A&D

Math Basis
- Asymp. Notation
- Induction
- Loop-Counting

Max-Subarray-Sum
- naive, divide-conquer, inductive
- complexity

Searchs
- Linear Search
- Binary Search
- Lower-Bound

Sorts
- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quicksort
- Heapsort
- Lower-Bound

Data Structures
- Array
- List (Linked, Doubly)
- Trees (BST, HeapTree, AVL Tree)

DP
- Fibonacci
- Max-Subarray-Sum
- Jump-Game
- Longest-Common-Subseq.
- Edit-Distance
- Subset Sum
- Knapsack
- Longest Ascending Subseq.

Graph Definitions
- Definitions
- Eulerian Tours
- Topological Sorting

Graph Searchs
- DFS
- BFS

Shortest Paths

one-to-all
- BFS usage
- Dijkstra
- Bellman-Ford
- negative-closed w. detection

all-to-all
- one-to-all usage
- Floyd-Warshall
- Johnson
- #walks using A_G

- overview

MST
- Prim
- Boruvka
- Kruskal

Math Basis

- ↳ Asymp. Notation
- ↳ Induction
- ↳ Loop - Counting

Asymptotic Growth

$\mathbb{N}_0$: natural numbers with 0

Definitions, Theorems

Definition 1 (O-Notation). For $f : N \to \mathbb{R}^+$,

$$O(f) \coloneqq \{g : N \to \mathbb{R}^+ \mid \exists C > 0\, \forall n \in N\, g(n) \leq C \cdot f(n)\}.$$

$$\mathcal{O}(n^2) = \{n, log(n), \sqrt{n}, log^2(n), 1, 100n^2, \cdots\}$$

Theorem 1 (Theorem 1.1 from the script). *Let N be an infinite subset of $\mathbb{N}$ and $f : N \to \mathbb{R}^+$ and $g : N \to \mathbb{R}^+$.*

- *If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f \leq O(g)$, but $f \neq \Theta(g)$.*
- *If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C \in \mathbb{R}^+$, then $f = \Theta(g)$.*
- *If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, then $f \geq \Omega(g)$, but $f \neq \Theta(g)$.*

Theorem 1 (L'Hôpital's rule). *Assume that functions $f : \mathbb{R}^+ \to \mathbb{R}^+$ and $g : \mathbb{R}^+ \to \mathbb{R}^+$ are differentiable, $\lim_{x\to\infty} f(x) = \lim_{x\to\infty} g(x) = \infty$ and for all $x \in \mathbb{R}^+$, $g'(x) \neq 0$. If $\lim_{x\to\infty} \frac{f'(x)}{g'(x)} = C \in \mathbb{R}_0^+$ or $\lim_{x\to\infty} \frac{f'(x)}{g'(x)} = \infty$, then*

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}.$$

Definition 1 (Ω-Notation). For $f : N \to \mathbb{R}^+$,

$$\Omega(f) \coloneqq \{g : N \to \mathbb{R}^+ \mid f \leq O(g)\}.$$

We write $g \geq \Omega(f)$ instead of $g \in \Omega(f)$.

Definition 2 (Θ-Notation). For $f : N \to \mathbb{R}^+$,

$$\Theta(f) \coloneqq \{g : N \to \mathbb{R}^+ \mid g \leq O(f) \text{ and } f \leq O(g)\}.$$

We write $g = \Theta(f)$ instead of $g \in \Theta(f)$.

In other words, for two functions $f, g : N \to \mathbb{R}^+$ we have

$$g \geq \Omega(f) \Leftrightarrow f \leq O(g)$$

and

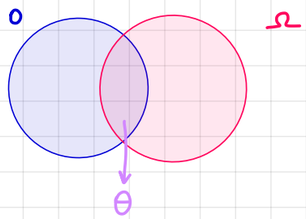
$$g = \Theta(f) \Leftrightarrow g \leq O(f) \text{ and } f \leq O(g).$$

Ω : $\sqrt{n} \leq O(n)$

$n \geq \Omega(\sqrt{n})$

Θ : $n \cdot \ln n = \Theta(\ln(n!))$

$\Rightarrow n \cdot \ln n \leq O(n!) \wedge n \cdot \ln n \geq \Omega(n!)$



Mini Cheat Sheet

$\lim_{n\to\infty}$: $1 < \log(\log(n)) < \log(n) < \sqrt{n} < n < n \cdot \log(n) < n \cdot \sqrt{n} < n^2 < 2^n < n! < n^n$

$n^x < x^n$ (x being fixed)

Sums

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots n^2 = \frac{n(n+1)(2n+1)}{6}$$

Geometric series : $\sum_{k=0}^{n} q^k = \frac{q^{n+1} - 1}{q - 1}$

$$\sum_{k=0}^{3} 3^k = 3^0 + 3^1 + 3^2 + 3^3 = \frac{3^4 - 1}{3 - 1} = 40$$

Factorial

$$\frac{n}{2}^{\frac{n}{2}} \leq n! \leq n^n$$

From Exercise Sheet 1 :

$\sum_{i=1}^{n} i^k \leq n^{k+1}$

$\sum_{i=1}^{n} i^k \geq \frac{1}{2^{k+1}} \cdot n^{k+1}$

$\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$

Θ : $n \cdot \ln n = \Theta(\ln(n!))$

$\Rightarrow n \cdot \ln n \leq O(n!) \wedge n \cdot \ln n \geq \Omega(n!)$

Some Tricks

ex (trick): (more for O-Notation)

$\lim_{n\to\infty} 2^n < n!$

$\log(2^n) < \log(n!) = \log(1 \cdot 2 \cdot 3 \cdot \ldots \cdot n)$

$n \cdot \log(2) < \log(1) + \log(2) + \ldots + \log(n)$

trick: $x^{100} = (e^{\ln(x)})^{100} = e^{\ln(x) \cdot 100}$

ex(trick):

$\frac{g(n)}{f(n)} = \frac{0}{0}$ → cannot apply d'Hôpital ?

$\frac{\frac{1}{g(n)}}{\frac{1}{f(n)}} = \frac{\infty}{\infty} \overset{\text{d'Hôpital}}{=} 0$

$\frac{1}{f(n)}$ faster — $f(n)$ slower — $g(n)$ grows faster ★

▸ **Converting exponentials from base b to base e**

Q3: How do we convert b^x to $e^{(\text{something})}$?

A3: Using $b = e^{\ln b}$ we have the **conversion formula:** $\boxed{b^x = (e^{\ln b})^x = e^{(\ln b) \cdot x}}$.

Example 3 Rewrite $\sqrt[3]{7}$ as an exponential with base e.

$$\sqrt[3]{7} = 7^{\frac{1}{3}} = \left(e^{\ln 7}\right)^{\frac{1}{3}} = e^{\frac{1}{3}\ln 7}$$

Why do the bases of log don't matter for the asymptotic growth?

- The general idea is that different bases for the logarithm change the function only by a constant
  - The log_a(n) and log_b(n) are within constant factors of each other
  - So they don't change the asymptotic behavior
    - If they're not used as an exponent or so (But there ^3 and ^4 are also different)
- Why exactly ?
  - Changing the base $\log_b(x) = \frac{\log_a(x)}{\log_a(b)}$

Log- Rules :

$$\begin{aligned}
\log_a(bc) &= \log_a(b) + \log_a(c) \\
\log_a(b^c) &= c \log_a(b) \\
\log_a(1/b) &= -\log_a(b) \\
\log_a(1) &= 0 \\
\log_a(a) &= 1 \\
\log_a(a^r) &= r \\
\log_{1/a}(b) &= -\log_a(b) \\
\log_a(b) \log_b(c) &= \log_a(c) \\
\log_b(a) &= \frac{1}{\log_a(b)} \\
\log_{a^m}(a^n) &= \frac{n}{m}, \qquad m \neq 0
\end{aligned}$$

$$\log_b a = \frac{\log_d a}{\log_d b}$$

We use :

- log for $\log_2(x)$
- ln for $\log_e(x)$

Induction

Overview :

$\mathcal{M}$: depends on Q

Task: Prove via induction that for every positive integer n, we have (for $n \geq 5$)

a property { .... = ... / .... ≤ .... / .... ≥ ... }

(Hint : ....)

⚠ Base Case: Prove for $n = 1$, $(n = 5)$

Induction Hypothesis: Assume property holds for some k

⚠ Inductive Step: Show that property holds for $k+1$

Final Sentence: By the principle of mathematical induction, the property is true for every positive integer n. $(n \geq 5)$

a) Base Case: $n = 1$ $\quad 1 = \frac{1(1+1)}{2} = 1$ ✓

I.H.: for $k \in \mathbb{Z}^*$ $\quad 1+2+\ldots+k = \frac{k(k+1)}{2}$ ★ don't mix the chosen k with n

I.S. $\quad 1+2+\ldots+k+k+1 \overset{\text{I.H.}}{=} \frac{k(k+1)}{2} + k+1$ $\quad k \to k+1$

Should be equal to $\frac{(k+1)(k+1+1)}{2}$ with using correct steps

$$= \frac{k(k+1) + 2k+2}{2}$$

$$= \frac{(k+1)(k+2)}{2} = \frac{(k+1)(k+1+1)}{2}$$ ∎

By the principe of mathematical induction it's proven that the statement holds.

(usually) By the principle of mathematical induction, this is true for any positive integer n.

write down don't forget!

Loop Counting

Your solution consist of :

1. Exact number of times f is called
2. Maximal simplification of the expression in θ-notation

Theorem

Master theorem. The following theorem is very useful for running-time analysis of divide-and-conquer algorithms.

Theorem 1 (master theorem). *Let $a, C > 0$ and $b \geq 0$ be constants and $T : \mathbb{N} \to \mathbb{R}^+$ a function such that for all even $n \in \mathbb{N}$,*

$$T(n) \leq aT(n/2) + Cn^b. \tag{1}$$

Then for all $n = 2^k$, $k \in \mathbb{N}$,

- *If $b > \log_2 a$, $T(n) \leq O(n^b)$.*
- *If $b = \log_2 a$, $T(n) \leq O(n^{\log_2 a} \cdot \log n)$.*[1]
- *If $b < \log_2 a$, $T(n) \leq O(n^{\log_2 a})$.*

If the function T is increasing, then the condition $n = 2^k$ can be dropped. If (1) holds with "$=$", then we may replace O with Θ in the conclusion.

This generalizes some results that you have already seen in this course. For example, the (worst-case) running time of Karatsuba's algorithm satisfies $T(n) \leq 3T(n/2) + 100n$, so we have $a = 3$ and $b = 1 < \log_2 3$, hence $T(n) \leq O(n^{\log_2 3})$. Another example is binary search: its running time satisfies $T(n) \leq T(n/2) + 100$, so $a = 1$ and $b = 0 = \log_2 1$, hence $T(n) \leq O(\log n)$.

Either won't be used, or will be written in the task description

Exercise 3.3 *Counting function calls in loops* **(1 point)**.

For each of the following code snippets, compute the number of calls to f as a function of $n \in \mathbb{N}$. Provide **both** the exact number of calls and a maximally simplified asymptotic bound in Θ notation.

Algorithm 1

(a)
```
i ← 0
while i ≤ n do
    f()
    f()
    i ← i + 1
j ← 0
while j ≤ 2n do
    f()
    j ← j + 1
```

Solution:

This algorithm performs $\sum_{i=0}^{n} 2 + \sum_{j=0}^{2n} 1 = 2(n+1) + (2n+1) = 4n + 3 = \Theta(n)$ calls to f.

Algorithm 2

(b)
```
i ← 1
while i ≤ n do
    j ← 1
    while j ≤ i^3 do
        f()
        j ← j + 1
    i ← i + 1
```

Solution:

This algorithm performs $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4} = \Theta(n^4)$ calls to f.

Algorithm 4

(a)
```
i ← 1
while i ≤ n do
    j ← i
    while 2^j ≤ n do
        f()
        j ← j + 1
    i ← i + 1
```

Solution:

If $i \leq \lfloor \log_2 n \rfloor$, the inner loop performs $\sum_{j=i}^{\lfloor \log_2 n \rfloor} 1 = \lfloor \log_2 n \rfloor - i + 1$ calls to f. If $i > \lfloor \log_2 n \rfloor$, it performs none. The full algorithm thus performs $\sum_{i=1}^{\lfloor \log_2 n \rfloor} (\lfloor \log_2 n \rfloor - i + 1) = \lfloor \log_2 n \rfloor (\lfloor \log_2 n \rfloor + 1)/2 = \Theta((\log n)^2)$ calls to f.

Hint: *To find the asymptotic bound, it might be helpful to consider n of the form $n = 2^k$.*

Algorithm 5

(b)
```
function A(n)
    i ← 0
    while i < n^2 do
        j ← n
        while j > 0 do
            f()
            f()
            j ← j − 1
        i ← i + 1
    k ← ⌊n/2⌋
    for l = 0 . . . 3 do
        if k > 0 then
            A(k)
            A(k)
```

You may assume that the function $T : \mathbb{N} \to \mathbb{R}^+$ denoting the number of calls of the algorithm to f is increasing.

Hint: *To deal with the recursion in the algorithm, you can use the master theorem.*

Solution:

Given i, the innermost loop performs $\sum_{j=1}^{n} 2 = 2n$ calls to f. Hence, the second loop (guarded by $i < n^2$) performs $\sum_{i=0}^{n^2-1} 2n = 2n^3$ calls to f.
If $\lfloor \frac{n}{2} \rfloor = 0$ (i.e. $n = 1$), then $k = 0$, so the algorithm makes just 2 calls to f. Thus, we have $T(1) = 2$.
For $n \geq 2$, we have $k = \lfloor \frac{n}{2} \rfloor > 0$ and thus we get the following relation $T(n) = 2n^3 + 8T(\lfloor \frac{n}{2} \rfloor)$.
For even n, this relation is $T(n) = 2n^3 + 8T(\frac{n}{2})$. Hence, we can apply the master theorem with $a = 8$, $b = 3$ and $C = 2$. We get ($\log_2(8) = 3$) $T(n) = \Theta(n^3 \log(n))$ for any integer $n \geq 2$ (we need $n \geq 2$ so that $\log(n) > 0$) since T is increasing. In conclusion, the algorithm performs $\Theta(n^3 \log(n))$ calls to the function f.

Exercise 3.3 *Counting function calls in loops* **(1 point)**.

For each of the following code snippets, compute the number of calls to f as a function of $n \in \mathbb{N}$. Provide **both** the exact number of calls and a maximally simplified asymptotic bound in Θ notation.

Algorithm 1

(a)
```
i ← 0
while i ≤ n do
    f()
    f()
    i ← i + 1
j ← 0
while j ≤ 2n do
    f()
    j ← j + 1
```

$\sum_{i=0}^{n} 2 + \sum_{j=0}^{2n} 1 = (n-0+1)\cdot 2 + (2n-0+1)\cdot 1$

$= 2n+2+2n+1 = 4n+3 = \Theta(n)$

Algorithm 2

(b)
```
i ← 1
while i ≤ n do
    j ← 1
    while j ≤ i^3 do
        f()
        j ← j + 1
    i ← i + 1
```

$\sum_{i=1}^{n} \sum_{j=1}^{i^3} 1 = \sum_{i=1}^{n} (i^3 - 1 + 1)\cdot 1 = \sum_{i=1}^{n} i^3 \overset{!}{=} \frac{n^2\cdot(n+1)^2}{4} = \Theta(n^4)$

for the mini cheat sheet:

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

$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

$\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2 = \frac{n^2(n+1)^2}{4}$

Master theorem. The following theorem is very useful for running-time analysis of divide-and-conquer algorithms.

Theorem 1 (master theorem). *Let $a, C > 0$ and $b \geq 0$ be constants and $T : \mathbb{N} \to \mathbb{R}^+$ a function such that for all even $n \in \mathbb{N}$,*

$$T(n) \leq aT(n/2) + Cn^b. \tag{1}$$

Then for all $n = 2^k$, $k \in \mathbb{N}$,

- *If $b > \log_2 a$, $T(n) \leq O(n^b)$.*
- *If $b = \log_2 a$, $T(n) \leq O(n^{\log_2 a} \cdot \log n)$.*[1]
- *If $b < \log_2 a$, $T(n) \leq O(n^{\log_2 a})$.*

If the function T is increasing, then the condition $n = 2^k$ can be dropped. If (1) holds with "$=$", then we may replace O with Θ in the conclusion.

This generalizes some results that you have already seen in this course. For example, the (worst-case) running time of Karatsuba's algorithm satisfies $T(n) \leq 3T(n/2) + 100n$, so we have $a = 3$ and $b = 1 < \log_2 3$, hence $T(n) \leq O(n^{\log_2 3})$. Another example is binary search: its running time satisfies $T(n) \leq T(n/2) + 100$, so $a = 1$ and $b = 0 = \log_2 1$, hence $T(n) \leq O(\log n)$.

Exercise 4.1 *Applying the master theorem.*

For this exercise, assume that n is a power of two (that is, $n = 2^k$, where $k \in \mathbb{N}_0 := \mathbb{N} \cup \{0\}$).

(a) Let $T(1) = 1$, $T(n) = 4T(n/2) + 100n$ for $n > 1$. Using the master theorem, show that

$$T(n) \leq O(n^2).$$

Solution:

We can apply the master theorem with $a = 4$, $b = 1$ and $C = 100$. In this case, $b < \log_2 a$, and therefore we have $T(n) \leq O(n^{\log_2 4}) = O(n^2)$.

(b) Let $T(1) = 5$, $T(n) = T(n/2) + \frac{3}{2}n$ for $n > 1$. Using the master theorem, show that

$$T(n) \leq O(n).$$

Solution:

We can apply the master theorem with $a = 1$, $b = 1$ and $C = \frac{3}{2}$. In this case, $b > \log_2 a$, and therefore we have $T(n) \leq O(n^b) = O(n)$.

(c) Let $T(1) = 4$, $T(n) = 4T(n/2) + \frac{7}{2}n^2$ for $n > 1$. Using the master theorem, show that

$$T(n) \leq O(n^2 \log n).$$

Solution:

We can apply the master theorem with $a = 4$, $b = 2$ and $C = \frac{7}{2}$. In this case, $b = \log_2 a$, and therefore we have $T(n) \leq O(n^{\log_2 a} \cdot \log n) = O(n^2 \log n)$.

Searchs

- Linear Search
- Binary Search
- Lower-Bound

Binary Search

sortiert ★ iterativ

BINARY-SEARCH$(A = (A[1], \ldots, A[n]), b)$

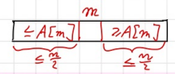
```
1 left ← 1; right ← n                          ▷ Initialer Suchbereich
2 while left ≤ right do
3     middle ← ⌊(left+right)/2⌋
4     if A[middle] = b then return middle      ▷ Element gefunden
5     else if A[middle] > b then right ← middle−1   ▷ Suche links weiter
6     else left ← middle+1                     ▷ Suche rechts weiter
7 return "Nicht vorhanden"
```

$T(n) \in O(\log n)$

$$T(n) = \begin{cases} c & \text{falls } n = 0 \text{ ist,} \\ T(n/2) + d & \text{falls } n \geq 1 \text{ ist,} \end{cases}$$

Fall 2: A sortiert, $A[1] \leq A[2] \leq \ldots \leq A[n]$

Algorithmus 2: Binäre Suche




```
Binary Search (A,b)  // für sortiertes A
l ← 1, r ← n   // A[l...r] gibt den Bereich an, in dem wir noch suchen müssen
while l ≤ r do
    m ← ⌊(l+r)/2⌋   // L...J, auch floor(...) geschrieben: abrunden
    if b = A[m]: return m
    if b < A[m]: r ← m-1
        else: l ← m+1
return „nicht gefunden"
```

Laufzeit (worst-case): $T(1) = c$ konstant

$T(n) \leq T(n/2) + d$, d Konstant

stillschweigend benutzt: Laufzeit wird höchstens größer für längere Arrays.

$\rightsquigarrow T(n) = O(\log n)$ geht nicht besser !

Binary Search

Input : sorted array , searched value

Output : the index of the search value (-1 if not found)

Runtime : O(logn) $T(n) \leq T(n/2) + d \leq d \log_2(n) + c.$

Pseudocode :

BINÄRE SUCHE **BINARY-SEARCH**$(A[1..n], b)$

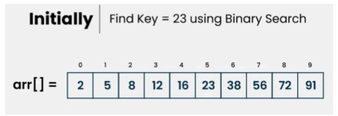
```
1 ℓ ← 1; r ← n                                 ▷ Initialer Suchbereich
2 while ℓ ≤ r do
3     m ← ⌊(ℓ + r)/2⌋
4     if A[m] = b then return m                ▷ Element gefunden
5     else if A[m] > b then r ← m − 1          ▷ Suche links weiter
6     else ℓ ← m + 1                           ▷ Suche rechts weiter
7 return "Nicht vorhanden"
```

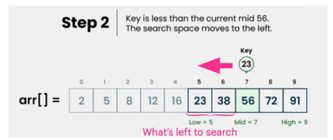
Illustration

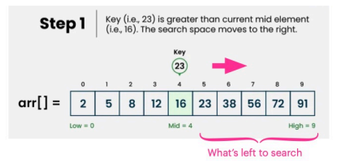
Code Example

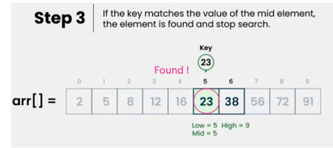
Binary Search

Illustration









Linear Search

Fall 1: A nicht sortiert

Algorithmus 1 (Lineare Suche):

```
LinearSearch (A,b)   // A = A[1...n] Array
for i = 1...n
    if A[i] = b: return i
return „nicht gefunden"
```

Laufzeit O(n)

Linear Search

Input : unsorted array , searched value

Output : the index of the search value (-1 if not found)

Runtime : O(n)

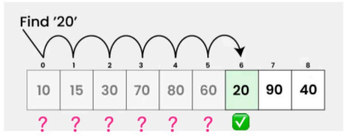
Pseudocode :

LINEAR-SEARCH$(A[1..n], b)$

```
1 for i ← 1, 2, ..., n do
2     if A[i] = b then return i
3 return "nicht gefunden"
```

Illustration

Code Example



Sorts

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quicksort
- Heapsort
- Lower-Bound

Bubble Sort — bubbles going up to the surface

Algo:

BubbleSort(A)

notwendig: ein einzelner Durchgang reicht nicht (siehe Beispiel)

```
for j = 1...n
    for i = 1...n-1
        if A[i] > A[i+1]
            tausche A[i] und A[i+1]
```

Bubble Sort — Idea: bubbles going up to the surface

Input: unsorted array — Output: sorted array

Runtime: $O(n^2)$ — #Comparisons: $O(n^2)$ — #Swaps: $O(n^2)$

Pseudocode:

Algorithm 1 Bubble Sort (input: array $A[1 \dots n]$).

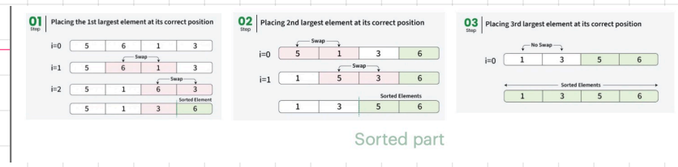
```
for j = 1, ..., n do
    for i = 1, ..., n − 1 do
        if A[i] > A[i + 1] then
            Swap A[i] and A[i + 1]
```

Bsp:

Beispiel:

j=1	j=2	j=3	j=4
5 3 7 1 4	3 5 1 4 7	3 1 4 5 7	1 3 4 5 7
3 5 7 1 4	3 5 1 4 7	1 3 4 5 7	1 3 4 5 7
3 5 7 1 4	3 1 5 4 7	1 3 4 5 7	1 3 4 5 7
3 5 1 7 4	3 1 4 5 7	1 3 4 5 7	1 3 4 5 7
3 5 1 4 7	3 1 4 5 7	1 3 4 5 7	1 3 4 5 7

7 in Bubble sind gewandert



Runtime:

Laufzeit: $O(n^2)$ Vergleiche

$O(n^2)$ Vertauschungen

Korrektheit Beweis:

Wie können wir beweisen/einsehen, dass BubbleSort korrekt ist?

Idee: Finde geeignete Invariante und beweise sie mit Induktion.

Invariante I(j): Nach j Durchgängen befinden sich die j größten Elemente am korrekten Ort

Beweis: Induktion (Übung)

Aus I(n) folgt: Nach n Durchläufen sind alle n Elemente korrekt.

Selection Sort — select the largest, put to the pos

Algo:

Idee: Finde größtes Element in $A[1..n-j+1]$ und tausche es mit $A[n-j+1]$

Algorithmus 2: Selection Sort (braucht $n-j \leq n$ Vergleiche)

```
for j = 1...n   // Alles ab A[n-j+2] schon korrekt
    K ← Index des Maximums in A[1...n-j+1]
    tausche A[K] mit A[n-j+1]
```

Selection Sort — Idea: select the largest, put to the pos

Input: unsorted array — Output: sorted array

Runtime: $O(n^2)$ — #Comparisons: $O(n^2)$ — #Swaps: $O(n)$

Pseudocode:

SELECTION-SORT($A[1..n]$)

```
1 for j ← n, n − 1, ..., 1 do
2     k ← Index des Maximums in A[1, ..., j]
3     tausche A[k] und A[j]
```

Bsp:

Beispiel: 5 3 7 1 4 1

I(0)
5 3 4 1 7 | I(1)
1 3 4 5 7 | I(2)
1 3 4 5 7 | I(3)
1 3 4 5 7 | I(4)
1 3 4 5 7 | I(5) ⇒ sortiert

Runtime:

Laufzeit: $O(n^2)$ Vergleiche $\sum_{j=0}^{n-1}(n-j-1) = \Theta(n^2)$

$O(n)$ Vertauschungen ← besser als Bubblesort

Insertion Sort — insert next to the correct position, "verschiebe" rest

Algo:

Algorithmus 3 (Insertion Sort)

```
InsertionSort(A)
for j = 2...n   // A[1..j-1] schon sortiert
    Finde Stelle k, an die A[j] in A[1..j-1] gehört   (geht mit binärer Suche in Zeit O(log(j-1)))
        K ← j, falls A[j] = max{A[1],...,A[j]}
        sonst K ← kleinster Index k mit A[j] ≤ A[k]
    x ← A[j]   // merke A[j], da es gleich überschrieben wird
    verschiebe A[k...j-1] nach A[k+1...j]
    A[k] ← x
```

aufwendig, braucht j−k Operationen (worst case: j−1)

Insertion Sort — Idea: insert curr to the correct position, "verschiebe", shift rest

Input: unsorted array — Output: sorted array

Runtime: $O(n^2)$ — #Comparisons: $O(n \log n)$ — #Swaps: $O(n^2)$

K can be found with binary search

Pseudocode:

INSERTION-SORT($A[1..n]$)

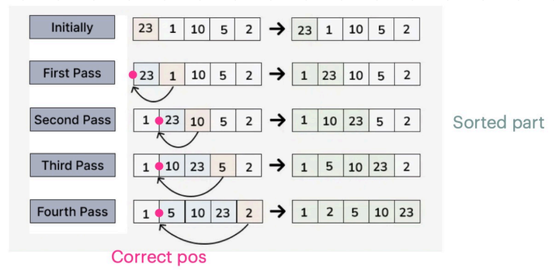
```
1 for j ← 2, 3, ..., n do
2     k ← kleinster Index in {1, ..., j − 1} mit A[j] ≤ A[k]
          // A[j] gehört an diese Stelle k
3     x ← A[j]
4     verschiebe A[k, ..., j − 1] nach A[k + 1, ..., j]
5     A[k] ← x
```

Illustration

Bsp:

Beispiel: 5 3 7 4 1

I(1)
3 5 7 4 1 | I(2)
3 5 7 4 1 | I(3)
3 4 5 7 1 | I(4)
1 3 4 5 7 | I(5) ⇒ sortiert



Runtime:

Laufzeit: Vergleiche $\leq \sum_{j=2}^{n} c \cdot \log(j-1) \leq \sum_{j=2}^{n} c \cdot \log n \leq O(n \cdot \log n)$

Vertauschungen $\leq \sum_{j=2}^{n}(j-1) = O(n^2)$

Merge Sort Divide and conquer

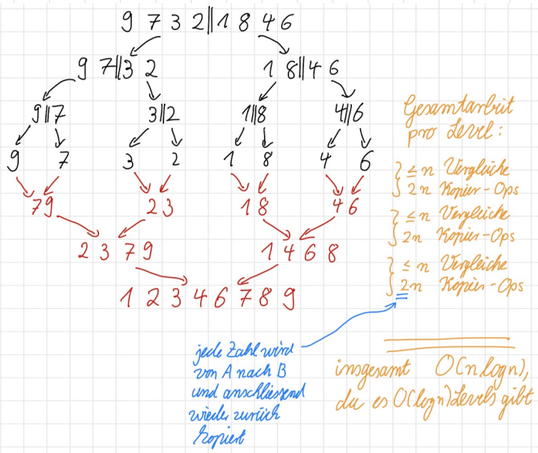
Algo:

```
MergeSort(A, l, r)  // sortiert den Bereich A[l...r]
  if l < r
      m ← ⌊ (l+r)/2 ⌋
      MergeSort(A, l, m)   // sortiere linke Hälfte
      MergeSort(A, m+1, r) // sortiere rechte Hälfte
      Merge(A, l, m, r)    // verschmelze beide Hälften

Merge(A, l, m, r)
  B ← new Array mit r-l+1 Zellen  // selbe Grösse wie A[l...r]
  i ← l   // erstes unbenutztes Element in linker Hälfte
  j ← m+1 // erstes unbenutztes Element in rechter Hälfte
  k ← 1   // nächste Position in B
  while i ≤ m and j ≤ r  // beide Hälften noch nicht ausgeschöpft
      if A[i] < A[j]
          B[k] ← A[i]
          i ← i+1
          k ← k+1
      else
          B[k] ← A[j]
          j ← j+1
          k ← k+1
  übernimm Rest links bzw. rechts  // wenn die andere Hälfte ausgeschöpft ist
  kopiere B nach A[l...r]
```

Bsp:

Beispiel:



Runtime:

Alternative Bestimmung der Laufzeit:

$T(1) \le c$, für $n = 2^k$: $T(n) \le 2 \cdot T(n/2) + d \cdot n$

Induktion $T(n) \le d \cdot n \cdot \log_2 n + c \cdot n$ (Übung!)

