no augmenting path in G_f, then there is an s-t cut (A^*, B^*) in G for which

$$|\boldsymbol{f}| = \boldsymbol{c}(\boldsymbol{A}^*, \boldsymbol{B}^*).$$

Proof:

Ford-Fulkerson Gives Optimal Solution

Lemma: If f is an s-t flow such that there is no augmenting path in G_f, then there is an s-t cut (A^*, B^*) in G for which

$$|\boldsymbol{f}| = \boldsymbol{c}(\boldsymbol{A}^*, \boldsymbol{B}^*).$$

Proof:

Ford-Fulkerson Gives Optimal Solution

Theorem: The flow returned by the Ford-Fulkerson algorithm is a maximum flow.

Proof:

Min-Cut Algorithm

Ford-Fulkerson also gives a min-cut algorithm:

Theorem: Given a flow f of maximum value, we can compute an s-t cut of minimum capacity in $O(m)$ time.

Proof:

Max-Flow Min-Cut Theorem

Theorem: (Max-Flow Min-Cut Theorem)

In every flow network, the maximum value of an s-t flow is equal to the minimum capacity of an s-t cut.

Proof:

Integer Capacities

Theorem: (Integer-Valued Flows)

If all capacities in the flow network are integers, then there is a maximum flow f for which the flow $f(e)$ of every edge e is an integer.

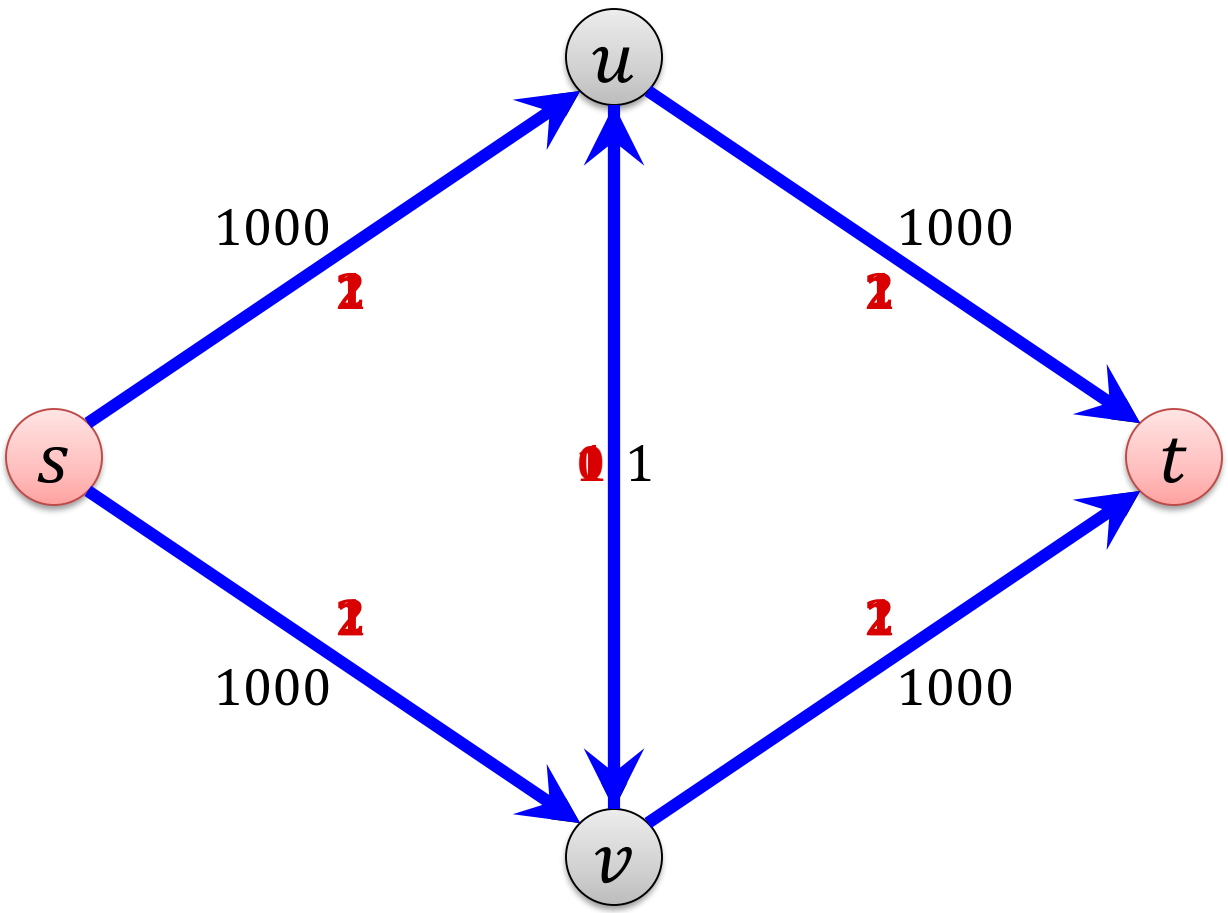
Proof:

Non-Integer Capacities

What if capacities are not integers?

- rational capacities:
  - can be turned into integers by multiplying them with large enough integer
  - algorithm still works correctly
- real (non-rational) capacities:
  - not clear whether the algorithm always terminates
- even for integer capacities, time can linearly depend on the value of the maximum flow

Slow Execution



- Number of iterations: 2000 (value of max. flow)

Improved Algorithm

Idea: Find the best augmenting path in each step

- best: path P with maximum $\text{bottleneck}(P, f)$

- Best path might be rather expensive to find
  → find almost best path

- **Scaling parameter Δ:**
  (initially, $\Delta = \text{"max } c_e \text{ rounded down to next power of 2"}$)

- As long as there is an augmenting path that improves the flow by at least Δ, augment using such a path

- If there is no such path: $\Delta := {}^{\Delta}/_{2}$

Scaling Parameter Analysis

Lemma: If all capacities are integers, number of different scaling parameters used is $\leq 1 + \lfloor \log_2 C \rfloor$.

- **Δ-scaling phase:** Time during which scaling parameter is Δ

Length of a Scaling Phase

Lemma: If f is the flow at the end of the Δ-scaling phase, the maximum flow in the network has value at most $|f| + m\Delta$.

Length of a Scaling Phase

Lemma: The number of augmentation in each scaling phase is at most $2m$.

Running Time: Scaling Max Flow Alg.

Theorem: The number of augmentations of the algorithm with scaling parameter and integer capacities is at most $O(m \log C)$. The algorithm can be implemented in time $O(m^2 \log C)$.

Strongly Polynomial Algorithm

- Time of regular Ford-Fulkerson algorithm with integer capacities:

$$O(mC)$$

- Time of algorithm with scaling parameter:

$$O(m^2 \log C)$$

- $O(\log C)$ is polynomial in the size of the input, but not in n
- Can we get an algorithm that runs in time polynomial in n?
- Always picking a shortest augmenting path leads to running time

$$O(m^2 n)$$

  – also works for arbitrary real-valued weights

Other Algorithms

- There are many other algorithms to solve the maximum flow problem, for example:

- **Preflow-push algorithm:**
  - Maintains a preflow (∀ nodes: inflow ≥ outflow)
  - Alg. guarantees: As soon as we have a flow, it is optimal
  - Detailed discussion in 2012/13 lecture
  - Running time of basic algorithm: $O(m \cdot n^2)$
  - Doing steps in the "right" order: $O(n^3)$

- **Current best known complexity: $\boldsymbol{O(m \cdot n)}$**
  - For graphs with $m \geq n^{1+\epsilon}$ [King,Rao,Tarjan 1992/1994]
    (for every constant $\epsilon > 0$)
  - For sparse graphs with $m \leq n^{16/15-\delta}$ [Orlin, 2013]

Chapter 6
Graph Algorithms

Algorithm Theory
WS 2018/19

Fabian Kuhn

Strongly Polynomial Algorithm

- Time of regular Ford-Fulkerson algorithm with integer capacities:

$$O(mC)$$

- Time of algorithm with scaling parameter:

$$O(m^2 \log C)$$

- $O(\log C)$ is polynomial in the size of the input, but not in n
- Can we get an algorithm that runs in time polynomial in n?
- Always picking a shortest augmenting path leads to running time

$$O(m^2 n)$$

  – also works for arbitrary real-valued weights

Other Algorithms

- There are many other algorithms to solve the maximum flow problem, for example:

- **Preflow-push algorithm:**
  - Maintains a preflow (∀ nodes: inflow ≥ outflow)
  - Alg. guarantees: As soon as we have a flow, it is optimal
  - Detailed discussion in 2012/13 lecture
  - Running time of basic algorithm: $O(m \cdot n^2)$
  - Doing steps in the "right" order: $O(n^3)$

- **Current best known complexity: $\boldsymbol{O(m \cdot n)}$**
  - For graphs with $m \geq n^{1+\epsilon}$ [King,Rao,Tarjan 1992/1994]
    (for every constant $\epsilon > 0$)
  - For sparse graphs with $m \leq n^{16/15-\delta}$ [Orlin, 2013]

Maximum Flow Applications

- Maximum flow has many applications

- Reducing a problem to a max flow problem can even be seen as an important algorithmic technique

- Examples:
  - related network flow problems
  - computation of small cuts
  - computation of matchings
  - computing disjoint paths
  - scheduling problems
  - assignment problems with some side constraints
  - …

Undirected Edges and Vertex Capacities

Undirected Edges:

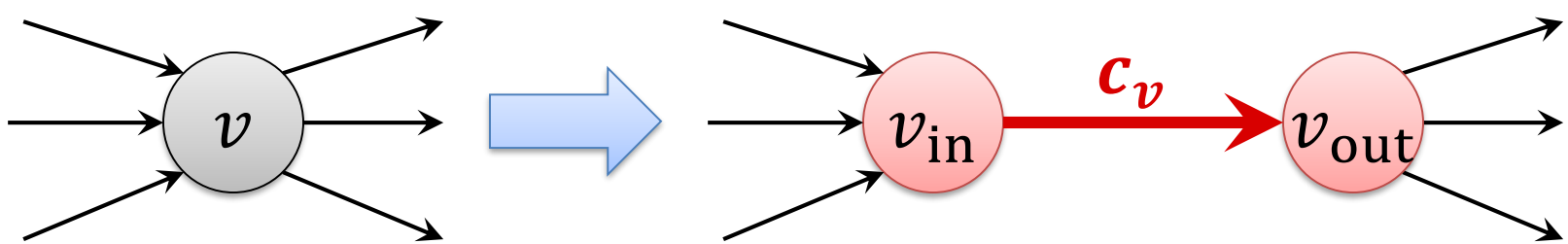
- Undirected edge $\{u, v\}$: add edges (u, v) and (v, u) to network

Vertex Capacities:

- Not only edges, but also (or only) nodes have capacities
- Capacity c_v of node $v \notin \{s, t\}$:

$$f^{\text{in}}(v) = f^{\text{out}}(v) \leq c_v$$

- Replace node v by edge $e_v = \{v_{\text{in}}, v_{\text{out}}\}$:



Minimum s-t Cut

Given: undirected graph $G = (V, E)$, nodes $s, t \in V$

s-t cut: Partition (A, B) of V such that $s \in A$, $t \in B$

Size of cut (A, B): number of edges crossing the cut

Objective: find s-t cut of minimum size

Edge Connectivity

Definition: A graph $G = (V, E)$ is k-edge connected for an integer $k \geq 1$ if the graph $G_X = (V, E \setminus X)$ is connected for every edge set

$$X \subseteq E, |X| \leq k - 1.$$

Goal: Compute edge connectivity $\lambda(G)$ of G
(and edge set X of size $\lambda(G)$ that divides G into ≥ 2 parts)

- minimum set X is a minimum s-t cut for some $s, t \in V$
  - Actually for all s, t in different components of $G_X = (V, E \setminus X)$
- Possible algorithm: fix s and find min s-t cut for all $t \neq s$

Minimum s-t Vertex-Cut

Given: undirected graph $G = (V, E)$, nodes $s, t \in V$

***s*-*t* vertex cut:** Set $X \subset V$ such that $s, t \notin X$ and s and t are in different components of the sub-graph $G[V \setminus X]$ induced by $V \setminus X$

Size of vertex cut: $|X|$

Objective: find s-t vertex-cut of minimum size

- Replace undirected edge $\{u, v\}$ by (u, v) and (v, u)
- Compute max s-t flow for edge capacities ∞ and node capacities

$$c_v = 1 \text{ for } v \neq s, t$$

- Replace each node v by v_{in} and v_{out}:
- Min edge cut corresponds to min vertex cut in G

Vertex Connectivity

Definition: A graph $G = (V, E)$ is k-vertex connected for an integer $k \geq 1$ if the sub-graph $G[V \setminus X]$ induced by $V \setminus X$ is connected for every edge set

$$X \subseteq V, |X| \leq k - 1.$$

Goal: Compute vertex connectivity $\kappa(G)$ of G
(and node set X of size $\kappa(G)$ that divides G into ≥ 2 parts)

- Compute minimum s-t vertex cut for all s and all $t \neq s$

Edge-Disjoint Paths

Given: Graph $G = (V, E)$ with nodes $s, t \in V$

Goal: Find as many edge-disjoint s-t paths as possible

Solution:

- Find max s-t flow in G with edge capacities $c_e = 1$ for all $e \in E$

Flow f induces $|f|$ edge-disjoint paths:

- Integral capacities → can compute integral max flow f
- Get $|f|$ edge-disjoint paths by greedily picking them
- Correctness follows from flow conservation $f^{\text{in}}(v) = f^{\text{out}}(v)$

Vertex-Disjoint Paths

Given: Graph $G = (V, E)$ with nodes $s, t \in V$

Goal: Find as many internally vertex-disjoint s-t paths as possible

Solution:

- Find max s-t flow in G with node capacities $c_v = 1$ for all $v \in V$

Flow f induces $|f|$ vertex-disjoint paths:

- Integral capacities → can compute integral max flow f
- Get $|f|$ vertex-disjoint paths by greedily picking them
- Correctness follows from flow conservation $f^{\text{in}}(v) = f^{\text{out}}(v)$

Menger's Theorem

Theorem: (edge version)
For every graph $G = (V, E)$ with nodes $s, t \in V$, the size of the minimum s-t (edge) cut equals the maximum number of pairwise edge-disjoint paths from s to t.

Theorem: (node version)
For every graph $G = (V, E)$ with nodes $s, t \in V$, the size of the minimum s-t vertex cut equals the maximum number of pairwise internally vertex-disjoint paths from s to t

- Both versions can be seen as a special case of the max flow min cut theorem

Baseball Elimination

Team i	Wins w_i	Losses ℓ_i	To Play r_i	Against = r_{ij}				
				NY	Balt.	T. Bay	Tor.	Bost.
New York	81	69	12	-	2	5	2	3
Baltimore	79	77	6	2	-	2	1	1
Tampa Bay	79	74	9	5	2	-	1	1
Toronto	76	80	6	2	1	1	-	2
Boston	71	84	7	3	1	1	2	-

- Only wins/losses possible (no ties), winner: team with most wins
- Which teams can still win (as least as many wins as top team)?
- Boston is eliminated (cannot win):
  - Boston can get at most 78 wins, New York already has 81 wins
- If for some i, j: $w_i + r_i < w_j$ → team i is eliminated
- Sufficient condition, but not a necessary one!

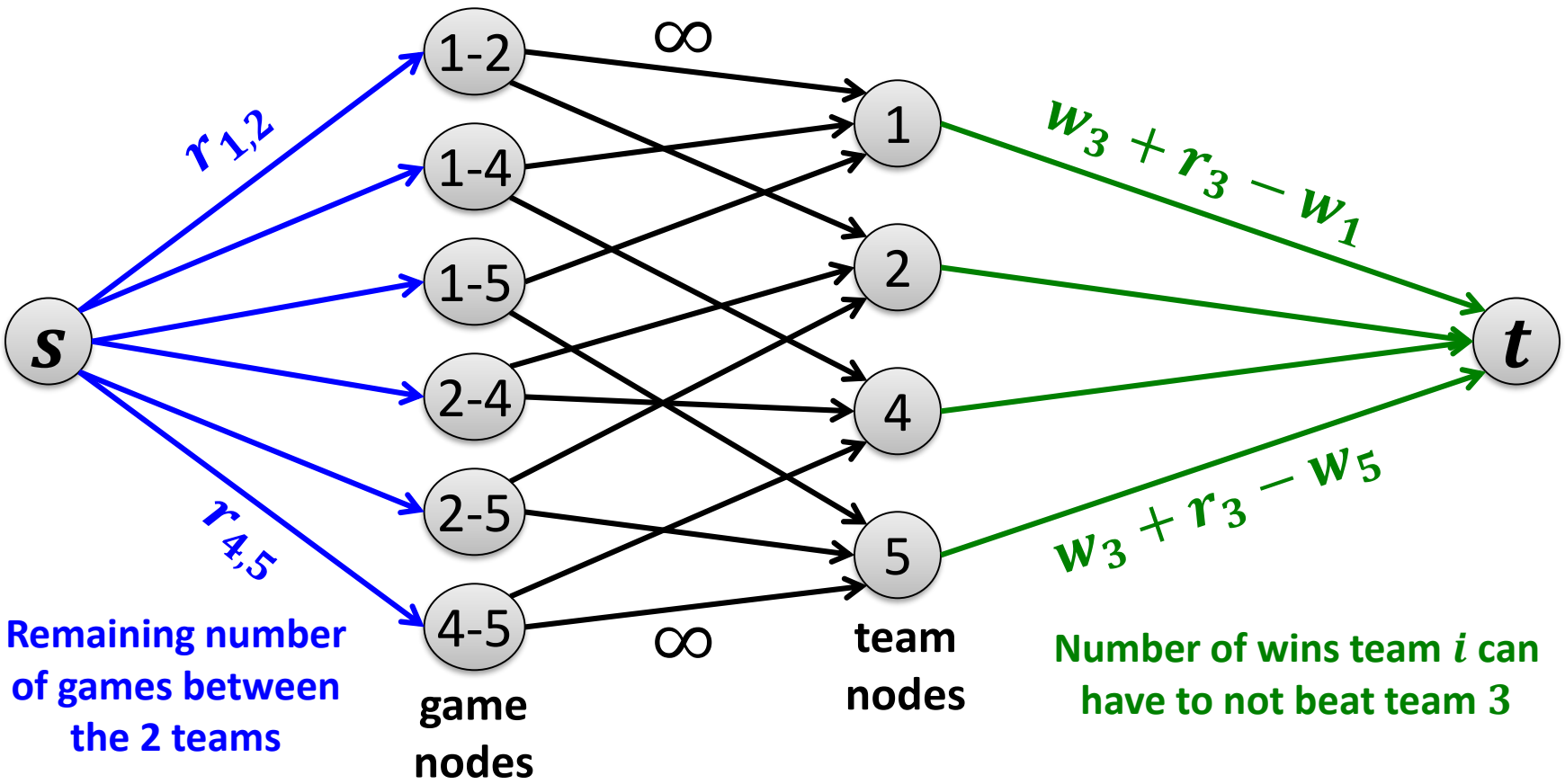
Baseball Elimination

Team i	Wins w_i	Losses ℓ_i	To Play r_i	Against = r_{ij}				
				NY	Balt.	T. Bay	Tor.	Bost.
New York	81	69	12	-	2	5	2	3
Baltimore	79	77	6	2	-	2	1	1
Tampa Bay	79	74	9	5	2	-	1	1
Toronto	76	80	6	2	1	1	-	2
Boston	71	84	7	3	1	1	2	-

- Can Toronto still finish first?
- Toronto can get $82 > 81$ wins, but:
  NY and Tampa have to play 5 more times against each other
  → if NY wins two, it gets 83 wins, otherwise, Tampa has 83 wins
- Hence: Toronto cannot finish first
- How about the others? How can we solve this in general?

Max Flow Formulation

- Can team 3 finish with most wins?



- Team 3 can finish first iff all source-game edges are saturated

Reason for Elimination

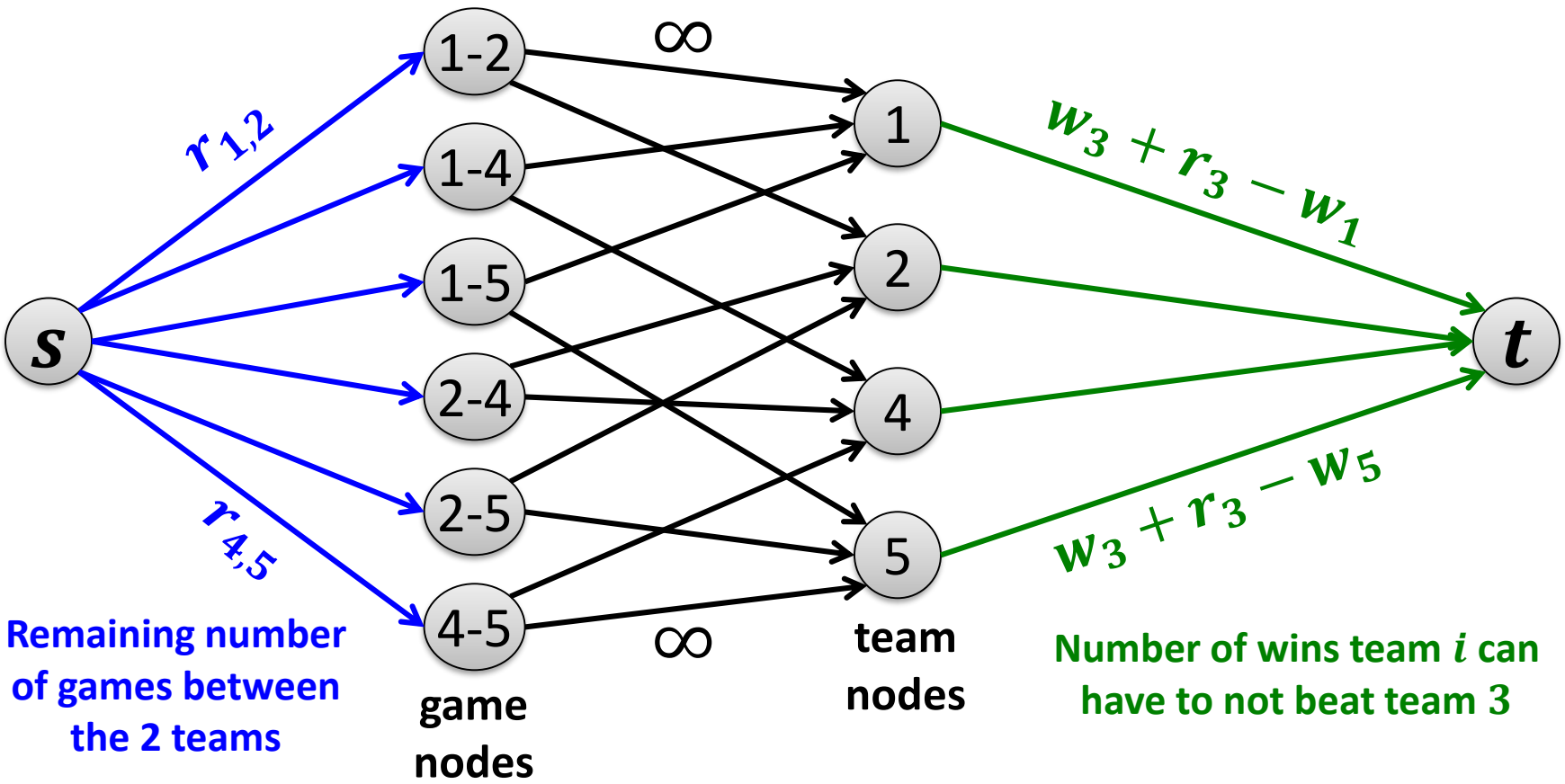
AL East: Aug 30, 1996

Team i	Wins w_i	Losses ℓ_i	To Play r_i	Against = r_{ij}				
				NY	Balt.	Bost.	Tor.	Detr.
New York	75	59	28	-	3	8	7	3
Baltimore	71	63	28	3	-	2	7	4
Boston	69	66	27	8	2	-	0	0
Toronto	63	72	27	7	7	0	-	0
Detroit	49	86	27	3	4	0	0	-

- Detroit could finish with $49 + 27 = 76$ wins
- Consider $R = \{\mathrm{NY}, \mathrm{Bal}, \mathrm{Bos}, \mathrm{Tor}\}$
  - Have together already won $w(R) = 278$ games
  - Must together win at least $r(R) = 27$ more games
- On average, teams in R win $\frac{278+27}{4} = 76.25$ games

Reason for Elimination

- Can team 3 finish with most wins?



- Team 3 cannot finish first $\Longleftrightarrow$ min cut of size < "all blue edges"

Reason for Elimination

Certificate of elimination:

$$R \subseteq X, \qquad w(R) := \underbrace{\sum_{i \in R} w_i}_{\text{\#wins of nodes in } R}, \qquad r(R) := \underbrace{\sum_{i,j \in R} r_{i,j}}_{\text{\#remaining games among nodes in } R}$$

Team $x \in X$ is eliminated by R if

$$\frac{w(R) + r(R)}{|R|} > w_x + r_x.$$

Reason for Elimination

Theorem: Team x is eliminated if and only if there exists a subset $R \subseteq X$ of the teams X such that x is eliminated by R.

Proof Idea:

- Minimum cut gives a certificate…
- If x is eliminated, max flow solution does not saturate all outgoing edges of the source.
- Team nodes of unsaturated source-game edges are saturated
- Source side of min cut contains all teams of saturated team-dest. edges of unsaturated source-game edges
- Set of team nodes in source-side of min cut give a certificate R

Circulations with Demands

Given: Directed network with positive edge capacities

Sources & Sinks: Instead of one source and one destination, several sources that generate flow and several sinks that absorb flow.

Supply & Demand: sources have supply values, sinks demand values

Goal: Compute a flow such that source supplies and sink demands are exactly satisfied

- The circulation problem is a feasibility rather than a maximization problem

Circulations with Demands: Formally

Given: Directed network $G = (V, E)$ with

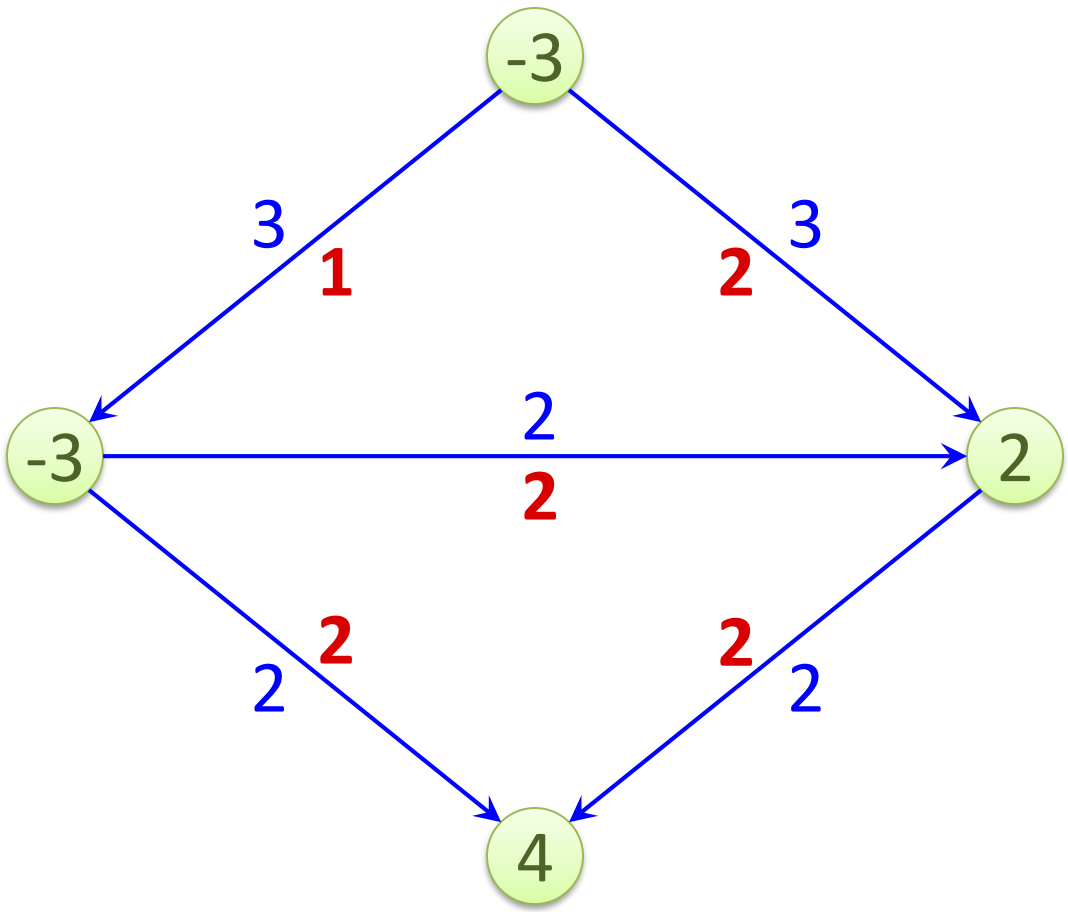
- Edge capacities $c_e > 0$ for all $e \in E$
- Node demands $d_v \in \mathbb{R}$ for all $v \in V$
  - $d_v > 0$: node needs flow and therefore is a sink
  - $d_v < 0$: node has a supply of $-d_v$ and is therefore a source
  - $d_v = 0$: node is neither a source nor a sink

Flow: Function $f: E \to \mathbb{R}_{\geq 0}$ satisfying

- *Capacity Conditions*: $\forall e \in E$: $\quad 0 \leq f(e) \leq c_e$
- *Demand Conditions*: $\forall v \in V$: $\quad f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$

Objective: Does a flow f satisfying all conditions exist?
If yes, find such a flow f.

Example



Condition on Demands

Claim: If there exists a feasible circulation with demands d_v for $v \in V$, then

$$\sum_{v \in V} d_v = 0.$$

Proof:

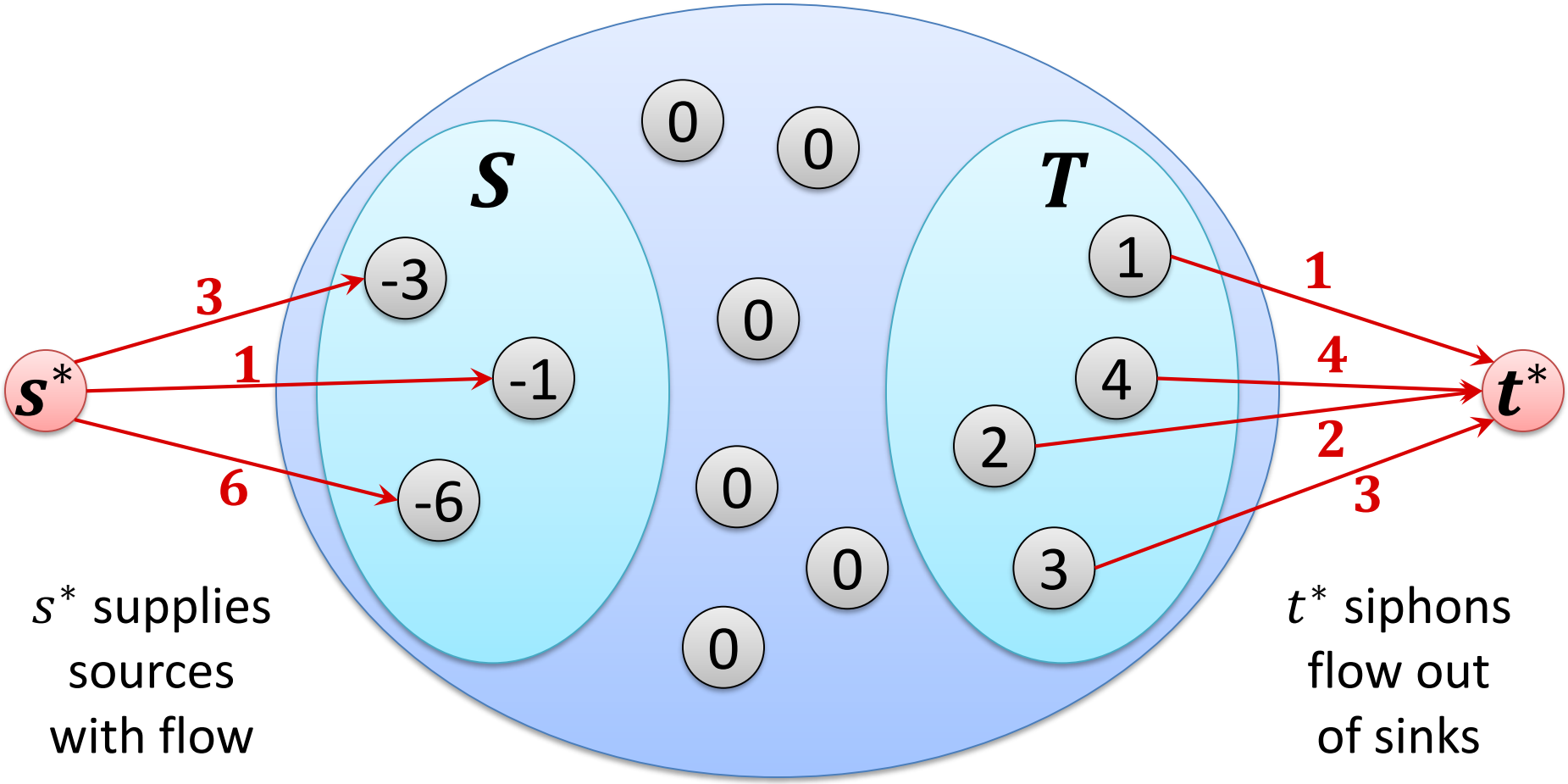
- $\sum_v d_v = \sum_v \left(f^{\text{in}}(v) - f^{\text{out}}(v) \right)$
- $f(e)$ of each edge e appears twice in the above sum with different signs → overall sum is 0

Total supply = total demand:

$$\text{Define } \boldsymbol{D} := \sum_{v: d_v > 0} \boldsymbol{d_v} = \sum_{v: d_v < 0} -\boldsymbol{d_v}$$

Reduction to Maximum Flow

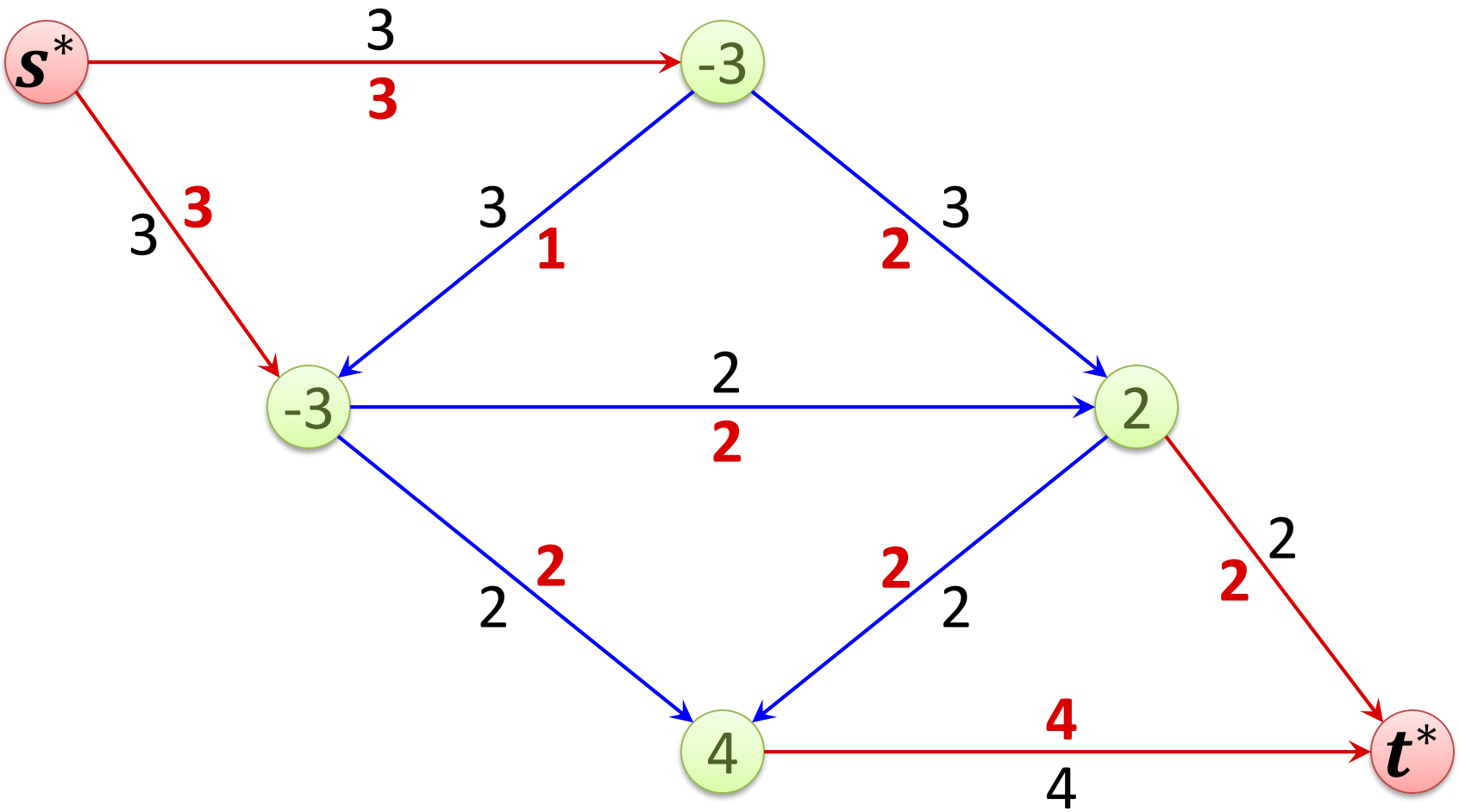
- Add "super-source" s^* and "super-sink" t^* to network



s^* supplies sources with flow

t^* siphons flow out of sinks

Example



Formally…

Reduction: Get graph G' from graph as follows

- Node set of G' is $V \cup \{s^*, t^*\}$
- Edge set is E and edges
  - (s^*, v) for all v with $d_v < 0$, capacity of edge is $-d_v$
  - (v, t^*) for all v with $d_v > 0$, capacity of edge is d_v

Observations:

- Capacity of min s^*-t^* cut is at most D (e.g., the cut $(s^*, V \cup \{t^*\})$
- A feasible circulation on G can be turned into a feasible flow of value D of G' by saturating all (s^*, v) and (v, t^*) edges.
- Any flow of G' of value D induces a feasible circulation on G
  - (s^*, v) and (v, t^*) edges are saturated
  - By removing these edges, we get exactly the demand constraints

Circulation with Demands

Theorem: There is a feasible circulation with demands d_v, $v \in V$ on graph G if and only if there is a flow of value D on G'.

- If all capacities and demands are integers, there is an integer circulation

The max flow min cut theorem also implies the following:

Theorem: The graph G has a feasible circulation with demands d_v, $v \in V$ if and only if for all cuts (A, B),

$$\sum_{v \in B} d_v \leq c(A, B)\,.$$

Circulation: Demands and Lower Bounds

Given: Directed network $G = (V, E)$ with

- Edge capacities $c_e > 0$ and **lower bounds $\boldsymbol{0 \leq \ell_e \leq c_e}$ for $\boldsymbol{e \in E}$**
- Node demands $d_v \in \mathbb{R}$ for all $v \in V$
  - $d_v > 0$: node needs flow and therefore is a sink
  - $d_v < 0$: node has a supply of $-d_v$ and is therefore a source
  - $d_v = 0$: node is neither a source nor a sink

Flow: Function $f: E \to \mathbb{R}_{\geq 0}$ satisfying

- *Capacity Conditions*: $\forall e \in E$: $\boldsymbol{\ell_e \leq f(e) \leq c_e}$
- *Demand Conditions*: $\forall v \in V$: $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$

Objective: Does a flow f satisfying all conditions exist?
If yes, find such a flow f.

Chapter 6
Graph Algorithms

Algorithm Theory
WS 2018/19

Fabian Kuhn

Circulations with Demands

Given: Directed network with positive edge capacities

Sources & Sinks: Instead of one source and one destination, several sources that generate flow and several sinks that absorb flow.

