All Slides

Nil Ozer

Nil Ozer

A&D

Exercise Session 1

Nil Ozer

Outline

- Logistics
- A&D Overview
- Exam
- How to study for A&D
- Get to know you

- Induction
- Rest of the exercise sheet 0

Logistics

- Written Exercise Sheets
  - Published every week on Monday (in the afternoon),
  - Hand-in deadline : Sunday, 23:59
  - Discussion on Monday during exercise session
  - Groups on Moodle
  - Peer Grading at the end of the session
- CodeExpert exercises published biweekly
  - 2 exercises
  - Youtube Videos (similar exercises from my year)
- Moodle quiz in the first ~5 minutes (from week 3)
- Bonus Grade

Logistics - for me !

- Maintaining Moodle forums for your questions

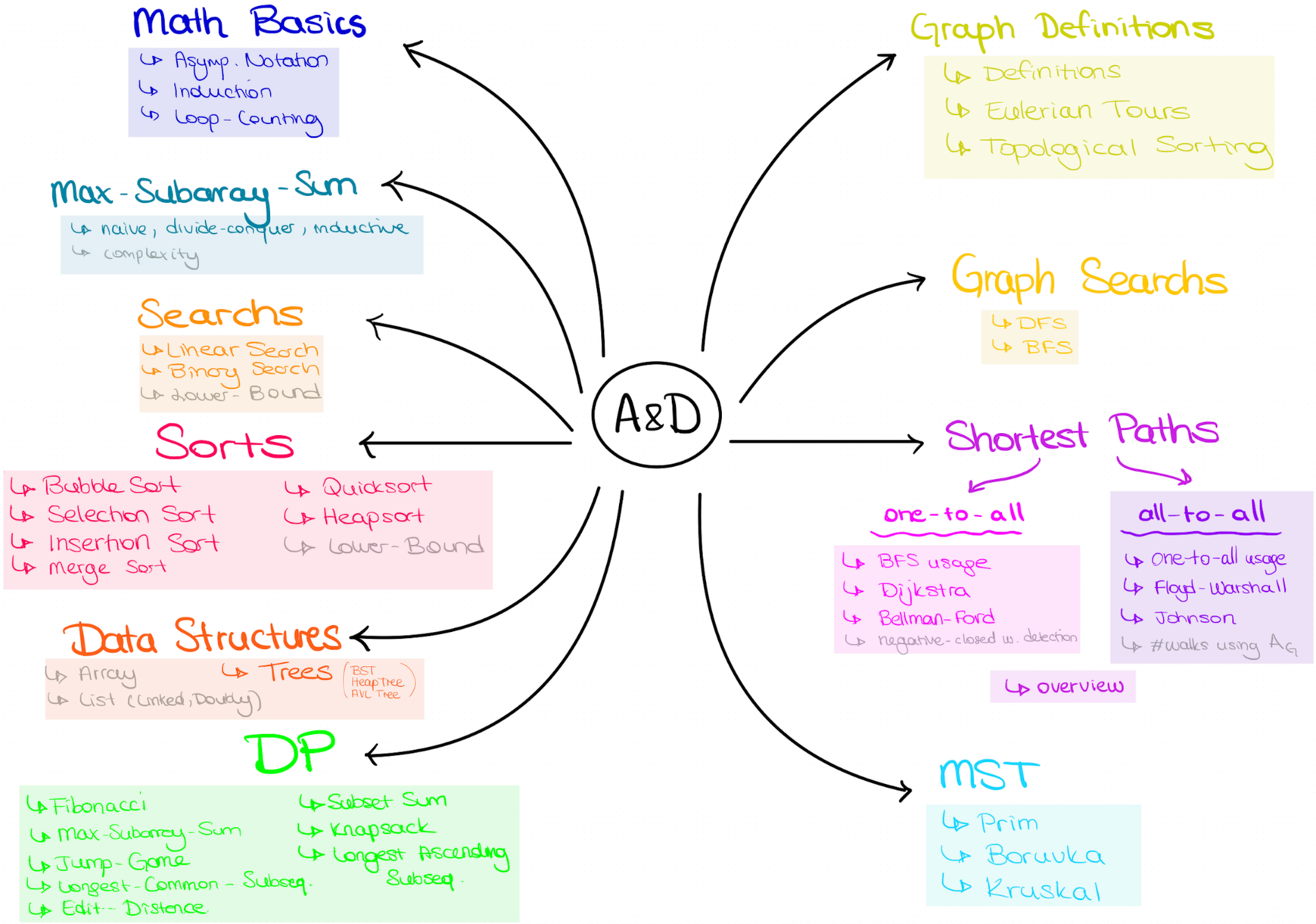
- We also have peer review
  - More consistent grading

- Weekly TA meetings with Head-TA’s and Prof. Lengler
  - Discussing each exercise
  - Another way for also your voice to be heard

Website Introduction

www.nilozer.com

A&D Overview


A&D
Math Basics
↳ 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, Heap Tree, 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

Exam

Exam

Written Exam

- %60
- 2h
- VIS

Programming Exam

- %40
- 3h
- CodeExpert

	T1 (23P)	T2 (15P)	T3 (8P)	T4 (14P)	Prog. (40P)	Σ (100P)
Score						
Corrected by						

Written Exam

- T1 & T2 : Basics
- T3 : DP
- T4 : Graph

- Tasks are always similar

Programming Exam

- One DP one Graph Exercise each year
- Old exams will be published at the end of the semester

Code Expert :

Exams	DP Exercise	Graph Exercise
2022 Summer	Left and Right	Two Trees
2022 Winter	Array Compression	Tree Augmentation
2021 Summer	Pair - Subsequence	Players on a Graph
2021 Winter 1	Shortest Uncommon Subsequence	Undirected Graph
2021 Winter 2	Longest Power-of-two Subsequence	Graph Sets
2020 Summer	Longest Palindromic Subsequence	Undirected Graph
2020 Winter	Shuffle	Binary Tree

Programming Exam

Test Exam 2022 Summer

- Two Trees
- Left and Right

Left and Right

You are given an array A of n integers, indexed from 0 to $n-1$.

You play the following game. You start with a score of 0. At each step of the game, you can make one of the following moves:

1. If A contains at least two elements, you can remove the **leftmost** and the **rightmost** element of A and add to your score the absolute value of their difference. For example, if the leftmost and the rightmost elements had values x and y, you add $|x-y|$ to your score.
2. You can remove the **leftmost** element of A with no change to your score.
3. You can remove the **rightmost** element of A with no change to your score.

Your task is to find the maximum score that you can obtain in the game. You need to implement your solution as a method **getMaximumScore(**n, A**)**.

Hint: Use dynamic programming with $D[i][j]$ representing the maximum score that you can obtain on $A[i], \ldots, A[j]$.

Grading (16 points):

- An $O(n^2)$ implementation gets 16 points and an $O(n^3)$ implementation gets 6 points.

Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.

Two Trees

You are given two rooted trees, A and B, with disjoint vertex sets. Tree A has vertices indexed by $a_0, \ldots, a_{n-1}$, with the root at index a_0, and tree B has vertices indexed by $b_0, \ldots, b_{n-1}$, with the root at index b_0. The edges in each tree are weighted by positive integers.

You want to add some new edges that connect leaves of A with leaves of B, thus creating a connected graph. Specifically, you can add an edge only if it goes between a leaf of A and a leaf of B. Any edge that you add has weight 0.

The distance between two vertices is defined as minimum total weight of a path that connects the two vertices.

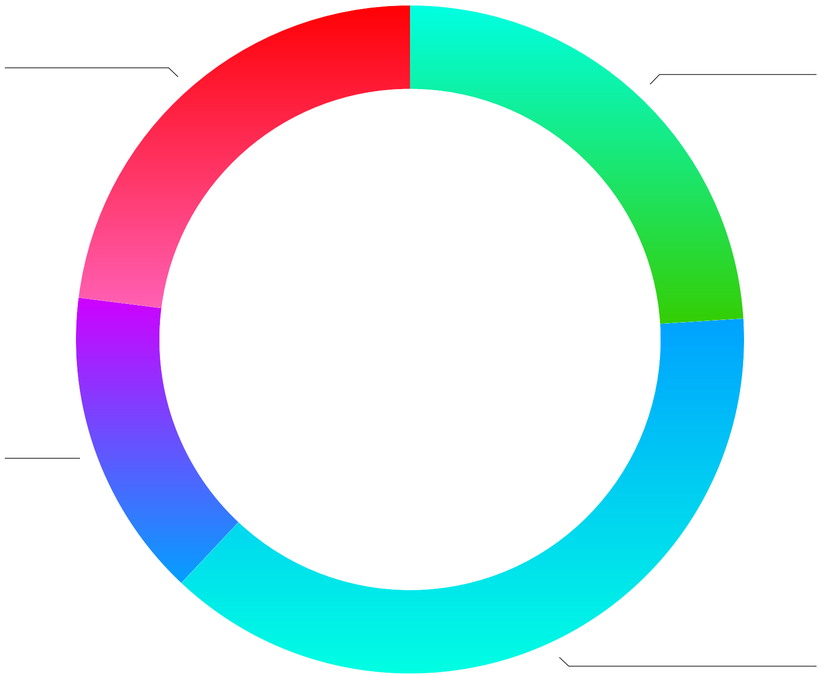
Given these two trees, you have to implement the following methods. All the answers are guaranteed to fit on an "int" type.

1. **edgeCount()**: Return the total number of edges that you can add between the two trees.
2. **minDistRoots()**: Return the minimum distance between the roots of the two trees that you can achieve by adding exactly one edge.
3. **cycle()**: Return 1 if you can add exactly two edges such that the resulting graph has a simple cycle (no repeated vertices) that contains the two roots.
4. **minDistCycle()**: Return the minimum length of a simple cycle (no repeated vertices) that contains the two roots that you can achieve by adding exactly two edges. You can assume such a simple cycle exists.

Point Distribution

Basics I

Basics II



DP

Graph

I got you !

How to study for A&D

During Semester

- Attend all lectures. If not possible, watch the lecture videos.
- Always come to the exercise session. Even if you fall back !
- Try to solve all written exercises. Work with your group !
- Ask questions ! exercise session , breaks, WhatsApp group, email , Moodle forum
- Summaries
- Feedback Feedback pools by me or contacting me directly

Get to know you

Let’s take a break

A&D 😊 👩🏻‍💻

WhatsApp group

Scan or upload this QR code using the WhatsApp camera to join this group

Induction

Summary

Importance :

* One of the most used tools in CS Area
* Appears everywhere → A&D, A&W (2nd)
  → DM
  → FMFP (4th)
* In every written exam for A&D

Induction

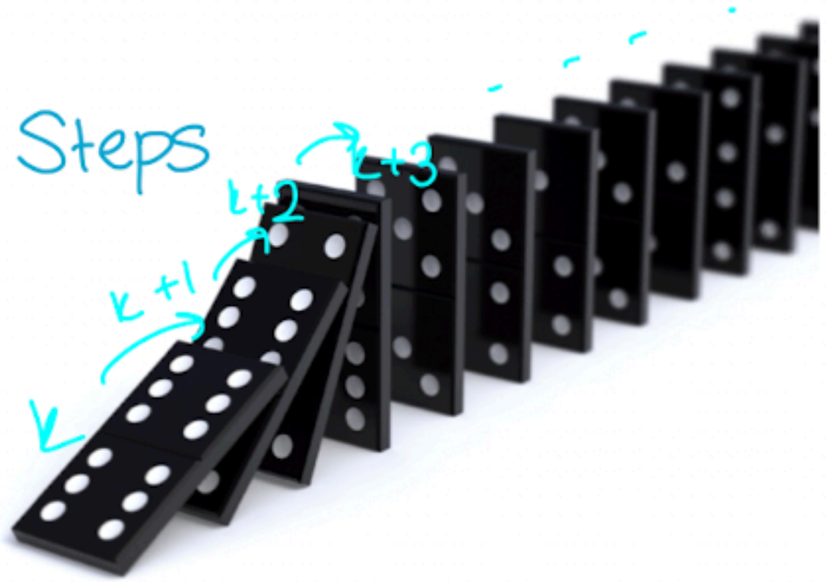
Summary

Intuition :

Domino Analogy

Base Case:

hit the first stone



Induction

Summary

Overview :

n : depends on Q

Task: Prove via induction that for every positive integer n, (for $n \geq 5$) we have a property { $\ldots = \ldots$ / $\ldots \leq \ldots$ / $\ldots \geq \ldots$ }

(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. ($n \geq 5$)

Induction

Exercises

Exercise 0.1 *Induction.*

(a) Prove by mathematical induction that for any positive integer n,

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

In your solution, you should address the base case, the induction hypothesis and the induction step.

Induction
Exercises

Exercise 0.1 *Induction.*

(a) Prove by mathematical induction that for any positive integer n,

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

In your solution, you should address the base case, the induction hypothesis and the induction step.

Solution:

Base Case.
Let $n = 1$. Then we have

$$1 = \frac{1 \cdot 2}{2}.$$

Induction Hypothesis.
Assume that the property holds for some positive integer k, that is we have

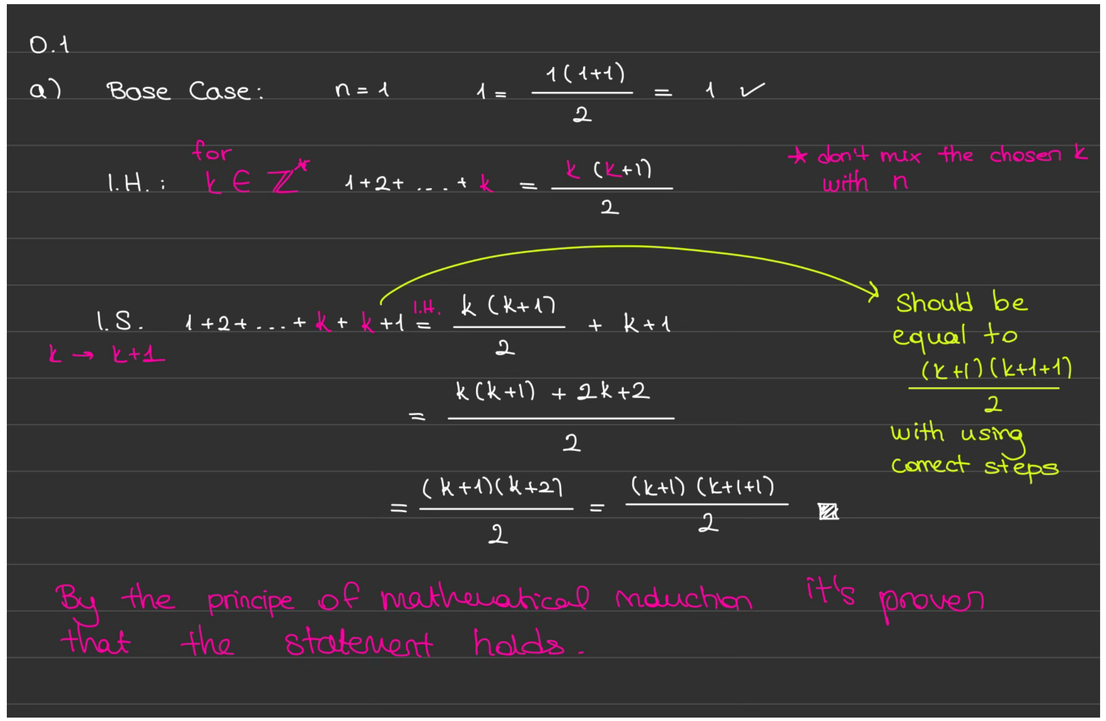
$$1 + 2 + \cdots + k = \frac{k(k+1)}{2}.$$

Induction Step.
We must show that the property holds for $k + 1$ summands. We have

$$\begin{aligned} 1 + 2 + \cdots + k + (k+1) &\overset{\text{I.H.}}{=} \frac{k(k+1)}{2} + k + 1 \\ &= \frac{k(k+1) + 2(k+1)}{2} \\ &= \frac{(k+1)(k+2)}{2}. \end{aligned}$$

We want to use the induction hypothesis, so in the first step above we separate the last term of the sum and then use the induction hypothesis.

By the principle of mathematical induction, the statement is true for any positive integer n.



Induction

Exam question

(b) **(This subtask is from August 2019 exam).** Let $T : \mathbb{N} \to \mathbb{R}$ be a function that satisfies the following two conditions:

$$T(n) \geq 4 \cdot T(\tfrac{n}{2}) + 3n \qquad \text{whenever } n \text{ is divisible by 2;}$$
$$T(1) = 4.$$

Prove by mathematical induction that

$$T(n) \geq 6n^2 - 2n$$

holds whenever n is a power of 2, i.e., $n = 2^k$ with $k \in \mathbb{N}_0$. In your solution, you should address the base case, the induction hypothesis and the induction step.

Induction

Tipps

- Solve whenever it appears on exercise sheets
- Exam task : FS 23

/ 3 P

b) *Induction:* Consider the sequence $\{L_n\}_{n\geq 1}$ defined via $L_1 = L_2 = 1$ and the recurrence $L_{n+1} = L_n + 2L_{n-1}$ for $n \geq 2$.

Show that for all $n \in \mathbb{N} \setminus \{0\}$, the following equation holds:

$$\sum_{i=1}^{n} 2^{n-i} \cdot L_i^2 = L_n L_{n+1}.$$

Exercise Sheet 0

Rest

- Preparation for **Asymptotic-Notation quiz** tasks

/ 5 P

a) *Asymptotic notation quiz:* For each of the following claims, state whether it is true or false. You get 1P for a correct answer, -1P for a wrong answer, 0P for a missing answer. You get at least 0 points in total.

Assume $n \geq 4$.

Claim	true	false
$n^3 + n^4 = \Theta(n^4)$	☐	☐
$n^{10} \leq O(\log(n)^{100})$	☐	☐
$1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \leq O(2^n)$	☐	☐
Suppose $a_1 = 1$ and $a_{i+1} = 3a_i + 1$ for all $i \geq 2$. Then $a_n \leq O(4^n)$.	☐	☐
$2^{10 \log(n)} = \Theta(2^{20 \log(n)})$	☐	☐

Exercise Sheet 0

Exercise 0.2

Asymptotic Growth

When we estimate the number of elementary operations executed by algorithms, it is often useful to ignore smaller order terms, and instead focus on the asymptotic growth defined below. We denote by $\mathbb{R}^+$ the set of all (strictly) positive real numbers and by $\mathbb{R}_0^+$ the set of nonnegative real numbers.

Definition 1. Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be two functions. We say that f *grows asymptotically faster than* g if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

This definition is also valid for functions defined on $\mathbb{R}^+$ instead of $\mathbb{N}$. In general, $\lim\limits_{n\to\infty} \frac{g(n)}{f(n)}$ is the same as $\lim\limits_{x\to\infty} \frac{g(x)}{f(x)}$ if the second limit exists.

For all the following exercises, you can assume that $n \in \mathbb{N}_{\geq 10}$. We make this assumption so that all functions are well-defined and take values in $\mathbb{R}^+$.

Exercise Sheet 0

Exercise 0.2

Definition 1. Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be two functions. We say that f *grows asymptotically faster than* g if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

Show that

(a) $f(n) := n \log n$ grows asymptotically faster than $g(n) := n$.

Exercise Sheet 0

Exercise 0.2

Definition 1. Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be two functions. We say that f *grows asymptotically faster than* g if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

Show that

(a) $f(n) := n \log n$ grows asymptotically faster than $g(n) := n$.

Solution:

We have

$$\lim_{n\to\infty} \frac{n}{n \log n} = \lim_{n\to\infty} \frac{1}{\log n} = 0$$

and hence, by Definition 1, $f(n) := n \log n$ grows asymptotically faster than $g(n) := n$.

Exercise Sheet 0

Exercise 0.3

The following theorem can be useful to compute some limits.

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*

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

Exercise Sheet 0

Exercise 0.3

Definition 1. Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be two functions. We say that f *grows asymptotically faster than* g if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

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

(b) $f(n) := e^n$ grows asymptotically faster than $g(n) := n$.

Exercise Sheet 0

Exercise 0.3

Definition 1. Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be two functions. We say that f *grows asymptotically faster than* g if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

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

(b) $f(n) := e^n$ grows asymptotically faster than $g(n) := n$.

Solution:

We apply Theorem 1 to compute

$$\lim_{x\to\infty} \frac{x}{e^x} \overset{\text{Thm.1}}{=} \lim_{x\to\infty} \frac{x'}{(e^x)'} = \lim_{x\to\infty} \frac{1}{e^x} = 0.$$

Hence by Definition 1, $f(n) := e^n$ grows asymptotically faster than $g(n) := n$.

Exercise Sheet 0

Exercise 0.4

Exercise 0.4 *Simplifying expressions.*

Simplify the following expressions as much as possible without changing their asymptotic growth rates.

Concretely, for each expression $f(n)$ in the following list, find an expression $g(n)$ that is as simple as possible and that satisfies $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+$.

(a) $f(n) := 5n^3 + 40n^2 + 100$

Exercise Sheet 0

Exercise 0.4

Exercise 0.4 *Simplifying expressions.*

Simplify the following expressions as much as possible without changing their asymptotic growth rates.

Concretely, for each expression $f(n)$ in the following list, find an expression $g(n)$ that is as simple as possible and that satisfies $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+$.

$$\text{(a)}\quad f(n) := 5n^3 + 40n^2 + 100$$

Let $g(n) := n^3$. Then we indeed have that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \left(5 + \frac{40}{n} + \frac{100}{n^3}\right) = 5 \in \mathbb{R}^+.$$

Exercise Sheet 0

Rest

- 0.2 , 0.3 , 0.4
- 0.5 : * Exercise
- This was all a math background preparation

Exercise Sheet 0

Exam tipps

/ 5 P

a) *Asymptotic notation quiz:* For each of the following claims, state whether it is true or false. You get 1P for a correct answer, -1P for a wrong answer, 0P for a missing answer. You get at least 0 points in total.

Assume $n \geq 4$.

Claim	**true**	**false**
$n^3 + n^4 = \Theta(n^4)$	☐	☐
$n^{10} \leq O(\log(n)^{100})$	☐	☐
$1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \leq O(2^n)$	☐	☐
Suppose $a_1 = 1$ and $a_{i+1} = 3a_i + 1$ for all $i \geq 2$. Then $a_n \leq O(4^n)$.	☐	☐
$2^{10\log(n)} = \Theta(2^{20\log(n)})$	☐	☐

Exercise Sheet 0

Exam tipps (informal)

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

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

Log- Rules:

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

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

Questions

Feedbacks , Recommendations

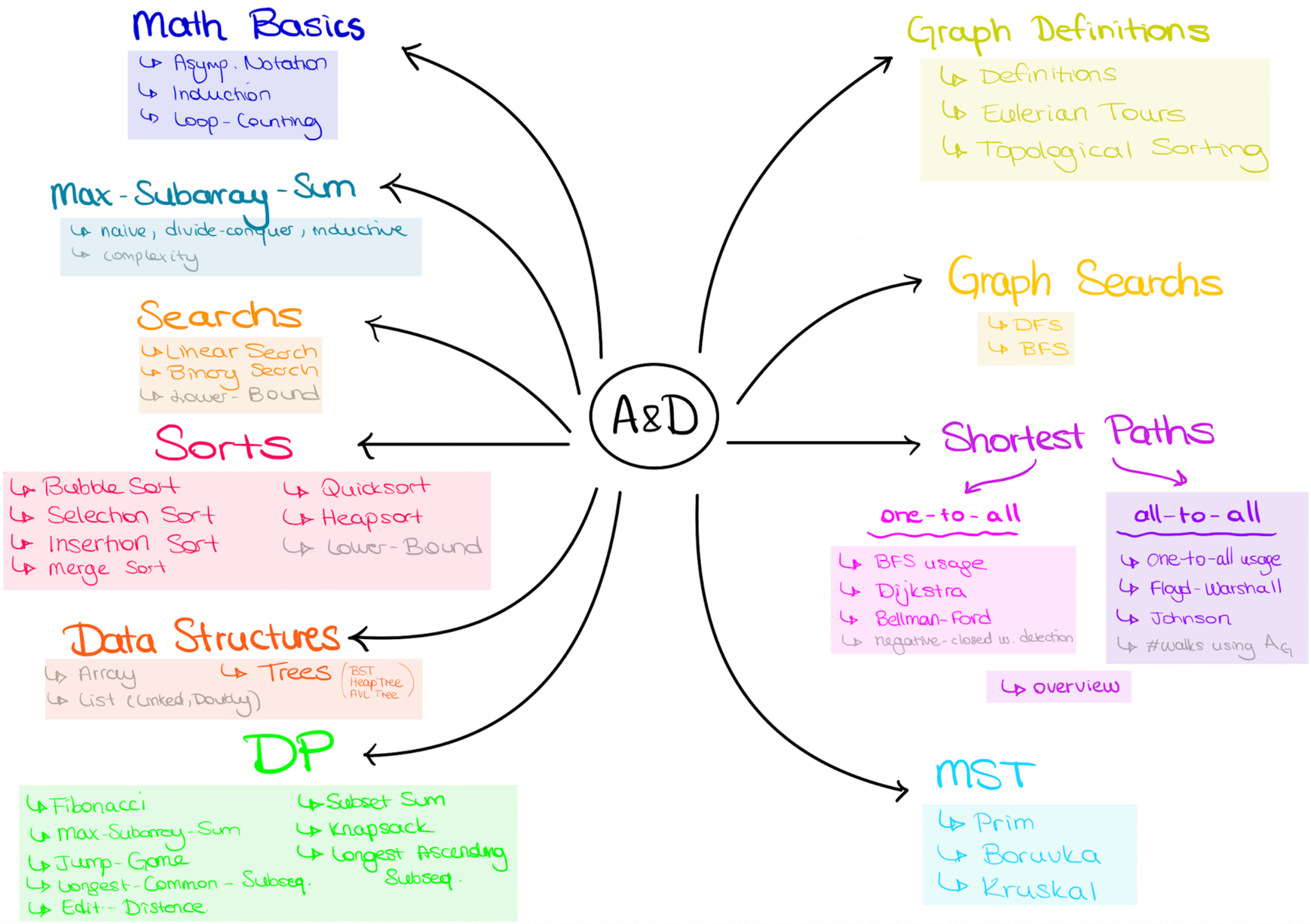
Nil Ozer

A&D

Exercise Session 2

Nil Ozer

A&D Overview


A&D
Math Basics
↳ 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, Heap Tree, 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

Outline

- Quick recap
- Quiz

- Asymptotic Notation

- Exercise Sheet 1
- Mini-exam

Quick recap

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

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

Log-Rules:

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

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

▶ **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**: $\boxed{b^x = \left(e^{\ln b}\right)^x = e^{(\ln b)\cdot x}}$.