$\le O(n \log n)$

Beste Laufzeit bisher: $O(n \log n)$

Nachteil: braucht Zusatzarray (nicht „in place")

Merge Sort

Idea : Divide and Conquer

Input : unsorted array Output : sorted array

Runtime: O(nlogn)

#Comparisons: O(nlogn)

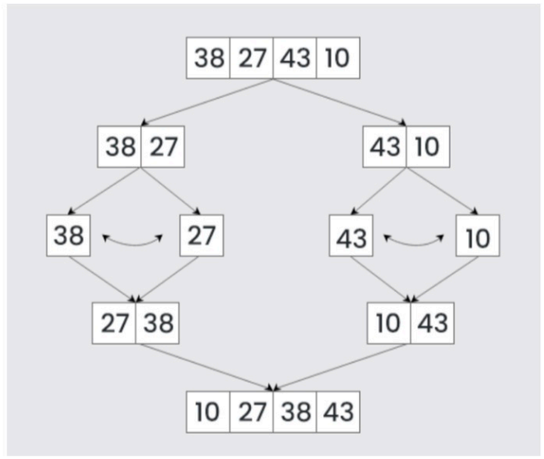
#Swaps: O(nlogn)

Pseudocode :

```
MERGE(A[1..n], l, m, r)
 1  B ← new Array with r − l + 1 cells        ▷ so gross wie A[l, ..., r]
 2  i ← l                                     ▷ erstes unbenutztes Element in linker Hälfte
 3  j ← m + 1                                 ▷ erstes unbenutztes Element in rechter Hälfte
 4  k ← 1                                     ▷ nächste Position in B
 5  while i ≤ m and j ≤ r do                  ▷ beide Hälften noch nicht ausgeschöpft
 6      if A[i] < A[j] then
 7          B[k] ← A[i]
 8          i ← i + 1
 9          k ← k + 1
10      else
11          B[k] ← A[j]
12          j ← j + 1
13          k ← k + 1
14  übernimm Rest links bzw. rechts           ▷ wenn die andere Hälfte ausgeschöpft ist
15  kopiere B nach A[l, ..., r]
```

Illustration

Code Example

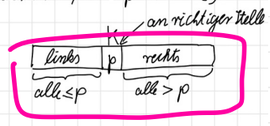


Quicksort PIVOT

Idee:

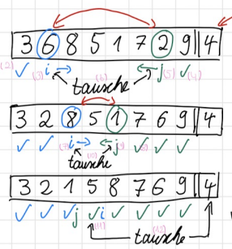
idee:
1) Wähle ein Element p ("Pivotelement", z.B. p := A[n])
2) Finde korrekte Position für p
3) Schaffe alle Elemente ≤ p nach links und alle Elemente > p nach rechts
4) Sortiere rekursiv linken und rechten Teil

Verschmelzen nicht notwendig



Aufteilen: Wähle Pivot, z.B. p = A[r]

ex:



Bewege i nach rechts und j nach links, bis ein Paar zum Tauschen findet

Am Ende (wenn i > j)
tausche Pivot an richtige Stelle i = j+1

Algo:

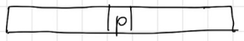
```
QuickSort(A, l, r)   // sortiere A[l...r]
  if l < r
    k = Aufteilen(A, l, r)   // 1-3
                             // räumt A[l...r] so um, dass
                             // A[i] ≤ A[k] für i = l...k-1
                             // A[i] > A[k] für i = k+1...r
    QuickSort(A, l, k-1)
    QuickSort(A, k+1, r)
```

```
Aufteilen(A, l, r)   // l < r      Laufzeit von Aufteilen ≤ O(r-l) ≤ O(n)
  i := l
  j := r-1
  p := A[r]
  repeat                          (is like while(i<j))
    while i < r and A[i] ≤ p do i := i+1     // i++ bis Element in falschen Seite gefunden
    while j ≥ l and A[j] > p do j := j-1     // j-- bis Tauschkandidat an der rechten Seite gefunden
    if i < j: tausche A[i] mit A[j]
  until i > j
  tausche A[i] mit A[r]   // am Ende ist i die richtige Stelle für Pivot
  return i
```

für den Extremfall, dass p das Maximum oder Minimum in A[l..r] ist

Runtime:

Laufzeit: Rekursion hängt davon ab, wo Pivot landet

gut: 

$T(n) \leq 2 \cdot T(\frac{n}{2}) + cn \leq O(n \log n)$

schlecht: 

$T(n) = T(n-1) + cn \leq O(n^2)$

2.B. wenn A schon sortiert war.

→ Lösung: p randomisiert wählen

Quick Sort

Idea : No merging, Pivot !!!

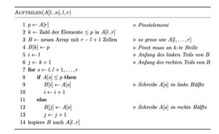
Input : unsorted array Output : sorted array

Runtime: Depends on the pivot !

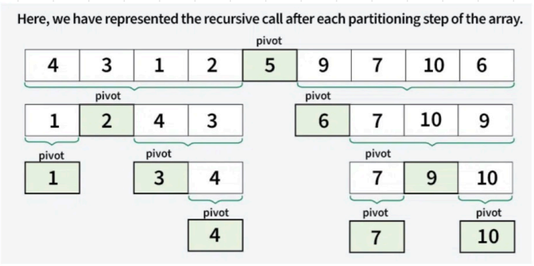
when the pivot element divides the array into two equal halves : O(nlogn)

when the smallest or largest element is always chosen as the pivot : (e.g., sorted arrays). $O(n^2)$

Pseudocode :



Illustration



Heapsort

```
HeapSort(A)
  H = empty Heap
  for i = 1...n: insert(H, A[i])
  for i = n...1: A[i] ← ExtractMax(H)
```

Laufzeit: n × insert ⇝ O(n · log n)
n × ExtractMax ⇝ O(n · log n)
} insgesamt O(n log n)

table max, and extract, restrore heap c.

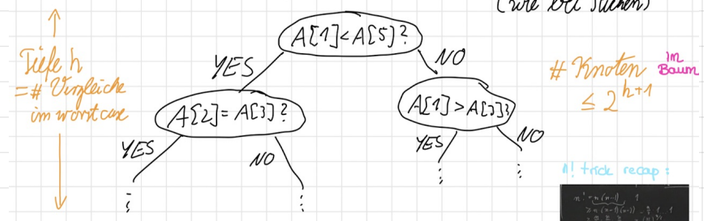
Heap Bedingung: key(Knoten) ≥ key(Children)

Lower Bound for Sorting

GEHT ES BESSER? LOWER BOUND OF SORTS ≤ ? zeigen

Antwort: NEIN (Annahme: vergleichsbasiertes Sortieren)

Beweis: Betrachte Entscheidungsbaum für einen beliebigen Algorithmus (wie bei Suchen)

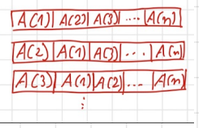


Tiefe h = # Vergleiche im worst case

Knoten im Baum ≤ 2^{h+1}

#! trick recap:

Es muss gelten: # Knoten ≥ # mögliche Output = n!
Umordnungen von A, z.B.



$\rightsquigarrow 2^{h+1} \geq \#\text{Knoten} \geq n!$ | log
$\rightsquigarrow h \geq \log_2(n!) - 1 \geq \Omega(n \log n)$
$h+1 \geq \log_2(n!) = \Omega(n \log n)$

worst case laufzeit $\geq h \geq \log_2(n!) - 1 \geq \Omega(n \log n)$

HeapSort, MergeSort und (randomisiertes) QuickSort haben Laufzeit O(n log n)

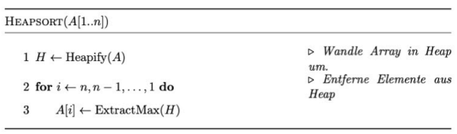
Heap Sort

Idea : Selection sort + Heap ! (Finding maximum faster)

Input : unsorted array Output : sorted array

Runtime: O(nlogn)

Pseudocode :



HEAPSORT(A[1..n])

1 $H \leftarrow$ Heapify(A) ▷ Wandle Array in Heap um.
2 **for** $i \leftarrow n, n-1, \ldots, 1$ **do** ▷ Entferne Elemente aus Heap
3 $A[i] \leftarrow$ ExtractMax(H)

Algorithmen bisher:

	Vergleiche	Bewegungen	Extra-Platz	Lokalität
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(1)$	gut
Selection Sort	$O(n^2)$	$O(n)$	$O(1)$	gut
Insertion Sort	$O(n \log n)$	$O(n^2)$	$O(1)$	gut
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	gut

in der Regel am wichtigsten

kann verbessert werden, ist aber kompliziert

Algorithmen springen nicht wild im Speicher hin und her

Erinnerung:

Merge Sort: teile Array

Exercise 5.1 *Sorting algorithms.*

Below you see four sequences of snapshots, each obtained in consecutive steps of the execution of one of the following algorithms: `InsertionSort`, `SelectionSort`, `QuickSort`, `MergeSort`, and `BubbleSort`. For each sequence, write down the corresponding algorithm.

Insertion Sort

3	6	5	1	2	4	8	7
3	6	5	1	2	4	8	7
3	5	6	1	2	4	8	7

Bubble Sort

3	6	5	1	2	4	8	7
3	5	1	2	4	6	7	8
3	1	2	4	5	6	7	8

Merge Sort

3 |	6 |	5 |	1	2 |	4 |	8 |	7
3	6 |	1	5	2	4 |	7	8
1	3	5	6	2	4	7	8

Selection Sort

3	6	5	1	2	4	8	7
3	6	5	1	2	4	7	8
3	6	5	1	2	4	7	8

Algorithmus	Vergleiche	Bewegungen	Extr. Platz	Lokalität
Bubble-Sort	$O(n^2)$	$O(n^2)$	$O(1)$	gut
Selection-Sort	$O(n^2)$	$O(n)$	$O(1)$	gut
Insertion-Sort	$O(n \log n)$	$O(n^2)$	$O(1)$	gut
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n)$	gut

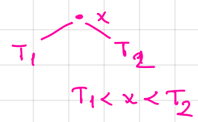
Data Structures

- Array
- List (Linked, Doubly)
- Trees (BST, Heap Tree, AVL Tree)

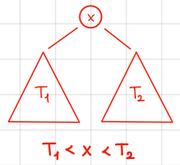
Search Trees

BST

Suchbaum Bedingung



$T_1 < x < T_2$



$T_1 < x < T_2$

Search (x, p) // Suche in Teilbaum unter p
  if p = null : Misserfolg
  else if p.key = x : Erfolg
  else
    if x < p.key : Search(x, p.left)
    else : Search(x, p.right)

→ O(h)

Insert (x): Search (x)
Ersetze null-Blatt durch [null x null] //falls nicht gefunden

remove(x) : erst search(x), dann Fallunterscheidung:

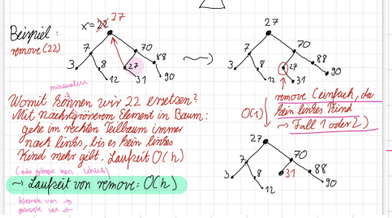
Fall 1: x ist Blatt, ✓

Fall 2: x hat ein Kind ✓

Fall 3: x hat zwei Kinder

→ remove w : minimal key in $T_r(v)$: symetric successor of v
v, w swap keys
delete w

Beispiel: remove(22)



Womit können wir 22 ersetzen? Mit nachstgrößerem Element im Baum: gehe im rechten Teilbaum immer nach links, bis es kein linkes Kind mehr gibt. Laufzeit O(h)

→ Laufzeit von remove: O(h)

→ laufe einmal nach rechts danach solange dem linken Nachfolger folgt, bis dieser ein Blatt als linken N. besitzt
(symetric predecessor geht auch)
größte key in $T_l(v)$

SymmetricSuccessor(v)

```
1 w ← v.Right                    ▷ Gehe genau einmal nach rechts
2 x ← w.Left                     ▷ Folge danach dem linken Nachfolger
3 while x ist kein Blatt do w ← x; x ← w.Left
4 return w                       ▷ w ist symmetrischer Nachfolger von v
```

remove: worst case O(h)

Beweis. Der zu entfernende Knoten muss zunächst gefunden werden, was in Zeit $\mathcal{O}(h)$ möglich ist. Danach tritt einer der Fälle 1 bis 3 auf. In den ersten beiden Fällen werden lediglich Zeiger verändert und Speicher freigegeben. Daher fällt für sie nur Zeit $\mathcal{O}(1)$ an. Im dritten Fall muss der zunächst mithilfe des Algorithmus SymmetricSuccessor der symmetrische Nachfolger gefunden werden. Dies kostet maximal Zeit $\mathcal{O}(h)$, da die Pfadlänge von der Wurzel bis zu einem Blatt höchstens h beträgt. Danach werden lediglich die Schlüssel vertauscht und der symmetrische Nachfolger gelöscht, was in Zeit $\mathcal{O}(1)$ möglich ist. Damit wird im dritten Fall maximal Zeit $\mathcal{O}(h)$ benötigt. ∎

Höhe des Baums T: T ein Blatt h(T) = 0

Wurzel v h(T) = 1 + max { h(T_l(v)), h(T_r(v)) }

Search(k)

```
1 v ← Root
2 while v ist kein Blatt do
3     if k = v.Key then return true        ▷ Element gefunden
4     else if k < v.Key then v ← v.Left    ▷ Suche links weiter
6     else v ← v.Right                     ▷ Suche rechts weiter
7 return false                             ▷ k nicht gefunden
```

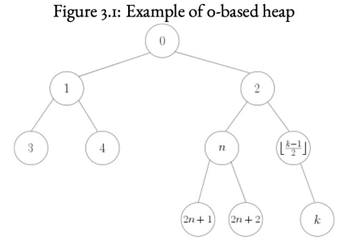
Schlüssel suchen

worst-case: O(h)

Heap Tree

If A is the parent of B, then val(A) < val(B)
(binary min-heaps)

Figure 3.1: Example of 0-based heap



Implementation

The most common implementation involves an array (of fixed or dynamic size). Assuming a 0-based array (cf. figure above), the *children* of node n are $2n+1$ and $2n+2$ and the *parent* of node k is node $\lfloor \frac{k-1}{2} \rfloor$.

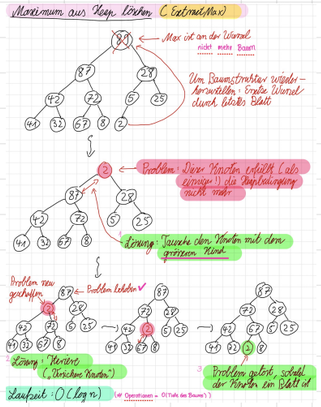
val(n) ≥ val(children(n))

ind. → val(Vorfahre) ≥ val(Nachkomme)
max is at the root

Algorithm 5 Restore heap invariant by percolating element down

```
function PercolateDown(H, i)                          ▷ Percolate element i in H
    e ← H[i]
    if 2i + 2 = H.length then
        if e > H[2i + 1] then
            Swap H[i] and H[2i + 1]
    else if 2i + 2 < H.length then
        l, r ← H[2i + 1], H[2i + 2]
        if l < r then
            if l < e then
                Swap H[i] and H[2i + 1]
                PercolateDown(H, 2i + 1)
        else
            if r < e then
                Swap H[i] and H[2i + 2]
                PercolateDown(H, 2i + 2)
```

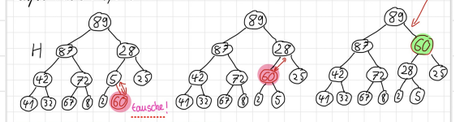
Maximum aus Heap löschen (ExtractMax)



insert(H, p) // füge einen neuen Knoten mit Schlüssel p in H ein
  Erzeuge neuen Knoten v mit Schlüssel p // Annahme: Wir kennen die Zahl Knoten in H
  an nächster freier Stelle
  Vertausche v so lange mit Elternknoten,
  bis die Heapbedingung erfüllt ist.

Heapbedingung erfüllt!

Bsp: Beispiel: insert(H, 60)

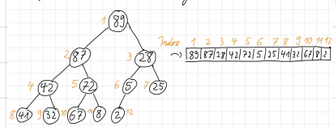


Laufzeit: O(log n) pro insert

HeapSort(A)
  H = empty Heap
  for i = 1 .. n : insert(H, A[i])
  for i = n .. 1 : A[i] ← ExtractMax(H)

Laufzeit: n × insert ⇝ O(n · log n)
          n × ExtractMax ⇝ O(n · log n)
} insgesamt O(n log n)

Wir speichern die Knoten in einem Array: erst Level 1, dann Level 2, ...



Beobachtung: Die Kinder von A[k] stehen in A[2k], A[2k+1] (solange 2k, 2k+1 ≤ n)

Beispiel: Testen der Heap-Bedingung für Knoten A[k]:

SatisfiesHeapCondition(k)
  if 2k > n return true // A[k] ist ein Blatt, Heapbedingung automatisch erfüllt
  if 2k = n and A[2k] ≥ A[k] return true // nur ein Kind
  if 2k < n and A[2k] ≥ A[k] and A[2k+1] ≥ A[k] return true
  return false

Operation	Binary	Binomial	Fibonacci
search min	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
delete min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
insert	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$
increase key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
union	$\Theta(m \log n)$	$O(\log n)$	$\Theta(1)$

AVL Tree is a BST

It's guaranteed that the tree is balanced

Balance factor

We define the following quantities:

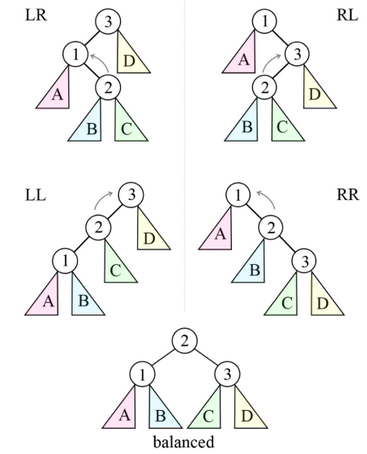
- depth(v) = distance of v to root (along unique path)
- height(T) = $\max_{v \in V}$ depth(v) + 1

Then, to each node v, we can assign a number b called the balance factor:

$$b(v) := \text{height}(v.R) - \text{height}(v.L)$$

↳ should be in range {-1, 0, 1}

Figure 3.2: All possible rotations and their next state



Θ(n) space O(log n) → worst time and average

zu 1: Satz: Ein AVL-Baum der Höhe h hat n ≥ Fib(h+2)−1 viele Knoten

Beweis: Induktion Fib: 1, 1, 2, 3, 5, ..

I-Anfang: h = 1: 0, n = 1 = Fib(3) − 1
h = 2: ○, ○○, n ≥ 2 = Fib(4) − 1

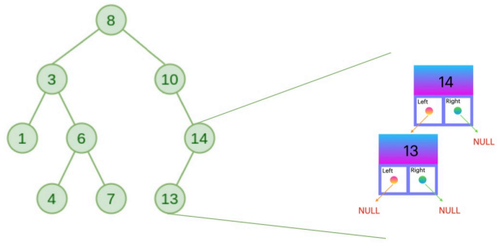
I-Schritt h ≥ 3

mindestens eines von $h_l(u), h_r(u) \geq h-1$
das andere $\geq h-2$ (AVL-Bed.)

⇝ n ≥ 1 + # Knoten links + # Knoten rechts
≥ (I-H.) 1 + Fib(h−1+2) − 1 + Fib(h−2+2) − 1
= Fib(h+2) − 1 ☑

Wir wissen Fib(h+2) ≥ $\frac{1}{3} \cdot 1.5^{h+2}$ (Übung 2)
⇝ Ein AVL-Baum der Höhe h hat n ≥ Fib(h+2) − 1 ≥ $\frac{1}{3} \cdot 1.5^{h+2} - 1$ Knoten
⇝ $1.5^{h+2} \leq 3 \cdot (n+1)$
⇝ $h+2 \leq \log_{1.5}(3 \cdot (n+1))$
⇝ h ∈ O(log n). → search in O(log n)

BST
Terminology



Root Node: The topmost node of the heap. Holds the maximum element !

Parent Node: A node that has one or more child nodes.

Child Node: A node directly connected to another node when moving away from the root.

Leaf Node: A node with no children (located at the bottom level).

Sibling Nodes: Nodes that share the same parent.

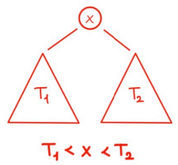
Level: The depth or layer of the node, where the root is at level 0.

Height: The longest path from the root node to a leaf.

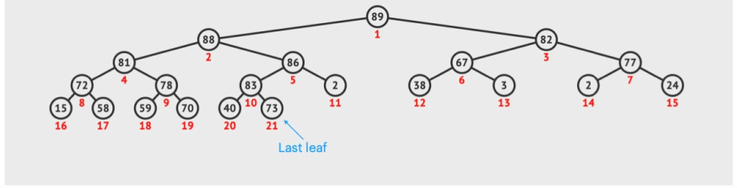
Height of the root = 1

BST Condition

Each node in a BST has at most two children, left child and a right child, with the left child containing values the parent node and the right child containing values greater than the parent node. Every node in the left subtree is less than the root, and every node in the right subtree is greater than the root.



Heap (here : Maxheap)
Terminology



Root Node: The topmost node of the heap. Holds the maximum element !

Parent Node: A node that has one or more child nodes.

Child Node: A node directly connected to another node when moving away from the root.

Leaf Node: A node with no children (located at the bottom level).

Sibling Nodes: Nodes that share the same parent.

Level: The depth or layer of the node, where the root is at level 0.

Height: The longest path from the root node to a leaf.

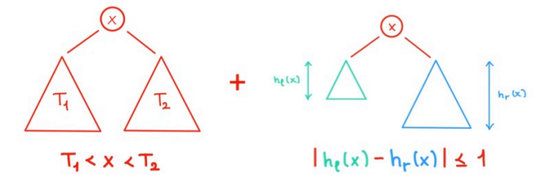
Heap Condition

For every node n in the heap, the value of the parent is greater than or equal to the value of n

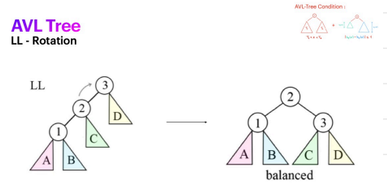
val(n) ≥ val(children(n) for all n

AVL Tree Condition

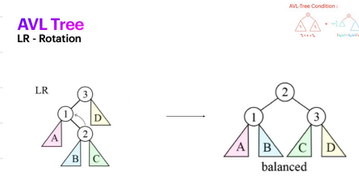
AVL tree is a self-balancing BST where the difference between heights of left and right subtrees cannot be more than one for all nodes.



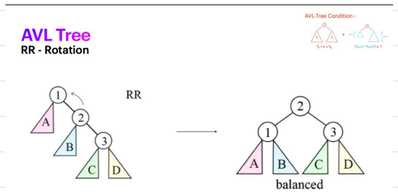
AVL Tree
LL - Rotation



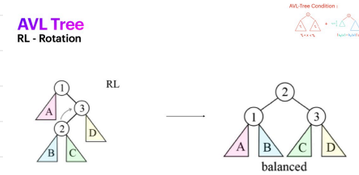
AVL Tree
LR - Rotation



AVL Tree
RR - Rotation



AVL Tree
RL - Rotation



Heap (here : Maxheap)
ExtractMax( )

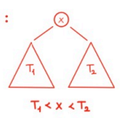
Heap Condition : val(n) ≥ val(children(n)

Max is at the root !

1. Swap the root with the last leaf
2. Swap the parent that does not satisfy the heap condition with the bigger child !
3. Repeat 2 until every node satisfies the heap condition !

BST
Search(x)

BST Condition :

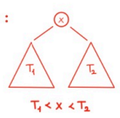


We use the BST condition

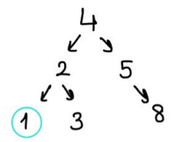
1. If curr == null , you're at the leaf and you haven't found x. Your search is done
2. If curr == x , you've found x !!
3. If curr < key , search the right subtree
4. If curr > key , search the left subtree
5. Repeat 3 and 4 starting from the root until you have one of the cases 1 and 2

BST
remove(x)

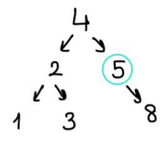
BST Condition :



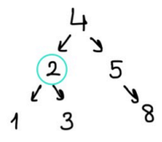
Case 1 : x has no children



Case 2: x has 1 child



Case 3 : x has 2 children



Heap (here : Maxheap)
insert( )

Heap Condition : val(n) ≥ val(children(n)

1. Place the node to the last free position
2. Swap the node with the parent, if it doesn't satisfy the heap condition
3. Repeat 2 until every node satisfies the heap condition !

Creating a heap means inserting one by one

Durchlaufordnungen für Bäume Sei T ein binärer Suchbaum mit der Wurzel v, dem linken Teilbaum $T_l(v)$ und dem rechten Teilbaum $T_r(v)$. Wir können die Knoten von T auf verschiedene Arten durchlaufen:

HAUPTREIHENFOLGE PREORDER

- In der *Hauptreihenfolge* (engl. *Preorder*) wird zunächst der in v gespeicherte Schlüssel ausgegeben. Danach wird zuerst rekursiv mit $T_l(v)$ fortgefahren und anschliessend rekursiv $T_r(v)$ verarbeitet.

NEBENREIHENFOLGE POSTORDER

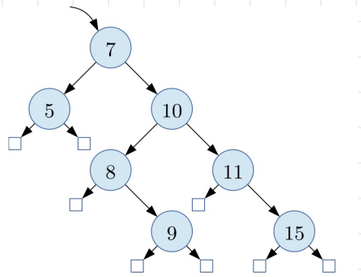
- In der *Nebenreihenfolge* (engl. *Postorder*) wird zunächst $T_l(v)$ rekursiv verarbeitet und danach $T_r(v)$. Schliesslich wird der in v gespeicherte Schlüssel ausgegeben.

SYMMETRISCHE REIHENFOLGE INORDER

- In der *symmetrischen Reihenfolge* (engl. *Inorder*) wird zunächst $T_l(v)$ rekursiv besucht, danach der in v gespeicherte Schlüssel ausgegeben und schliesslich $T_r(v)$ rekursiv besucht. Da alle Schlüssel in $T_l(v)$ kleiner und alle Schlüssel in $T_r(v)$ grösser sind als der in v gespeicherte Schlüssel, gibt die symmetrische Reihenfolge die in T gespeicherten Schlüssel in sortierter Reihenfolge aus.

Beispiel Die Knoten des binären Suchbaums aus Abbildung 4.13 werden in den folgenden Reihenfolgen durchlaufen:

- $7, 5, 10, 8, 9, 11, 15$ bei Verwendung der Hauptreihenfolge,
- $5, 9, 8, 15, 11, 10, 7$ bei Verwendung der Nebenreihenfolge,
- $5, 7, 8, 9, 10, 11, 15$ bei Verwendung der symmetrischen Reihenfolge.



Zusammenfassung und Ausblick Sei T ein binärer Suchbaum mit Höhe h, der n Schlüssel verwaltet. Binäre Suchbäume implementieren die Wörterbuchoperationen in Zeit $\mathcal{O}(h)$. Ausserdem werden eine Reihe weiterer Operationen unterstützt, zum Beispiel:

MINIMALER SCHLÜSSEL

- MIN(T): Liefert den Schlüssel aus T mit minimalem Wert zurück. Diese Operation kann in Zeit $\mathcal{O}(h)$ realisiert werden, indem ausgehend von der Wurzel von T so lange dem linken Nachfolgerknoten gefolgt wird, bis dieser ein Blatt als linken Nachfolger besitzt.

MINIMUM EXTRAHIEREN

- EXTRACT-MIN(T): Sucht und entfernt den Schlüssel mit minimalem Wert aus T. Auch diese Operation kann in Zeit $\mathcal{O}(h)$ durchgeführt werden.

SCHLÜSSEL AUSGEBEN

- LIST(T): Liefert eine sortierte Liste der in T gespeicherten Schlüssel zurück. Diese Funktion wird von einem Durchlauf in symmetrischer Reihenfolge in Zeit $\mathcal{O}(n)$ realisiert.

VEREINIGUNG

- JOIN(T_1, T_2): Seien T_1 und T_2 zwei Suchbäume zu den disjunkten Schlüsselmengen $\mathcal{K}_1$ und $\mathcal{K}_2$, und zusätzlich der maximale Wert eines Schlüssels in $\mathcal{K}_1$ kleiner als der minimale Wert eines Schlüssels in $\mathcal{K}_2$. JOIN(T_1, T_2) berechnet einen binären Suchbaum zur Schlüsselmenge $\mathcal{K}_1 \cup \mathcal{K}_2$. Dazu führen wir zunächst EXTRACT-MIN(T_2) aus, erhalten einen Schlüssel k sowie den aus T_2 resultierenden Baum T_2'. Dann erzeugen wir einen neuen Baum, dessen Wurzel den Schlüssel k speichert und die Teilbäume T_1 sowie T_2' besitzt. Die Laufzeit beträgt $\mathcal{O}(h)$.

Mindestblattzahl:

Jeder binäre Suchbaum (und damit auch jeder AVL-Baum), der n Schlüssel speichert, hat genau n + 1 Blätter

MINDESTBLATTZAHL

Verankerung I (AVL-Bäume mit Höhe $h = 1$): Es gibt nur einen AVL-Baum mit Höhe 1, nämlich denjenigen, der genau einen Schlüssel speichert und zwei Blätter besitzt (siehe Abbildung 4.18(a)). Die Mindestblattzahl eines AVL-Baums mit Höhe 1 beträgt daher $MB(1) = 2$.

Verankerung II (AVL-Bäume mit Höhe $h = 2$): Es gibt genau drei AVL-Bäume mit Höhe 2. Zwei davon verwalten genau zwei Schlüssel, von denen einer in der Wurzel des Baums und der andere im linken bzw. im rechten Nachfolger der Wurzel gespeichert ist (siehe Abbildung 4.18(b)). Ausserdem gibt es noch einen Baum mit einer Wurzel und jeweils genau einem linken und einem rechten Nachfolger. Da jeder dieser Bäume mindestens drei Blätter hat, beträgt die Mindestblattzahl eines AVL-Baums mit Höhe 2 genau $MB(2) = 3$.

Rekursion (AVL-Bäume mit Höhe $h \geq 3$): Betrachte einen beliebigen AVL-Baum mit Höhe h. Dieser besteht aus einem Wurzelknoten v mit einem linken Teilbaum $T_l(v)$ und einem rechten Teilbaum $T_r(v)$ (siehe Abbildung 4.17). Die Höhe dieser Teilbäume ist höchstens $h - 1$. Mindestens einer der Teilbäume muss Höhe $h - 1$ haben (hätten beide eine Höhe strikt kleiner als $h - 1$, dann wäre die Höhe des gesamten Baums strikt kleiner als h). Der andere Teilbaum hat entweder Höhe $h - 1$, oder Höhe $h - 2$, denn gemäss AVL-Bedingung dürfen sich die Höhen der Teilbäume höchstens um 1 unterscheiden. Da wir die *Mindestblattzahl* untersuchen, nehmen wir an, der andere Teilbaum hätte Höhe $h - 2$ (denn ein Baum mit Höhe $h - 1$ hat mit Sicherheit nicht weniger Blätter als ein Baum mit Höhe $h - 2$). Die Anzahl der Blätter des gesamten Baums ist nun die Summe der Blattanzahlen in den beiden Teilbäumen. Es gilt also $MB(h) = MB(h - 1) + MB(h - 2)$.

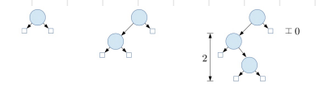


Abb. 4.18 Da sich bei AVL-Bäumen die Höhen der Teilbäume um maximal 1 unterscheiden dürfen, sind (a) und (b) Beispiele für AVL-Bäume, (c) hingegen nicht.

Die Rekursionsformel für die Mindestblattzahl erinnert an die der Fibonacci-Zahlen. Diese waren als

$$F_1 := 1, \quad F_2 := 1, \quad F_h := F_{h-1} + F_{h-2} \text{ für } h \geq 3 \tag{105}$$

definiert. Damit gilt offenbar die Beziehung $MB(h) = F_{h+2}$, und man kann zeigen, dass $MB(h) \in \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^h\right) \subseteq \Omega(1.6^h)$ gilt. Ein AVL-Baum mit Höhe h hat also mindestens 1.6^h viele Blätter, d.h., mindestens $1.6^h - 1$ viele Schlüssel. Im Umkehrschluss bedeutet dies, dass die Höhe eines AVL-Baums mit n Schlüsseln durch $1.44 \log_2 n$ nach oben beschränkt ist, also wie gehofft tatsächlich nur logarithmisch in der Anzahl der Schlüssel n ist. Das heisst also, ein AVL-Baum ist nur höchstens 44% höher als ein perfekt balancierter Binärbaum!

Insert: füge ein wie ein normalen ST

check the nodes for AVL-Bedingung

Die AVL-Bedingung kann höchstens bei den Knoten auf dem Weg von der Wurzel zum neu eingefügten Schlüssel verletzt sein. Bei allen anderen Knoten ist die Balance unverändert, folglich gilt die AVL-Bedingung nach wie vor.

Wir laufen also vom neu eingefügten Knoten zur Wurzel hoch und prüfen, ob die AVL-Bedingung noch immer gilt.

AVL-Tree (Bep)

Exercise 5.5 *AVL trees* **(1 point)**.

(a) Draw the tree obtained by inserting the keys 3, 8, 6, 5, 2, 9, 1 and 0 in this order into an initially empty AVL tree. Give also all the intermediate states after every insertion and before and after each rotation that is performed during the process.