Supply & Demand: sources have supply values, sinks demand values

Goal: Compute a flow such that source supplies and sink demands are exactly satisfied

- The circulation problem is a feasibility rather than a maximization problem

Circulations with Demands: Formally

Given: Directed network $G = (V, E)$ with

- Edge capacities $c_e > 0$ for all $e \in E$
- Node demands $d_v \in \mathbb{R}$ for all $v \in V$
  - $d_v > 0$: node needs flow and therefore is a sink
  - $d_v < 0$: node has a supply of $-d_v$ and is therefore a source
  - $d_v = 0$: node is neither a source nor a sink

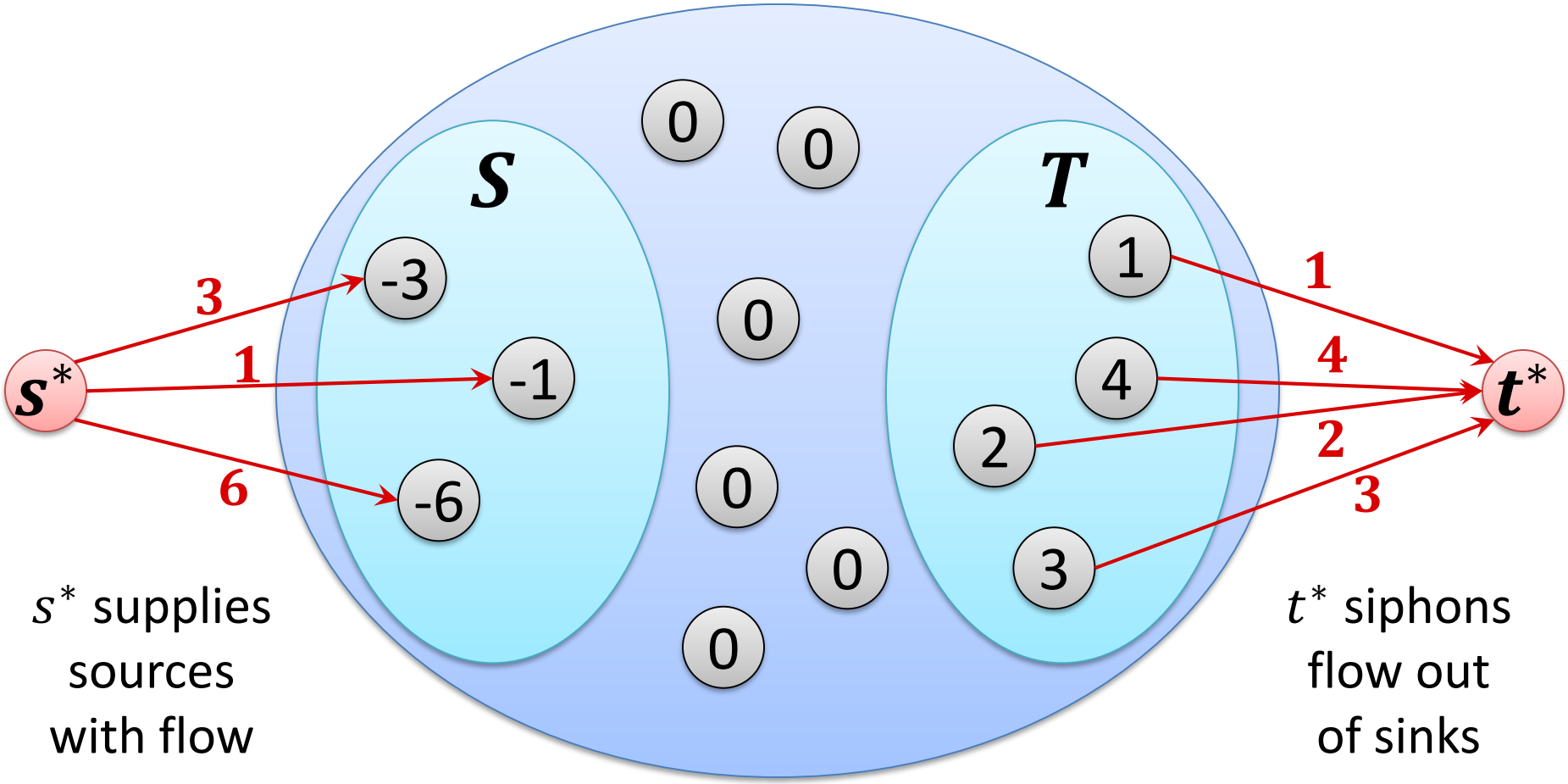
Flow: Function $f: E \to \mathbb{R}_{\geq 0}$ satisfying

- *Capacity Conditions*: $\forall e \in E$: $\quad 0 \leq f(e) \leq c_e$
- *Demand Conditions*: $\forall v \in V$: $\quad f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$

Objective: Does a flow f satisfying all conditions exist?
If yes, find such a flow f.

Reduction to Maximum Flow

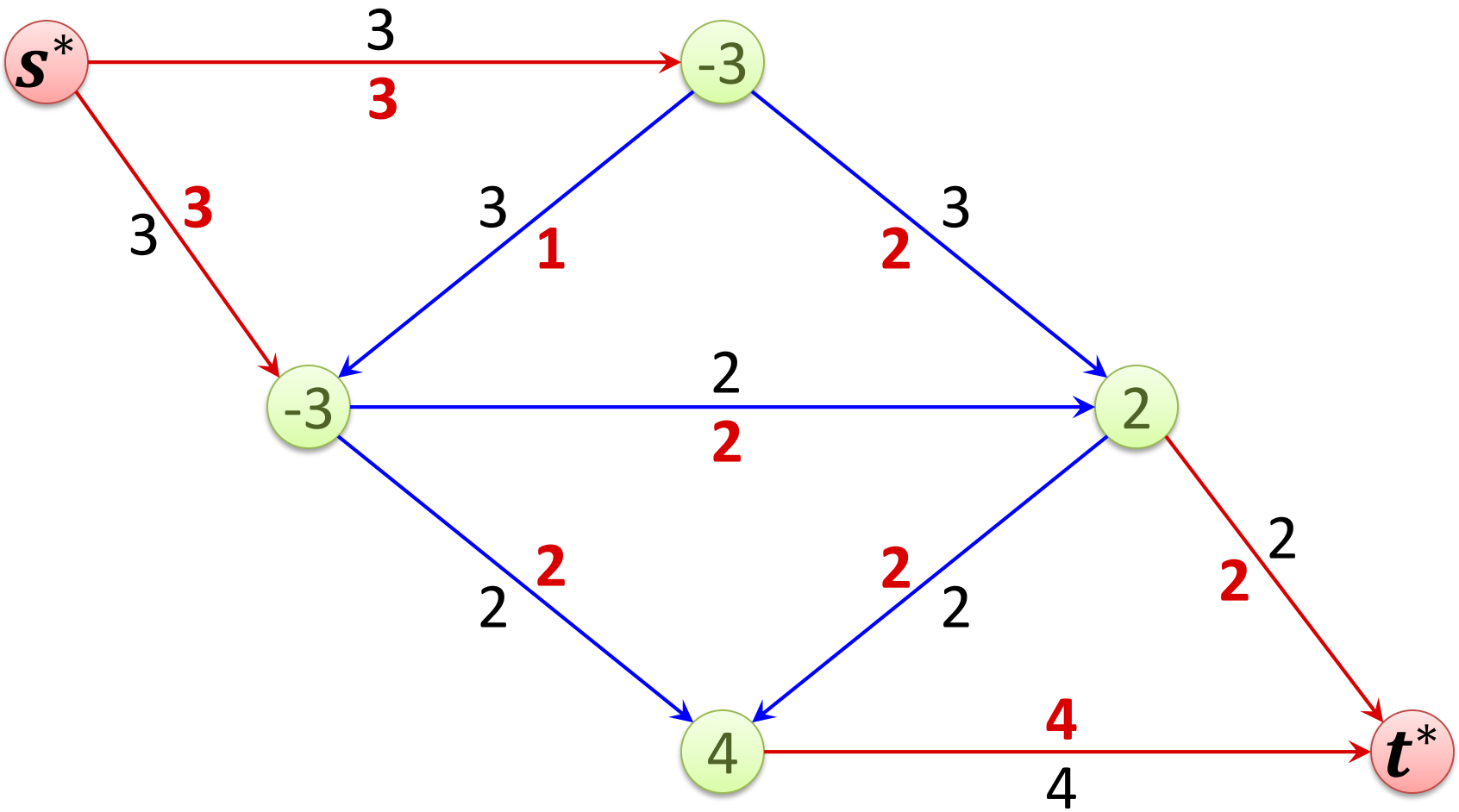
- Add "super-source" s^* and "super-sink" t^* to network



s^* supplies sources with flow

t^* siphons flow out of sinks

Example



Circulation with Demands

Theorem: There is a feasible circulation with demands d_v, $v \in V$ on graph G if and only if there is a flow of value D on G'.

- If all capacities and demands are integers, there is an integer circulation

The max flow min cut theorem also implies the following:

Theorem: The graph G has a feasible circulation with demands d_v, $v \in V$ if and only if for all cuts (A, B),

$$\sum_{v \in B} d_v \leq c(A, B)\,.$$

Circulation: Demands and Lower Bounds

Given: Directed network $G = (V, E)$ with

- Edge capacities $c_e > 0$ and **lower bounds $\boldsymbol{0 \leq \ell_e \leq c_e}$ for $\boldsymbol{e \in E}$**
- Node demands $d_v \in \mathbb{R}$ for all $v \in V$
  - $d_v > 0$: node needs flow and therefore is a sink
  - $d_v < 0$: node has a supply of $-d_v$ and is therefore a source
  - $d_v = 0$: node is neither a source nor a sink

Flow: Function $f: E \to \mathbb{R}_{\geq 0}$ satisfying

- *Capacity Conditions*: $\forall e \in E$: $\boldsymbol{\ell_e \leq f(e) \leq c_e}$
- *Demand Conditions*: $\forall v \in V$: $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$

Objective: Does a flow f satisfying all conditions exist?
If yes, find such a flow f.

Solution Idea

- Define initial circulation $f_0(e) = \ell_e$
  Satisfies capacity constraints: $\forall e \in E: \ell_e \leq f_0(e) \leq c_e$

- Define

$$L_v := f_0^{\text{in}}(v) - f_0^{\text{out}}(v) = \sum_{e \text{ into } v} \ell_e - \sum_{e \text{ out of } v} \ell_e$$

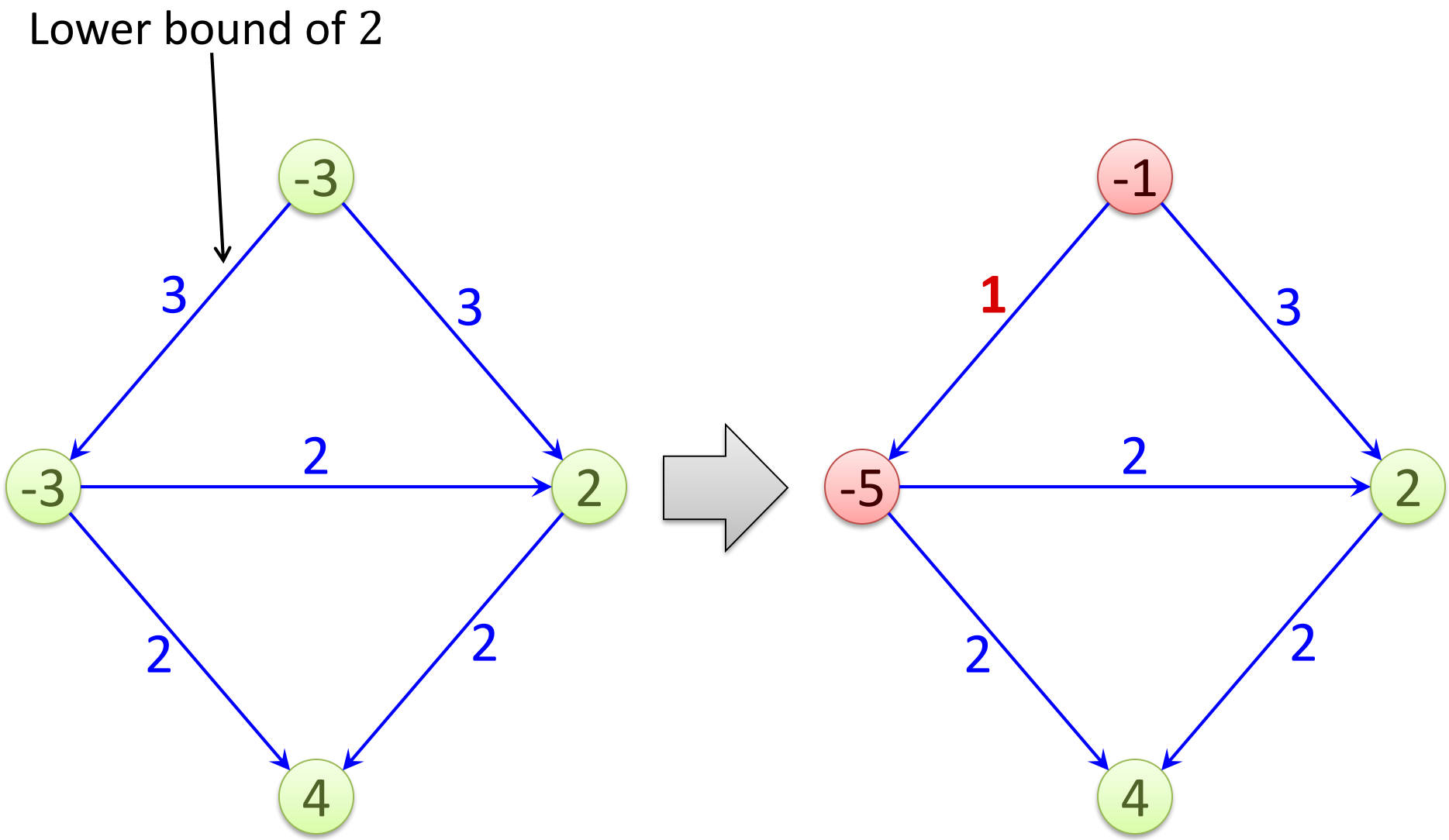
- If $L_v = d_v$, demand condition is satisfied at v by f_0, otherwise, we need to superimpose another circulation f_1 such that

$$d'_v := f_1^{\text{in}}(v) - f_1^{\text{out}}(v) = d_v - L_v$$

- Remaining capacity of edge e: $c'_e := c_e - \ell_e$

- We get a circulation problem with new demands d'_v, new capacities c'_e, and no lower bounds

Eliminating a Lower Bound: Example

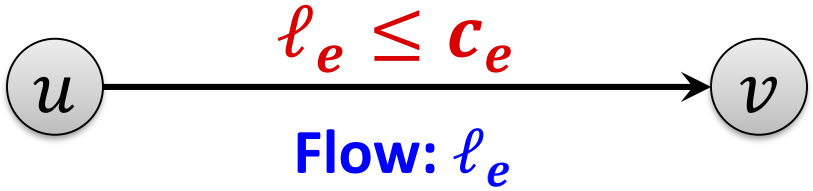


Reduce to Problem Without Lower Bounds

Graph $G = (V, E)$:

- Capacity: For each edge $e \in E$: $\ell_e \leq f(e) \leq c_e$
- Demand: For each node $v \in V$: $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$

Model lower bounds with supplies & demands:



Create Network G' (without lower bounds):

- For each edge $e \in E$: $c'_e = c_e - \ell_e$
- For each node $v \in V$: $d'_v = d_v - L_v$

Circulation: Demands and Lower Bounds

Theorem: There is a feasible circulation in G (with lower bounds) if and only if there is feasible circulation in G' (without lower bounds).

- Given circulation f' in G', $f(e) = f'(e) + \ell_e$ is circulation in G
  - The capacity constraints are satisfied because $f'(e) \leq c_e - \ell_e$
  - Demand conditions:

$$f^{\text{in}}(v) - f^{\text{out}}(v) = \sum_{e \text{ into } v} \left(\ell_e + f'(e)\right) - \sum_{e \text{ out of } v} \left(\ell_e + f'(e)\right)$$
$$= L_v + (d_v - L_v) = d_v$$

- Given circulation f in G, $f'(e) = f(e) - \ell_e$ is circulation in G'
  - The capacity constraints are satisfied because $\ell_e \leq f(e) \leq c_e$
  - Demand conditions:

$$f'^{\text{in}}(v) - f'^{\text{out}}(v) = \sum_{e \text{ into } v} (f\ (e) - \ell_e) - \sum_{e \text{ out of } v} (f\ (e) - \ell_e)$$
$$= d_v - L_v$$

Integrality

Theorem: Consider a circulation problem with integral capacities, flow lower bounds, and node demands. If the problem is feasible, then it also has an integral solution.

Proof:

- Graph G' has only integral capacities and demands
- Thus, the flow network used in the reduction to solve circulation with demands and no lower bounds has only integral capacities
- The theorem now follows because a max flow problem with integral capacities also has an optimal integral solution

- It also follows that with the max flow algorithms we studied, we get an integral feasible circulation solution.

Matrix Rounding

- **Given:** $p \times q$ matrix $D = \{d_{i,j}\}$ of real numbers
- **row *i* sum:** $a_i = \sum_j d_{i,j}$, **column *j* sum:** $b_j = \sum_i d_{i,j}$
- **Goal:** Round each $d_{i,j}$, as well as a_i and b_j up or down to the next integer so that the sum of rounded elements in each row (column) equals the rounded row (column) sum
- **Original application:** publishing census data

Example:

3.14	6.80	7.30	17.24
9.60	2.40	0.70	12.70
3.60	1.20	6.50	11.30
16.34	10.40	14.50	

original data

3	7	7	17
10	2	1	13
3	1	7	11
16	10	15	

possible rounding

Matrix Rounding

Theorem: For any matrix, there exists a feasible rounding.

Remark: Just rounding to the nearest integer doesn't work

0.35	0.35	0.35	1.05
0.55	0.55	0.55	1.65
0.90	0.90	0.90	

original data

0	0	0	0
1	1	1	3
1	1	1	

rounding to nearest integer

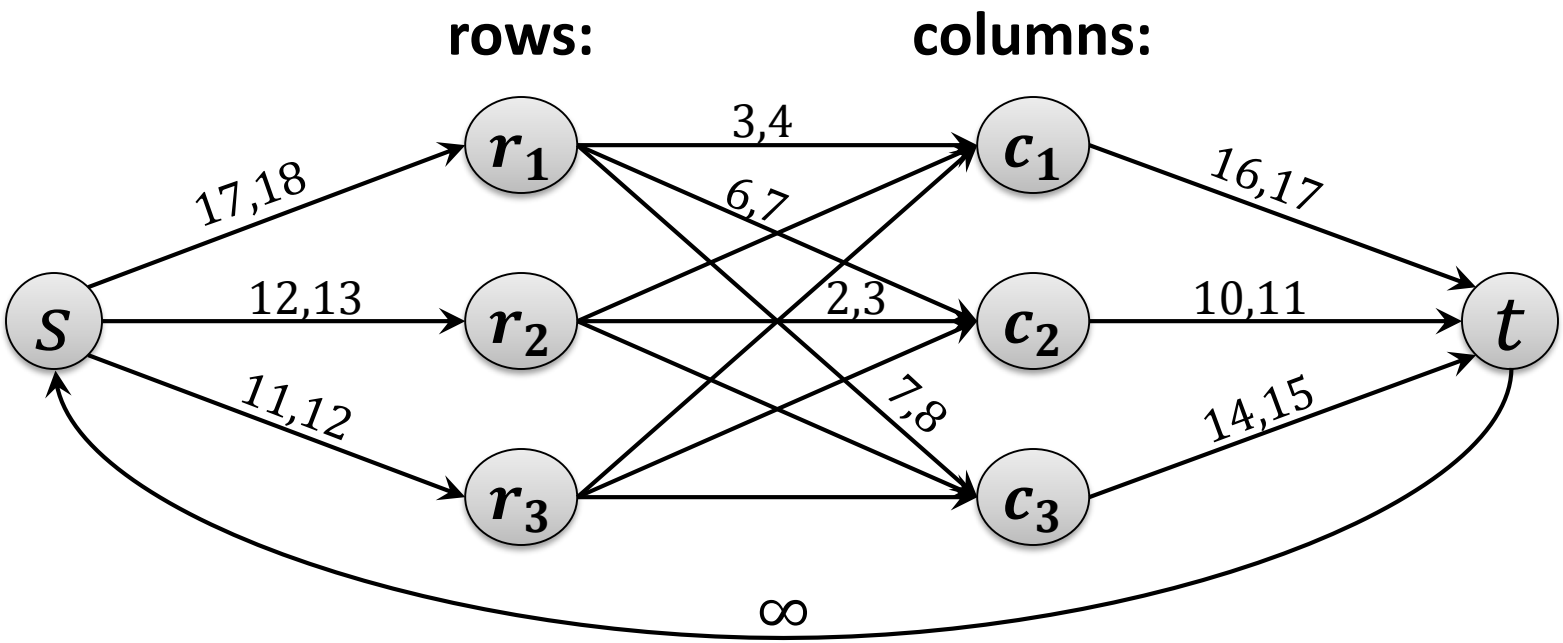
0	0	1	1
1	1	0	2
1	1	1	

feasible rounding

Reduction to Circulation

3.14	6.80	7.30	17.24
9.60	2.40	0.70	12.70
3.60	1.20	6.50	11.30
16.34	10.40	14.50	

Matrix elements and row/column sums give a feasible circulation that satisfies all lower bound, capacity, and demand constraints



all demands $d_v = 0$

Matrix Rounding

Theorem: For any matrix, there exists a feasible rounding.

Proof:

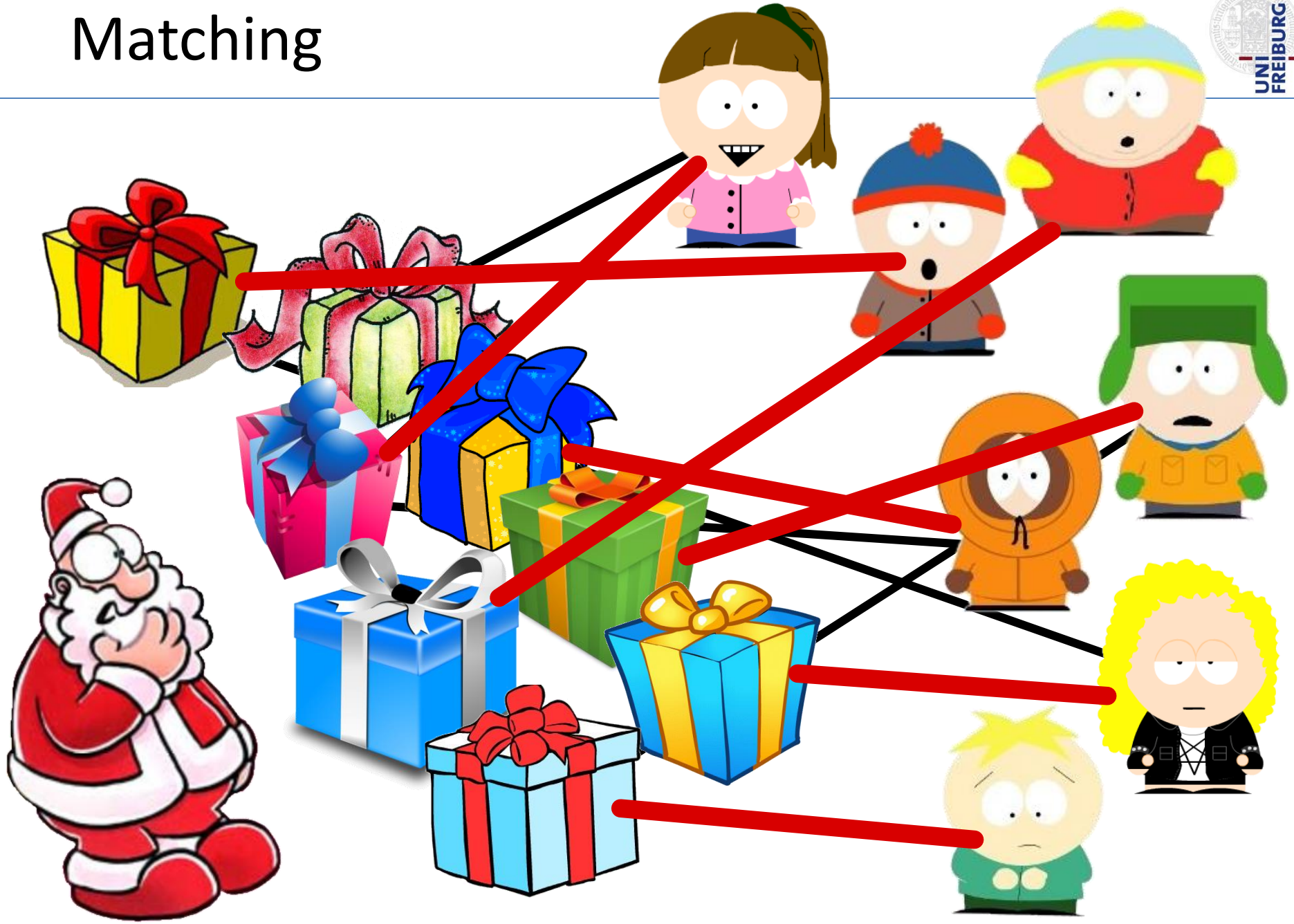
- The matrix entries $d_{i,j}$ and the row and column sums a_i and b_j give a feasible circulation for the constructed network

- Every feasible circulation gives matrix entries with corresponding row and column sums (follows from demand constraints)

- Because all demands, capacities, and flow lower bounds are integral, there is an integral solution to the circulation problem

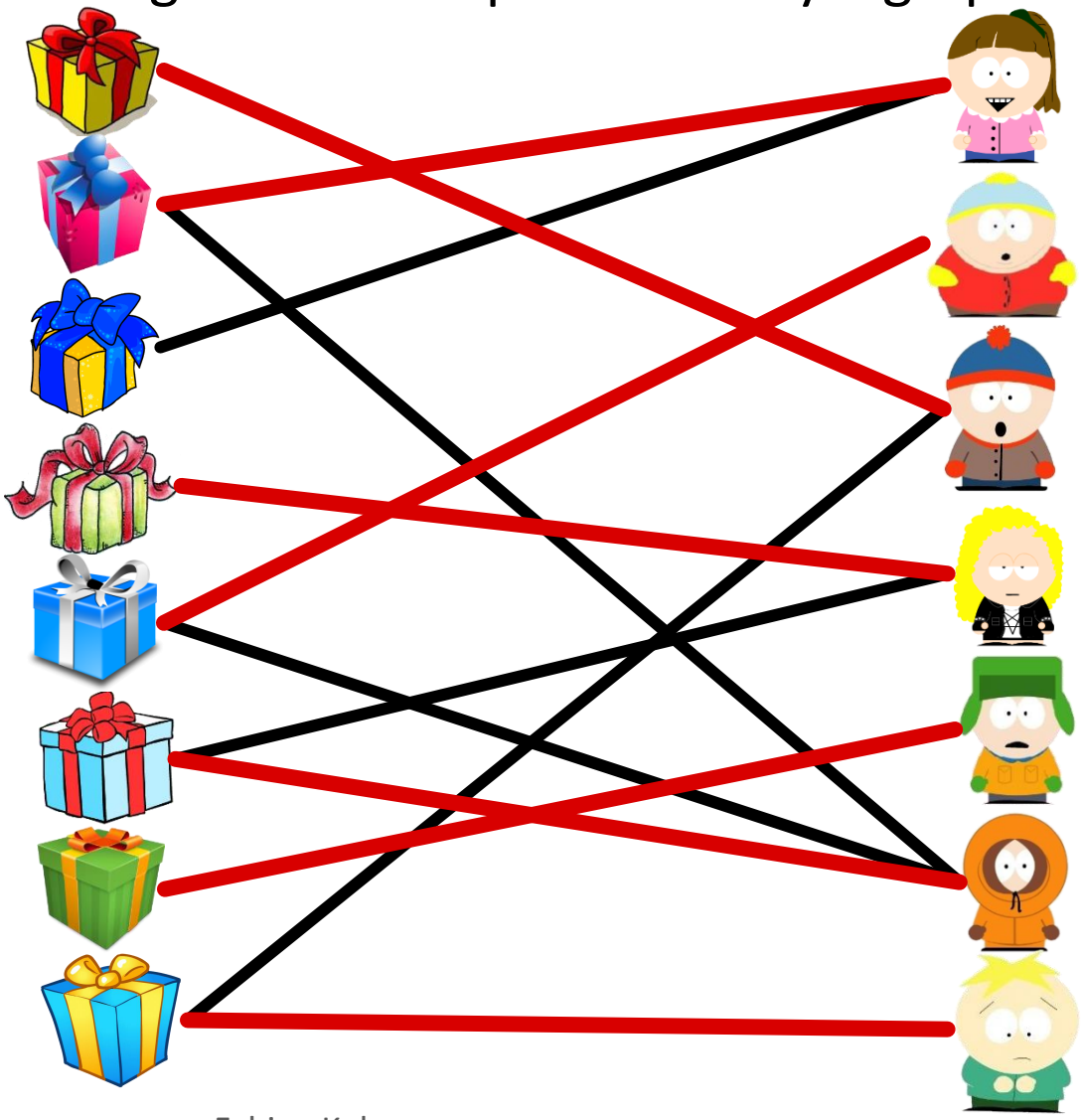
  → **gives a feasible rounding!**

Matching



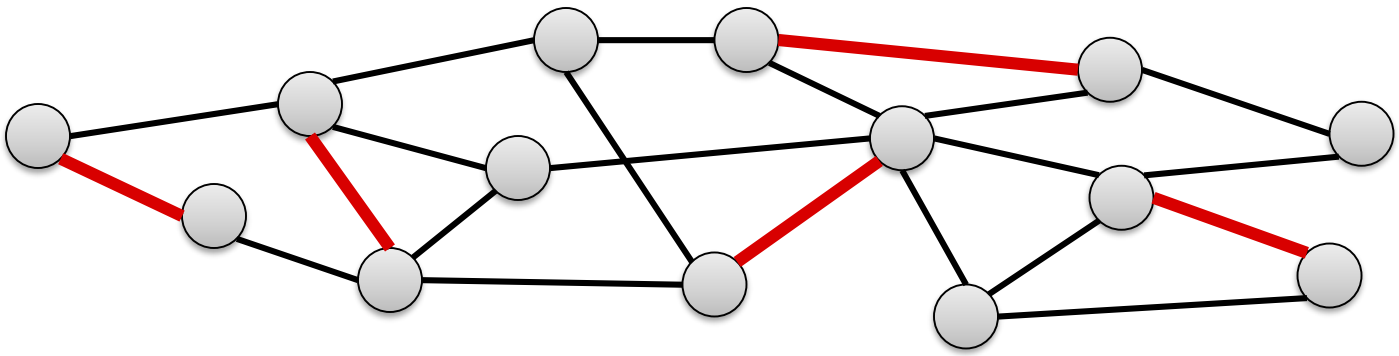
Gifts-Children Graph

- Which child likes which gift can be represented by a graph



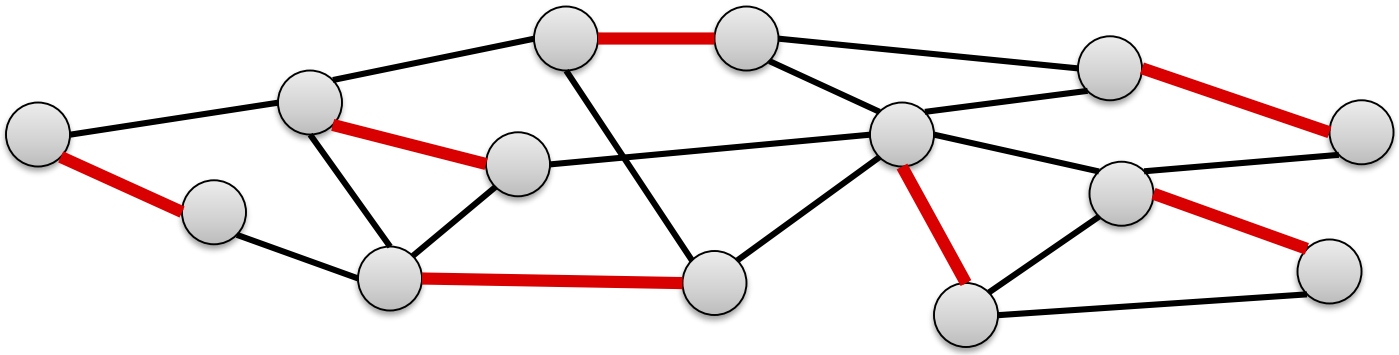
Matching

Matching: Set of pairwise non-incident edges



Maximal Matching: A matching s.t. no more edges can be added

Maximum Matching: A matching of maximum possible size



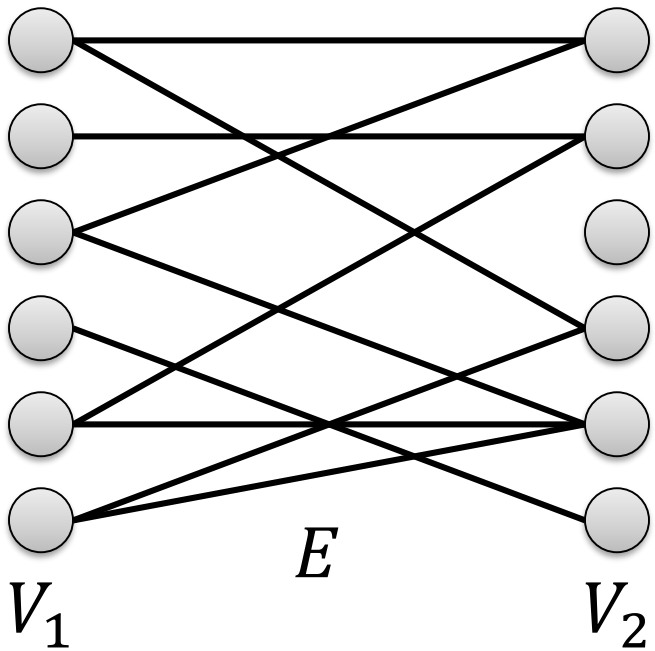
Perfect Matching: Matching of size $n/2$ (every node is matched)

Bipartite Graph

Definition: A graph $G = (V, E)$ is called bipartite iff its node set can be partitioned into two parts $V = V_1 \dot\cup V_2$ such that for each edge $\{\mathrm{u}, \mathrm{v}\} \in E$,

$$|\{u, v\} \cap V_1| = 1.$$

- Thus, edges are only between the two parts



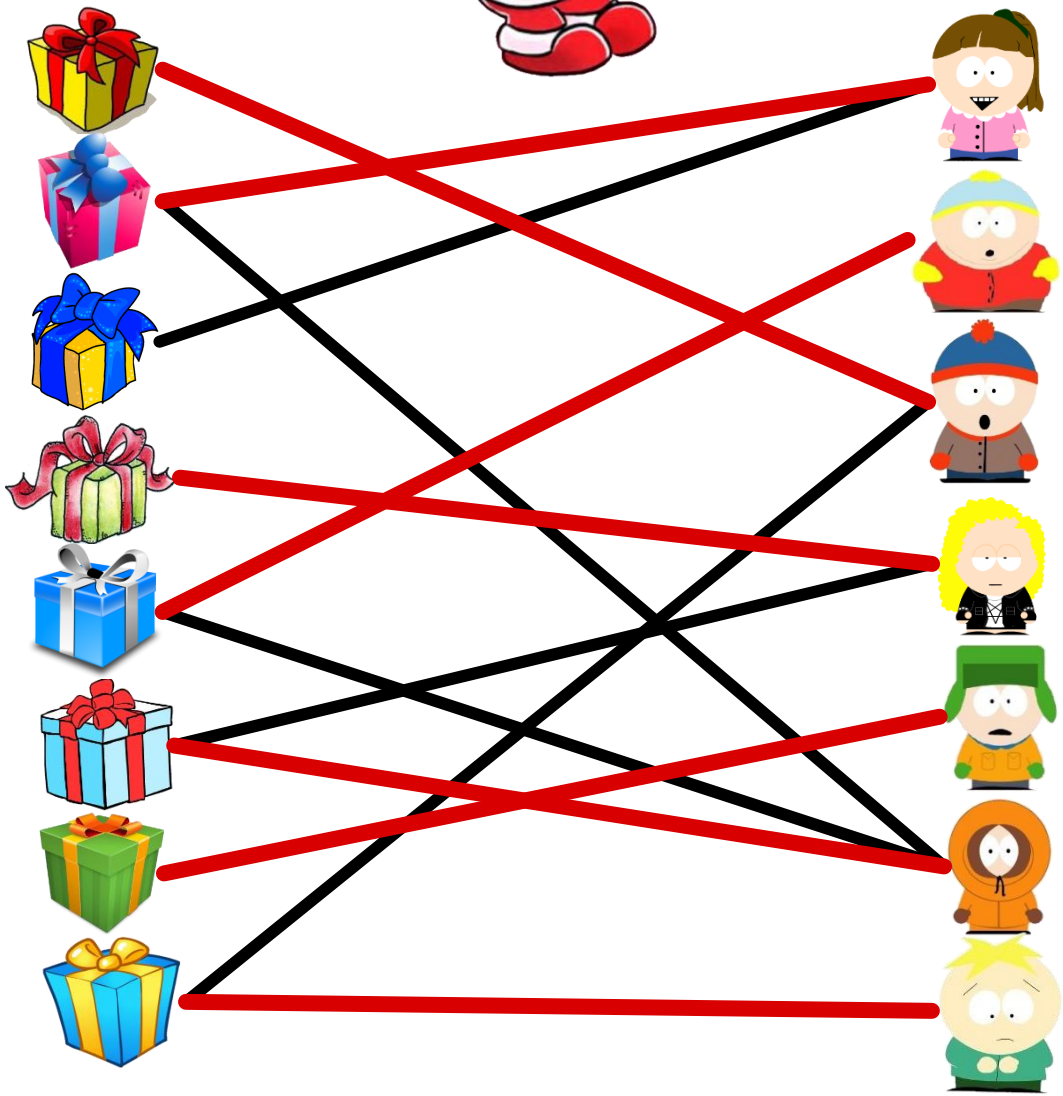
Santa's Problem

Maximum Matching in Bipartite Graphs:

Every child can get a gift iff there is a matching of size #children

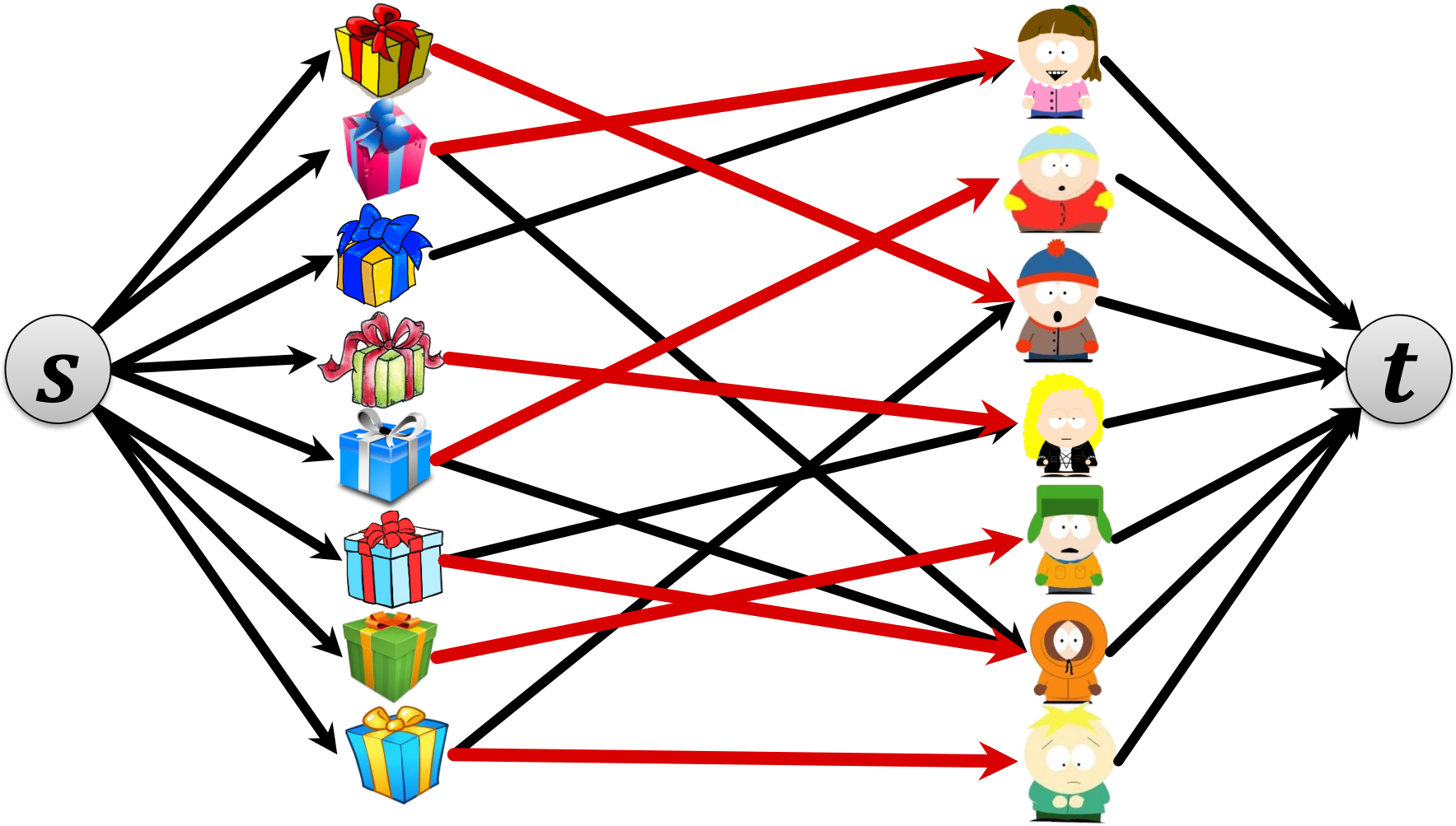
Clearly, every matching is at most as big

If #children = #gifts, there is a solution iff there is a perfect matching



Reducing to Maximum Flow

- Like edge-disjoint paths…



all capacities are 1

Reducing to Maximum Flow

Theorem: Every integer solution to the max flow problem on the constructed graph induces a maximum bipartite matching of G.