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

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

$x^{100} = \left(e^{\ln(x)}\right)^{100} = e^{\ln(x)\cdot 100}$

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

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

Theorem 1. *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* $f \leq O(g)$ *and* $g \not\leq O(f)$.
- *If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C \in \mathbb{R}^+$, *then* $f \leq O(g)$ *and* $g \leq O(f)$.
- *If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, *then* $f \not\leq O(g)$ *and* $g \leq O(f)$.

Quiz

Versuche: 230 (24 von Ihrer Gruppe)

Asymptotic Notation

Asymptotic Notation

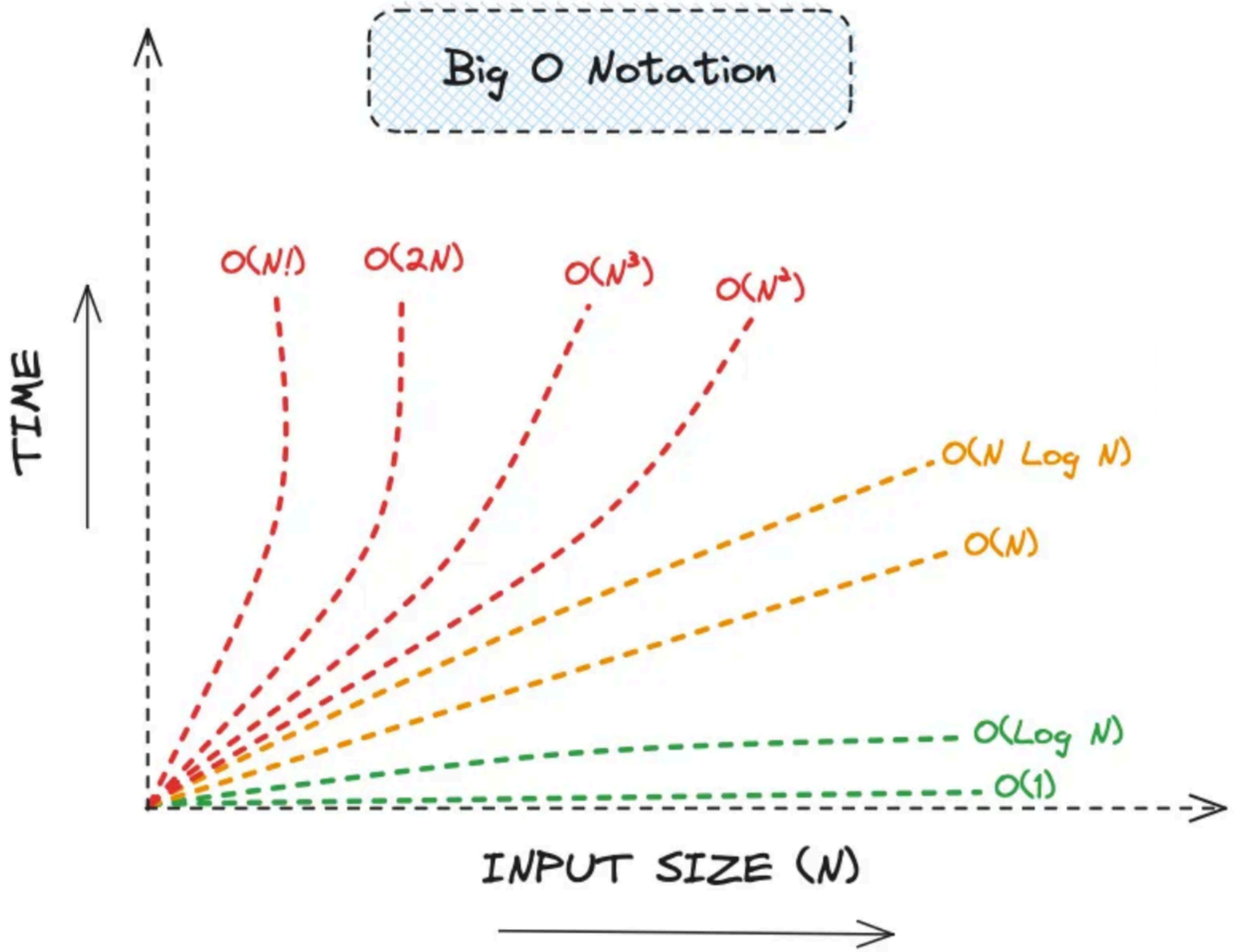
O-Notation

- Describes the upper-bound of an algorithm
- To express the worst-case running time
- Describes the performance of an algorithm as the amount of data increases
- As the input size of the algorithm grows, the execution time of the algorithm is also going to grow. (Constant, linear, logarithmic, exponential...)

- n represents the amount of data that we're passing in
- We don't care about the differences, when those differences are constant values. We care about n

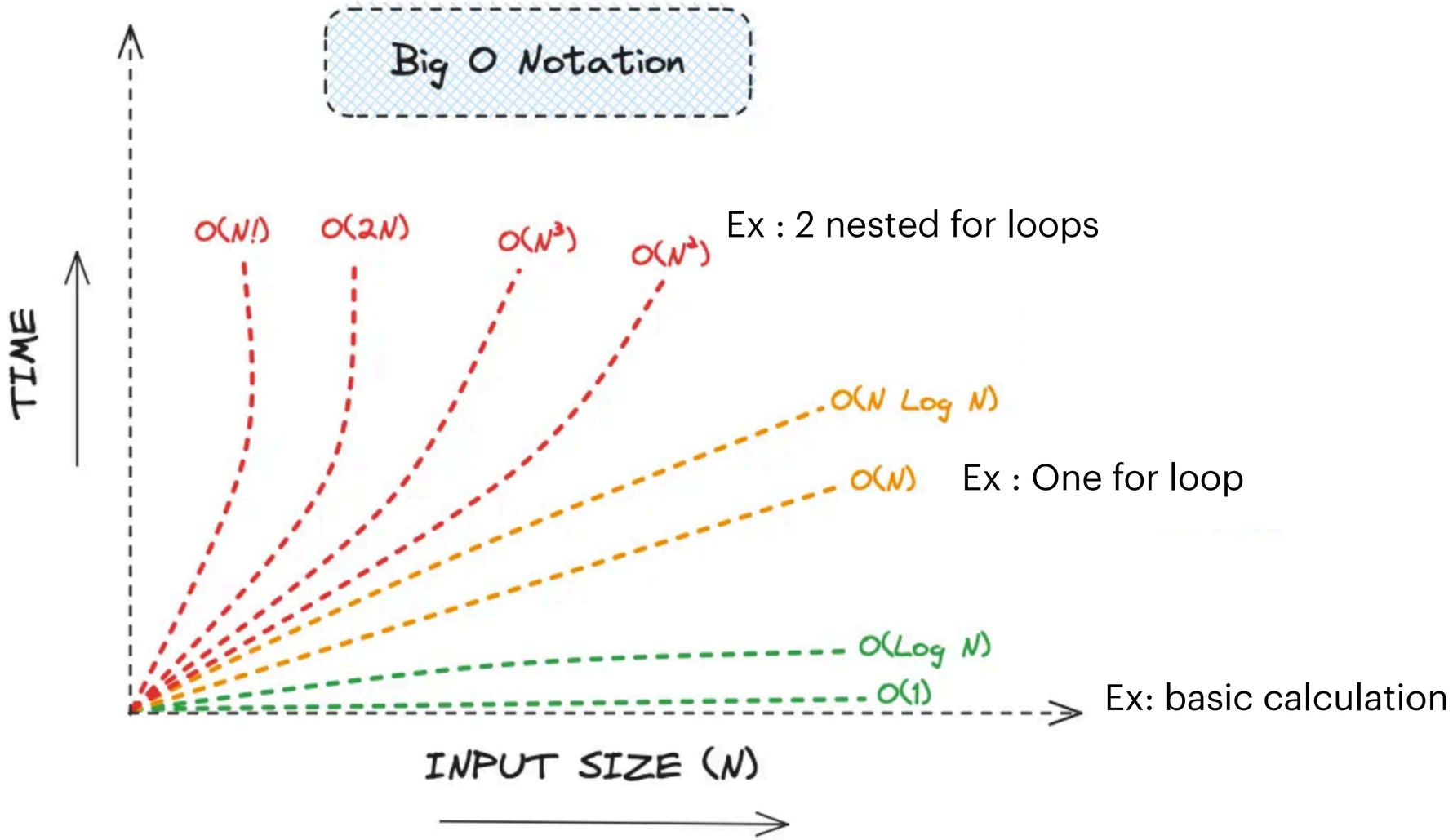
Asymptotic Notation

O-Notation



Asymptotic Notation

O-Notation



Ex : 2 nested for loops

Ex : One for loop

Ex: basic calculation

Asymptotic Notation

Ω-Notation

- Describes the lower-bound of an algorithm
- To express the best-case running time
- “ O-Notation reversed “

Ω :

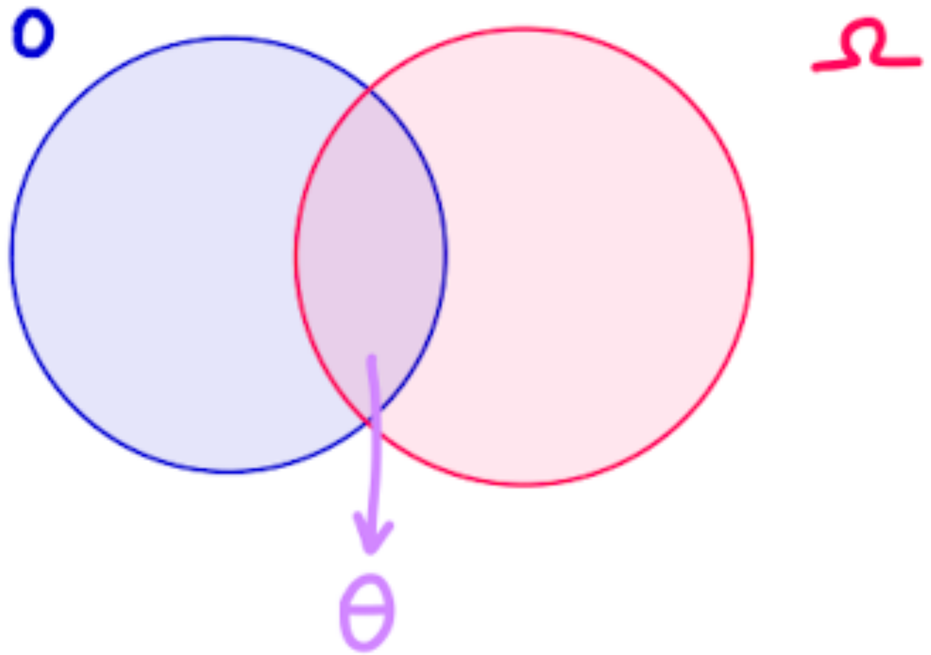
$\sqrt{n} \leq O(n)$

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

- n represents the amount of data that we’re passing in
- We don’t care about the differences, when those differences are constant values. We care about n

Asymptotic Notation

θ-Notation



Asymptotic Notation

Definitions

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

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

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

Asymptotic Notation

Definitions

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

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

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

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

Asymptotic Notation

Definitions

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

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

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

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

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* $f \neq \Theta(g)$.

Asymptotic Notation

Theorem

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* $f \neq \Theta(g)$.

Asymptotic Notation

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

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

Θ : $n \cdot \ln n = \Theta(\ln(n!))$

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

Asymptotic Notation

Exam question

FS20

/ 4 P

a) *Landau notation:* Fill out the quiz about asymptotic notation below. You get 1P for a correct answer, -1P for a wrong answer, 0P for a missing answer. You get at least 0 points in total.

claim	true	false
$(2n + n^2 + 3)^2 = \Theta(n^4)$	☐	☐
$\frac{n}{\log n} \leq \mathcal{O}(\sqrt{n})$	☐	☐
$\log(n!) = \Theta(n \log n)$	☐	☐
$\sum_{i=1}^{\log_5 n} 5^i \geq \Omega(n \log n)$	☐	☐

Let’s take a break

Asymptotic Notation

Exam Tipps

- Try to gain an intuition
- Mind the time ! (Spend 1 min on each exercise at max)
- Don’t forget the sub results of exercise sheets

- Our “cheat sheet” to the rescue !

- Remember the definitions

Exercise Sheet 1

(Non Bonus)

Gruppen	12
Abgegeben	12

Mini Exam

Induction + Asymptotic Notation

Questions

Feedbacks , Recommendations

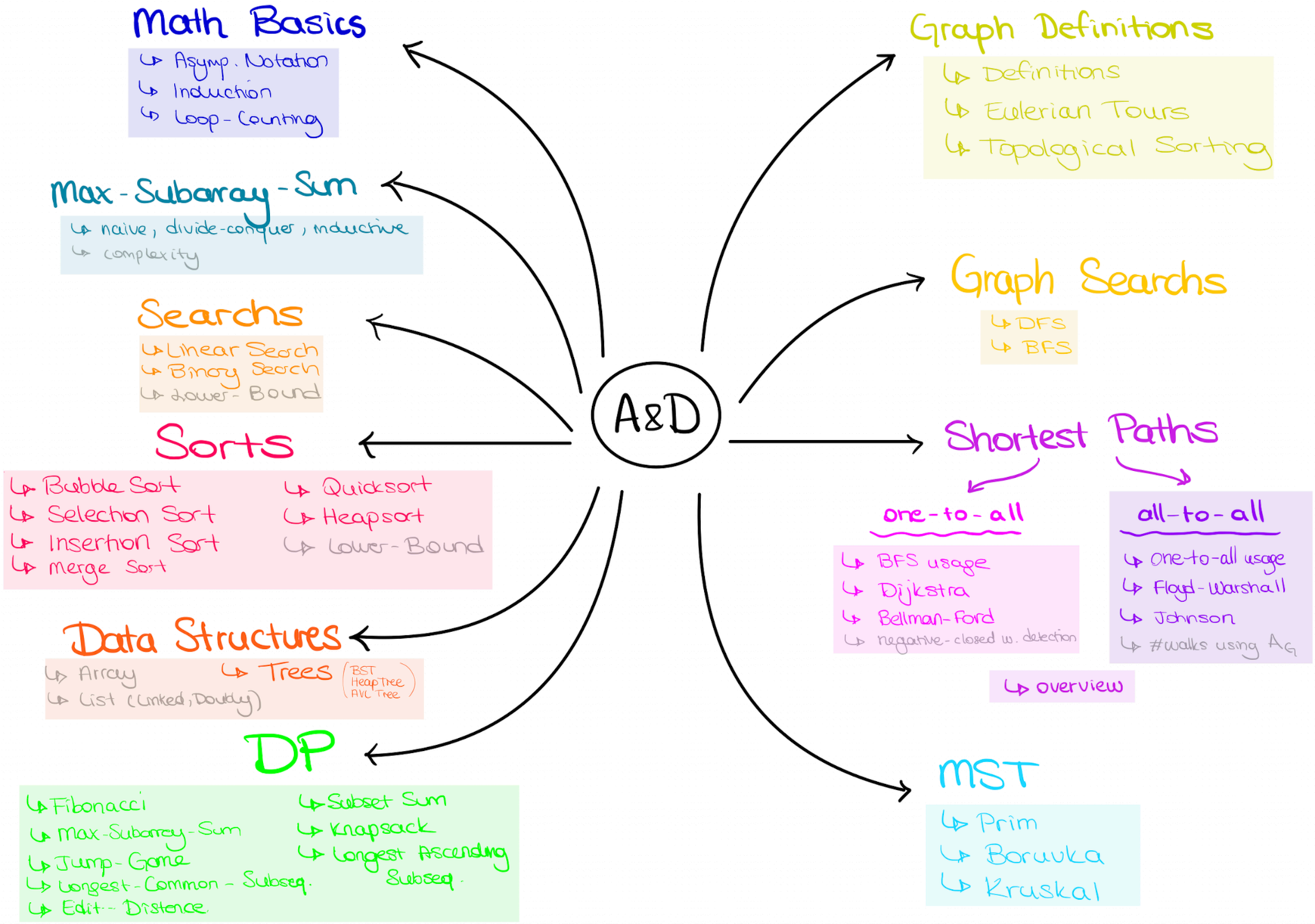
Nil Ozer

A&D

Exercise Session 3

Nil Ozer

A&D Overview


A&D
Math Basics
↳ 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

Outline

- Quick recap
- Quiz

- Exercise Sheet 1 Bonus Feedback
- Asymptotic Notation Kahoot

- Loop Counting

- Exercise Sheet 2 - non Bonus
- Mini-exam discussion
- Code Expert Introduction

Quick Recap

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 :

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

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

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

Quick Recap

Definitions

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

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

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

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

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* $f \neq \Theta(g)$.

Quiz

select one or more vs. select one

Log notation clarified

- From now on we use :
  - log for $\log_2(x)$
  - ln for $\log_e(x)$

- Will be in the notation sheet

Exercise Sheet 1

Bonus Feedback

Keep up the good work !!

- Watch out for the base case (n>=0 -> n=0)
- Try to have all of the intermediate steps

- Don’t rush to the solution, until you’ve gained the intuition by solving the task slowly!
- Log notation !

$$\log(m^3) = \log(m \cdot m \cdot m)$$
$$\log(m)^3 = \log m \cdot \log m \cdot \log m = \log^3(m)$$

work with
* log computation: $\frac{\log(m^3)}{\log(m)^3} = \frac{3 \log m}{\log(m)^3} = \frac{3}{\log(m)^2} \xrightarrow{n \to \infty} 0$

Asymptotic Notation Kahoot

Loop Counting

Let’s take a break

Loop Counting

Task Description

/ 4 P

a) *Counting iterations:* For the following code snippets, derive an asymptotic bound for the number of times f is called. Simplify the expression as much as possible and state it in Θ-notation as concisely as possible.

i) Snippet 1:

Algorithm 1

```
for j = 1, ..., n^2 do
    for k = 1, ..., j do
        f()
    f()
```

ii) Snippet 2:

Algorithm 2

```
for j = 1, ..., n do
    k ← 1
    while k ≤ j^2 − 1 do
        ℓ ← 1
        while ℓ ≤ n do
            f()
            ℓ ← 2ℓ
        k ← k + 1
```

Your solution consist of :

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

Loop Counting

Learn with an example !

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.

(a)

Algorithm 1

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

(b)

Algorithm 2

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

Loop Counting

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

Loop Counting

Exam question

FS23

Theory Task T2. / 15 P

In this part, you should **justify your answers briefly**.

/ 4 P a) *Counting iterations:* For the following code snippets, derive an asymptotic bound for the number of times f is called. Simplify the expression as much as possible and state it in Θ-notation as concisely as possible.

i) Snippet 1:

Algorithm 1

```
for i = 1, ..., n do
    for j = 1, ..., i^2 do
        f()
    f()
```

ii) Snippet 2:

Algorithm 2

```
for i = 1, ..., n do
    k ← 1
    while k ≤ i^2 do
        f()
        k ← 2k
    f()
```

Loop Counting

Exam Tipps

- T2 task
- 2 snippets
  - First one easier second one harder
- If it’s needed, the master theorem will be there

- Show your work !
- Solve it, whenever it appears !

- It’s relatively easy once you’ve practiced it !

Exercise Sheet 2

(Non Bonus)

Gruppen	11
Abgegeben	11

Mini Exam Discussion

Induction + Asymptotic Notation

Code Expert Introduction

Questions

Feedbacks , Recommendations

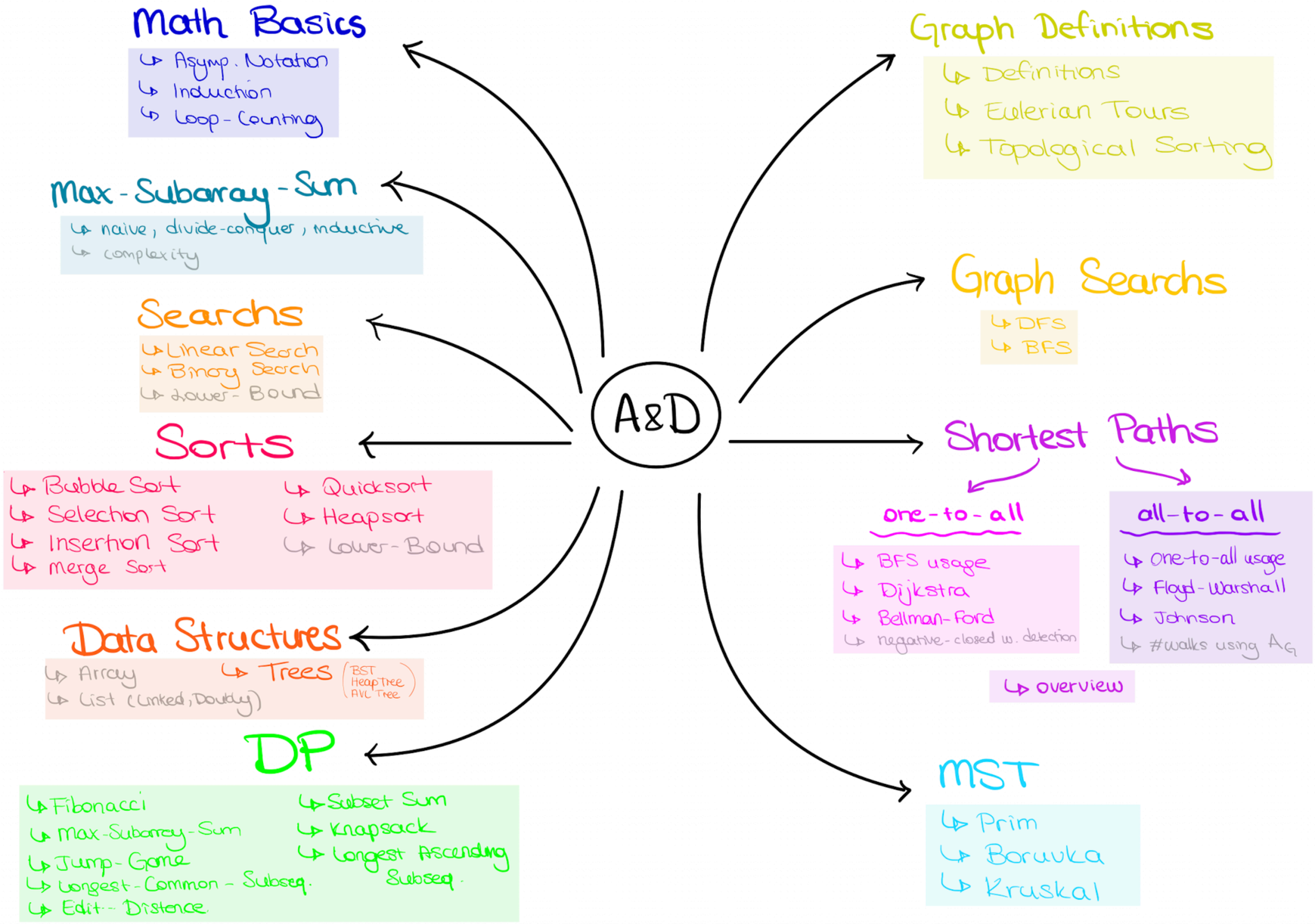
Nil Ozer

A&D

Exercise Session 4

Nil Ozer

A&D Overview


A&D
Math Basics
↳ 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

Outline

- Quiz

- Log Bases
- Exercise Sheet 2 Bonus Feedback
- Exercise Sheet 3 - non Bonus

- Search
- Sorts I

- Next week

Quiz

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

$$\log_b(x) = \frac{\log_a(x)}{\log_a(b)}$$

Exercise Sheet 2

Bonus Feedback

- Follow the task description.
  - If it says use theorem 1, use theorem 1!