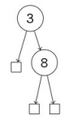
Exercise 5.5 *AVL trees* **(1 point)**.

(a) Draw the tree obtained by inserting the keys 3, 8, 6, 5, 2, 9, 1 and 0 in this order into an initially empty AVL tree. Give also all the intermediate states after every insertion and before and after each rotation that is performed during the process.

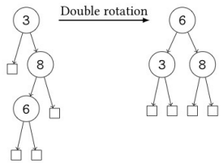
Solution:

Insert 3:

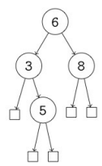
Insert 8:



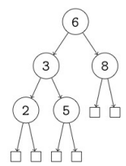
Insert 6:



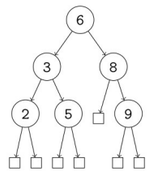
Insert 5:



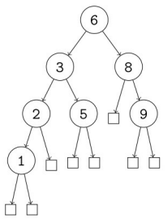
Insert 2:



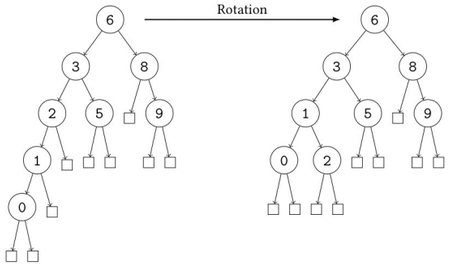
Insert 9:



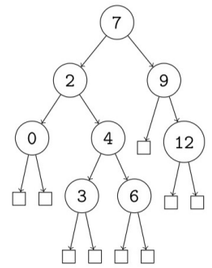
Insert 1:



Insert 0:

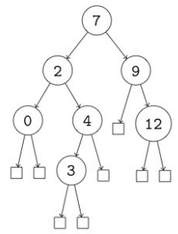


(b) Consider the following AVL tree.

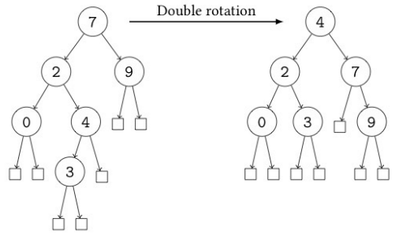


Draw the tree obtained by deleting 6, 12, 7 and 4 in this order from this tree. Give also all the intermediate states after every deletion and before and after each rotation that is performed during the process.

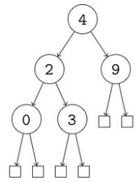
Delete 6 :



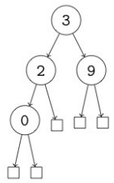
Delete 12:



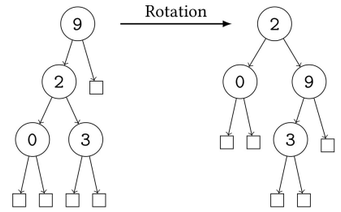
Delete 7:



Delete 4: Key 4 can either be replaced by its predecessor key, 3, or its successor key, 9. If key 4 is replaced by its predecessor:



If key 4 is replaced by its successor:



List

1. ADT Liste (Crash course, Details später in E Prog)

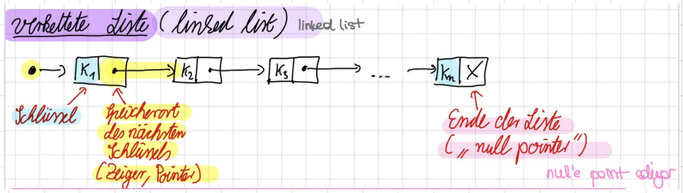
enthält Objekte / Schlüssel in fester Reihenfolge

Operationen (Auswahl):

- insert(K, L) // fügt Objekt mit Schlüssel K am Ende der Liste an
- get(i, L) // gibt i-ten Schlüssel aus
- delete(o, L) // löscht Objekt o aus Liste
- insertAfter(o, K, L) // fügt Objekt mit Schlüssel K hinter o ein

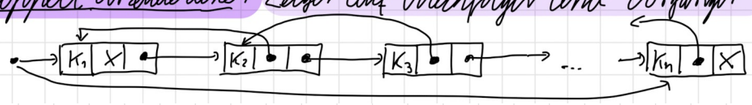
Linked List

verkettete Liste (linked list) linked list



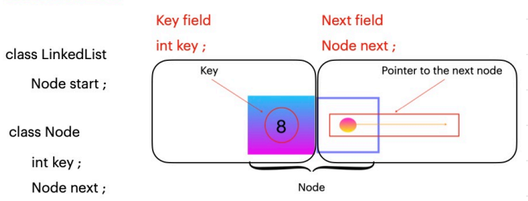
Doubly Linked List

doppelt verkettete Liste: Zeiger auf Nachfolger und Vorgänger

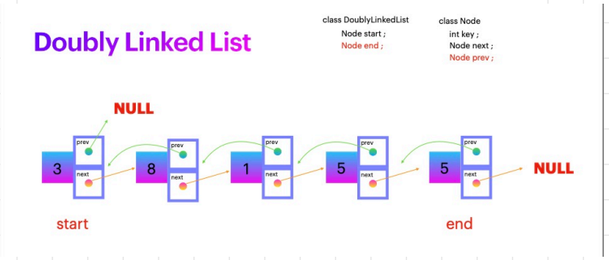


	insert	get	delete	insertAfter
Array	O(1)	O(1)	O(n)	O(n) (meist, schnell verschieben)
einfach verkettete Liste	O(n)	O(n)	O(n)	O(1)
doppelt verkettete Liste	O(1)	O(n)	O(1)	O(1)
Suchbäume	O(log n)	O(log n)	sofern Zeiger auf Objekt o bekannt O(log n)	O(log n)

Linked List



Doubly Linked List



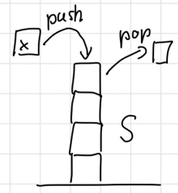
Stack

2. ADT Stapel (Stack)

push(x, S) : legt x auf Stapel S

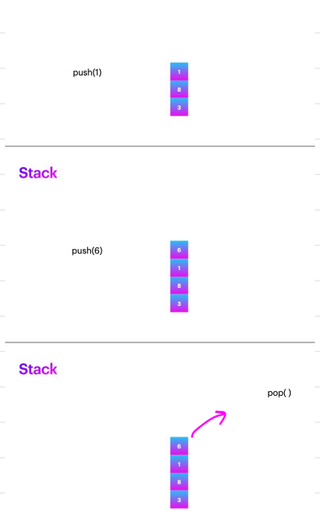
pop(S) : entfernt (und liefert) oberstes Element

top(x) : liefert oberstes Element



Datenstruktur: verkettete Liste ⇝ alle Ops in O(1)

Data structure	Search	Insert	Delete
Unsorted Array	$O(n)$	$O(1)$	$O(n)$
Sorted Array	$O(\log n)$	$O(n)$	$O(n)$
Unsorted List	$O(n)$	$O(1)$	$O(n)$
Sorted List	$O(n)$	$O(n)$	$O(n)$
Unbalanced Tree	$O(n)$	$O(n)$	$O(n)$
AVL tree	$O(\log n)$	$O(\log n)$	$O(\log n)$



unsortiertes Array: alle Ops O(n)

sortiertes Array: search O(log n), insert/remove O(n)

verkettete Liste: search/insert/remove O(n) (egal ob sortiert oder nicht)

Heap: search O(n)

Queue

3. ADT Schlange (Warteschlange, Queue)

enqueue(x, S): füge S hinten an

dequeue(S) : entferne (und liefere) vorderstes Element

Datenstruktur: z. B. doppelt verkettete Liste ⇝ beide Ops in O(1)

Priority Queue

4. ADT Prioritätsschlange (Priority Queue)

insert(x, P) füge x ein

extractMax(P) entferne (und liefere) Maximum

Datenstruktur: Heap ⇝ beide Ops in O(log n)

DP

- Fibonacci
- Max-Subarrey-Sum
- Jump-Game
- Longest-Common-Subseq.
- Edit--Distence.
- Subset Sum
- Knapsack
- Longest Ascending Subseq.

Ideas

Maximum Subarray Sum

Problem : find the subarray that has the maximum sum

Idea : Randmax $R_j = \max_{i \le j} S_{ij}$ where $S_{ij} = a_i + a_{i+1} + \dots + a_j$

Definition of the DP table : DP[i] = R_i

Computation of an entry :

Initialization : DP[0] = A[0]

Recursion : DP[i] = max{ a_i , $R_{i-1} + a_i$ }

New subarray starting with a_i | Adding a_i to the current subarray

Extracting the solution : The solution is max{ DP[i], 0 }

Longest Common Subsequence

Problem : Given two strings, A and B, the task is to find the length of the Longest Common Subsequence

A subsequence is a string generated from the original string by deleting 0 or more characters and without changing the relative order of the remaining characters.

Examples :	Inputs :	Outputs :	Subsequence :
	"ABC" and "ACD"	2	"AC"
	'AGGTAB" and "GXTXAYB"	4	"GTAB"
	"ABC" and "CBA"	1	"A" , "B" or "C"

Longest Common Subsequence

A subsequence is a string generated from the original string by deleting 0 or more characters and without changing the relative order of the remaining characters.

Idea : For every pair A[i] and B[j] there are exactly 3 options

- Use them in the subsequence
- Don't use A[i]
- Don't use B[i]

DP[0...n][0...m]

Definition of the DP table : DP[i][j] = LCS of A[0..i] and B[0..j]

Computation of an entry :

Initialization : DP[0][j] = 0 DP[i][0] = 0

Recursion :

$$DP[i][j] = \begin{cases} \max\{ DP[i-1][j-1] + 1 , DP[i-1][j] , DP[i][j-1] \} & \text{if } A[i] == B[j] \\ \max\{ DP[i-1][j] , DP[i][j-1] \} & \text{else} \end{cases}$$

use them in the subsequence | don't use A[i] | don't use B[j]

Extracting the solution : The solution is at DP[n][m]

Subset Sum

Problem : Given an array A , check if there's a subset of A s.t. it's sum is equal to a given number b.

Return true if we find I else false $I \subseteq \{1 \dots n\}$ s.t. $\sum_{i \in I} A[i] = b$

Examples :	Inputs :	Output :	what's used
	[1,2,3,4,5] , 1000	False	
	[1,2,3,4,5] , 10	True	2,3,5 or 1,2,3,4
	[] , 0	True	

Subset Sum

Idea : Two things can happen to each element

- It gets used in I
- It doesn't get used in I

DP[0...n][0...S]

Definition of the DP table : DP[i][s] = "Can I find a subset sum from A[0...i] that's equal to s "

Computation of an entry :

Initialization : DP[0][0] = True DP[i][0] = True DP[0][s] = False

Recursion :

$$DP[i][s] = DP[i-1][s] \ \| \ DP[i-1][s - A[i]]$$

we don't use i in I | we use i in I

Extracting the solution : The solution is at DP[n][S]

Jump Game

Problem : Given an array where each element represents the max number of steps that can be made forward from that index,find the minimum number of jumps to reach the end of the array starting from index 0.

Definition of the DP table : DP[i] = "Minimum number of jumps to reach i"

Computation of an entry :

Initialization : DP[0] = 0

Recursion : DP[i] = min { 1 + DP[j] | $1 \le j < i \ \wedge \ j + A[j] \ge i$ }

Extracting the solution : The solution is at DP[n-1]

Edit Distance

Problem : Given two strings A and B, find the minimum number of edits (operations) to convert A into B

Operations : Replace : Replace a character at any index of A with some other character

Insert : Insert any character into A

Remove : Remove a character of A

Examples :	Inputs :	Output :	Operations :
	"cat" and "cut"	1	replace a with u
	"sunday" and "saturday"	3	convert un to atur : replace n by r insert a, insert t

Edit Distance

Operations: Replace : Replace a character at any index of A with some other character

Insert : Insert any character after or before any index of A

Remove : Remove a character of A

Idea : For every element of A , 3 things can happen

- will be deleted
- B[j] gets inserted after
- will be replaced to match B[j]

Definition of the DP table : DP[i][j] = ED of A[0..i] and B[0..j] DP[0...n][0...m]

the minimum number of edits to convert A[0..i] into B[0..j]

Computation of an entry :

Initialization : DP[i][0] = i DP[0][j] = j

Recursion :

$$DP[i][j] = \begin{cases} DP[i-1][j-1] & \text{if } A[i] == B[j] \\ 1 + \min\{ DP[i-1][j] , DP[i][j-1] , DP[i-1][j-1] \} & \text{else} \end{cases}$$

add B[j] to the end | delete A[i] | replace A[i] with B[j]

Extracting the solution : The solution is at DP[n][m]

Knapsack

Problem :

Given : $\mathcal{W}$: weight limit w_i : weight of each item p_i : profit of each item

Searched : Maximum profit that one can have

Examples :	Inputs :	Output :	Explanation :
	$\mathcal{W} = 0$ $p = [1,2,3]$ $w = [5,5,5]$	0	all items are above weight limit
	$\mathcal{W} = 5$ $p = [1,2,3]$ $w = [5,5,5]$	3	we can only pick one item, and we pick the most profitable
	$\mathcal{W} = 10$ $p = [1,2,3]$ $w = [5,5,5]$	5	we can pick two items, and we pick 2+3 = 5

Knapsack

Idea : Two things can happen to each element

- We use it and get profit
- We don't use it

DP[0...n][0...W]

Definition of the DP table : DP[i][w] = "Maximum profit from A[0..i] with weight limit w"

Computation of an entry :

Initialization : DP[i][0] = 0

Recursion :

$$DP[i][w] = DP[i-1][w] \ \| \ p[i] + DP[i-1][w - w[i]]$$

we don't use i in I | we use i in I

Extracting the solution : The solution is at DP[n][W]

DP - Introduction

Consider the recurrence

$$A_1 = 1$$
$$A_2 = 2$$
$$A_3 = 3$$
$$A_4 = 4$$
$$A_n = A_{n-1} + A_{n-3} + 2A_{n-4} \text{ for } n \geq 5.$$

Compute A_n using bottom-up dynamic programming and state the run time of your algorithm. Address the following aspects in your solution:

(1) *Definition of the DP table:* What are the dimensions of the table $DP[\ldots]$? What is the meaning of each entry?

(2) *Computation of an entry:* How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries that do not depend on others.

(3) *Calculation order:* In which order can entries be computed so that values needed for each entry have been determined in previous steps?

(4) *Extracting the solution:* How can the final solution be extracted once the table has been filled?

(5) *Run time:* What is the run time of your solution?

0) Dimensions of the DP table: DP[1 ... n]

1) Definition of the DP table: DP[i] = A_i for $1 \leq i \leq n$

2) Computation of an entry:

Initialization: DP[1] = 1 , DP[2]=2 , DP[3]=3 , DP[4]=4

Recursion: DP[k] = DP[k-1] + DP[k-3] + 2·DP[k-4] for k≥5

3) Calculation Order:

We calculate with ~~increasing~~ i: now for (i=1 ... n)

4) Extracting Solution: The result is at: DP[n]

5) Runtime: Each entry can be computed in: $\Theta(1)$

We're filling a DP table of size: n

So the runtime is $\Theta(n)$

DP

Σ

elements a_i according to indexes i

$$I \subseteq \{1, \ldots, n\} \qquad \sum_{i \in I} a_i = A$$

indexes i from Index-set I

sum of the elements a_i

example:

$$\sum_{i \in I} a_i = A$$

A = 3

I = {1,2}

a_1	a_2	a_3	a_4	a_5	a_6	a_7
1	2	0	1	0	0	0

$a_1 + a_2 = 1 + 2 = 3 = A$

Longest Increasing Subsequence

Problem : Given array A find the length of the Longest Increasing Subsequence (LIS)

LIS

The longest possible subsequence in which the elements of the subsequence are sorted in increasing order.

Subsequence

A subsequence is a sequence generated from the original array by deleting 0 or more elements without changing the relative order of the remaining elements.

Is it a subsequence ?

A [1, 3, 5, 7]

[1, 5, 7]

[3, 7]

[ ]

[1, 7, 5]

Longest Increasing Subsequence

Problem : Given array A find the length of the Longest Increasing Subsequence (LIS)

LIS

The longest possible subsequence in which the elements of the subsequence are sorted in increasing order.

Subsequence

A subsequence is a sequence generated from the original array by deleting 0 or more elements without changing the relative order of the remaining elements.

Examples :	Input :	Output :	LIS:
	[10, 9, 2, 5, 3, 7, 101, 18]	4	[2, 3, 7, 18]
	[3, 2, 1]	1	[3] , [2] or [1]

Longest Increasing Subsequence

Idea : We need to mark the smallest ending !

Definition of the DP table : DP[0...n-1][1...n]

DP[i][l] = "smallest ending of an increasing subsequence of length l in A[0...i]"

∞ if no such incresing subsequence exists

Computation of an entry :

Initialization : DP[0][1] = A[0] DP[0][l] = ∞ for l > 1

Recursion :

$$DP[i][l] = \begin{cases} A[i] & \text{if } DP[i-1][l-1] < A[i] \text{ and } A[i] < DP[i-1][l] \\ DP[i-1][l] & \text{else} \end{cases}$$

A[i] fits the element coming before by being bigger than it (it should be increasing)

A[i] improves the current smallest ending of length l by being smaller

Extracting the solution : The solution is found by backtracking

Graph Definitions

- Definitions
- Eulerian Tours
- Topological Sorting

Introduction to Graphs

Falls ein G einen Eulerzyklus hat , dann gibt es an jedem Knoten eine gerade Anzahl Kanten
(jeder Knoten hat einen geraden Grad)

predecessor / successor

Predecessor set of a v: $N^-(v) := \{u \in V \mid (u,v) \in E\}$

Sucessor set of V: $N^+(v) := \{w \in V \mid (v,w) \in E\}$

deg_{in} , Eingangsgrad: $deg^-(v) = |N^-(v)|$

deg_{out} , Ausgangsgrad: $deg^+(v) = |N^+(v)|$

In undirected G

$N(v) := \{w \in V \mid \{v,w\} \in E\}$ neighborhood of v

Spezialfall : Loops*

undirected G: a loop increases deg by 2

directed G: a loop increases deg_{in} deg_{out} by 1.

Lemma 5.3. *In jedem Graphen $G = (V, E)$ gilt*

(i) $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$, *falls G gerichtet ist, und*

(ii) $\sum_{v \in V} \deg(v) = 2|E|$, *falls G ungerichtet ist.*

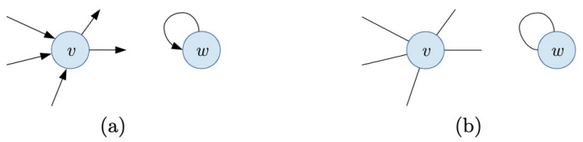
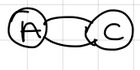


Abb. 5.4 Im gerichteten Graphen links (a) sind $\deg^-(v) = 3$, $\deg^+(v) = 2$, $\deg^-(w) = 1$ und $\deg^+(w) = 1$. Im ungerichteten Graphen rechts (b) sind $\deg(v) = 5$ und $\deg(w) = 2$.

- **undirected / directed G**
  - undirected: $E \subseteq \{\{u,v\} \mid u,v \in V\}$ — Kanten sind eine Menge; $u \neq v$ in ABW (keine Self-loops); per default G ohne Multikanten
  - directed: $E \subseteq V \times V$ — Kanten sind eine Tupel (u,v)
- **Degree** $deg(v) := |N_G(v)|$ (#inzidente Kanten; loops are counted x2)
  - $\sum_{v \in V} deg(v) = 2|E|$
  - neighborhood of v (all vertices adjacent to v)

~~Multigraph~~ : mehrfach Kanten



E is NIEMALS eine Multimenge in A&D
NIEMALS Multigraph

- **adjacent (benachbart)**: 2 vertices u,v are adjacent $\Leftrightarrow$ $(u,v) \vee (v,u) \in E$
- **incident (inzident)**: A vertex v and an edge e are incident $\Leftrightarrow$ $v \in e$

- **Tree** (*undirected)

Properties:
- connected
- no cycles
- n-1 edges

(2'si sağlanırsa diğeri otomatik sağlanır)

Additional properties: $(n \geq 2)$
- T has at least 2 leaves
- remove a leaf, T' is also a tree
- between any 2 nodes (u,v), there is exactly one u-v path in T.
  - ↳ any G that satisfies this is also a T

MST T with cost function on edges
↳ $\sum_{e \in E(T)} c(e)$ will be minimized

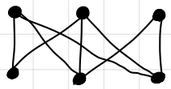
ein Blatt hat Grad 1

• Graph Types

alle Teilmengen von Grösse 2

K_n / complete / vollständiger G: every different pair of vertices is connected by an edge — $(V_n, \binom{V_n}{2})$

Bipartite G: A G that can be splitted into two different vertex sets, and there are no edges inside those sets



$V = A \cup B \quad (A \cap B = \emptyset)$

$E \subseteq \{\{u,w\} \mid u \in U, w \in W\}$ ungerichtet

$E \subseteq (U \times W) \cup (W \times U)$ gerichtet

Weighted G: G mit einer Kantengewichtsfunktion $c: E \to \mathbb{R}$
$c(e)$: weight of edge e

Definition 1. Let $G = (V, E)$ be a graph.

- A sequence of vertices $(v_0, v_1, \ldots, v_k)$ (with $v_i \in V$ for all i) is a **walk** (german "Weg") if $\{v_i, v_{i+1}\}$ is an edge for each $0 \leq i \leq k-1$. We say that v_0 and v_k are the **endpoints** (german "Startknoten" and "Endknoten") of the walk.
- A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **closed walk** (german "Zyklus") if it is a walk, $k \geq 2$ and $v_0 = v_k$.
- A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **path** (german "Pfad") if it is a walk and all vertices are distinct (i.e., $v_i \neq v_j$ for $0 \leq i < j \leq k$).
- A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **cycle** (german "Kreis") if it is a closed walk, $k \geq 3$ and all vertices (except v_0 and v_k) are distinct.
- A **Eulerian walk** (german "Eulerweg") is a walk that contains every edge exactly once.
- A **Hamiltonian path** (german "Hamiltonweg") is a path that contains every vertex.
- A **Hamiltonian cycle** (german "Hamiltonkreis") is a cycle that contains every vertex.
- A graph G is **connected** (german "zusammenhängend") if for every two vertices $u, v \in V$ there exists a path with endpoints u and v.[1]
- A graph G is a **tree** (german "Baum") if it is connected and has no cycles.

[1] We will see in exercise 8.5 that this definition is equivalent to the one given in the lecture (which was that a graph G is connected if for every two vertices $u, v \in V$ there exists a walk with endpoints u and v)

• Sequence Types: $\langle v_0, v_1, \ldots, v_k \rangle$

walk: $\langle v_0 \ldots, v_i, v_{i+1}, \ldots, v_k \rangle$ (endpoints) : Die Länge des Weges : k (# Kanten)

path: a walk without repeated vertices

closed walk: a walk where $v_1 = v_k$
(Zyklus)

ein Kreis ist auch ein Zyklus

connected G: Between every pair of 2 vertices, there exist a walk

cycle (Kreis): a closed walk without repeated vertices (except start=end) → loop is a cycle mit Länge 1.

$\forall v \in V \ \deg(v) = 2$ und G zshgd.

any vertices	walk / weg	closed walk / zyklus
distinct vertices	path / pfad	circle / kreis

ungerichtete Graphen mit dieser Def haben niemals Kreise der Länge 2.

loop: ↻

trail: a walk without repeated edges

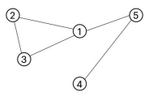
Let $G = (V, E)$ be an undirected graph. We say that G is

- **transitive** when, for any two edges $\{u, v\}$ and $\{v, w\}$ in E, the edge $\{u, w\}$ is also in E;
- **complete** when its set of edges is $\{\{u, v\} \mid u, v \in V, u \neq v\}$;
- the **disjoint union** of $G_1 = (V_1, E_1), \ldots, G_k = (V_k, E_k)$ iff $V = V_1 \cup \cdots \cup V_k$, $E = E_1 \cup \cdots \cup E_k$, and the $(V_i)_{1 \leq i \leq k}$ are pairwise disjoint.

Definitions

Graph
Node / Vertex $G = (V, E)$ $V = \{v_1, ..., v_n\}, |V| = n$ the set of nodes

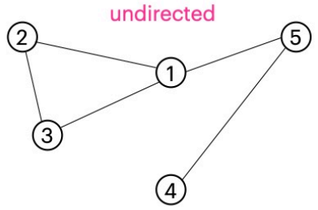
Graph
Edge $G = (V, E)$ $E = \{e_1, ..., e_m\}, |E| = m$ the set of edges



Graph
Undirected / Directed Graph

$G = (V, E)$ $E = \{e_1, ..., e_m\}, |E| = m$ the set of edges

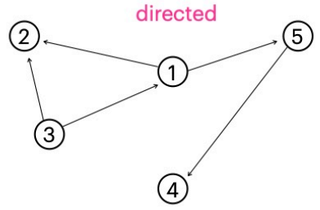
undirected



$E \subseteq \{\{u, v\} \mid u, v \in V\}$

$e_k = \{v_i, v_j\}$

directed

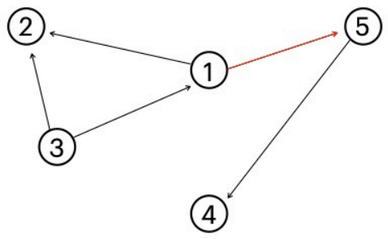


$E \subseteq V \times V$

$e_k = (v_i, v_j)$

Graph
adjacent / incident

- v_i and v_j are adjacent iff : $\{v_i, v_j\} \in E$ $(v_i, v_j) \in E$
- v_i and e_k are incident iff : $v_i \in e_k$.

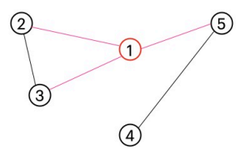


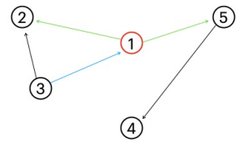
Graph
Degree

- deg(v) : The degree of a vertex v of a graph is the number of edges incident to this vertex, with loops counted twice.
- In directed graphs:
  - in-degree $\deg^-(v)$
  - out-degree $\deg^+(v)$

$$\sum_{v \in V} \deg(v) = 2|E| = 2m$$

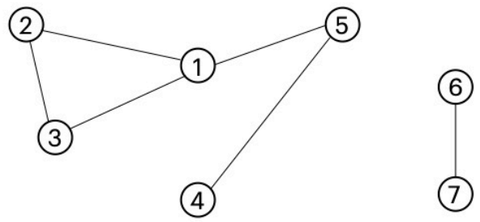
Handshaking Lemma





Graph
Connectivity

- For $u, v \in V$, we say u **reaches** v (or v is **reachable** from u; german "u erreicht v") if there exists a walk with endpoints u and v.
- A **connected component** of G is an equivalence class of the (equivalence) relation defined as follows: Two vertices $u, v \in V$ are equivalent if u reaches v.
- A graph G is **connected** (german "zusammenhängend") if for every two vertices $u, v \in V$ u reaches v or equivalently if there is only one connected component.

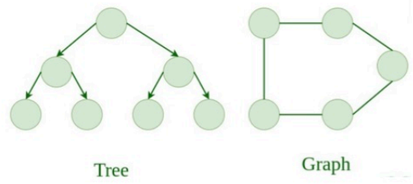


Graph
Tree

- A graph G is a **tree** (german "Baum") if it is connected and has no cycles.

A connected graph is a tree iff it has exactly m= n−1 edges.

Tree v/s Graph



Graph
Eulerian Lemmas

- A **Eulerian walk** (german "Eulerweg") is a walk that contains every edge exactly once.
- A **closed Eulerian walk** (german "Eulerzyklus") is a closed walk that contains every edge exactly once.

∃ Eulerian Walk ⟺ <= 2 vertices have odd degree (start and end)

Ausser für Start- und Endknoten gilt : #hin = #weg

∃ Eulerian Closed Walk ⟺ G is connected and every vertex has even degree

Graph
Walk vs Path

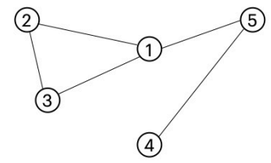
walk • A sequence of vertices $(v_0, v_1, \ldots, v_k)$ (with $v_i \in V$ for all i) is a **walk** (german "Weg") if $\{v_i, v_{i+1}\}$ is an edge for each $0 \leq i \leq k-1$. We say that v_0 and v_k are the **endpoints** (german "Startknoten" and "Endknoten") of the walk. The **length** of the walk $(v_0, v_1, \ldots, v_k)$ is k.

path • A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **path** (german "Pfad") if it is a walk and all vertices are distinct (i.e., $v_i \neq v_j$ for $0 \leq i < j \leq k$).

Is it a walk? Is it a path?

(5, 1, 3, 2, 1) ✅ ❌

(5, 1, 3) ✅ ✅



Graph
Closed Walk vs Cycle

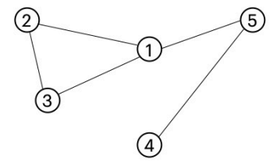
Closed walk • A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **closed walk** (german "Zyklus") if it is a walk, $k \geq 2$ and $v_0 = v_k$.

Cycle • A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **cycle** (german "Kreis") if it is a closed walk, $k \geq 3$ and all vertices (except v_0 and v_k) are distinct.

Is it a closed walk? Is it a cycle?

(5, 1, 3, 1, 5) ✅ ❌

(1, 3, 2, 1) ✅ ✅



Graph
Eulerian Walk / Closed Eulerian Walk

• A sequence of vertices $(v_0, v_1, \ldots, v_k)$ (with $v_i \in V$ for all i) is a **walk** (german "Weg") if $\{v_i, v_{i+1}\}$ is an edge for each $0 \leq i \leq k-1$. We say that v_0 and v_k are the **endpoints** (german "Startknoten" and "Endknoten") of the walk. The **length** of the walk $(v_0, v_1, \ldots, v_k)$ is k.

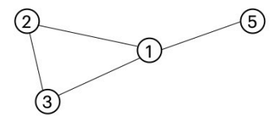
• A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **closed walk** (german "Zyklus") if it is a walk, $k \geq 2$ and $v_0 = v_k$.

Eulerian Walk • A **Eulerian walk** (german "Eulerweg") is a walk that contains every edge exactly once.

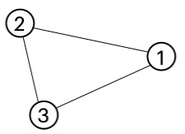
Closed Eulerian Walk • A **closed Eulerian walk** (german "Eulerzyklus") is a closed walk that contains every edge exactly once.

Is there an eulerian walk ? ✅ (1, 2, 3, 1, 5)

Is there a closed eulerian walk ? ❌



Is there a closed eulerian walk ? ✅ (1, 2, 3, 1)



Graph
Hamiltonian Path / Hamiltonian Cycle

• A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **path** (german "Pfad") if it is a walk and all vertices are distinct (i.e., $v_i \neq v_j$ for $0 \leq i < j \leq k$).

• A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **cycle** (german "Kreis") if it is a closed walk, $k \geq 3$ and all vertices (except v_0 and v_k) are distinct.

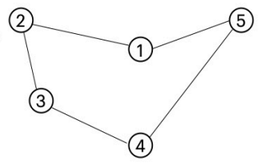
Hamiltonian Path • A **Hamiltonian path** (german "Hamiltonpfad") is a path that contains every vertex.

Hamiltonian Cycle • A **Hamiltonian cycle** (german "Hamiltonkreis") is a cycle that contains every vertex.

Is it a hamiltonian path? Is it a hamiltonian cycle?