Proof:

1. An integer flow f of value $|f|$ induces a matching of size $|f|$
   - Left nodes (gifts) have incoming capacity 1
   - Right nodes (children) have outgoing capacity 1
   - Left and right nodes are incident to ≤ 1 edge e of G with $f(e) = 1$

2. A matching of size k implies a flow f of value $|f| = k$
   - For each edge $\{u, v\}$ of the matching:

$$f\big((s,u)\big) = f\big((u,v)\big) = f\big((v,t)\big) = 1$$

   - All other flow values are 0

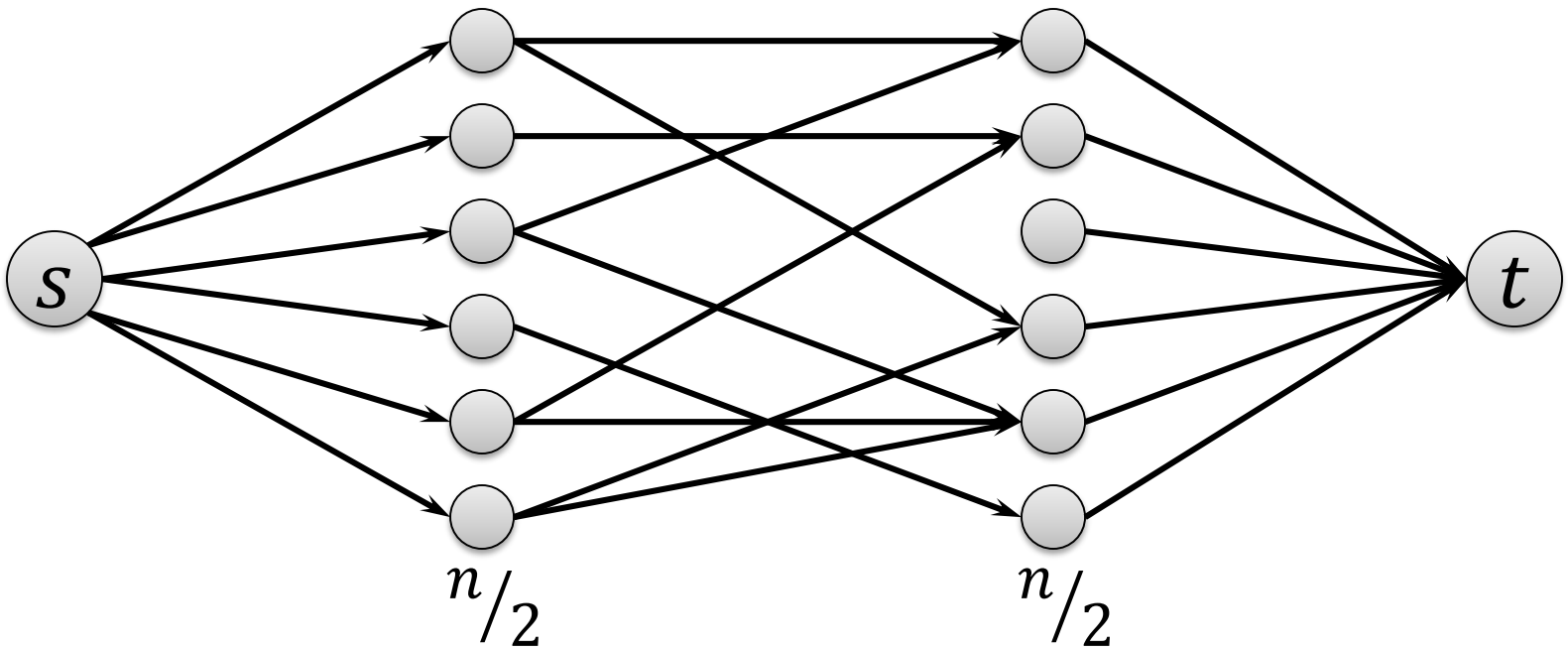
Running Time of Max. Bipartite Matching

Theorem: A maximum matching of a bipartite graph can be computed in time $O(m \cdot n)$.

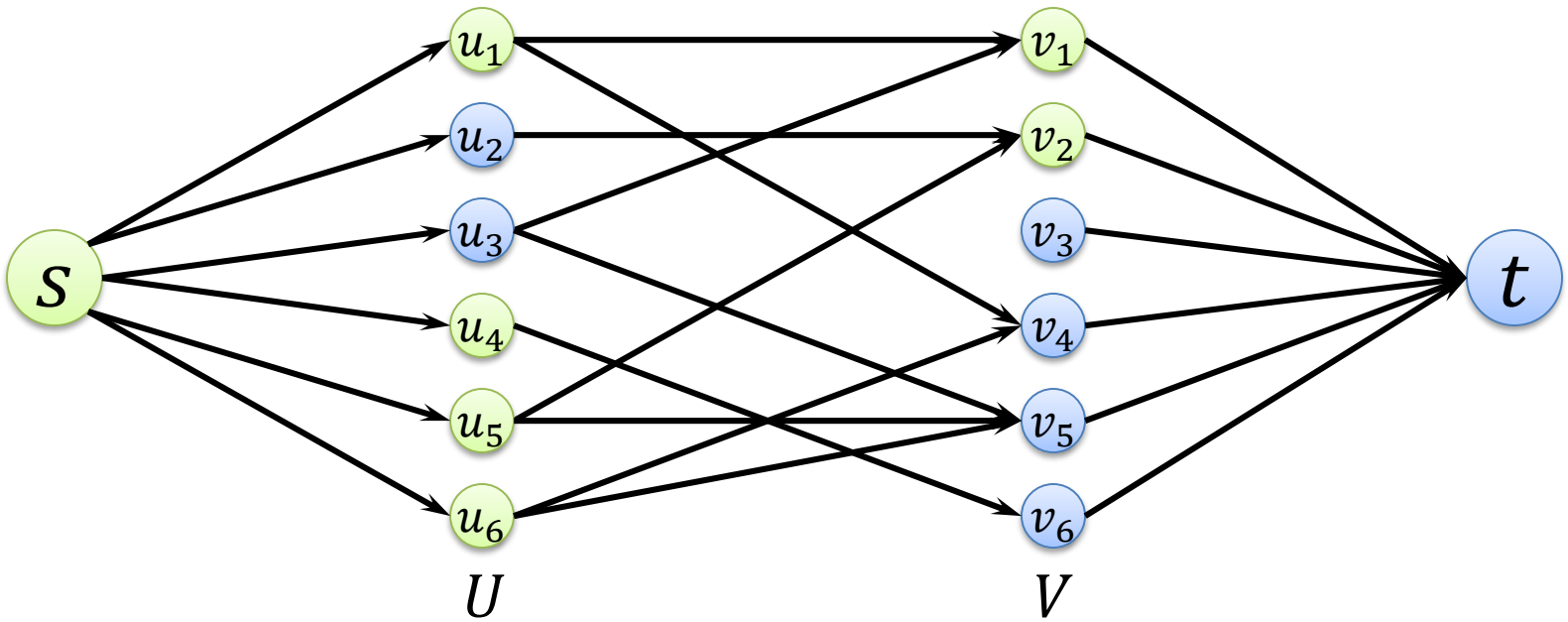
Perfect Matching?

- There can only be a perfect matching if both sides of the partition have size $n/2$.

- There is no perfect matching, iff there is an s-t cut of size $< n/2$ in the flow network.



s-t Cuts



Partition (A, B) of node set such that $s \in A$ and $t \in B$

- If $v_i \in A$: edge (v_i, t) is in cut (A, B)
- If $u_i \in B$: edge (s, u_i) is in cut (A, B)
- Otherwise (if $u_i \in A$, $v_i \in B$), all edges from u_i to some $v_j \in B$ are in cut (A, B)

Hall's Marriage Theorem

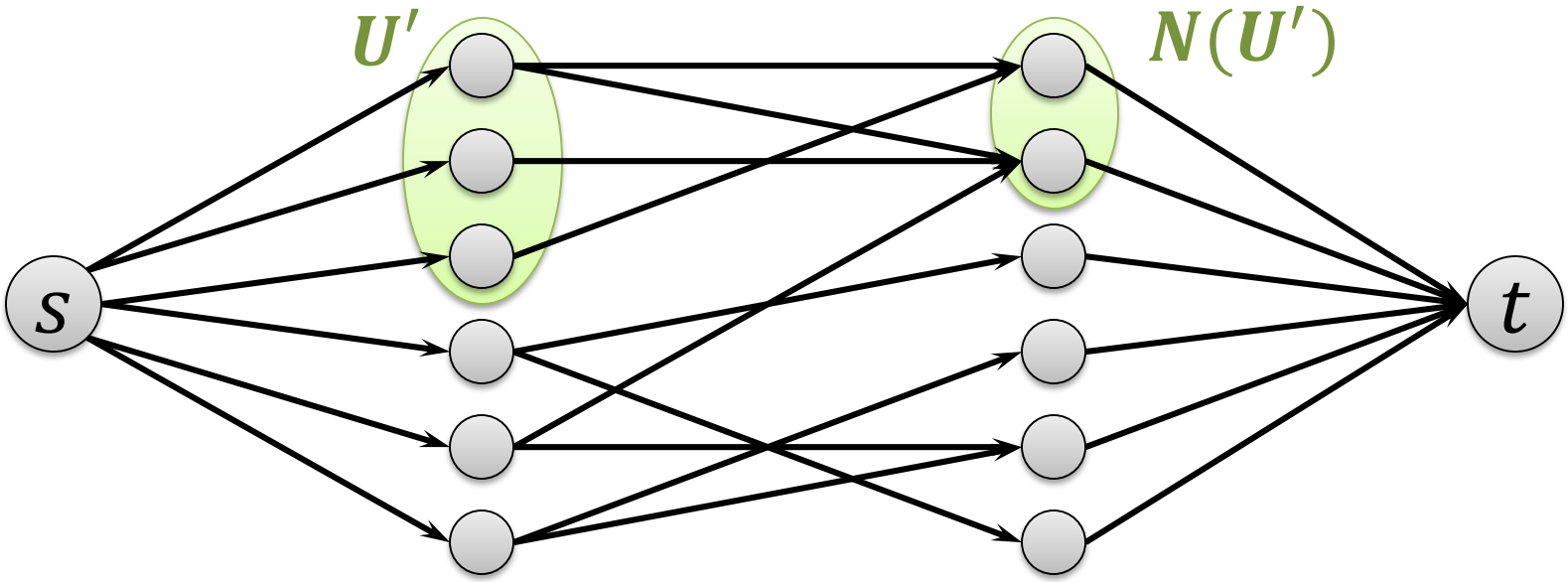
Theorem: A bipartite graph $G = (U \cup V, E)$ for which $|U| = |V|$ has a perfect matching if and only if

$$\forall \boldsymbol{U'} \subseteq \boldsymbol{U}: |\boldsymbol{N}(\boldsymbol{U'})| \geq |\boldsymbol{U'}|,$$

where $N(U') \subseteq V$ is the set of neighbors of nodes in U'.

Proof: No perfect matching $\Leftrightarrow$ some s-t cut has capacity $< n/2$

1. Assume there is U' for which $|N(U')| < |\mathrm{U}'|$:



Hall's Marriage Theorem

Theorem: A bipartite graph $G = (U \cup V, E)$ for which $|U| = |V|$ has a perfect matching if and only if

$$\forall \boldsymbol{U'} \subseteq \boldsymbol{U}: |\boldsymbol{N}(\boldsymbol{U'})| \geq |\boldsymbol{U'}|,$$

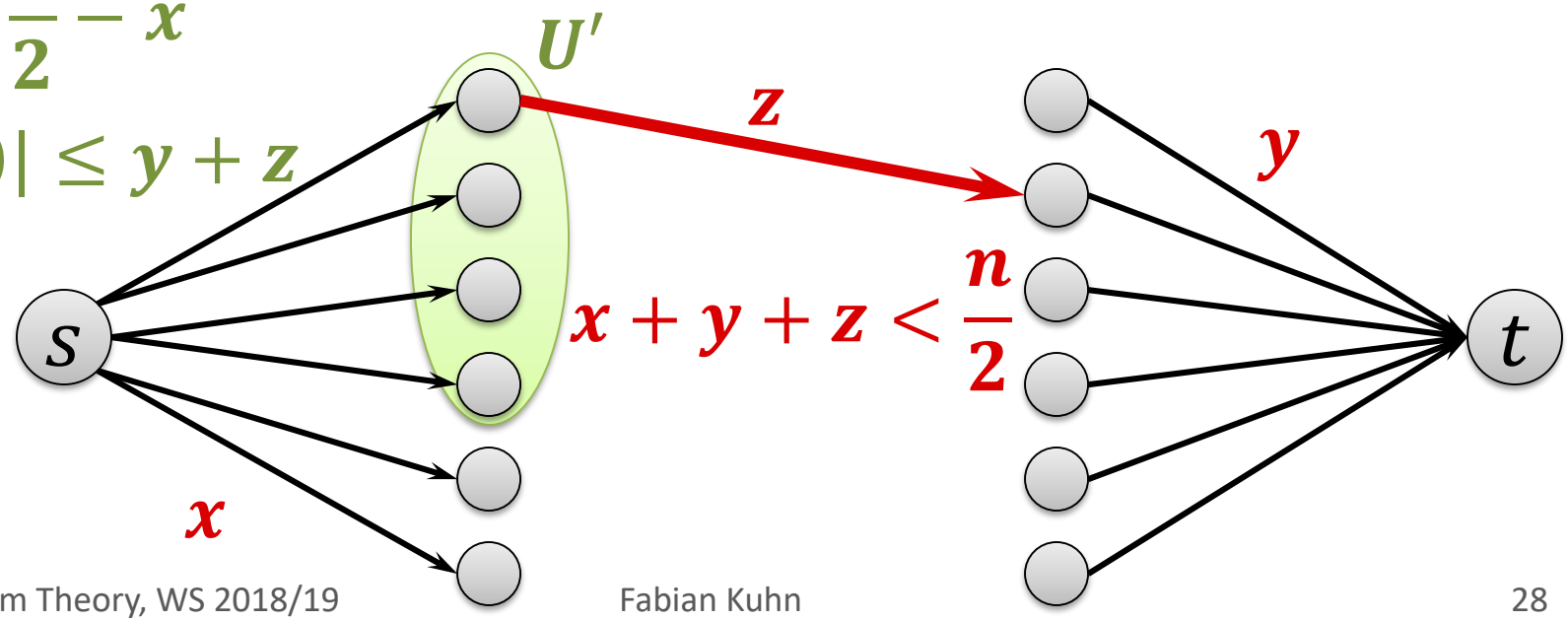
where $N(U') \subseteq V$ is the set of neighbors of nodes in U'.

Proof: No perfect matching $\Leftrightarrow$ some s-t cut has capacity $< n/2$

2. Assume that there is a cut (A, B) of capacity $< n/2$

$$|\boldsymbol{U'}| = \frac{\boldsymbol{n}}{\boldsymbol{2}} - \boldsymbol{x}$$

$$|\boldsymbol{N}(\boldsymbol{U'})| \leq \boldsymbol{y} + \boldsymbol{z}$$



Hall's Marriage Theorem

Theorem: A bipartite graph $G = (U \cup V, E)$ for which $|U| = |V|$ has a perfect matching if and only if

$$\forall \boldsymbol{U'} \subseteq \boldsymbol{U}: |\boldsymbol{N}(\boldsymbol{U'})| \geq |\boldsymbol{U'}|,$$

where $N(U') \subseteq V$ is the set of neighbors of nodes in U'.

Proof: No perfect matching $\Leftrightarrow$ some s-t cut has capacity $< n$

2. Assume that there is a cut (A, B) of capacity $< n$

$$|\boldsymbol{U'}| = \frac{\boldsymbol{n}}{\boldsymbol{2}} - \boldsymbol{x}$$

$$|\boldsymbol{N}(\boldsymbol{U'})| \leq \boldsymbol{y} + \boldsymbol{z}$$

$$\boldsymbol{x} + \boldsymbol{y} + \boldsymbol{z} < \frac{\boldsymbol{n}}{\boldsymbol{2}}$$

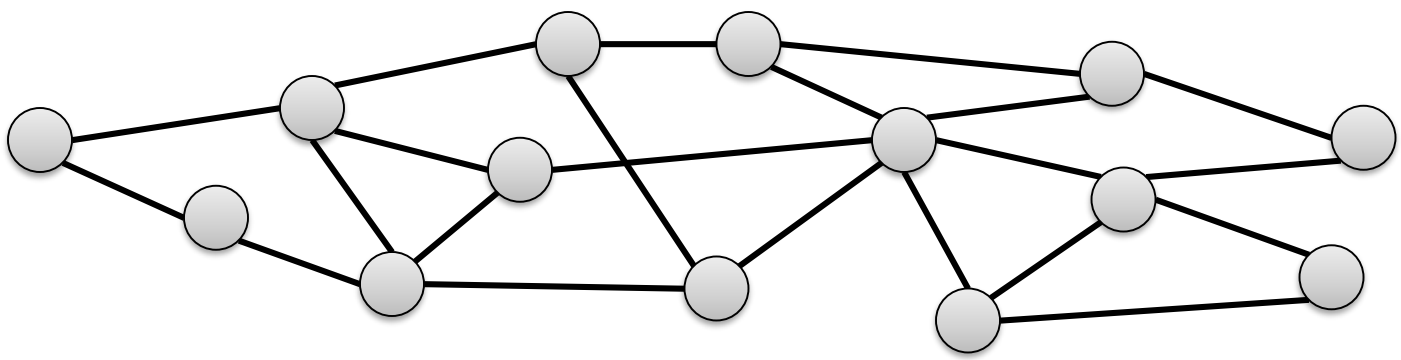
What About General Graphs

- Can we efficiently compute a maximum matching if G is not bipartite?

- How good is a maximal matching?
  - A matching that cannot be extended…

- **Vertex Cover:** set $S \subseteq V$ of nodes such that

$$\forall \{u, v\} \in E, \qquad \{u, v\} \cap S \neq \emptyset.$$



- A vertex cover covers all edges by incident nodes

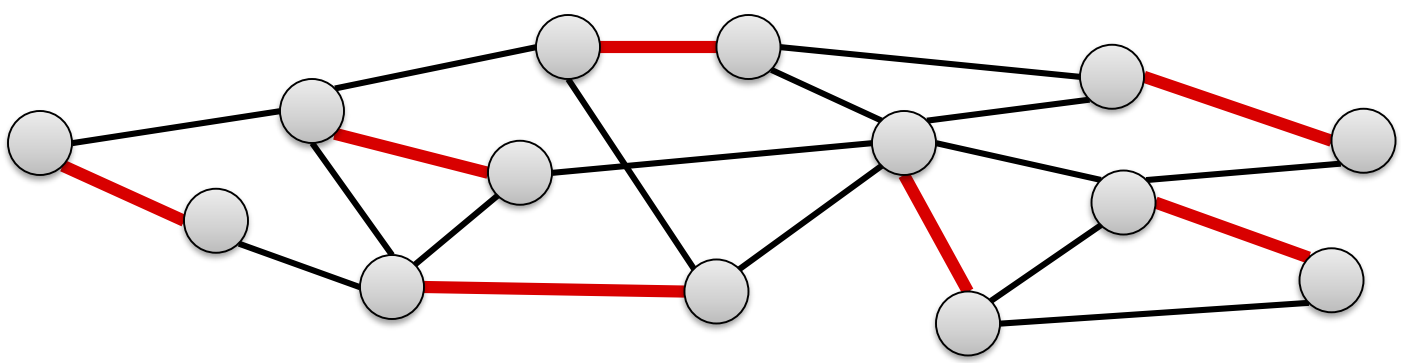
Vertex Cover vs Matching

Consider a matching M and a vertex cover S

Claim: $|M| \leq |S|$

Proof:

- At least one node of every edge $\{u, v\} \in M$ is in S
- Needs to be a different node for different edges from M



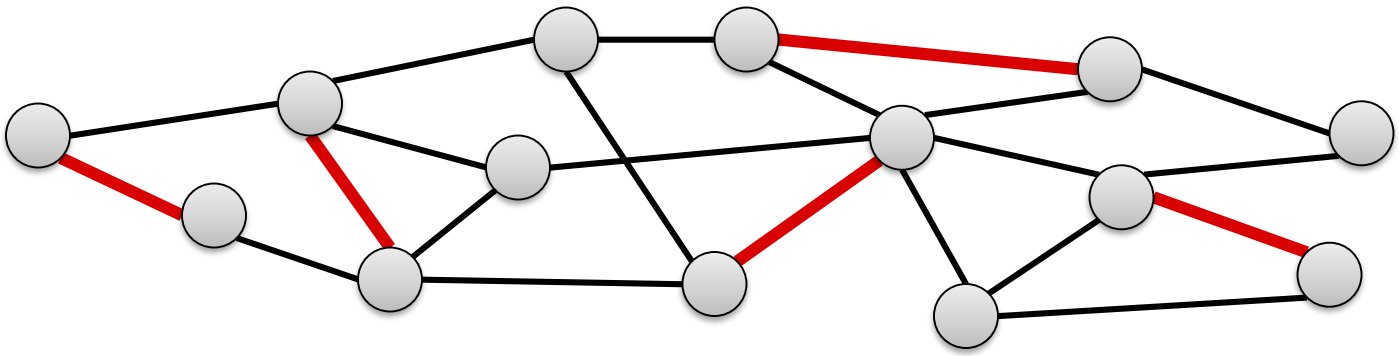
Vertex Cover vs Matching

Consider a matching M and a vertex cover S

Claim: If M is maximal and S is minimum, $|S| \leq 2|M|$

Proof:

- M is maximal: for every edge $\{u, v\} \in E$, either u or v (or both) are matched



- Every edge $e \in E$ is "covered" by at least one matching edge
- Thus, the set of the nodes of all matching edges gives a vertex cover S of size $|S| = 2|M|$.

Maximal Matching Approximation

Theorem: For any maximal matching M and any maximum matching M^*, it holds that

$$|M| \geq \frac{|M^*|}{2}.$$

Proof:

Theorem: The set of all matched nodes of a maximal matching M is a vertex cover of size at most twice the size of a min. vertex cover.

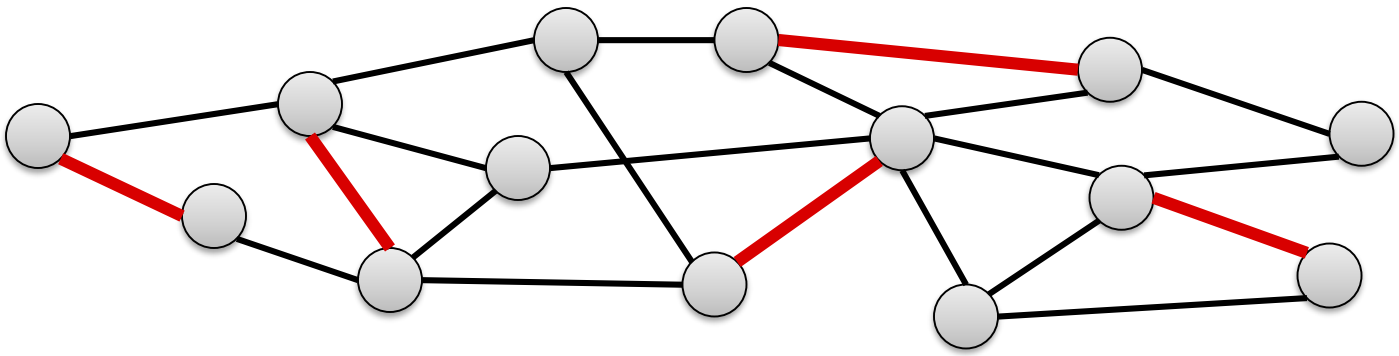
Chapter 6
Graph Algorithms

Algorithm Theory
WS 2018/19

Fabian Kuhn

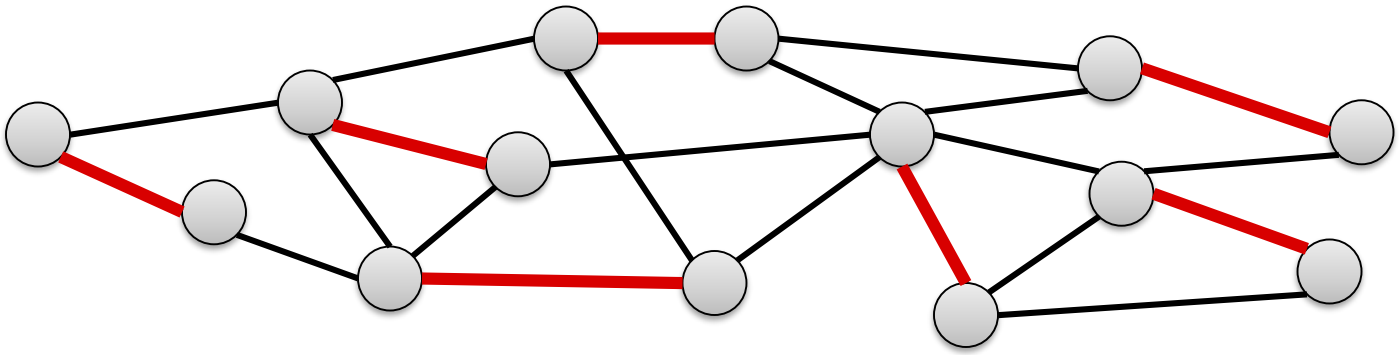
Matching

Matching: Set of pairwise non-incident edges



Maximal Matching: A matching s.t. no more edges can be added

Maximum Matching: A matching of maximum possible size



Perfect Matching: Matching of size $n/2$ (every node is matched)

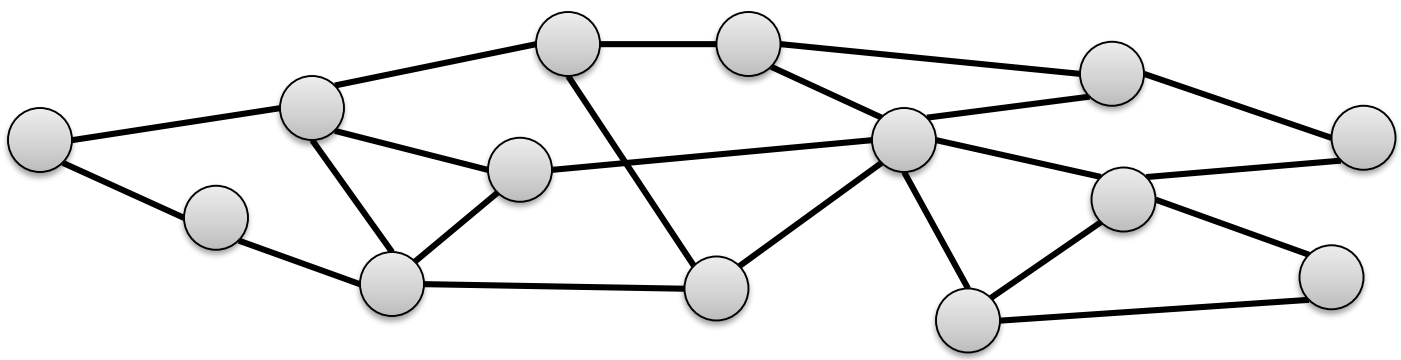
What About General Graphs

- Can we efficiently compute a maximum matching if G is not bipartite?

- How good is a maximal matching?
  - A matching that cannot be extended…

- **Vertex Cover:** set $S \subseteq V$ of nodes such that

$$\forall \{u, v\} \in E, \qquad \{u, v\} \cap S \neq \emptyset.$$



- A vertex cover covers all edges by incident nodes

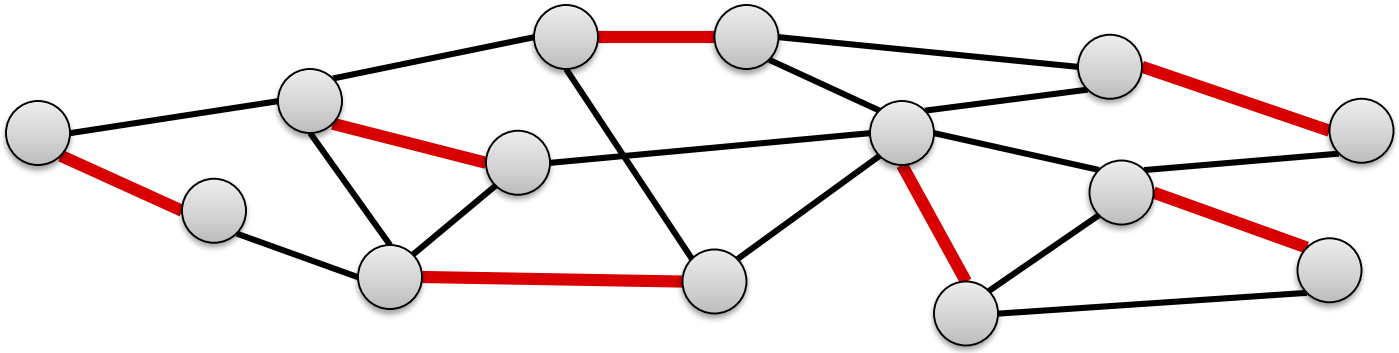
Vertex Cover vs Matching

Consider a matching M and a vertex cover S

Claim: $|M| \leq |S|$

Proof:

- At least one node of every edge $\{u, v\} \in M$ is in S
- Needs to be a different node for different edges from M



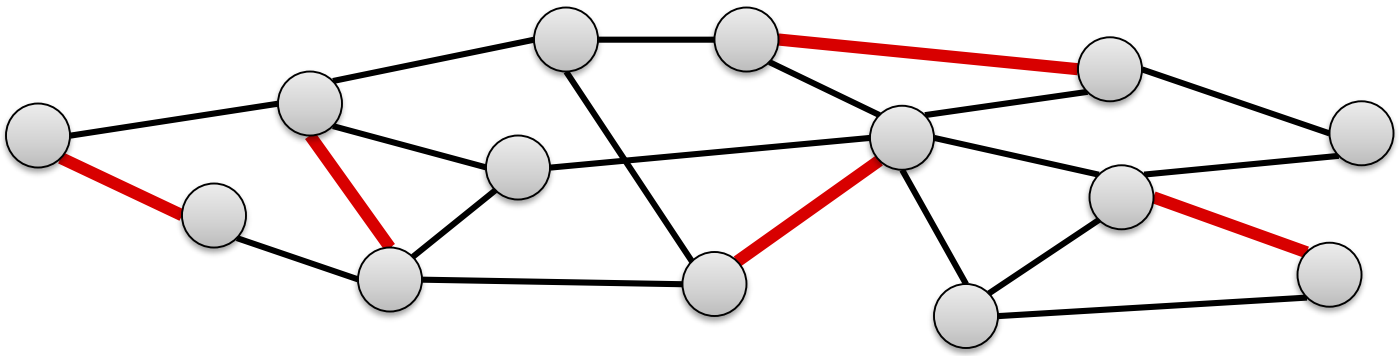
Vertex Cover vs Matching

Consider a matching M and a vertex cover S

Claim: If M is maximal and S is minimum, $|S| \leq 2|M|$

Proof:

- M is maximal: for every edge $\{u, v\} \in E$, either u or v (or both) are matched



- Every edge $e \in E$ is "covered" by at least one matching edge
- Thus, the set of the nodes of all matching edges gives a vertex cover S of size $|S| = 2|M|$.

Maximal Matching Approximation

Theorem: For any maximal matching M and any maximum matching M^*, it holds that

$$|M| \geq \frac{|M^*|}{2}.$$

Proof:

- Let S^* be a minimum vertex cover:

$$|M^*| \leq |S^*| \leq 2|M|$$

Theorem: The set of all matched nodes of a maximal matching M is a vertex cover S of size at most twice the size of a min. vertex cover.

Proof:

- Let S^* be a minimum vertex cover

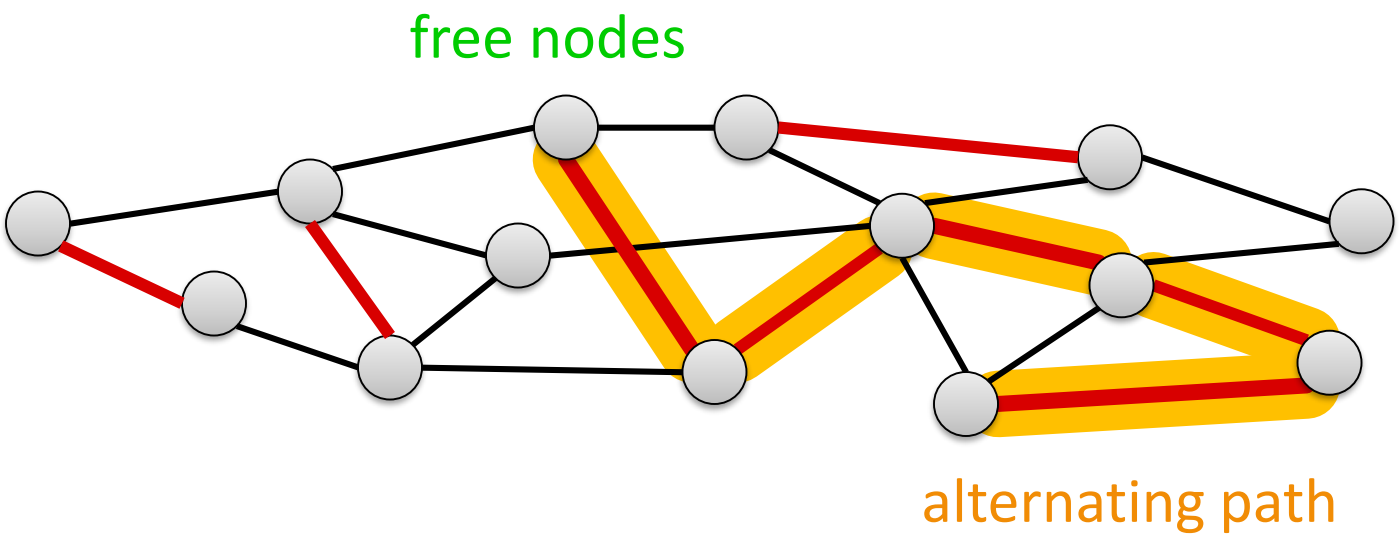
$$|S| = 2|M| \leq 2|S^*|$$

Augmenting Paths

Consider a matching M of a graph $G = (V, E)$:

- A node $v \in V$ is called **free** iff it is not matched

Augmenting Path: A (odd-length) path that starts and ends at a free node and visits edges in $E \setminus M$ and edges in M alternatingly.



- Matching M can be improved using an augmenting path by switching the role of each edge along the path

Augmenting Paths

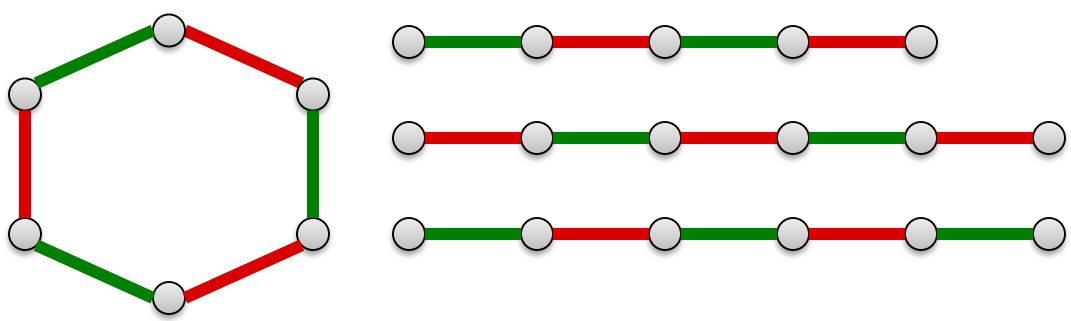
Theorem: A matching M of $G = (V, E)$ is maximum if and only if there is no augmenting path.

Proof:

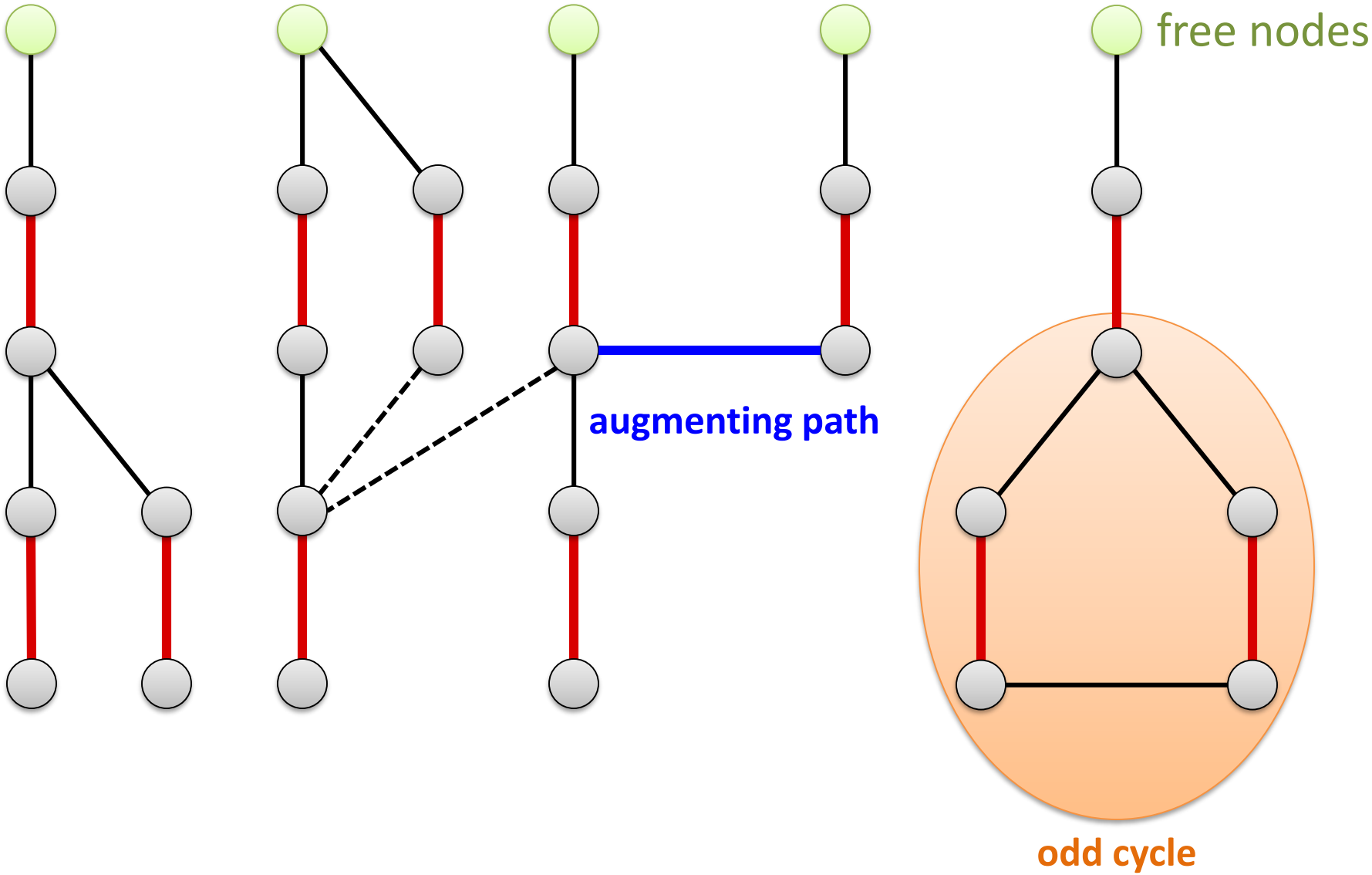
- Consider non-max. matching M and max. matching M^* and define

$$F := M \setminus M^*, \qquad F^* := M^* \setminus M$$

- Note that $F \cap F^* = \emptyset$ and $|F| < |F^*|$
- Each node $v \in V$ is incident to at most one edge in both F and F^*
- $F \cup F^*$ induces even cycles and paths

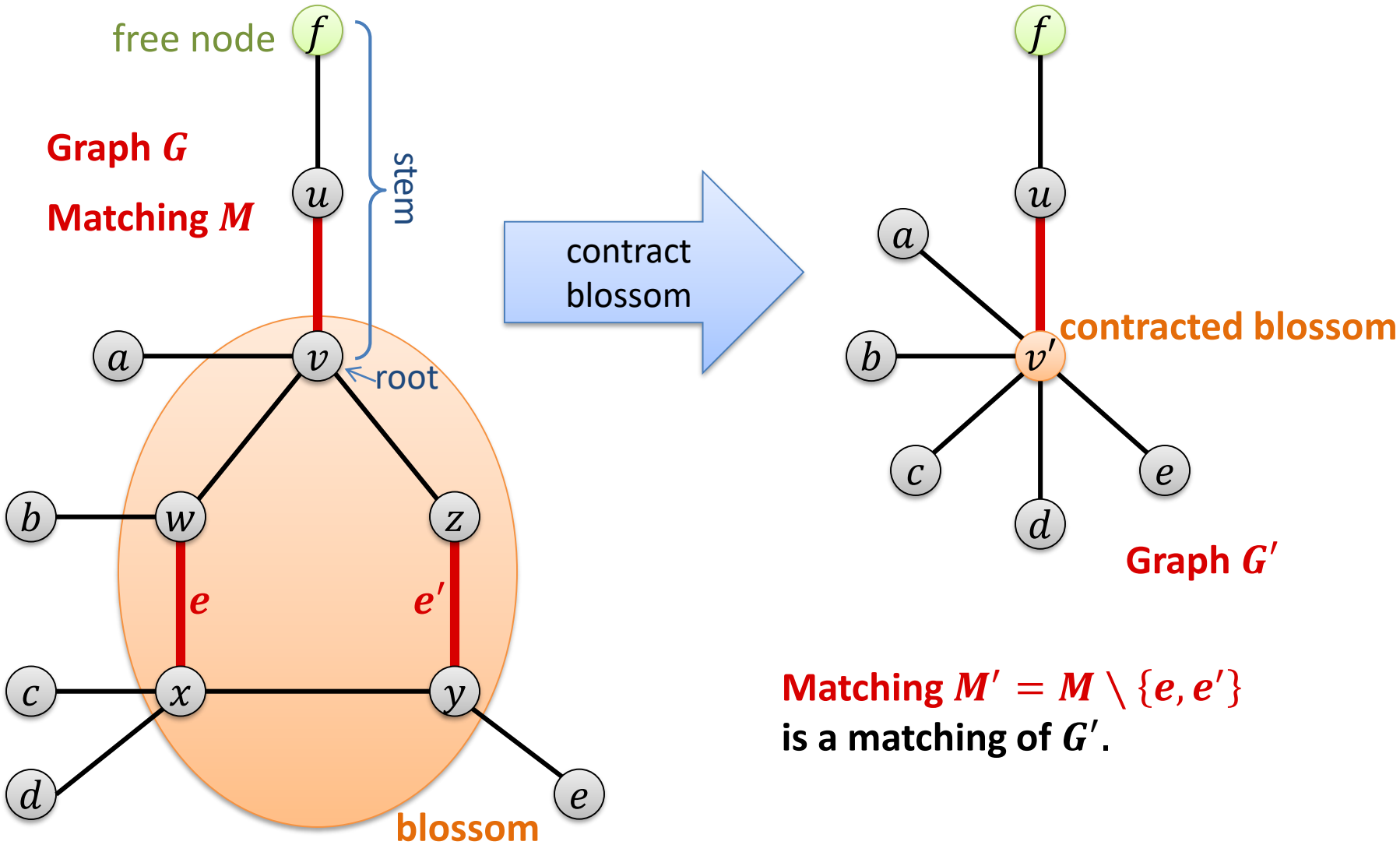


Finding Augmenting Paths



Blossoms

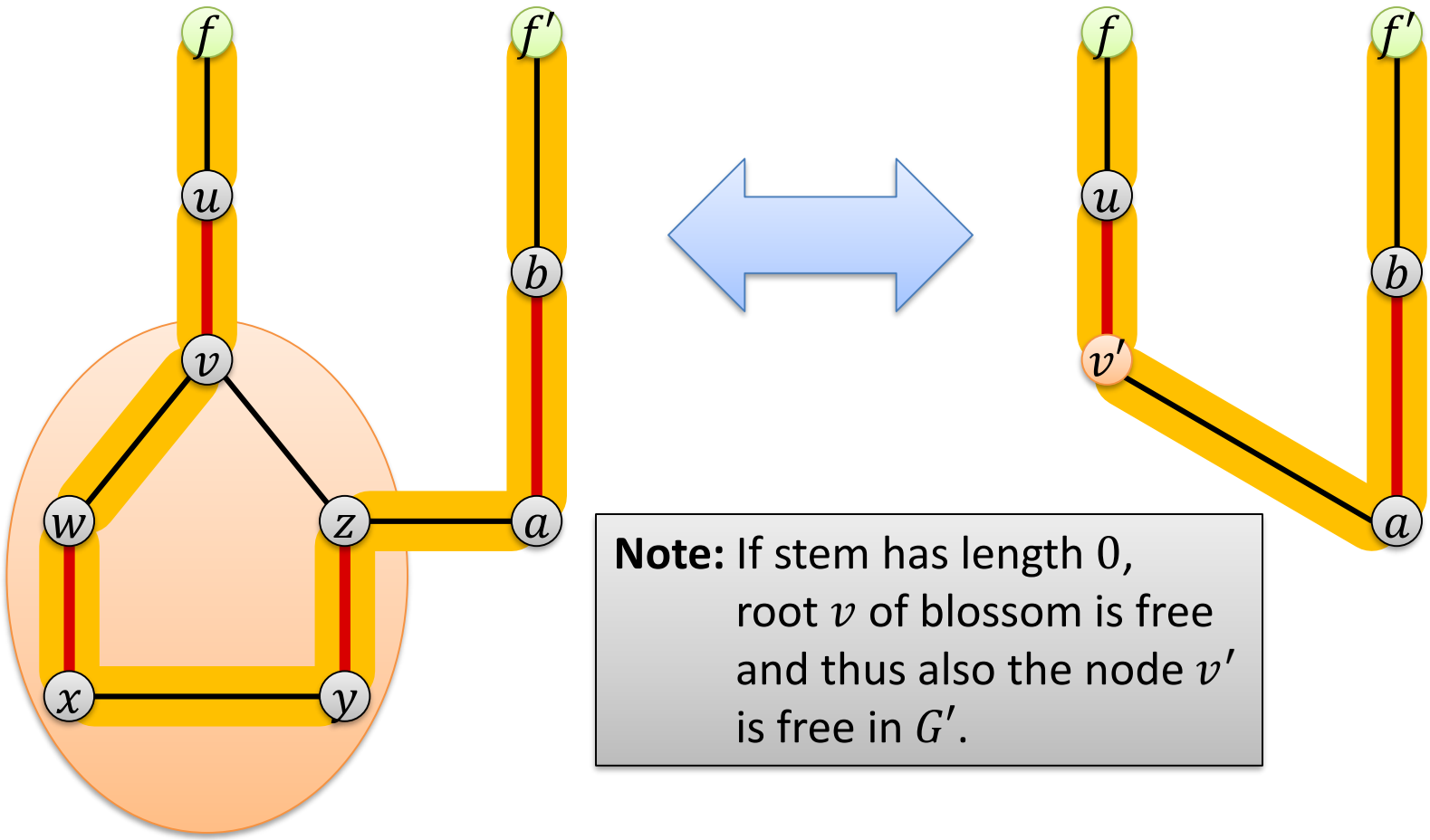
- If we find an odd cycle…



Matching $M' = M \setminus \{e, e'\}$ is a matching of G'.