- Pay attention to the little notes.
  - Exam grading won’t be like this

- Keep up the good work !

Exercise Sheet 3

Non Bonus

- First experience with algorithm construction (no points)
  - Pseudocode
  - Correctness
  - Runtime
  - Making it “besser”
- Next time , you’re graded ! (Exercise Sheet 4)

- Hope you solved it ! If not, you can still send it via email for this week !
  - Look at the notes ! Ask questions !

Exercise Sheet 3

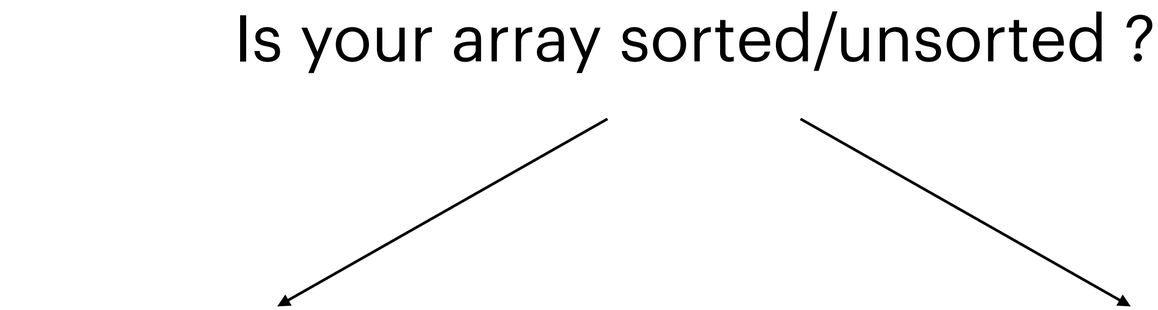
Peer Grading

- 3.1.b this week
- Emails are already sent
  - Correction : In the email I accidentally forgot to change the part where it says that you'll be using Exercise Sheet 2 to peer grade. (It's obviously Exercise Sheet 3)
  - Did not want to spam you

Searchs/Sorts I

Searchs

Searchs



Linear Search

Binary Search

Linear Search

Input : unsorted array , searched value

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

Runtime : O(n)

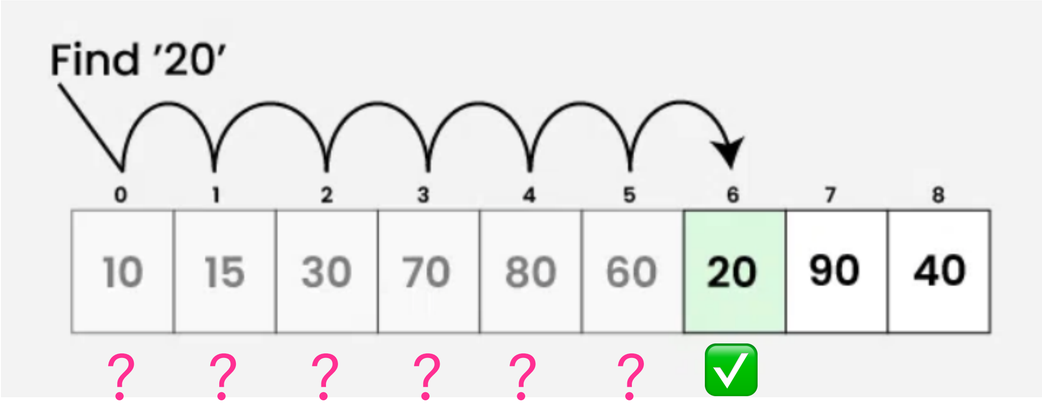
Pseudocode :

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



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

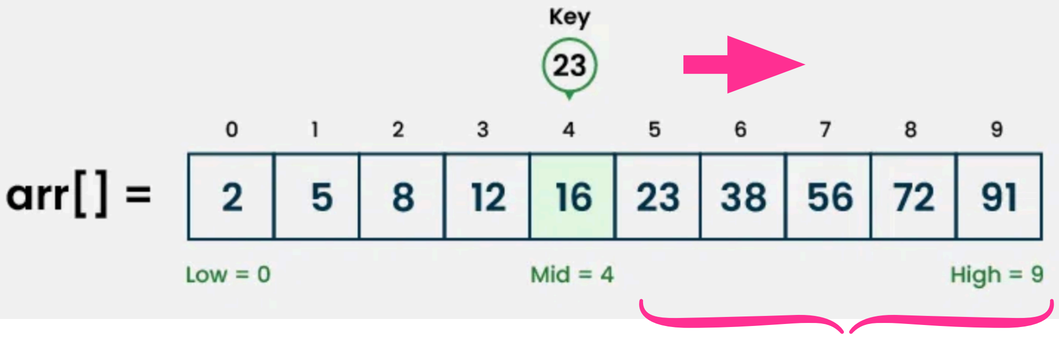
Code Example

Binary Search

Illustration

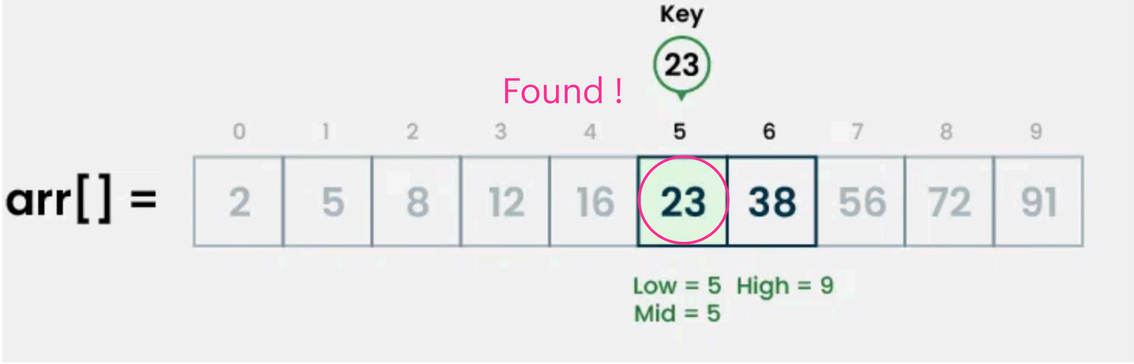
Initially | Find Key = 23 using Binary Search

Step 1 | Key (i.e., 23) is greater than current mid element (i.e., 16). The search space moves to the right.



Step 2 | Key is less than the current mid 56. The search space moves to the left.

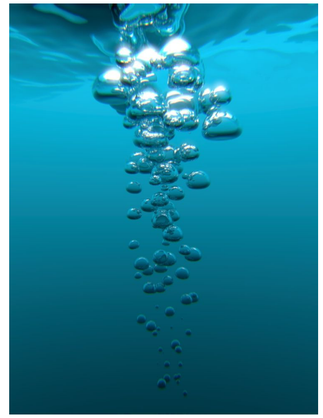
Step 3 | If the key matches the value of the mid element, the element is found and stop search.



Sorts I

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]
```

Illustration

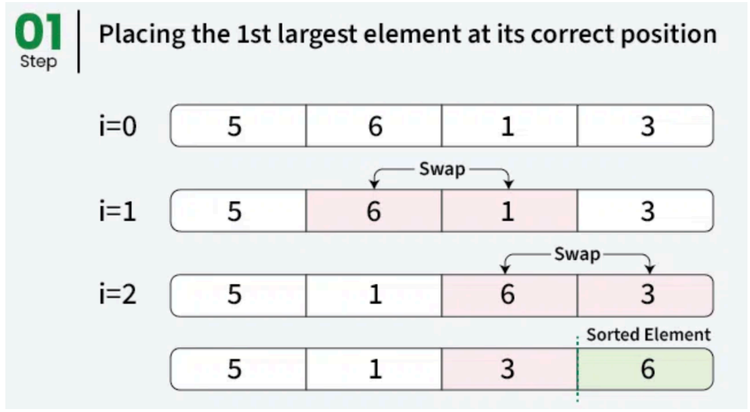
Code Example

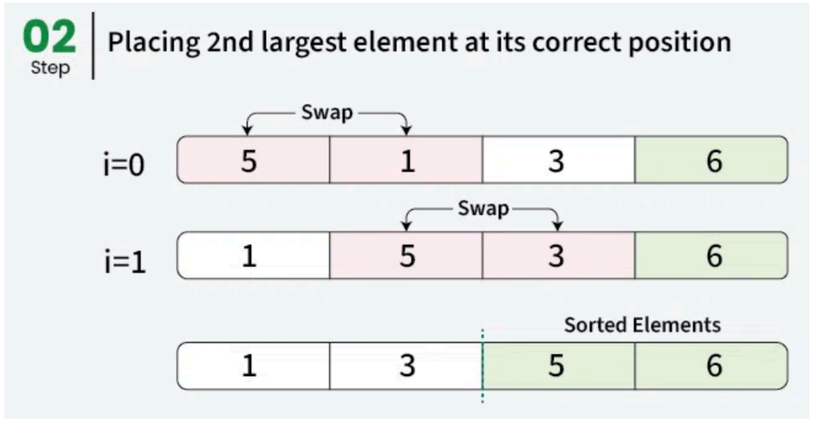
Bubble Sort

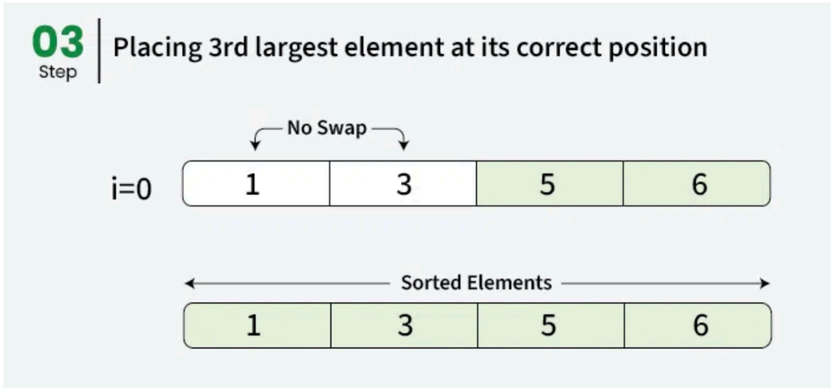
Illustration

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]
```







Sorted part

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]
```

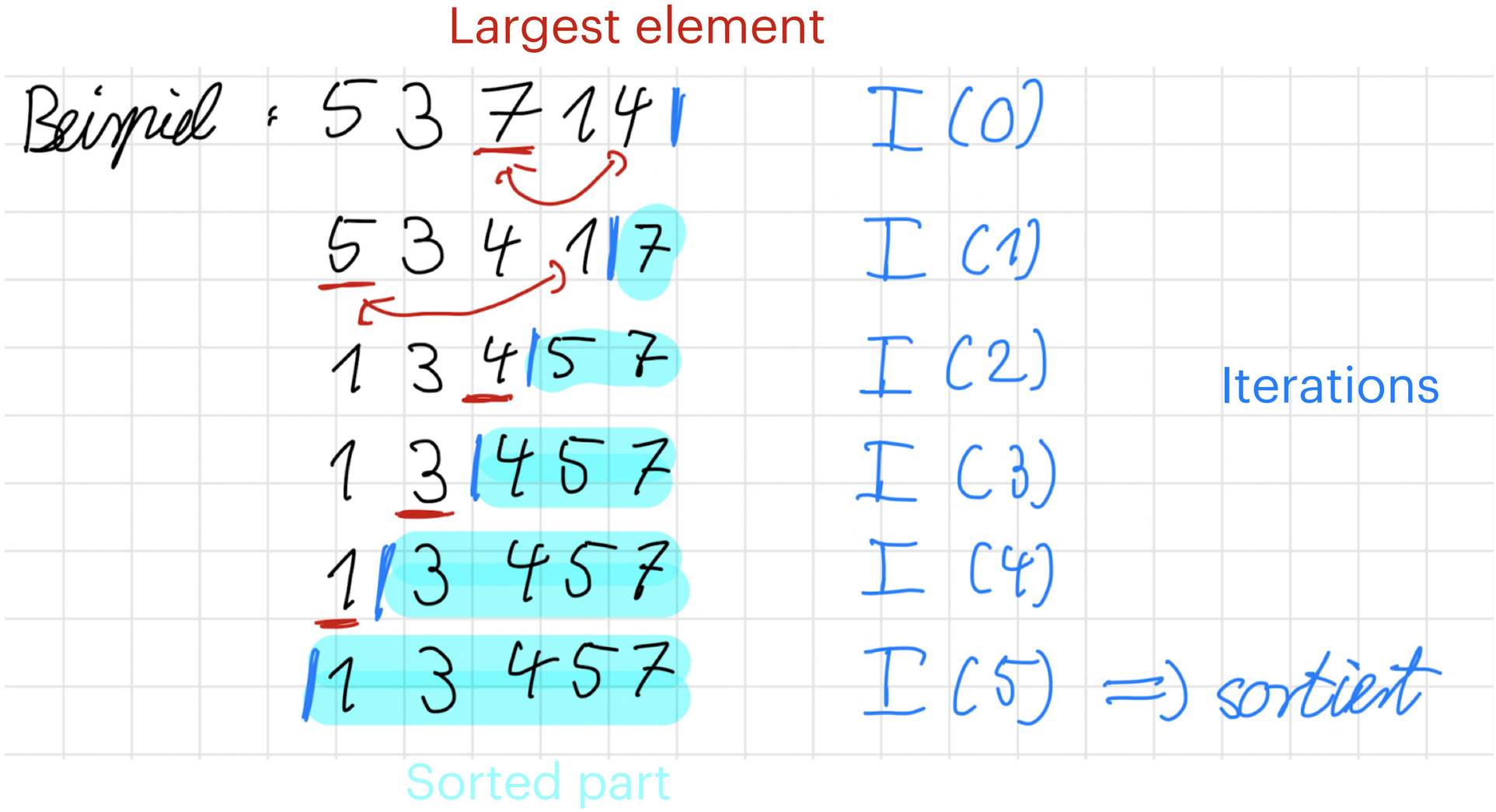
Illustration

Code Example

Selection Sort

Illustration

```
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]
```



Insertion Sort

Idea : insert curr to the correct position , “verschiebe”, shift rest

Input : unsorted array

Output : sorted array

Runtime: $O(n^2)$

#Comparisons: O(nlogn)

#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]                           ▷ merke A[j]
4      verschiebe A[k, ..., j − 1] nach A[k + 1, ..., j]
5      A[k] ← x
```

Illustration

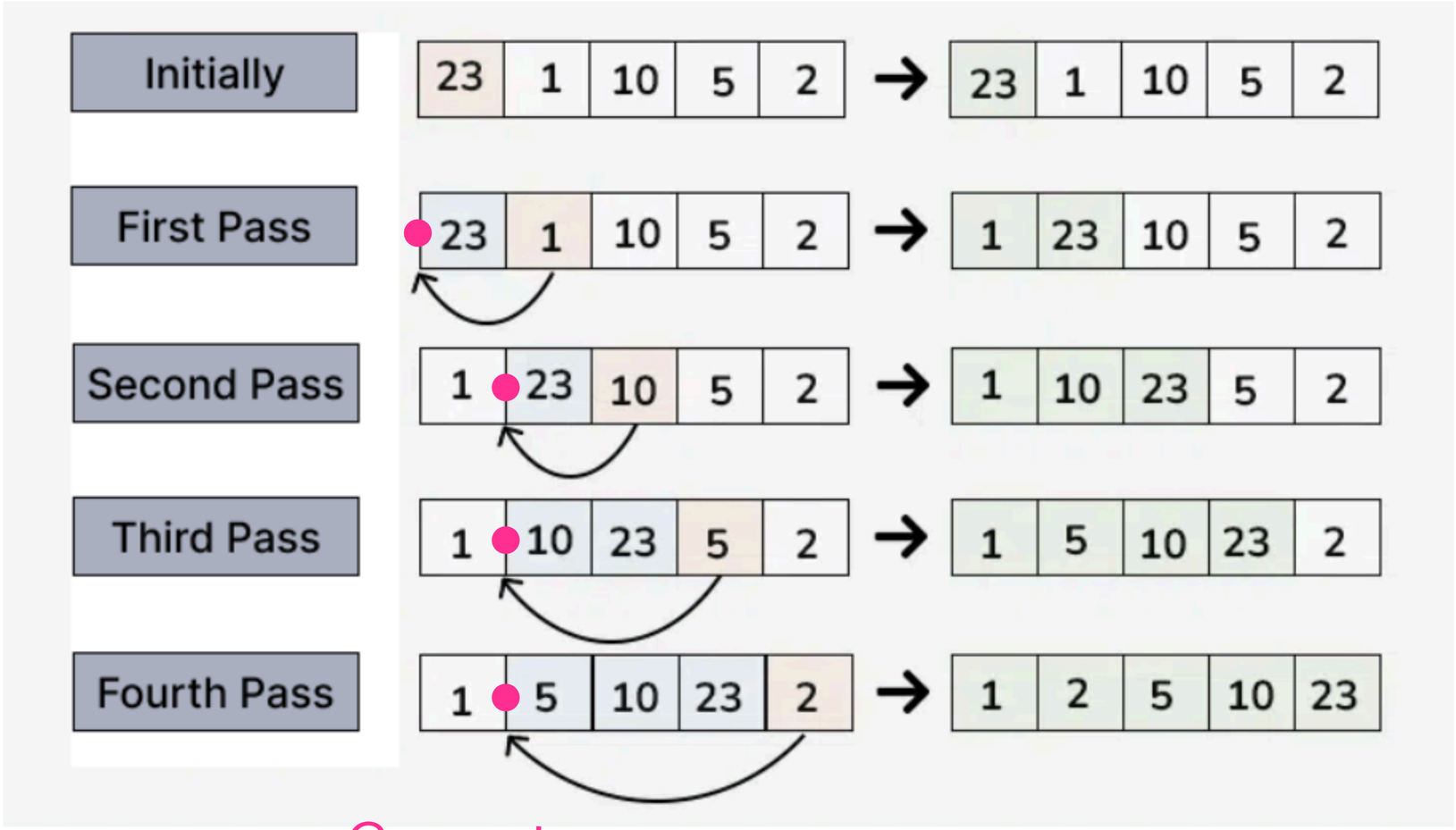
Code Example

Insertion Sort

Illustration

Algorithm 1 Insertion Sort (input: array $A[1 \ldots n]$).

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



Merge Sort

 Idea : Divide and Conquer

Input : unsorted array

Output : sorted array

Runtime: O(nlogn)

#Comparisons: O(nlogn)

#Swaps: O(nlogn)

Pseudocode :

MERGE$(A[1..n], l, m, r)$

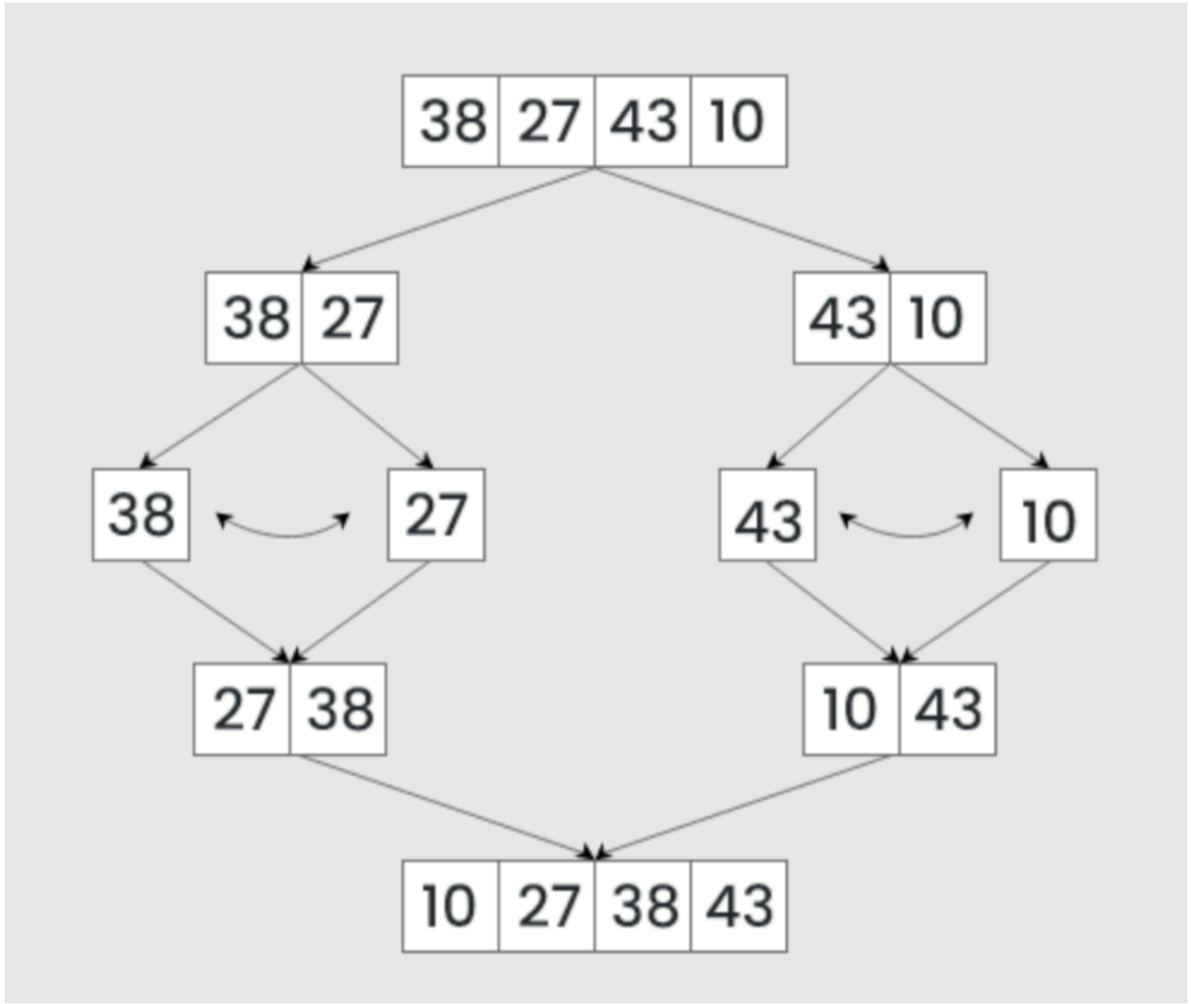
```
1  B ← new Array with r − l + 1 cells          ▷ so gross wie A[l, ..., r]
2  i ← l                                       ▷ erstes unbenutztes Element in linker Hälfte
3  j ← m + 1                                   ▷ erstes unbenutztes Element in rechter Hälfte
4  k ← 1                                       ▷ 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

Merge Sort

Illustration



Costs Overview

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

Let’s take a break

Searchs/Sorts 1

Exam Question (FS20)

/ 2 P

e) *Sorting algorithms:*

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

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

Algorithm:

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

Algorithm:

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

Algorithm:

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

Algorithm:

Searchs/Sorts 1

Exam Question (FS23)

/ 4 P

g) *Sorting algorithms quiz:* For each of the following claims, state whether it is true or false. You get 1P for a correct answer, -1P for a wrong answer, 0P for a missing answer. You get at least 0 points in total.

Claim	**true**	**false**
There exist arrays of length n which can be sorted with `BubbleSort` after $\Theta(n)$ swaps.	☐	☐
There exist arrays of length n for which the runtime of `InsertionSort` is $\Theta(n)$.	☐	☐
Consider a sequence of n numbers $\{x_1, ..., x_n\}$, where $0 \leq x_i \leq 1000$, $\forall i = 1, ..., n$, is given as input. There exists an algorithm with runtime $O(n)$ which sorts any such sequence.	☐	☐
There exist a comparison-based sorting algorithm that can sort any array of length n in runtime $O(n)$.	☐	☐

Searchs/Sorts 1

Exam Question (HS20)

/ 3 P

g) *Sorting algorithms:*

i) Consider the sequence $6, 5, 4, 1, 2, 3$. How many swaps does `Bubble Sort` perform to sort this sequence? *Give the exact number of swaps required.*

ii) Consider the sequence $6, 5, 4, 1, 2, 3$. How many swaps does `Selection Sort` perform to sort this sequence? *Give the exact number of swaps required.*

iii) Let $n \in \mathbb{N}$ be an even number and consider the sequence with the following structure:

$$2, 1, 4, 3, 6, 5, \ldots, n, n-1.$$

How many swaps does `Insertion Sort` perform to sort this sequence? *Give the exact number, not just the asymptotics.*

Searchs/Sorts 1

Exam Tipps

- Whenever you have to learn a new algorithm :
  - Know the pseudocode
  - Be able to do an example by hand
  - Know the runtime ! (Here also #comparisons, #swaps)
  - (Know how to implement it)
- Types of questions in the exam
  - T/F (Sorting algorithms quiz) (most recent exams, until FS21)
  - #comparisons, #swaps
  - Counting swaps for a particular example (HS20)
  - Invariant proving (FS21 , HS20)

Formal proof of correctness

Bubble Sort

Exercise Sheet 4

Algorithm Construction expectations

```
SMALLESTINTEGER ( n , l, r)
    m ← ⌊(l+r)/2⌋ ,  f_m ← f(m)
    if  l>=r && f_m <= n  then      // bounds meet , f(T) ≥ N
        return  m                   //  return m (T)
    else if  f_m > n                // if value > N , reduce upperB
        return  SMALLESTINT (n, l, m)   // recursively call with r = m
    else if  f_m < n                // if value < N , increase lowerB
        return  SMALLESTINT (n, m+1, r)  // recursively call with l = m+1

upperB ← UPPERBOUND ( N, 1]     // upperBound calculation
SMALLESTINT (N, 0, upperB)      // call with l=0 , r= upperB

Correctness: Follows directly from explanation with comments.
             A value is only returned if bounds have
             met and desired T is found.