(5, 1, 2, 3, 4) ✅ ❌

(5, 1, 2, 3, 4, 5) ❌ ✅

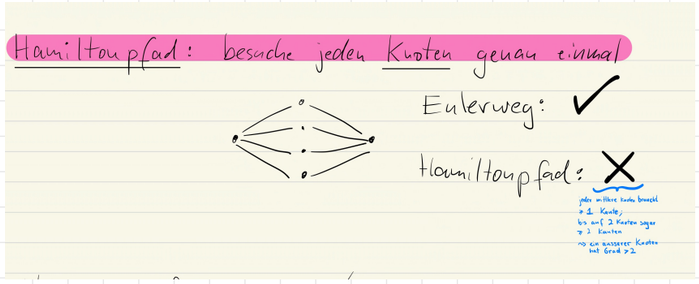


Eulerian Tours

Eulerian Walk : Durchlaufe jede Kante genau einmal

Wenn Eulerweg existiert, dann <= 2 Knotengrade ungerade (nur endknoten)

Ausser für Start- und Endknoten gilt : #hin = #weg



Eulerweg finden in $O(n+m)$

Handschlaglemma: $\sum_{v \in V} \deg(v) = 2|E|$

Beobachtung: Walk is closed: Endknoten inzident zu geraden #Kanten in Walk.

Eulerzyklus: Zyklus mit allen Kanten genau einmal

Behauptung: ∃ Eulerzyklus ⟺ zusammenhängend & alle Knotengrade gerade

u erreicht v: ∃ Weg zwischen u und v (reachable)

Äquivalenzrelation (symmetrisch, reflexiv, transitiv)

Zusammenhangskomponente: Äquivalenzklasse dieser Relation (ZHK; connected component)

Graph zusammenhängend: genau eine ZHK

jeder Knoten erreicht jeden anderen

eulerian trail:

Eulerian trail (or **Eulerian path**) is a trail in a finite graph that visits every edge exactly once (allowing for revisiting vertices)

Given the graph in the image, is it possible to construct a path (or a cycle; i.e., a path starting and ending on the same vertex) that visits each edge exactly once?

Walk(u): (finde möglichst langen Weg von u ohne wiederholte Kanten)

if ∃v. uv ∈ E, nicht markiert

markiere uv

Walk(v)

Properties [edit]

- An undirected graph has an Eulerian cycle if and only if every vertex has even degree, and all of its vertices with nonzero degree belong to a single connected component.
- An undirected graph can be decomposed into edge-disjoint cycles if and only if all of its vertices have even degree. So, a graph has an Eulerian cycle if and only if it can be decomposed into edge-disjoint cycles and its nonzero-degree vertices belong to a single connected component.
- An undirected graph has an Eulerian trail if and only if exactly zero or two vertices have odd degree, and all of its vertices with nonzero degree belong to a single connected component
- A directed graph has an Eulerian cycle if and only if every vertex has equal in degree and out degree, and all of its vertices with nonzero degree belong to a single strongly connected component. Equivalently, a directed graph has an Eulerian cycle if and only if it can be decomposed into edge-disjoint directed cycles and all of its vertices with nonzero degree belong to a single strongly connected component.
- A directed graph has an Eulerian trail if and only if at most one vertex has (out-degree) − (in-degree) = 1, at most one vertex has (in-degree) − (out-degree) = 1, every other vertex has equal in-degree and out-degree, and all of its vertices with nonzero degree belong to a single connected component of the underlying undirected graph.[citation needed]

Eulerian Cycle: eulerian trail that starts and ends on same vertex

= eulertour

Euler(G): (finde Eulerzyklus in G wenn existent) — G muss zsgd sein; 2 shaded: alle Knotengrade gerade

- Leere Liste Z, alle Kanten unmarkiert
- Euler Walk(u_0) für $u_0 \in V$ beliebig
- return Z

Euler Walk(u): ⚠ auch nicht in Rekursion markiert!

for uv ∈ E, nicht markiert

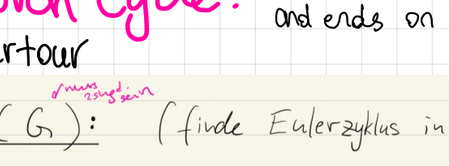
  markiere Kante uv

  EulerWalk(v)

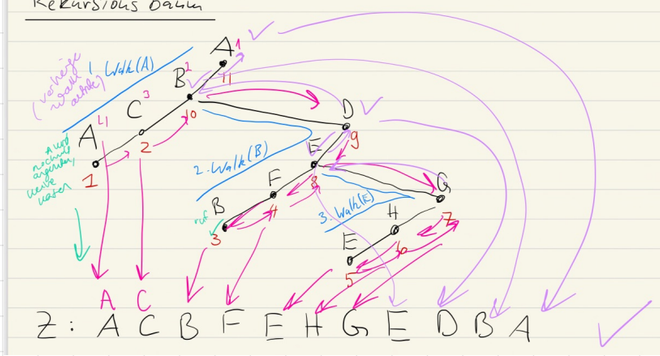
Z ⇐ (Z, u) — u am Ende der Liste hinzufügen

Eulerwalk (A)

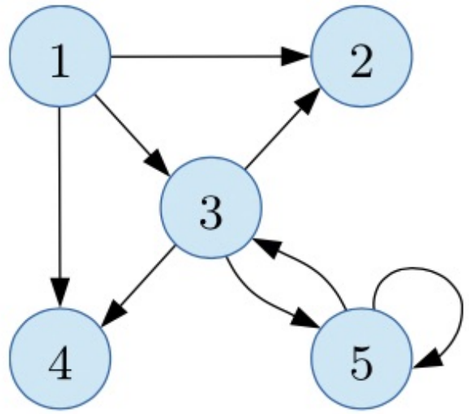
Beispiel:



Rekursions baum

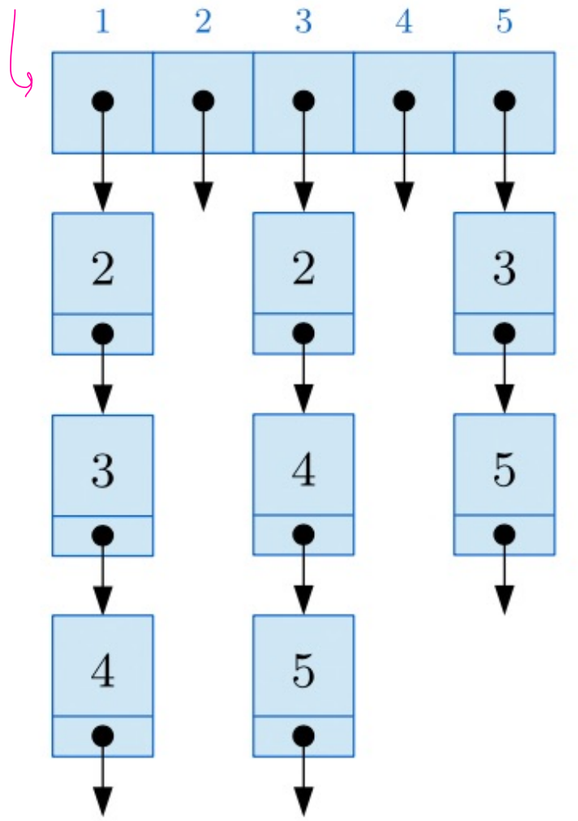


$G = (V, E)$ $V = \{v_1, \dots, v_n\}$



• Adjacency list

Array(A[1] ... A[n])



A[i]: linked list of all vertices in $N^+(v)$
(in any order, beliebige Reihenfolge)

Space: $\Theta(|V| + |E|)$
Array; lists have overall $|E|$ elements

• Adjacency Matrix

$A_G = (a_{ij})_{1 \le i,j \le n}$

$n \times n$

$a_{ij} = 1 \iff (v_i, v_j) \in E$

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

in undirected Gs:
$\{v_i, v_j\} \in E \iff a_{ij} = 1$, $a_{ji} = 1$

Loop um v_i: $a_{ii} = 1$ (i-te Diagonalelement)

$\Theta(|V|^2)$: Space

Operations:

	Adjazenz matrix	Adjazenz list
Alle Nachbarn von $v \in V$ ermitteln	$\Theta(n)$	$\Theta(\deg^+(v))$
Finde $v \in V$ ohne Nachbar	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Ist $(u, v) \in E$?	$\mathcal{O}(1)$	$\mathcal{O}(\deg^+(v))$
Kante einfügen	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Kante löschen	$\mathcal{O}(1)$	$\mathcal{O}(\deg^+(v))$

LA Beziehung

ANZAHL WEGE DER LÄNGE t

Theorem 5.4. *Sei G ein Graph (gerichtet oder ungerichtet) und $t \in \mathbb{N}$ beliebig. Das Element an der Position (i,j) in der Matrix A_G^t gibt die Anzahl der Wege der Länge t von v_i nach v_j an.*

Betrachten wir noch einmal die Adjazenzmatrix des in Abbildung 5.5 dargestellten Graphen und quadrieren wir sie, dann erhalten wir die Matrix

$$B := A_G^2 = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 2 \end{pmatrix} \tag{106}$$

Wie kann diese anschaulich interpretiert werden? Ein Eintrag b_{ij} dieser Matrix wird durch $b_{ij} = \sum_{k=1}^{n} a_{ik} \cdot a_{kj}$ berechnet. Wir beobachten nun, dass das Produkt $a_{ik} \cdot a_{kj}$ entweder 0 oder 1 ist, und es ist genau dann 1, wenn sowohl $(v_i, v_k) \in E$ als auch $(v_k, v_j) \in E$ sind. Das heisst also, b_{ij} zählt die Anzahl der Wege der Länge 2 von i nach j. Wir können die Aussage sogar auf A_G^t für beliebige $t \in \mathbb{N}$ verallgemeinern, wie das folgende Theorem zeigt:

Beweis. Wir beweisen die Aussage durch vollständige Induktion über t.

Induktionsanfang ($t = 1$): Die Adjazenzmatrix $A_G = A_G^1$ enthält die Anzahl der Wege der Länge 1. Zwischen zwei Knoten gibt es entweder einen Weg der Länge 1 (wenn die entsprechende Kante existiert), und keinen sonst.

Induktionshypothese: Angenommen, in A_G^t gibt jeder Eintrag (i,j) die Anzahl der Wege der Länge t von v_i nach v_j an.

Induktionsschritt ($t \to t+1$): Sei $B = (b_{ij})_{1 \le i,j \le n} := A_G^t$. Betrachte nun $C = (c_{ij})_{1 \le i,j \le n} := A_G^{t+1} = B \times A_G$. Für einen Eintrag c_{ij} gilt

$$c_{ij} = \sum_{k=1}^{n} b_{ik} \cdot a_{kj}. \tag{107}$$

Dabei zählt b_{ik} gemäss Induktionshypothese die Anzahl der Wege von v_i nach v_k der Länge t. Ausserdem ist a_{kj} entweder 0 oder 1, und 1 genau dann, wenn eine Kante (also ein Weg der Länge 1) von v_k nach v_j existiert. Das heisst, die Anzahl der Wege von v_i nach v_j über den Zwischenknoten v_k ist genau b_{ik}, falls (v_k, v_j) existiert, und 0 sonst. Da die Summe über alle möglichen Zwischenknoten v_k gebildet wird, gibt c_{ij} also korrekt die Anzahl der Wege von v_i nach v_j der Länge $t+1$ an. ∎

ANWENDUNGEN

Um in einem Graphen den kürzesten Weg (mit einer minimalen Anzahl von Kanten) zwischen zwei gegebenen Knoten v_i und v_j zu bestimmen, genügt es also offenbar, A_G zu potenzieren, bis zum ersten Mal der Eintrag an der Position (i,j) ungleich Null ist. Da ein *kürzester* Weg offenbar keinen Knoten mehrfach benutzt und damit höchstens Länge $n-1$ hat, reicht eine Potenzierung bis n aus; enthält die Position (i,j) dann noch immer keine Eins, dann gibt es keinen Weg von v_i nach v_j.

DREIECK

Eine andere Anwendung besteht in der Berechnung der Anzahl von Dreiecken in ungerichteten Graphen. Ein *Dreieck* ist eine Menge D von drei Knoten, wobei zwischen jedem Paar zweier Knoten in D eine Kante verläuft. Um für einen gegebenen ungerichteten Graphen G die Anzahl der Dreiecke zu bestimmen, entfernen wir zunächst alle Schleifen von G, indem wir die Diagonaleinträge in A_G auf 0 setzen. Danach berechnen wir die Matrix $B = (b_{ij})_{1 \le i,j \le n} := A_G^3$ (dazu genügen zwei Matrixmultiplikationen). Ein Eintrag b_{ij} gibt nun an, wie viele Wege der Länge 3 von v_i nach v_j existieren, d.h., er zählt die Anzahl der Dreiecke in denen v_i enthalten ist. Da jedes Dreieck irgendeinen Knoten des Graphen enthält und auf sechs verschiedene Arten gezählt werden wird ($\langle x,y,z,x\rangle$, $\langle x,z,y,x\rangle$, $\langle y,z,x,y\rangle$, $\langle y,x,z,y\rangle$, $\langle z,x,y,z\rangle$, $\langle z,y,x,z\rangle$), beträgt die Anzahl der Dreiecke exakt $(\sum_{i=1}^{n} b_{ii})/6$. Als Beispiel betrachten wir den folgenden Graph:

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

Nach Entfernung der Schleifen ergeben sich die neue Adjazenzmatrix bzw. ihre dritte Potenz

$$\tilde{A}_G = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \text{ bzw. } B = \tilde{A}_G^3 = \begin{pmatrix} 2 & 3 & 4 & 1 \\ 2 & 2 & 4 & 1 \\ 4 & 4 & 2 & 3 \\ 1 & 1 & 3 & 0 \end{pmatrix}.$$

$6/6 = 1$

Offenbar hat G genau ein Dreieck (nämlich die Menge $\{v_1, v_2, v_3\}$), und die Summe der Diagonaleinträge in B ist genau 6.

DM Beziehung

5.1.5 Beziehung zu Relationen

Einen Graphen $G = (V,E)$ können wir auch als Relation $E \subseteq V \times V$ auf V auffassen, und umgekehrt. Sei wie zuvor $A_G = (a_{ij})_{1 \le i,j \le n}$ die Adjazenzmatrix von G. Konzepte von Relationen können wie folgt auf Graphen übertragen werden:

- E ist genau dann *reflexiv*, wenn E jede Kante (v,v) mit $v \in V$ enthält, G also eine Schleife um jeden Knoten $v \in V$ hat. Dies ist genau dann der Fall, wenn $a_{ii} = 1$ für alle $i \in \{1, \dots, n\}$ ist. — REFLEXIVITÄT
- E ist genau dann *symmetrisch*, wenn für alle $(v,w) \in E$ auch $(w,v) \in E$ gilt, d.h. G ungerichtet ist. — SYMMETRIE
- E ist genau dann transitiv, wenn für jedes Paar zweier Kanten $(u,v) \in E$ und $(v,w) \in E$ auch $(u,w) \in E$ ist. — TRANSITIVITÄT

Eine Äquivalenzrelation erfüllt bekanntermassen alle drei obigen Eigenschaften. Folglich ist der Graph zu einer Äquivalenzrelation eine Kollektion vollständiger, ungerichteter Graphen, wobei jeder Knoten eine Schleife hat.

Die reflexive und transitive Hülle eines Graphens $G = (V,E)$ beschreibt die sog. *Erreichbarkeitsrelation* E^*. Dies ist eine zweistellige Relation, die ein Paar (v,w) genau dann enthält, wenn es in G einen Weg vom Knoten v zum Knoten w gibt. — ERREICHBARKEITSRELATION

5.1.6 Berechnung der reflexiven und transitiven Hülle

Gegeben sei ein gerichteter oder ungerichteter Graph $G = (V,E)$ durch seine Adjazenzmatrix $A_G = (a_{ij})_{1 \le i,j \le n}$. Wir wollen nun die reflexive und transitive Hülle von G als Matrix berechnen. Das Ziel ist also die Berechnung einer Matrix $B = (b_{ij})_{1 \le i,j \le n}$, wobei b_{ij} genau dann den Wert 1 annimmt, wenn $(v_i, v_j) \in E^*$ ist (und 0 sonst). Unser Algorithmus darf in-place arbeiten, d.h. er darf die Matrix A überschreiben, sodass am Ende an ihrer Stelle B steht. Wir beobachten, dass es niemals nötig ist, eine 1 zu überschreiben, da eine 1 an einer Position (i,j) signalisiert, dass es (irgendeinen) Weg von v_i nach v_j gibt. Nur Nullen müssen überschrieben werden, und zwar genau dann, wenn ein Weg zwischen zwei Knoten gefunden wird und bisher nicht bekannt war, dass zwischen ihnen ein Weg verläuft. — IN-PLACE

Um nun Wege der Länge 2 zu finden, können wir über alle $i,j,k \in \{1, \dots, n\}$ iterieren und $a_{ij} \leftarrow 1$ setzen, wenn sowohl $a_{ik} = 1$ als auch $a_{kj} = 1$ sind. Zusätzlich setzen wir natürlich $a_{ii} \leftarrow 1$ für alle $i \in \{1, \dots, n\}$, da die Hülle ja auch reflexiv sein soll. Nach diesen Schritten codiert die Matrix bereits alle Wege der Längen 1 und 2. Führen wir dieses Verfahren erneut aus, wurden alle Wege der Länge höchstens 4 gefunden, bei einer weiteren Ausführung alle Wege der Länge höchstens 8, usw. Daher reichen $\lceil \log_2 n \rceil$ viele Ausführungen, um alle Wege zu finden. — ERSTE IDEE

Tatsächlich reicht sogar eine einzige Ausführung, wenn wir sicherstellen können, dass sowohl a_{ik} als auch a_{kj} bereits ihren entgültigen Wert enthalten (also 1, falls es irgendeinen Weg von v_i nach v_k bzw. von v_k nach v_j gibt, und 0 sonst). Warshall beobachtete nun, dass es ausreicht, die Wege in der Reihenfolge des grössten Zwischenknotens v_k zu berechnen, was der folgende Algorithmus tut: — VERBESSERUNG

REFLEXIVE UND TRANSITIVE HÜLLE

```
ReflexiveTransitiveHull(A)
1 for k ← 1,...,n do
2     a_kk ← 1                                  ▷ Reflexive Hülle
3     for i ← 1,...,n do
4         for j ← 1,...,n do
5             if a_ik = 1 and a_kj = 1 then a_ij ← 1   ▷ Berechne Weg über v_k
```

LAUFZEIT

Die Laufzeit dieses Algorithmus ist offenbar in $\Theta(n^3)$. Die Korrektheit weisen wir induktiv nach, dass in der äusseren Schleife die folgende Invariante stets erfüllt bleibt: Alle Wege, bei denen der grösste Zwischenknoten (und damit alle Zwischenknoten) einen Index $< k$ hat, wurden berücksichtigt.

Induktionsanfang ($k = 1$): Alle direkten Wege ohne Zwischenknoten (dies entspricht allen Kanten) sind bereits in A_G enthalten.

Induktionshypothese: Sei die obige Invariante vor dem k-ten Schleifendurchlauf wahr, d.h. alle Wege, bei denen der grösste Zwischenknoten einen Index $< k$ hat, wurden berücksichtigt.

Induktionsschritt ($k \to k+1$): Betrachte nun den k-ten Schleifendurchlauf sowie einen beliebigen Weg von einem Knoten v_i zu einem Knoten v_j, dessen grösster Zwischenknoten den Index k hat. Wir müssen zeigen, dass der Algorithmus dann tatsächlich $a_{ij} \leftarrow 1$ setzt. Dies ist der Fall, denn nach Induktionshypothese haben sowohl a_{ik} als auch a_{kj} den Wert 1 (da k der grösste Index eines Zwischenknotens auf einem Weg von v_i nach v_j ist, müssen die Zwischenknoten auf den Wegen von v_i nach v_k bzw. von v_k nach v_j entsprechend kleinere Indizes haben). Daher setzt der Algorithmus in Schritt 5 auch tatsächlich $a_{ij} \leftarrow 1$, und da auf diese Weise alle Wege gefunden werden, deren grösster Zwischenknoten den Index k hat, bleibt die Invariante erhalten. ∎

Topological Sorting

∃ top. sorting ⇔ ∄ directed closed walk

Here's the algorithm step by step:

- Find a vertex that has indegree = 0 (no incoming edges)
- Remove all the edges from that vertex that go outward (make its outdegree = 0, remove outgoing edges)
- Add that vertex to the array representing the topological sorting of the graph.
- Repeat till there are no more vertices left.

4.2.4 Topological sort

A topological sorting of a graph G is an ordering $\prec$ of V such that for all u, v s.t. u and v are adjacent, $u \prec v$. If a topological sorting exists, it can be produced by running DFS on the graph and adding each node to the ordering after handling its children. The multiple runs of DFS in the algorithm below handle possible non-connectedness.

Algorithm 17 Topological sort

function DFS(s, G, T, D)
  for w s.t. v and w are adjacent in G **do**
    if $w \notin D$ **then**
      $D \leftarrow D \cup \{w\}$
      DFS (w, G, T, D)
  $T \leftarrow T \cup \{s\}$
$T \leftarrow []$
$D \leftarrow \emptyset$
for $v \in V$ **do**
  if $v \notin D$ **then**
    DFS (v, G, T, D)

The complexity of topological sort is $O(n + m)$.

Begriffe gerichtete Graphen

Der Graph $G = (V, E)$ ist definiert wie beim ungerichteten **ausser** Kanten sind geordnete Paare $e = (u, v) \in E$

- v ist Nachfolger von u
- u ist Vorgänger von v
- $deg_{in}(u)$ = Eingangsgrad
- $deg_{out}(u)$ = Ausgangsgrad
- Quelle u: $deg_{in}(u) = 0$
- Senke v: $deg_{out}(v) = 0$

Definition Graph

Ein **Graph** ist ein Tupel $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ wobei

- $V :=$ Knotenmenge (vertices)
- $E :=$ Kantenmenge (edges)

jede Kante ist ein ungeordnetes Paar zweier Knoten $u \neq v$, $e = \{u, v\} \in E$ (Kurzform: uv)

Weg, Pfad, Zyklus

- **Weg:** Folge von benachbarten Knoten (engl. walk)
- **Pfad:** Weg ohne wiederholte Knoten
- **Zyklus:** Weg mit $v_0 = v_l$, $l \geq 2$ (engl. closed walk)

Die Länge eines Wegs bzw. Pfads ist die Anzahl an Kanten, nicht die Anzahl an Knoten

Begriffe

- u, v adjazent/benachbart $\Leftrightarrow e = \{u, v\} \in E$
- $e \in E$ inzident/anliegend zu $v \Leftrightarrow \exists u \in V$, so dass $e = \{u, v\}$
- $deg(u)$ = Knotengrad von u (Anzahl Nachbarn)
- u erreicht $v \Leftrightarrow \exists$ Weg zwischen u und v (engl. reachable)
  Äquivalenzrelation (symmetrisch, reflexiv, transitiv)
- Zusammenhangskomponente (ZHK): Äquivalenzklasse der "erreichbar"-relation (engl. connected component)
- Graph ist zusammenhängend $\Leftrightarrow$ es gibt gibt genau eine ZHK

Handschlag Lemma: $\sum_{v \in V} deg(v) = 2 \cdot |E|$

Eulerweg, Hamiltonpfad, Eulerzyklus

- **Eulerweg:** Weg welcher jede Kante **genau** einmal enthält (engl. Eulerian walk)
- **Hamiltonpfad:** Pfad der jeden Knoten **genau** einmal enthält
- **Eulerzyklus:** Zyklus welcher jede Kante **genau** einmal enthält

∃ Eulerzyklus $\Leftrightarrow$ alle Knotengrade gerade und alle Kanten in einer ZHK

Algorithmus Eulertour / Eulerwalk

Euler(G):

- Input: Graph $G = (V, E)$
- Output: Liste Z mit Eulerzyklus, falls existiert.
- Laufzeit: $\mathcal{O}(m)$

EulerWalk(u):

- Input: Knoten $u \in V$
- Output: Keiner
- Laufzeit: $\mathcal{O}(m)$

Algorithm 1 Euler(G)
Require: Alle Kanten unmarkiert
1: $Z \leftarrow$ Leere Liste
2: EulerWalk(u_0) ▷ für $u_0 \in V$ beliebig
3: **return** Z

Algorithm 2 EulerWalk(u)
1: **for** $uv \in E$, nicht markiert **do**
2: markiere Kante uv
3: EulerWalk(v)
4: $Z \leftarrow Z \cup \{u\}$

d) *Directed Acyclic Tournament*

A *tournament* is a directed graph $G = (V, E)$ such that:

- G has no self loops, i.e., $(v, v) \notin E$, for all $v \in V$. (Note that the graphs that we usually consider have no self loops.)
- For every two distinct vertices $u, v \in V$, either $(u, v) \in E$ or $(v, u) \in E$ but not both.

Let G be a directed acyclic graph that is also a tournament. Show that G has a unique topological sorting.

Solution:

Let n be the number of vertices of G. Since G is a directed acyclic graph (DAG), it has at least one topological sorting $(v_1, \ldots, v_n)$. On the other hand, since G is a tournament, for every $1 \leq i < n$, there is an edge between v_i and v_{i+1} in G. Furthermore, since v_i appears before v_{i+1} in the topological sorting $(v_1, \ldots, v_n)$ of G, the edge between v_i and v_{i+1} must be (v_i, v_{i+1}) and cannot be (v_{i+1}, v_i). Therefore, $(v_1, \ldots, v_n)$ is a path in G, hence v_i must appear before v_{i+1} in any topological sorting of G. We conclude that $(v_1, \ldots, v_n)$ is the unique topological sorting of G.

Trees

- Every tree with n vertices has exactly n-1 edges

Proof :

We proceed by induction on n.

Base case: When $n = 1$, there can only be $0 = n - 1$ edges. When $n = 2$, there exists a unique tree (two vertices connected by an edge), and that one has $1 = n - 1$ edge. This completes the base case.

Induction hypothesis: Assume that the hypothesis is true for every tree with $n \geq 2$ vertices, i.e. it contains $n - 1$ edges.

Induction step: We now show the property holds for every tree $G = (V, E)$ with $|V| = n + 1$ vertices.

Let u be a leaf in G (it must exist by part (a)), and let v be u's only neighbor in the tree $G = (V, E)$. Consider the graph $G' := (V \setminus \{u\}, E \setminus \{u, v\})$. We first argue that G' is a tree.

Claim: G' is connected.

Proof of Claim: Let $a, b \in V \setminus \{u\}$. Since G is a tree, there exists a path P in G with endpoints a, b. It is immediate that no path can contain a leaf except on its endpoints (or the leaf's only incident edge is used twice). Hence, P is also a path in G'. Thus, a and b are connected in G'. Since a and b were arbitrary, G' is connected. This completes the proof of this claim.

Claim: G' has no cycles.

Proof of Claim: Suppose for the sake of contradiction that P is a cycle in G'. But since G' is a subgraph of G, P is also a cycle in G. However, G is a tree and thus cannot have a cycle. This completes the proof of the second claim.

Combining the two claims, we proved that G' is a tree. It contains $|V \setminus \{u\}| = (n + 1) - 1 = n$ vertices. Hence, by the induction hypothesis, $|E \setminus \{u, v\}| = n - 1$. Therefore, $|E| = n$, which completes the induction step and the proof.

- G with n vertices is a tree $\iff$ G has n-1 edges and is connected

Proof :

Hint: *One direction is immediate by part (b). For the other direction (every connected graph with $n-1$ edges is a tree), use induction on n. First, prove there always exists a leaf by considering the average degree. Then, disconnect the leaf from the graph and argue that the remaining graph is still connected and has exactly one edge less. Apply the induction hypothesis and conclude.*

Solution:

Suppose G is a tree. By definition, G is connected. By part (b), it has $n - 1$ edges. This completes one direction of the implication.

We now prove the other direction. Suppose G is connected and has $n - 1$ edges. We proceed by induction on n.

Base case: Let $n = 1$. The graph with a single vertex and 0 edges is trivially a tree. Let $n = 2$. There exists one unique graph with 2 vertices and 1 edge, and that graph is also obviously a tree. This completes the base case.

Induction hypothesis: Assume the hypothesis: Every connected graph with $n \geq 2$ vertices and $n - 1$ edges is a tree.

Induction step: We now show the property holds for $n+1$. Let $G = (V, E)$ be a connected graph with $n + 1$ vertices and n edges. The average degree in this graph is $2|E|/|V| = 2n/(n + 1) < 2$. Hence, there must exist a vertex u with degree 1 (no connected graph with at least 2 vertices can have 0-degree vertices). In other words, u is a leaf and let v be u's only neighbor in G. Consider the graph $G' := (V \setminus \{u\}, E \setminus \{u, v\})$. Clearly, G' has $n - 1$ edges.

Claim: G' is connected.

Proof of Claim: Let $a, b \in V \setminus \{u\}$. Since G is connected, there exists a path P in G with endpoints a, b. As in part (b), no path can contain a leaf except on its endpoints. Hence, P is also a path in G' and thus G' is connected. This completes the proof of this claim.

Therefore, we can apply the induction hypothesis on G' and conclude G' is a tree. It is simple to conclude that then G is also a tree: Any cycle in G must be fully contained in G' (since it cannot contain a leaf), and this is impossible since G' is a tree.

- Algorithm for checking if G is a tree :

Algorithm 2

Input: integers n, m. Collection of integers $a_1, b_1, a_2, b_2, \ldots, a_m, b_m$.

Let $visited[1 \ldots n]$ be a global variable, initialized to $False$.

function $walk(u)$ ▷ Find all neighbors of u that have not been visited and walk there.
    $visited[u] \leftarrow True$
    for $i \leftarrow 1 \ldots m$ **do** ▷ Iterate over all edges.
        if $a_i = u$ and not $visited[b_i]$ **then**
            $walk(b_i)$
        if $b_i = u$ and not $visited[a_i]$ **then**
            $walk(a_i)$

$walk(1)$ ▷ Find all vertices connected to 1.
$connected \leftarrow True$ if $visited[\cdot] = [True, True, \ldots, True]$ and $connected \leftarrow False$ otherwise
if $connected = True$ and $m = n - 1$ **then** ▷ Use the characterization from part (c).
    Print("YES")
else
    Print("NO")

Short Questions about Graphs:

Exercise 8.5 *Short questions about graphs* **(2 points).**

In the following, let $G = (V, E)$ be a graph, $n = |V|$ and $m = |E|$.

(a) Let $v \neq w \in V$. Prove that if there is a walk with endpoints v and w, then there is a path with endpoints v and w.

Solution:

If there is a walk with endpoints v and w, consider a walk $W : v = v_0, v_1, \ldots, v_k = w$ between these two vertices with the shortest length. Since $v \neq w$, $k \geq 1$. If W is not a path, then there are $0 \leq i < j \leq k$ with $v_i = v_j$. But then, $v = v_0, v_1, \ldots, v_i = v_j, v_{j+1}, \ldots, v_k = w$ is a walk between v and w with length $k - (j - i) < k$, contradicting the choice of the walk. Hence, W is a path, so there is a path with endpoints v and w.

For each of the following statements, decide whether the statement is true or false. If the statement is true, provide a proof; if it is false, provide a counterexample.

(b) Every graph with $m \geq n$ is connected.

Solution:

This statement is false.
Consider the graph G that is a disjoint union of two cycles of length 3, i.e.

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$$

and

$$E = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_1\}, \{v_4, v_5\}, \{v_5, v_6\}, \{v_6, v_4\}\}.$$

Then, both $n = m = 6$, but G is not connected since there is for example no path from v_1 to v_4.

(c) If G contains a Hamiltonian path, then G contains a Eulerian walk.

Solution:

This statement is false.
Consider the complete graph on 4 vertices, i.e.

$$V = \{v_1, v_2, v_3, v_4\}$$

and

$$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}\}.$$

Then G has a Hamiltonian path v_1, v_2, v_3, v_4. But since every vertex has odd degree 3, it follows that G has no Eulerian walk as we know from the lecture that a graph that has an Eulerian walk has at most 2 vertices with odd degree.

(d) If every vertex of a non-empty graph G has degree at least 2, then G contains a cycle.

Solution:

This statement is true.

Proof using bound on the number of edges in a tree. Assume for a contradiction that G does not contain a cycle. Then each of its connected components must be a tree. By a previous exercise, we know that this implies that the total number of edges in G is at most $n - 1$. But each vertex of G has degree at least 2, implying G has at least $(2 \cdot n)/2 = n$ edges, a contradiction.