Contracting Blossoms

Lemma: Graph G has an augmenting path w.r.t. matching M iff G' has an augmenting path w.r.t. matching M'



Note: If stem has length 0, root v of blossom is free and thus also the node v' is free in G'.

Also: The matching M can be computed efficiently from M'.

Edmond's Blossom Algorithm

Algorithm Sketch:

1. Build a tree for each free node
2. Starting from an explored node u at even distance from a free node f in the tree of f, explore some unexplored edge $\{u, v\}$:

   1. If v is an unexplored node, v is matched to some neighbor w:
      add w to the tree (w is now explored)

   2. If v is explored and in the same tree:
      at odd distance from root → ignore and move on
      at even distance from root → **blossom found**

   3. If v is explored and in another tree
      at odd distance from root → ignore and move on
      at even distance from root → **augmenting path found**

Running Time

Finding a Blossom: Repeat on smaller graph

Finding an Augmenting Path: Improve matching

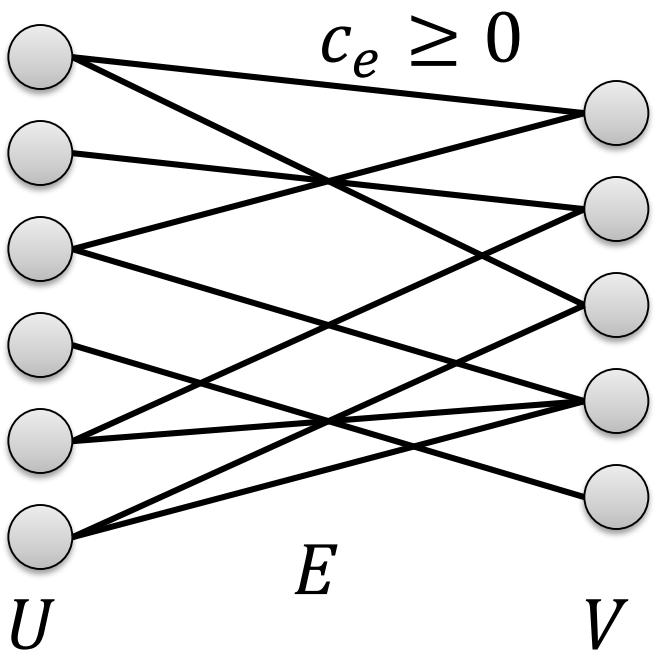
Theorem: The algorithm can be implemented in time $O(mn^2)$.

Maximum Weight Bipartite Matching

- Let's again go back to bipartite graphs…

Given: Bipartite graph $G = (U \dot{\cup} V, E)$ with edge weights $c_e \geq 0$

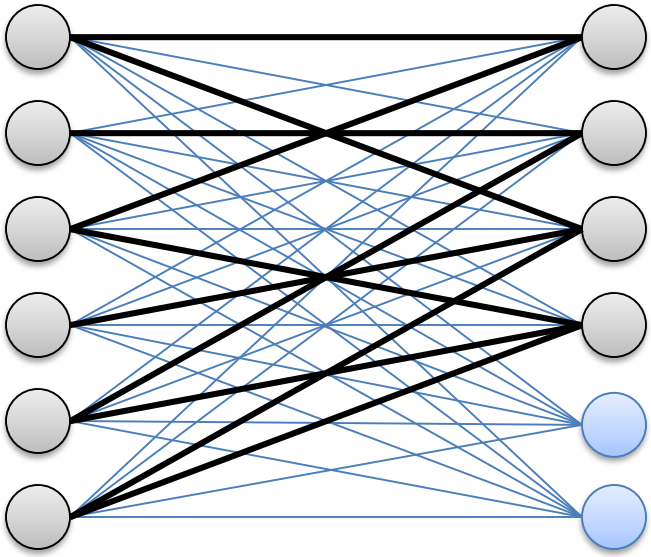
Goal: Find a matching M of maximum total weight



Minimum Weight Perfect Matching

Claim: Max weight bipartite matching is equivalent to finding a minimum weight perfect matching in a complete bipartite graph.

1. Turn into maximum weight perfect matching
   - add dummy nodes to get two equal-sized sides
   - add edges of weight 0 to make graph complete bipartite
2. Replace weights: $c'_e := \max_f \{c_f\} - c_e$



As an Integer Linear Program

- We can formulate the problem as an integer linear program

Var. x_{uv} for every edge $(u, v) \in U \times V$ to encode matching M:

$$x_{uv} = \begin{cases} 1, & \text{if } \{u, v\} \in M \\ 0, & \text{if } \{u, v\} \notin M \end{cases}$$

Minimum Weight Perfect Matching

Linear Programming (LP) Relaxation

Linear Program (LP)

- Continuous optimization problem on multiple variables with a linear objective function and a set of linear side constraints

LP Relaxation of Minimum Weight Perfect Matching

- Weight c_{uv} & variable x_{uv} for ever edge $(u, v) \in U \times V$

$$\min \sum_{(u,v)\in U\times V} c_{uv} \cdot x_{uv}$$

$$\text{s.t.}$$

$$\forall u \in U: \sum_{v\in V} x_{uv} = 1,$$

$$\forall v \in V: \sum_{u\in U} x_{uv} = 1$$

$$\forall u \in U, \forall v \in V: \; x_{uv} \geq 0$$

Dual Problem

- Every linear program has a dual linear program
  - The dual of a minimization problem is a maximization problem
  - Strong duality: primal LP and dual LP have the same objective value

In the case of the minimum weight perfect matching problem

- Assign a variable $a_u \geq 0$ to each node $u \in U$
  and a variable $b_v \geq 0$ to each node $v \in V$
- **Condition:** for every edge $(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{U} \times \boldsymbol{V}$: $\boldsymbol{a_u + b_v \leq c_{uv}}$
- Given perfect matching M:

$$\sum_{(u,v)\in M} c_{uv} \geq \sum_{u\in U} a_u + \sum_{v\in V} b_v$$

Dual Linear Program

- Variables $a_u \geq 0$ for $u \in U$ and $b_v \geq 0$ for $v \in V$

$$\max \sum_{u \in U} a_u + \sum_{v \in V} b_v$$
$$s.t.$$
$$\forall u \in U, \forall v \in V: \; a_u + b_v \leq c_{uv}$$

- For every perfect matching M:

$$\sum_{(u,v) \in M} c_{uv} \geq \sum_{u \in U} a_u + \sum_{v \in V} b_v$$

Complementary Slackness

- A perfect matching M is optimal if

$$\sum_{(u,v)\in M} c_{uv} = \sum_{u\in U} a_u + \sum_{v\in V} b_v$$

- In that case, for every $(u,v) \in M$

$$\boldsymbol{w_{uv}} := c_{uv} - a_u - b_v = 0$$

  - In this case, M is also an optimal solution to the LP relaxation of the problem
  - Every optimal LP solution can be characterized by such a property, which is then generally referred to as complementary slackness

- **Goal:** Find a dual solution a_u, b_v and a perfect matching such that the complementary slackness condition is satisfied!
  - i.e., for every matching edge (u,v), we want $w_{uv} = 0$
  - We then know that the matching is optimal!

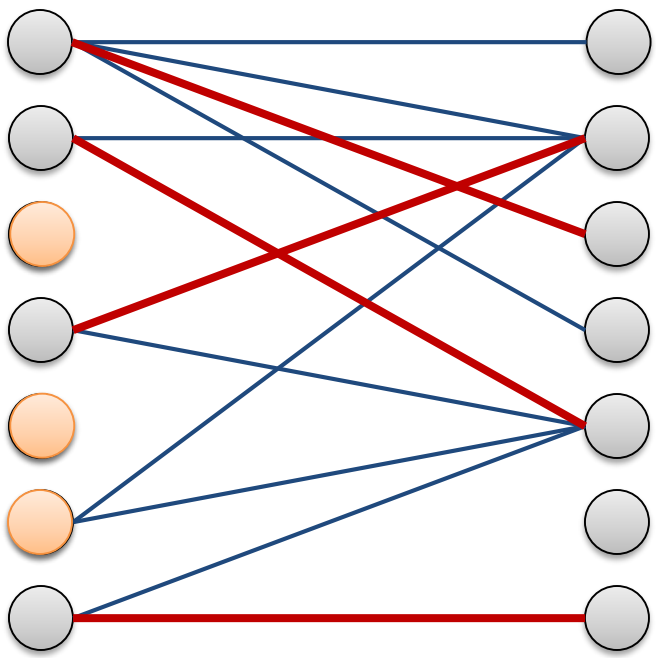
Algorithm Overview

- Start with any feasible dual solution a_u, b_v
  - i.e., solution satisfies that for all (u, v): $c_{uv} \geq a_u + b_v$
- Let E_0 be the edges for which $w_{uv} = 0$
  - Recall that $w_{uv} = c_{uv} - a_u - b_v$
- Compute maximum cardinality matching M of E_0
- All edges (u, v) of M satisfy $w_{uv} = 0$
  - Complementary slackness if satisfied
  - If M is a perfect matching, we are done
- If M is not a perfect matching, dual solution can be improved

Marked Nodes

Define set of marked nodes L:

- Set of nodes which can be reached on alternating paths on edges in E_0 starting from unmatched nodes in U



edges E_0 with $w_{uv} = 0$

optimal matching M

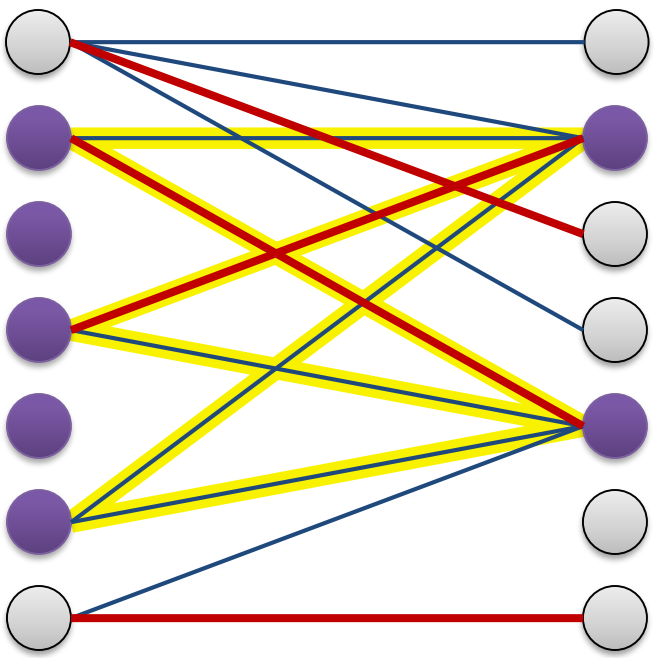
L_0: unmatched nodes in U

L: all nodes that can be reached on alternating paths starting from L_0

Marked Nodes

Define set of marked nodes L:

- Set of nodes which can be reached on alternating paths on edges in E_0 starting from unmatched nodes in U



edges E_0 with $w_{uv} = 0$

optimal matching M

L_0: unmatched nodes in U

L: all nodes that can be reached on alternating paths starting from L_0

Marked Nodes – Vertex Cover

Lemma:

a) There are no E_0-edges between $U \cap L$ and $V \setminus L$

b) The set $(U \setminus L) \cup (V \cap L)$ is a vertex cover of size $|M|$ of the graph induced by E_0

Improved Dual Solution

Recall: all edges (u, v) between $U \cap L$ and $V \setminus L$ have $w_{uv} > 0$

New dual solution:

$$\delta := \min_{u \in U \cap L,\, v \in V \setminus \mathrm{L}} \{w_{uv}\}$$

$$a'_u := \begin{cases} a_u, & \text{if } u \in U \setminus \mathrm{L} \\ a_u + \delta, & \text{if } u \in U \cap L \end{cases}$$

$$b'_v := \begin{cases} b_v, & \text{if } v \in V \setminus \mathrm{L} \\ a_v - \delta, & \text{if } v \in V \cap L \end{cases}$$

Claim: New dual solution is feasible (all w_{uv} remain ≥ 0)

Improved Dual Solution

Lemma: Obj. value of the dual solution grows by $\delta\left(\frac{n}{2} - |M|\right)$.

Proof:

$$\delta := \min_{u \in U \cap L,\, v \in V \setminus \mathrm{L}} \{w_{uv}\}, \qquad a'_u := \begin{cases} a_u, & \text{if } u \in U \setminus \mathrm{L} \\ a_u + \delta, & \text{if } u \in U \cap L \end{cases}, \qquad b'_v := \begin{cases} b_v, & \text{if } v \in V \setminus \mathrm{L} \\ a_v - \delta, & \text{if } v \in V \cap L \end{cases}$$

Termination

Some terminology

- Old dual solution: $a_u, \quad b_v, \quad w_{uv} \coloneqq c_{uv} - a_u - b_v$
- New dual solution: $a'_u, \quad b'_v, \quad w'_{uv} \coloneqq c_{uv} - a'_u - b'_v$
- $E_0 \coloneqq \{(u,v) : w_{uv} = 0\}, \quad E'_0 \coloneqq \{(u,v) : w'_{uv} = 0\}$
- M, M' : max. cardinality matchings of graphs ind. By $E_0,\ E'_0$

Claim: $|\boldsymbol{M}'| \geq |\boldsymbol{M}|$ and if $|M'| = |M|$, we can assume that $M = M'$.

Termination

Lemma: The algorithm terminates in at most $O(n^2)$ iterations.

Proof:

- Each iteration: $M' > M$ or $M' = M$ and $|V \cap L'| > |V \cap L|$

Min. Weight Perfect Matching: Summary

Theorem: A minimum weight perfect matching can be computed in time $O(n^4)$.

- First dual solution: e.g., $a_u = 0$, $b_v = \min_{u \in U} c_{uv}$

- Compute set E_0: $O(n^2)$

- Compute max. cardinality matching of graph induced by E_0
  - First iteration: $O(n^2) \cdot O(n) = O(n^3)$
  - Other iterations: $O(n^2) \cdot O(1 + |M'| - |M|)$

Matching Algorithms

We have seen:

- $O(mn)$ time alg. to compute a max. matching in *bipartite graphs*
- $O(mn^2)$ time alg. to compute a max. matching in *general graphs*

Better algorithms:

- Best known running time (bipartite and general gr.): $O(m\sqrt{n})$

Weighted matching:

- Edges have weight, find a matching of **maximum total weight**
- *Bipartite graphs*: polynomial-time primal-dual algorithm
- *General graphs*: can also be solved in polynomial time
  (Edmond's algorithms is used as blackbox)

Chapter 7
Randomization

Algorithm Theory
WS 2018/19

Fabian Kuhn

Randomization

Randomized Algorithm:

- An algorithm that uses (or can use) random coin flips in order to make decisions

We will see: randomization can be a powerful tool to

- Make algorithms faster
- Make algorithms simpler
- Make the analysis simpler
  - Sometimes it's also the opposite…
- Allow to solve problems (efficiently) that cannot be solved (efficiently) without randomization
  - True in some computational models (e.g., for distributed algorithms)
  - Not clear in the standard sequential model

Contention Resolution

A simple starter example (from distributed computing)

- Allows to introduce important concepts
- … and to repeat some basic probability theory

Setting:

- n processes, 1 resource
  (e.g., communication channel, shared database, …)
- There are time slots $1,2,3$, …
- In each time slot, only one client can access the resource
- All clients need to regularly access the resource
- If client i tries to access the resource in slot t:
  - Successful iff no other client tries to access the resource in slot t

Algorithm

Algorithm Ideas:

- Accessing the resource deterministically seems hard
  - need to make sure that processes access the resource at different times
  - or at least: often only a single process tries to access the resource

- **Randomized solution:**
  In each time slot, each process tries with probability p.

Analysis:

- How large should p be?
- How long does it take until some process i succeeds?
- How long does it take until all processes succeed?
- What are the probabilistic guarantees?

Analysis

Events:

- $\boldsymbol{\mathcal{A}_{x,t}}$: process x tries to access the resource in time slot t
  - Complementary event: $\overline{\mathcal{A}_{x,t}}$

$$\mathbb{P}(\mathcal{A}_{x,t}) = p, \qquad \mathbb{P}(\overline{\mathcal{A}_{x,t}}) = 1 - p$$

- $\boldsymbol{\mathcal{S}_{x,t}}$: process x is successful in time slot t

$$\mathcal{S}_{x,t} = \mathcal{A}_{x,t} \cap \left(\bigcap_{y \neq x} \overline{\mathcal{A}_{y,t}} \right)$$

- **Success probability** (for process x)**:**

Fixing p

- $\mathbb{P}(\mathcal{S}_{x,t}) = p(1-p)^{n-1}$ is maximized for

$$\boldsymbol{p = \frac{1}{n}} \qquad \Longrightarrow \qquad \mathbb{P}(\mathcal{S}_{x,t}) = \frac{1}{n}\left(1-\frac{1}{n}\right)^{n-1}.$$

- **Asymptotics:**

$$\text{For } n \geq 2: \quad \frac{1}{4} \leq \left(1-\frac{1}{n}\right)^{n} < \frac{1}{e} < \left(1-\frac{1}{n}\right)^{n-1} \leq \frac{1}{2}$$

- **Success probability:**

$$\boldsymbol{\frac{1}{en} < \mathbb{P}(\mathcal{S}_{x,t}) \leq \frac{1}{2n}}$$

Time Until First Success

Random Variable $\boldsymbol{T_x}$:

- $T_x = t$ if proc. x is successful in slot t for the first time

- **Distribution:**

- T_x is geometrically distributed with parameter

$$q = \mathbb{P}(\mathcal{S}_{x,t}) = \frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1} > \frac{1}{en}.$$

- **Expected time** until first success:

$$\mathbb{E}[\boldsymbol{T_x}] = \frac{\boldsymbol{1}}{\boldsymbol{q}} < \boldsymbol{en}$$

Time Until First Success

Failure Event $\mathcal{F}_{x,t}$: Process x does not succeed in time slots $1, \dots, t$

- The events $\mathcal{S}_{x,t}$ are independent for different t:

$$\mathbb{P}(\mathcal{F}_{x,t}) = \mathbb{P}\left(\bigcap_{r=1}^{t} \overline{\mathcal{S}_{x,r}}\right) = \prod_{r=1}^{t} \mathbb{P}(\overline{\mathcal{S}_{x,r}}) = \left(1 - \mathbb{P}(\mathcal{S}_{x,r})\right)^t$$

- We know that $\mathbb{P}(\mathcal{S}_{x,r}) > {}^{1}/_{en}$:

$$\mathbb{P}(\mathcal{F}_{x,t}) < \left(1 - \frac{1}{en}\right)^t < e^{-t/en}$$

Time Until First Success

No success by time t: $\mathbb{P}(\mathcal{F}_{x,t}) < e^{-t/en}$

$t = \lceil en \rceil$: $\mathbb{P}(\mathcal{F}_{x,t}) < 1/e$

- Generally if $t = \Theta(n)$: constant success probability

$t \geq en \cdot c \cdot \ln n$: $\mathbb{P}(\mathcal{F}_{x,t}) < 1/e^{c \cdot \ln n} = 1/n^c$

- For success probability $1 - 1/n^c$, we need $t = \Theta(n \log n)$.

- We say that x succeeds **with high probability** in $O(n \log n)$ time.

Time Until All Processes Succeed

Event $\mathcal{F}_t$: some process has not succeeded by time t

$$\mathcal{F}_t = \bigcup_{x=1}^{n} \mathcal{F}_{x,t}$$

Union Bound: For events $\mathcal{E}_1, \dots, \mathcal{E}_k$,

$$\mathbb{P}\left(\bigcup_{x}^{k} \mathcal{E}_x\right) \leq \sum_{x}^{k} \mathbb{P}(\mathcal{E}_x)$$

Probability that not all processes have succeeded by time t:

$$\mathbb{P}(\mathcal{F}_t) = \mathbb{P}\left(\bigcup_{x=1}^{n} \mathcal{F}_{x,t}\right) \leq \sum_{x=1}^{n} \mathbb{P}\left(\mathcal{F}_{x,t}\right) < n \cdot e^{-t/en}.$$

Time Until All Processes Succeed

Claim: With high probability, all processes succeed in the first $O(n \log n)$ time slots.

Proof:

- $\mathbb{P}(\mathcal{F}_t) < n \cdot e^{-t/en}$
- Set $t = \lceil en \cdot (c+1) \ln n \rceil$

Remark: $\Theta(n \log n)$ time slots are necessary for all processes to succeed with reasonable probability

Primality Testing

Problem: Given a natural number $n \geq 2$, is n a prime number?

Simple primality test:

1. **if** n is even **then**
2.   **return** $(n = 2)$
3. **for** $i := 1$ **to** $\lfloor \sqrt{n}/2 \rfloor$ **do**
4.   **if** $2i + 1$ divides n **then**
5.   **return false**
6. **return true**

- **Running time:** $O(\sqrt{n})$

A Better Algorithm?

- How can we test primality efficiently?
- We need a little bit of basic number theory…

Square Roots of Unity: In $\mathbb{Z}_p^*$, where p is a prime, the only solutions of the equation $x^2 \equiv 1 \pmod{p}$ are $x \equiv \pm 1 \pmod{p}$

- If we find an $x \not\equiv \pm 1 \pmod{n}$ such that $x^2 \equiv 1 \pmod{n}$, we can conclude that n is not a prime.

Algorithm Idea

Claim: Let $p > 2$ be a prime number such that $p - 1 = 2^s d$ for an integer $s \geq 1$ and some odd integer $d \geq 3$. Then for all $a \in \mathbb{Z}_p^*$,

$$a^d \equiv 1 \pmod{p} \;\; \textbf{or} \;\; a^{2^r d} \equiv -1 \pmod{p} \;\; \text{for some } 0 \leq r < s.$$

Proof:

- **Fermat's Little Theorem:** Given a prime number p,

$$\forall a \in \mathbb{Z}_p^*: \quad a^{p-1} \equiv 1 \pmod{p}$$

Primality Test

We have: If n is an odd prime and $n-1 = 2^s d$ for an integer $s \geq 1$ and an odd integer $d \geq 3$. Then for all $a \in \{1, \dots, n-1\}$,

$$a^d \equiv 1 \pmod{n} \ \textbf{or} \ a^{2^r d} \equiv -1 \pmod{n} \ \text{for some } 0 \leq r < s.$$

Idea: If we find an $a \in \{1, \dots, n-1\}$ such that

$$a^d \not\equiv 1 \pmod{n} \ \textbf{and} \ a^{2^r d} \not\equiv -1 \pmod{n} \ \text{for all } 0 \leq r < s,$$

we can conclude that n is not a prime.

- For every odd composite $n > 2$, at least $^3/_4$ of all possible a satisfy the above condition
- How can we find such a *witness* a efficiently?

Miller-Rabin Primality Test

- Given a natural number $n \geq 2$, is n a prime number?

Miller-Rabin Test:

1. **if** n is even **then return** $(n = 2)$
2. compute s, d such that $n - 1 = 2^s d$;
3. choose $a \in \{2, \dots, n-2\}$ uniformly at random;
4. $x \coloneqq a^d \bmod n$;
5. **if** $x = 1$ **or** $x = n - 1$ **then return probably prime;**
6. **for** $r \coloneqq 1$ **to** $s - 1$ **do**
7. $\quad x \coloneqq x^2 \bmod n$;
8. $\quad$ **if** $x = n - 1$ **then return probably prime;**
9. **return composite;**

Analysis

Theorem:

- If n is prime, the Miller-Rabin test always returns **true**.
- If n is composite, the Miller-Rabin test returns **false** with probability at least $^3/_4$.

Proof:

- If n is prime, the test works for all values of a
- If n is composite, we need to pick a good witness a

Corollary: If the Miller-Rabin test is repeated k times, it fails to detect a composite number n with probability at most 4^{-k}.

Running Time

Cost of Modular Arithmetic:

- Representation of a number $x \in \mathbb{Z}_n$: $O(\log n)$ bits
- Cost of adding two numbers $x + y \bmod n$:
- Cost of multiplying two numbers $x \cdot y \bmod n$:
  - It's like multiplying degree $O(\log n)$ polynomials
    → use FFT to compute $z = x \cdot y$

Running Time

Cost of exponentiation $x^d \bmod n$:

- Can be done using $O(\log d)$ multiplications
- Base-2 representation of d: $d = \sum_{i=0}^{\lfloor \log d \rfloor} d_i 2^i$
- **Fast exponentiation:**

```
y := 1;
for i := ⌊log d⌋ to 0 do
    y := y^2 mod n;
    if d_i = 1 then y := y · x mod n;
return y;
```

- **Example:** $d = 22 = 10110_2$

Running Time

Theorem: One iteration of the Miller-Rabin test can be implemented with running time $O(\log^2 n \cdot \log\log n \cdot \log\log\log n)$.

1. **if** n is even **then return** $(n = 2)$
2. compute s, d such that $n - 1 = 2^s d$;
3. choose $a \in \{2, \dots, n-2\}$ uniformly at random;
4. $x \coloneqq a^d \bmod n$;
5. **if** $x = 1$ **or** $x = n - 1$ **then return probably prime;**
6. **for** $r \coloneqq 1$ **to** $s - 1$ **do**
7. $\quad x \coloneqq x^2 \bmod n$;
8. $\quad$ **if** $x = n - 1$ **then return probably prime;**
9. **return composite;**

Deterministic Primality Test

- If a conjecture called the generalized Riemann hypothesis (GRH) is true, the Miller-Rabin test can be turned into a polynomial-time, deterministic algorithm

  → It is then sufficient to try all $a \in \{1, \dots, O(\log^2 n)\}$

- It has long not been proven whether a deterministic, polynomial-time algorithm exists

- In 2002, Agrawal, Kayal, and Saxena gave an $\tilde{O}(\log^{12} n)$-time deterministic algorithm
  - Has been improved to $\tilde{O}(\log^6 n)$

- In practice, the randomized Miller-Rabin test is still the fastest algorithm

Chapter 7
Randomization

Algorithm Theory
WS 2018/19

Fabian Kuhn

Randomization

Randomized Algorithm:

- An algorithm that uses (or can use) random coin flips in order to make decisions

We will see: randomization can be a powerful tool to

- Make algorithms faster
- Make algorithms simpler
- Make the analysis simpler
  - Sometimes it's also the opposite…
- Allow to solve problems (efficiently) that cannot be solved (efficiently) without randomization
  - True in some computational models (e.g., for distributed algorithms)
  - Not clear in the standard sequential model

Randomized Quicksort

Quicksort:

S	v

$S_\ell < v$	v	$S_r > v$

function Quick (S: sequence): sequence;

{returns the sorted sequence S}

begin

if $\#S \leq 1$ then **return** S

else { choose pivot element v in S;

partition S into S_ℓ with elements $< v$,

and S_r with elements $> v$

return | Quick(S_ℓ) | v | Quick(S_r) |

end;

Randomized Quicksort Analysis

Randomized Quicksort: pick uniform random element as pivot

Running Time of sorting n elements:

- Let's just count the number of comparisons
- In the partitioning step, all $n-1$ non-pivot elements have to be compared to the pivot

- **Number of comparisons:**

$$\boldsymbol{n-1\ +\ \#\text{comparisons in recursive calls}}$$

- **If rank of pivot is r:**
  recursive calls with $\boldsymbol{r-1}$ and $\boldsymbol{n-r}$ elements

Law of Total Expectation

- Given a random variable X and
- a set of events $A_1, \dots, A_k$ that partition Ω
  - E.g., for a second random variable Y, we could have

$$A_i := \{\omega \in \Omega : Y(\omega) = i\}$$

Law of Total Expectation

$$\mathbb{E}[X] = \sum_{i=1}^{k} \mathbb{P}(A_i) \cdot \mathbb{E}[X \mid A_i] = \sum_{y} \mathbb{P}(Y = y) \cdot \mathbb{E}[X \mid Y = y]$$

Example:

- X: outcome of rolling a die
- $A_0 = \{X \text{ is even}\}, \quad A_1 = \{X \text{ is odd}\}$

Randomized Quicksort Analysis

Random variables:

- C: total number of comparisons (for a given array of length n)
- R: rank of first pivot
- C_ℓ, C_r: number of comparisons for the 2 recursive calls

$$\mathbb{E}[C] = n - 1 + \mathbb{E}[C_\ell] + \mathbb{E}[C_r]$$

Law of Total Expectation:

$$\mathbb{E}[C] = \sum_{r=1}^{n} \mathbb{P}(R = r) \cdot \mathbb{E}[C|R = r]$$

$$= \sum_{r=1}^{n} \mathbb{P}(R = r) \cdot (n - 1 + \mathbb{E}[C_\ell|R = r] + \mathbb{E}[C_r|R = r])$$

Randomized Quicksort Analysis

We have seen that:

$$\mathbb{E}[C] = \sum_{r=1}^{n} \mathbb{P}(R = r) \cdot (n - 1 + \mathbb{E}[C_\ell | R = r] + \mathbb{E}[C_r | R = r])$$

Define:

- $\boldsymbol{T(n)}$: expected number of comparisons when sorting n elements

$$\mathbb{E}[C] = T(n)$$
$$\mathbb{E}[C_\ell | R = r] = T(r - 1)$$
$$\mathbb{E}[C_r | R = r] = T(n - r)$$

Recursion:

$$\boldsymbol{T(n) = \sum_{r=1}^{n} \frac{1}{n} \cdot \big(n - 1 + T(r - 1) + T(n - r)\big)}$$
$$\boldsymbol{T(0) = T(1) = 0}$$

Randomized Quicksort Analysis

Theorem: The expected number of comparisons when sorting n elements using randomized quicksort is $T(n) \leq 2n \ln n$.

Proof:

$$T(n) = \sum_{r=1}^{n} \frac{1}{n} \cdot \big(n - 1 + T(r-1) + T(n-r)\big), \qquad T(0) = 0$$

Randomized Quicksort Analysis

Theorem: The expected number of comparisons when sorting n elements using randomized quicksort is $T(n) \leq 2n \ln n$.

Proof:

$$T(n) \leq n - 1 + \frac{4}{n} \cdot \int_1^n x \ln x \, dx$$

$$\int x \ln x \, dx = \frac{x^2 \ln x}{2} - \frac{x^2}{4}$$

Alternative Analysis

Array to sort: [7 , 3 , 1 , 10 , 14 , 8 , 12 , 9 , 4 , 6 , 5 , 15 , 2 , 13 , 11]

Viewing quicksort run as a tree:

Comparisons

- Comparisons are only between pivot and non-pivot elements
- Every element can only be the pivot once:
  → every 2 elements can only be compared once!

- W.l.o.g., assume that the elements to sort are $1,2,\dots,n$
- Elements i and j are compared if and only if either i or j is a pivot before any element h: $i < h < j$ is chosen as pivot
  - i.e., iff i is an ancestor of j or j is an ancestor of i

$$\mathbb{P}(\text{comparison betw. } i \text{ and } j) = \frac{2}{j-i+1}$$

Counting Comparisons

Random variable for every pair of elements (i, j):

$$\boldsymbol{X_{ij}} = \begin{cases} 1, & \text{if there is a comparison between } i \text{ and } j \\ 0, & \text{otherwise} \end{cases}$$

Number of comparisons: $\boldsymbol{X}$

$$X = \sum_{i<j} X_{ij}$$

- What is $\mathbb{E}[X]$?

Randomized Quicksort Analysis

Theorem: The expected number of comparisons when sorting n elements using randomized quicksort is $T(n) \leq 2n \ln n$.

Proof:

- **Linearity of expectation:**
  For all random variables $X_1, \dots, X_n$ and all $a_1, \dots, a_n \in \mathbb{R}$,

$$\mathbb{E}\left[\sum_{i}^{n} a_i X_i\right] = \sum_{i}^{n} a_i \mathbb{E}[X_i].$$

Randomized Quicksort Analysis

Theorem: The expected number of comparisons when sorting n elements using randomized quicksort is $T(n) \leq 2n \ln n$.

Proof:

$$\mathbb{E}[X] = 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{j-i+1} = 2 \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{1}{k}$$

Quicksort: High Probability Bound

- We have seen that the number of comparisons of randomized quicksort is $O(n \log n)$ in expectation.

- Can we also show that the number of comparisons is $O(n \log n)$ with high probability?

- **Recall:**

  On each recursion level, each pivot is compared once with each other element that is still in the same "part"

Counting Number of Comparisons

- We looked at 2 ways to count the number of comparisons
  - recursive characterization of the expected number
  - number of different pairs of values that are compared

Let's consider yet another way:

- Each comparison is between a pivot and a non-pivot
- How many times is a specific array element x compared as a non-pivot?

Value x is compared as a non-pivot to a pivot once in every recursion level until one of the following two conditions apply:

1. x is chosen as a pivot
2. x is alone

Successful Recursion Level

- Consider a specific recursion level ℓ
- Assume that at the beginning of recursion level ℓ, element x is in a sub-array of length K_ℓ that still needs to be sorted.
- If x has been chosen as a pivot before level ℓ, we set $K_\ell := 1$

Definition: We say that recursion level ℓ is successful for element x iff the following is true:

$$K_{\ell+1} = 1 \quad \text{or} \quad K_{\ell+1} \leq \frac{2}{3} \cdot K_\ell$$

Successful Recursion Level

Lemma: For every recursion level ℓ and every array element x, it holds that level ℓ is successful for x with probability at least $1/3$, independently of what happens in other recursion levels.

Proof:

Number of Successful Recursion Levels

Lemma: If among the first ℓ recursion levels, at least $\log_{3/2}(n)$ are successful for element x, we have $K_\ell = 1$.

Proof:

Chernoff Bounds

- Let $X_1, \dots, X_n$ be independent 0-1 random variables and define $p_i := \mathbb{P}(X_i = 1)$.
- Consider the random variable $X = \sum_{i=1}^{n} X_i$
- We have $\mu := \mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} p_i$

Chernoff Bound (Lower Tail):

$$\forall \boldsymbol{\delta} > \mathbf{0}: \ \mathbb{P}(\boldsymbol{X} < (\mathbf{1} - \boldsymbol{\delta})\boldsymbol{\mu}) < \boldsymbol{e}^{-\boldsymbol{\delta}^2 \boldsymbol{\mu}/2}$$

Chernoff Bound (Upper Tail):

$$\forall \boldsymbol{\delta} > \mathbf{0}: \ \mathbb{P}(\boldsymbol{X} > (\mathbf{1} + \boldsymbol{\delta})\boldsymbol{\mu}) < \left(\frac{\boldsymbol{e}^{\boldsymbol{\delta}}}{(\mathbf{1} + \boldsymbol{\delta})^{\mathbf{1}+\boldsymbol{\delta}}}\right)^{\boldsymbol{\mu}} < \boldsymbol{e}^{-\boldsymbol{\delta}^2 \boldsymbol{\mu}/3}$$

holds for $\boldsymbol{\delta} \leq \mathbf{1}$

Chernoff Bounds, Example

Assume that a fair coin is flipped n times. What is the probability to have

1. less than $n/3$ heads?

2. more than $0.51n$ tails?

3. less than ${}^{n}/_{2} - \sqrt{c \cdot \ n \ln n}$ tails?