Runtime: T(T) = 2 · T(n/2) + 3·C   ≤ O(log n)
```

```
UPPERBOUND (N, T)        // We start by calling algo for (N, 1)
    if (f(T) ≥ N) then   // desired T found
        return T         // return T
    else
        return UPPERBOUND (N, 2T)  // T is less than N still
                                   // until it's ≥ N double T , repeat

Runtime  T(T) = T(T/2) + C  --MT a=0, b=0-->  O(T^(log_2 1) log T) = O(log T)

Correctness is proven by the descriptions of algo.
```

- Proper Pseudocode
- Correctness proof easier with comments
- Runtime

Next week ...

Group Change !

Prize ?

Questions

Feedbacks , Recommendations

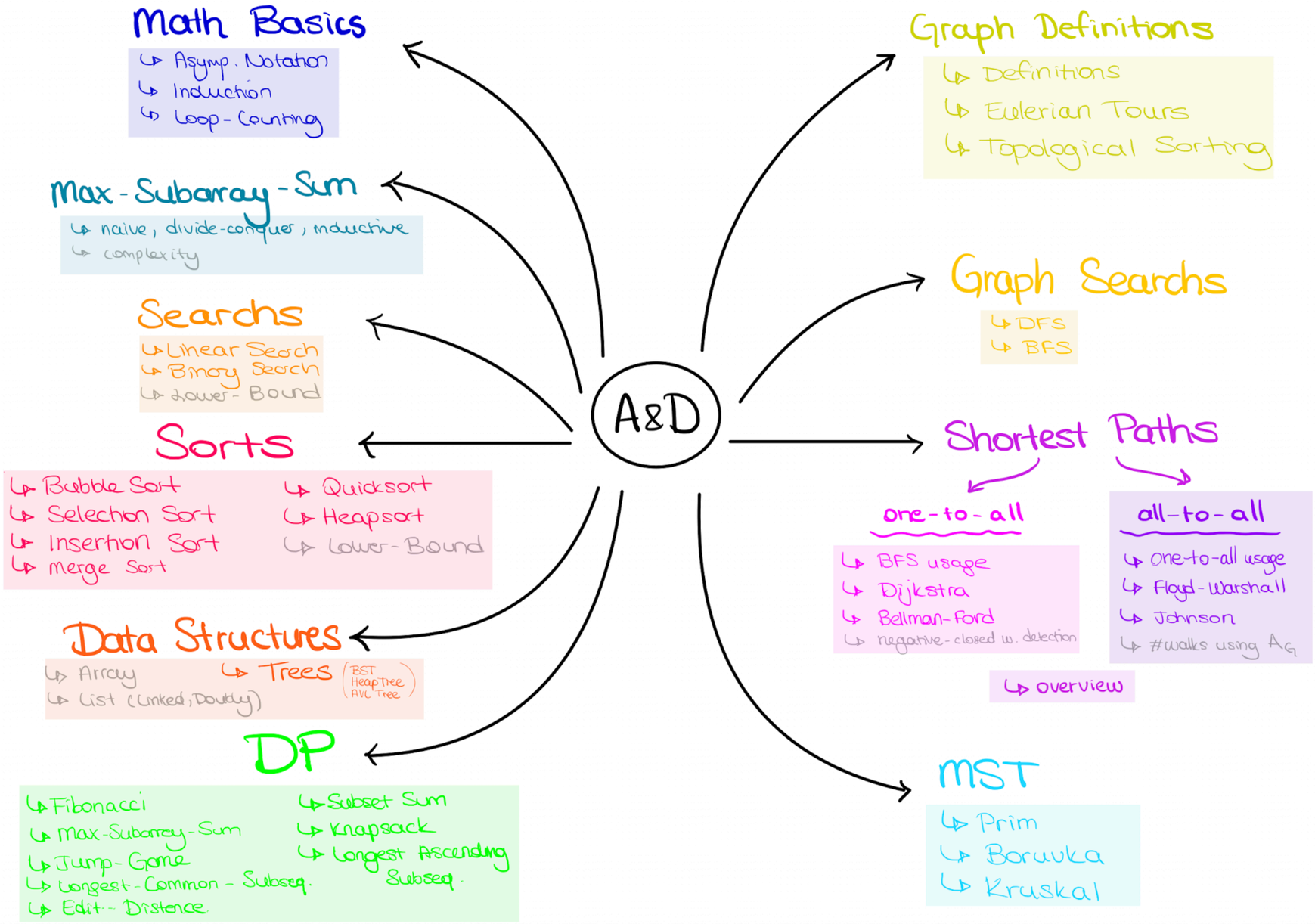
Nil Ozer

A&D

Exercise Session 5

Nil Ozer

A&D Overview


A&D
Math Basics
↳ 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

Outline

- Quiz

- Exercise Sheet 3 Bonus Feedback
- Exercise Sheet 4 - non bonus

- Sorts II
- Sorting Algorithms Kahoot !

- Data Structures I

Quiz

Exercise Sheet 3

Bonus Feedback

- Follow the task description.
  - If it says use the definition, use the definition !
  - Check the master solution to learn how to argue with definitions

- Loop Counting
  - You can switch to Θ-Approximation earlier, but this is risky !
    - Check the master solution

- Pay attention to the little notes. Keep up the good work !

Exercise Sheet 4

Non Bonus

- 4.1 Applying the master theorem

- 4.2 Asymptotic Notation Quiz

Exercise Sheet 4

Peer Grading

- 4.4 this week
- Emails are already sent
- New groups !

Sorts II

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 :

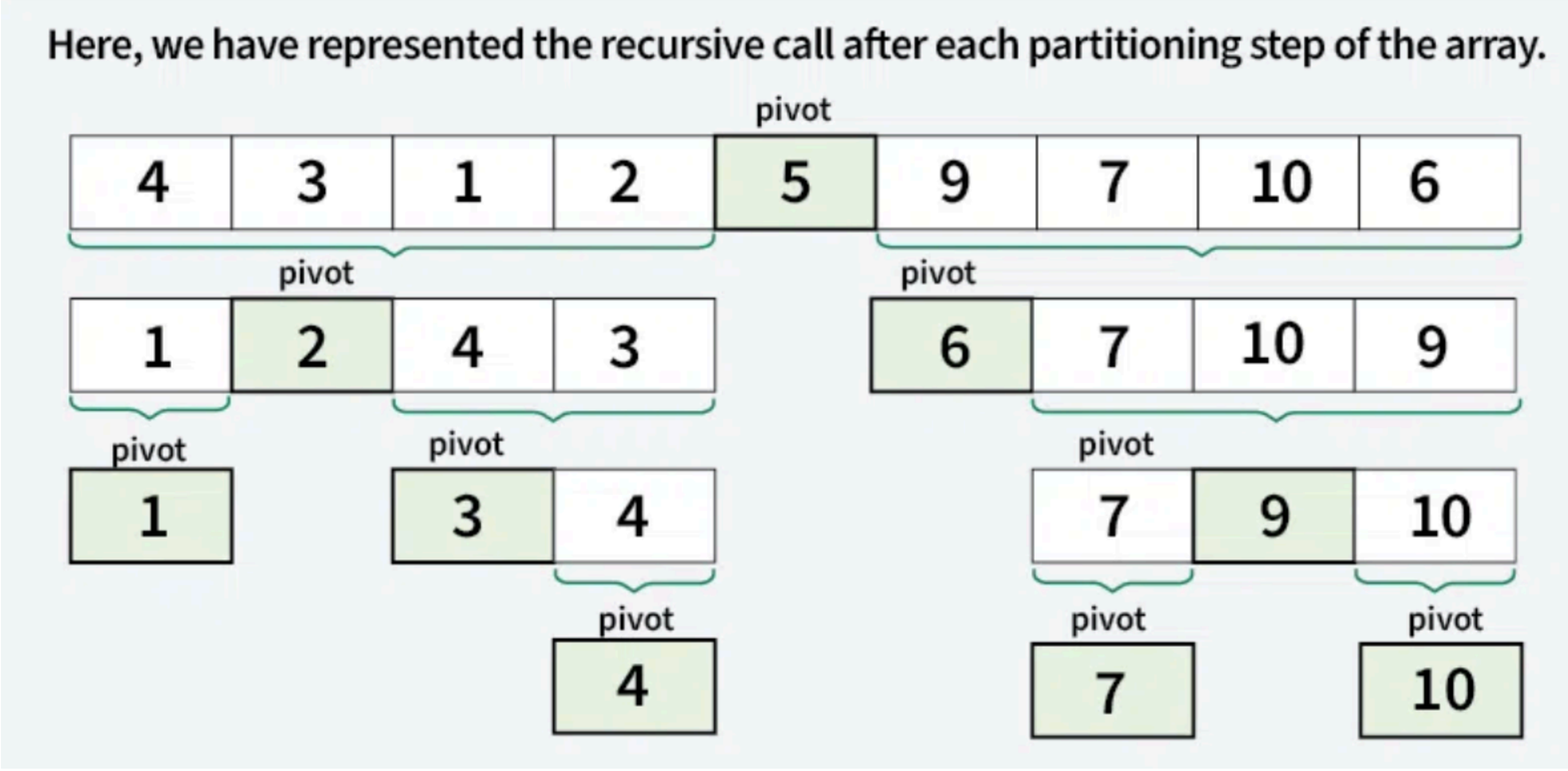
```
QUICKSORT(A[1..n], l, r)
1  if l < r then
2      k ← Aufteilen(A, l, r)        ▷ Teile A[l..r] in zwei Gruppen auf
3      Quicksort(A, l, k − 1)        ▷ Sortiere linke Gruppe
4      Quicksort(A, k + 1, r)        ▷ Sortiere rechte Gruppe
```

```
AUFTEILEN(A[1..n], l, r)
 1  p ← A[r]                          ▷ Pivotelement
 2  k ← Zahl der Elemente ≤ p in A[l..r]
 3  B ← neues Array mit r − l + 1 Zellen   ▷ so gross wie A[l, ..., r]
 4  B[k] ← p                          ▷ Pivot muss an k-te Stelle
 5  i ← l                             ▷ Anfang des linken Teils von B
 6  j ← k + 1                         ▷ Anfang des rechten Teils von B
 7  for s ← l, l + 1, ..., r
 8      if A[s] ≤ p then
 9          B[i] ← A[s]               ▷ Schreibe A[s] in linke Hälfte
10          i ← i + 1
11      else
12          B[j] ← A[s]               ▷ Schreibe A[s] in rechte Hälfte
13          j ← j + 1
14  kopiere B nach A[l..r]
```

Illustration

Quick Sort

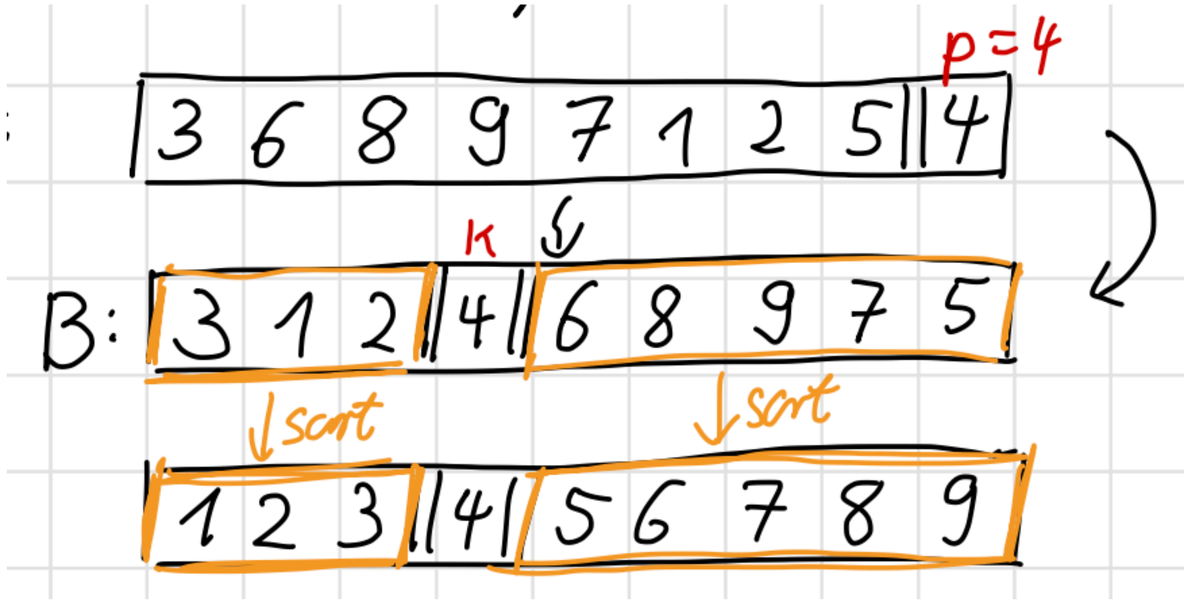
Illustration



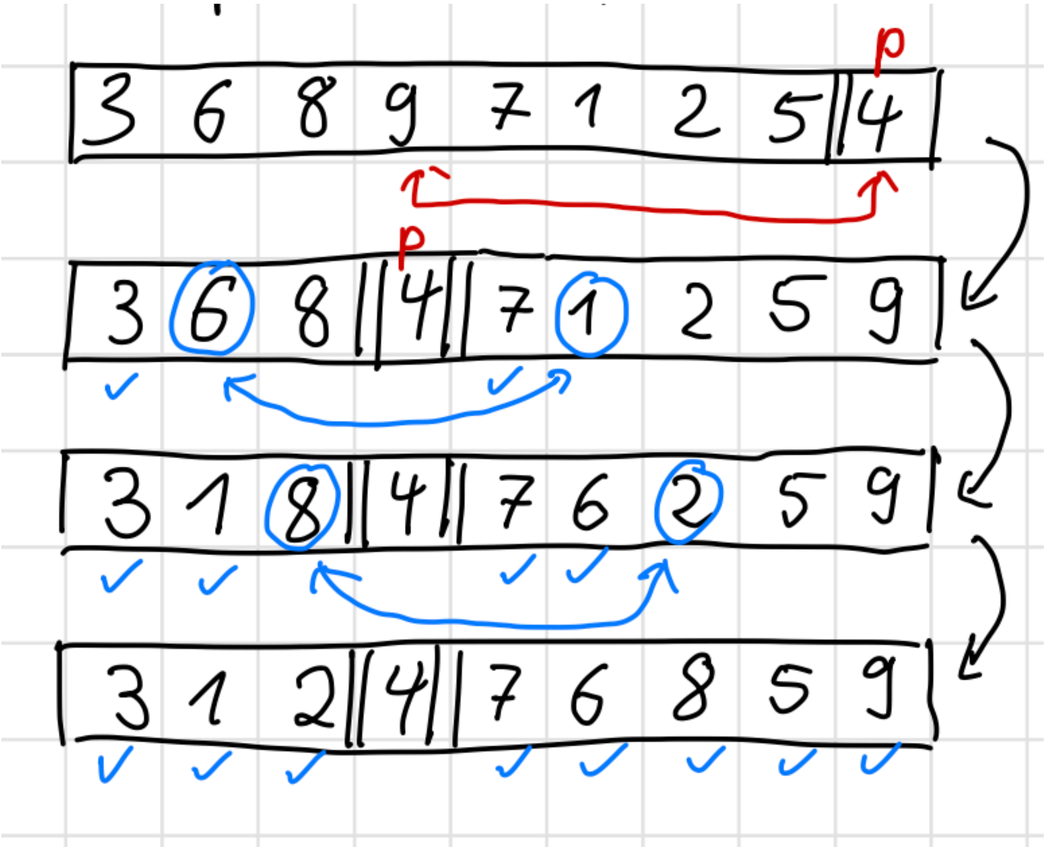
Quick Sort

Illustration

With helper array



In-place



Heap Sort

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

Input : unsorted array Output : sorted array

Runtime: O(nlogn)

Pseudocode :

HEAPSORT($A[1..n]$)

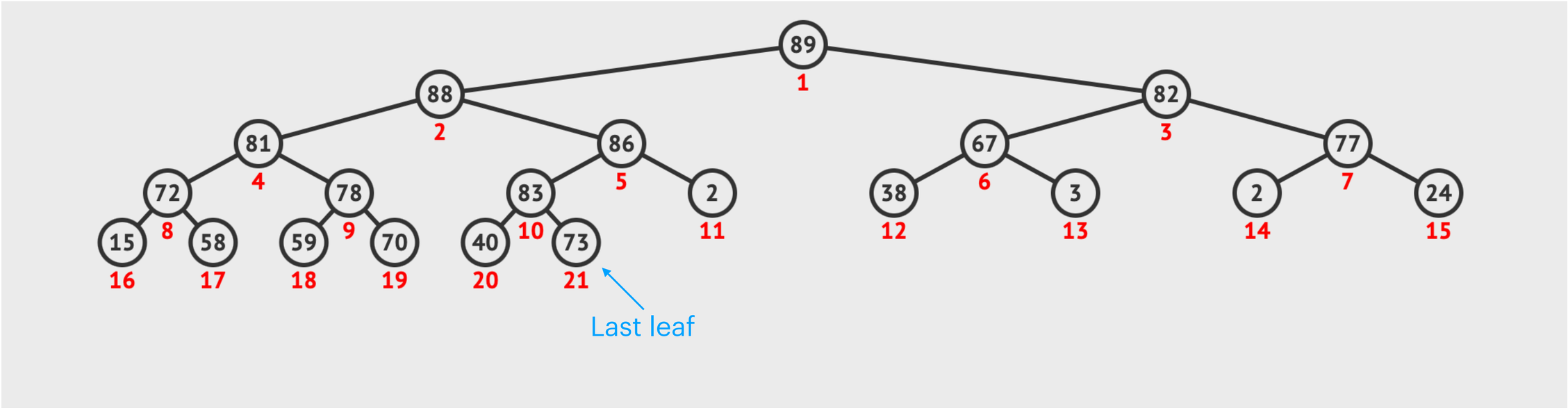
```
1  H ← Heapify(A)                    ▷ Wandle Array in Heap um.
2  for i ← n, n − 1, . . . , 1 do    ▷ Entferne Elemente aus Heap
3      A[i] ← ExtractMax(H)
```

Sorting Algorithms Kahoot

Let’s take a break

Data Structures I

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 (here : Maxheap)

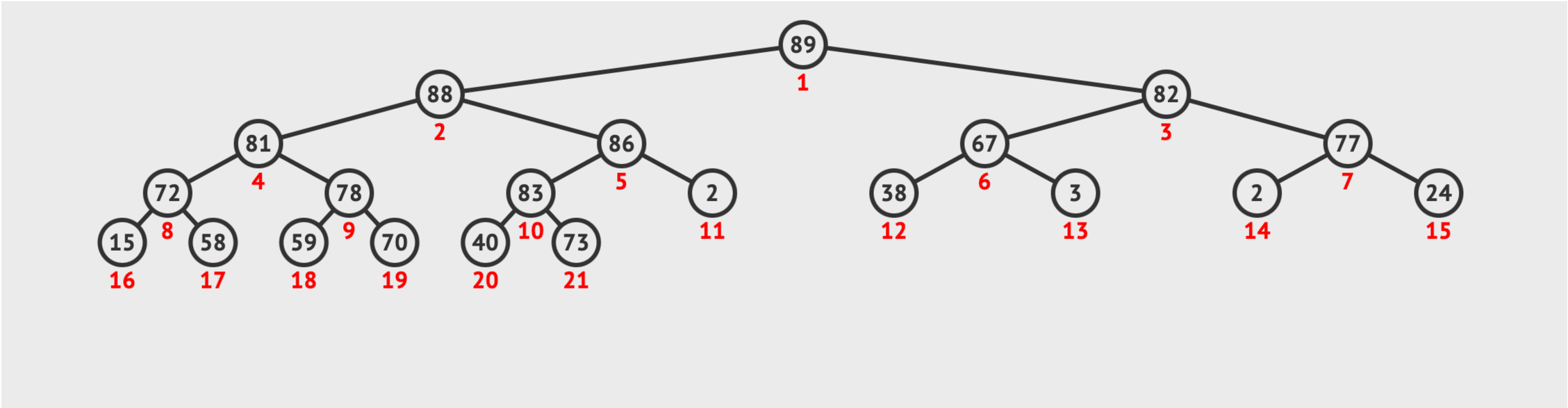
Heap Condition

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

$$val(n) \geq val(children(n) \text{ for all } n$$

Heap (here : Maxheap)

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



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 (here : Maxheap)

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

ExtractMax( )

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 !

Heap (here : Maxheap)

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

insert( )

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

Trees

Exam Tipps

- Know the tree condition , always keep in mind !

- Know how to insert, know how to delete
- Be able to illustrate an example by hand !

- Don’t mix up the trees !!!

- Is it min or max ?

Heap

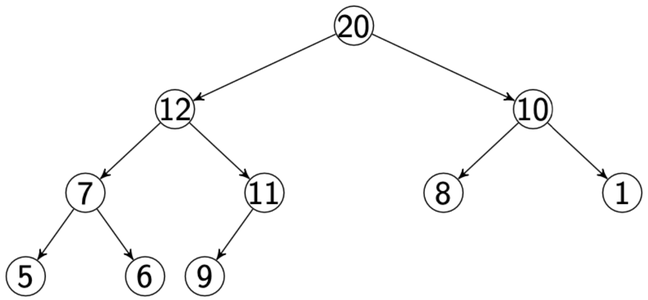
Exam Question (FS19)

/ 1 P

a) *Min-Heap*: Draw the Min-Heap that is obtained when inserting into an empty heap the keys `8`, `3`, `2`, `7`, `4`, `1` in this order.

/ 1 P

b) *Max-Heap*: Draw the resulting Max-Heap obtained from the following Max-Heap by performing the operation `DELETE-MAX` **twice**.

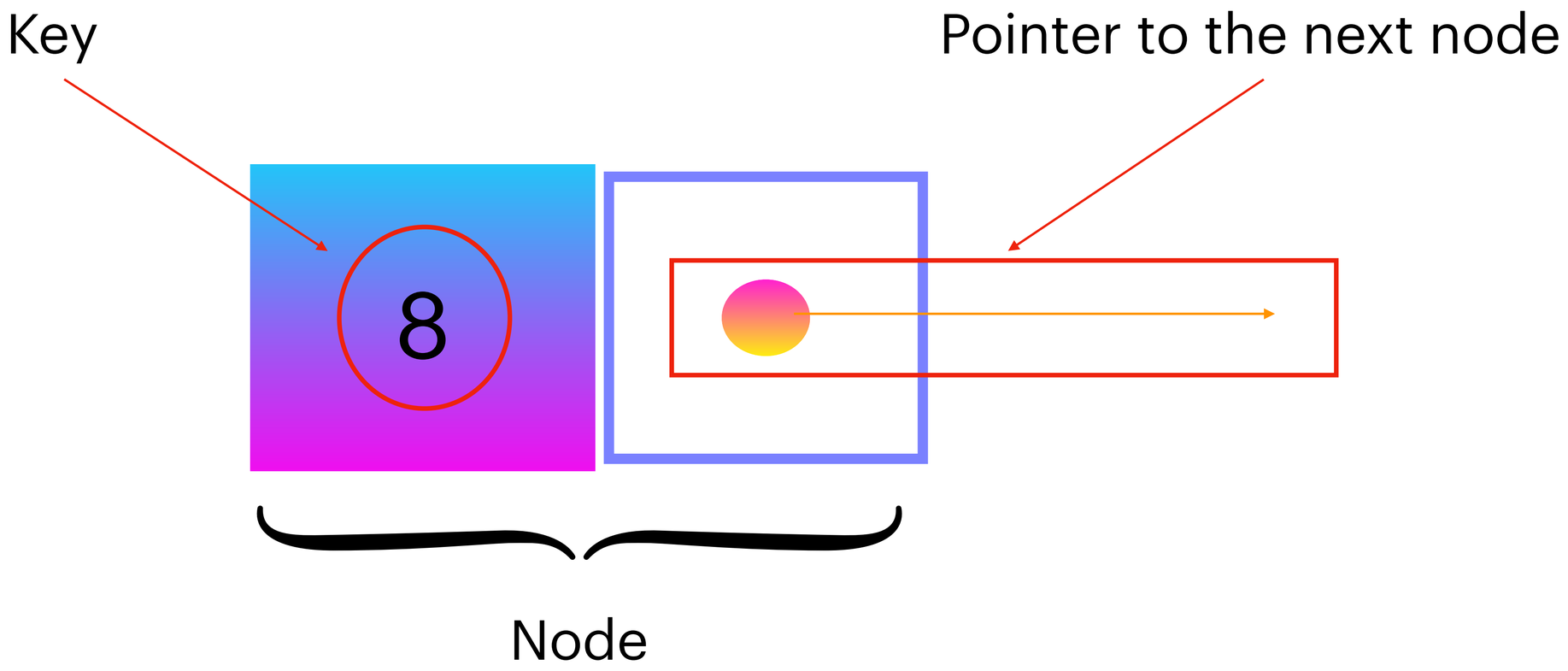


Abstract Data Types vs Data Structures

- List
- Stack
- Queue
- Priority Queue

- Array
- Linked List
- Doubly Linked List
- Heaps
- ...

Linked List



Linked List

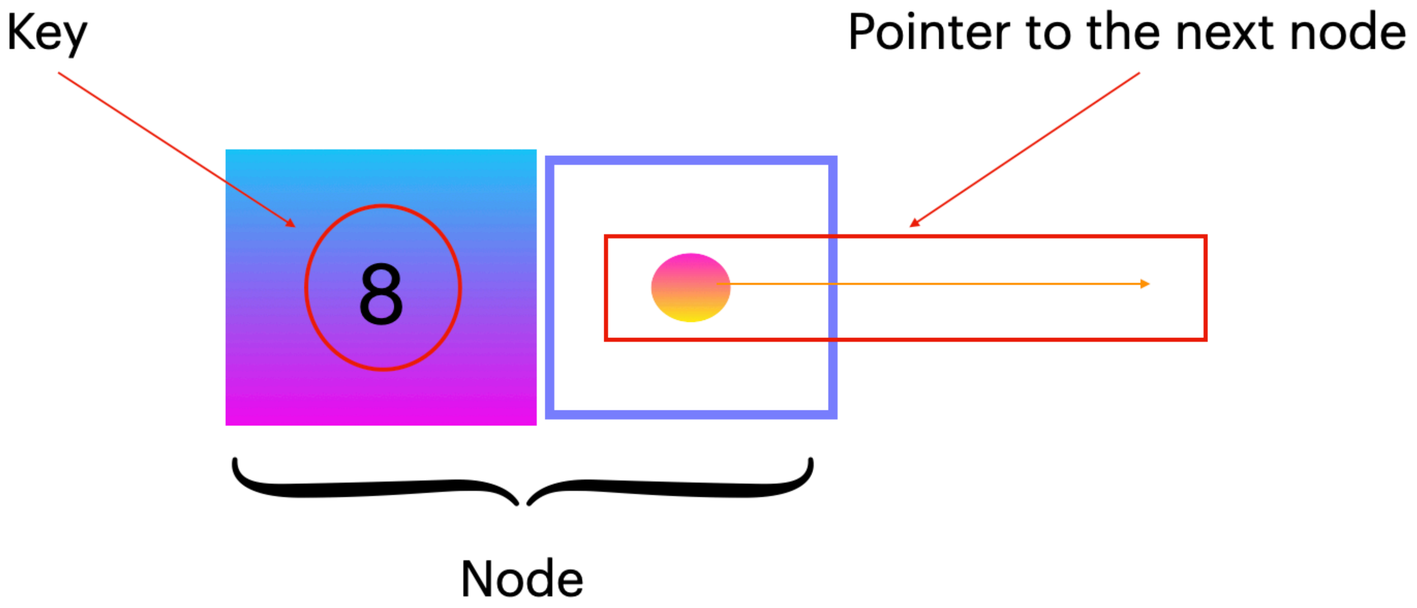
class LinkedList

Node start ;

class Node

int key ;

Node next ;



Linked List

class LinkedList

Node start ;

class Node

int key ;

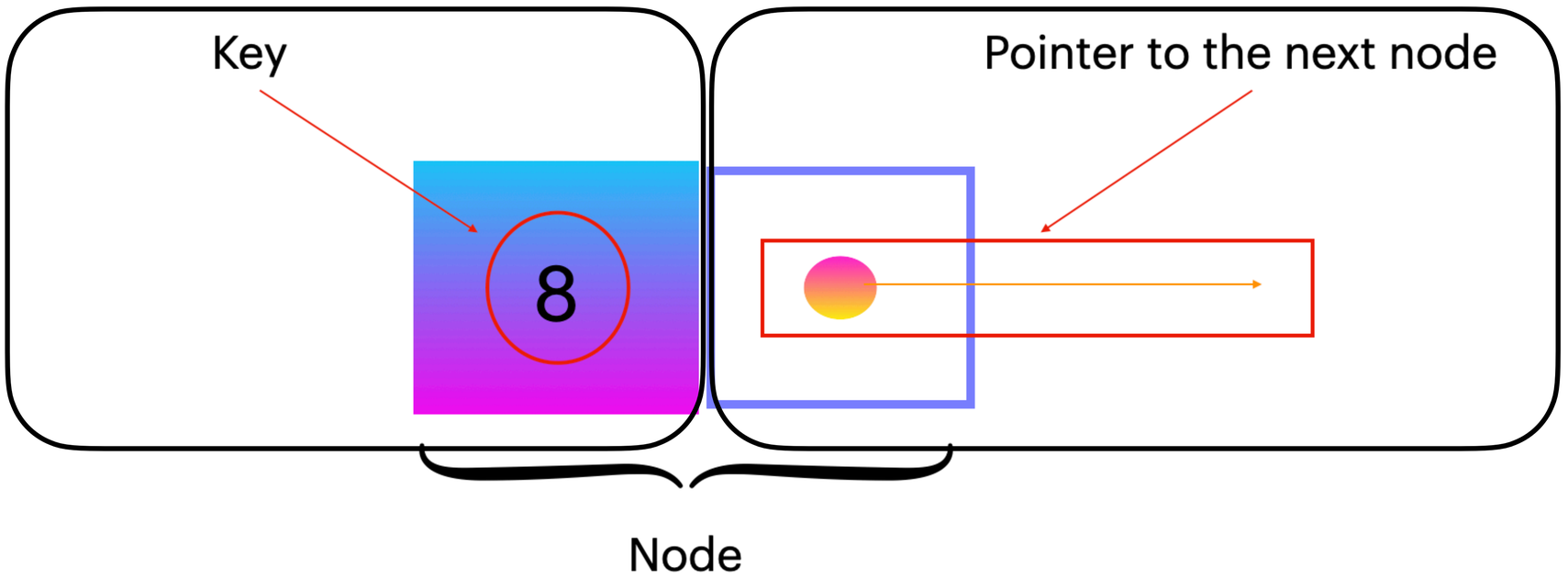
Node next ;

Key field

int key ;

Next field

Node next ;



Linked List

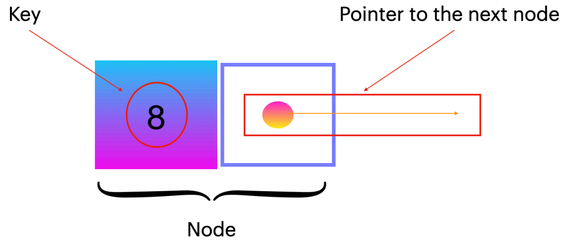
class LinkedList

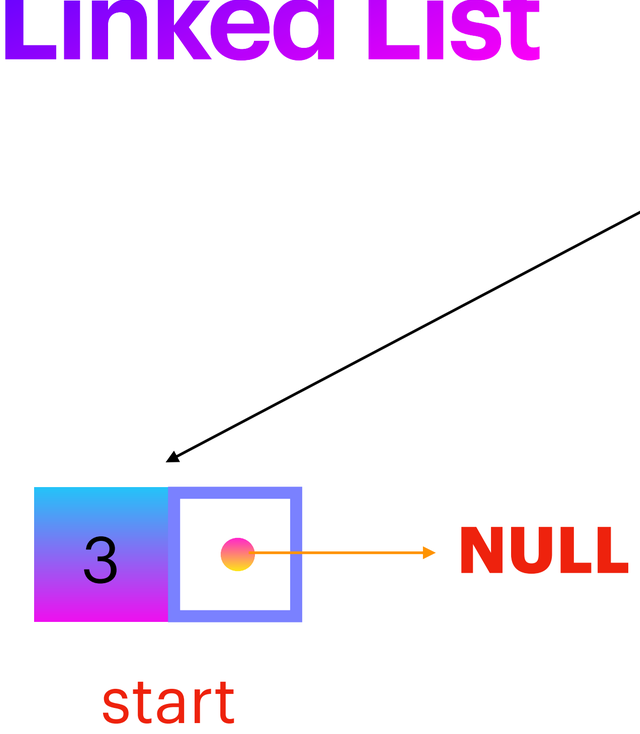
Node start ;

class Node

int key ;

Node next ;

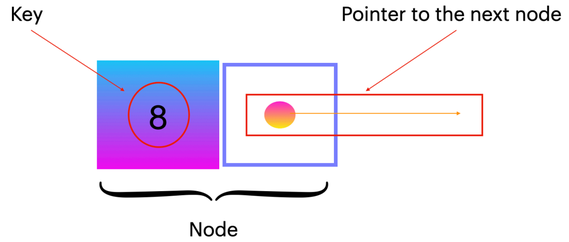


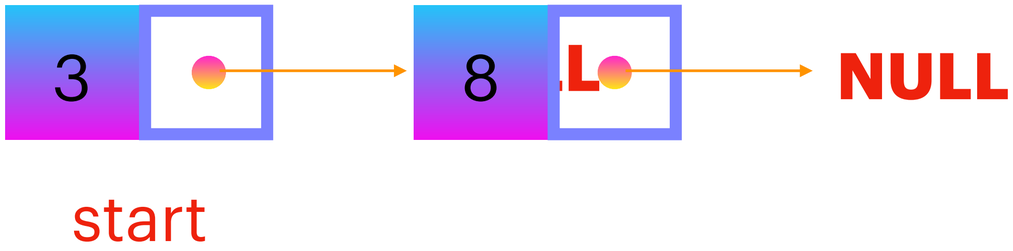


Linked List