Direct proof. We construct a cycle in G as follows. Let v_1 be any vertex of G, and let v_2 be a neighbour of v_1 (i.e., $\{v_1, v_2\}$ is an edge). For $3 \leq i \leq n + 1$, inductively choose vertices v_i so that $\{v_i, v_{i-1}\}$ is an edge, and $v_i \neq v_{i-2}$. This is possible because each vertex of G has degree at least 2, and so v_{i-1} always has at least one neighbour which is not equal to v_{i-2}.

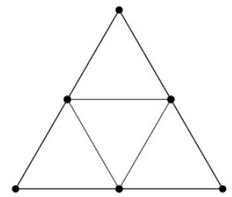
We have thus created a walk in G of length n. As G only has n vertices, this means there exist distinct $1 \leq j, k \leq n+1$ such that $v_j = v_k$, and $v_j, v_{j+1}, \ldots, v_{k-1}$ are all distinct. By construction, $k \geq j + 3$, and so the vertices $v_j, v_{j+1}, \ldots, v_k$ form a cycle (of length $k - j$).

(e) Suppose in a graph G every pair of vertices v, w has a common neighbour (i.e., for all distinct vertices v, w, there is a vertex x such that $\{v, x\}$ and $\{w, x\}$ are both edges). Then there exists a vertex p in G which is a neighbour of every other vertex in G (i.e., p has degree $n - 1$).

Solution:

This statement is false.

A counterexample is given by the following graph:



(f) Let G be a connected graph with at least 3 vertices. Suppose there exists a vertex v_{cut} in G so that after deleting v_{cut}, G is no longer connected. Then G does not have a Hamiltonian cycle. (Deleting a vertex v means that we remove v and any edge containing v from the graph).

Solution:

This statement is true. Suppose for a contradiction that G has a Hamiltonian cycle $v_1, v_2, v_3, \ldots, v_n, v_1$, which we order so that $v_1 = v_{\text{cut}}$. Then, after removing $v_1 = v_{\text{cut}}$ from G, the resulting graph still has a Hamiltonian path, namely $v_2, v_3, \ldots, v_n$. In particular, the resulting graph is still connected.

Exercise 9.2 *Short statements about graphs (cont'd)* **(1 point)**.

In the following, let $G = (V, E)$ be a directed graph. For each of the following statements, decide whether the statement is true or false. If the statement is true, provide a proof; if it is false, provide a counterexample.

(a) If for every vertex $v \in V$ its in-degree $\deg_{\text{in}}(v)$ is even, then $|E|$ is even.

Solution:

This statement is true.
The following equality holds $\sum_{v \in V} \deg_{\text{in}}(v) = |E|$ since on both sides every edge is counted exactly once. Hence, if all terms on the left side are even, then also $|E|$ is even.

(b) For a longest directed path $P : v_0, \ldots, v_\ell$ in G, the endpoint has to be a sink.

Solution:

This statement is false.
Consider the graph with three vertices

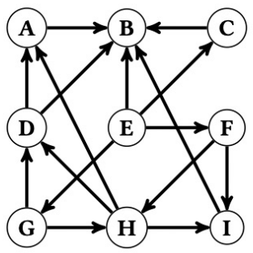
$$V = \{v_1, v_2, v_3\}$$

and the following edges

$$E = \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}.$$

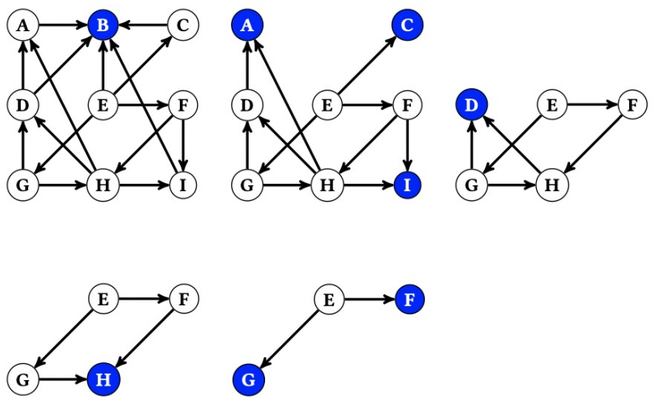
A longest directed path is for example v_1, v_2, v_3. However, v_3 is not a sink since it has the outgoing edge (v_3, v_1).
Remark: If the graph is assumed to have no directed cycles, i.e. we have a directed acyclic graph, then the statement holds as was used and proven in the lecture.

(c) The following graph has a topological sorting. If so, give a topological sorting; if not, prove why no topological sorting can exist.



Solution:

This statement is true.
We construct a topological sorting with the strategy that was discussed in the lecture, that is, we find a sink v in the graph, remove v (and all incident edges) from the graph and iterate. Since the graph is small, we find a sink "by hand", but we could as well do it with the algorithm that was discussed in the lecture. All graphs that occur in the process are described below, where the current sinks are marked in blue. Note that there are different possible topological sortings, the one described here is just an example.
In the first graph the vertex "B" is a sink, so "B" appears last in our topological sorting.
Now, there are several choices, all the vertices "A", "C" and "I" are sinks. We can add all these three vertices into our sorting, so a possible current end of the sorting is (A,C,I,B).
After removing these vertices, the only sink is "D", leading to the current end of the sorting (D,A,C,I,B).
In the next graph, only "H" is a sink, leading to (H,D,A,C,I,B).
In the last graph, both "F" and "G" are sinks. After removing these two vertices only the vertex "E" remains, so we get for example the following topological sorting (E,F,G,H,D,A,C,I,B). One can verify in the original graph that this is indeed a topological sorting. This is however not strictly necessary since we know from the lecture that the algorithm we executed is correct.



Graph Definitions – Short Proofs

In the following, let $G = (V, E)$ be a graph, $n = |V|$ and $m = |E|$.

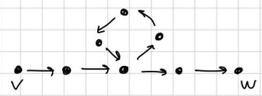
(a) Let $v \neq w \in V$. if there is a walk with endpoints v and w, then there is a path with endpoints v and w. T/F ? Proof ?

True

∃ walk W with endpoints v and w

If all vertices in W are distinct, W is already a path

If W contains repeated vertices, remove the cycles (to obtain a path from v to w)



If there exists a cycle $v_{i-1}, v_i, v_{i+1}, \ldots, v_j$, v_j, v_{j+1}

Remove the vertices $v_{i+1} \ldots v_j$

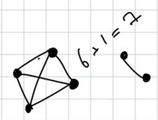
$v_{i-1}, v_i, v_j, v_{j+1} \ldots$

Repeat this for every remaining repeated vertices

After removing all cycles, remaining walk will be a path (has distinct vertices + still connects v to w)

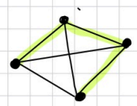
(b) Every graph with $m \geq n$ is connected.

False



(c) If G contains a Hamiltonian path, then G contains a Eulerian walk.

False



But every vertex has odd degree ⟺ no eulerian walk
(at most 2 vertices could have odd degree, start and end)

(d) If every vertex of a non-empty graph G has degree at least 2, then G contains a cycle.

True

Proof by using properties of a tree

Assume that G does not contain a cycle

⇒ All of it's connected components must be a tree

⇒ $|E| \leq |V| - 1$

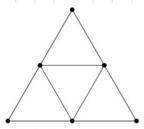
But each vertex has degree at least 2 → $\frac{2 \cdot |V|}{2} = |V|$ edges

Contradiction!

G has to contain a cycle!

(e) Suppose in a graph G every pair of vertices v, w has a common neighbour (i.e., for all distinct vertices v, w, there is a vertex x such that $\{v, x\}$ and $\{w, x\}$ are both edges). Then there exists a vertex p in G which is a neighbour of every other vertex in G (i.e., p has degree $n - 1$).

False



(f) Let G be a connected graph with at least 3 vertices. Suppose there exists a vertex v_{cut} in G so that after deleting v_{cut}, G is no longer connected. Then G does not have a Hamiltonian cycle. (Deleting a vertex v means that we remove v and any edge containing v from the graph).

True

Indirect proof

Assume G has a hamiltonian cycle

Resulting G after removing v_{cut} would still have a hamiltonian path.

It's still connected!

Graph Searchs

- DFS
- BFS

DFS

DFS (Depth-First-Search)

DFS(G) (und Visit(u)):

- Input: Graph $G = (V, E)$
- Output: Implementationsabhängig (kann für sehr viel verwendet werden)
- Laufzeit: $\mathcal{O}(n + m)$ (für Adjazenzlisten)

Algorithm 3 Visit(u)

1: pre[u] $\leftarrow T$; $T \leftarrow T + 1$
2: markiere u
3: **for** Nachvolger v von u, unmarkiert **do**
4: Visit(v)
5: post[u] $\leftarrow T$; $T \leftarrow T + 1$

Algorithm 4 DFS(G)

1: $T \leftarrow 1$
2: alle Knoten unmarkiert
3: **for** $u_0 \in V$, unmarkiert **do**
4: Visit(u_0)

Visit(u): Visited ← ∅ T ← empty list

markiere u Visited ← Visited ∪ {u}

For Nachfolger v, unmarkiert:

| Visit(v)

Füge u zur top. Sort. hinzu T ← T ∪ {u}

DFS

DFS(G): (Tiefensuche, Rahmenprogramm)

alle Knoten unmarkiert

for u ∈ V if u ∉ visited

For $u_0 \in V$, unmarkiert:

Visit(u_0)

Laufzeit analyse

Adjazenzmatrix

Anzahl Visit Aufrufe: n (einen Aufruf pro Knoten)

Zeit für Visit(u): $O(n)$ + Zeit in rekursiven Aufrufen

Total: $n \cdot O(n) = O(n^2)$ (laufe über alle Nachfolger von u)

Adjazenzliste

Anzahl Visit Aufrufe: n

Zeit für Visit(u): $O(1 + \deg_{out}(u))$ + Rekursion

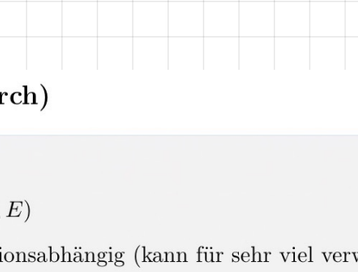
Total:

$$\sum_{u \in V} (1 + \deg_{out}(u)) = |V| + \sum_{u \in V} \deg_{out}(u)$$

$|E|$ (Handschlaglemma: in ungerichtet $\deg(u) = 2|E|$; in gerichtet $\deg_{out}(u) = |E|$)

$\rightsquigarrow$ Tiefensuche hat Laufzeit $O(|V| + |E|)$ für Adjazenzliste

Beispiel: pre/post

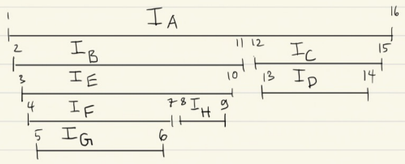


pre-order: ABEFGHCD

post-order: GFHEBDCA

vorherige Vorlesung: falls G. azyklisch, dann ist umgekehrte post-order eine topologische Sortierung

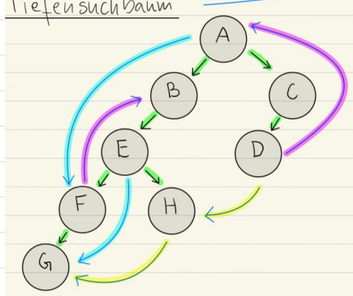
Bildlich: Intervall $I_u = \{pre[u], \dots, post[u]\}$



unmassstablich

Verschachtelung entspricht Rekursionsbaum

Tiefensuchbaum

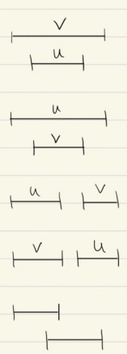


eigentlich: Tiefensuchwald

Kanten klassifizieren

im Tiefensuchbaum: tree (kann auch wald sein)

jede andere Kante $(u,v) \in E$: anhand I_u und I_v (pre/post Zahlen von u und v)



back z.B.: (F,B), (D,A)

forward (wenn nicht tree) z.B.: (A,F) (E,G)

nicht möglich! (Visit(u) würde Visit(v) aufrufen da v unmarkierter Nachfolger von u)

cross z.B.: (H,G), (D,H)

nicht möglich! (Aufruf endet erst wenn alle rekursiven Aufrufe beendet)

Beobachtungen

(1) ∃ back Kante ⇒ ∃ gerichteter Zyklus

(2) ∀ nicht-back $(u,v) \in E$. post(u) > post(v) (haben wesentlich gebraucht)

(1),(2) $\rightsquigarrow$ ∄ gerichteter Zyklus ⇒ ∄ back Kante

(2) ⇒ Kante $\in E$ post[u] > post[v]

⇒ umgekehrte post-order ist topologische Sortierung

Kürzer: Korrektheitsbeweis für DFS zum topologischen Sortieren

Algorithm 15 Depth-first search, imperative style

$S \leftarrow$ **new** stack()
S.PUSH(r)
$D \leftarrow \{r\}$
while $\neg S$.ISEMPTY() **do**
  $v \leftarrow S$.POP()
  /*do something with v*/
  for w s.t. v and w are adjacent in G **do**
    if $w \notin D$ **then**
      S.PUSH(w)
      $D \leftarrow D \cup \{w\}$

The second one is recursive, more compact and in some sense more natural:

Algorithm 16 Depth-first search, recursive style

function DFS(v, G, $D = \emptyset$)
  /*do something with v*/
  for w s.t. v and w are adjacent in G **do**
    if $w \notin D$ **then**
      $D \leftarrow D \cup \{w\}$
      DFS(w, G, D)
DFS(s, G)

Finding strongly connected vertices u-v (finding cycle) there is a path from u to v and v to u

Algorithm 1

```
1:  Input: integer n. Adjacency list Adj[1 ... n].
2:
3:  Let status[1 ... n] be a global array, with all entries initialized to UNVISITED.
4:
5:  function visit(u)
6:      status[u] ← VISITING
7:      for each v in Adj[u] do                    ▷ Iterate over all neighbours v.
8:          if status[v] = VISITING then           ▷ There is a directed cycle containing u and v.
9:              Output (u, v) and terminate
10:         if status[v] = UNVISITED then
11:             visit(v)
12:     status[u] ← VISITED.
13: for u = 1, 2, ..., n do
14:     if status[u] = UNVISITED then
15:         visit[u]
16: Output "no strongly connected vertices exist"
```

distances

0	1	2	3	4	5	6	7
0	1	1	1	1	2	2	2

Q

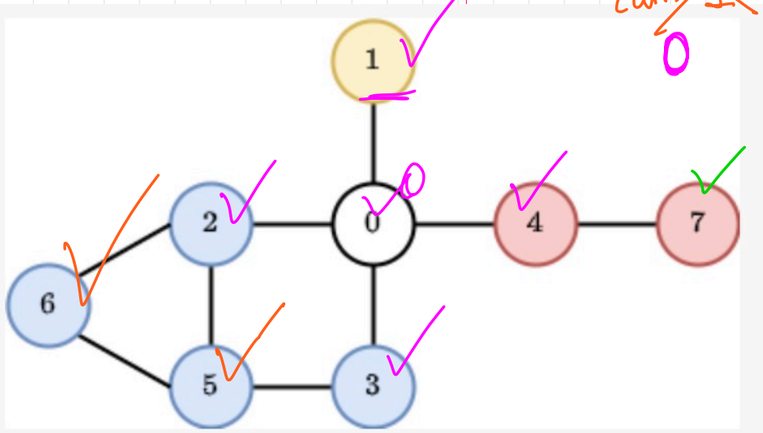
cur = 1

0

cur 2

cur 3

cur 4



Graph Searchs

DFS - with pre and post order

Runtime : O (|V| + |E|)

Algorithm 4 DFS(G)

```
1: T ← 1
2: alle Knoten unmarkiert
3: for u_0 ∈ V, unmarkiert do
4:     Visit(u_0)
```

Algorithm 3 Visit(u)

```
1: pre[u] ← T; T ← T + 1
2: markiere u
3: for Nachvolger v von u, unmarkiert do
4:     Visit(v)
5: post[u] ← T; T ← T + 1
```

Graph Searchs

DFS - Lemmas, Facts

∃ a back edge ⟺ ∃ a directed closed walk

For all edges (u,v) in E except back edges : post(u) > post(v)

Reversed post-order is the topological ordering !!!

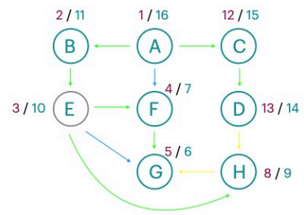
Topological Sorting

Reversed post-order

post-order :
G, F, H, E, B, D, C, A

reversed post-order :
A, C, D, B, E, H, F, G

topological sort



pre-order
post-order

tree edge

back edge
forward edge
cross edge

Topological Sorting

Lemmas, Facts

Term (German)	Term (English)	Definition
Quelle	Source	Vertex with only outgoing edges (in-degree = 0).
Senke	Sink	Vertex with only incoming edges (out-degree = 0).

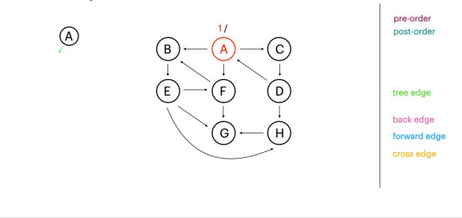
∃ a topological sorting ⟺ G is a DAG (Directed Acyclic Graph)

Topological Sorting doesn't have to be unique, there can be multiple valid orders depending on the graph's structure.

1

Graph Searchs

DFS - Example



pre-order
post-order

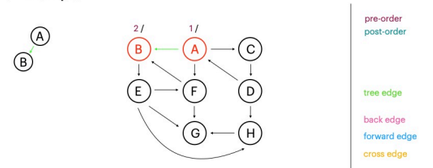
tree edge

back edge
forward edge
cross edge

2

Graph Searchs

DFS - Example



pre-order
post-order

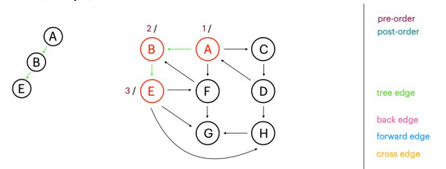
tree edge

back edge
forward edge
cross edge

3

Graph Searchs

DFS - Example



pre-order
post-order

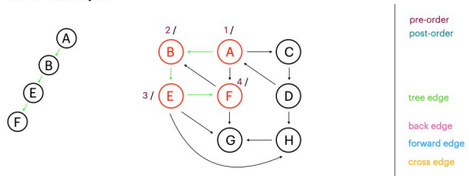
tree edge

back edge
forward edge
cross edge

4

Graph Searchs

DFS - Example



pre-order
post-order

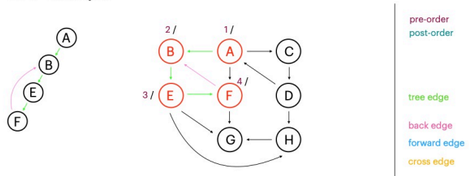
tree edge

back edge
forward edge
cross edge

5

Graph Searchs

DFS - Example



pre-order
post-order

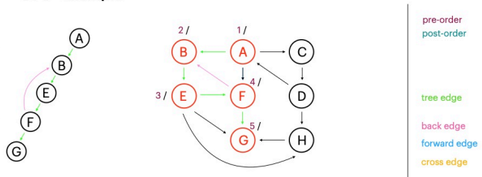
tree edge

back edge
forward edge
cross edge

6

Graph Searchs

DFS - Example



pre-order
post-order

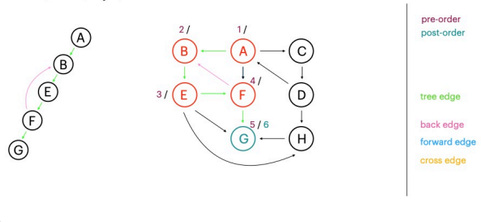
tree edge

back edge
forward edge
cross edge

7

Graph Searchs

DFS - Example



pre-order
post-order

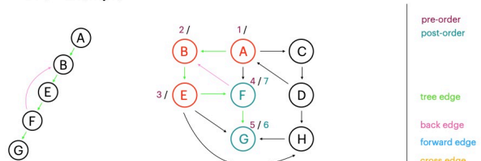
tree edge

back edge
forward edge
cross edge

8

Graph Searchs

DFS - Example



pre-order
post-order

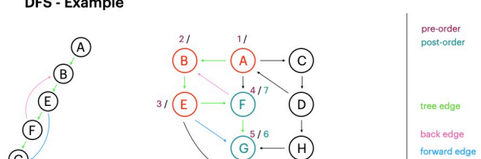
tree edge

back edge
forward edge
cross edge

9

Graph Searchs

DFS - Example



pre-order
post-order

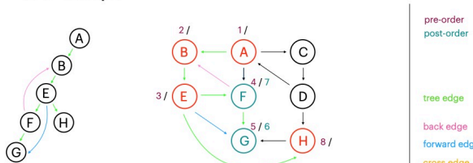
tree edge

back edge
forward edge
cross edge

10

Graph Searchs

DFS - Example



pre-order
post-order

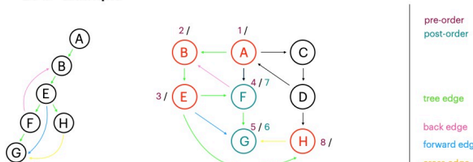
tree edge

back edge
forward edge
cross edge

11

Graph Searchs

DFS - Example



pre-order
post-order

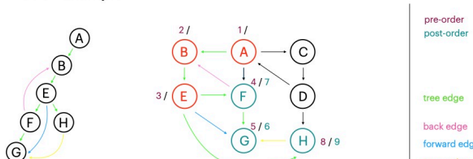
tree edge

back edge
forward edge
cross edge

12

Graph Searchs

DFS - Example



pre-order
post-order

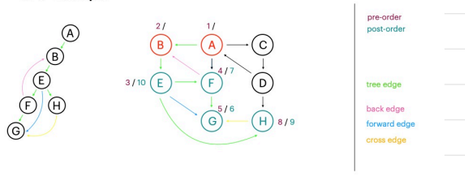
tree edge

back edge
forward edge
cross edge

13

Graph Searchs

DFS - Example



pre-order
post-order

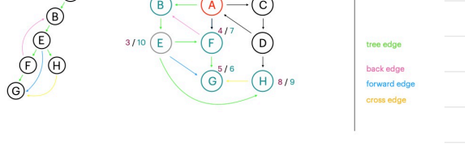
tree edge

back edge
forward edge
cross edge

14

Graph Searchs

DFS - Example



pre-order
post-order

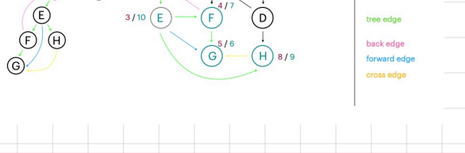
tree edge

back edge
forward edge
cross edge

15

Graph Searchs

DFS - Example



pre-order
post-order

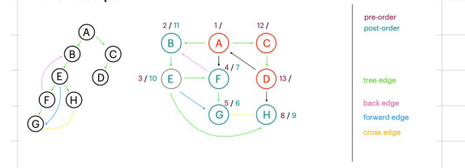
tree edge

back edge
forward edge
cross edge

16

Graph Searchs

DFS - Example



pre-order
post-order

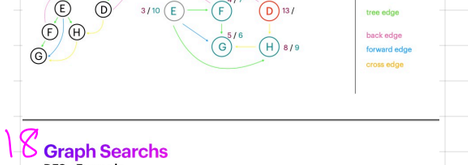
tree edge

back edge
forward edge
cross edge

17

Graph Searchs

DFS - Example



pre-order
post-order

tree edge

back edge
forward edge
cross edge

18

Graph Searchs

DFS - Example

pre-order
post-order

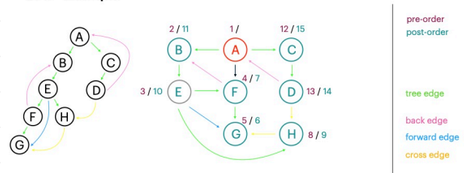
tree edge

back edge
forward edge
cross edge

19

Graph Searchs

DFS - Example



pre-order
post-order

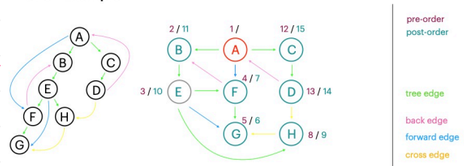
tree edge

back edge
forward edge
cross edge

20

Graph Searchs

DFS - Example



pre-order
post-order

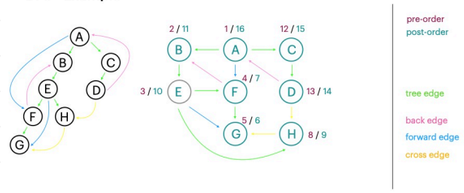
tree edge

back edge
forward edge
cross edge

21

Graph Searchs

DFS - Example



pre-order
post-order

tree edge

back edge
forward edge
cross edge

BFS

BFS (Breadth-First-Search)

BFS(s):

- Input: Knoten $s \in V$
- Output: Implementationsabhängig (kann für sehr viel verwendet werden, ähnlich wie DFS)
- Laufzeit: $\mathcal{O}(n+m)$

Algorithm 5 BFS(s)

```
1: Q ← {s}                                          ▷ Q ist eine Queue
2: enter[s] ← 0;   T ← 1
3: while Q ≠ ∅ do
4:     u ← dequeue(Q)
5:     leave[u] ← T;   T ← T + 1
6:     for (u, v) ∈ E, enter[v] nicht zugewiesen do
7:         enqueue(Q, v)
8:         enter[v] ← T;   T ← T + 1
```

Graph Searchs

BFS - with pre and post order + distances

Runtime : O (|V| + |E|)

Algorithm 5 BFS(s)

```
1: Q ← {s}                                          Q is a FIFO queue
2: enter[s] ← 0;   T ← 1   distance[s] = 0 ;
3: while Q ≠ ∅ do
4:     u ← dequeue(Q)
5:     leave[u] ← T;   T ← T + 1
6:     for (u, v) ∈ E, enter[v] nicht zugewiesen do
7:         enqueue(Q, v)
8:         enter[v] ← T;   T ← T + 1   distance[v] <- distance[u] + 1 ;
```

Algorithm 14 Breadth-first search

```
Q ← new queue ()
Q.PUSH (r)
D ← {r}                          ▷ Stores nodes that are done (in Q or already processed)
while ¬Q.ISEMPTY () do
    v ← Q.POP ()
    /*do something with v*/
    for w s.t. v and w are adjacent in G do
        if w ∉ D then
            Q.PUSH (w)
            D ← D ∪ {w}
```

1 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : A

u = A

enter[] :	A	B	C	D	E	F	G	H
	0							

leave[] :	A	B	C	D	E	F	G	H

distance[] :	A	B	C	D	E	F	G	H
	0							

2 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q :

u = A

enter[] :	A	B	C	D	E	F	G	H
	0							

leave[] :	A	B	C	D	E	F	G	H
	1							

distance[] :	A	B	C	D	E	F	G	H
	0							

3 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q :

u = A

enter[] :	A	B	C	D	E	F	G	H
	0							

leave[] :	A	B	C	D	E	F	G	H
	1							

distance[] :	A	B	C	D	E	F	G	H
	0							

4 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : B

u = A

enter[] :	A	B	C	D	E	F	G	H
	0	2						

leave[] :	A	B	C	D	E	F	G	H
	1							

distance[] :	A	B	C	D	E	F	G	H
	0	1						

5 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : B - C

u = A

enter[] :	A	B	C	D	E	F	G	H
	0	2	3					

leave[] :	A	B	C	D	E	F	G	H
	1							

distance[] :	A	B	C	D	E	F	G	H
	0	1	1					

6 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : B - C - F

u = A

enter[] :	A	B	C	D	E	F	G	H
	0	2	3			4		

leave[] :	A	B	C	D	E	F	G	H
	1							

distance[] :	A	B	C	D	E	F	G	H
	0	1	1			1		

7 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : B - C - F

enter[] :	A	B	C	D	E	F	G	H
	0	2	3			4		

leave[] :	A	B	C	D	E	F	G	H
	1							

distance[] :	A	B	C	D	E	F	G	H
	0	1	1			1		

8 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : B - C - F

FIFO

enter[] :	A	B	C	D	E	F	G	H
	0	2	3			4		

leave[] :	A	B	C	D	E	F	G	H
	1							

distance[] :	A	B	C	D	E	F	G	H
	0	1	1			1		

9 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : C - F

u = B

enter[] :	A	B	C	D	E	F	G	H
	0	2	3			4		

leave[] :	A	B	C	D	E	F	G	H
	1	5						

distance[] :	A	B	C	D	E	F	G	H
	0	1	1			1		

10 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : C - F

u = B

enter[] :	A	B	C	D	E	F	G	H
	0	2	3			4		

leave[] :	A	B	C	D	E	F	G	H
	1	5						

distance[] :	A	B	C	D	E	F	G	H
	0	1	1			1		

11 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : C - F - E

u = B

enter[] :	A	B	C	D	E	F	G	H
	0	2	3		6	4		

leave[] :	A	B	C	D	E	F	G	H
	1	5						

distance[] :	A	B	C	D	E	F	G	H
	0	1	1		2	1		

12 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : C - F - E

enter[] :	A	B	C	D	E	F	G	H
	0	2	3		6	4		

leave[] :	A	B	C	D	E	F	G	H
	1	5						

distance[] :	A	B	C	D	E	F	G	H
	0	1	1		2	1		

13 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : C - F - E

enter[] :	A	B	C	D	E	F	G	H
	0	2	3		6	4		

leave[] :	A	B	C	D	E	F	G	H
	1	5						

distance[] :	A	B	C	D	E	F	G	H
	0	1	1		2	1		

14 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : F - E

u = C

enter[] :	A	B	C	D	E	F	G	H
	0	2	3		6	4		

leave[] :	A	B	C	D	E	F	G	H
	1	5	7					

distance[] :	A	B	C	D	E	F	G	H
	0	1	1		2	1		

15 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : F - E

u = C

enter[] :	A	B	C	D	E	F	G	H
	0	2	3		6	4		

leave[] :	A	B	C	D	E	F	G	H
	1	5	7					

distance[] :	A	B	C	D	E	F	G	H
	0	1	1		2	1		

16 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : F - E - D

u = C

enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4		

leave[] :	A	B	C	D	E	F	G	H
	1	5	7					

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1		

17 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : F - E - D

enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4		

leave[] :	A	B	C	D	E	F	G	H
	1	5	7					

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1		

18 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : F - E - D

enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4		

leave[] :	A	B	C	D	E	F	G	H
	1	5	7					

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1		

19 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : E - D

u = F

enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4		

leave[] :	A	B	C	D	E	F	G	H
	1	5	7			9		

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1		

20 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : E - D

u = F

enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4		

leave[] :	A	B	C	D	E	F	G	H
	1	5	7			9		

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1		

21 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : E - D - G

u = F

enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	

leave[] :	A	B	C	D	E	F	G	H
	1	5	7			9		

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	

22 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : E - D

u = F

enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4		

leave[] :	A	B	C	D	E	F	G	H
	1	5	7			9		

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1		

23 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : E - D

u = F

enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4		

leave[] :	A	B	C	D	E	F	G	H
	1	5	7			9		

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1		

24 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : E - D - G

u = F

enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	

leave[] :	A	B	C	D	E	F	G	H
	1	5	7			9		

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	

25 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : D - G

u = E

enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	

leave[] :	A	B	C	D	E	F	G	H
	1	5	7		11	9		

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	

26 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : D - G - H

u = E

enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	12

leave[] :	A	B	C	D	E	F	G	H
	1	5	7		11	9		

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	3

27 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : D - G - H

enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	12

leave[] :	A	B	C	D	E	F	G	H
	1	5	7		11	9		

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	3

28 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : D - G - H

enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	12

leave[] :	A	B	C	D	E	F	G	H
	1	5	7		11	9		

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	3

29 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : G - H

u = D

enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	12

leave[] :	A	B	C	D	E	F	G	H
	1	5	7	13	11	9		

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	3

30 Graph Searchs
BFS - Example

Algorithm 5 BFS(s)
1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] ← distance[u] + 1 ;