Proof of Chernoff Bound

- Independent Bernoulli random variables $X_1, X_2, \dots, X_n$
- $\mathbb{P}(\boldsymbol{X_i = 1}) \geq \boldsymbol{p_i}, \boldsymbol{X} := \sum_{i=1}^{n} \boldsymbol{X_i}, \boldsymbol{\mu} := \sum_{i=1}^{n} \boldsymbol{p_i} \geq \mathbb{E}[\boldsymbol{X}]$

Chernoff Lower Tail: $\mathbb{P}(\boldsymbol{X} < (\boldsymbol{1} - \boldsymbol{\delta})\boldsymbol{\mu}) < \boldsymbol{e}^{-\boldsymbol{\delta}^2 \boldsymbol{\mu}/2}$

Recall

- Markov Inequality: Given non-negative rand. var. $Z \geq 0$

$$\forall t > 0\colon\ \mathbb{P}(Z > t) < \frac{\mathbb{E}[Z]}{t}$$

- Independent random variables Y, Z:

$$\mathbb{E}[Y \cdot Z] = \mathbb{E}[Y] \cdot \mathbb{E}[Z]$$

Proof of Chernoff Bound

- $\mathbb{P}(X_i = 1) \geq p_i, X \coloneqq \sum_{i=1}^{n} X_i, \mu \coloneqq \sum_{i=1}^{n} p_i \geq \mathbb{E}[X]$

Chernoff Lower Tail: $\mathbb{P}(\boldsymbol{X} < (\boldsymbol{1} - \boldsymbol{\delta})\boldsymbol{\mu}) < \boldsymbol{e}^{-\boldsymbol{\delta}^2\boldsymbol{\mu}/\boldsymbol{2}}$

Proof of Chernoff Bound

- $\mathbb{P}(X_i = 1) \geq p_i, X \coloneqq \sum_{i=1}^{n} X_i, \mu \coloneqq \sum_{i=1}^{n} p_i \geq \mathbb{E}[X]$

Chernoff Lower Tail: $\mathbb{P}(\boldsymbol{X} < (\mathbf{1} - \boldsymbol{\delta})\boldsymbol{\mu}) < \boldsymbol{e}^{-\boldsymbol{\delta}^2\boldsymbol{\mu}/\mathbf{2}}$

Proof of Chernoff Bound

- $\mathbb{P}(X_i = 1) \geq p_i, X \coloneqq \sum_{i=1}^{n} X_i, \mu \coloneqq \sum_{i=1}^{n} p_i \geq \mathbb{E}[X]$

Chernoff Lower Tail: $\mathbb{P}(\boldsymbol{X} < (\mathbf{1} - \boldsymbol{\delta})\boldsymbol{\mu}) < \boldsymbol{e}^{-\boldsymbol{\delta}^2\boldsymbol{\mu}/\mathbf{2}}$

Number of Comparisons for x

Lemma: For every array element x, with high probability, as a non-pivot, x is compared to a pivot at most $O(\log n)$ times.

Proof:

Chapter 7
Randomization

Algorithm Theory
WS 2018/19

Fabian Kuhn

Types of Randomized Algorithms

Las Vegas Algorithm:

- always a correct solution
- running time is a random variable

- **Example:** randomized quicksort, contention resolution

Monte Carlo Algorithm:

- probabilistic correctness guarantee (**m**ostly **c**orrect)
- fixed (deterministic) running time

- **Example:** primality test

Minimum Cut

Reminder: Given a graph $G = (V, E)$, a cut is a partition (A, B) of V such that $V = A \cup B$, $A \cap B = \emptyset$, $A, B \neq \emptyset$

Size of the cut $(\boldsymbol{A}, \boldsymbol{B})$: # of edges crossing the cut

- For weighted graphs, total edge weight crossing the cut

Goal: Find a cut of minimal size (i.e., of size $\lambda(G)$)

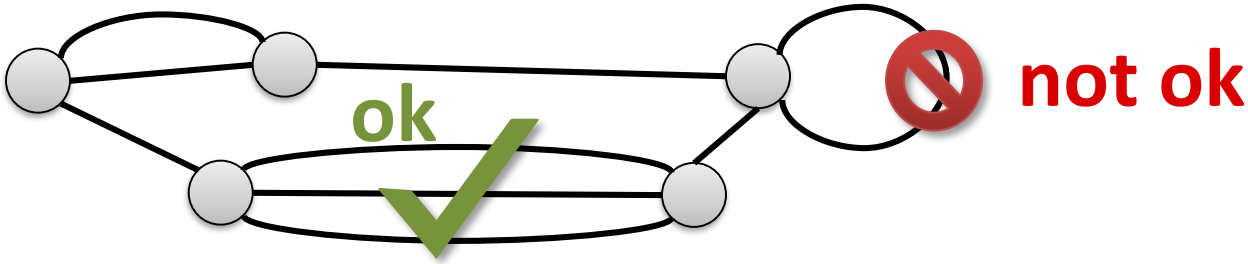
Maximum-flow based algorithm:

- Fix s, compute min s-t-cut for all $t \neq s$
- $O\big(m \cdot \lambda(G)\big) = O(mn)$ per s-t cut
- Gives an $\mathrm{O}\big(mn\lambda(G)\big) = O(mn^2)$-algorithm

Best-known deterministic algorithm: $O(mn + n^2 \log n)$

Edge Contractions

- In the following, we consider multi-graphs that can have multiple edges (but no self-loops)



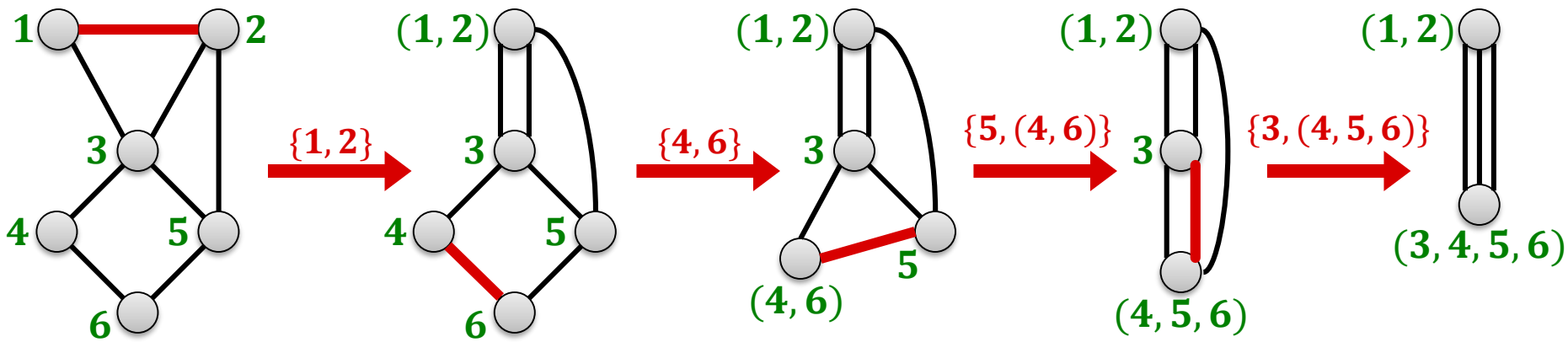
Contracting edge $\{u, v\}$:

- Replace nodes u, v by new node w
- For all edges $\{u, x\}$ and $\{v, x\}$, add an edge $\{w, x\}$
- Remove self-loops created at node w

Properties of Edge Contractions

Nodes:

- After contracting $\{u, v\}$, the new node represents u and v
- After a series of contractions, each node represents a subset of the original nodes



Cuts:

- Assume in the contracted graph, w represents nodes $S_w \subset V$
- The edges of a node w in a contracted graph are in a one-to-one correspondence with the edges crossing the cut $(S_w, V \setminus S_w)$

Randomized Contraction Algorithm

Algorithm:

while there are > 2 nodes **do**

contract a uniformly random edge

return cut induced by the last two remaining nodes

(cut defined by the original node sets represented by the last 2 nodes)

Theorem: The random contraction algorithm returns a minimum cut with probability at least $1/O(n^2)$.

- We will show this next.

Theorem: The random contraction algorithm can be implemented in time $O(n^2)$.

- There are $n-2$ contractions, each can be done in time $O(n)$.
- We will see this later.

Contractions and Cuts

Lemma: If two original nodes $u, v \in V$ are merged into the same node of the contracted graph, there is a path connecting u and v in the original graph s.t. all edges on the path are contracted.

Proof:

- Contracting an edge $\{x, y\}$ merges the node sets represented by x and y and does not change any of the other node sets.
- The claim the follows by induction on the number of edge contractions.

Contractions and Cuts

Lemma: During the contraction algorithm, the edge connectivity (i.e., the size of the min. cut) cannot get smaller.

Proof:

- All cuts in a (partially) contracted graph correspond to cuts of the same size in the original graph G as follows:
  - For a node u of the contracted graph, let S_u be the set of original nodes that have been merged into u (the nodes that u represents)
  - Consider a cut (A, B) of the contracted graph
  - (A', B') with

$$A' := \bigcup_{u \in A} S_u\,, \qquad B' := \bigcup_{v \in B} S_v$$

    is a cut of G.
  - The edges crossing cut (A, B) are in one-to-one correspondence with the edges crossing cut (A', B').

Contraction and Cuts

Lemma: The contraction algorithm outputs a cut (A, B) of the input graph G if and only if it never contracts an edge crossing (A, B).

Proof:

1. If an edge crossing (A, B) is contracted, a pair of nodes $u \in A$, $v \in V$ is merged into the same node and the algorithm outputs a cut different from (A, B).

2. If no edge of (A, B) is contracted, no two nodes $u \in A$, $v \in B$ end up in the same contracted node because every path connecting u and v in G contains some edge crossing (A, B)

   In the end there are only 2 sets → output is (A, B)

Getting The Min Cut

Theorem: The probability that the algorithm outputs a minimum cut is at least $2/n(n-1)$.

To prove the theorem, we need the following claim:

Claim: If the minimum cut size of a multigraph G (no self-loops) is k, G has at least $kn/2$ edges.

Proof:

- Min cut has size $k \Longrightarrow$ all nodes have degree $\geq k$
  - A node v of degree $< k$ gives a cut $(\{v\}, V \setminus \{v\})$ of size $< k$
- Number of edges $m = {}^1/_2 \cdot \sum_v \deg(v)$

Getting The Min Cut

Theorem: The probability that the algorithm outputs a minimum cut is at least $2/n(n-1)$.

Proof:

- Consider a fixed min cut (A, B), assume (A, B) has size k
- The algorithm outputs (A, B) iff none of the k edges crossing (A, B) gets contracted.
- Before contraction i, there are $n+1-i$ nodes
  → and thus $\geq (n+1-i)k/2$ edges
- If no edge crossing (A, B) is contracted before, the probability to contract an edge crossing (A, B) in step i is at most

$$\frac{k}{\frac{(n+1-i)k}{2}} = \frac{2}{n+1-i}.$$

Getting The Min Cut

Theorem: The probability that the algorithm outputs a minimum cut is at least $2/n(n-1)$.

Proof:

- If no edge crossing (A, B) is contracted before, the probability to contract an edge crossing (A, B) in step i is at most ${}^{2}/_{n+1-i}$.
- Event $\mathcal{E}_i$: edge contracted in step i is **not** crossing (A, B)

Getting The Min Cut

Theorem: The probability that the algorithm outputs a minimum cut is at least $2/n(n-1)$.

Proof:

- $\mathbb{P}(\mathcal{E}_{i+1} | \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_i) \geq 1 - {}^{2}/_{n-i} = \frac{n-i-2}{n-i}$
- No edge crossing (A, B) contracted: event $\mathcal{E} = \bigcap_{i=1}^{n-2} \mathcal{E}_i$

Randomized Min Cut Algorithm

Theorem: If the contraction algorithm is repeated $O(n^2 \log n)$ times, one of the $O(n^2 \log n)$ instances returns a min. cut w.h.p.

Proof:

- Probability to not get a minimum cut in $c \cdot \binom{n}{2} \cdot \ln n$ iterations:

$$\left(1 - \frac{1}{\binom{n}{2}}\right)^{c \cdot \binom{n}{2} \cdot \ln n} < e^{-c \ln n} = \frac{1}{n^c}$$

Corollary: The contraction algorithm allows to compute a minimum cut in $O(n^4 \log n)$ time w.h.p.

- It remains to show that each instance can be implemented in $O(n^2)$ time.

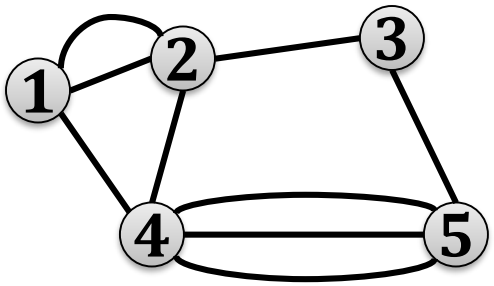
Implementing Edge Contractions

Edge Contraction:

- Given: multigraph with n nodes
  - assume that set of nodes is $\{1, \dots, n\}$
- Goal: contract edge $\{u, v\}$

Data Structure

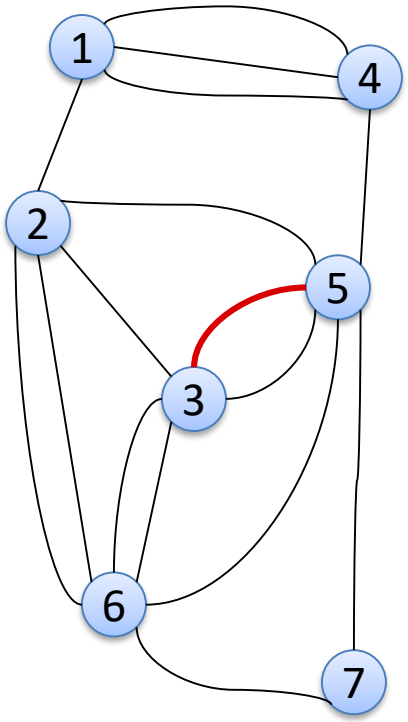
- We can use either adjacency lists or an adjacency matrix
- Entry in row i and column j: #edges between nodes i and j
- Example:



$$A = \begin{pmatrix} 0 & 2 & 0 & 1 & 0 \\ 2 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 3 \\ 0 & 0 & 1 & 3 & 0 \end{pmatrix}$$

Contracting An Edge

Example: Contract one of the edges between 3 and 5

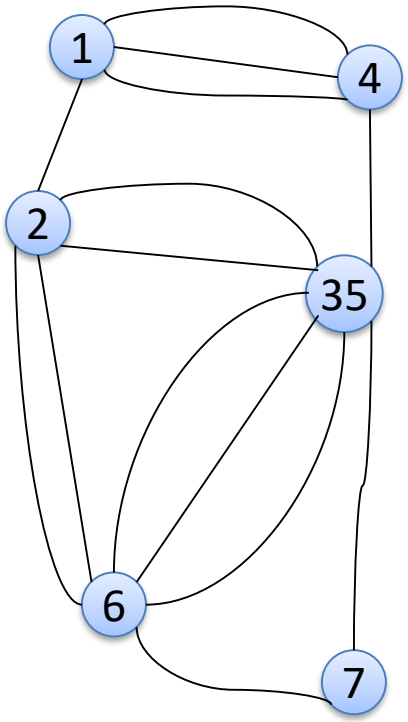


	1	2	3	4	5	6	7
1	0	1	0	3	0	0	0
2	1	0	1	0	1	2	0
3	0	1	0	0	2	2	0
4	3	0	0	0	1	0	0
5	0	1	2	1	0	1	1
6	0	2	2	0	1	0	1
7	0	0	0	0	1	1	0

{3,5}							

Contracting An Edge

Example: Contract one of the edges between 3 and 5

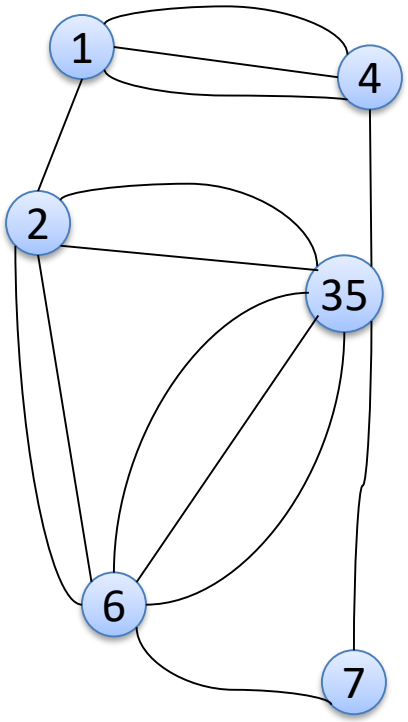


	1	2	3	4	5	6	7
1	0	1	0	3	0	0	0
2	1	0	1	0	1	2	0
3	0	1	0	0	2	2	0
4	3	0	0	0	1	0	0
5	0	1	2	1	0	1	1
6	0	2	2	0	1	0	1
7	0	0	0	0	1	1	0

{3,5}	0	2		1		3	1

Contracting An Edge

Example: Contract one of the edges between 3 and 5



	1	2	35	4		6	7
1	0	1	0	3		0	0
2	1	0	2	0		2	0
35	0	2	0	1		3	1
4	3	0	1	0		0	0
6	0	2	3	0		0	1
7	0	0	1	0		1	0

{3,5}	0	2		1		3	1

Contracting an Edge

Claim: Given the adjacency matrix of an n-node multigraph and an edge $\{u, v\}$, one can contract the edge $\{u, v\}$ in time $O(n)$.

- Row/column of combined node $\{u, v\}$ is sum of rows/columns of u and v
- Row/column of u can be replaced by new row/column of combined node $\{u, v\}$
- Swap row/column of v with last row/column in order to have the new $(n-1)$-node multigraph as a contiguous $(n-1) \times (n-1)$ submatrix

Finding a Random Edge

- We need to contract a uniformly random edge
- How to find a uniformly random edge in a multigraph?
  - Finding a random non-zero entry (with the right probability) in an adjacency matrix costs $O(n^2)$.

Idea for more efficient algorithm:

- First choose a random node u
  - with probability proportional to the degree (#edges) of u
- Pick a random edge of u
  - only need to look at one row → time $O(n)$

Choose a Random Node

Edge Sampling:

1. Choose a node $u \in V$ with probability

$$\frac{\deg(u)}{\sum_{v \in V} \deg(v)} = \frac{\deg(u)}{2m}$$

2. Choose a uniformly random edge of u

Choose a Random Node

- We need to choose a random node u with probability $\frac{\deg(u)}{2m}$
- Keep track of the number of edges m and maintain an array with the degrees of all the nodes
  - Can be done with essentially no extra cost when doing edge contractions

Choose a random node:

```
degsum = 0;
for all nodes u ∈ V:
    with probability deg(u)/(2m−degsum):
        pick node u; terminate
    else
        degsum += deg(u)
```

Randomized Min Cut Algorithm

Theorem: If the contraction algorithm is repeated $O(n^2 \log n)$ times, one of the $O(n^2 \log n)$ instances returns a min. cut w.h.p.

Corollary: The contraction algorithm allows to compute a minimum cut in $O(n^4 \log n)$ time w.h.p.

- One instance consists of $n - 2$ edge contractions
- Each edge contraction can be carried out in time $O(n)$
  - Actually: $O(\text{current \#nodes})$
- Time per instance of the contraction algorithm: $O(n^2)$

Can We Do Better?

- Time $O(n^4 \log n)$ is not very spectacular, a simple max flow based implementation has time $O(n^4)$.

However, we will see that the contraction algorithm is nevertheless very interesting because:

1. The algorithm can be improved to beat every known deterministic algorithm.

1. It allows to obtain strong statements about the distribution of cuts in graphs.

Better Randomized Algorithm

Recall:

- Consider a fixed min cut (A, B), assume (A, B) has size k
- The algorithm outputs (A, B) iff none of the k edges crossing (A, B) gets contracted.
- Throughout the algorithm, the edge connectivity is at least k and therefore each node has degree $\geq k$
- Before contraction i, there are $n + 1 - i$ nodes and thus at least $(n + 1 - i)k/2$ edges
- If no edge crossing (A, B) is contracted before, the probability to contract an edge crossing (A, B) in step i is at most

$$\frac{k}{\frac{(n+1-i)k}{2}} = \frac{2}{n+1-i}.$$

Improving the Contraction Algorithm

- For a specific min cut (A, B), if (A, B) survives the first i contractions,

$$\mathbb{P}(\text{edge crossing } (A, B) \text{ in contraction } i + 1) \leq \frac{2}{n - i}.$$

- **Observation:** The probability only gets large for large i

- **Idea:** The early steps are much safer than the late steps.

  Maybe we can repeat the late steps more often than the early ones.

Safe Contraction Phase

Lemma: A given min cut (A, B) of an n-node graph G survives the first $n - \left\lceil {}^{n}/_{\sqrt{2}} + 1 \right\rceil$ contractions, with probability $> {}^{1}/_{2}$.

Proof:

- Event $\mathcal{E}_i$: cut (A, B) survives contraction i
- Probability that (A, B) survives the first $n - t$ contractions:

Better Randomized Algorithm

Let's simplify a bit:

- Pretend that $n/\sqrt{2}$ is an integer (for all n we will need it).
- Assume that a given min cut survives the first $n - {}^{n}/_{\sqrt{2}}$ contractions with probability $\geq {}^{1}/_{2}$.

$\mathbf{contract}(\boldsymbol{G}, \boldsymbol{t})$**:**

- Starting with n-node graph G, perform $n - t$ edge contractions such that the new graph has t nodes.

$\mathbf{mincut}(\boldsymbol{G})$**:**

1. $X_1 \coloneqq \mathrm{mincut}\left(\mathrm{contract}\left(G, n/\sqrt{2}\right)\right)$;
2. $X_2 \coloneqq \mathrm{mincut}\left(\mathrm{contract}\left(G, n/\sqrt{2}\right)\right)$;
3. **return** $\min\{X_1, X_2\}$;

Success Probability

$\mathbf{mincut}(\boldsymbol{G})$**:**

1. $X_1 \coloneqq \text{mincut}\left(\text{contract}\left(G, n/\sqrt{2}\right)\right)$;
2. $X_2 \coloneqq \text{mincut}\left(\text{contract}\left(G, n/\sqrt{2}\right)\right)$;
3. **return** $\min\{X_1, X_2\}$;

$\boldsymbol{P}(\boldsymbol{n})$**:** probability that the above algorithm returns a min cut when applied to a graph with n nodes.

- Probability that X_1 is a min cut $\geq$

Recursion:

Success Probability

Theorem: The recursive randomized min cut algorithm returns a minimum cut with probability at least $1/\log_2 n$.

Proof (by induction on n):

$$P(n) = P\left(\frac{n}{\sqrt{2}}\right) - \frac{1}{4} \cdot P\left(\frac{n}{\sqrt{2}}\right)^2, \qquad P(2) = 1$$

Running Time

1. $X_1 \coloneqq \text{mincut}\left(\text{contract}\left(G, n/\sqrt{2}\right)\right)$;
2. $X_2 \coloneqq \text{mincut}\left(\text{contract}\left(G, n/\sqrt{2}\right)\right)$;
3. **return** $\min\{X_1, X_2\}$;

Recursion:

- $T(n)$: time to apply algorithm to n-node graphs
- Recursive calls: $2T\left({}^{n}/_{\sqrt{2}}\right)$
- Number of contractions to get to ${}^{n}/_{\sqrt{2}}$ nodes: $O(n)$

$$T(n) = 2T\left(\frac{n}{\sqrt{2}}\right) + O(n^2), \qquad T(2) = O(1)$$

Running Time

Theorem: The running time of the recursive, randomized min cut algorithm is $O(n^2 \log n)$.

Proof:

- Can be shown in the usual way, by induction on n

Remark:

- The running time is only by an $O(\log n)$-factor slower than the basic contraction algorithm.
- The success probability is exponentially better!

Chapter 7
Randomization

Algorithm Theory
WS 2018/19

Fabian Kuhn

Minimum Cut

Reminder: Given a graph $G = (V, E)$, a cut is a partition (A, B) of V such that $V = A \cup B$, $A \cap B = \emptyset$, $A, B \neq \emptyset$

Size of the cut $(\boldsymbol{A}, \boldsymbol{B})$: # of edges crossing the cut

- For weighted graphs, total edge weight crossing the cut

Goal: Find a cut of minimal size (i.e., of size $\lambda(G)$)

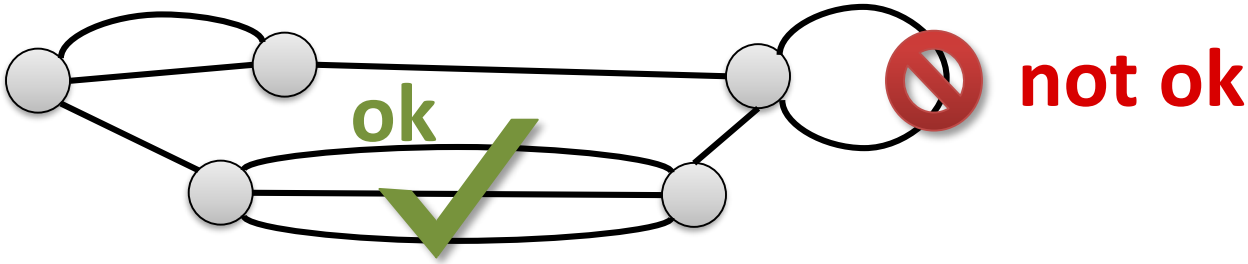
Maximum-flow based algorithm:

- Fix s, compute min s-t-cut for all $t \neq s$
- $O\big(m \cdot \lambda(G)\big) = O(mn)$ per s-t cut
- Gives an $\mathrm{O}\big(mn\lambda(G)\big) = O(mn^2)$-algorithm

Best-known deterministic algorithm: $O(mn + n^2 \log n)$

Edge Contractions

- In the following, we consider multi-graphs that can have multiple edges (but no self-loops)



Contracting edge $\{u, v\}$:

- Replace nodes u, v by new node w
- For all edges $\{u, x\}$ and $\{v, x\}$, add an edge $\{w, x\}$
- Remove self-loops created at node w

Randomized Contraction Algorithm

Algorithm:

while there are > 2 nodes **do**

contract a uniformly random edge

return cut induced by the last two remaining nodes

(cut defined by the original node sets represented by the last 2 nodes)

Theorem: The probability that the algorithm outputs a (specific) minimum cut (A, B) is at least $2/n(n-1) = 2/\binom{n}{2}$.

- Consider a fixed min cut (A, B), assume (A, B) has size k
- If no edge crossing (A, B) is contracted before, the probability to contract an edge crossing (A, B) in step i is at most

$$\frac{k}{\frac{(n+1-i)k}{2}} = \frac{2}{n+1-i}.$$

Getting The Min Cut

Theorem: The probability that the algorithm outputs a (specific) minimum cut (A, B) is at least $2/n(n-1) = 2/\binom{n}{2}$.

Proof:

- Event $\mathcal{E}_i$: edge contracted in step i is **not** crossing (A, B)

$$\mathbb{P}(\mathcal{E}_{i+1} | \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_i) \geq 1 - \frac{2}{n-i} = \frac{n-i-2}{n-i}$$

- Probability that cut (A, B) survives to the end:

$$\mathbb{P}(\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}) = \prod_{i=0}^{n-3} \mathbb{P}(\mathcal{E}_{i+1} | \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_i)$$

Number of Minimum Cuts

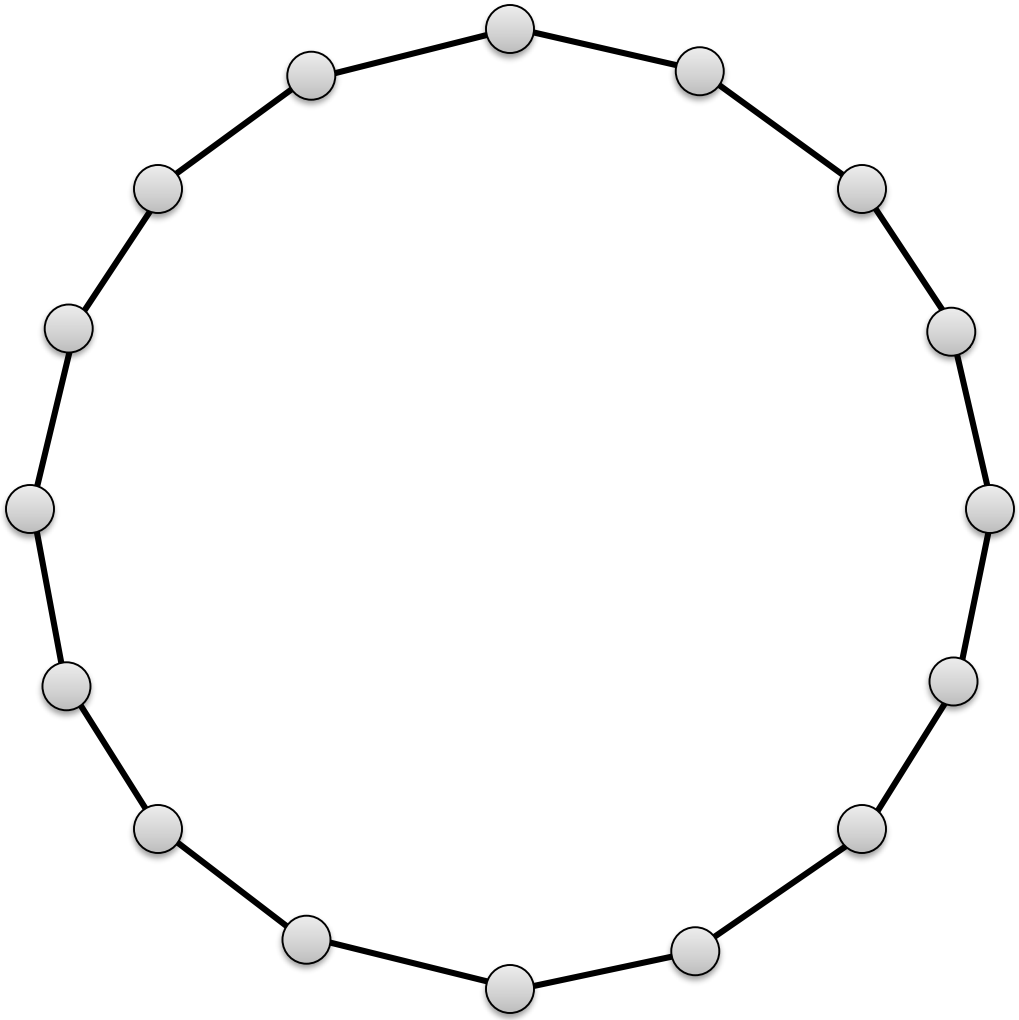
- Given a graph G, how many minimum cuts can there be?

- Or alternatively: If G has edge connectivity k, how many ways are there to remove k edges to disconnect G?

- Note that the total number of cuts is large.

Number of Minimum Cuts

Example: Ring with n nodes



- Minimum cut size: 2
- Every two edges induce a min cut
- Number of edge pairs:

$$\binom{n}{2}$$

- Are there graphs with more min cuts?

Number of Min Cuts

Theorem: The number of minimum cuts of a graph is at most $\binom{n}{2}$.

Proof:

- Assume there are s min cuts
- For $i \in \{1, \dots, s\}$, define event $\mathcal{C}_i$:

  $\mathcal{C}_i \coloneqq \{\text{basic contraction algorithm returns min cut } i\}$

- We know that for $i \in \{1, \dots, s\}$: $\mathbb{P}(\mathcal{C}_i) = 1/\binom{n}{2}$
- Events $\mathcal{C}_1, \dots, \mathcal{C}_s$ are disjoint:

$$\mathbb{P}\left(\bigcup_{i=1}^{s} \mathcal{C}_i\right) = \sum_{i=1}^{s} \mathbb{P}(\mathcal{C}_i) = \frac{s}{\binom{n}{2}}$$

Counting Larger Cuts

- In the following, assume that min cut has size $\lambda = \lambda(G)$
- How many cuts of size $\leq k = \alpha \cdot \lambda$ can a graph have?
- Consider a specific cut (A, B) of size $\leq k$
- As before, during the contraction algorithm:
  - min cut size $\geq \lambda$
  - number of edges $\geq \lambda \cdot \#\text{nodes}/2$
  - cut (A, B) remains as long as none of its edges gets contracted
- Prob. that an edge crossing (A, B) is chosen in i^{th} contraction

$$= \frac{k}{\#\text{edges}} \leq \frac{2k}{\lambda \cdot \#\text{nodes}} = \frac{2\alpha}{n - i + 1}$$

For simplicity, in the following, assume that 2α is an integer

Counting Larger Cuts

Lemma: If $2\alpha \in \mathbb{N}$, the probability that cut (A, B) of size $\alpha \cdot \lambda$ survives the first $n - 2\alpha$ edge contractions is at least

$$\frac{(2\alpha)!}{n(n-1) \cdot \ldots \cdot (n - 2\alpha + 1)} \geq \frac{2^{2\alpha-1}}{n^{2\alpha}}.$$

Proof:

- As before, event $\mathcal{E}_i$: cut (A, B) survives contraction i

Number of Cuts

Theorem: If $2\alpha \in \mathbb{N}$, the number of edge cuts of size at most $\alpha \cdot \lambda(G)$ in an n-node graph G is at most $n^{2\alpha}$.

Proof:

Remark: The bound also holds for general $\alpha \geq 1$.

Application: Resilience To Edge Failures

- Consider a connected network (graph) G with n nodes
- Assume that each link (edge) of G fails independently with probability p
- How large can p be such that the remaining graph is still connected with probability $1 - \varepsilon$?
- In the exercises, you will show that if $p \leq 1 - \frac{c \cdot \ln(n)}{n}$ for a sufficiently large constant $c > 0$, the graph remains connected with high probability.
  - Idea: graph connected $\Leftrightarrow$ there is an edge over each possible cut
  - Analyze the probability that a cut of a specific size survives
  - Do a union bound over all possible cut sizes and all cuts of that size

Chapter 8
Approximation Algorithms

Algorithm Theory
WS 2018/19

Fabian Kuhn

Approximation Algorithms

- Optimization appears everywhere in computer science
- We have seen many examples, e.g.:
  - scheduling jobs
  - traveling salesperson
  - maximum flow, maximum matching
  - minimum spanning tree
  - minimum vertex cover
  - …
- Many discrete optimization problems are NP-hard
- They are however still important and we need to solve them
- As algorithm designers, we prefer algorithms that produce solutions which are provably good, even if we can't compute an optimal solution.

Approximation Algorithms: Examples

We have already seen two approximation algorithms

- **Metric TSP:** If distances are positive and satisfy the triangle inequality, the greedy tour is only by a log-factor longer than an optimal tour

- **Maximum Matching and Vertex Cover:** A maximal matching gives solutions that are within a factor of 2 for both problems.

Approximation Ratio

An approximation algorithm is an algorithm that computes a solution for an optimization with an objective value that is provably within a bounded factor of the optimal objective value.

Formally:

- $\mathrm{OPT} \geq 0$: optimal objective value
  $\mathrm{ALG} \geq 0$: objective value achieved by the algorithm

- **Approximation Ratio α:**

$$\textbf{Minimization}: \ \alpha := \max_{\text{input instances}} \frac{\mathrm{ALG}}{\mathrm{OPT}}$$

$$\textbf{Maximization}: \ \alpha := \min_{\text{input instances}} \frac{\mathrm{ALG}}{\mathrm{OPT}}$$

Example: Load Balancing

We are given:

- m machines $M_1, \dots, M_m$
- n jobs, processing time of job i is t_i

Goal:

- Assign each job to a machine such that the makespan is minimized

  makespan: largest total processing time of any machine

The above load balancing problem is NP-hard and we therefore want to get a good approximation for the problem.

Greedy Algorithm

There is a simple greedy algorithm:

- Go through the jobs in an arbitrary order
- When considering job i, assign the job to the machine that currently has the smallest load.

Example: 3 machines, 12 jobs

3	4	2	3	1	6	4	4	3	2	1	5

Greedy Assignment:

M_1:	3	1	6	1	5
M_2:	4	4	3		
M_3:	2	3	4	2	

Optimal Assignment:

M_1:	3	4	2	3	1
M_2:	6	4	3		
M_3:	4	2	1	5	

Greedy Analysis

- We will show that greedy gives a 2-approximation
- To show this, we need to compare the solution of greedy with an optimal solution (that we can't compute)
- Lower bound on the optimal makespan T^*:

$$T^* \geq \frac{1}{m} \cdot \sum_{i=1}^{n} t_i$$

- Lower bound can be far from T^*:
  - m machines, m jobs of size 1, 1 job of size m

$$T^* = m, \qquad \frac{1}{m} \cdot \sum_{i=1}^{n} t_i = 2$$

Greedy Analysis

- We will show that greedy gives a 2-approximation
- To show this, we need to compare the solution of greedy with an optimal solution (that we can't compute)
- Lower bound on the optimal makespan T^*:

$$T^* \geq \frac{1}{m} \cdot \sum_{i=1}^{n} t_i$$

- Second lower bound on optimal makespan T^*:

$$T^* \geq \max_{1 \leq i \leq n} t_i$$

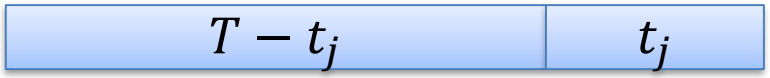
Greedy Analysis

Theorem: The greedy algorithm has approximation ratio ≤ 2, i.e., for the makespan T of the greedy solution, we have $T \leq 2T^*$.

Proof:

- For machine k, let T_k be the time used by machine k
- Consider some machine M_i for which $T_i = T$
- Assume that job j is the last one schedule on M_i:

$\boldsymbol{M_i}$:



- When job j is scheduled, M_i has the minimum load

Greedy Analysis

Theorem: The greedy algorithm has approximation ratio ≤ 2, i.e., for the makespan T of the greedy solution, we have $T \leq 2T^*$.

Proof:

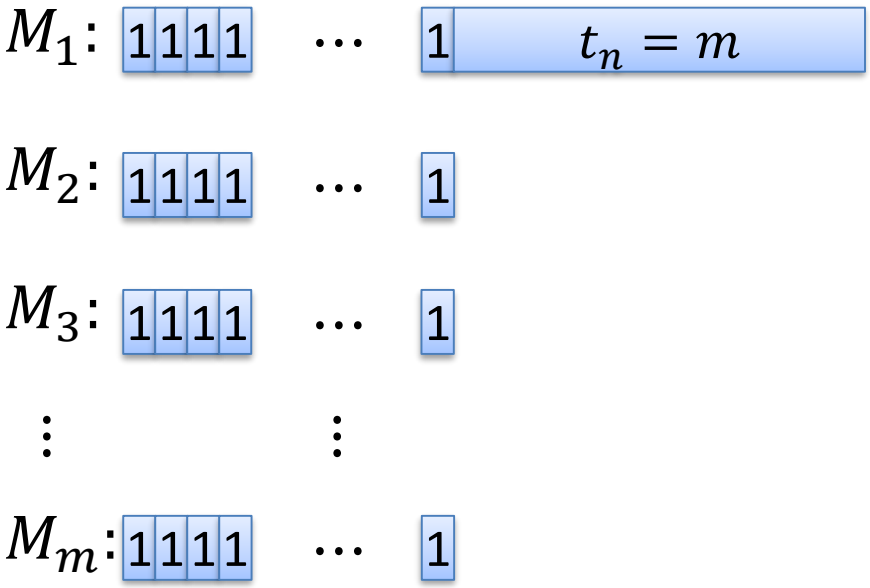
- For all machines M_k: load $T_k \geq T - t_j$

Can We Do Better?

The analysis of the greedy algorithm is almost tight:

- Example with $n = m(m-1) + 1$ jobs
- Jobs $1, \dots, n-1 = m(m-1)$ have $t_i = 1$, job n has $t_n = m$

Greedy Schedule:



Improving Greedy

Bad case for the greedy algorithm:
One large job in the end can destroy everything

Idea: assign large jobs first

Modified Greedy Algorithm:

1. Sort jobs by decreasing length s.t. $t_1 \geq t_2 \geq \cdots \geq t_n$
2. Apply the greedy algorithm as before (in the sorted order)

Lemma: If $n > m$: $T^* \geq t_m + t_{m+1} \geq 2t_{m+1}$

Proof:

- Two of the first $m + 1$ jobs need to be scheduled on the same machine
- Jobs m and $m + 1$ are the shortest of these jobs

Analysis of the Modified Greedy Alg.

Theorem: The modified algorithm has approximation ratio $\leq {}^{3}/_{2}$.