```
class LinkedList
    Node start ;
```

```
class Node
    int key ;
    Node next ;
```



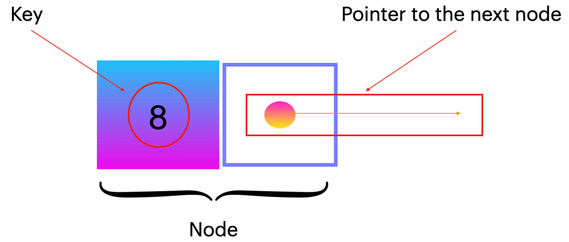


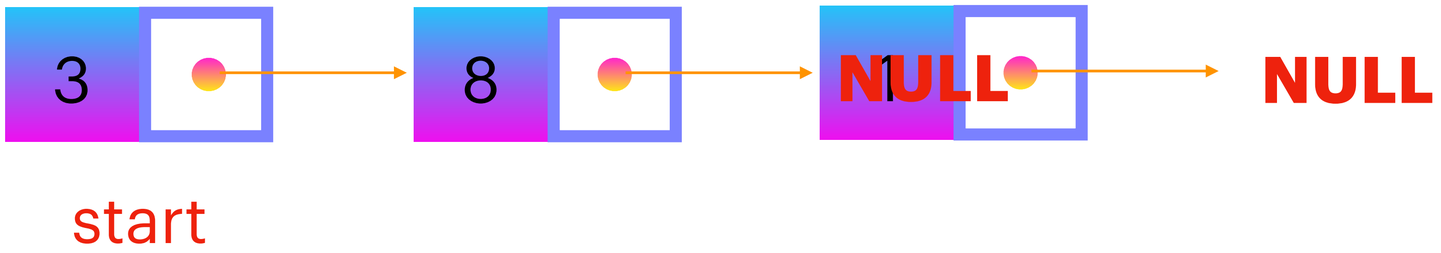
start

Linked List

```
class LinkedList
    Node start ;
```

```
class Node
    int key ;
    Node next ;
```

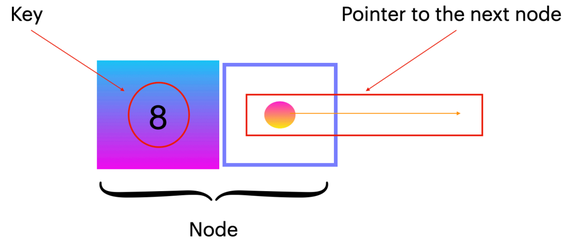


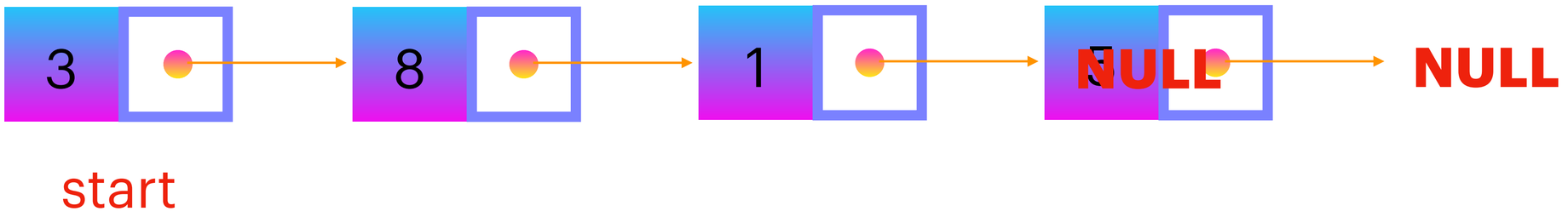


Linked List

```
class LinkedList
    Node start ;
```

```
class Node
    int key ;
    Node next ;
```

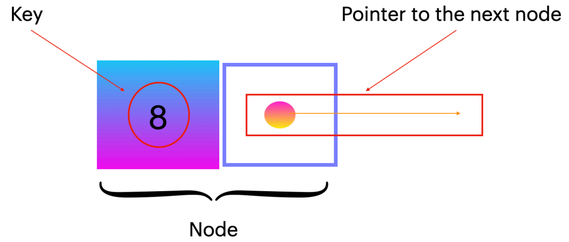


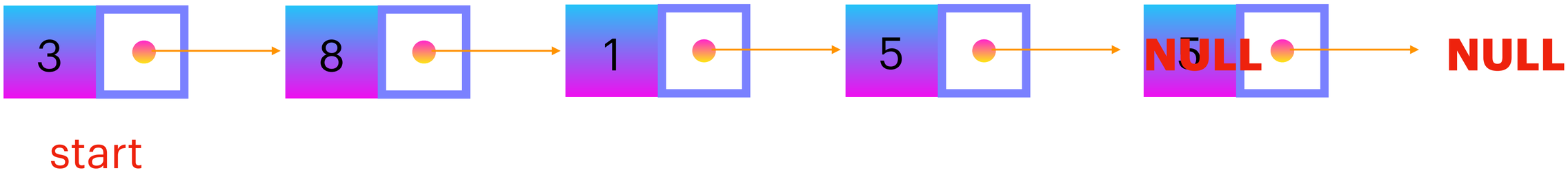


Linked List

```
class LinkedList
    Node start ;
```

```
class Node
    int key ;
    Node next ;
```



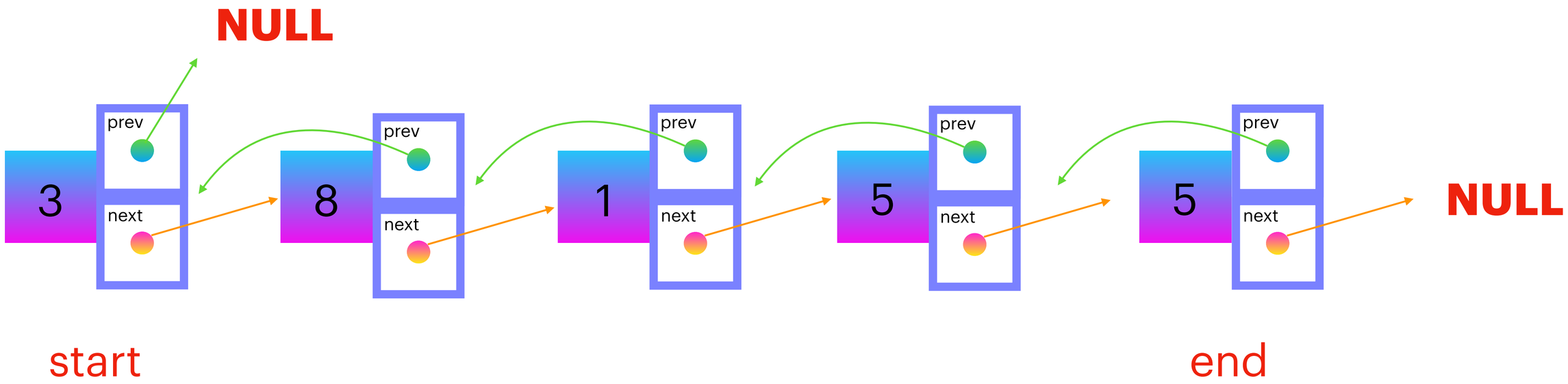


start

Doubly Linked List