Q : G - H

u = D

enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	12

leave[] :	A	B	C	D	E	F	G	H
	1	5	7	13	11	9		

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	3

31

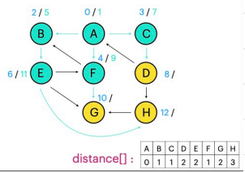
Graph Searchs
BFS - Example

Algorithm 5 BFS(s)

1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] <- distance[u] + 1 ;

Q : D - G - H



enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	12

leave[] :	A	B	C	D	E	F	G	H
	1	5	7		11	9		

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	3

32

Graph Searchs
BFS - Example

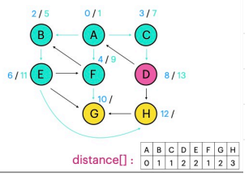
Algorithm 5 BFS(s)

1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] <- distance[u] + 1 ;

Q : G - H

u = D



enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	12

leave[] :	A	B	C	D	E	F	G	H
	1	5	7	13	11	9		

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	3

33

Graph Searchs
BFS - Example

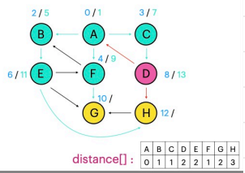
Algorithm 5 BFS(s)

1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] <- distance[u] + 1 ;

Q : G - H

u = D



enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	12

leave[] :	A	B	C	D	E	F	G	H
	1	5	7	13	11	9		

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	3

34

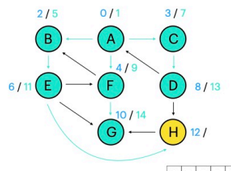
Graph Searchs
BFS - Example

Algorithm 5 BFS(s)

1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] <- distance[u] + 1 ;

Q : H



enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	12

leave[] :	A	B	C	D	E	F	G	H
	1	5	7	13	11	9	14	

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	3

35

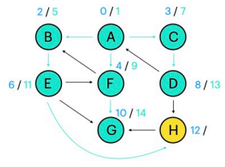
Graph Searchs
BFS - Example

Algorithm 5 BFS(s)

1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] <- distance[u] + 1 ;

Q : H



enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	12

leave[] :	A	B	C	D	E	F	G	H
	1	5	7	13	11	9	14	

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	3

36

Graph Searchs
BFS - Example

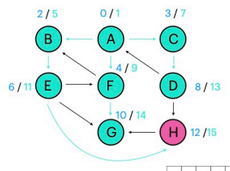
Algorithm 5 BFS(s)

1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] <- distance[u] + 1 ;

Q :

u = H



enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	12

leave[] :	A	B	C	D	E	F	G	H
	1	5	7	13	11	9	14	15

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	3

37

Graph Searchs
BFS - Example

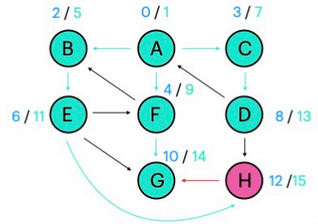
Algorithm 5 BFS(s)

1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] <- distance[u] + 1 ;

Q :

u = H



enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	12

leave[] :	A	B	C	D	E	F	G	H
	1	5	7	13	11	9	14	15

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	3

38

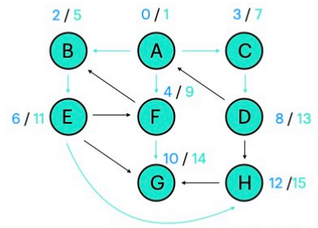
Graph Searchs
BFS - Example

Algorithm 5 BFS(s)

1: $Q \leftarrow \{s\}$
2: enter[s] $\leftarrow 0$; $T \leftarrow 1$
distance[s] = 0 ;

3: **while** $Q \neq \varnothing$ **do**
4: $u \leftarrow$ dequeue(Q)
5: leave[u] $\leftarrow T$; $T \leftarrow T+1$
6: **for** $(u,v) \in E$, enter[v] nicht zugewiesen **do**
7: enqueue(Q, v)
8: enter[v] $\leftarrow T$; $T \leftarrow T+1$
distance[v] <- distance[u] + 1 ;

Q : **Q = ∅**



enter[] :	A	B	C	D	E	F	G	H
	0	2	3	8	6	4	10	12

leave[] :	A	B	C	D	E	F	G	H
	1	5	7	13	11	9	14	15

distance[] :	A	B	C	D	E	F	G	H
	0	1	1	2	2	1	2	3

Minimum Spanning Tree

- Definition
- Prim's Algo
- Boruvka's Algo
- Kruskal's Algo

Definition:

MST: Minimal Spanning Tree of a $G(V,E)$:

is a tree T with vertices $V(T) = V$

and edges $E(T) \subset E$ s.t.

$\sum_{e \in E} w(e)$ is minimal among all such trees

Minimum Spanning Tree

- Among all spanning trees, MST has the minimum total weight
  (smallest possible sum of edge weights)
- Spans all the vertices in the graph
- Every vertex is included in the tree.
- no cycles
- |V| - 1 edges

Note for time complexity:

we have: $m \leq n^2$

and therefore

$\log(m) \leq \log(n^2) = 2\log(n) = O(\log n)$

Boruvka's Algorithm

Boruvka(G):

- Input: Graph $G = (V, E)$
- Output: Minimaler Spannbaum F
- Laufzeit: $\mathcal{O}((m+n) \cdot log(n))$ (wie Dijkstra)

$O(m \log n)$

Idea Constructs a spanning forest iteratively until it becomes a spanning tree. Starts with each vertex in a distinct component (a trivial tree with one vertex and zero edge). All vertices select their nearest neighbor simultaneously, and all corresponding edges are added. This results in a number of trees being formed. Then, all of these trees select again their smallest outgoing edge, which are added to the forest, dividing the number of trees by two at each iteration. The process goes on for at most $\Theta(\log n)$ rounds until only one connected component remains.

Algorithm 8 Boruvka(G)

1: $F \leftarrow \varnothing$ ▷ sichere Kanten
2: **while** F nicht Spannbaum **do** ▷ $\leq log(n)$ Iterationen, $\mathcal{O}(m+n)$ pro Iteration
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$ → Disjunkte Vereinigung

MST

Boruvka's Algorithm Runtime : O ((|V| + |E|) * log n)

Algorithm 8 Boruvka(G)

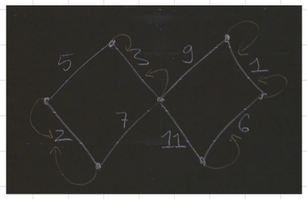
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do** — find with DFS only using the edges in F !
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F — connected components with edges from F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$ — minimum edges near Ss

F : edges of the MST

DFS für ZHK

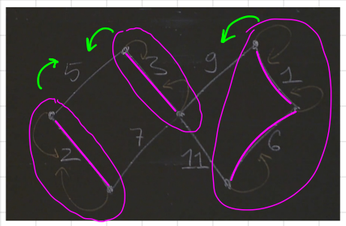
Bsp:

Anfang:

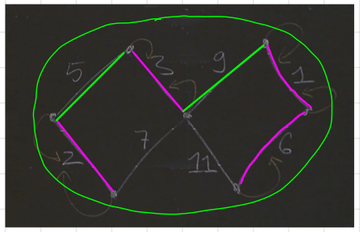


↶ : minimale Kanten für jede Knoten

füge alle Sicheren Kanten zu F hinzu



In 2. Iteration, die ZHK sind diese



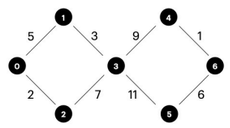
1

MST
Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : ∅



2

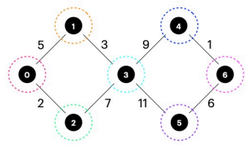
MST
Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : ∅

$S_1 = \{0\}$
$S_2 = \{1\}$
$S_3 = \{2\}$
$S_4 = \{3\}$
$S_5 = \{4\}$
$S_6 = \{5\}$
$S_7 = \{6\}$



3

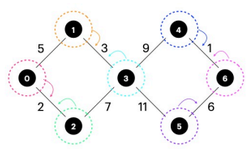
MST
Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : ∅

$S_1 = \{0\}$
$S_2 = \{1\}$
$S_3 = \{2\}$
$S_4 = \{3\}$
$S_5 = \{4\}$
$S_6 = \{5\}$
$S_7 = \{6\}$



4

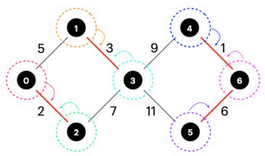
MST
Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : ∅

$S_1 = \{0\}$
$S_2 = \{1\}$
$S_3 = \{2\}$
$S_4 = \{3\}$
$S_5 = \{4\}$
$S_6 = \{5\}$
$S_7 = \{6\}$



5

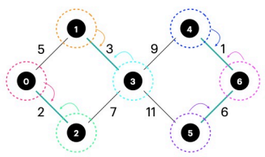
MST
Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : { {0,2} , {1,3} , {4,6} , {5,6} }

$S_1 = \{0\}$
$S_2 = \{1\}$
$S_3 = \{2\}$
$S_4 = \{3\}$
$S_5 = \{4\}$
$S_6 = \{5\}$
$S_7 = \{6\}$



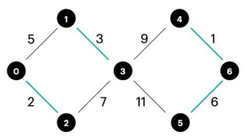
6

MST
Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : { {0,2} , {1,3} , {4,6} , {5,6} }



7

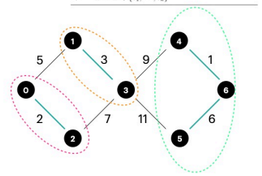
MST
Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : { {0,2} , {1,3} , {4,6} , {5,6} }

$S_1 = \{0,2\}$
$S_2 = \{1,3\}$
$S_3 = \{4,5,6\}$



8

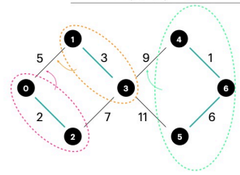
MST
Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : { {0,2} , {1,3} , {4,6} , {5,6} }

$S_1 = \{0,2\}$
$S_2 = \{1,3\}$
$S_3 = \{4,5,6\}$



9

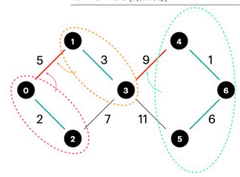
MST
Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : { {0,2} , {1,3} , {4,6} , {5,6} }

$S_1 = \{0,2\}$
$S_2 = \{1,3\}$
$S_3 = \{4,5,6\}$



10

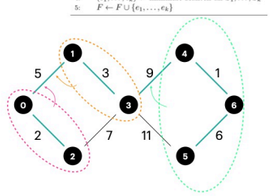
MST
Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : { {0,2} , {1,3} , {4,6} , {5,6} , {0,1} , {4,3} }

$S_1 = \{0,2\}$
$S_2 = \{1,3\}$
$S_3 = \{4,5,6\}$



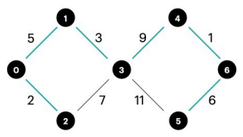
11

MST
Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : { {0,2} , {1,3} , {4,6} , {5,6} , {0,1} , {4,3} }



12

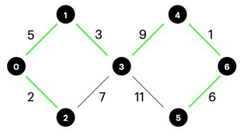
MST
Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : { {0,2} , {1,3} , {4,6} , {5,6} , {0,1} , {4,3} }

F is a spanning tree



Prim's Algo

Idee: Konzentrieren uns auf eine ZHK

Prim(G, s)**:**

- Input: Graph $G = (V, E)$ und $s \in V$
- Output: Minimaler Spannbaum F
- Laufzeit: $\mathcal{O}((m+n) \cdot log(n))$ (wie Dijkstra und Boruvka)

O(mlogn)

Idea A variant of Dijkstra's algorithm, where for each vertex v not already processed, we keep the cost of the shortest edge connecting v to the tree under construction. Just as in Dijkstra, extends the current tree with the edge $e = (w, v)$ and vertex v of minimal cost, updating the costs of neighbors of v. Repeats until all vertices are covered.

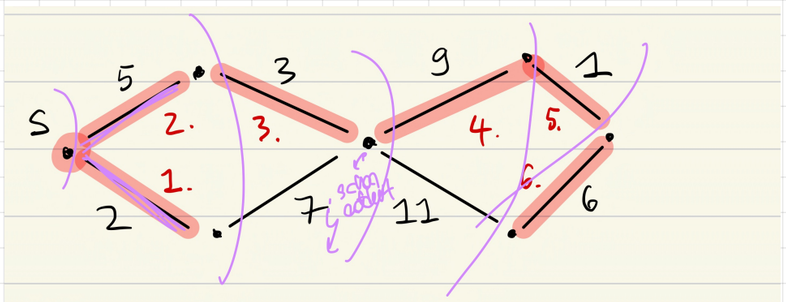
Algorithm 9 Prim(G, s) (allgemeine Form)

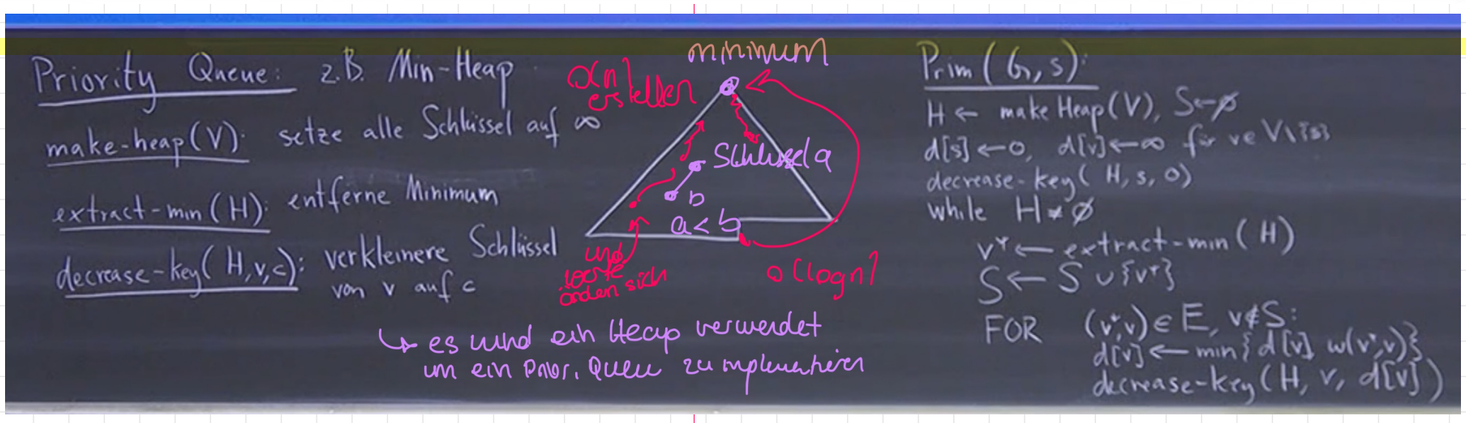
1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$ ▷ ZHK von s in F
3: **while** F nicht Spannbaum **do**
4: $u^*v^* \leftarrow$ minimale Kante an S $(u^* \in S, v^* \notin S)$
5: $F \leftarrow F \cup \{u^*v^*\}$
6: $S \leftarrow S \cup \{v^*\}$

Algorithm 10 Prim(G, s) (mit min-heap)

1: $H \leftarrow$ make-heap(V, ∞), $S \leftarrow \varnothing$
2: d$[s] \leftarrow 0$; d$[v] \leftarrow \infty$ $\forall v \in V \setminus \{s\}$
3: decrease-key$(H, s, 0)$
4: **while** $H \neq \varnothing$ **do**
5: $v^* \leftarrow$ extract-min(H)
6: $S \leftarrow S \cup \{v^*\}$
7: **for** $v^*v \in E$, $v \notin S$ **do**
8: d$[v] \leftarrow \min\{$d$[v], c(v^*, v)\}$ ▷ Unterschied zu Dijkstra
9: decrease-key$(H, v,$ d$[v])$

example :





MST

Prim's Algorithm - Dijkstra approach

Runtime : O ((|V| + |E|) * log n)

Algorithm 6 Dijkstra(s)

1: d$[s] \leftarrow 0$; d$[v] \leftarrow \infty$ $\forall v \in V \setminus \{s\}$
2: $S \leftarrow \varnothing$
3: $H \leftarrow$ make-heap(V); decrease-key$(H, s, 0)$
4: **while** $S \neq V$ **do**
5: $v^* \leftarrow$ extract-min(H)
6: $S \leftarrow S \cup \{v^*\}$
7: **for** $(v^*, v) \in E$, $v \notin S$ **do**
8: d$[v] \leftarrow \min\{$d$[v],$ d$[v^*] + c(v^*, v)\}$
9: decrease-key$(H, v,$ d$[v])$

Algorithm 10 Prim(G, s) (mit min-heap)

1: $H \leftarrow$ make-heap(V, ∞), $S \leftarrow \varnothing$
2: d$[s] \leftarrow 0$; d$[v] \leftarrow \infty$ $\forall v \in V \setminus \{s\}$
3: decrease-key$(H, s, 0)$
4: **while** $H \neq \varnothing$ **do**
5: $v^* \leftarrow$ extract-min(H)
6: $S \leftarrow S \cup \{v^*\}$
7: **for** $v^*v \in E$, $v \notin S$ **do**
8: d$[v] \leftarrow \min\{$d$[v], c(v^*, v)\}$
9: decrease-key$(H, v,$ d$[v])$

MST

Prim's Algorithm - connected components approach

Runtime : O ((|V| + |E|) * log n)

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $u^*v^* \leftarrow$ minimale Kante an S $(u^* \in S, v^* \notin S)$
5: $F \leftarrow F \cup \{u^*v^*\}$
6: $S \leftarrow S \cup \{v^*\}$

F : edges of the MST

S : connected component set

4 : find the minimum edge {u*,v*} s.t. u* is in S but v* is not

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $F \leftarrow F \cup \{u^*v^*\}$
6: $S \leftarrow S \cup \{v^*\}$

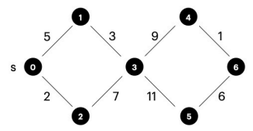
F : edges of the MST
S : connected component set
4 : find the minimum edge (u^*,v^*) s.t. u^* is in S but v^* is not
edges (u^*,v^*) s.t. u^* is in S but v^* is not

---

1 **MST**
Prim's Algorithm

F : ∅

S : { 0 }

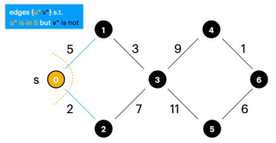


---

2 **MST**
Prim's Algorithm

F : ∅

S : { 0 }

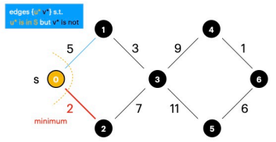


---

3 **MST**
Prim's Algorithm

F : ∅

S : { 0 }

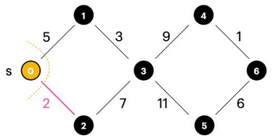


---

4 **MST**
Prim's Algorithm

F : { {0,2} }

S : { 0 }

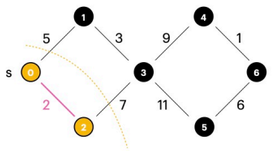


---

5 **MST**
Prim's Algorithm

F : { {0,2} }

S : { 0 , 2 }

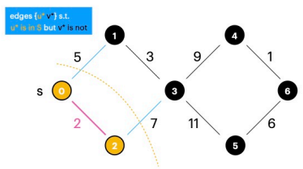


---

6 **MST**
Prim's Algorithm

F : { {0,2} }

S : { 0 , 2 }

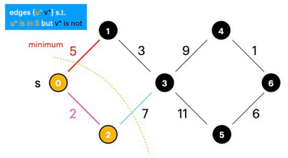


---

7 **MST**
Prim's Algorithm

F : { {0,2} }

S : { 0 , 2 }

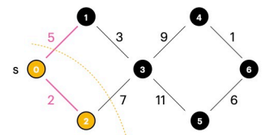


---

8 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} }

S : { 0 , 2 }

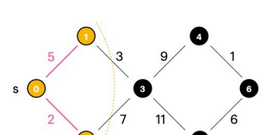


---

9 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} }

S : { 0 , 2 , 1 }

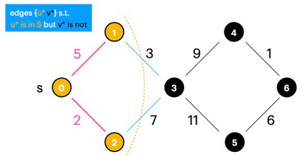


---

10 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} }

S : { 0 , 2 , 1 }

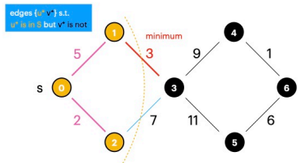


---

11 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} }

S : { 0 , 2 , 1 }

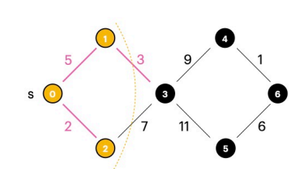


---

12 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} }

S : { 0 , 2 , 1 }

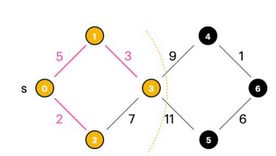


---

13 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} }

S : { 0 , 2 , 1 , 3 }

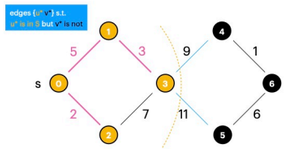


---

14 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} }

S : { 0 , 2 , 1 , 3 }

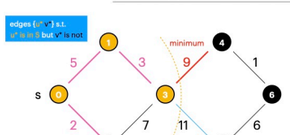


---

15 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} }

S : { 0 , 2 , 1 , 3 }

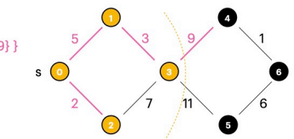


---

16 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} , {3,9} }

S : { 0 , 2 , 1 , 3 }

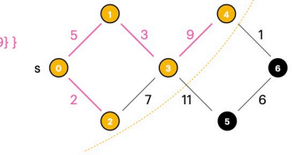


---

17 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} , {3,9} }

S : { 0 , 2 , 1 , 3 , 4 }

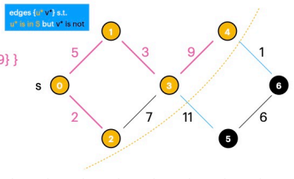


---

18 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} , {3,9} }

S : { 0 , 2 , 1 , 3 , 4 }

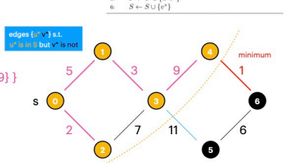


---

19 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} , {3,9} }

S : { 0 , 2 , 1 , 3 , 4 }

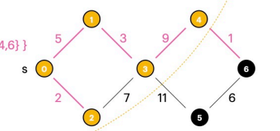


---

20 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} }

S : { 0 , 2 , 1 , 3 , 4 }

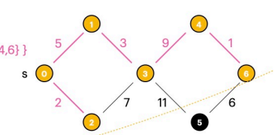


---

21 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} }

S : { 0 , 2 , 1 , 3 , 4 , 6 }

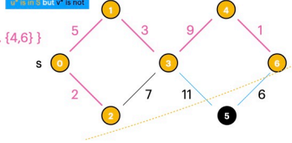


---

22 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} }

S : { 0 , 2 , 1 , 3 , 4 , 6 }

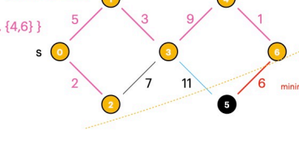


---

23 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} }

S : { 0 , 2 , 1 , 3 , 4 , 6 }

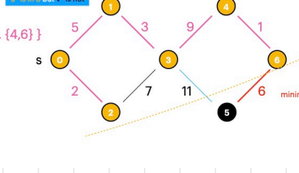


---

24 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} }

S : { 0 , 2 , 1 , 3 , 4 , 6 }

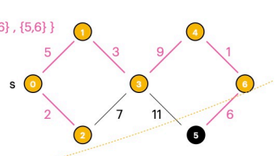


---

25 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} , {5,6} }

S : { 0 , 2 , 1 , 3 , 4 , 6 }

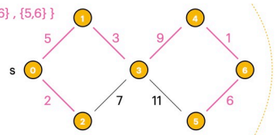


---

26 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} , {5,6} }

S : { 0 , 2 , 1 , 3 , 4 , 6 , 5 }



---

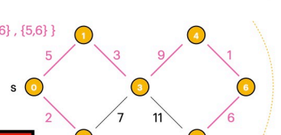
27 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} , {5,6} }

S : { 0 , 2 , 1 , 3 , 4 , 6 , 5 }

S = V

F is a spanning tree



---

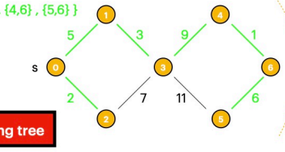
28 **MST**
Prim's Algorithm

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} , {5,6} }

S : { 0 , 2 , 1 , 3 , 4 , 6 , 5 }

S = V

F is a spanning tree



Kruskal's Algorithm

Idee: sichere Kanten sortiert nach Gewicht

Kruskal(G):

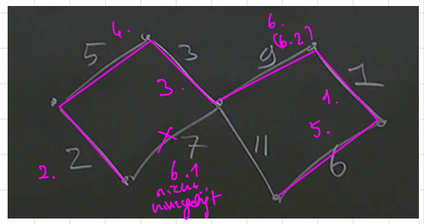
- Input: Graph $G = (V, E)$ und $s \in V$
- Output: Minimaler Spannbaum F
- Brauchen eine Union-Find Datenstruktur für effiziente Laufzeit
- Laufzeit: $\mathcal{O}(\underbrace{m \cdot log(m)}_{\text{Sortieren}} + \underbrace{n \cdot log(n)}_{\text{Union-Find}})$ (↳ genauere Laufzeit Analyse in L13 notes)

Idea Sorts edges by increasing order of weight and tries to add them in this order, abstaining from it whenever adding the new edge would result in a cycle. The final set of edges is a spanning tree.

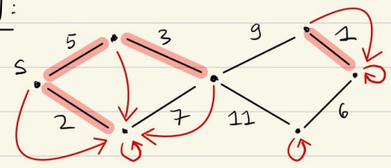
Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

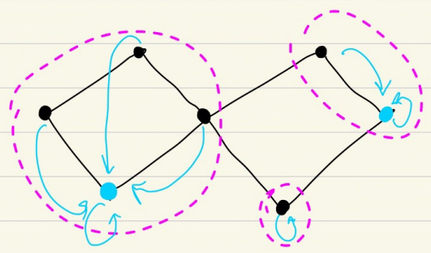
```
1: F ← ∅
2: UF ← MAKE(V)                          ▷ UF-Datenstruktur initialisieren
3: SORT(E)                               ▷ Sortiere Kanten nach Gewicht
4: for uv ∈ E, aufsteigend sortiert do
5:     if SAME(u,v) = false then         ▷ u, v in verschiedenen ZHKs von F
6:         F ← F ∪ {uv}
7:         UNION(u,v)
```

Bsp:



Beispiel:





repräsentant

Union-Find DS

Operationen:

- make(V): erstelle Datenstruktur für $F = \emptyset$ (↳ initializing)
- same(u, v): teste ob u, v in derselben ZHK von F
- union(u, v): vereinige ZHKs von u und v (füge Kante uv zu F hinzu)

Union-find Datenstruktur

Speicher:

- rep[v]: eindeutiger Repräsentant für ZHK(v)
- members[v]: Liste alle Knoten in ZHK(v) falls v der Repräsentant

make(V): rep[v] ← v, members[v] ← {v}, ∀v∈V

same(u,v): teste ob rep[u] = rep[v]

union(u,v):

FOR x ∈ members[rep[u]]:
  rep[x] ← rep[v]
  members[rep[v]] = members[rep[v]] ∪ {x}

Laufzeit: $\mathcal{O}(|ZHK(u)|)$

Idee: durchlaufe nur Knoten der kleineren ZHK

union Laufzeit:

$\Theta(\min\{|ZHK(u)|, |ZHK(v)|\})$

Speicher:

- rep[v]: eindeutiger Repräsentant der ZHK von V (↳ $rep[u] = rep[v] \Leftrightarrow ZHK(u) = ZHK(v)$)

Algorithm 11 Union-Find(G)

```
1: Implementierung:
2: MAKE(V):    rep[v] ← v  ∀v ∈ V                          ▷ O(n)
3:
4: SAME(u,v):  teste ob rep[u] = rep[v]                    ▷ O(1)
5:
6: UNION(u,v):                                             ▷ O(|ZHK(u)|)
7: for x ∈ members[rep[u]] do
8:     rep[x] ← rep[v]
9:     members[rep[v]] ← members[rep[v]] ∪ {x}
```

MST

Kruskal's Algorithm

Runtime : O ((|V| + |E|) * log n)

Algorithm 11 Union-Find(G)
1: **Implementierung:**
2: MAKE(V): rep[v] ← v ∀v ∈ V
3:
4: SAME(u,v): teste ob rep[u] = rep[v]
5:
6: UNION(u,v):
7: **for** $x \in$ members[rep[u]] **do**
8: rep[x] ← rep[v]
9: members[rep[v]] ← members[rep[v]] ∪ {x}

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

```
1: F ← ∅   edges of the MST
2: UF ← MAKE(V)   UF : Union Find
3: SORT(E)   sort the edges in increasing order
4: for uv ∈ E, aufsteigend sortiert do
5:     if SAME(u,v) = false then   if its not in the same concomp
6:         F ← F ∪ {uv}
7:         UNION(u,v)
```

rep[v] : unique representative of ConComp(v)

members[rep[v]] : list of the nodes in ConComp(rep[v])

Union-Find Datenstruktur

Idee: verwalte alle Knoten einer ZHK als Liste

Can be used both in Boruvka's and Kruskal's Algo

1 **MST**
Kruskal's Algorithm

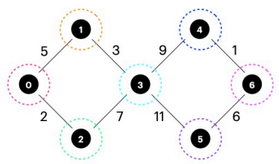
F : ∅

rep[] :

0	1	2	3	4	5	6
0	1	2	3	4	5	6

members[] :

0	{0}
1	{1}
2	{2}
3	{3}
4	{4}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

2 **MST**
Kruskal's Algorithm

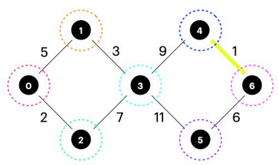
F : ∅

rep[] :

0	1	2	3	4	5	6
0	1	2	3	4	5	6

members[] :

0	{0}
1	{1}
2	{2}
3	{3}
4	{4}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

3 **MST**
Kruskal's Algorithm

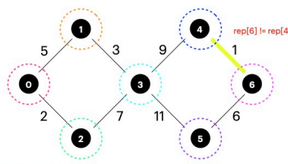
F : ∅

rep[] :

0	1	2	3	4	5	6
0	1	2	3	4	5	6

members[] :

0	{0}
1	{1}
2	{2}
3	{3}
4	{4}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

4 **MST**
Kruskal's Algorithm

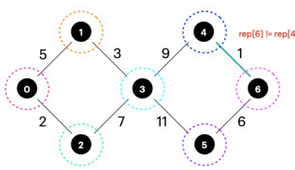
F : { {6,4} }

rep[] :

0	1	2	3	4	5	6
0	1	2	3	4	5	6

members[] :

0	{0}
1	{1}
2	{2}
3	{3}
4	{4}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

5 **MST**
Kruskal's Algorithm

F : { {6,4} }

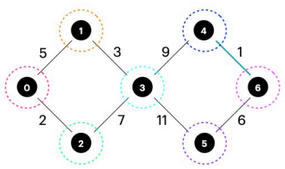
UNION(6,4)

rep[] :

0	1	2	3	4	5	6
0	1	2	3	4	5	6

members[] :