Proof:

- We show that $T \leq {}^{3}/_{2} \cdot T^*$
- As before, we consider the machine M_i with $T_i = T$
- Job j (of length t_j) is the last one scheduled on machine M_i
- If j is the only job on M_i, we have $T = T^*$
- Otherwise, we have $j \geq m + 1$
  - The first m jobs are assigned to m distinct machines

Set Cover

Input:

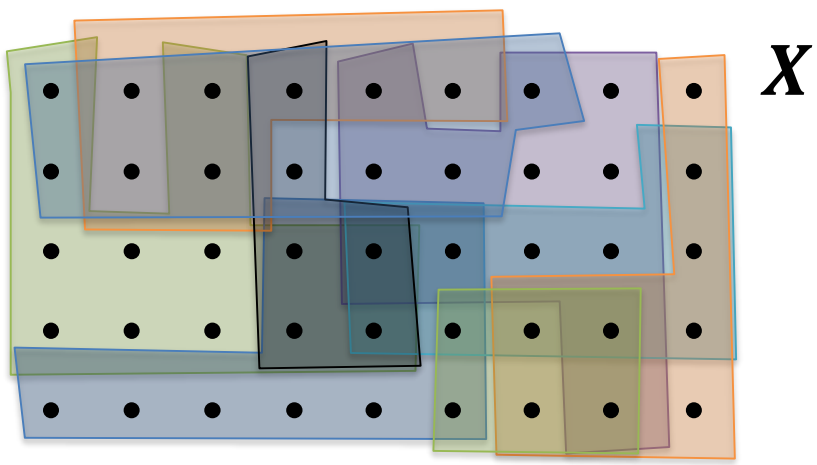
- A set of elements X and a collection $\mathcal{S}$ of subsets X, i.e., $\mathcal{S} \subseteq 2^X$
  - such that $\bigcup_{S \in \mathcal{S}} S = X$

Set Cover:

- A set cover $\mathcal{C}$ of $(X, \mathcal{S})$ is a subset of the sets $\mathcal{S}$ which covers X:

$$\bigcup_{S \in \mathcal{C}} S = X$$

Example:



Minimum (Weighted) Set Cover

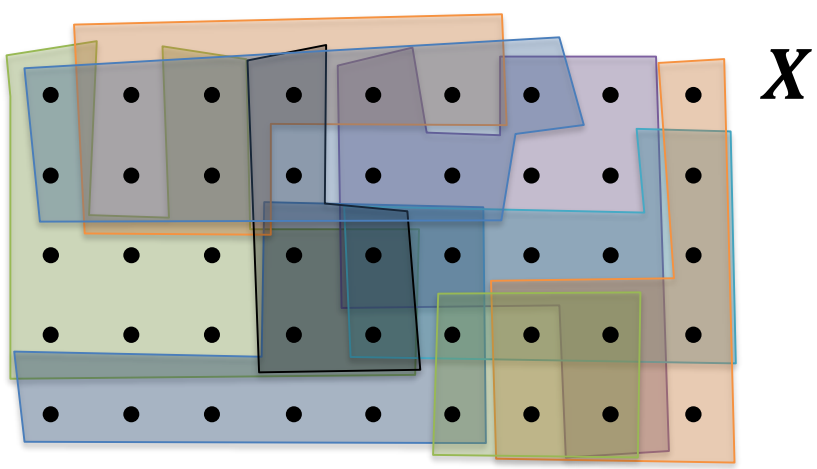
Minimum Set Cover:

- **Goal:** Find a set cover $\mathcal{C}$ of smallest possible size
  - i.e., over X with as few sets as possible

Minimum Weighted Set Cover:

- Each set $S \in \mathcal{S}$ has a weight $w_S > 0$
- **Goal:** Find a set cover $\mathcal{C}$ of minimum weight

Example:

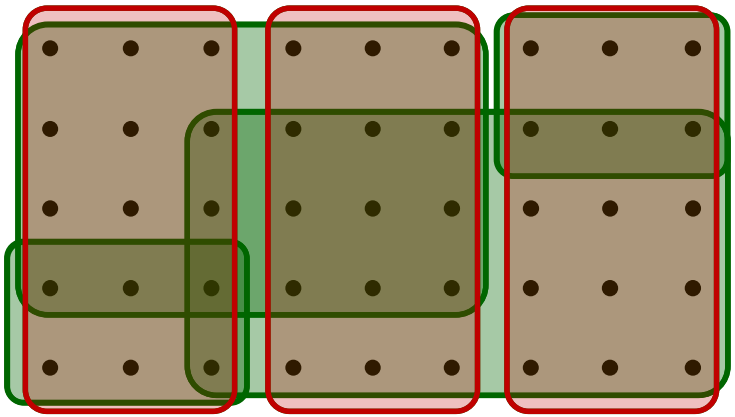


Minimum Set Cover: Greedy Algorithm

Greedy Set Cover Algorithm:

- Start with $\mathcal{C} = \emptyset$
- In each step, add set $S \in \mathcal{S} \setminus \mathcal{C}$ to $\mathcal{C}$ s.t. S covers as many uncovered elements as possible

Example:



Weighted Set Cover: Greedy Algorithm

Greedy Weighted Set Cover Algorithm:

- Start with $\mathcal{C} = \emptyset$
- In each step, add set $S \in \mathcal{S} \setminus \mathcal{C}$ with the best weight per newly covered element ratio (set with best efficiency):

$$S = \arg\min_{S \in \mathcal{S} \setminus \mathcal{C}} \frac{w_S}{\left|S \setminus \bigcup_{T \in \mathcal{C}} T\right|}$$

Analysis of Greedy Algorithm:

- Assign a price $p(x)$ to each element $x \in X$:
  The efficiency of the set when covering the element
- If covering x with set S, if partial cover is $\mathcal{C}$ before adding S:

$$p(e) = \frac{w_S}{\left|S \setminus \bigcup_{T \in \mathcal{C}} T\right|}$$

Weighted Set Cover: Greedy Algorithm

Example:

- Universe $X = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- Sets $\mathcal{S} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$

$$
\begin{aligned}
S_1 &= \{1, 2, 3, 4, 5\}, & w_{S_1} &= 4 \\
S_2 &= \{2, 6, 7\}, & w_{S_2} &= 1 \\
S_3 &= \{1, 6, 7, 8, 9\}, & w_{S_3} &= 4 \\
S_4 &= \{2, 4, 7, 9, 10\}, & w_{S_4} &= 6 \\
S_5 &= \{1, 3, 5, 6, 7, 8, 9, 10\}, & w_{S_5} &= 9 \\
S_6 &= \{9, 10\}, & w_{S_6} &= 3
\end{aligned}
$$

Weighted Set Cover: Greedy Algorithm

Lemma: Consider a set $S = \{x_1, x_2, \dots, x_k\} \in \mathcal{S}$ be a set and assume that the elements are covered in the order $x_1, x_2, \dots, x_k$ by the greedy algorithm (ties broken arbitrarily).

Then, the price of element x_i is at most $p(x_i) \leq \frac{w_S}{k-i+1}$

Weighted Set Cover: Greedy Algorithm

Lemma: Consider a set $S = \{x_1, x_2, \dots, x_k\} \in \mathcal{S}$ be a set and assume that the elements are covered in the order $x_1, x_2, \dots, x_k$ by the greedy algorithm (ties broken arbitrarily).

Then, the price of element x_i is at most $p(x_i) \leq \frac{w_S}{k-i+1}$

Corollary: The total price of a set $S \in \mathcal{S}$ of size $|S| = k$ is

$$\sum_{x \in S} p(x) \leq w_S \cdot H_k, \qquad \text{where } H_k = \sum_{i=1}^{k} \frac{1}{i} \leq 1 + \ln k$$

Weighted Set Cover: Greedy Algorithm

Corollary: The total price of a set $S \in \mathcal{S}$ of size $|S| = k$ is

$$\sum_{x \in S} p(x) \leq w_S \cdot H_k, \qquad \text{where } H_k = \sum_{i=1}^{k} \frac{1}{i} \leq 1 + \ln k$$

Theorem: The approximation ratio of the greedy minimum (weighted) set cover algorithm is at most $\boldsymbol{H_s \leq 1 + \ln s}$, where s is the cardinality of the largest set ($s = \max_{S \in \mathcal{S}} |S|$).

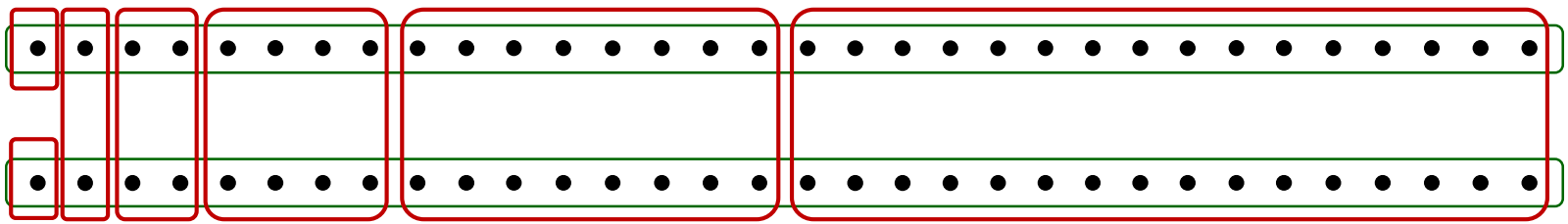
Set Cover Greedy Algorithm

Can we improve this analysis?

No! Even for the unweighted minimum set cover problem, the approximation ratio of the greedy algorithm is $\geq \big(1 - o(1)\big) \cdot \ln s$.

- if s is the size of the largest set... (s can be linear in n)

Let's show that the approximation ratio is at least $\Omega(\log n)$...



$\mathbf{OPT = 2}$

$\mathbf{GREEDY \geq \log_2 n}$

Set Cover: Better Algorithm?

An approximation ratio of $\ln n$ seems not spectacular...

Can we improve the approximation ratio?

No, unfortunately not, unless $\mathrm{P} \approx \mathrm{NP}$

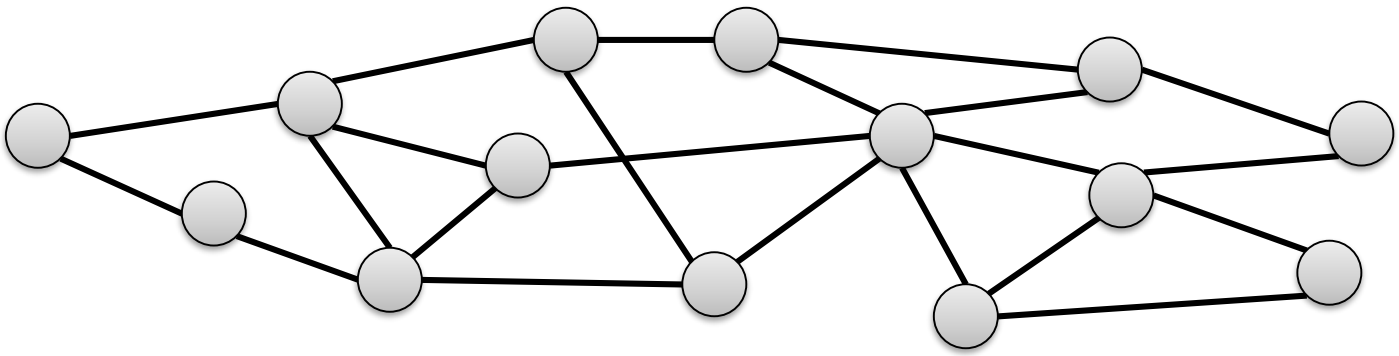
Feige showed that unless NP has deterministic $n^{O(\log \log n)}$-time algorithms, minimum set cover cannot be approximated better than by a factor $\big(1 - o(1)\big) \cdot \ln n$ in polynomial time.

- Proof is based on the so-called PCP theorem
  - PCP theorem is one of the main (relatively) recent advancements in theoretical computer science and the major tool to prove approximation hardness lower bounds
  - Shows that every language in NP has certificates of polynomial length that can be checked by a randomized algorithm by only querying a constant number of bits (for any constant error probability)

Set Cover: Special Cases

Vertex Cover: set $S \subseteq V$ of nodes of a graph $G = (V, E)$ such that

$$\forall \{\boldsymbol{u}, \boldsymbol{v}\} \in \boldsymbol{E}, \qquad \{\boldsymbol{u}, \boldsymbol{v}\} \cap \boldsymbol{S} \neq \emptyset.$$



Minimum Vertex Cover:

- Find a vertex cover of minimum cardinality

Minimum Weighted Vertex Cover:

- Each node has a weight
- Find a vertex cover of minimum total weight

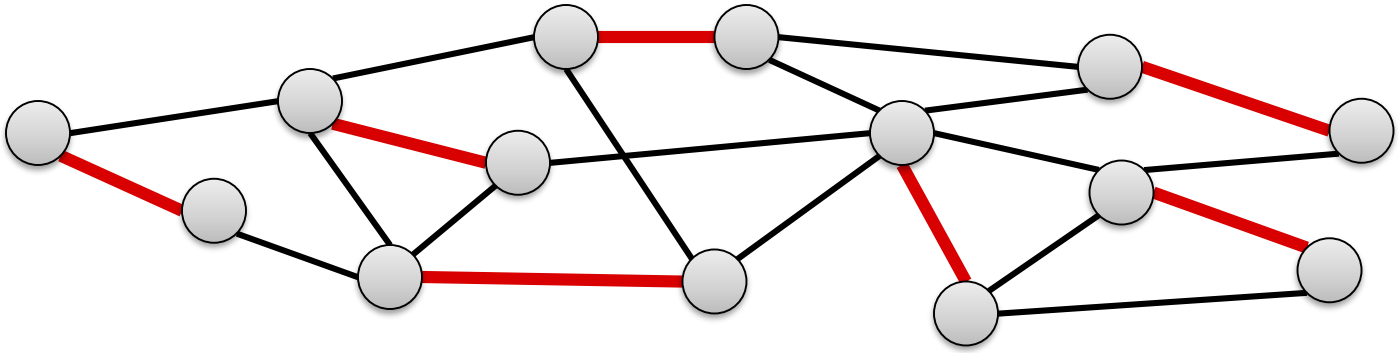
Vertex Cover vs Matching

Consider a matching M and a vertex cover S

Claim: $|M| \leq |S|$

Proof:

- At least one node of every edge $\{u, v\} \in M$ is in S
- Needs to be a different node for different edges from M



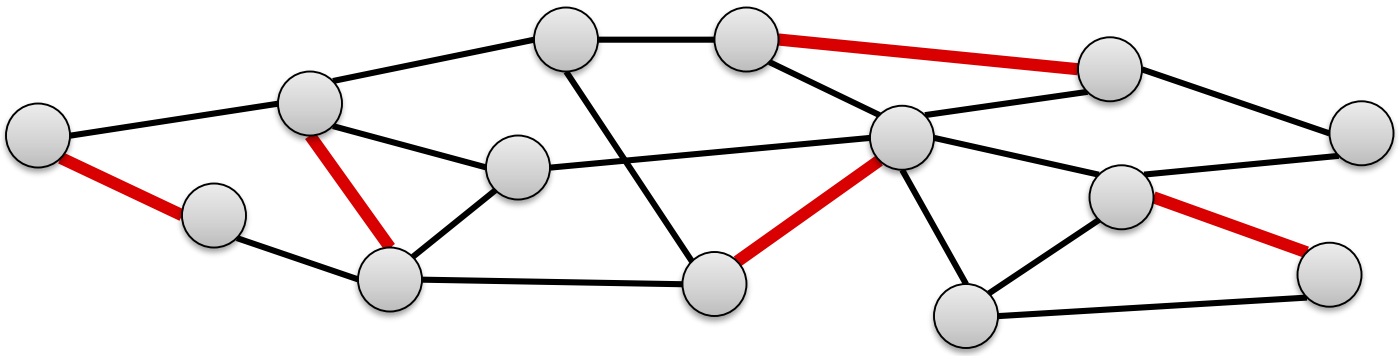
Vertex Cover vs Matching

Consider a matching M and a vertex cover S

Claim: If M is maximal and S is minimum, $|S| \leq 2|M|$

Proof:

- M is maximal: for every edge $\{u, v\} \in E$, either u or v (or both) are matched



- Every edge $e \in E$ is "covered" by at least one matching edge
- Thus, the set of the nodes of all matching edges gives a vertex cover S of size $|S| = 2|M|$.

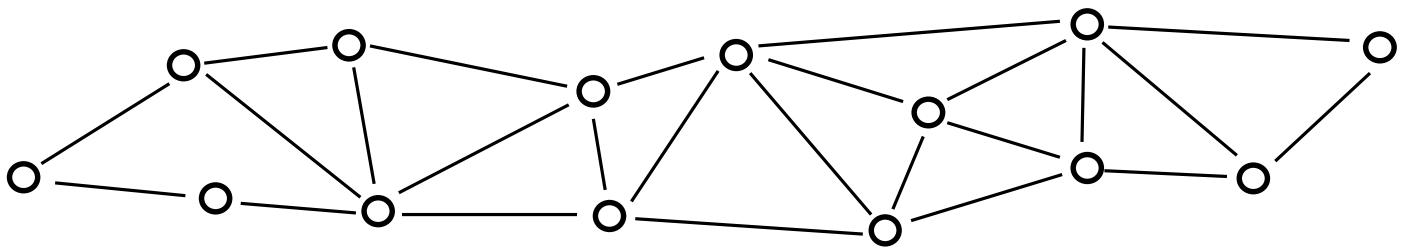
Maximal Matching Approximation

Theorem: The set of all matched nodes of a maximal matching M is a vertex cover of size at most twice the size of a min. vertex cover.

Set Cover: Special Cases

Dominating Set:

Given a graph $G = (V, E)$, a dominating set $S \subseteq V$ is a subset of the nodes V of G such that for all nodes $u \in V \setminus S$, there is a neighbor $v \in S$.



Minimum Hitting Set

Given: Set of elements X and collection of subsets $\mathcal{S} \subseteq 2^X$

- Sets cover X: $\bigcup_{S \in \mathcal{S}} S = X$

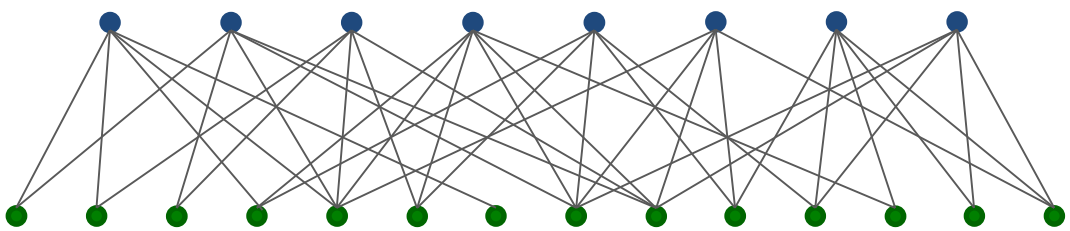
Goal: Find a min. cardinality subset $H \subseteq X$ of elements such that

$$\forall S \in \mathcal{S} : S \cap H \neq \emptyset$$

Problem is equivalent to min. set cover with roles of sets and elements interchanged

Sets

Elements



Chapter 8
Approximation Algorithms

Algorithm Theory
WS 2018/19

Fabian Kuhn

Approximation Ratio

An approximation algorithm is an algorithm that computes a solution for an optimization with an objective value that is provably within a bounded factor of the optimal objective value.

Formally:

- $\mathrm{OPT} \geq 0$: optimal objective value
  $\mathrm{ALG} \geq 0$: objective value achieved by the algorithm

- **Approximation Ratio α:**

$$\textbf{Minimization}: \ \alpha := \max_{\text{input instances}} \frac{\mathrm{ALG}}{\mathrm{OPT}}$$

$$\textbf{Maximization}: \ \alpha := \min_{\text{input instances}} \frac{\mathrm{ALG}}{\mathrm{OPT}}$$

Metric TSP

Input:

- Set V of n nodes (points, cities, locations, sites)
- Distance function $d: V \times V \to \mathbb{R}$, i.e., $d(u,v)$ is dist from u to v
- Distances define a metric on V:

$$d(u,v) = d(v,u) \geq 0, \qquad d(u,v) = 0 \Leftrightarrow u = v$$
$$\forall u,v,w \in V : d(u,v) \leq d(u,w) + d(w,v)$$

Solution:

- Ordering/permutation $v_1, v_2, \dots, v_n$ of the vertices
- Length of TSP path: $\sum_{i=1}^{n-1} d(v_i, v_{i+1})$
- Length of TSP tour: $d(v_1, v_n) + \sum_{i=1}^{n-1} d(v_i, v_{i+1})$

Goal:

- Minimize length of TSP path or TSP tour

Metric TSP

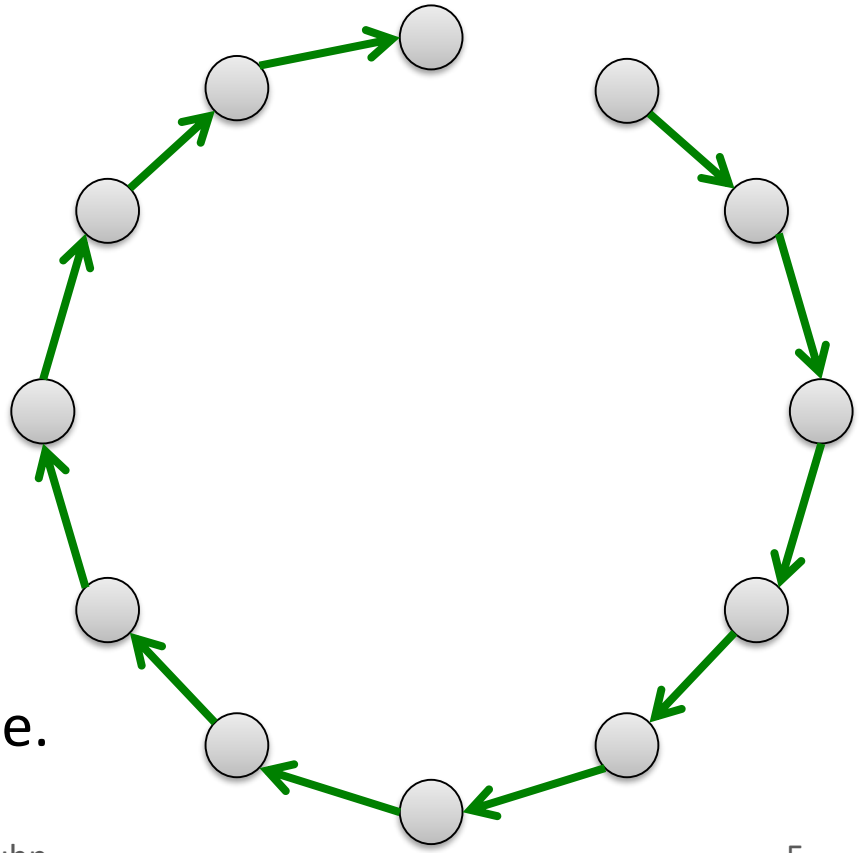
- The problem is NP-hard
- We have seen that the greedy algorithm (always going to the nearest unvisited node) gives an $O(\log n)$-approximation
- Can we get a constant approximation ratio?
- We will see that we can…

TSP and MST

Claim: The length of an optimal TSP path is lower bounded by the weight of a minimum spanning tree

Proof:

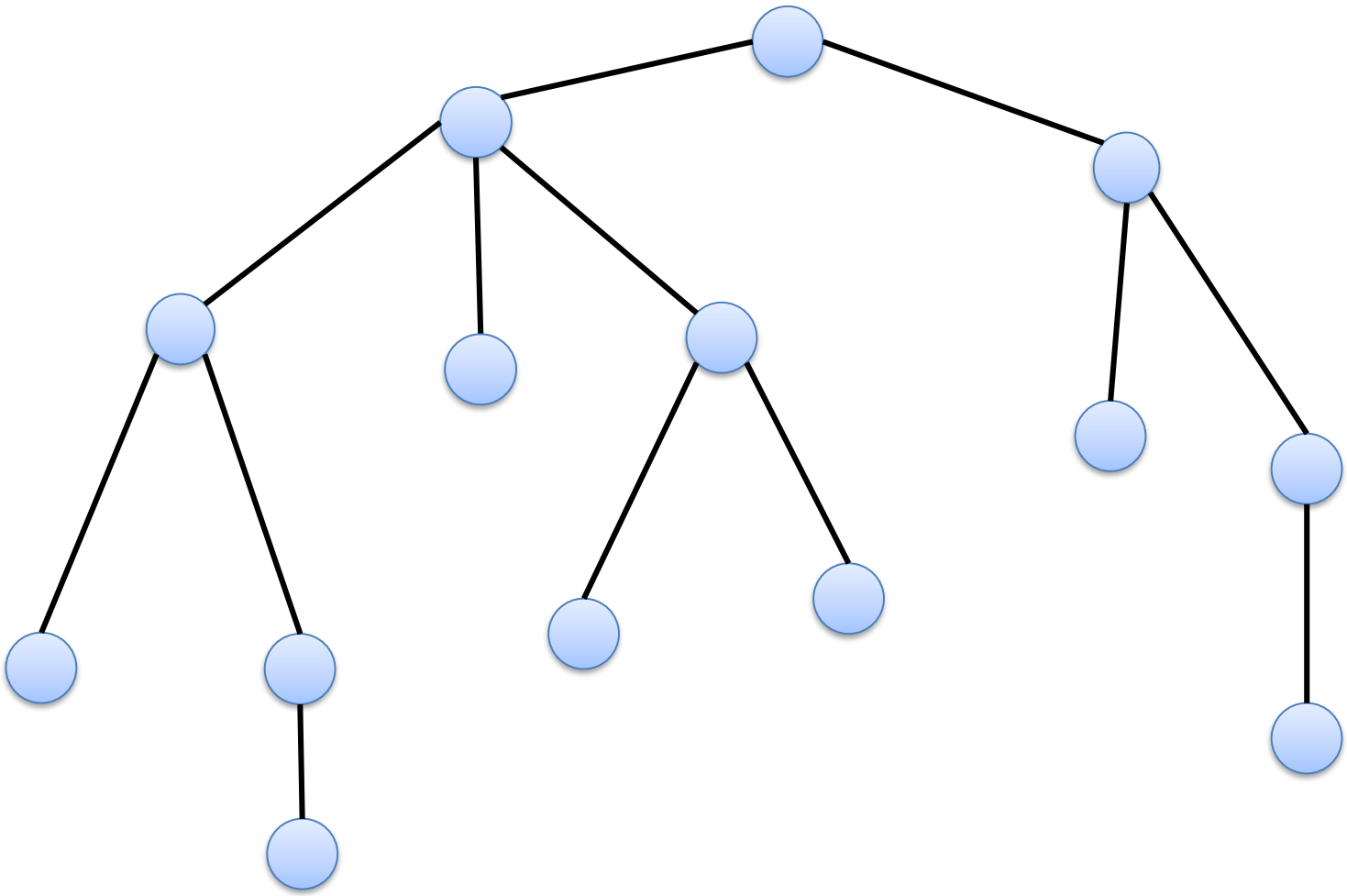
- A TSP path is a spanning tree, it's length is the weight of the tree



Corollary: Since an optimal TSP tour is longer than an optimal TSP path, the length of an optimal TSP tour is also lower bounded by the weight of a minimum spanning tree.

The MST Tour

Walk around the MST…

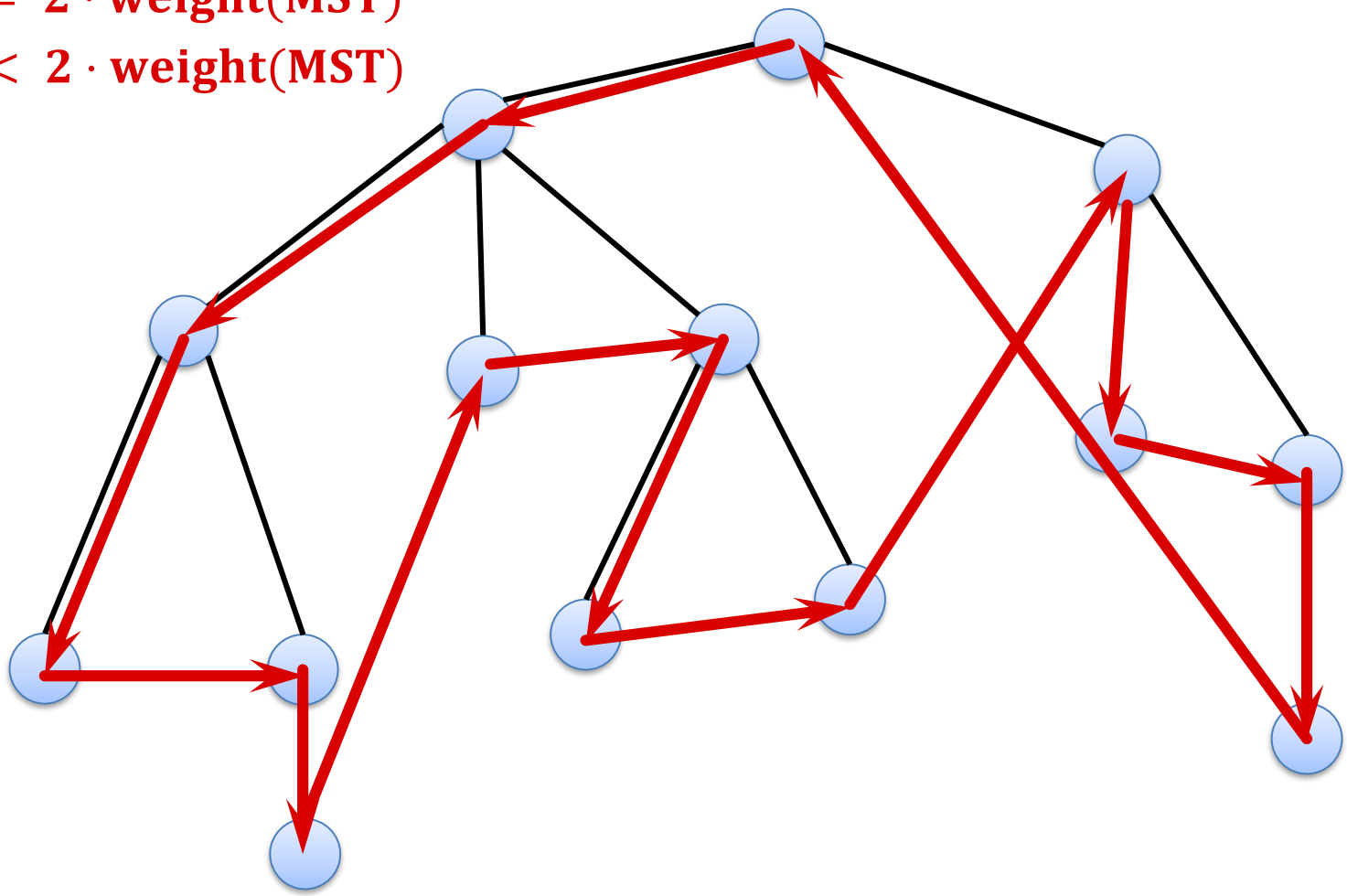


The MST Tour

Walk around the MST…

Cost (walk) $= \ 2 \cdot \mathbf{weight(MST)}$

Cost (tour) $< \ 2 \cdot \mathbf{weight(MST)}$



Approximation Ratio of MST Tour

Theorem: The MST TSP tour gives a 2-approximation for the metric TSP problem.

Proof:

- Triangle inequality → length of tour is at most $2 \cdot \mathrm{weight}(\mathrm{MST})$
- We have seen that $\mathrm{weight}(\mathrm{MST}) < \mathrm{opt.\,tour\,length}$

Can we do even better?

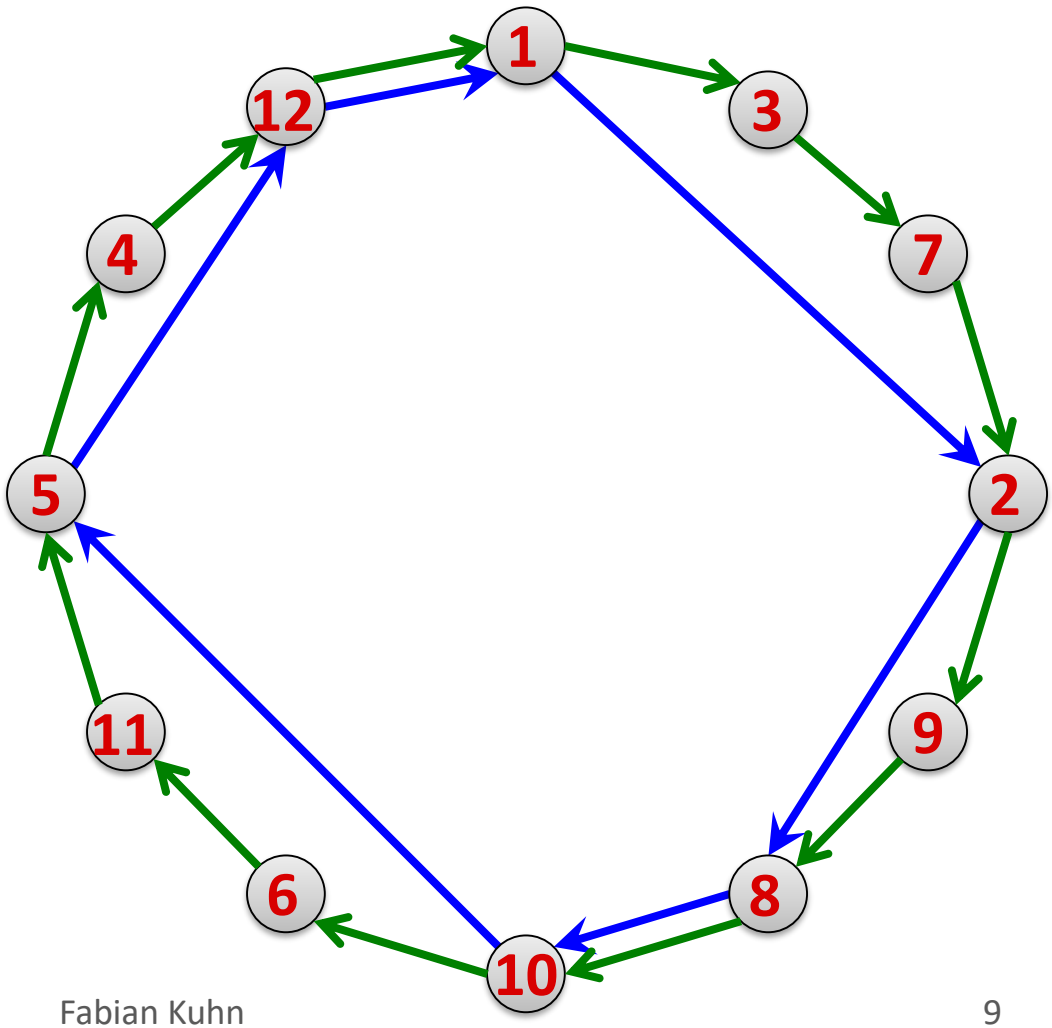
Metric TSP Subproblems

Claim: Given a metric (V, d) and (V', d) for $V' \subseteq V$, the optimal TSP path/tour of (V', d) is at most as large as the optimal TSP path/tour of (V, d).

Optimal TSP tour of nodes $1, 2, \ldots, 12$

Induced TSP tour for nodes $1, 2, 5, 8, 10, 12$

blue tour $\leq$ green tour



TSP and Matching

- Consider a metric TSP instance (V, d) with an even number of nodes $|V|$

- Recall that a perfect matching is a matching $M \subseteq V \times V$ such that every node of V is incident to an edge of M.

- Because $|V|$ is even and because in a metric TSP, there is an edge between any two nodes $u, v \in V$, any partition of V into $|V|/2$ pairs is a perfect matching.

- The weight of a matching M is the sum of the distances represented by all edges in M:

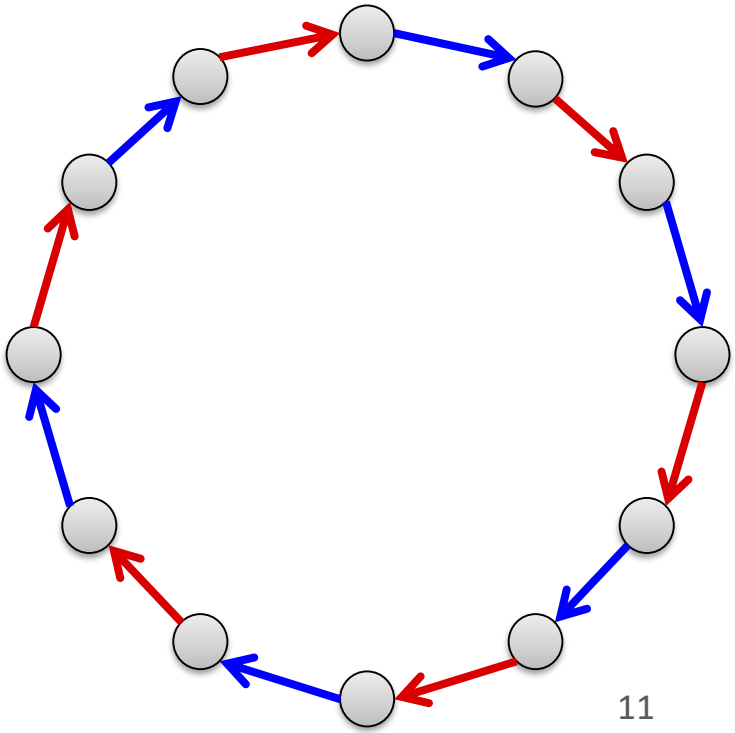
$$w(M) = \sum_{\{u,v\} \in M} d(u, v)$$

TSP and Matching

Lemma: Assume we are given a TSP instance (V, d) with an even number of nodes. The length of an optimal TSP tour of (V, d) is at least twice the weight of a minimum weight perfect matching of (V, d).

Proof:

- The edges of a TSP tour can be partitioned into 2 perfect matchings



Minimum Weight Perfect Matching

Claim: If $|V|$ is even, a minimum weight perfect matching of (V, d) can be computed in polynomial time

Proof Sketch:

- We have seen that a minimum weight perfect matching in a complete bipartite graph can be computed in polynomial time
- With a more complicated algorithm, also a minimum weight perfect matching in complete (non-bipartite) graphs can be computed in polynomial time
- The algorithm uses similar ideas as the bipartite weighted matching algorithm and it uses the Blossom algorithm as a subroutine

Algorithm Outline

Problem of MST algorithm:

- Every edge has to be visited twice

Goal:

- Get a graph on which every edge only has to be visited once (and where still the total edge weight is small compared to an optimal TSP tour)

Euler Tours:

- A tour that visits each edge of a graph exactly once is called an Euler tour
- An Euler tour in a (multi-)graph exists if and only if every node of the graph has even degree
- That's definitely not true for a tree, but can we modify our MST suitably?

Euler Tour

Theorem: A connected (multi-)graph G has an Euler tour if and only if every node of G has even degree.