```
class DoublyLinkedList
    Node start ;
    Node end ;
```

```
class Node
    int key ;
    Node next ;
    Node prev ;
```



Runtimes

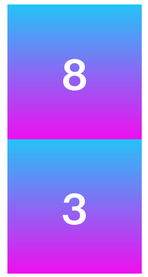
	Array	einf. verlinkte Liste	dopp. verlinkte Liste
`insert`(k, L)	$O(1)$	$O(1)$	$O(1)$
`get`(i, L)	$O(1)$	$O(\ell)$	$O(\ell)$
`insertAfter`(k, k', L)	$O(\ell)$	$O(1)$	$O(1)$
`delete`(k, L)	$O(\ell)$	$O(\ell)$	$O(1)$

Stack

push(3)

Stack

push(8)



Stack

push(1)

Stack

push(6)

Stack

pop( )

Queue

enqueue(3)

Queue

enqueue(1)

Queue

enqueue(8)

Queue

dequeue( )

Questions

Feedbacks , Recommendations

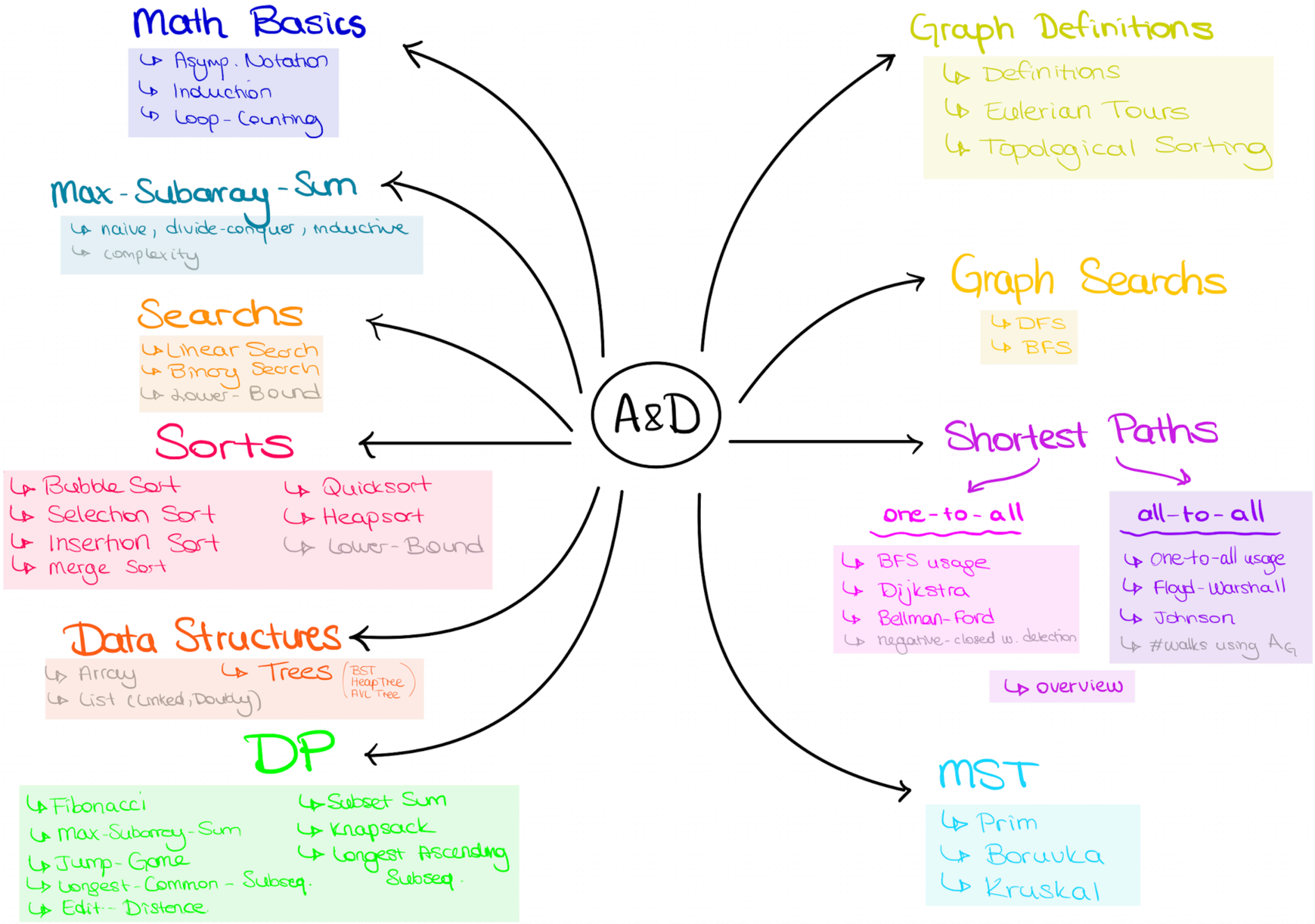
Nil Ozer

A&D

Exercise Session 6

Nil Ozer

A&D Overview


A&D
Math Basics
↳ 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

Outline

- Quiz
- Exercise Sheets

- Data Structures II

- DP I

Quiz

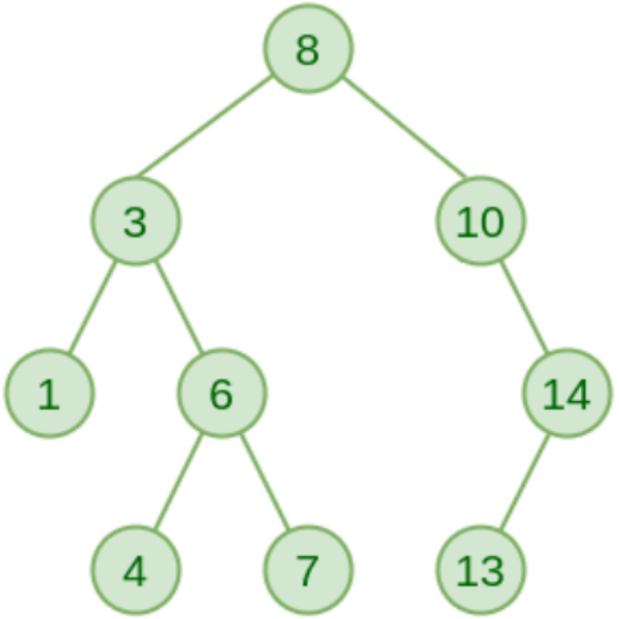
Exercise Sheets

- Exercise Sheet 4 bonus feedback left for next time
- Exercise Sheet 5 non-bonus questions left for next time

- Exercise Sheet 5 peergrading
  - 5.3 this week
  - Emails will be sent

Data Structures II

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).

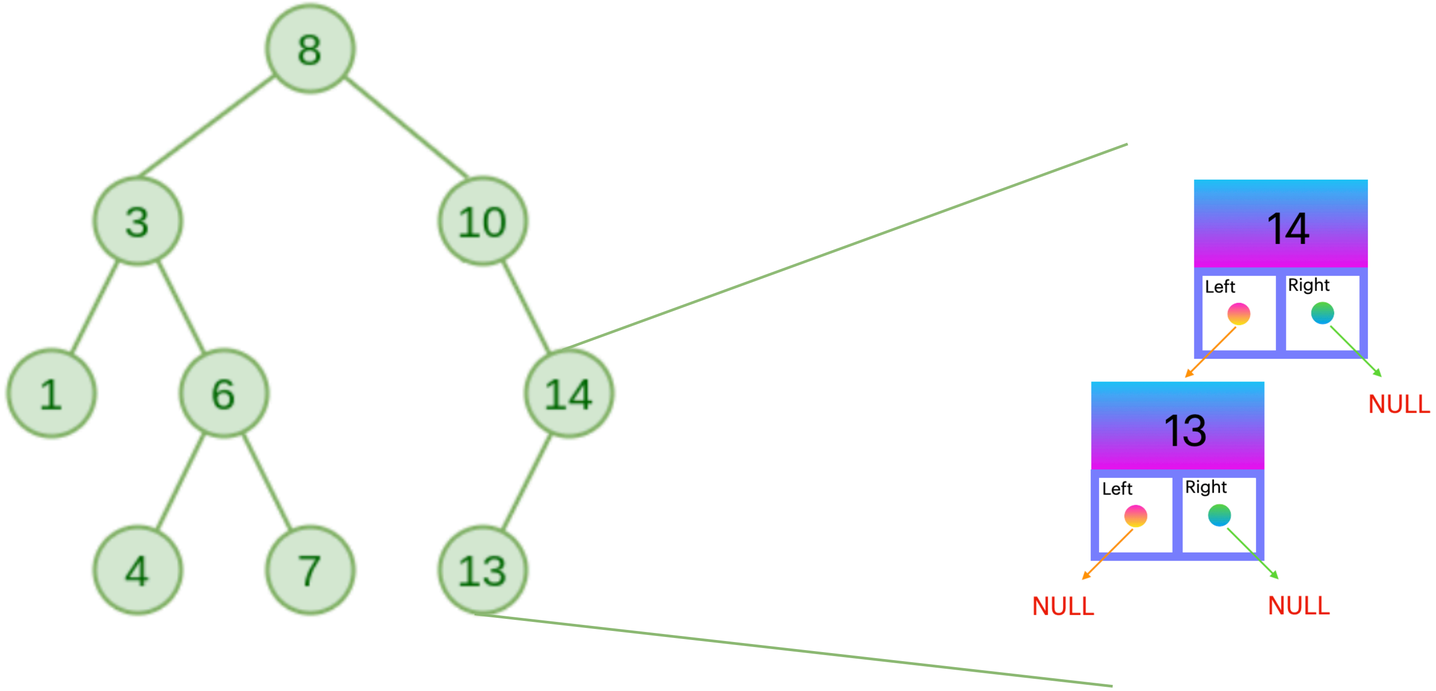
Sibling Nodes: Nodes that share the same parent.

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
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.

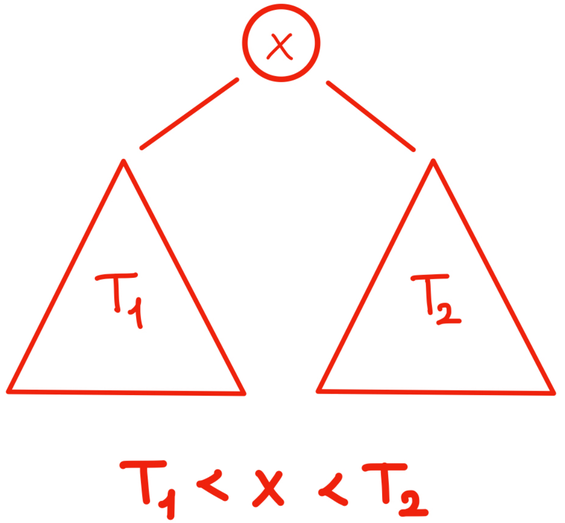
Height: The longest path from the root node to a leaf.

Height of the root = 1

BST

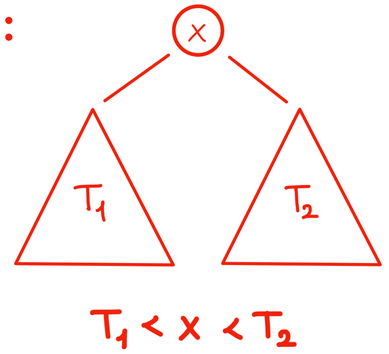
BST Condition

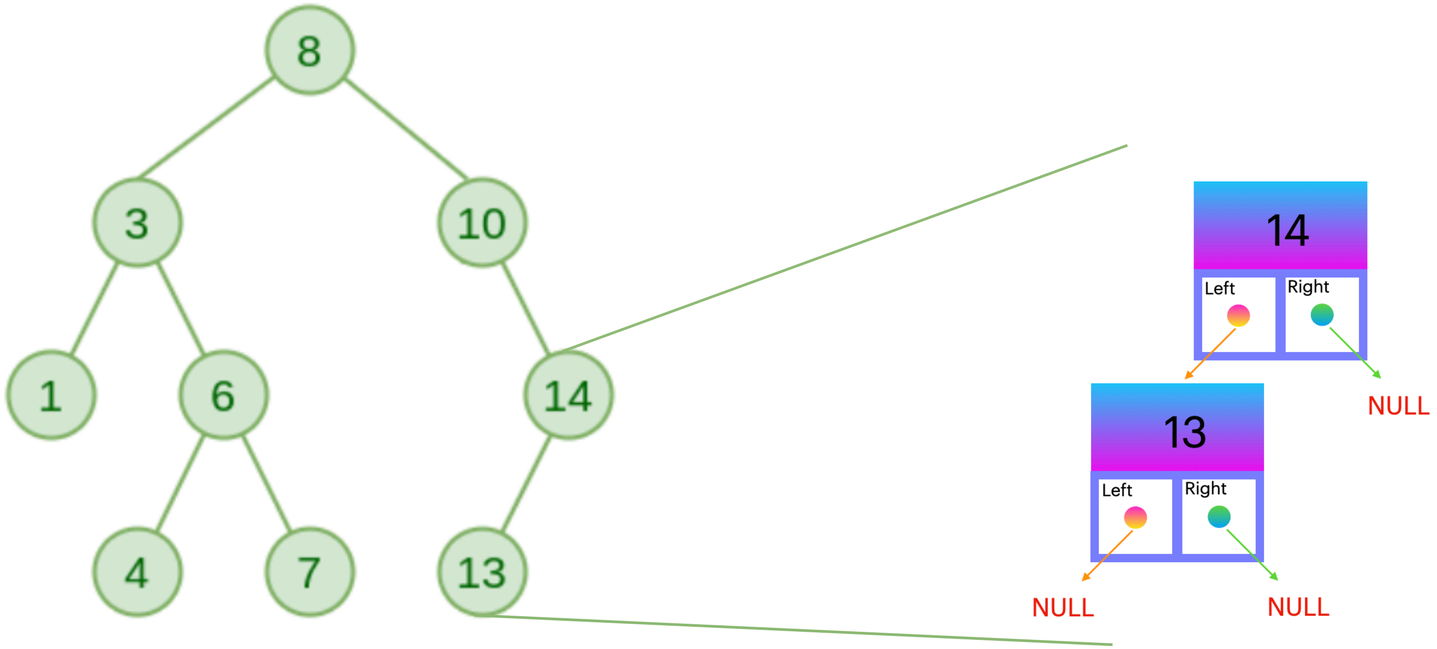
Each node in a BST has at most two children, left child and a right child, with the left child containing values the parent node and the right child containing values greater than the parent node. Every node in the left subtree is less than the root, and every node in the right subtree is greater than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



BST

BST Condition :





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.

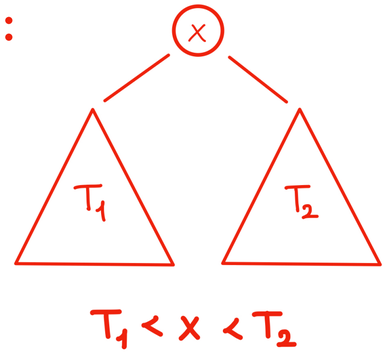
Height: The longest path from the root node to a leaf.

Height of the root = 1

BST

Search(x)

BST Condition :



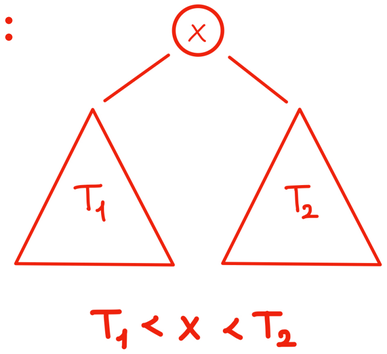
We use the BST condition

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

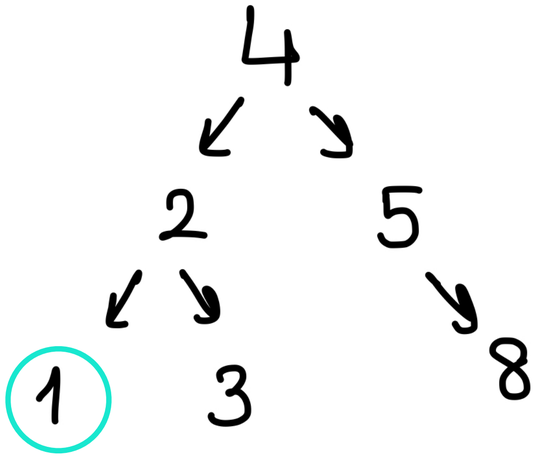
BST

remove(x)

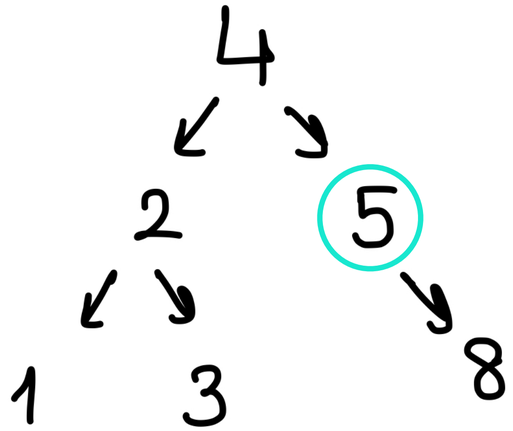
BST Condition :



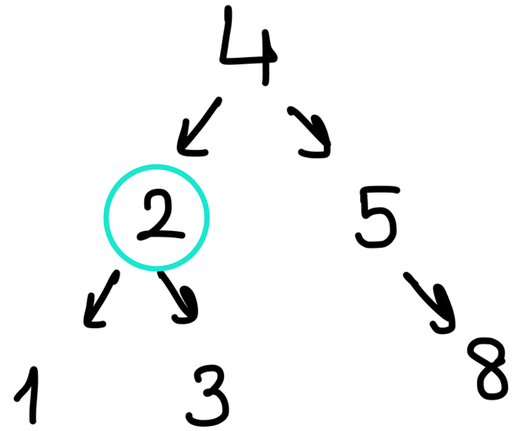
Case 1 : x has no children



Case 2: x has 1 child

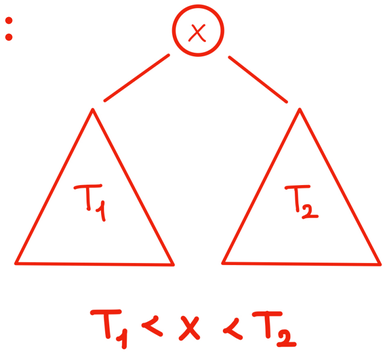


Case 3 : x has 2 children

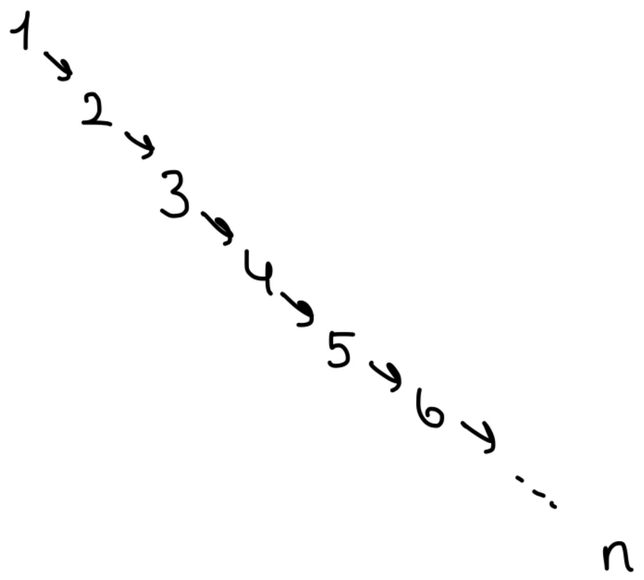


BST
Problem

BST Condition :



- Searching is in O(h)
- Our h doesn't have to be log(n). There's a case where h = n :

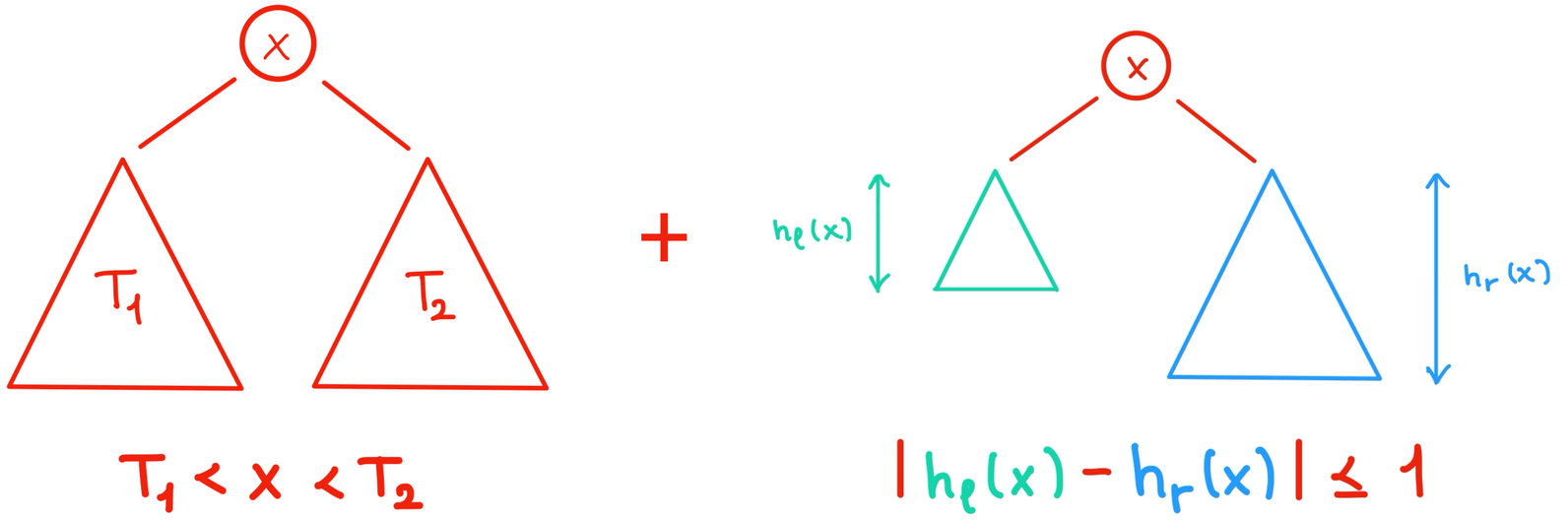


← Still a BST !!

AVL Tree

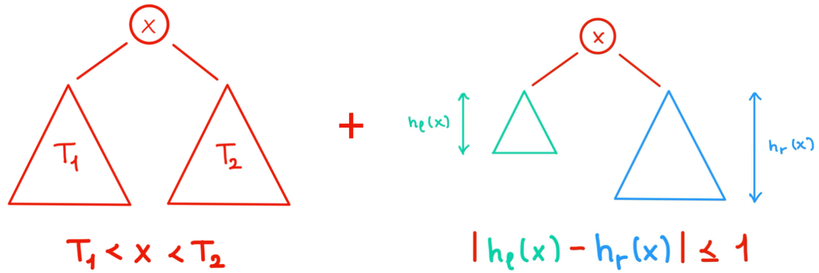
AVL Tree Condition

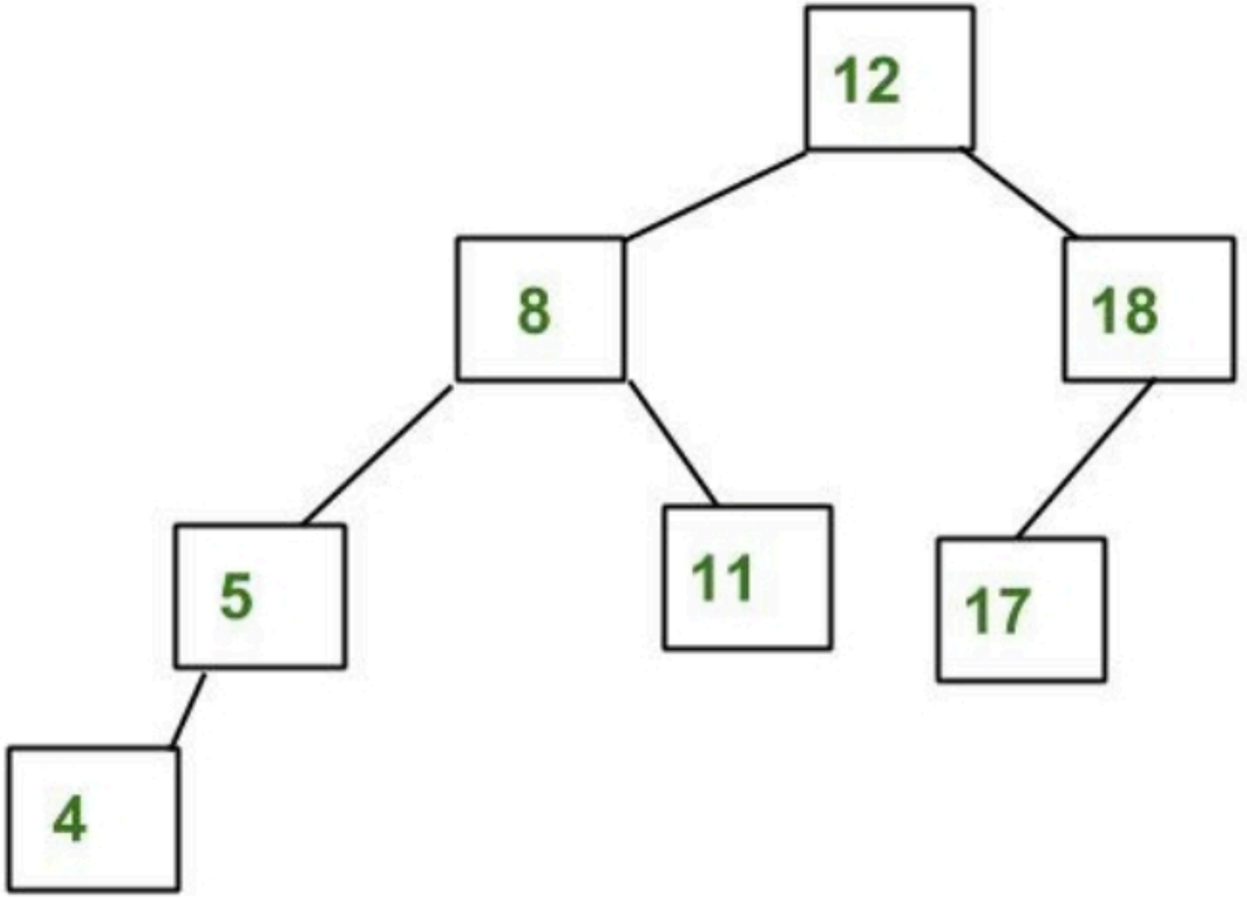
AVL tree is a self-balancing BST where the difference between heights of left and right subtrees cannot be more than one for all nodes.



AVL-Tree or not ?

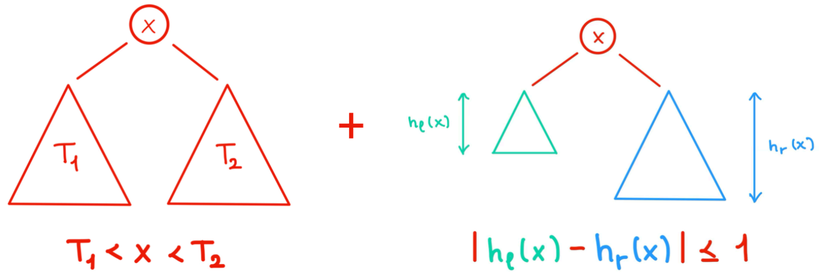
AVL-Tree Condition :

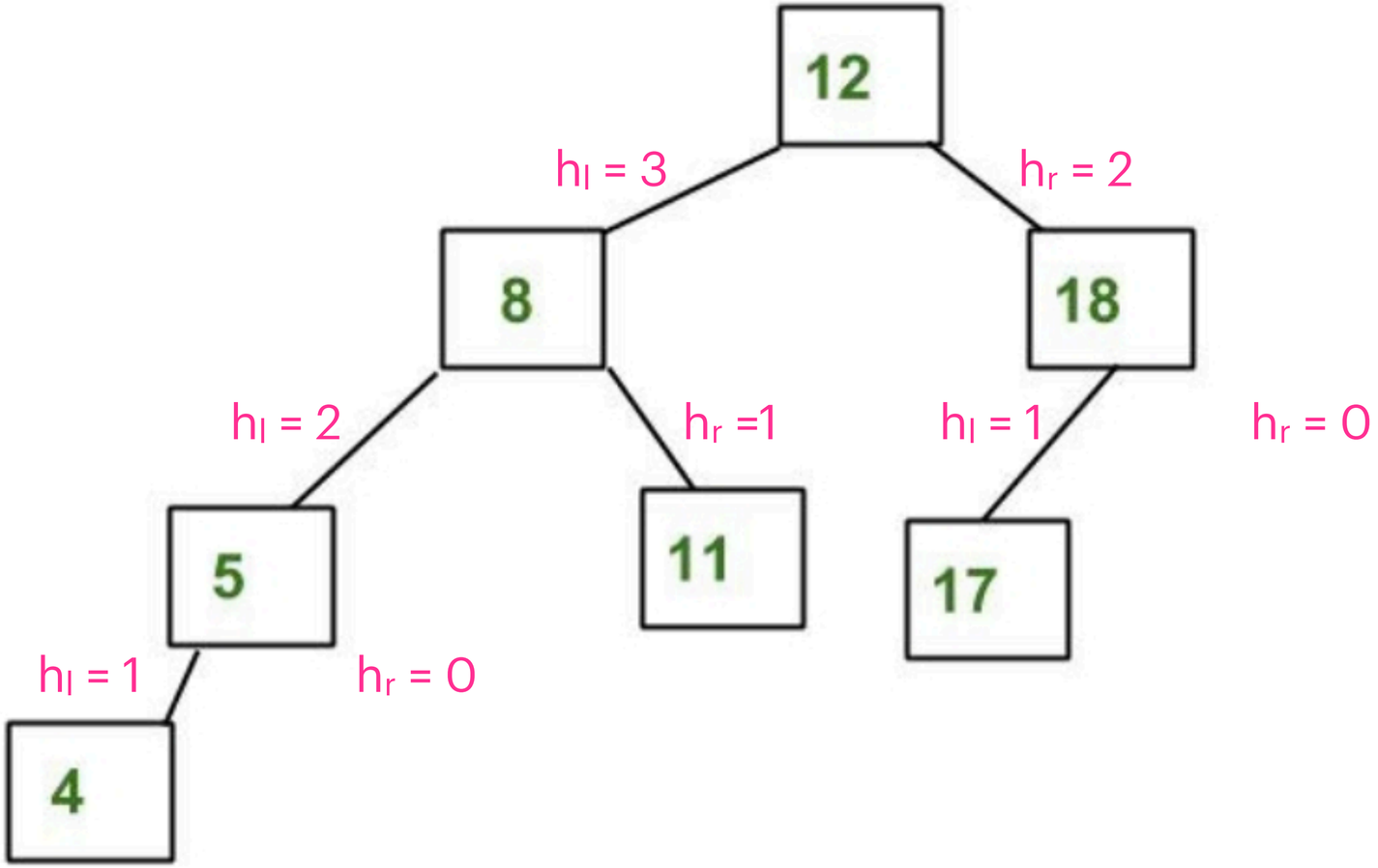




AVL-Tree or not ?

AVL-Tree Condition :

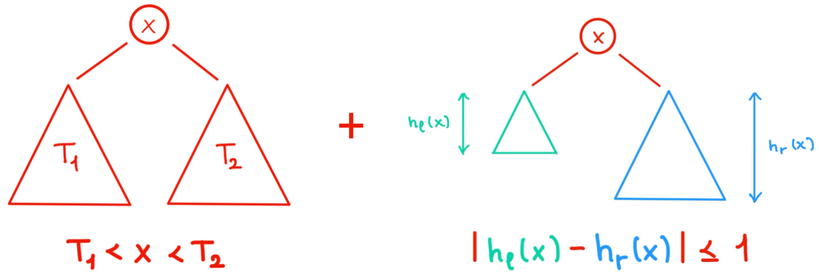


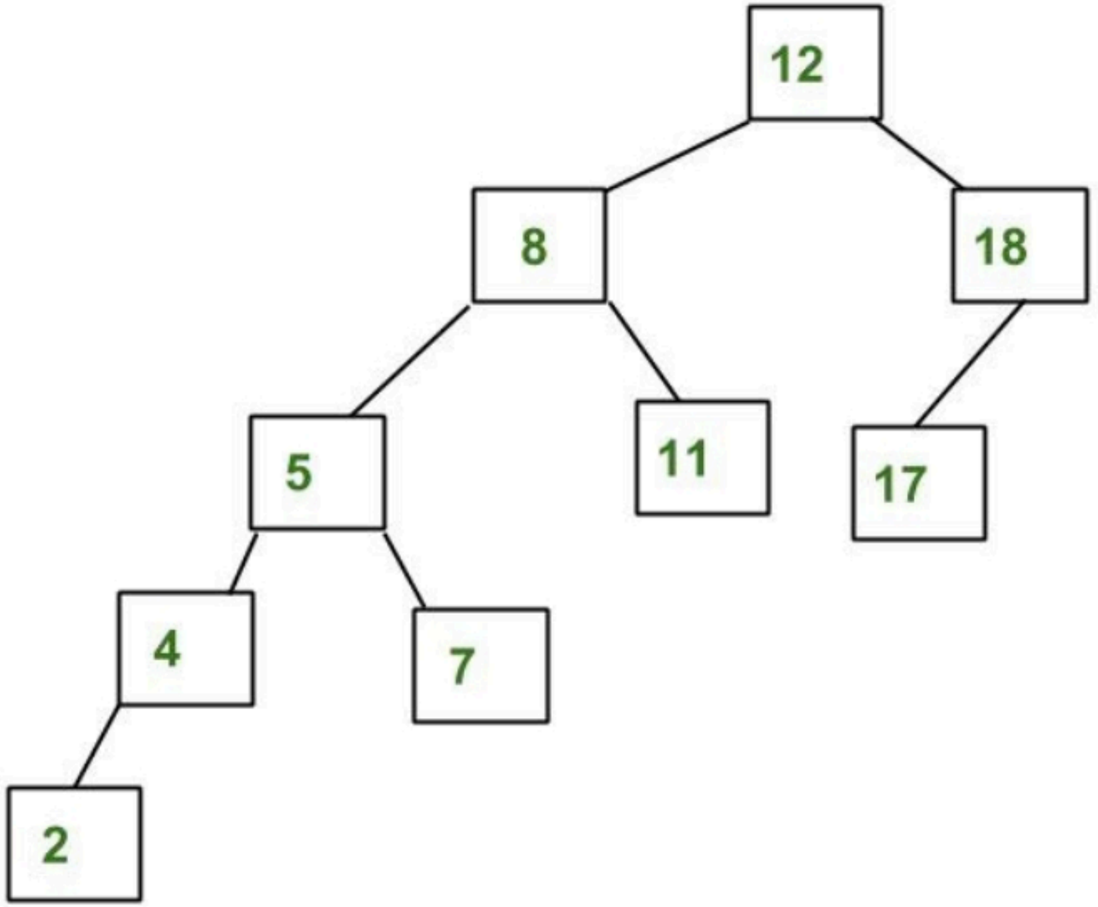


✅

AVL-Tree or not ?

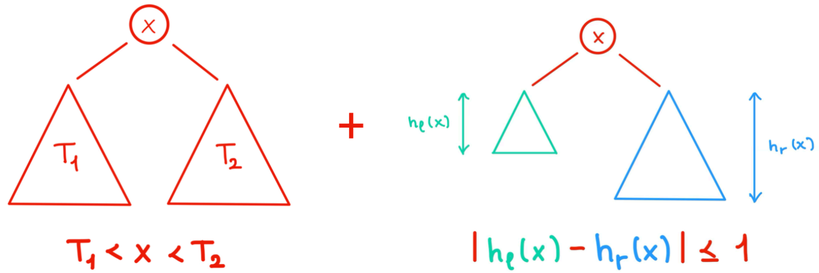
AVL-Tree Condition :

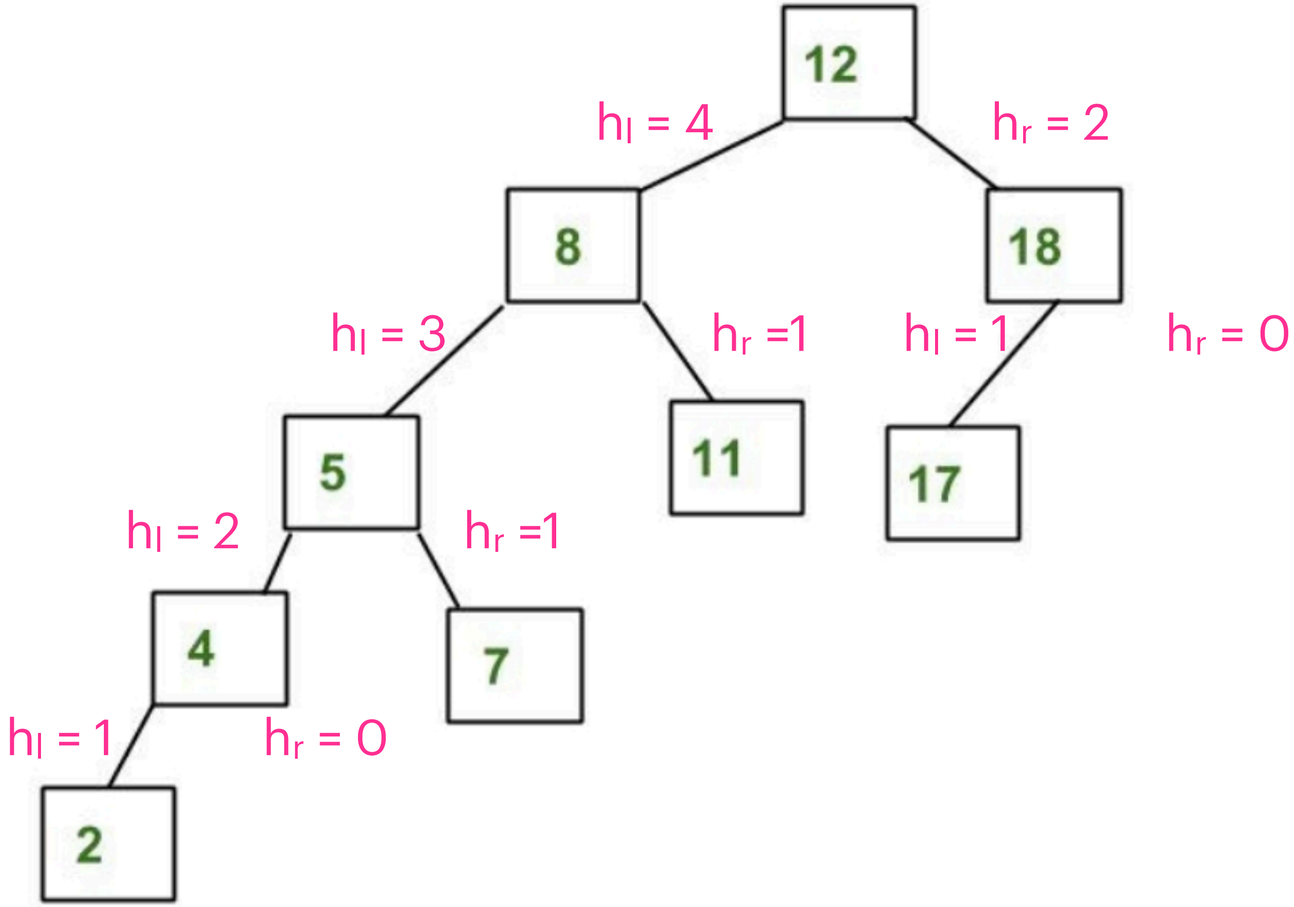




AVL-Tree or not ?

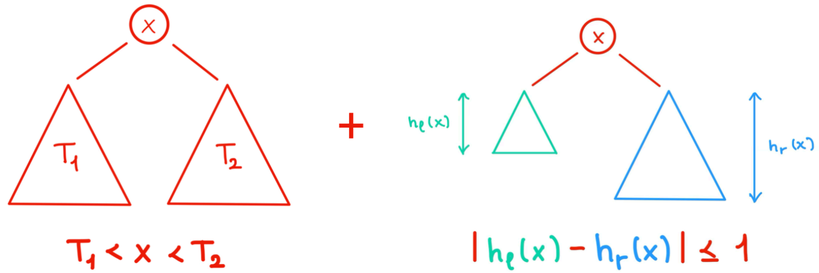
AVL-Tree Condition :

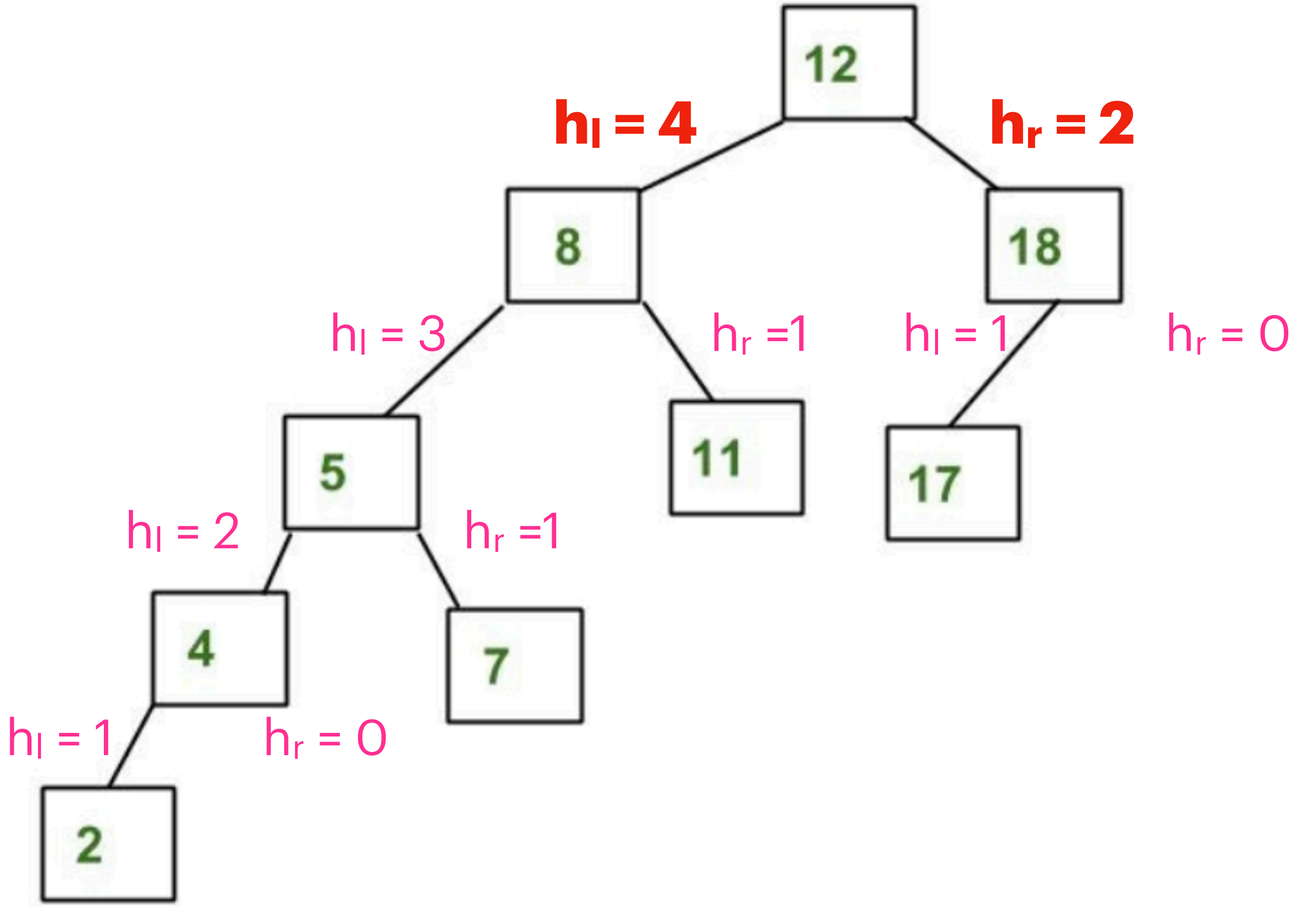




AVL-Tree or not ?

AVL-Tree Condition :





AVL Tree
Rebalancing

AVL-Tree Condition :

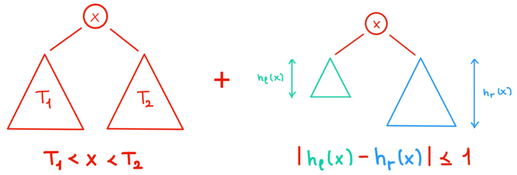
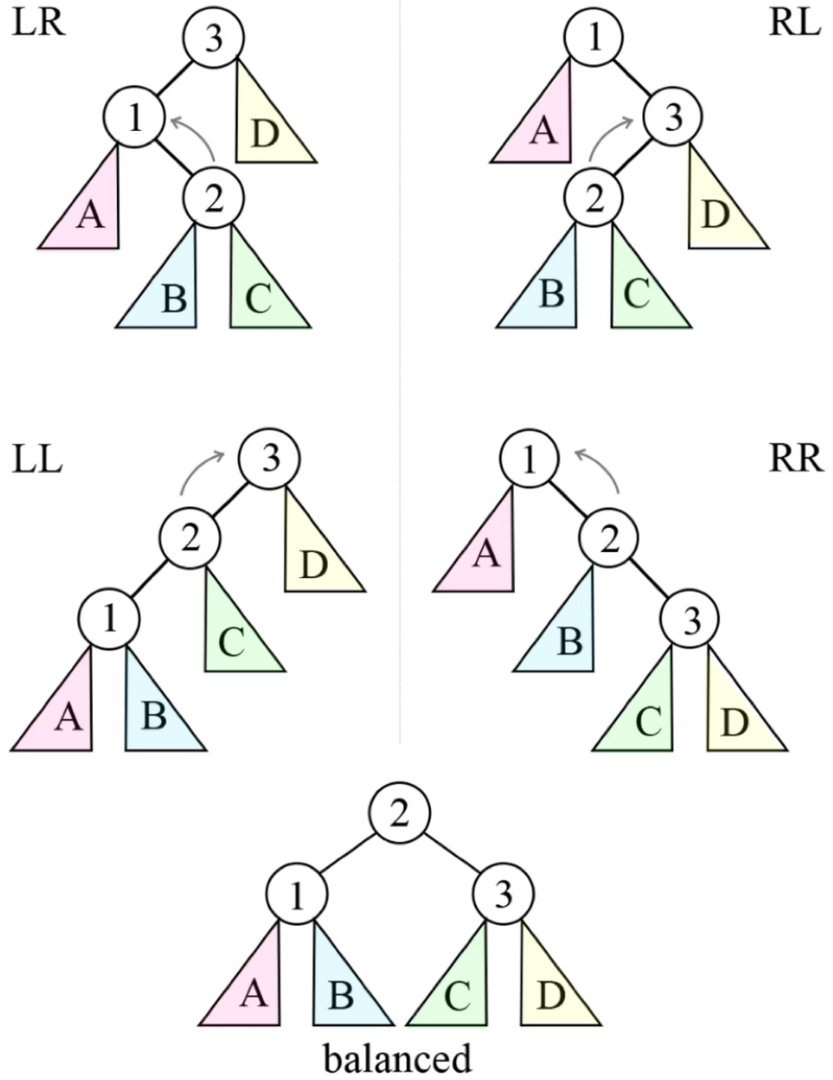
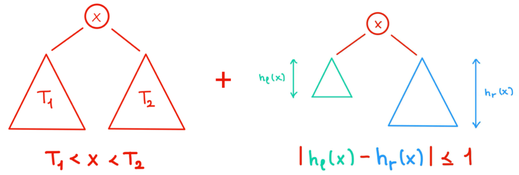


Figure 3.2: All possible rotations and their next state

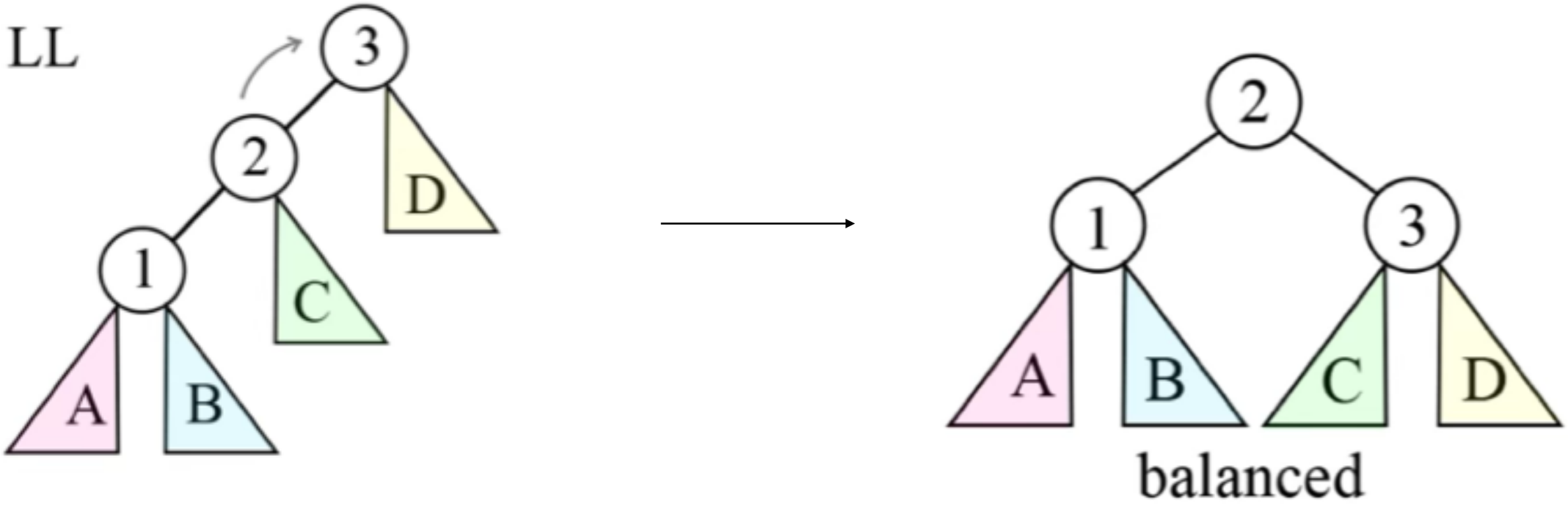


AVL-Tree Condition :



AVL Tree

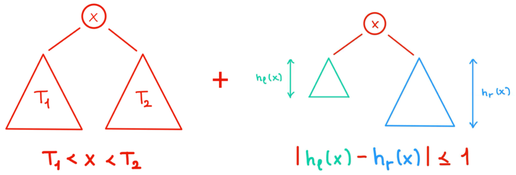
LL - Rotation

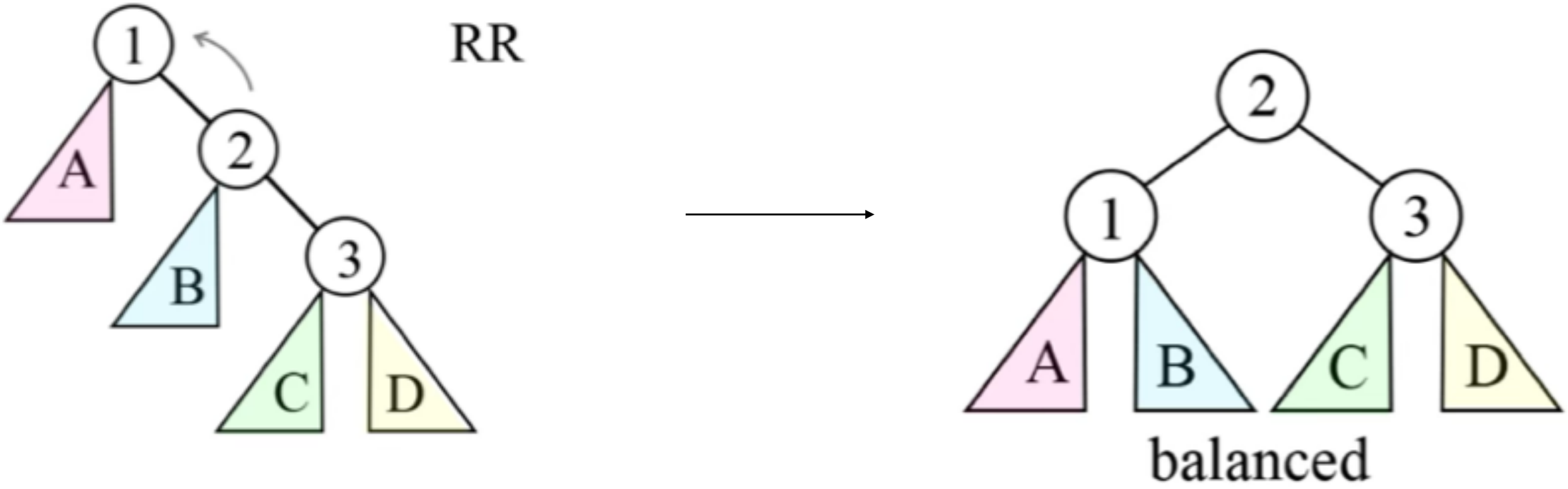


AVL Tree

RR - Rotation

AVL-Tree Condition :

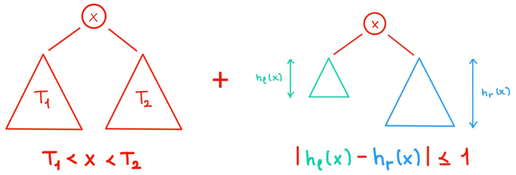


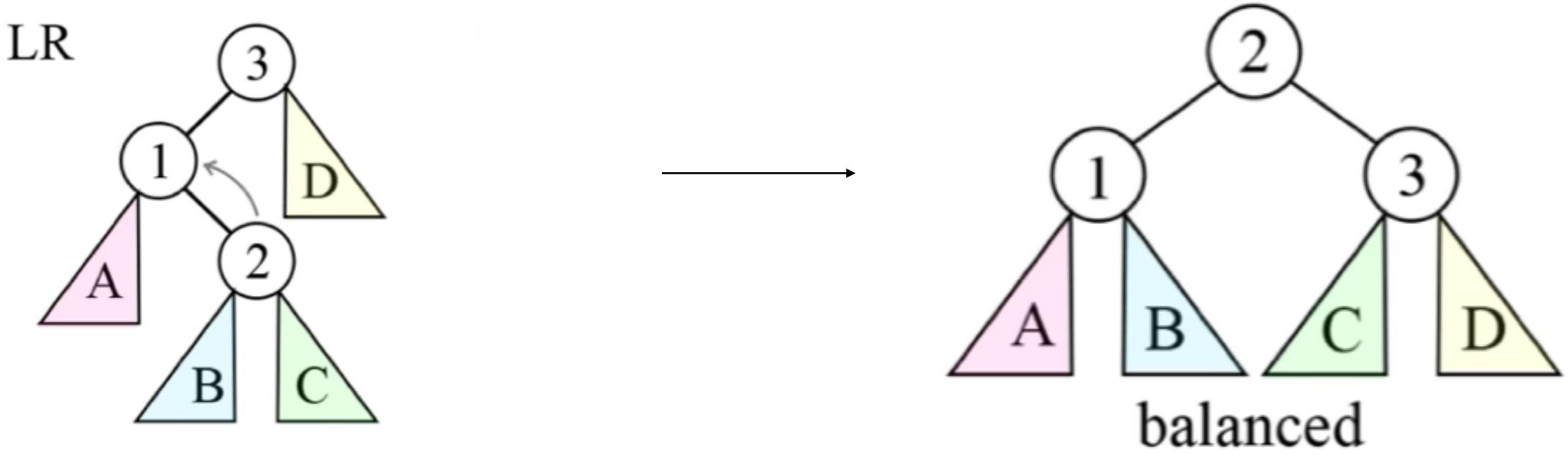


AVL Tree

LR - Rotation

AVL-Tree Condition :

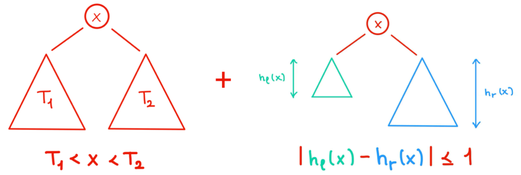


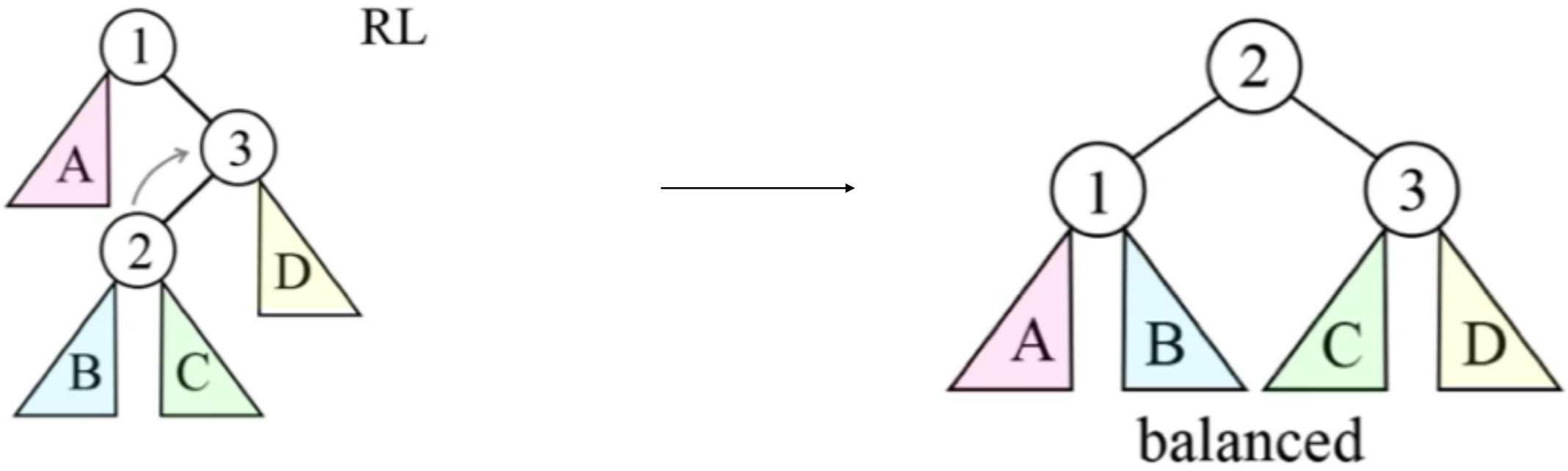


AVL Tree

RL - Rotation

AVL-Tree Condition :



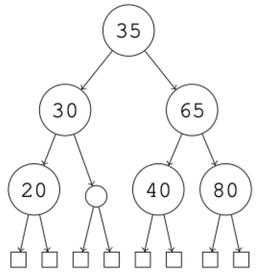


BST and AVL-Tree

Exam Question (FS23)

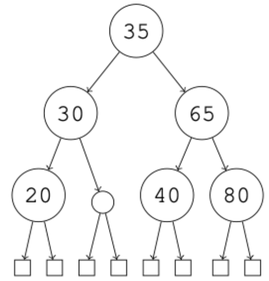
/ 3 P b) *Search trees:*