0	{0}
1	{1}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

6 **MST**
Kruskal's Algorithm

F : { {6,4} }

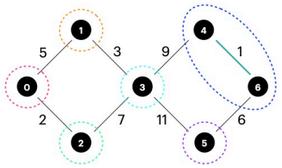
UNION(6,4)

rep[] :

0	1	2	3	4	5	6
0	1	2	3	4	5	4

members[] :

0	{0}
1	{1}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

7 **MST**
Kruskal's Algorithm

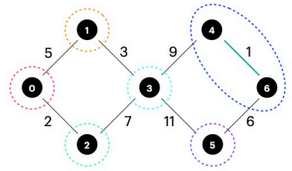
F : { {6,4} }

rep[] :

0	1	2	3	4	5	6
0	1	2	3	4	5	4

members[] :

0	{0}
1	{1}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

8 **MST**
Kruskal's Algorithm

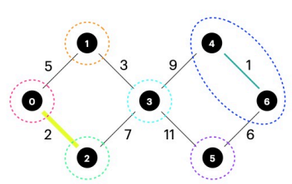
F : { {6,4} }

rep[] :

0	1	2	3	4	5	6
0	1	2	3	4	5	4

members[] :

0	{0}
1	{1}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

9 **MST**
Kruskal's Algorithm

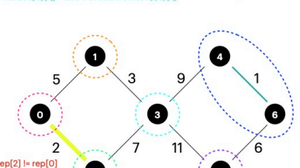
F : { {6,4} }

rep[] :

0	1	2	3	4	5	6
0	1	2	3	4	5	4

members[] :

0	{0}
1	{1}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

10 **MST**
Kruskal's Algorithm

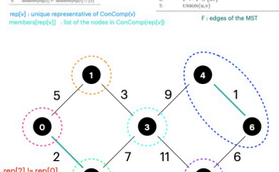
F : { {6,4} , {2,0} }

rep[] :

0	1	2	3	4	5	6
0	1	2	3	4	5	4

members[] :

0	{0}
1	{1}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

11 **MST**
Kruskal's Algorithm

F : { {6,4} , {2,0} }

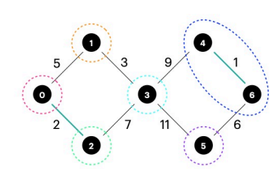
UNION(2,0)

rep[] :

0	1	2	3	4	5	6
0	1	0	3	4	5	4

members[] :

0	{0}
1	{1}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

12 **MST**
Kruskal's Algorithm

F : { {6,4} , {2,0} }

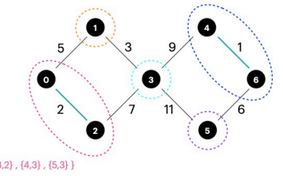
UNION(2,0)

rep[] :

0	1	2	3	4	5	6
0	1	0	3	4	5	4

members[] :

0	{0,2}
1	{1}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

13 **MST**
Kruskal's Algorithm

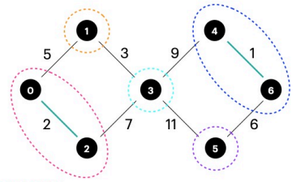
F : { {6,4} , {2,0} }

rep[] :

0	1	2	3	4	5	6
0	1	0	3	4	5	4

members[] :

0	{0,2}
1	{1}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

14 **MST**
Kruskal's Algorithm

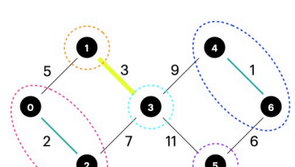
F : { {6,4} , {2,0} }

rep[] :

0	1	2	3	4	5	6
0	1	0	3	4	5	4

members[] :

0	{0,2}
1	{1}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

15 **MST**
Kruskal's Algorithm

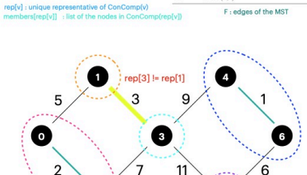
F : { {6,4} , {2,0} }

rep[] :

0	1	2	3	4	5	6
0	1	0	3	4	5	4

members[] :

0	{0,2}
1	{1}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

16 **MST**
Kruskal's Algorithm

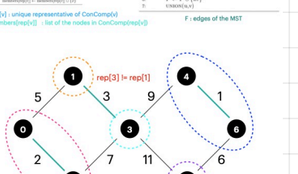
F : { {6,4} , {2,0} , {3,1} }

rep[] :

0	1	2	3	4	5	6
0	1	0	3	4	5	4

members[] :

0	{0,2}
1	{1}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

17 **MST**
Kruskal's Algorithm

F : { {6,4} , {2,0} , {3,1} }

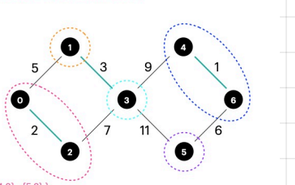
UNION(3,1)

rep[] :

0	1	2	3	4	5	6
0	1	0	3	4	5	4

members[] :

0	{0,2}
1	{1}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

18 **MST**
Kruskal's Algorithm

F : { {6,4} , {2,0} , {3,1} }

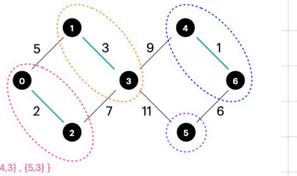
UNION(3,1)

rep[] :

0	1	2	3	4	5	6
0	1	0	1	4	5	4

members[] :

0	{0,2}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

19 **MST**
Kruskal's Algorithm

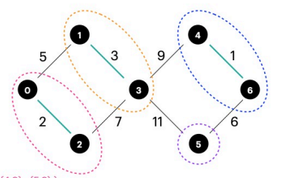
F : { {6,4} , {2,0} , {3,1} }

rep[] :

0	1	2	3	4	5	6
0	1	0	1	4	5	4

members[] :

0	{0,2}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

20 **MST**
Kruskal's Algorithm

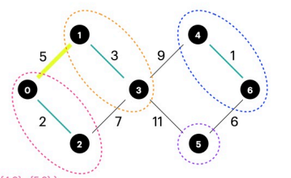
F : { {6,4} , {2,0} , {3,1} }

rep[] :

0	1	2	3	4	5	6
0	1	0	1	4	5	4

members[] :

0	{0,2}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

21 **MST**
Kruskal's Algorithm

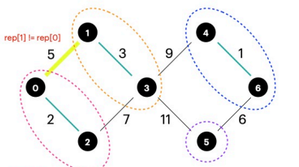
F : { {6,4} , {2,0} , {3,1} }

rep[] :

0	1	2	3	4	5	6
0	1	0	1	4	5	4

members[] :

0	{0,2}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

22 **MST**
Kruskal's Algorithm

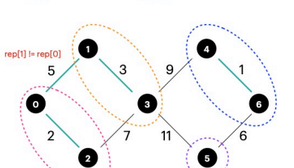
F : { {6,4} , {2,0} , {3,1} , {1,0} }

rep[] :

0	1	2	3	4	5	6
0	1	0	1	4	5	4

members[] :

0	{0,2}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

23 **MST**
Kruskal's Algorithm

F : { {6,4} , {2,0} , {3,1} , {1,0} }

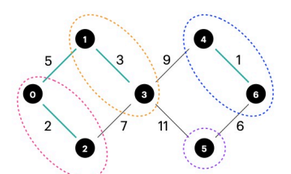
UNION(1,0)

rep[] :

0	1	2	3	4	5	6
0	1	0	1	4	5	4

members[] :

0	{0,2}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

24 **MST**
Kruskal's Algorithm

F : { {6,4} , {2,0} , {3,1} , {1,0} }

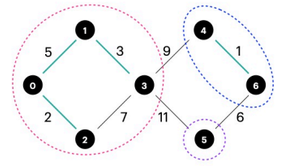
UNION(1,0)

rep[] :

0	1	2	3	4	5	6
0	0	0	0	4	5	4

members[] :

0	{0,2,1,3}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

Algorithm 11 Union-Find(G)
1: **Implementierung:**
2: MAKE(V): $\text{rep}[v] \leftarrow v \ \forall v \in V$
3:
4: SAME(u,v): teste ob $\text{rep}[u] = \text{rep}[v]$
5:
6: UNION(u,v):
7: **for** $x \in \text{members}[\text{rep}[u]]$ **do**
8: $\text{rep}[x] \leftarrow \text{rep}[v]$
9: $\text{members}[\text{rep}[v]] \leftarrow \text{members}[\text{rep}[v]] \cup \{x\}$

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)
1: $F \leftarrow \emptyset$
2: $UF \leftarrow$ MAKE(V)
3: SORT(E)
4: **for** $uv \in E$, aufsteigend sortiert **do**
5: **if** SAME(u,v) = **false then**
6: $F \leftarrow F \cup \{uv\}$
7: UNION(u,v)

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

F : edges of the MST

25

MST
Kruskal's Algorithm

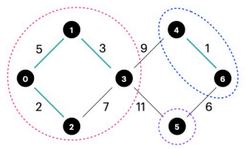
F : { {6,4} , {2,0} , {3,1} , {1,0} }

rep[] :

0	1	2	3	4	5	6
0	0	0	0	4	5	4

members[] :

0	{0,2,1,3}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

26

MST
Kruskal's Algorithm

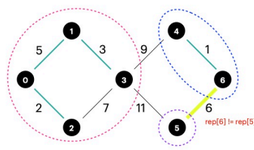
F : { {6,4} , {2,0} , {3,1} , {1,0} }

rep[] :

0	1	2	3	4	5	6
0	0	0	0	4	5	4

members[] :

0	{0,2,1,3}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

27

MST
Kruskal's Algorithm

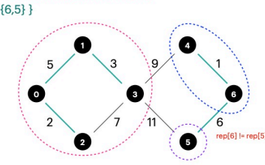
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

rep[] :

0	1	2	3	4	5	6
0	0	0	0	4	5	4

members[] :

0	{0,2,1,3}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

28

MST
Kruskal's Algorithm

F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

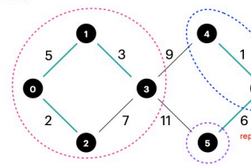
UNION(6,5)

rep[] :

0	1	2	3	4	5	6
0	0	0	0	4	5	4

members[] :

0	{0,2,1,3}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5,4,6}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

29

MST
Kruskal's Algorithm

F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

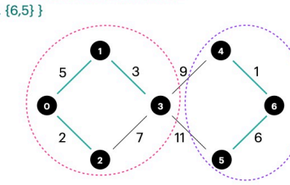
UNION(6,5)

rep[] :

0	1	2	3	4	5	6
0	0	0	0	5	5	5

members[] :

0	{0,2,1,3}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5,4,6}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

30

MST
Kruskal's Algorithm

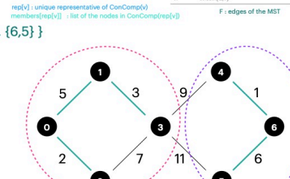
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

rep[] :

0	1	2	3	4	5	6
0	0	0	0	5	5	5

members[] :

0	{0,2,1,3}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5,4,6}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

31

MST
Kruskal's Algorithm

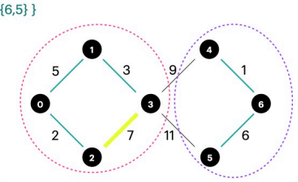
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

rep[] :

0	1	2	3	4	5	6
0	0	0	0	5	5	5

members[] :

0	{0,2,1,3}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5,4,6}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

32

MST
Kruskal's Algorithm

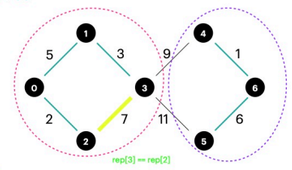
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

rep[] :

0	1	2	3	4	5	6
0	0	0	0	5	5	5

members[] :

0	{0,2,1,3}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5,4,6}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

33

MST
Kruskal's Algorithm

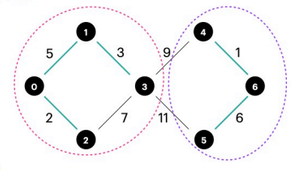
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

rep[] :

0	1	2	3	4	5	6
0	0	0	0	5	5	5

members[] :

0	{0,2,1,3}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5,4,6}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

34

MST
Kruskal's Algorithm

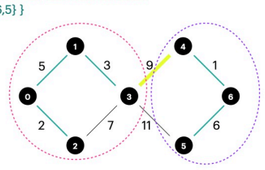
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

rep[] :

0	1	2	3	4	5	6
0	0	0	0	5	5	5

members[] :

0	{0,2,1,3}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5,4,6}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

35

MST
Kruskal's Algorithm

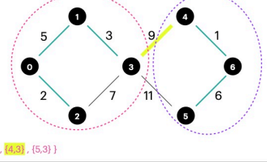
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

rep[] :

0	1	2	3	4	5	6
0	0	0	0	5	5	5

members[] :

0	{0,2,1,3}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5,4,6}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

36

MST
Kruskal's Algorithm

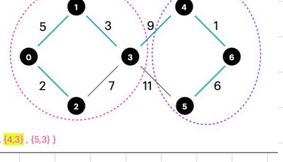
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {4,3} }

rep[] :

0	1	2	3	4	5	6
0	0	0	0	5	5	5

members[] :

0	{0,2,1,3}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5,4,6}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

37

MST
Kruskal's Algorithm

F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {4,3} }

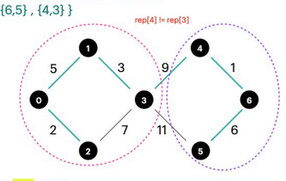
UNION(4,3)

rep[] :

0	1	2	3	4	5	6
0	0	0	0	5	5	5

members[] :

0	{0,2,1,3}
1	{1,3}
2	{2}
3	{3}
4	{4,6,5}
5	{5,4,6}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

38

MST
Kruskal's Algorithm

F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {4,3} }

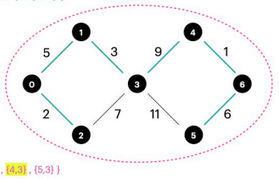
UNION(4,3)

rep[] :

0	1	2	3	4	5	6
0	0	0	0	0	0	0

members[] :

0	{0,2,1,3,5,4,6}
1	{1,3}
2	{2}
3	{3}
4	{4,6,5}
5	{5,4,6}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

39

MST
Kruskal's Algorithm

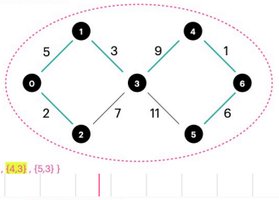
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {4,3} }

rep[] :

0	1	2	3	4	5	6
0	0	0	0	0	0	0

members[] :

0	{0,2,1,3,5,4,6}
1	{1,3}
2	{2}
3	{3}
4	{4,6,5}
5	{5,4,6}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

40

MST
Kruskal's Algorithm

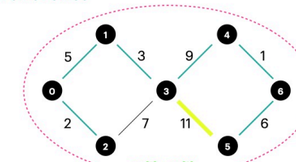
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {4,3} }

rep[] :

0	1	2	3	4	5	6
0	0	0	0	0	0	0

members[] :

0	{0,2,1,3,5,4,6}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5,4,6}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

41

MST
Kruskal's Algorithm

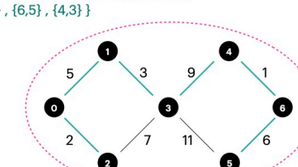
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {4,3} }

rep[] :

0	1	2	3	4	5	6
0	0	0	0	0	0	0

members[] :

0	{0,2,1,3,5,4,6}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5,4,6}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} } ✅ All edges are done

42

MST
Kruskal's Algorithm

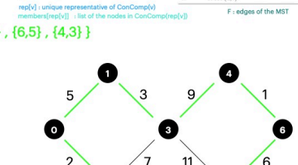
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {4,3} }

rep[] :

0	1	2	3	4	5	6
0	0	0	0	0	0	0

members[] :

0	{0,2,1,3,5,4,6}
1	{1,3}
2	{2}
3	{3}
4	{4,6}
5	{5,4,6}
6	{6}



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} } ✅ All edges are done

Shortest Paths

one-to-all

- ↳ BFS usage
- ↳ Dijkstra
- ↳ Bellman-Ford
  - ↳ negative-closed w. detection

all-to-all

- ↳ one-to-all usage
- ↳ Floyd-Warshall
- ↳ Johnson
- ↳ #walks using A_G

↳ overview

Overview

one-to-all

all-to-all

Kürzeste Wege, one-to-all (single source), Wiederholung

$G = (V, E, c)$ Graph mit Kostenfunktion $c: E \to \mathbb{R}$, $|V| = n$, $|E| = m$

gerichtet oder ungerichtet ★

allgemeiner ↓

Kosten	Algorithmus	Laufzeit	
alle Kosten 1	Breitensuche	$O(m+n)$	
$c(e) \geq 0$	Dijkstra	$O((m+n) \cdot \log n)$	nicht besprochen, Verbesserung mit Fibonacci-Heaps
		oder $O(m + n \log n)$	
$c(e) \in \mathbb{R}$ auch negativ	Bellman-Ford	$O(n\,m)$	

- One-to-one ist nicht schneller als one-to-all
- Falls G keine Zyklen hat, kann man $O(m+n)$ für beliebige Kosten erreichen: topologische Sortierung + DP

All Pairs Shortest Path ← Name etwas irreführend: wir suchen Wege, nicht Pfade

Finde für alle Paare $u, v \in V$ einen kürzesten u-v-Weg

allgemeiner ↓

Kosten	Algorithmus	Laufzeit	
alle Kosten 1	$n \times$ Breitensuche	$O(nm + n^2)$	
$c(e) \geq 0$	$n \times$ Dijkstra	$O(n \cdot (m+n) \cdot \log n)$	Fibonacci-Heaps
		oder $O(nm + n^2 \log n)$	
$c(e) \in \mathbb{R}$ auch negativ	$n \times$ ~~Bellman-Ford~~	$O(n^2 m)$	
	Floyd-Warshall	$O(n^3)$	
	Johnson	$O(n \cdot (m+n) \cdot \log n)$	selbe Laufzeit wie $n \times$ Dijkstra
		oder $O(nm + n^2 \log n)$	Fibonacci-Heaps

falls es keine negativen Zyklen gibt

one-to-all

G (directed/undirected)	Algorithm	Runtime
unweighted , all edges with the same positive weight	BFS usage	O (|V| + |E|)
weighted , nonnegative edge weights $c(e) \geq 0$	Dijsktra	O ((|V| + |E|) * log n)
weighted, positive and (possibly) negative edge weights $c(e) \in \mathbb{R}$	Belmann-Ford*	O (|V| * |E|)
G has no cycles	topological sorting + DP	O (|V| + |E|)

all-to-all

G (directed/undirected)	Algorithm	Runtime
unweighted , all edges with the same positive weight	n x BFS	O (|V| * (|V| + |E|))
weighted , nonnegative edge weights $c(e) \geq 0$	n x Dijsktra	O (|V| * (|V| + |E|) * log(|V|))
weighted, positive and (possibly) negative edge weights $c(e) \in \mathbb{R}$	n x Belmann-Ford	O (|V| * |V| * |E|)
	Floyd - Warshall*	O($\|V\|^3$)
weighted, positive and (possibly) negative edge weights $c(e) \in \mathbb{R}$, no negative cycles	Johnson	O (|V| * (|V| + |E|) * log n)

*negative closed walk detection

One-to-all

Problemstellung

directed G , start-vertex $s \in V$ $n = |V|$ $m = |E|$

Edge-Costs: $c(e) \in \mathbb{R}$ $u \xrightarrow{e} v$

Walk-Costs: $c(W) := c(v_0, v_1) + \ldots + c(v_{\ell-1}, v_\ell)$ $u \overset{w}{\rightsquigarrow} v$ (walk from u to v)

distance: $d(u,v) := \min \{ c(W) \mid u \overset{w}{\rightsquigarrow} v \}$

BFS

HS21

Algorithm 14 Breadth-first search

```
Q ← new queue ()
Q.PUSH (r)
D ← {r}                                   dist[r] = 0
                                          ▷ Stores nodes that are done (in Q or already processed)
while ¬Q.ISEMPTY () do
    v ← Q.POP ()
    /*do something with v*/
    for w s.t. v and w are adjacent in G do
        if w ∉ D then
            Q.PUSH (w)
            D ← D ∪ {w}
                                          dist[w] = dist[v] + 1
```

Proofs

- ∄ negative closed walk
  ⇒ every walk can be shortened to paths
  ⇒ $d(s,s) = 0$ (cannot be negative)
- $d(u,v) \leq d(u,w) + d(w,v)$
- $\forall v \notin s$ $d(s,v) = \min_{u \to v} d(s,u) + c(u,v)$

Shortest Paths

BFS usage - with distances

Runtime : O (|V| + |E|)

Algorithm 5 BFS(s)

```
1: Q ← {s}                                   Q is a FIFO queue
2: enter[s] ← 0;   T ← 1                     distance[s] = 0 ;
3: while Q ≠ ∅ do
4:     u ← dequeue(Q)
5:     leave[u] ← T;   T ← T + 1
6:     for (u, v) ∈ E, enter[v] nicht zugewiesen do
7:         enqueue(Q, v)
8:         enter[v] ← T;   T ← T + 1         distance[v] <- distance[u] + 1 ;
```

BFS with Distance

```
BFS(s)
    Mark all vertices as unvisited; for each v set dist(v) = ∞
    Initialize search tree T to be empty
    Mark vertex s as visited and set dist(s) = 0
    set Q to be the empty queue
    enq(s)
    while Q is nonempty do
        u = deq(Q)
        for each vertex v ∈ Adj(u) do
            if v is not visited do
                add edge (u, v) to T
                Mark v as visited, enq(v)
                and set dist(v) = dist(u) + 1
```

BFS with Layers

```
BFSLayers(s):
    Mark all vertices as unvisited and initialize T to be empty
    Mark s as visited and set L_0 = {s}
    i = 0
    while L_i is not empty do
        initialize L_{i+1} to be an empty list
        for each u in L_i do
            for each edge (u, v) ∈ Adj(u) do
            if v is not visited
                mark v as visited
                add (u, v) to tree T
                add v to L_{i+1}
        i = i + 1
```

Dijkstra (s) : no negative weights

Input: $s \in V$

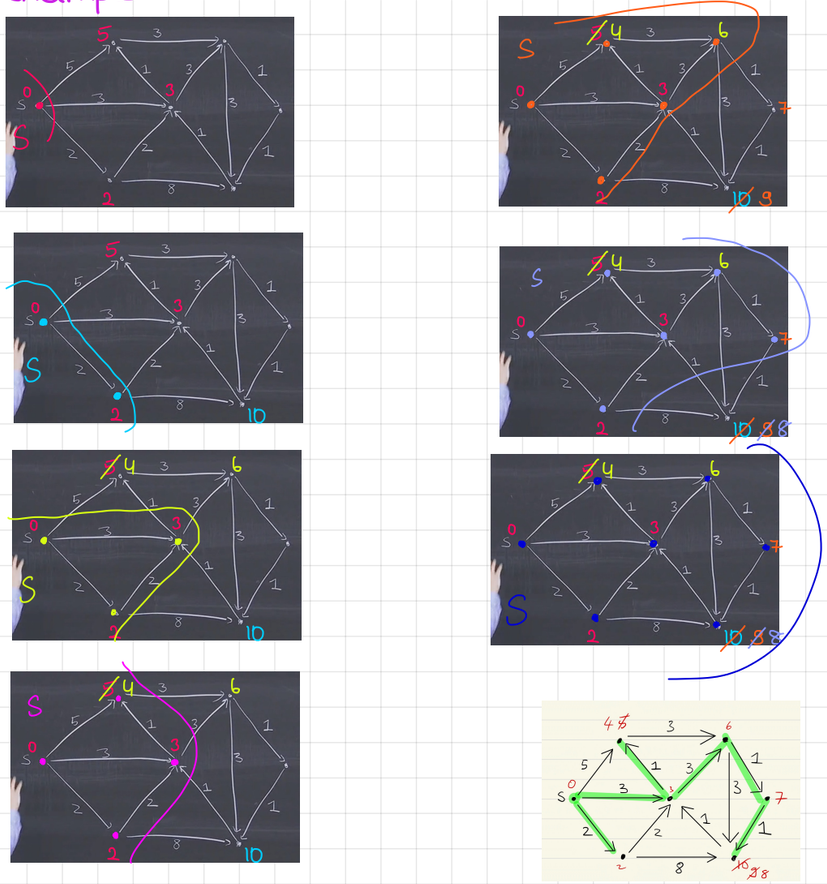
Output: dist(s,v) for all $v \in V$

Runtime: $O((m+n) \cdot \log(n))$

Algorithm 6 Dijkstra(s)

```
1: d[s] ← 0;   d[v] ← ∞  ∀v ∈ V \ {s}
2: S ← ∅
3: H ← make-heap(V); decrease-key(H, s, 0)
4: while S ≠ V do
5:     v* ← extract-min(H)
6:     S ← S ∪ {v*}
7:     for (v*, v) ∈ E,  v ∉ S do
8:         d[v] ← min{d[v], d[v*] + c(v*, v)}
9:         decrease-key(H, v, d[v])
```

example:



Cost's are equal: ≤ 2 (endliche) Schrankenwerte: $d[v^*]$, $d[v^*]+1$

↝ find v^* in $O(1)$ with FIFO Queue (BFS)

for general costs (not negative):

differences with Prim:

Algorithm 10 Prim(G, s) (mit min-heap)

```
1: H ← make-heap(V, ∞), S ← ∅
2: d[s] ← 0;   d[v] ← ∞  ∀v ∈ V \ {s}
3: decrease-key(H, s, 0)
4: while H ≠ ∅ do
5:     v* ← extract-min(H)
6:     S ← S ∪ {v*}
7:     for v*v ∈ E,  v ∉ S do
8:         d[v] ← min{d[v], c(v*, v)}        ▷ Unterschied zu Dijkstra
9:         decrease-key(H, v, d[v])
```

Runtime Analysis:

Laufzeit Dijkstra

extract-min $\leq n$ # decrease-key $\leq m$

total: $O(n + (\#\text{extract-min} + \#\text{decrease-key}) \cdot \log n)$

$= O((n+m) \cdot \log(n))$

Shortest Paths

Dijkstra's Algorithm

Algorithm 6 Dijkstra(s)

```
1: d[s] ← 0;   d[v] ← ∞  ∀v ∈ V \ {s}
2: S ← ∅
3: H ← make-heap(V); decrease-key(H, s, 0)
4: while S ≠ V do
5:     v* ← extract-min(H)
6:     S ← S ∪ {v*}
7:     for (v*, v) ∈ E,  v ∉ S do
8:         d[v] ← min{d[v], d[v*] + c(v*, v)}
9:         decrease-key(H, v, d[v])
```

Runtime : O ((|V| + |E|) * log n)

weighted , positive edge weights
c(e) ≥ 0

d[] : distance array S : visited set

H : min-heap

make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

Shortest Paths
Dijkstra's Algorithm

25 Shortest Paths — Dijkstra's Algorithm

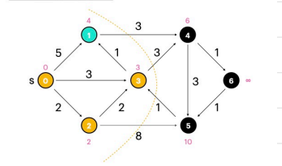
S : { 0 , 2 , 3 , 1 }

v* = 1

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	10	∞

"Heap" :

4	5	6
6	10	∞



make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

26 Shortest Paths — Dijkstra's Algorithm

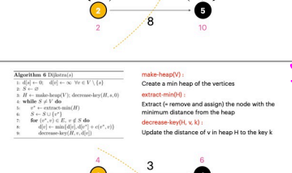
S : { 0 , 2 , 3 , 1 }

v* = 1

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	10	∞

"Heap" :

4	5	6
6	10	∞



make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

27 Shortest Paths — Dijkstra's Algorithm

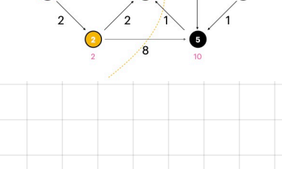
S : { 0 , 2 , 3 , 1 }

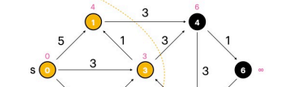
v* = 1

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	10	∞

"Heap" :

4	5	6
6	10	∞





make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

28 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 }

v* = 1

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	10	∞

"Heap" :

4	5	6
6	10	∞

make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

29 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 }

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	10	∞

"Heap" :

4	5	6
6	10	∞

make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

30 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 }

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	10	∞

"Heap" :

4	5	6
6	10	∞

make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

31 Shortest Paths — Dijkstra's Algorithm

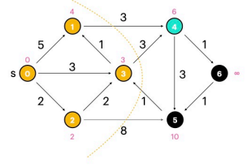
S : { 0 , 2 , 3 , 1 }

v* = 4

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	10	∞

"Heap" :

5	6
10	∞



make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

32 Shortest Paths — Dijkstra's Algorithm

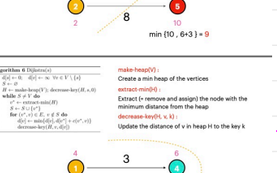
S : { 0 , 2 , 3 , 1 , 4 }

v* = 4

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	10	∞

"Heap" :

5	6
10	∞



make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

33 Shortest Paths — Dijkstra's Algorithm

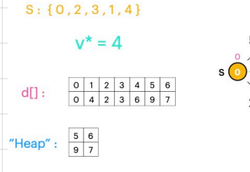
S : { 0 , 2 , 3 , 1 , 4 }

v* = 4

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	9	7

"Heap" :

5	6
9	7



make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

34 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 , 4 }

v* = 4

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	9	7

"Heap" :

5	6
9	7

make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

35 Shortest Paths — Dijkstra's Algorithm

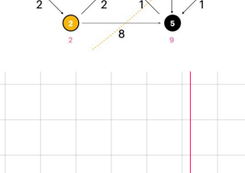
S : { 0 , 2 , 3 , 1 , 4 }

v* = 4

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	9	∞

"Heap" :

5	6
9	∞



make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

36 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 , 4 }

v* = 4

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	9	7

"Heap" :

5	6
9	7

make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

37 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 , 4 }

v* = 4

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	9	7

"Heap" :

5	6
9	7

make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

38 Shortest Paths — Dijkstra's Algorithm

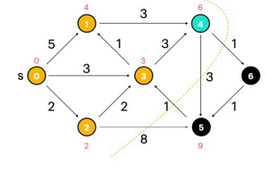
S : { 0 , 2 , 3 , 1 , 4 , 6 }

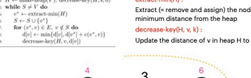
v* = 6

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	9	7

"Heap" :

5
9





make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

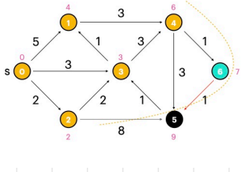
39 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 , 4 }

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	9	7

"Heap" :

5	6
9	7



make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

40 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 , 4 }

v* = 6

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	9	7

"Heap" :

5
9

make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

41 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 , 4 , 6 }

v* = 6

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	9	7

"Heap" :

5
9

make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

42 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 , 4 , 6 }

v* = 6

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	9	7

"Heap" :

5
9

make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

43 Shortest Paths — Dijkstra's Algorithm

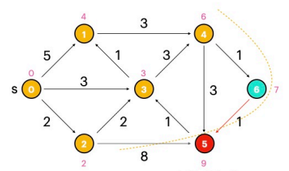
S : { 0 , 2 , 3 , 1 , 4 , 6 }

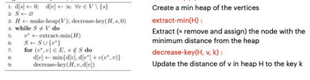
v* = 6

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	8	7

"Heap" :

5
8





make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

min {9 , 7+1 } = 8

44 Shortest Paths — Dijkstra's Algorithm

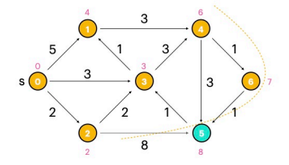
S : { 0 , 2 , 3 , 1 , 4 , 6 }

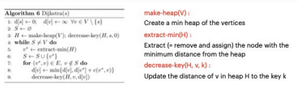
v* = 6

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	8	7

"Heap" :

5
8





make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

45 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 , 4 , 6 }

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	8	7

"Heap" :

5
8

make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

46 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 , 4 , 6 }

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	8	7

"Heap" :

5
8

make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

47 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 , 4 , 6 }

v* = 5

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	8	7

"Heap" :

make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

48 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 , 4 , 6 , 5 }

v* = 5

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	8	7

"Heap" :

make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

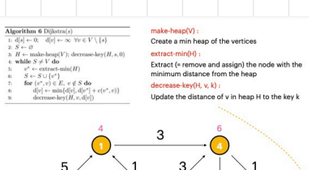
decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

49 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 , 4 , 6 , 5 }

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	8	7

"Heap" :



make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

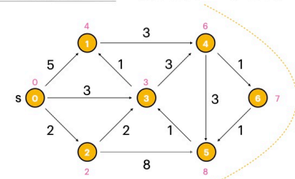
50 Shortest Paths — Dijkstra's Algorithm

S : { 0 , 2 , 3 , 1 , 4 , 6 , 5 }

S = V

d[] :	0	1	2	3	4	5	6
	0	4	2	3	6	8	7

"Heap" :



make-heap(V) :
Create a min heap of the vertices

extract-min(H) :
Extract (= remove and assign) the node with the minimum distance from the heap

decrease-key(H, v, k) :
Update the distance of v in heap H to the key k

Bellman-Ford (s) : negative edges + negative cycles are allowed

Input: $s \in V$

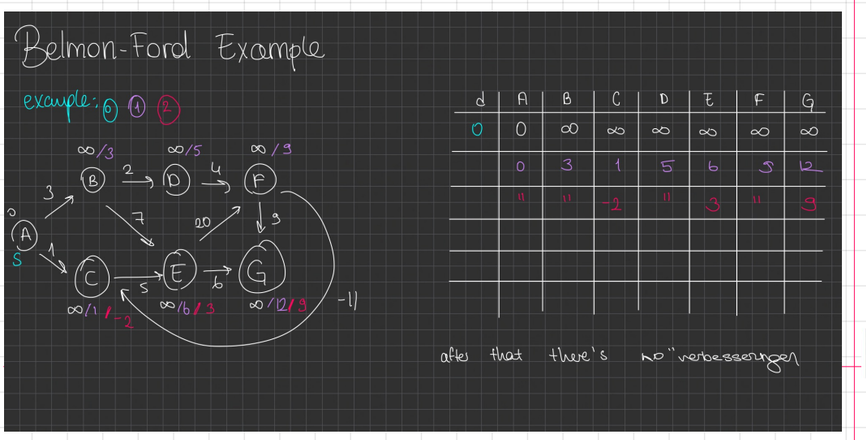
Output: dist(s,v) for all $v \in V$

Runtime: $O(m \cdot n)$

Algorithm 7 Bellman-Ford(s)

```
1: d[s] ← 0;   d[v] ← ∞ ∀v ∈ V \ {s}        ▷ 0-gute Schranken
2: for i ∈ {1, ... n − 1} do                ▷ Verbessere Schranken (n − 1)-mal
3:     for (u, v) ∈ E do
4:         d[v] ← min{d[v], d[u] + c(u, v)}
```

n-1 iterations ∄path with n-1 length



d	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
	0	3	1	5	6	5	12
	0	3	1	5	6	5	9

Idea:

Find the shortest paths from starting node to every other node with <= *i* edges

Recurrence:

d[u] with i+1 edges =

min(min {d[v] + c(v,u) | d[v] is the shortest dist. with <= i edges}, d[u])

When to stop:

The shortest path cannot have more the n – 1 edges. Why?