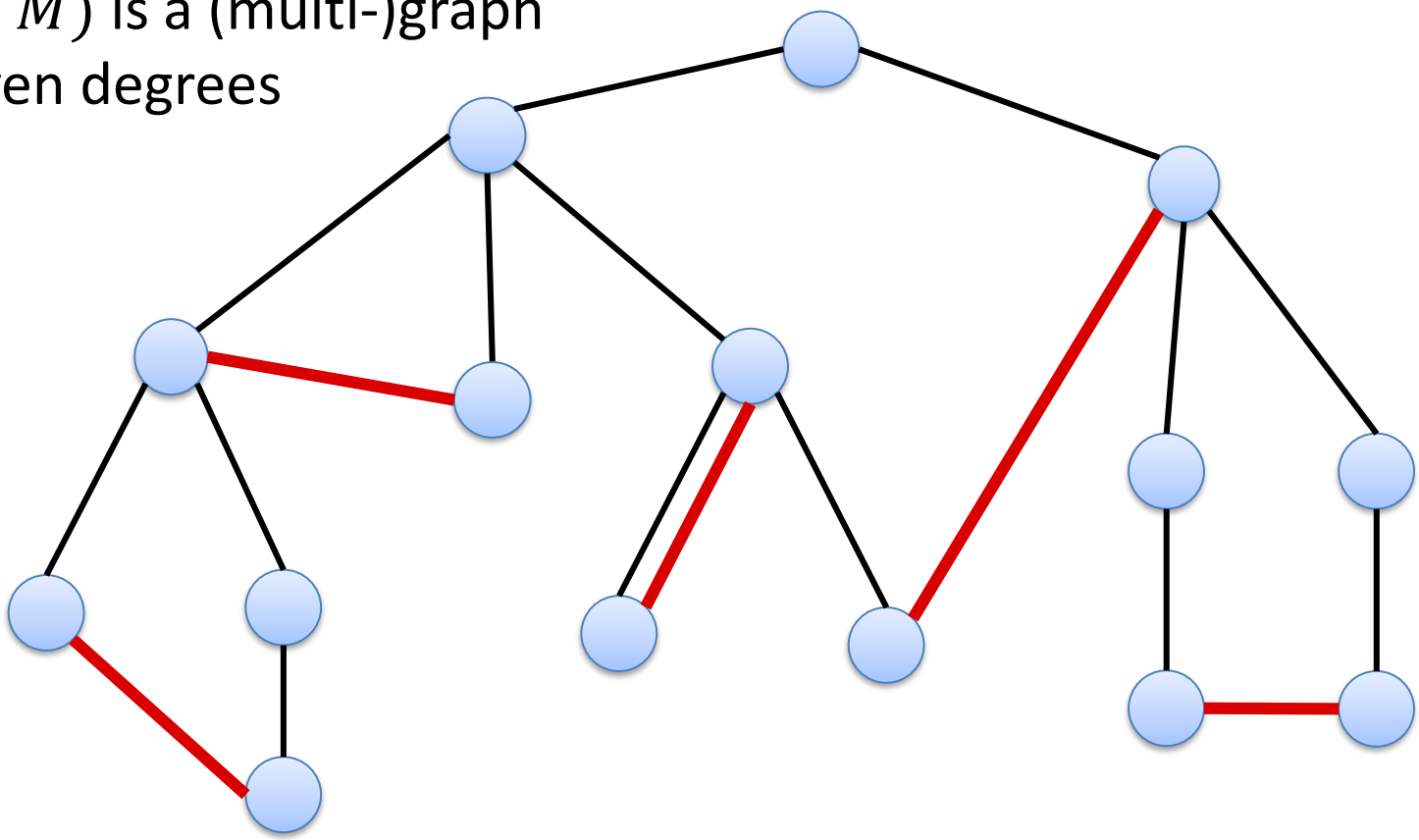
Proof:

- If G has an odd degree node, it clearly cannot have an Euler tour
- If G has only even degree nodes, a tour can be found recursively:

1. Start at some node
2. As long as possible, follow an unvisited edge
   - Gives a partial tour, the remaining graph still has even degree
3. Solve problem on remaining components recursively
4. Merge the obtained tours into one tour that visits all edges

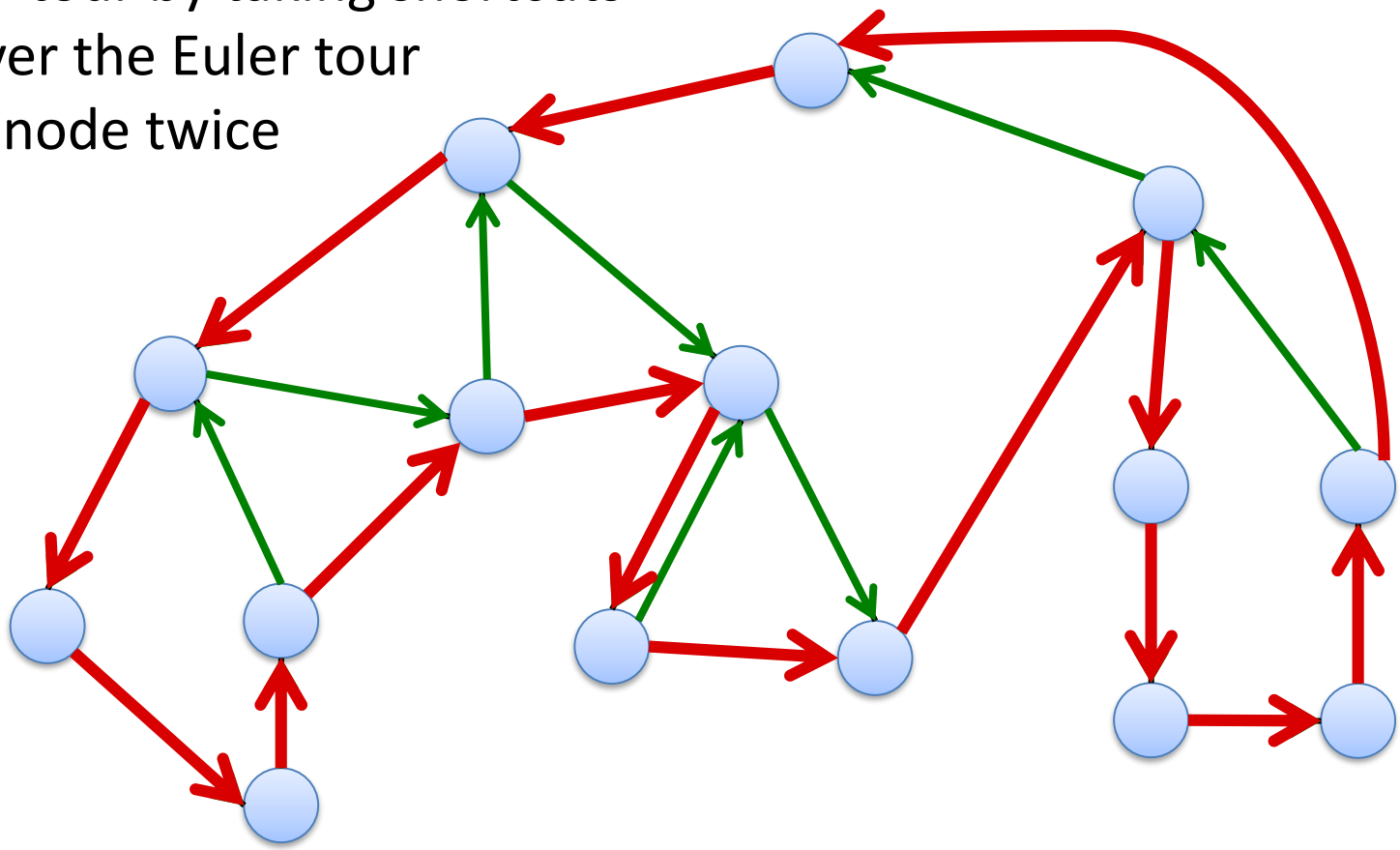
TSP Algorithm

1. Compute MST T
2. V_{odd}: nodes that have an odd degree in T ($|V_{\text{odd}}|$ is even)
3. Compute min weight perfect matching M of (V_{odd}, d)
4. $(V, T \cup M)$ is a (multi-)graph with even degrees



TSP Algorithm

5. Compute Euler tour on $(V, T \cup M)$
6. Total length of Euler tour $\leq \frac{3}{2} \cdot \mathbf{TSP_{OPT}}$
7. Get TSP tour by taking shortcuts wherever the Euler tour visits a node twice



TSP Algorithm

- The described algorithm is by Christofides

Theorem: The Christofides algorithm achieves an approximation ratio of at most $^3/_2$.

Proof:

- The length of the Euler tour is $\leq {}^3/_2 \cdot \mathrm{TSP}_{\mathrm{OPT}}$
- Because of the triangle inequality, taking shortcuts can only make the tour shorter

Knapsack

- n items $1, \dots, n$, each item has weight $w_i > 0$ and value $v_i > 0$
- Knapsack (bag) of capacity W

- Goal: pack items into knapsack such that total weight is at most W and total value is maximized:

$$\max \sum_{i \in S} v_i$$

$$\text{s.t.} \quad S \subseteq \{1, \dots, n\} \text{ and } \sum_{i \in S} w_i \leq W$$

- E.g.: jobs of length w_i and value v_i, server available for W time units, try to execute a set of jobs that maximizes the total value

Knapsack: Dynamic Programming Alg.

We have shown:

- If all item weights w_i are integers, using dynamic programming, the knapsack problem can be solved in time $O(nW)$
- If all values v_i are integers, there is another dynamic progr. algorithm that runs in time $O(n^2V)$, where V is the max. value.

Knapsack: Dynamic Programming Alg.

We have shown:

- If all item weights w_i are integers, using dynamic programming, the knapsack problem can be solved in time $O(nW)$
- If all values v_i are integers, there is another dynamic progr. algorithm that runs in time $O(n^2V)$, where V is the max. value.

Problems:

- If W and V are large, the algorithms are not polynomial in n
- If the values or weights are not integers, things are even worse (and in general, the algorithms cannot even be applied at all)

Idea:

- Can we adapt one of the algorithms to at least compute an approximate solution?

Approximation Algorithm

- The algorithm has a parameter $\varepsilon > 0$
- We assume that each item alone fits into the knapsack
- We define:

$$V := \max_{1 \le i \le n} v_i, \qquad \forall i: \widehat{v_i} := \left\lceil \frac{v_i n}{\varepsilon V} \right\rceil, \qquad \hat{V} := \max_{1 \le i \le n} \widehat{v_i}$$

- We solve the problem with integer values $\widehat{v_i}$ and weights w_i using dynamic programming in time $O(n^2 \cdot \hat{V})$
- If solution value $< V$, we take item with value V instead

Theorem: The described algorithm runs in time $O(n^3/\varepsilon)$.

Proof:

$$\hat{V} = \max_{1 \le i \le n} \widehat{v_i} = \max_{1 \le i \le n} \left\lceil \frac{v_i n}{\varepsilon V} \right\rceil = \left\lceil \frac{V n}{\varepsilon V} \right\rceil = \left\lceil \frac{n}{\varepsilon} \right\rceil$$

Approximation Algorithm

Theorem: The approximation algorithm computes a feasible solution with approximation ratio at least $1 - \varepsilon$.

Proof:

- Define the set of all feasible solutions (subsets of $[n]$)

$$\mathcal{S} := \left\{ S \subseteq \{1, \dots, n\} : \sum_{i \in S} w_i \leq W \right\}$$

- $v(S)$: value of solution S w.r.t. values $v_1, v_2, \dots$
  $\hat{v}(S)$: value of solution S w.r.t. values $\hat{v}_1, \hat{v}_2, \dots$
- S^*: an optimal solution w.r.t. values $v_1, v_2, \dots$
  $\hat{S}$: an optimal solution w.r.t. values $\hat{v}_1, \hat{v}_2, \dots$

- Weights are not changed at all, hence, $\hat{S}$ is a feasible solution

Approximation Algorithm

Theorem: The approximation algorithm computes a feasible solution with approximation ratio at least $1 - \varepsilon$.

Proof:

- We have

$$v(S^*) = \sum_{i \in S^*} v_i = \max_{S \in \mathcal{S}} \sum_{i \in S} v_i \,,$$

$$\hat{v}(\hat{S}) = \sum_{i \in \hat{S}} \hat{v}_i = \max_{S \in \mathcal{S}} \sum_{S \in \mathcal{S}} \hat{v}_i$$

- Because every item fits into the knapsack, we have

$$\forall i \in \{1, \dots, n\}\colon\ v_i \leq V \leq \sum_{j \in S^*} v_j$$

- Also: $\hat{v}_i = \left\lceil \frac{v_i n}{\varepsilon V} \right\rceil \implies v_i \leq \frac{\varepsilon V}{n} \cdot \hat{v}_i$, and $\hat{v}_i \leq \frac{v_i n}{\varepsilon V} + 1$

Approximation Algorithm

Theorem: The approximation algorithm computes a feasible solution with approximation ratio at least $1 - \varepsilon$.

Proof:

- We have

$$v(S^*) = \sum_{i \in S^*} v_i \leq \frac{\varepsilon V}{n} \cdot \sum_{i \in S^*} \widehat{v_i} \leq \frac{\varepsilon V}{n} \cdot \sum_{i \in \hat{S}} \widehat{v_i} \leq \frac{\varepsilon V}{n} \cdot \sum_{i \in \hat{S}} \left(1 + \frac{v_i n}{\varepsilon V}\right)$$

- Therefore

$$v(S^*) = \sum_{i \in S^*} v_i \leq \frac{\varepsilon V}{n} \cdot |\hat{S}| + \sum_{i \in \hat{S}} v_i \leq \varepsilon V + v(\hat{S})$$

- We have $v(S^*) \geq V$ and therefore

$$\boldsymbol{(1 - \varepsilon) \cdot v(S^*) \leq v(\widehat{S})}$$

Approximation Schemes

- For every parameter $\varepsilon > 0$, the knapsack algorithm computes a $(1+\varepsilon)$-approximation in time $O(n^3/\varepsilon)$.
- For every fixed ε, we therefore get a polynomial time approximation algorithm
- An algorithm that computes an $(1+\varepsilon)$-approximation for every $\varepsilon > 0$ is called an approximation scheme.
- If the running time is polynomial for every fixed ε, we say that the algorithm is a polynomial time approximation scheme (PTAS)
- If the running time is also polynomial in $1/\varepsilon$, the algorithm is a fully polynomial time approximation scheme (FPTAS)
- Thus, the described alg. is an FPTAS for the knapsack problem

Chapter 9
Online Algorithms

Algorithm Theory
WS 2018/19

Fabian Kuhn

Online Computations

- Sometimes, an algorithm has to start processing the input before the complete input is known

- For example, when storing data in a data structure, the sequence of operations on the data structure is not known

Online Algorithm: An algorithm that has to produce the output step-by-step when new parts of the input become available.

Offline Algorithm: An algorithm that has access to the whole input before computing the output.

- Some problems are inherently online
  - Especially when real-time requests have to be processed over a significant period of time

Competitive Ratio

- Let's again consider optimization problems
  - For simplicity, assume, we have a minimization problem

Optimal offline solution $\mathbf{OPT}(\boldsymbol{I})$:

- Best objective value that an offline algorithm can achieve for a given input sequence I

Online solution $\mathbf{ALG}(\boldsymbol{I})$:

- Objective value achieved by an online algorithm ALG on I

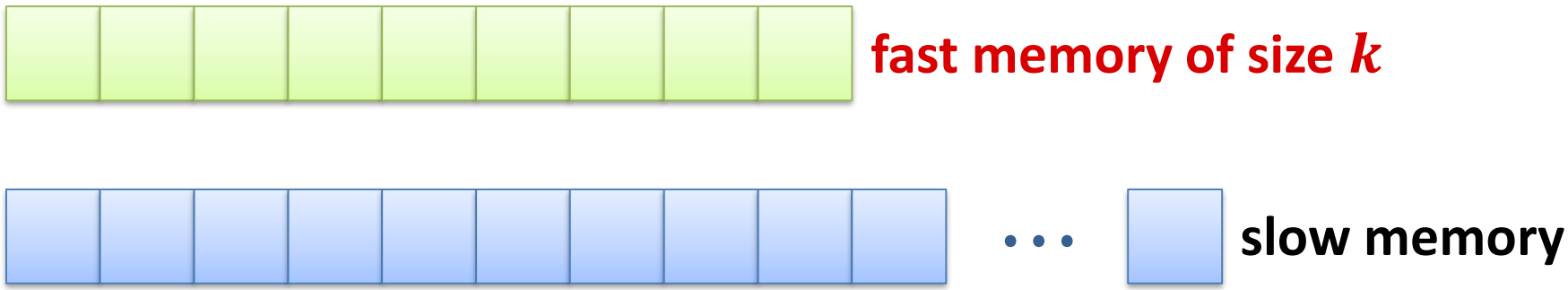
Competitive Ratio: An algorithm has competitive ratio $c \geq 1$ if

$$\mathbf{ALG}(\boldsymbol{I}) \leq \boldsymbol{c} \cdot \mathbf{OPT}(\boldsymbol{I}) + \boldsymbol{\alpha}.$$

- If $\alpha = 0$, we say that ALG is strictly c-competitive.

Paging Algorithm

Assume a simple memory hierarchy:



If a memory page has to be accessed:

- Page in fast memory (hit): take page from there
- Page not in fast memory (miss): leads to a page fault
- Page fault: the page is loaded into the fast memory and some page has to be evicted from the fast memory
- Paging algorithm: decides which page to evict
- Classical online problem: we don't know the future accesses

Paging Strategies

Least Recently Used (LRU):

- Replace the page that hasn't been used for the longest time

First In First Out (FIFO):

- Replace the page that has been in the fast memory longest

Last In First Out (LIFO):

- Replace the page most recently moved to fast memory

Least Frequently Used (LFU):

- Replace the page that has been used the least

Longest Forward Distance (LFD):

- Replace the page whose next request is latest (in the future)
- LFD is **not** an online strategy!

LFD is Optimal

Theorem: LFD (longest forward distance) is an optimal offline alg.

Proof:

- For contradiction, assume that LFD is not optimal
- Then there exists a finite input sequence σ on which LFD is not optimal (assume that the length of σ is $|\sigma| = n$)
- Let OPT be an optimal solution for σ such that
  - OPT processes requests $1, \dots, i$ in exactly the same way as LFD
  - OPT processes request $i + 1$ differently than LFD
  - Any other optimal strategy processes one of the first $i + 1$ requests differently than LFD
- Hence, OPT is the optimal solution that behaves in the same way as LFD for as long as possible → we have $i < n$
- Goal: Construct OPT′ that is identical with LFD for req. $1, \dots, i + 1$

LFD is Optimal

Theorem: LFD (longest forward distance) is an optimal offline alg.

Proof:

Case 1: Request $i+1$ does **not** lead to a page fault

- LFD does not change the content of the fast memory

- OPT behaves differently than LFD
  → OPT replaces some page in the fast memory
  - As up to request $i+1$, both algorithms behave in the same way, they also have the same fast memory content
  - OPT therefore does not require the new page for request $i+1$
  - Hence, OPT can also load that page later (without extra cost) → OPT′

LFD is Optimal

Theorem: LFD (longest forward distance) is an optimal offline alg.

Proof:

Case 2: Request $i + 1$ does lead to a **page fault**

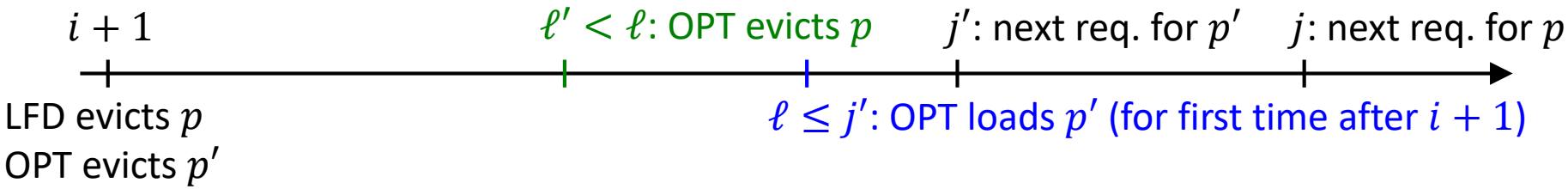
- LFD and OPT move the same page into the fast memory, but they evict different pages
  - If OPT loads more than one page, all pages that are not required for request $i + 1$ can also be loaded later
- Say, LFD evicts page p and OPT evicts page p'
- By the definition of LFD, p' is required again before page p

LFD is Optimal

Theorem: LFD (longest forward distance) is an optimal offline alg.

Proof:

Case 2: Request $i+1$ does lead to a **page fault**



a) OPT keeps p in fast memory until request ℓ
   - Evict p at request $i+1$, keep p' instead and load p (instead of p') back into the fast memory at request ℓ

b) OPT evicts p at request $\ell' < \ell$
   - Evict p at request $i+1$ and p' at request ℓ' (switch evictions of p and p')

Chapter 9
Online Algorithms

Algorithm Theory
WS 2018/19

Fabian Kuhn

Competitive Ratio

- Let's again consider optimization problems
  - For simplicity, assume, we have a minimization problem

Optimal offline solution $\mathbf{OPT}(\boldsymbol{I})$:

- Best objective value that an offline algorithm can achieve for a given input sequence I

Online solution $\mathbf{ALG}(\boldsymbol{I})$:

- Objective value achieved by an online algorithm ALG on I

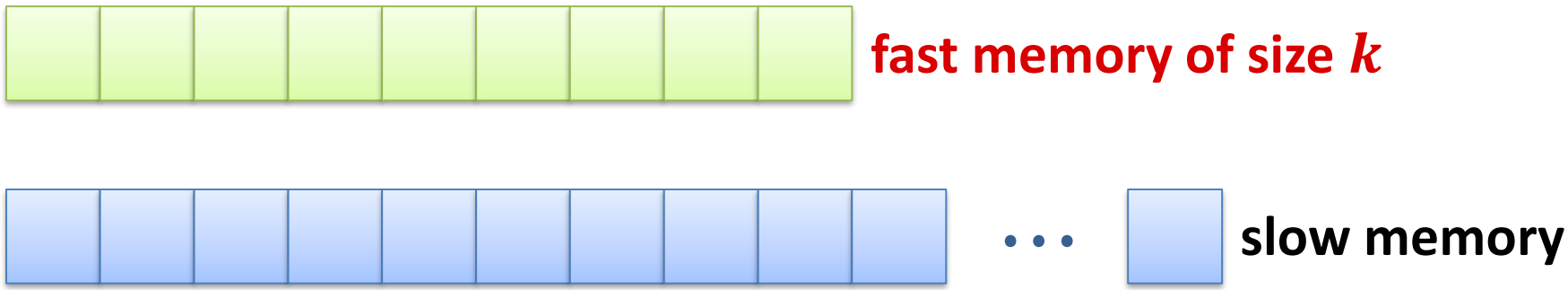
Competitive Ratio: An algorithm has competitive ratio $c \geq 1$ if

$$\mathbf{ALG}(\boldsymbol{I}) \leq \boldsymbol{c} \cdot \mathbf{OPT}(\boldsymbol{I}) + \boldsymbol{\alpha}.$$

- If $\alpha = 0$, we say that ALG is strictly c-competitive.

Paging Algorithm

Assume a simple memory hierarchy:



If a memory page has to be accessed:

- Page in fast memory (hit): take page from there
- Page not in fast memory (miss): leads to a page fault
- Page fault: the page is loaded into the fast memory and some page has to be evicted from the fast memory
- Paging algorithm: decides which page to evict
- Classical online problem: we don't know the future accesses

Paging Strategies

Least Recently Used (LRU):

- Replace the page that hasn't been used for the longest time

First In First Out (FIFO):

- Replace the page that has been in the fast memory longest

Last In First Out (LIFO):

- Replace the page most recently moved to fast memory

Least Frequently Used (LFU):

- Replace the page that has been used the least

Longest Forward Distance (LFD):

- Replace the page whose next request is latest (in the future)
- LFD is **not** an online strategy!

Phase Partition

We partition a given request sequence σ into phases as follows:

- **Phase $\mathbf{0}$**: empty sequence
- **Phase $\boldsymbol{i}$** : maximal sequence that immediately follows phase $i-1$ and contains at most k distinct page requests

Example sequence ($\boldsymbol{k=4}$):

$$2, 5, 12, 5, 4, 2, 10, 8, 3, 6, 2, 2, 6, 6, 8, 3, 2, 6, 9, 10, 6, 3, 10, 2, 1, 3, 5$$

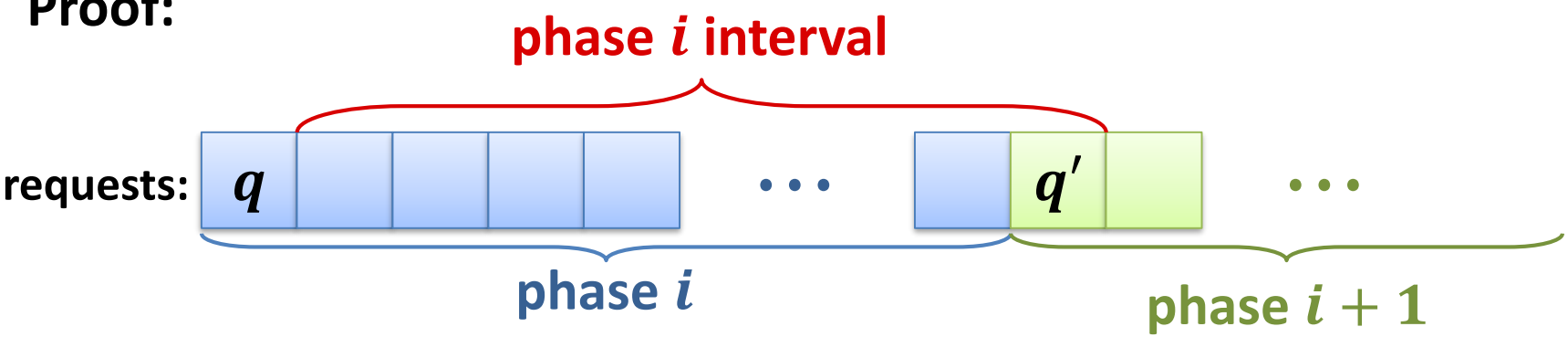
Phase $\boldsymbol{i}$ Interval: interval starting with the second request of phase i and ending with the first request of phase $i+1$

- If the last phase is phase p, phase i interval is defined for $i = 1, \dots, p-1$

Optimal Algorithm

Lemma: Algorithm LFD has at least one page fault in each phase i interval (for $i = 1, \dots, p-1$, where p is the number of phases).

Proof:



- q is in fast memory after first request of phase i
- Number of distinct requests in phase i: k
- By maximality of phase i: q' does not occur in phase i
- Number of distinct requests $\neq q$ in phase interval i: k

→ at least one page fault

LRU and FIFO Algorithms

Lemma: Algorithm LFD has at least one page fault in each phase i interval (for $i = 1, \dots, p-1$, where p is the number of phases).

Corollary: The number of page faults of an optimal offline algorithm is at least $p-1$, where p is the number of phases

Theorem: The LRU and the FIFO algorithms both have a competitive ratio of at most k.

Proof:

- We will show that both have at most k page faults per phase
- We then have (for every input I):

$$\mathrm{LRU}(I), \mathrm{FIFO}(I) \leq k \cdot p \leq k \cdot \mathrm{OPT}(I) + k$$

LRU and FIFO Algorithms

Theorem: The LRU and the FIFO algorithms both have a competitive ratio of at most k.

Proof:

- Need to show that both have at most k page faults per phase
- LRU:
  - The k last pages used are the k least recently used
  - Throughout a phase i, the k distinct pages of phase i are the l.r.u.
  - Once in the fast memory, these pages are therefore not evicted until the end of the phase
- FIFO:
  - In each page fault in phase i, one of the k pages of phase i is loaded into fast memory
  - Once a page is loaded in a page fault of phase i it belongs to the least k pages loaded into fast memory throughout the rest of the phase
  - Hence: Each of the k pages leads to ≤ 1 page fault in phase i

Lower Bound

Theorem: Even if the slow memory contains only $k+1$ pages, any deterministic algorithm has competitive ratio at least k.

Proof:

- Consider some given deterministic algorithm ALG
- Because ALG is deterministic, the content of the fast memory after the first i requests is determined by the first i requests.
- Construct a request sequence inductively as follows:
  - Assume some initial slow memory content
  - The $(i+1)^{\text{st}}$ request is for the page which is not in fast memory after the first i requests (throughout we only use $k+1$ different pages)
- There is a page fault for every request
- OPT has a page fault at most every k requests
  - There is always a page that is not required for the next $k-1$ requests

Randomized Algorithms

- We have seen that deterministic paging algorithms cannot be better than k-competitive

- Does it help to use randomization?

Competitive Ratio: A randomized online algorithm has competitive ratio $c \geq 1$ if for all inputs I,

$$\mathbb{E}[\mathbf{ALG}(\boldsymbol{I})] \leq \boldsymbol{c} \cdot \mathbf{OPT}(\boldsymbol{I}) + \boldsymbol{\alpha}.$$

- If $\alpha \leq 0$, we say that ALG is strictly c-competitive.

Adversaries

- For randomized algorithm, we need to distinguish between different kinds of adversaries (providing the input)

Oblivious Adversary:

- Has to determine the complete input sequence before the algorithm starts
  - The adversary cannot adapt to random decisions of the algorithm

Adaptive Adversary:

- The input sequence is constructed during the execution
- When determining the next input, the adversary knows how the algorithm reacted to the previous inputs
- Input sequence depends on the random behavior of the alg.
- Sometimes, two adaptive adversaries are distinguished
  - offline, online : different way of measuring the adversary cost

Lower Bound

The adversaries can be ordered according to their strength

$$\text{oblivious} < \text{online adaptive} < \text{offline adaptive}$$

- An algorithm that achieves a given comp. ratio with an adaptive adversary is at least as good with an oblivious one
- A lower bound that holds against an oblivious adversary also holds for the two adaptive adversaries
- …

Theorem: No randomized paging algorithm can be better than k-competitive against an adaptive adversary.

Proof: The same proof as for deterministic algorithms works.

- Are there better algorithms with an oblivious adversary?

The Randomized Marking Algorithm

- Every entry in fast memory has a marked flag

- Initially, all entries are unmarked.
- If a page in fast memory is accessed, it gets marked
- When a page fault occurs:
  - If all k pages in fast memory are marked, all marked bits are set to 0
  - The page to be evicted is chosen uniformly at random among the unmarked pages
  - The marked bit of the new page in fast memory is set to 1

Example

Input Sequence (k=6):

$$\underbrace{2, 5, 3, 3, 6, 8, 2, 9, 5}_{\textbf{phase 1}}, \underbrace{7, 1, 2, 5, 2, 3, 7, 4}_{\textbf{phase 2}}, \underbrace{8, 1, 2, 7, 5, 3}_{\textbf{phase 3}}, \underbrace{6, 9, 6, 10, 4, 1, 2}_{\textbf{phase 4}} \ldots$$

Fast Memory:

10	6	1	9	4	2

Observations:

- At the end of a phase, the fast memory entries are exactly the k pages of that phase
- At the beginning of a phase, all entries get unmarked
- #page faults depends on #new pages in a phase

Page Faults per Phase

Consider a fixed phase $\boldsymbol{i}$:

- Assume that of the k pages of phase i, m_i are new and $k - m_i$ are old (i.e., they already appear in phase $i - 1$)
- All m_i new pages lead to page faults (when they are requested for the first time)
- When requested for the first time, an old page leads to a page fault, if the page was evicted in one of the previous page faults
- We need to count the number of page faults for old pages

Page Faults per Phase

Phase i, j^{th} old page that is requested (for the first time):

- There is a page fault if the page has been evicted
- There have been at most $m_i + j - 1$ distinct requests before
- The old places of the $j - 1$ first old pages are occupied
- The other $\leq m_i$ pages are at uniformly random places among the remaining $k - (j - 1)$ places (oblivious adv.)
- Probability that the old place of the jth old page is taken:

$$\leq \frac{m_i}{k - (j - 1)}$$

Page Faults per Phase

Phase $i > 1$, j^{th} old page that is requested (for the first time):

- Probability that there is a page fault:

$$\leq \frac{m_i}{k-(j-1)}$$

Number of page faults for old pages in phase i: F_i

$$\mathbb{E}[F_i] = \sum_{j=1}^{k-m_i} \mathbb{P}\left(j^{\text{th}} \text{ old page incurs page fault}\right)$$

$$\leq \sum_{j=1}^{k-m_i} \frac{m_i}{k-(j-1)} = m_i \cdot \sum_{\ell=m_i+1}^{k} \frac{1}{\ell}$$

$$= m_i \cdot \left(H(k) - H(m_i)\right) \leq m_i \cdot (H(k) - 1)$$

Competitive Ratio

Theorem: Against an oblivious adversary, the randomized marking algorithm has a competitive ratio of at most $2H(k) \leq 2\ln(k) + 2$.

Proof:

- Assume that there are p phases
- #page faults of rand. marking algorithm in phase i: $F_i + m_i$
- We have seen that

$$\mathbb{E}[F_i] \leq m_i \cdot (H(k) - 1) \leq m_i \cdot \ln(k)$$

- Let F be the total number of page faults of the algorithm:

$$\mathbb{E}[F] \leq \sum_{i=1}^{p} (\mathbb{E}[F_i] + m_i) \leq H(k) \cdot \sum_{i=1}^{p} m_i$$

Competitive Ratio

Theorem: Against an oblivious adversary, the randomized marking algorithm has a competitive ratio of at most $2H(k) \leq 2\ln(k) + 2$.

Proof:

- Let F_i^* be the number of page faults in phase i in an opt. exec.
- Phase 1: m_1 pages have to be replaced → $F_1^* \geq m_1$
- Phase $i > 1$:
  - Number of distinct page requests in phases $i-1$ and i: $\boldsymbol{k + m_i}$
  - Therefore, $\boldsymbol{F_{i-1}^* + F_i^* \geq m_i}$
- Total number of page requests F^*:

$$F^* = \sum_{i=1}^{p} F_i^* \geq \frac{1}{2} \cdot \left(F_1^* + \sum_{i=2}^{p} (F_{i-1}^* + F_i^*) \right) \geq \frac{1}{2} \cdot \sum_{i=1}^{p} m_i$$

Competitive Ratio

Theorem: Against an oblivious adversary, the randomized marking algorithm has a competitive ratio of at most $2H(k) \leq 2\ln(k) + 2$.

Proof:

- Randomized marking algorithm:

$$\mathbb{E}[F] \leq H(k) \cdot \sum_{i=1}^{p} m_i$$

- Optimal algorithm:

$$F^* \geq \frac{1}{2} \cdot \sum_{i=1}^{p} m_i$$

Remark: It can be shown that no randomized algorithm has a competitive ratio better than $H(k)$ (against an obl. adversary)

Chapter 10
Parallel Algorithms

Algorithm Theory
WS 2018/19

Fabian Kuhn

Sequential Algorithms

Classical Algorithm Design:

- One machine/CPU/process/… doing a computation

RAM (Random Access Machine):

- Basic standard model
- Unit cost basic operations
- Unit cost access to all memory cells

Sequential Algorithm / Program:

- Sequence of operations
  (executed one after the other)

Parallel and Distributed Algorithms

Today's computers/systems are not sequential:

- Even cell phones have several cores
- Future systems will be highly parallel on many levels
- This also requires appropriate algorithmic techniques

Goals, Scenarios, Challenges:

- Exploit parallelism to speed up computations
- Shared resources such as memory, bandwidth, …
- Increase reliability by adding redundancy
- Solve tasks in inherently decentralized environments
- …

Parallel and Distributed Systems

- Many different forms

- Processors/computers/machines/… communicate and share data through
  - Shared memory or message passing
- Computation and communication can be
  - Synchronous or asynchronous
- Many possible topologies for message passing
- Depending on system, various types of faults

Challenges

Algorithmic and theoretical challenges:

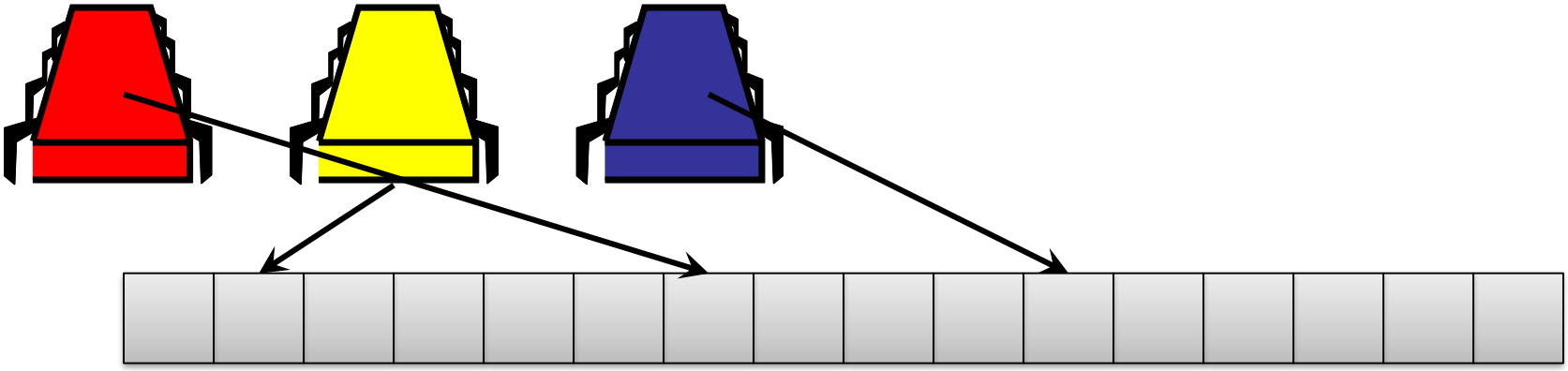
- How to parallelize computations
- Scheduling (which machine does what)
- Load balancing
- Fault tolerance
- Coordination / consistency
- Decentralized state
- Asynchrony
- Bounded bandwidth / properties of comm. channels
- …

Models

- A large variety of models, e.g.:
- **PRAM** (Parallel Random Access Machine)
  - Classical model for parallel computations
- **Shared Memory**
  - Classical model to study coordination / agreement problems, distributed data structures, …
- **Message Passing** (fully connected topology)
  - Closely related to shared memory models
- Message Passing in **Networks**
  - Decentralized computations, large parallel machines, comes in various flavors…
- Computations in large clusters of powerful individual machines: Massively Parallel Computations (MPC)

PRAM

- Parallel version of RAM model
- p processors, shared random access memory



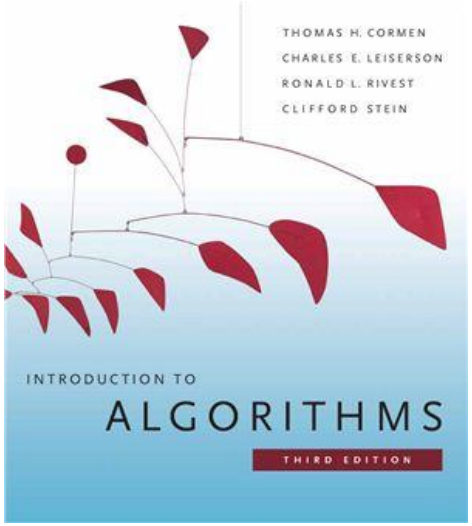
- Basic operations / access to shared memory cost 1
- Processor operations are synchronized
- Focus on parallelizing computation rather than cost of communication, locality, faults, asynchrony, …

Other Parallel Models

- **Message passing:** Fully connected network, local memory and information exchange using messages

- **Dynamic Multithreaded Algorithms:** Simple parallel programming paradigm
  - E.g., used in Cormen, Leiserson, Rivest, Stein (CLRS)

```
FIB(n)
if n < 2
    then return n
x ← spawn FIB(n − 1)
y ← spawn FIB(n − 2)
sync
return (x + y)
```



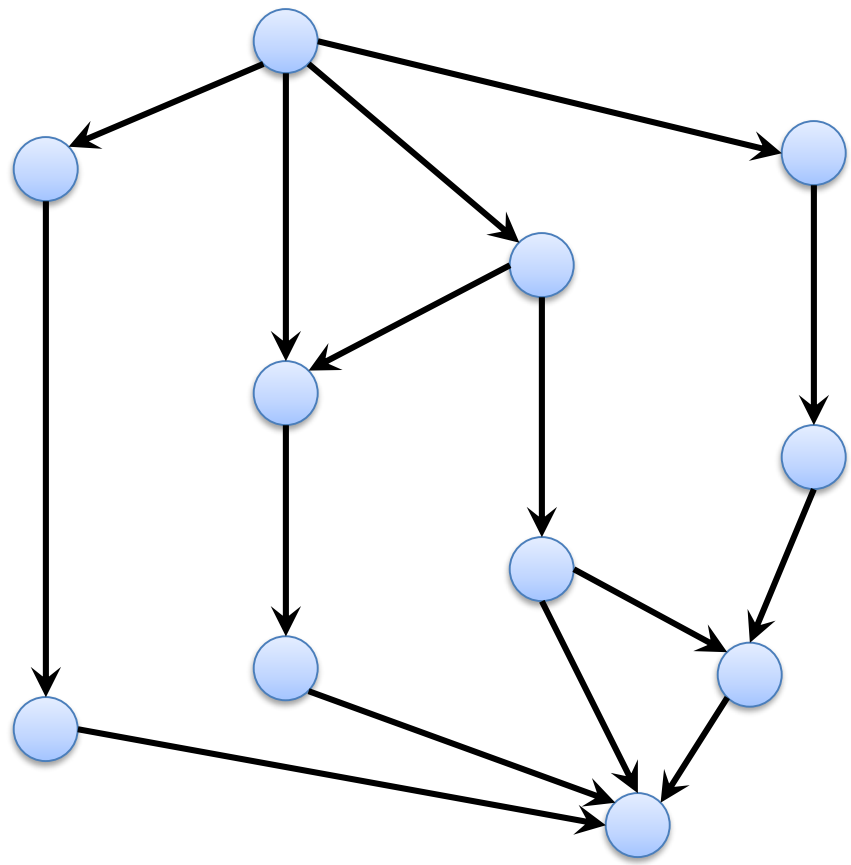
Parallel Computations

Sequential Computation:

- Sequence of operations

Parallel Computation:

- Directed Acyclic Graph (DAG)

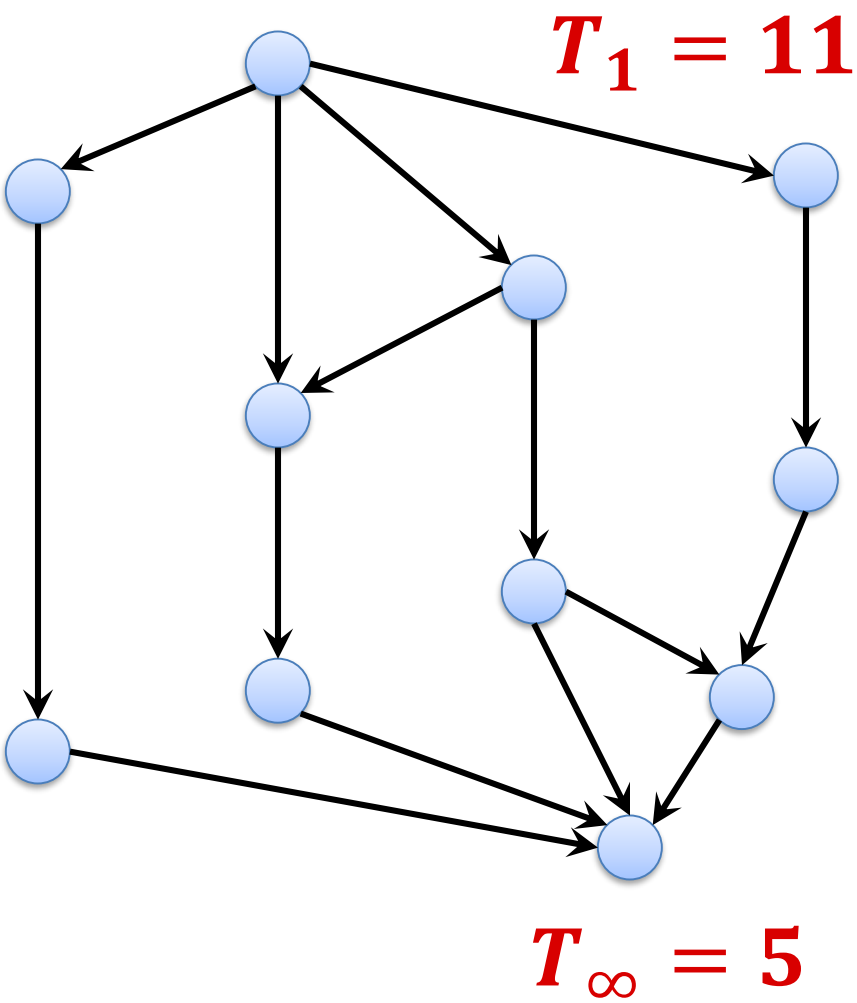


Parallel Computations

T_p: time to perform comp. with p procs

- T_1: **work** (total # operations)
  - Time when doing the computation sequentially
- T_∞: **critical path / span**
  - Time when parallelizing as much as possible
- **Lower Bounds**:

$$T_p \geq \frac{T_1}{p}, \qquad T_p \geq T_\infty$$



Chapter 10
Parallel Algorithms

Algorithm Theory
WS 2018/19

Fabian Kuhn

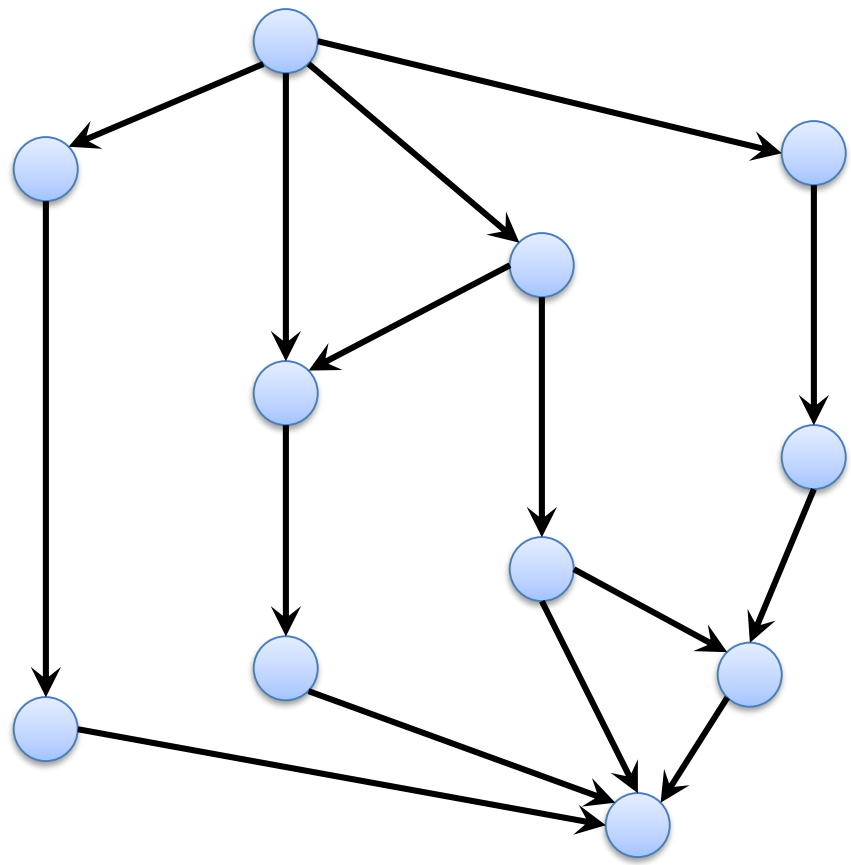
Parallel Computations

Sequential Computation:

- Sequence of operations

Parallel Computation:

- Directed Acyclic Graph (DAG)

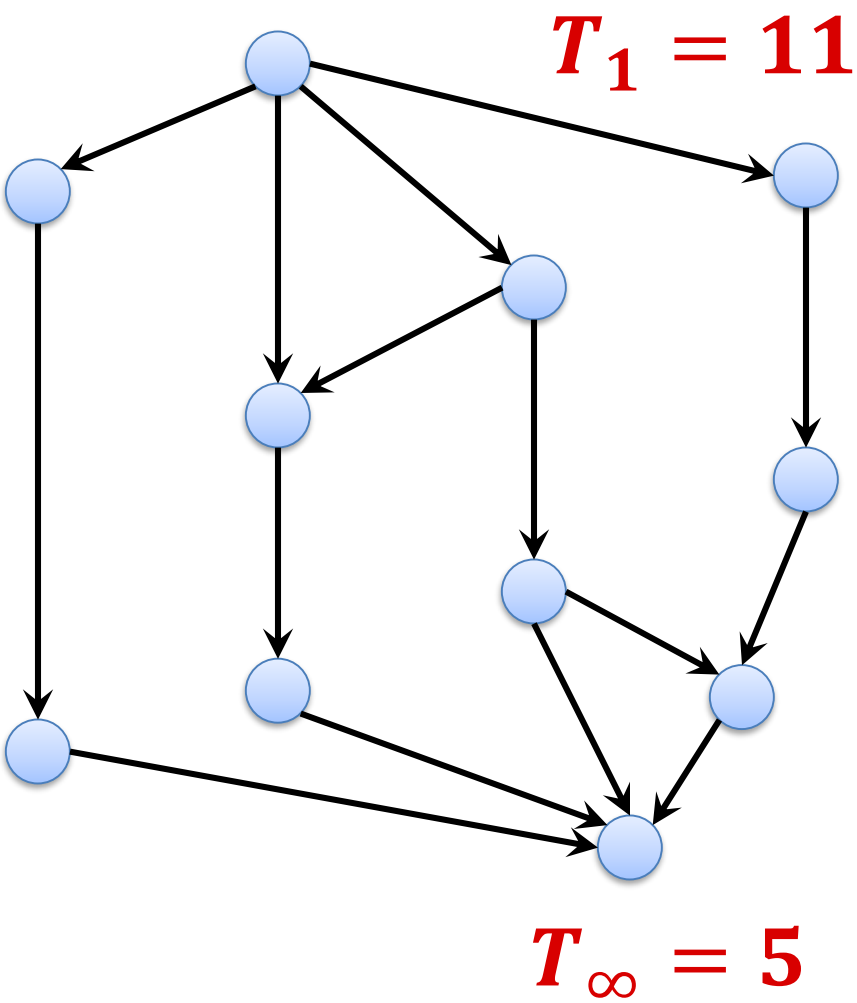


Parallel Computations

T_p: time to perform comp. with p procs

- T_1: **work** (total # operations)
  - Time when doing the computation sequentially
- T_∞: **critical path / span**
  - Time when parallelizing as much as possible
- **Lower Bounds**:

$$T_p \geq \frac{T_1}{p}, \qquad T_p \geq T_\infty$$



Parallel Computations

T_p: time to perform comp. with p procs

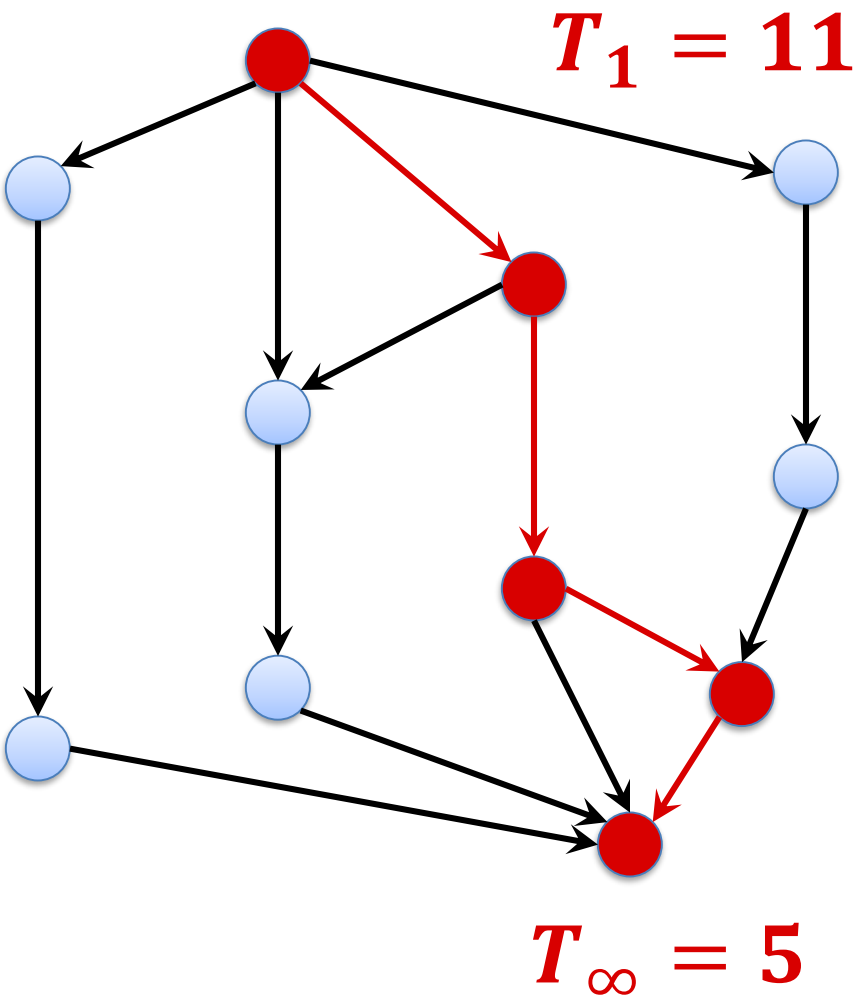
- **Lower Bounds**:

$$T_p \geq \frac{T_1}{p}, \qquad T_p \geq T_\infty$$

- **Parallelism**: $\frac{T_1}{T_\infty}$
  - maximum possible speed-up

- **Linear Speed-up**:

$$\frac{T_1}{T_p} = \Theta(p)$$



Scheduling

- How to assign operations to processors?

- Generally an online problem
  - When scheduling some jobs/operations, we do not know how the computation evolves over time

Greedy (offline) scheduling:

- Order jobs/operations as they would be scheduled optimally with ∞ processors (topological sort of DAG)
  - Easy to determine: With ∞ processors, one always schedules all jobs/ops that can be scheduled
- Always schedule as many jobs/ops as possible
- Schedule jobs/ops in the same order as with ∞ processors
  - i.e., jobs that become available earlier have priority

Brent's Theorem

Brent's Theorem: On p processors, a parallel computation can be performed in time

$$T_p \leq \frac{T_1 - T_\infty}{p} + T_\infty.$$

Proof:

- Greedy scheduling achieves this…
- #operations scheduled with ∞ processors in round i: x_i

Brent's Theorem

Brent's Theorem: On p processors, a parallel computation can be performed in time

$$T_p \leq \frac{T_1 - T_\infty}{p} + T_\infty.$$

Proof:

- Greedy scheduling achieves this…
- #operations scheduled with ∞ processors in round i: x_i

Brent's Theorem

Brent's Theorem: On p processors, a parallel computation can be performed in time

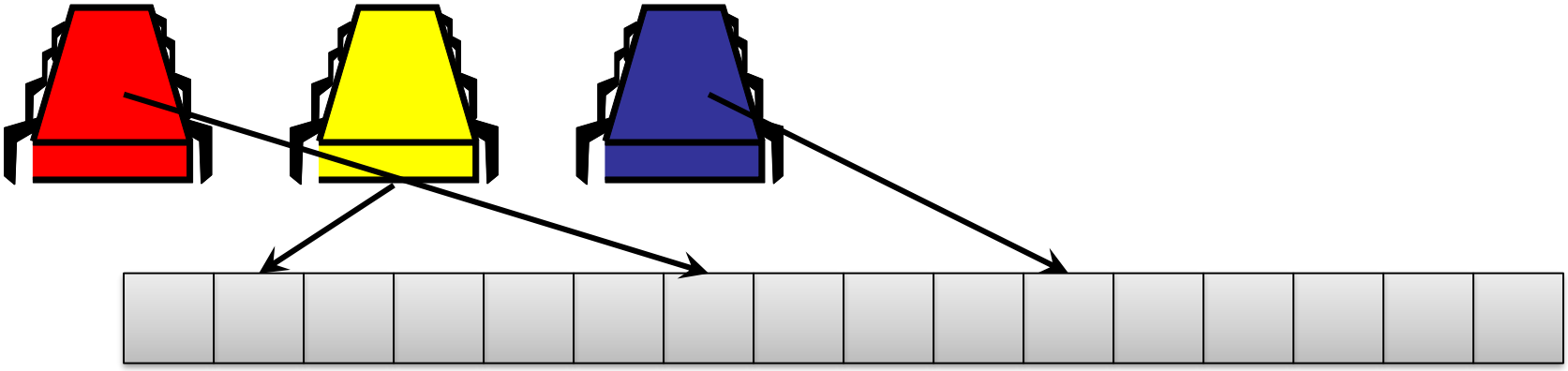
$$\boldsymbol{T_p \leq \frac{T_1 - T_\infty}{p} + T_\infty}.$$

Corollary: Greedy is a 2-approximation algorithm for scheduling.

Corollary: As long as the number of processors $p = \mathrm{O}(T_1/T_\infty)$, it is possible to achieve a linear speed-up.

PRAM

- Parallel version of RAM model
- p processors, shared random access memory



- Basic operations / access to shared memory cost 1
- Processor operations are synchronized
- Focus on parallelizing computation rather than cost of communication, locality, faults, asynchrony, …

PRAM

The Classic Computational Model to Study Parallel Computations:

- The PRAM model comes in variants…

EREW (exclusive read, exclusive write):

- Concurrent memory access by multiple processors is not allowed
- If two or more processors try to read from or write to the same memory cell concurrently, the behavior is not specified

CREW (concurrent read, exclusive write):

- Reading the same memory cell concurrently is OK
- Two concurrent writes to the same cell lead to unspecified behavior
- This is the first variant that was considered (already in the 70s)

PRAM

The PRAM model comes in variants…

CRCW (concurrent read, concurrent write):

- Concurrent reads and writes are both OK
- Behavior of concurrent writes has to specified
  - Weak CRCW: concurrent write only OK if all processors write 0
  - Common-mode CRCW: all processors need to write the same value
  - Arbitrary-winner CRCW: adversary picks one of the values
  - Priority CRCW: value of processor with highest ID is written
  - Strong CRCW: largest (or smallest) value is written

- The given models are ordered in strength:

  weak $\leq$ common-mode $\leq$ arbitrary-winner $\leq$ priority $\leq$ strong

Some Relations Between PRAM Models

Theorem: A parallel computation that can be performed in time t, using p proc. on a strong CRCW machine, can also be performed in time $O(t \log p)$ using p processors on an EREW machine.

- Each (parallel) step on the CRCW machine can be simulated by $O(\log p)$ steps on an EREW machine

Some Relations Between PRAM Models

Theorem: A parallel computation that can be performed in time t, using p proc. on a strong CRCW machine, can also be performed in time $O(t \log p)$ using p processors on an EREW machine.

- Each (parallel) step on the CRCW machine can be simulated by $O(\log p)$ steps on an EREW machine

Some Relations Between PRAM Models

Theorem: A parallel computation that can be performed in time t, using p proc. on a strong CRCW machine, can also be performed in time $O(t \log p)$ using p processors on an EREW machine.

- Each (parallel) step on the CRCW machine can be simulated by $O(\log p)$ steps on an EREW machine

Theorem: A parallel computation that can be performed in time t, using p probabilistic processors on a strong CRCW machine, can also be performed in expected time $O(t \log p)$ using $O(p/\log p)$ processors on an arbitrary-winner CRCW machine.

- The same simulation turns out more efficient in this case

Some Relations Between PRAM Models

Theorem: A computation that can be performed in time t, using p processors on a strong CRCW machine, can also be performed in time $O(t)$ using $O(p^2)$ processors on a weak CRCW machine

Proof:

- Strong: largest value wins, weak: only concurrently writing 0 is OK

Some Relations Between PRAM Models

Theorem: A computation that can be performed in time t, using p processors on a strong CRCW machine, can also be performed in time $O(t)$ using $O(p^2)$ processors on a weak CRCW machine

Proof:

- Strong: largest value wins, weak: only concurrently writing 0 is OK

Computing the Maximum

Given: n values

Goal: find the maximum value

Observation: The maximum can be computed in parallel by using a binary tree.

Computing the Maximum

Observation: On a strong CRCW machine, the maximum of a n values can be computed in $O(1)$ time using n processors

- Each value is concurrently written to the same memory cell

Lemma: On a weak CRCW machine, the maximum of n integers between 1 and $\sqrt{n}$ can be computed in time $O(1)$ using $O(n)$ proc.

Proof:

- We have $\sqrt{n}$ memory cells $f_1, \dots, f_{\sqrt{n}}$ for the possible values
- Initialize all $f_i := 1$
- For the n values $x_1, \dots, x_n$, processor j sets $f_{x_j} := 0$
  - Since only zeroes are written, concurrent writes are OK
- Now, $f_i = 0$ iff value i occurs at least once
- Strong CRCW machine: max. value in time $O(1)$ w. $O(\sqrt{n})$ proc.
- Weak CRCW machine: time $O(1)$ using $O(n)$ proc. (prev. lemma)

Computing the Maximum

Theorem: If each value can be represented using $O(\log n)$ bits, the maximum of n (integer) values can be computed in time $O(1)$ using $O(n)$ processors on a weak CRCW machine.

Proof:

- First look at $\frac{\log_2 n}{2}$ highest order bits
- The maximum value also has the maximum among those bits
- There are only $\sqrt{n}$ possibilities for these bits
- max. of $\frac{\log_2 n}{2}$ highest order bits can be computed in $O(1)$ time
- For those with largest $\frac{\log_2 n}{2}$ highest order bits, continue with next block of $\frac{\log_2 n}{2}$ bits, …

Prefix Sums

- The following works for any associative binary operator $\oplus$:

  associativity: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$

All-Prefix-Sums: Given a sequence of n values $a_1, \dots, a_n$, the all-prefix-sums operation w.r.t. $\oplus$ returns the sequence of prefix sums:

$$s_1, s_2, \dots, s_n = a_1, a_1 \oplus a_2, a_1 \oplus a_2 \oplus a_3, \dots, a_1 \oplus \cdots \oplus a_n$$

- Can be computed efficiently in parallel and turns out to be an important building block for designing parallel algorithms

Example: Operator: $+$, input: $a_1, \dots, a_8 = 3, 1, 7, 0, 4, 1, 6, 3$

$s_1, \dots, s_8 =$

Computing the Sum

- Let's first look at $s_n = a_1 \oplus a_2 \oplus \cdots \oplus a_n$
- Parallelize using a binary tree:

Computing the Sum

Lemma: The sum $s_n = a_1 \oplus a_2 \oplus \cdots \oplus a_n$ can be computed in time $O(\log n)$ on an EREW PRAM. The total number of operations (total work) is $O(n)$.

Proof:

Corollary: The sum s_n can be computed in time $O(\log n)$ using $O(n/\log n)$ processors on an EREW PRAM.

Proof:

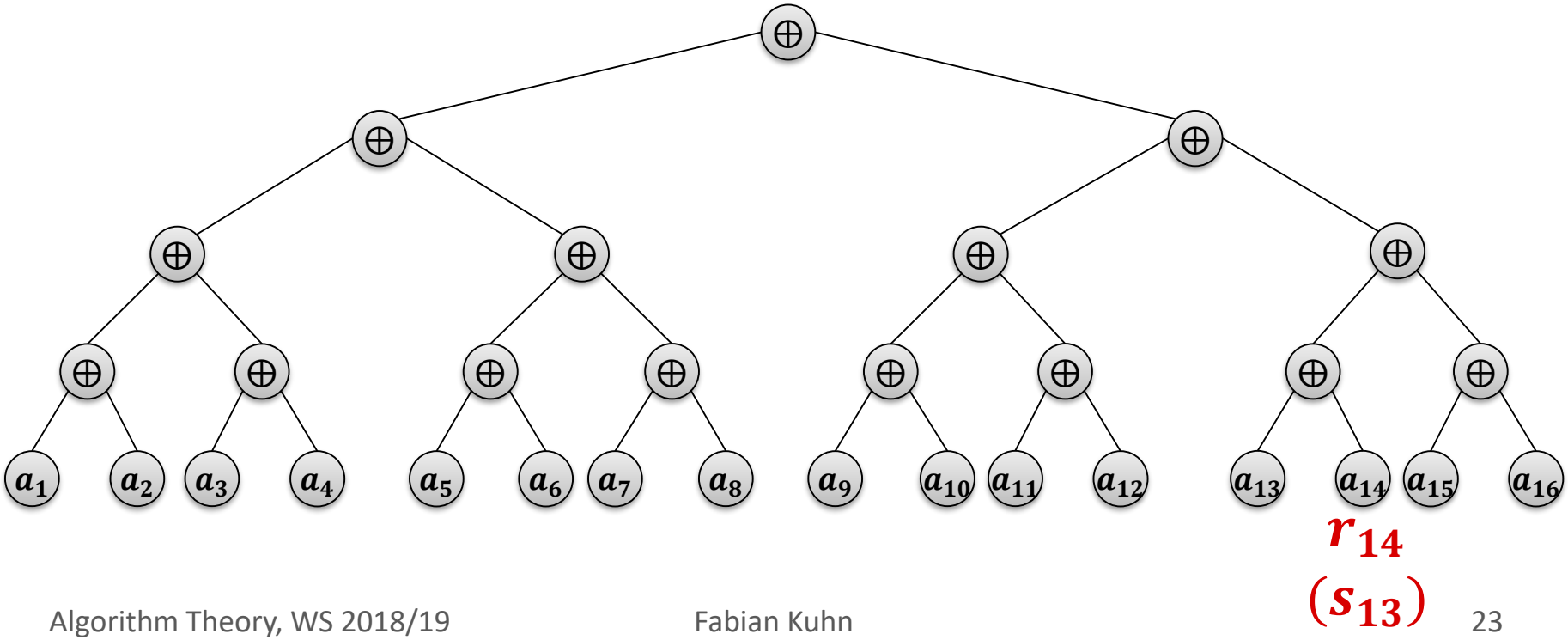
- Follows from Brent's theorem ($T_1 = O(n), T_\infty = O(\log n)$)

Getting The Prefix Sums

- Instead of computing the sequence $s_1, s_2, \dots, s_n$ let's compute $r_1, \dots, r_n = 0, s_1, s_2, \dots, s_{n-1}$ (0: neutral element w.r.t. $\oplus$)

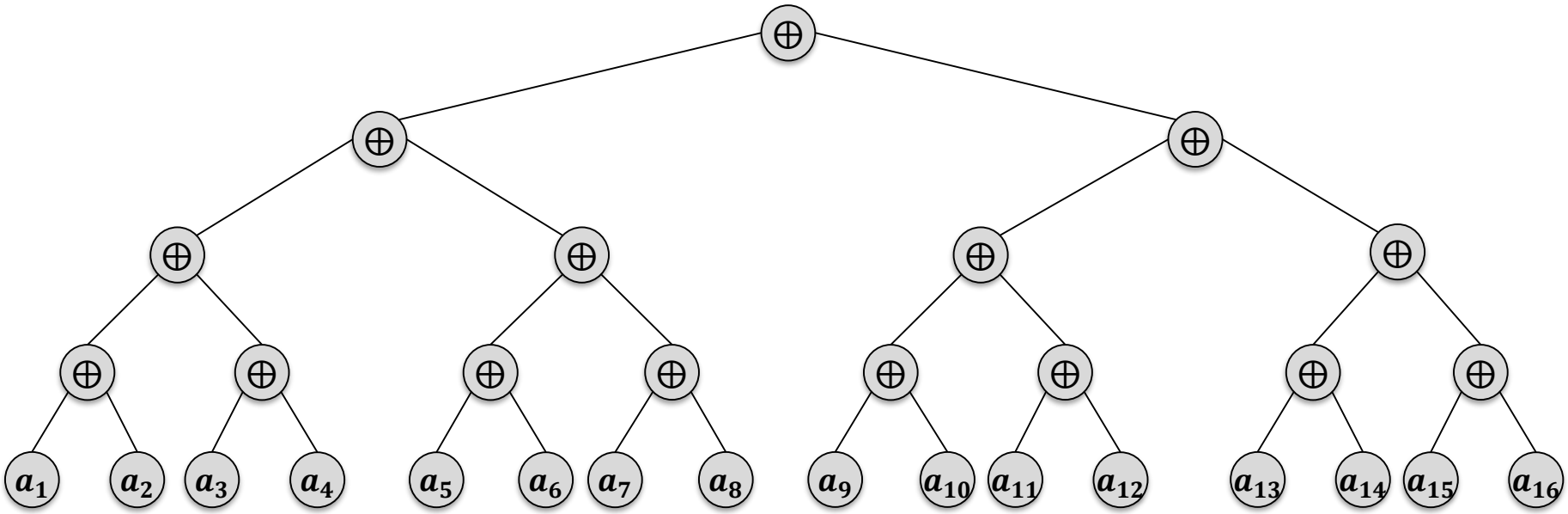
$$r_1, \dots, r_n = 0, a_1, a_1 \oplus a_2, \dots, a_1 \oplus \cdots \oplus a_{n-1}$$

- Together with s_n, this gives all prefix sums
- Prefix sum $r_i = s_{i-1} = a_1 \oplus \cdots \oplus a_{i-1}$:



Getting The Prefix Sums

Claim: The prefix sum $r_i = a_1 \oplus \cdots \oplus a_{i-1}$ is the sum of all the leaves in the left sub-tree of ancestor u of the leaf v containing a_i such that v is in the right sub-tree of u.



Computing The Prefix Sums

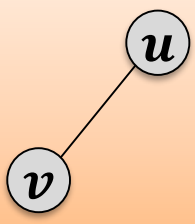
For each node v of the binary tree, **define $r(v)$ as follows:**

- $r(v)$ is the sum of the values a_i at the leaves in all the left sub-trees of ancestors u of v such that v is in the right sub-tree of u.

For a leaf node v holding value a_i: $\boldsymbol{r(v) = r_i = s_{i-1}}$

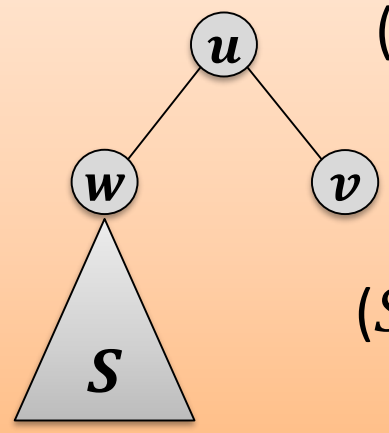
For the root node: $\boldsymbol{r(\mathrm{root}) = 0}$

For all other nodes v:



v is the left child of u:

$$r(v) = r(u)$$



v is the right child of u:
(u has left child w)

$$r(v) = r(u) + S$$

(S: sum of values in sub-tree of w)

Computing The Prefix Sums

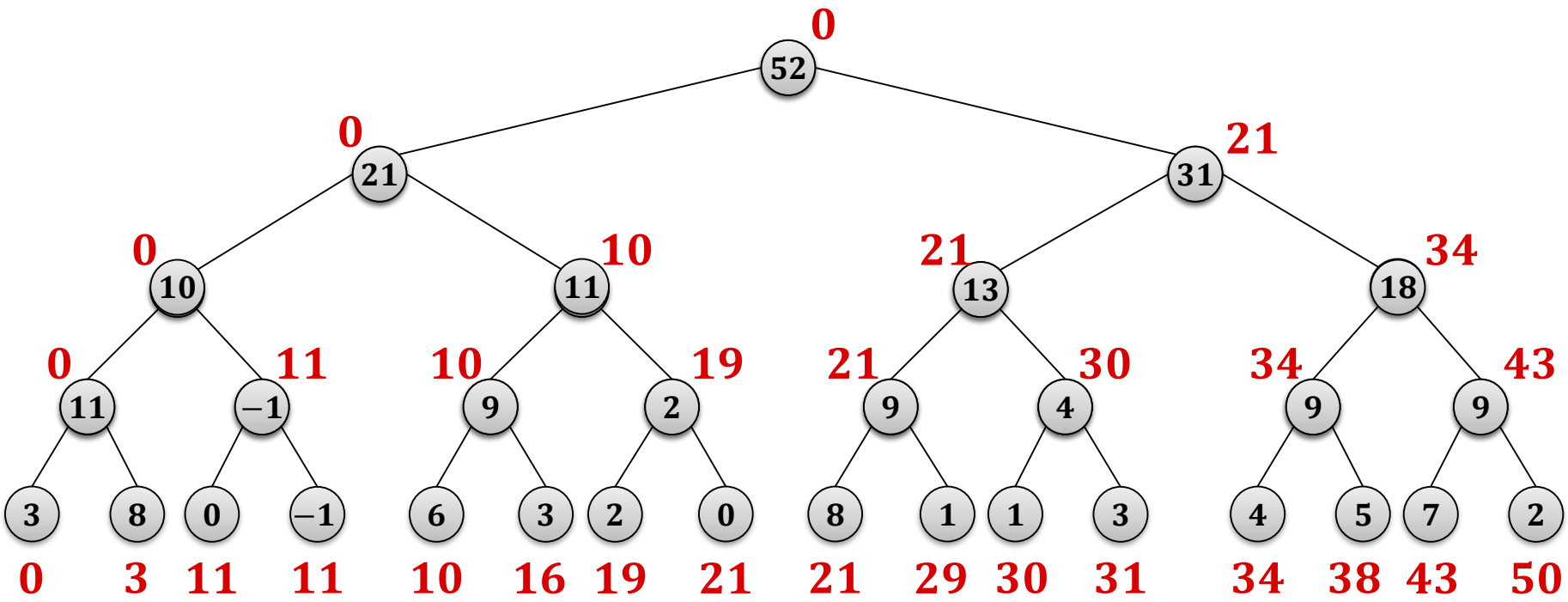
- leaf node v holding value a_i: $\boldsymbol{r(v) = r_i = s_{i-1}}$
- root node: $\boldsymbol{r(\mathrm{root}) = 0}$
- Node v is the left child of u: $r(v) = r(u)$
- Node v is the right child of u: $r(v) = r(u) + S$
  - Where: $S =$ sum of values in left sub-tree of u

Algorithm to compute values $\boldsymbol{r(v)}$:

1. Compute sum of values in each sub-tree (bottom-up)
   - Can be done in parallel time $O(\log n)$ with $O(n)$ total work
2. Compute values $r(v)$ top-down from root to leaves:
   - To compute the value $r(v)$, only $r(u)$ of the parent u and the sum of the left sibling (if v is a right child) are needed
   - Can be done in parallel time $O(\log n)$ with $O(n)$ total work

Example

1. Compute sums of all sub-trees
   - Bottom-up (level-wise in parallel, starting at the leaves)
2. Compute values $r(v)$
   - Top-down (starting at the root)



Computing Prefix Sums

Theorem: Given a sequence $a_1, \dots, a_n$ of n values, all prefix sums $s_i = a_1 \oplus \cdots \oplus a_i$ (for $1 \leq i \leq n$) can be computed in time $O(\log n)$ using $O(n/\log n)$ processors on an EREW PRAM.

Proof:

- Computing the sums of all sub-trees can be done in parallel in time $O(\log n)$ using $O(n)$ total operations.
- The same is true for the top-down step to compute the $r(v)$
- The theorem then follows from Brent's theorem:

$$T_1 = O(n), \qquad T_\infty = O(\log n) \qquad \Longrightarrow \qquad T_p < T_\infty + \frac{T_1}{p}$$

Remark: This can be adapted to other parallel models and to different ways of storing the value (e.g., array or list)

Parallel Quicksort

- Key challenge: parallelize partition

pivot

5	14	18	8	19	21	3	1	25	17	11	4	20	10	26	2	9	13	23	16

partition

5	14	8	3	1	11	4	10	2	9	13	16	18	19	21	25	17	20	26	23

- How can we do this in parallel?
- For now, let's just care about the values $\leq$ pivot
- What are their new positions

Using Prefix Sums

- Goal: Determine positions of values $\leq$ pivot after partition

pivot

5	14	18	8	19	21	3	1	25	17	11	4	20	10	26	2	9	13	23	16

1	1	0	1	0	0	1	1	0	0	1	1	0	1	0	1	1	1	0	1

prefix sums

1	2	2	3	3	3	4	5	5	5	6	7	7	8	8	9	10	11	11	12

partition

5	14	8	3	1	11	4	10	2	9	13	16	18	19	21	25	17	20	26	23

Partition Using Prefix Sums

- The positions of the entries $>$ pivot can be determined in the same way
- **Prefix sums:** $T_1 = O(n), \quad T_\infty = O(\log n)$
- **Remaining computations:** $T_1 = O(n), \quad T_\infty = O(1)$
- **Overall:** $T_1 = O(n), \quad T_\infty = O(\log n)$

Lemma: The partitioning of quicksort can be carried out in parallel in time $O(\log n)$ using $O\left(\frac{n}{\log n}\right)$ processors.

Proof:

- By Brent's theorem: $T_p \leq \frac{T_1}{p} + T_\infty$

Applying to Quicksort

Theorem: On an EREW PRAM, using p processors, randomized quicksort can be executed in time T_p (in expectation and with high probability), where

$$T_p = O\left(\frac{n \log n}{p} + \log^2 n\right).$$

Proof:

Remark:

- We get optimal (linear) speed-up w.r.t. to the sequential algorithm for all $p = O(n/\log n)$.

Other Applications of Prefix Sums

- Prefix sums are a very powerful primitive to design parallel algorithms.
  - Particularly also by using other operators than “+”

Example Applications:

- Lexical comparison of strings
- Add multi-precision numbers
- Evaluate polynomials
- Solve recurrences
- Radix sort / quick sort
- Search for regular expressions
- Implement some tree operations
- …

Repetition
Probability Theory

Algorithm Theory
WS 2018/19

Fabian Kuhn

Randomized Algorithms

Randomized Algorithms

- An algorithm that uses (or can use) random coin flips in order to make decisions
- randomization can be a powerful tool to make algorithms faster or simpler

First: Short Repetition of Basic Probability Theory

- We need: basic discrete probability theory
  - probability spaces, probability events, independence, random variables, expectation, linearity of expectation, Markov inequality
- Literature, for example
  - your old probability theory book / lecture notes / ...
  - Appendix C of book of Cormen, Rivest, Leiserson, Stein
  - http://www.ti.inf.ethz.ch/ew/courses/APC15/material/ra.pdf

Probability Space and Events

Definition: A (discrete) **probability space** is a pair $(\Omega, \mathbb{P})$, where

- Ω: (countable) set of elementary events
- $\mathbb{P}$: assigns a probability to each $\omega \in \Omega$

$$\mathbb{P} : \Omega \to \mathbb{R}_{\geq 0} \quad \text{s.t.} \quad \sum_{\omega \in \Omega} \mathbb{P}(\omega) = 1$$

Definition: An **event $\mathcal{E}$** is a subset of Ω

- Event $\mathcal{E} \subseteq \Omega$: set of basic events
- Probability of $\mathcal{E}$

$$\mathbb{P}(\mathcal{E}) := \sum_{\omega \in \mathcal{E}} \mathbb{P}(\omega)$$

Example: Probability Space, Events

Example: Probability Space, Events

Independent Events

Definition: Events $\mathcal{A} \subseteq \Omega$ and $\mathcal{B} \subseteq \Omega$ are **independent** iff

$$\mathbb{P}(\boldsymbol{\mathcal{A}} \cap \boldsymbol{\mathcal{B}}) = \mathbb{P}(\boldsymbol{\mathcal{A}}) \cdot \mathbb{P}(\boldsymbol{\mathcal{B}})$$

Example:

Random Variables

Definition: A **random variable $\boldsymbol{X}$** is a real-valued function on the elementary events Ω

$$\boldsymbol{X} : \boldsymbol{\Omega} \to \mathbb{R}$$

- We usually write X instead of $X(\omega)$
- We also write

$$\mathbb{P}(X = x) = \mathbb{P}(\{\omega \in \Omega : X(\omega) = x\})$$

Examples:

- $\boldsymbol{X^{top}}$: $X^{top}(1) = 1, X^{top}(2) = 2, \dots, X^{top}(6) = 6$
- $\boldsymbol{X^{bot}}$: $X^{bot}(1) = 6, X^{bot}(2) = 5, \dots, X^{bot}(6) = 1$

- Note that for all $\omega \in \Omega$, $X^{top}(\omega) + X^{bot}(\omega) = 7$
- To denote this, we write $X^{top} + X^{bot} = 7$

Indicator Random Variables

A random variable with only takes values 0 and 1 is called a **Bernoulli random variable** or an **indicator random variable**.

Independent Random Variables

Definition: Two random variables X and Y are called **independent** if

$$\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}: \ \mathbb{P}(\boldsymbol{X} = \boldsymbol{x} \wedge \boldsymbol{Y} = \boldsymbol{y}) = \mathbb{P}(\boldsymbol{X} = \boldsymbol{x}) \cdot \mathbb{P}(\boldsymbol{Y} = \boldsymbol{y})$$

Independent Random Variables

Definition: A collection of andom variables $X_1, X_2, \dots, X_n$ on a probability space Ω is called **mutually independent** if

$$\forall \boldsymbol{k} \geq \mathbf{2}, \mathbf{1} \leq \boldsymbol{i_1} < \cdots < \boldsymbol{i_k} \leq \boldsymbol{n}, \forall \boldsymbol{x_{i_1}}, \dots, \boldsymbol{x_{i_k}} \in \mathbb{R}:$$
$$\mathbb{P}\left(\boldsymbol{X_{i_1}} = \boldsymbol{x_{i_1}} \wedge \cdots \wedge \boldsymbol{X_{i_k}} = \boldsymbol{x_{i_k}}\right) = \mathbb{P}\left(\boldsymbol{X_{i_1}} = \boldsymbol{x_{i_1}}\right) \cdot \dots \cdot \mathbb{P}(\boldsymbol{X_{i_k}} = \boldsymbol{x_{i_k}})$$

Expectation

Definition: The **expectation** of a random variable X is defined as

$$\mathbb{E}[\boldsymbol{X}] := \sum_{\boldsymbol{x} \in \boldsymbol{X}(\boldsymbol{\Omega})} \boldsymbol{x} \cdot \mathbb{P}(\boldsymbol{X} = \boldsymbol{x}) = \sum_{\omega \in \Omega} X(\omega) \cdot \mathbb{P}(\omega)$$

Example:

- recall: X^{top} is outcome of rolling a die

Expectation: Examples

Sums and Products of Random Variables

Linearity of Expectation:

For random variables X and Y and any $c \in \mathbb{R}$, we have

$$\mathbb{E}[\boldsymbol{cX}] = \boldsymbol{c} \cdot \mathbb{E}[\boldsymbol{X}]$$
$$\mathbb{E}[\boldsymbol{X} + \boldsymbol{Y}] = \mathbb{E}[\boldsymbol{X}] + \mathbb{E}[\boldsymbol{Y}]$$

- holds also if the random variables are not independent

Product of Random Variables:

For two **independent** random variables X and Y, we have

$$\mathbb{E}[\boldsymbol{X} \cdot \boldsymbol{Y}] = \mathbb{E}[\boldsymbol{X}] \cdot \mathbb{E}[\boldsymbol{Y}]$$

Sums and Products of Random Variables

Linearity of Expectation:

For random variables X and Y and any $c \in \mathbb{R}$, we have

$$\mathbb{E}[\boldsymbol{cX}] \quad = \boldsymbol{c} \cdot \mathbb{E}[\boldsymbol{X}], \qquad \mathbb{E}[\boldsymbol{X} + \boldsymbol{Y}] = \mathbb{E}[\boldsymbol{X}] + \mathbb{E}[\boldsymbol{Y}]$$

Sums and Products of Random Variables

Product of Random Variables:

For two **independent** random variables X and Y, we have

$$\mathbb{E}[\boldsymbol{X} \cdot \boldsymbol{Y}] = \mathbb{E}[\boldsymbol{X}] \cdot \mathbb{E}[\boldsymbol{Y}]$$

Linearity of Expectation: Example

Sequence of coin flips: $C_1, C_2, \ldots \in \{H, T\}$

- Stop as soon as the first H turns up

Random variable $\boldsymbol{X}$: number of T before first H

Indicator random variable $\boldsymbol{X_i}$ ($\boldsymbol{i \geq 1}$):

- $X_i = 1$: i^{th} coin flip happens and its outcome is T
  $X_i = 0$: otherwise

Markov's Inequality

Lemma: Let X be a nonnegative random variable.
Then for all $c > 0$

$$\mathbb{P}(X \geq c \cdot \mathbb{E}[X]) \leq \frac{1}{c}$$

Conditional Probabilities

For events $\mathcal{A} \subseteq \Omega$ and $\mathcal{B} \subseteq \Omega$, the **conditional probability** of $\mathcal{A}$ given $\mathcal{B}$ is defined as

$$\mathbb{P}(\boldsymbol{\mathcal{A}}|\boldsymbol{\mathcal{B}}) := \frac{\mathbb{P}(\boldsymbol{\mathcal{A}} \cap \boldsymbol{\mathcal{B}})}{\mathbb{P}(\boldsymbol{\mathcal{B}})}$$

Conditioning on event $\mathcal{B}$ defines a new probability space $(\mathcal{B}, \mathbb{P}')$

$$\forall \omega \in B : \ \mathbb{P}'(\omega) = \frac{\mathbb{P}(\omega)}{\mathbb{P}(\mathcal{B})}.$$

Two events are **independent iff** $\mathbb{P}(\boldsymbol{\mathcal{A}}|\boldsymbol{\mathcal{B}}) = \mathbb{P}(\boldsymbol{\mathcal{A}})$

Law of Total Probability / Expectation

Lemma: Let X and Y be two random variables on the same probability space $(\Omega, \mathbb{P})$. We then have

$$\forall \boldsymbol{x} \in \mathbb{R} : \mathbb{P}(\boldsymbol{X} = \boldsymbol{x}) = \sum_{\boldsymbol{y} \in \boldsymbol{Y}(\boldsymbol{\Omega})} \mathbb{P}(\boldsymbol{X} = \boldsymbol{x} \mid \boldsymbol{Y} = \boldsymbol{y}) \cdot \mathbb{P}(\boldsymbol{Y} = \boldsymbol{y}) .$$

$$\mathbb{E}[X] = \sum_{y \in \mathrm{Y}(\Omega)} \mathbb{E}[X \mid Y = y] \cdot \mathbb{P}(Y = y)$$