i) Draw the **binary search tree** obtained from the following tree by performing the two operations `INSERT`(45) and `DELETE`(30), in that order.



ii) Draw the **AVL tree** that is obtained when inserting the keys $3, 2, 7, 6, 8, 9$ in this order into an empty tree (it suffices to draw only the final tree).

iii) Draw the **AVL tree** that is obtained by deleting key 65 from the tree below.



Trees

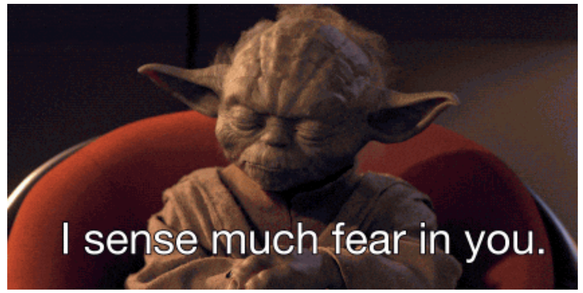
Exam Tipps

- Know the tree condition , always keep in mind !

- Know how to insert, know how to delete
- Be able to illustrate an example by hand !

- Don’t mix up the trees !!!

Let’s take a break



DP

DP

How to learn

- Theory, written tasks :
  - Exam questions T3 !!
  - Exercise sheets
  - geeksforgeeks

Always a combination of the ideas dicussed in lecture !

- Coding :
  - CodeEx exercises , my videos
  - Old Exam exercises
  - Leetcode https://leetcode.com/studyplan/dynamic-programming/

Table ? 🤦‍♀️

DP

Written Question Format

a) Provide a *dynamic programming* algorithm that, given an array A of n pairwise distinct positive integers as above, returns `True` if there are two different (non-empty) subsets I_1 and I_2 of $\{1, \ldots, n\}$ (i.e. $\emptyset \neq I_1, I_2 \subseteq \{1, \ldots, n\}$ and $I_1 \neq I_2$) such that $\sum_{i \in I_1} a_i = \sum_{i \in I_2} a_i$ and `False` otherwise. In particular, the algorithm does *not* have to return the sets I_1 and I_2.

For example,

- For the array $[2, 3, 4, 5, 7]$ the output should be `True` since for $I_1 = \{1, 2, 4\}$ and $I_2 = \{2, 5\}$ we have $\sum_{i \in I_1} a_i = 2 + 3 + 5 = 10 = 3 + 7 = \sum_{i \in I_2} a_i$.
- For the array $[2, 3, 4, 10, 20]$ the output should be `False` since there are no two different non-empty subsets I_1 and I_2 of $\{1, 2, 3, 4, 5\}$ that satisfy $\sum_{i \in I_1} a_i = \sum_{i \in I_2} a_i$.

In order to obtain full points, your algorithm should run in time $O(n \cdot S)$, where $S = \sum_{i=1}^{n} a_i$. In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

DP

Introduction Exercise

Consider the recurrence

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

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?

DP

How to learn

- Theory, written tasks :
  - Exam questions T3 !!
  - Exercise sheets
  - geeksforgeeks

Always a combination of the ideas dicussed in lecture !

- Coding :
  - CodeEx exercises , my videos
  - Old Exam exercises
  - Leetcode https://leetcode.com/studyplan/dynamic-programming/

Table ? 🤦‍♀️

Maximum Subarray Sum

Problem : find the subarray that has the maximum sum

Examples :	Inputs :	Outputs :	Subarray :
	[2, 3, -8, 7, -1, 2, 3]	11	[7, -1, 2, 3]
	[-2, -4]	-2	[-2]
	[5, 4, 1, 7, 8]	25	[5, 4, 1, 7, 8]

Maximum Subarray Sum

Problem : find the subarray that has the maximum sum

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

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

Computation of an entry :

Initialization : DP[0] = A[0]

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

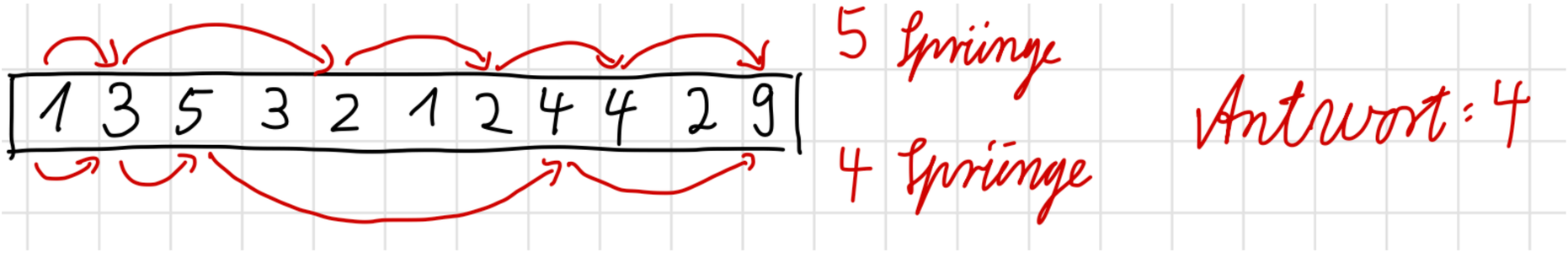
New subarray starting with a_i Adding a_i to the current subarray

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

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.

Example:



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

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

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.

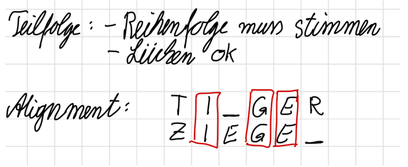
Examples :	Inputs :	Outputs :	Subsequence :
	“ABC" and “ACD”	2	“AC”
	‘AGGTAB” and “GXTXAYB”	4	“GTAB”
	“ABC” and “CBA”	1	“A” , “B” or “C”

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]

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 after or before any index of A

Remove : Remove a character of A

Examples :	Inputs :	Outputs :	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
- something gets inserted afterwards
- 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]

Computation of an entry :

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

Recursion :

$$DP[i][j] = \begin{cases} \min\{ DP[i-1][j] + 1 \text{ (delete A[i])}, DP[i][j-1] + 1 \text{ (add B[j] to the end)}, DP[i-1][j-1] \} & \text{if } A[i] == B[j] \\ \min\{ DP[i-1][j] + 1, DP[i][j-1] + 1, DP[i-1][j-1] + 1 \text{ (replace A[i] with B[j])} \} & \text{else} \end{cases}$$

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

CodeExpert

Edit Distance to Subsequence

Next Week

Questions

Feedbacks , Recommendations

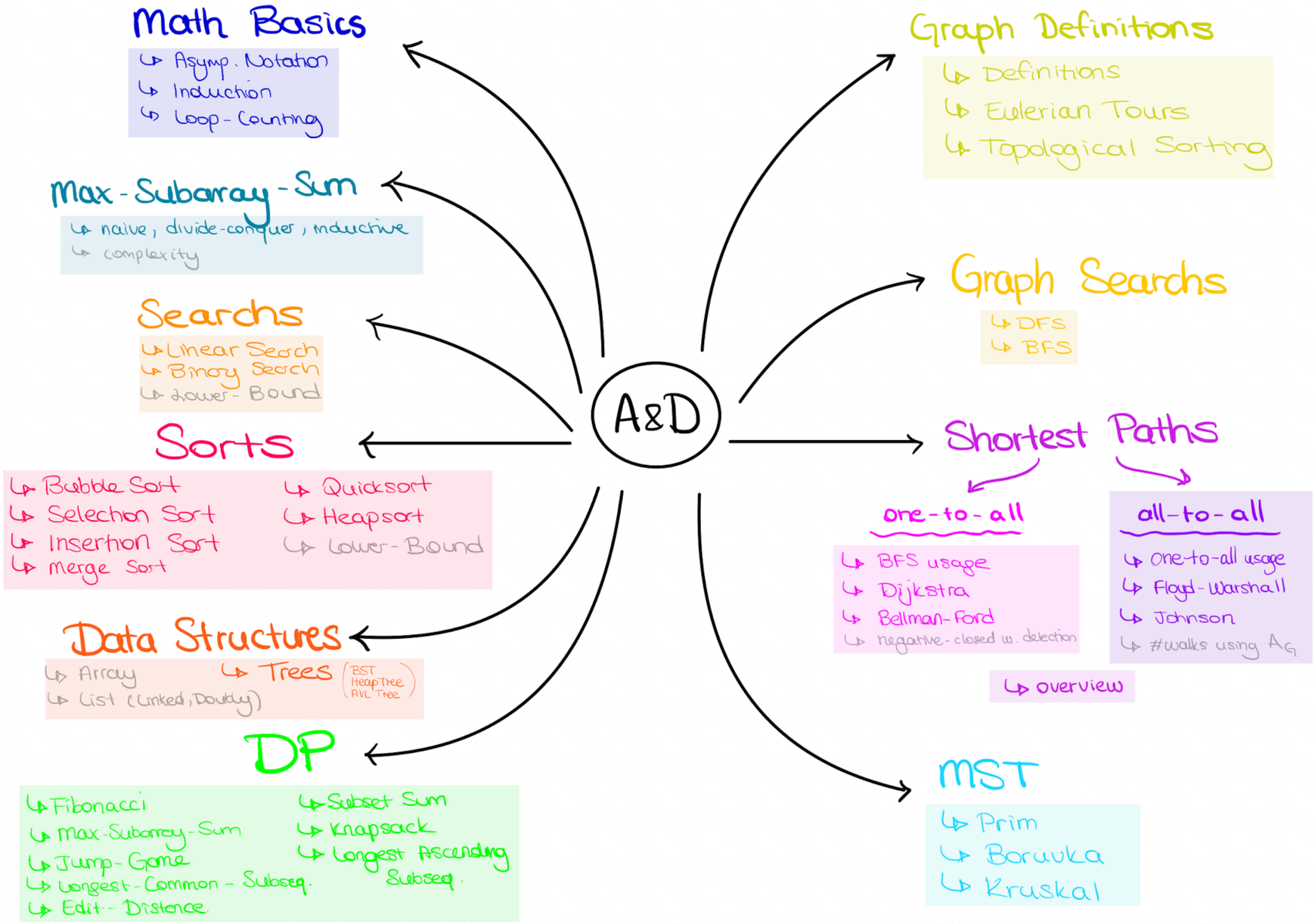
Nil Ozer

A&D