> After k iterations, if there is a path from s to u with at most k edges, then `distance(u)` is at most the length of the shortest path from s to u with at most k edges.

```
distance := list of size n
predecessor := list of size n

// Step 1: initialize graph
for each vertex v in vertices do
    // Initialize the distance to all vertices to infinity
    distance[v] := inf
    // And having a null predecessor
    predecessor[v] := null

// The distance from the source to itself is, of course, zero
distance[source] := 0

// Step 2: relax edges repeatedly
repeat |V|-1 times:
    for each edge (u, v) with weight w in edges do
        if distance[u] + w < distance[v] then
            distance[v] := distance[u] + w
            predecessor[v] := u
```

Shortest Paths

Bellman Ford

weighted, positive and (possibly) negative edge weights ,
(possibly) negative closed walks
c(e) ∈ ℝ

Runtime : O (|V| *|E|)

Algorithm 7 Bellman-Ford(s)

```
1: d[s] ← 0;   d[v] ← ∞ ∀v ∈ V \ {s}
2: for i ∈ {1, ... n − 1} do        relax the edges for n-1 times
3:     for (u, v) ∈ E do
4:         d[v] ← min{d[v], d[u] + c(u, v)}
```

Why n-1 iterations ?

A shortest path in a graph without cycles will have at most n−1 edges

(since a path cannot visit any vertex more than once in a simple graph)

How to detect negative closed walks :

Do one extra iteration

If any distance improves, it indicates the existence of a directed closed walk with negative total weight

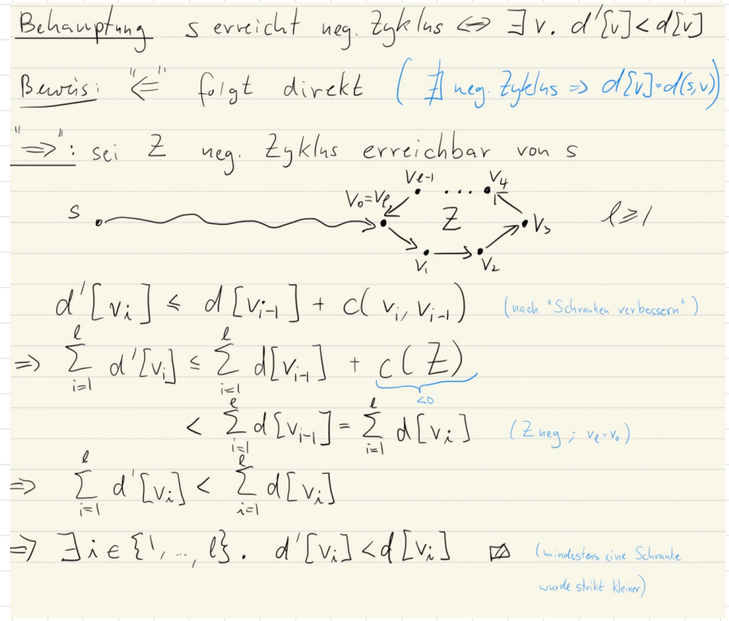
Runtime Analysis:

pro "Schranken Verbessern" : $O(m)$

gesamt: $O(n \cdot m)$

Negative Closed Walks Detection:

↳ after (n-1)-th iteration do one more iteration, if the values change, ∃ a directed closed walk with negative total weight



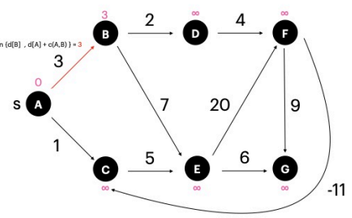
1

Shortest Paths
Bellman Ford

Algorithm 7 Bellman-Ford(s)
1: $d[s] \leftarrow 0$; $d[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \dots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3					
2							
3							
4							
5							
6							



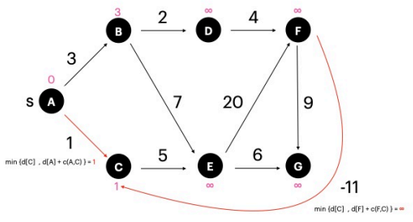
2

Shortest Paths
Bellman Ford

Algorithm 7 Bellman-Ford(s)
1: $d[s] \leftarrow 0$; $d[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \dots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1				
2							
3							
4							
5							
6							



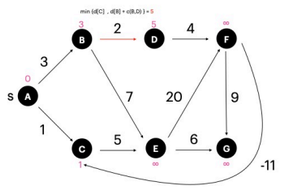
3

Shortest Paths
Bellman Ford

Algorithm 7 Bellman-Ford(s)
1: $d[s] \leftarrow 0$; $d[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \dots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5			
2							
3							
4							
5							
6							



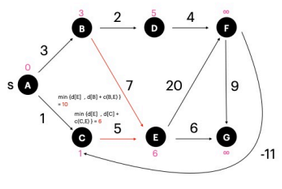
4

Shortest Paths
Bellman Ford

Algorithm 7 Bellman-Ford(s)
1: $d[s] \leftarrow 0$; $d[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \dots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5	6		
2							
3							
4							
5							
6							



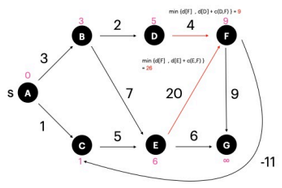
5

Shortest Paths
Bellman Ford

Algorithm 7 Bellman-Ford(s)
1: $d[s] \leftarrow 0$; $d[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \dots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5	6	9	
2							
3							
4							
5							
6							



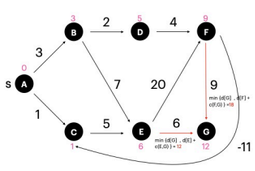
6

Shortest Paths
Bellman Ford

Algorithm 7 Bellman-Ford(s)
1: $d[s] \leftarrow 0$; $d[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \dots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5	6	9	12
2							
3							
4							
5							
6							



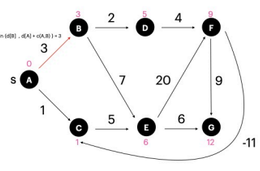
7

Shortest Paths
Bellman Ford

Algorithm 7 Bellman-Ford(s)
1: $d[s] \leftarrow 0$; $d[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \dots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5	6	9	12
2	0	3	1	5	6	9	12
3							
4							
5							
6							



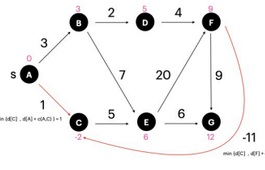
8

Shortest Paths
Bellman Ford

Algorithm 7 Bellman-Ford(s)
1: $d[s] \leftarrow 0$; $d[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \dots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5	6	9	12
2	0	3	-2	5	6	9	12
3							
4							
5							
6							



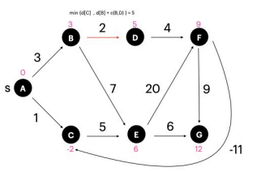
9

Shortest Paths
Bellman Ford

Algorithm 7 Bellman-Ford(s)
1: $d[s] \leftarrow 0$; $d[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \dots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5	6	9	12
2	0	3	-2	5	6	9	12
3							
4							
5							
6							



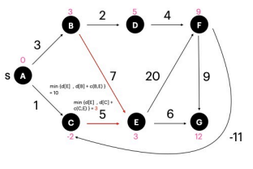
10

Shortest Paths
Bellman Ford

Algorithm 7 Bellman-Ford(s)
1: $d[s] \leftarrow 0$; $d[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \dots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5	6	9	12
2	0	3	-2	5	3	9	12
3							
4							
5							
6							



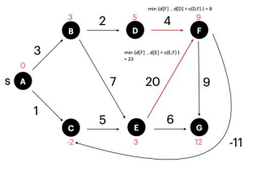
11

Shortest Paths
Bellman Ford

Algorithm 7 Bellman-Ford(s)
1: $d[s] \leftarrow 0$; $d[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \dots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5	6	9	12
2	0	3	-2	5	3	9	12
3							
4							
5							
6							



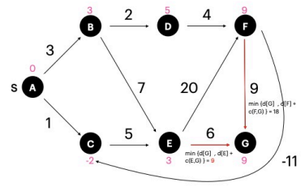
12

Shortest Paths
Bellman Ford

Algorithm 7 Bellman-Ford(s)
1: $d[s] \leftarrow 0$; $d[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \dots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5	6	9	12
2	0	3	-2	5	3	9	9
3							
4							
5							
6							



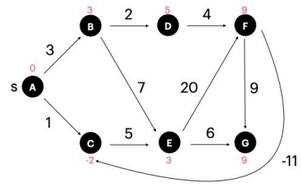
13

Shortest Paths
Bellman Ford

Algorithm 7 Bellman-Ford(s)
1: $d[s] \leftarrow 0$; $d[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \dots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5	6	9	12
2	0	3	-2	5	3	9	9
3							
4							
5							
6							



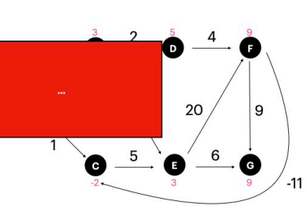
14

Shortest Paths
Bellman Ford

Algorithm 7 Bellman-Ford(s)
1: $d[s] \leftarrow 0$; $d[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \dots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5	6	9	
2	0	3	-2	5	3	9	
3							
4							
5							
6							



Theory Task T4. / 13 P

/ 3 P a) It is 2050 and the city of Zurich has finally built some proper cycling lanes to travel around the city. It turns out that one of the cycling lanes starts exactly at the road crossing where you are living, and of course there are also cycling lanes at the ETH road crossing. Since you are commuting to ETH by e-bike every day and you care about the environment, you would like to know which way from your place to ETH requires the least power from your e-bike. Your commuting way should only go through cycling lanes, and not use any other streets.

There are several cycling lanes around the city and you can switch between them at crossings. For each pair of crossings that are connected directly by a cycling lane, you know how much battery is needed to travel from the first crossing to the second one. Note that your e-bike battery gets charged when you go downhill, so there are some crossing pairs for which you gain some battery by going from the first one to the second one. However, due to the laws of physics, it is impossible that you leave a crossing, bike around cycling lanes and come back to the same crossing with more battery than you started with. Biking from a crossing C to another crossing C' does not necessarily require the same amount of power than biking from C' to C (for example, think of a steep slope from C to C').

i) Model the problem as a graph problem. Describe the set of vertices, the set of edges and the weights in words. What is the corresponding graph problem ?

Solution: The network of cycling lanes defines a directed graph. The vertices V are the crossings, and for each pair of crossings $u, v \in V$ the two directed edges (u, v) and (v, u) are present if u and v are connected directly by a cycling lane. The weight $w((u, v))$ of an edge (u, v) is the amount of power needed to go from u to v using this direct connection. If you gain battery by going from u to v then $w((u, v))$ will be negative. Note that the assumption about the laws of physics guarantees that the graph does not contain a negative cycle.

The graph problem corresponding to finding the most energy-economic way from your place (denoted by vertex s) to ETH (denoted by vertex t) is the computation of a shortest path from s to t.

ii) Which algorithm from the lecture can you use to solve the graph problem ? Justify why you can use this algorithm, and state its running time in terms of $|V|$ and $|E|$ in Θ-notation.

Solution: Since the graph contains edges with negative weights, but no negative cycle, we can use the Bellman-Ford algorithm to compute the shortest path from s to t, which has a running time of $\Theta(|V| \cdot |E|)$.

/ 4 P b) A friend of yours now tells you that no matter which pair of crossings you are considering, the most energy-economic way (i.e., the one that requires the least battery) from the first one to the second one always goes through at most k other crossings (for some natural number $k \leq \sqrt{|V|}$). How can you modify the previous algorithm to get a lower runtime? *In order to achieve full points, your algorithm should run in time* $O(k \cdot |E|)$.

i) Describe your algorithm (using text or pseudocode). A high-level description is enough.

Solution: In this case, we know that any shortest path in the graph consists of at most $k+1$ edges. The algorithm therefore consists of running the "bound improvement" update of the Bellman-Ford algorithm $k+1$ times (instead of the usual $|V|-1$ times), with source vertex s. This is captured by the following pseudocode, where the distances from s to other vertices are saved in the array d:

Algorithm 3 `TruncatedBellmanFord`(V, E, w, s, k)

```
d[s] ← 0
for v ∈ V \ {s} do
    d[v] ← ∞
for ℓ = 1, ..., k + 1 do
    for (u, v) ∈ E do
        if d[v] > d[u] + w((u, v)) then
            d[v] ← d[u] + w((u, v))
return d[t]
```

ii) Prove the correctness of your algorithm and show that it runs in time $O(k \cdot |E|)$.

Solution: The correctness of the algorithm basically follows from the correctness of the Bellman-Ford algorithm and the guarantee that all shortest paths have at most $k + 1$ edges. Indeed for any $\ell \in \mathbb{N}$, we know that after ℓ "bound improvement" updates of the Bellman-Ford algorithm, the values stored in the array $(d[v])_{v \in V}$ contain the length of the shortest paths from s to v among all paths of at most ℓ edges. Therefore, after $k+1$ updates, these will be the length of the shortest paths from s to all other vertices in V.

Since one "bound improvement" update runs in time $O(|E|)$, and we perform it $k + 1$ times, the total running time is indeed $O(k \cdot |E|)$.

/ 6 P c) On the weekend you would like to explore the city with your e-bike. You start from the road crossing where you are living with your battery charged with an amount b of power, and you want to know what are the crossings that you can reach. In other words, you are interested in finding all places that can be visited without needing to charge your battery on the way. Describe an algorithm that gives you the set of such reachable places. *In order to achieve full points, your algorithm should run in time* $O(|V| \cdot |E|)$.

Note that this part builds on part (a), not on part (b). In other words, here we do not assume that the most energy-economic way from any crossing to any other crossing always goes through at most k other crossings.

i) Describe your algorithm (using text or pseudocode). A high-level description is enough.

Solution: The algorithm is a modified version of the Bellman-Ford algorithm. In order to make sure that we never run out of battery on the computed paths, we make an update $d[v] \leftarrow d[u] + w((u, v))$ only if the condition $d[u] \leq b$ is satisfied. After filling the array d in this way, the set of reachable vertices is simply given by $\{v \in V : d[v] \leq b\}$.

Algorithm 4 `BatteryBellmanFord`(V, E, w, s, b)

```
d[s] ← 0
for v ∈ V \ {s} do
    d[v] ← ∞
for ℓ = 1, ..., |V| do
    for (u, v) ∈ E do
        if d[v] > d[u] + w((u, v)) and d[u] ≤ b then
            d[v] ← d[u] + w((u, v))
return {v ∈ V : d[v] ≤ b}
```

ii) Prove the correctness of your algorithm and show that it runs in time $O(|V| \cdot |E|)$.

Solution: Adding the condition $d[u] \leq b$ in the update rule means that now, after the ℓ-th iteration of the for-loop, $d[v]$ contains the length of the shortest path from s to v among paths on at most ℓ edges such that any subpath from s to an intermediate node $v' \neq v$ has length at most b. Therefore after $|V|$ iterations, $d[v]$ contains the length of the shortest path from s to v such that you never run out of battery on the way. Clearly, the vertices we can reach are then those $v \in V$ with $d[v] \leq b$.

Again, each "bound improvement" update runs in time $O(|E|)$, so the total running time is $O(|V| \cdot |E|)$.

All-to-all

same problemstellung, but now there's no source

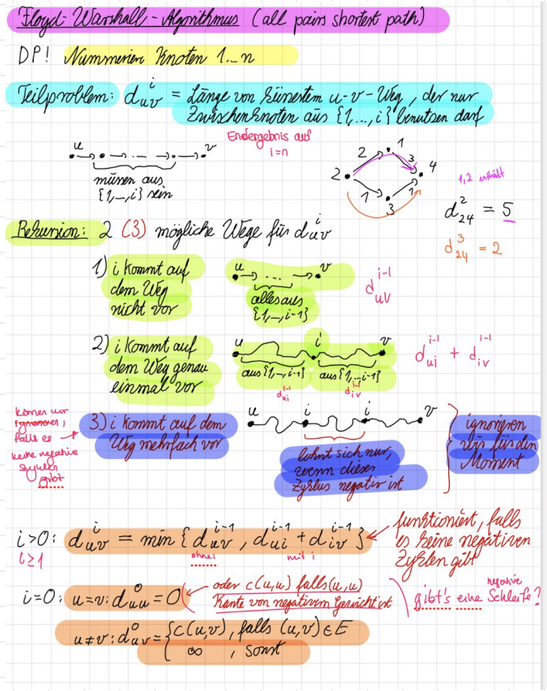
We're calculating the shortest path from every vertex to every vertex

Runtime:

$O(n^3)$

Floyd-Warshall

Idee:



Es gibt negativen Zyklus $\iff$ Es gibt v mit $d_{vv}^n < 0$

Algo:

```
Floyd-Warshall(G)        // Annahme: V = {1,...,n}
// Initialisierung
for u∈V: d⁰_uu ← 0   ← falls keine negativen Schleifen
for u∈V, v∈V\{u}: if (u,v)∈E then d⁰_uv ← c(u,v); else d⁰_uv ← ∞
// DP
for i = 1...n
    for u = 1...n
        for v = 1...n
            d^i_uv ← min{ d^(i-1)_uv , d^(i-1)_ui + d^(i-1)_iv }
return d^n   ↖ n×n-Matrize mit Resultaten
```

```
for u∈V   D[0][u][u] = 0
for u∈V, v∈V\{u}
    if (u,v)∈E  D[0][u][v] = c(u,v)
    else D[0][u][v] = ∞

for i = 1...n
  for u = 1 ... n
    for v = 1 ... n

      D[i][u][v] = min{ D[i-1][u][v] , D[i-1][u][i]
                                       D[i-1][i][v] }
return D[n][ ][ ]  ...
```

Zusammenfassung Floyd-Warshall:

1) Lasse FloydWarshall laufen
2) Falls es ein v gibt mit $d_{vv}^n < 0$ → negative Zyklen, Ausgabe falsch
3) sonst: keine negativen Zyklen, Ausgabe korrekt

Allgemeine Antwort für kürzeste u-v-Wege:

Falls es einen Knoten w gibt mit $d_{uw}^n < \infty$, $d_{ww}^n < \infty$ und $d_{wv}^n < 0$, dann ist die „kürzeste Länge" $-\infty$, sonst d_{uv}^n.

All-to-all Shortest Paths

Floyd-Warshall

Idea : Two things can happen about a vertex i, considering a walk from vertex u to v

- i does not get used in walk u-v
- i gets used in walk u-v

Definition of the DP table :

DP[i][u][v] = "The length of the shortest u-v walk that only uses the intermediate vertices from {1...i}"

intermediate vertices :

All-to-all Shortest Paths

Floyd-Warshall

DP[i][u][v] = "The length of the shortest u-v walk that only uses the intermediate vertices from {1...i}"

```
FLOYD-WARSHALL(G = (V, E), c)
1  DP[0][u][v] ← { 0          falls u = v ist,        no need to walk, we're already there
                 { c(u, v)    falls (u, v) ∈ E ist,   just walk that edge, you don't use any intermediate vertices
                 { ∞          sonst                   you can't reach v without using intermediate vertices
2  for i ← 1, ..., n do
3      for u ← 1, ..., n do
4          for v ← 1, ..., n do
5              DP[i][u][v] ← min(DP[i−1][u][v], DP[i−1][u][i] + DP[i−1][i][v])
                                 (don't use vertex i)          (use vertex i)
6  return DP
```

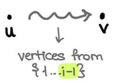
Runtime: O(n^3)

Solution at: DP[n][][]

"using all vertices"

DP [i-1][u][v]

shortest walk from u to v using {1...i-1}



DP [i-1][u][i] + DP[i-1][i][v]

shortest walk from u to i using {1...i-1} + shortest walk from i to v using {1...i-1}

All-to-all Shortest Paths

Floyd-Warshall , Negative Closed Walk Detection

∃ a negative closed walk ⟺ ∃ v with DP[n][v][v] < 0

DP[n][v][v] : The shortest walk from v to v using {1...n}

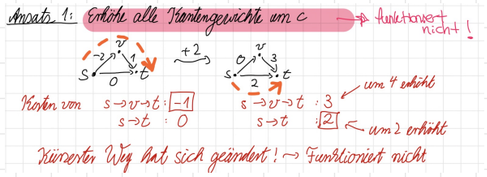
negative cycle detection

Johnson

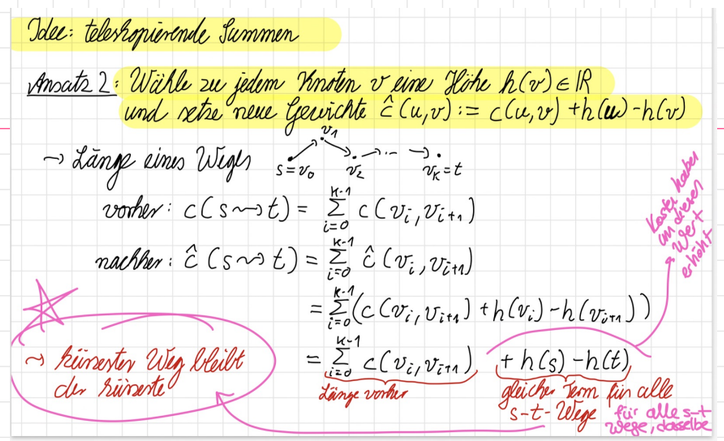
Idee:
- make all edge-weights ≥ 0
- n x Dijkstra

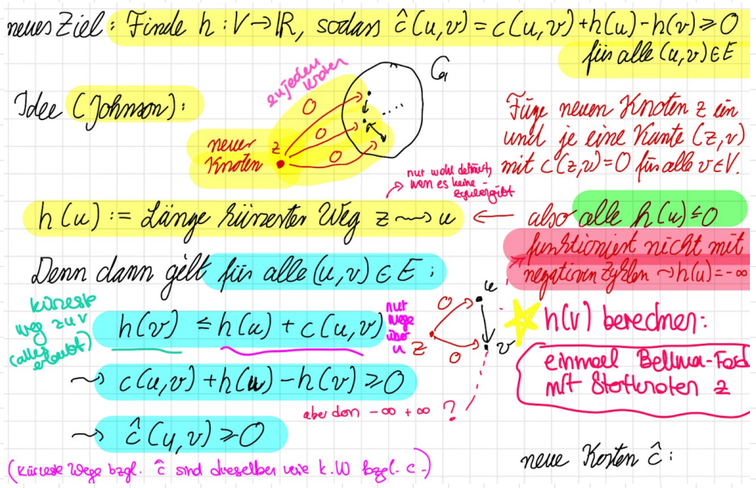
Making all edges ≥ 0:

- Why increasing every edge by minc doesn't work:



- Telescoping Sums Idee:





Algo:

Algorithmus: 1) Erzeuge z und neue Kanten — O(n)
2) Berechne h mit Bellman-Ford — O(m·n)
3) n-mal Dijkstra mit ĉ — n × O((m+n) log n) — oder n × O(m + n log n)

Insgesamt: O(n·(m+n)·log n) — O(mn + n² log n)

Besser als Floyd-Warshall für dünnbesetzte Graphen

Problem is the negative edges ! (we can't use dijkstra)

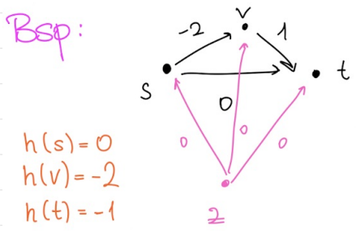
Idea : • Make all edge weights ≥ 0
• n x Dijkstra

DOES NOT WORK WITH NEGATIVE CYCLES !

h(v): length of shortest path from z to v → with Bellman-Ford

c'(u,v) = c(u,v) + h(u) - h(v)

Bsp:

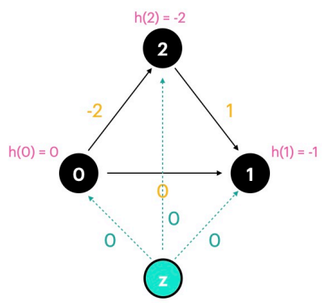


All-to-all Shortest Paths

Johnson - Making all edge weights ≥ 0

- Add a new vertex z , and connect it to every vertex in the original G with and edge with cost 0
- Find h(u) for every u
  h(u) := length of the shortest path from z to u
  with Bellman-Ford x1 from z
- Calculate c'(u,v) for every edge
  c'(u,v) := c(u,v) + h(u) - h(v)

c(0,2) = -2 c(2,1) = 1 c(0,1) = 0



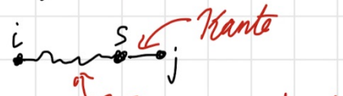
c'(0,2) = c(0,2) + h(0) - h(2) = -2 + 0 - (-2) = 0

c'(0,1) = c(0,1) + h(0) - h(1) = 0 + 0 - (-1) = 1

c'(2,1) = c(2,1) + h(2) - h(1) = 1 + (-2) - (-1) = 0

Matrices and Graphs

Annahme: $V = \{1, \dots, n\}$. Betrachte i-j-Wege der Länge k.

Jeder solche Weg sieht so aus: 

1) $L_{ij}^{(k)}$:= „Gibt es so einen Weg?"

↝ $L_{ij}^{(k)} = \bigvee_{s=1}^{n} \left(L_{is}^{(k-1)} \wedge L_{sj}^{(1)} \right)$ ↘ Veroderung

2) $M_{ij}^{(k)}$:= minimale Kosten von so einem Weg

↝ $M_{ij}^{(k)} = \min_{s=1,\dots,n} \{ M_{is}^{(k-1)} + M_{sj}^{(1)} \}$ ← „tropischer Halbring", min-plus-Halbring, fast ein Ring, außer dass additive Inverse fehlen

3) $N_{ij}^{(k)}$:= Anzahl solcher Wege ★ N^k ist die k-te Potenz von A_G.

↝ $N_{ij}^{(k)} = \sum_{s=1}^{n} N_{is}^{(k-1)} \cdot N_{sj}^{(1)}$, $N_{ij}^{(1)} = \begin{cases} 1, & \text{falls } (i,j) \in E \\ 0, & \text{sonst} \end{cases}$

einfach A_G potenzieren — Matrixprodukt $N^{(k-1)} \cdot N^{(1)}$ — Matrixmultiplikation — Adjazenzmatrix A_G ← Notation

★ **Satz:** Das Element (i,j) in A_G^k ← k-te Potenz ist die Anzahl der i-j-Wege der Länge k.

Beweis (Induktion nach k):

$k = 1$: ein Weg, falls $(i,j) \in E$, sonst kein Weg. ✓

$k > 1$: $N_{ij}^{(k)} = \sum_{s=1}^{n} N_{is}^{(k-1)} \cdot N_{sj}^{(1)}$

Einträge von A_G^{k-1} nach I.-H. — Einträge von A_G

$\overset{\text{I.-H.}}{=}$ Produkt (i-ter Zeilenvektor von A_G^{k-1}) $\cdot$ (j-ter Spaltenvektor von A_G)

$=$ (i,j)-Eintrag von A_G^k ▨

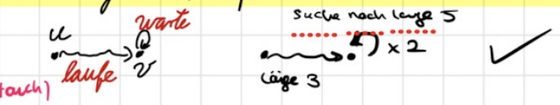
Anwendung 1: Wie viele Dreiecke gibt es in G? ← Kreise der Länge 3 — G gerichtet ohne Schleifen ← kein (v,v) in E

Antwort: Spur$(A_G^3)/3$ — $\mathrm{tr}((A_G)^3)/3$ — Σ Diagonale

Anwendung 2: Kann man von jedem Knoten aus alle anderen Knoten erreichen?

Trick: Füge alle Schleifen zu E hinzu, also alle $(v,v)^{(u,u)}$. ☺

↝ Wenn es einen $u \rightsquigarrow v$-Pfad gibt, dann gibt es auch einen $u \sim v$-Weg der Länge n. (Laufe zu v und warte dort.)



↝ Es genügt, A_G^n zu berechnen und zu prüfen, ob alle Einträge > 0 sind.

Laufzeit zur Berechnung von A_G^n:

naiv: $n-1$ Matrixmultiplikationen

Übungsaufgabe 3.5 ↘ iteriertes Quadrieren: $A_G^n = \begin{cases} A_G^{n/2} \cdot A_G^{n/2}, & \text{falls } n \text{ gerade} \\ A_G^{(n-1)/2} \cdot A_G^{(n-1)/2} \cdot A, & \text{falls } n \text{ ungerade} \end{cases}$

↝ $O(\log n)$ Matrixmultiplikationen