Exercise Session 7

Nil Ozer

A&D Overview


A&D
Math Basics
↳ 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

Outline

- Quiz
- Exercise Sheets

- DP I - Edit Distance
- DP II

- DP Mini Exam

- Next week

Quiz

Exercise Sheet 5

Bonus Feedback

- 5.1
  - “Attention mistakes” won’t be tolerated in exam
- 5.3
  - Don’t forget to refer to the pseudocode !
  - Is it O or Θ !!
- 5.4
  - Tree Proofs structure

- Feel free to correct me !

Tree Proofs

General Structure

- Base Case
  - Usually leaves !
- I.H.
  - Assume the property for some n
- I.S.
  - Show what happens to n in one iteration (assuming I.H.) , describe briefly
  - How does the recursion/iteration end ?
    - arriving to the leaf , root
    - fulfilling an if condition

Exercise Sheets

- Exercise Sheet 4 left for next time, again :(

- Exercise Sheet 6 peergrading
  - 6.1 this week
  - Emails are sent

- New groups for Exercise Sheet 7 !

DP I

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]

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 :

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

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

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

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 :

we don’t use i in I we use i in I

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

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

Let’s take a break

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.

A [1, 3, 5, 7]

Is it a subsequence ?

[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 !

DP[0...n-1][1...n]

Definition of the DP table :

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 :

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 i by being smaller

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

Extracting the solution : The solution is found by backtracking

DP

Exam Question

/ 9 P

Theory Task T3.

You are given an array of n natural numbers $a_1, \ldots, a_n \in \mathbb{N}$, and two natural numbers $A, B \in \mathbb{N}$. You want to determine whether there is a subset $I \subseteq \{1, \ldots, n\}$ satisfying

$$\sum_{i \in I} a_i = A \quad \text{and} \quad \sum_{i \in I} a_i^2 = B.$$

For example,

- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1, 4, 5, 3]$, $A = 8$ and $B = 30$ is *yes* because the set of indices $I = \{1, 4, 6\}$, which corresponds to $(a_i)_{i \leq I} = [2, 1, 5]$, yields the *sum* $2 + 1 + 5 = 8$ and the *sum-of-squares* $2^2 + 1^2 + 5^2 = 30$.
- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1]$, $A = 6$ and $B = 15$ is *no*.

Provide a *dynamic programming* algorithm that determines whether such a subset I exists. In order to get full points, your algorithm should have an $O(n \cdot A \cdot B)$ runtime. Address the following aspects in your solution:

DP

How to learn

- Theory, written tasks :
  - Exam questions T3 !!
  - Exercise sheets
  - geeksforgeeks

Always a combination of the ideas dicussed in lecture !

- Coding :
  - CodeEx exercises , my videos
  - Old Exam exercises
  - Leetcode https://leetcode.com/studyplan/dynamic-programming/

Table ? 🤦‍♀️

DP

Exam Tipps

- Get a hint from the running time
  - Doesn’t always work!
- Have an order for yourself

Always a combination of the ideas dicussed in lecture !

- The definition of an entry should be very clear to you, at all times !

- Initialization : What should the entry be in base cases (ex : A = [ ])
- Recursion : How can you use the previous entries to get the current entry
  - This is the only question that you’re answering !!

Done with DP !

DP Mini Exam (lol)

Next Week

Questions

Feedbacks , Recommendations

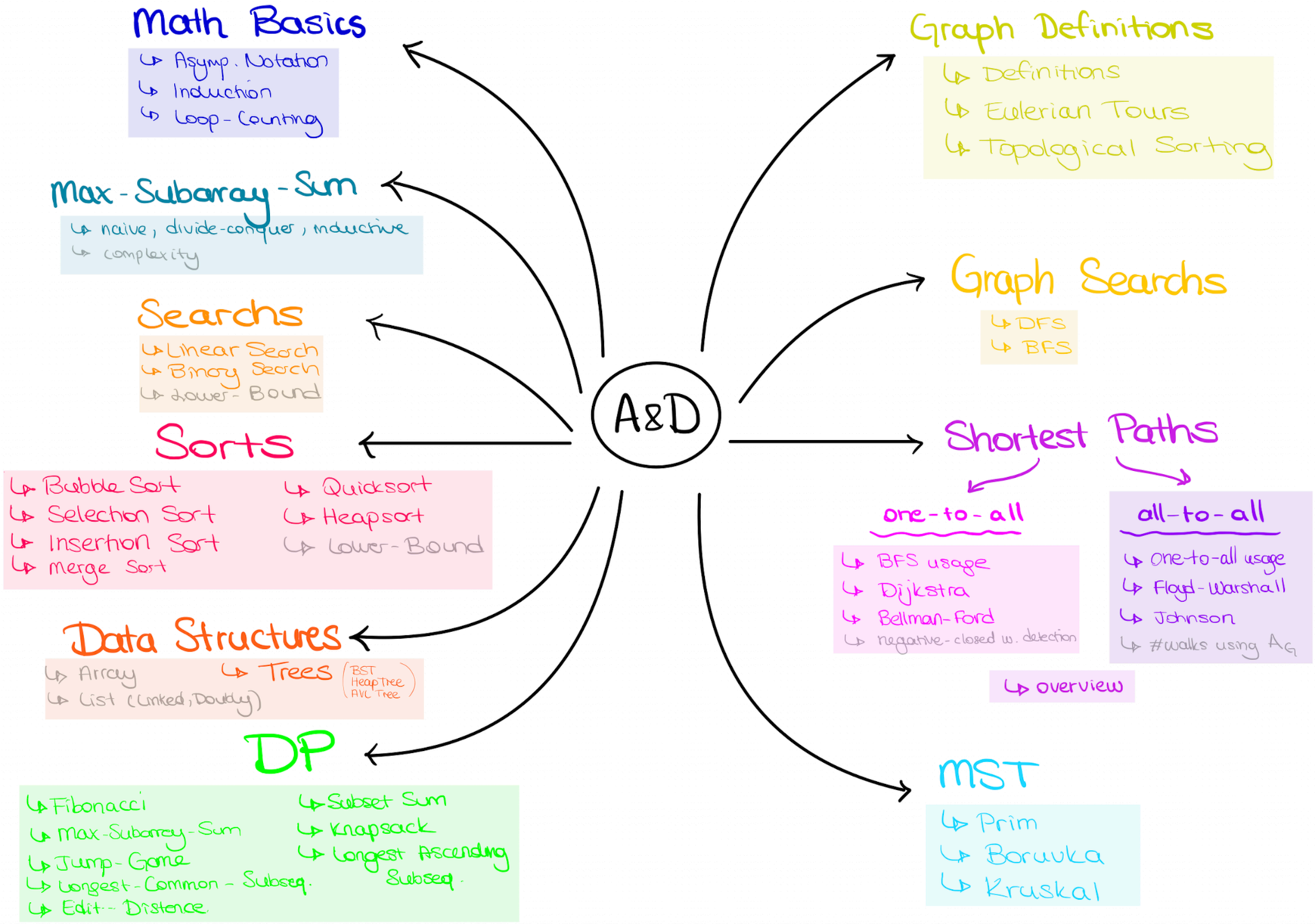
Nil Ozer

A&D

Exercise Session 8

Nil Ozer

A&D Overview


A&D
Math Basics
↳ 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

Outline

- Quiz
- Exercise Sheets

- Graph Definitions
- Short Proofs
- Exam Question

- Next week

Quiz

Mid Semester Feedback

Exercise Sheets

- Exercise Sheet 6 feedback left for next time

- Exercise Sheet 7 peergrading
  - 7.5 this week
  - Emails will be sent

- New groups for Exercise Sheet 7,8 and 9 !

Graph Definitions

Graph

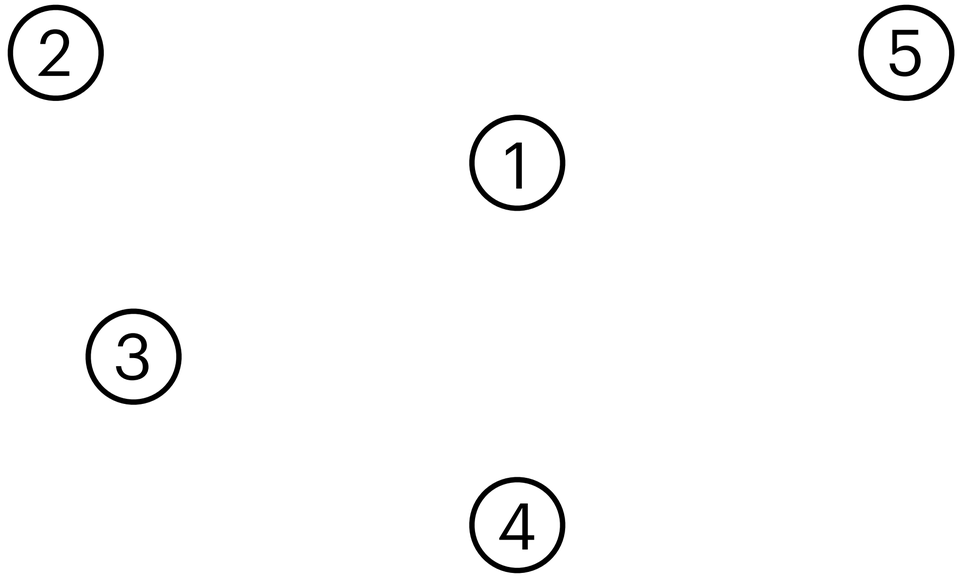
$$G = (V, E)$$

Graph

Node / Vertex

$G = (V, E)$

$V = \{v_1, ..., v_n\}, |V| = n$ the set of nodes

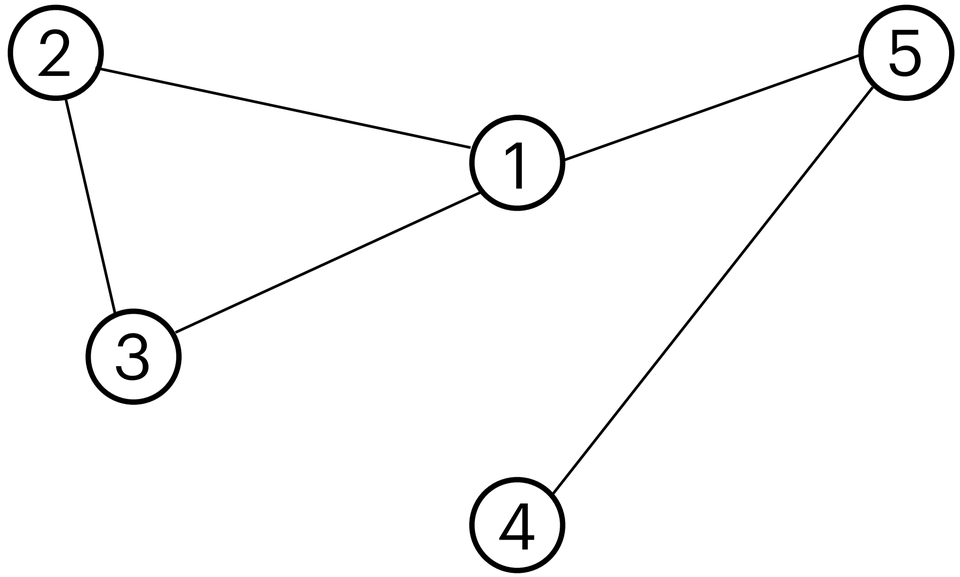


Graph

Edge

$G = (V, E)$

$E = \{e_1, \ldots, e_m\}, |E| = m$ the set of edges



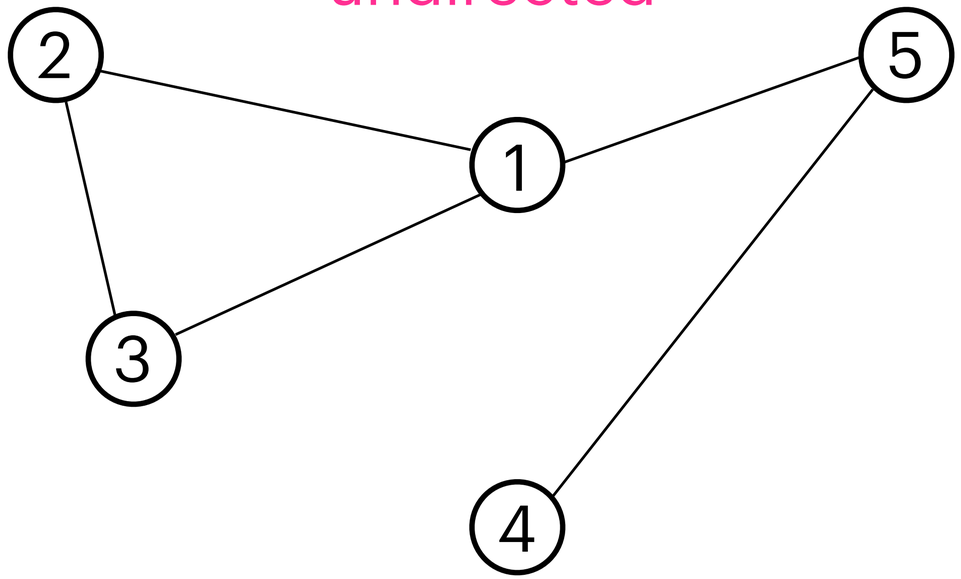
Graph

$G = (V, E)$

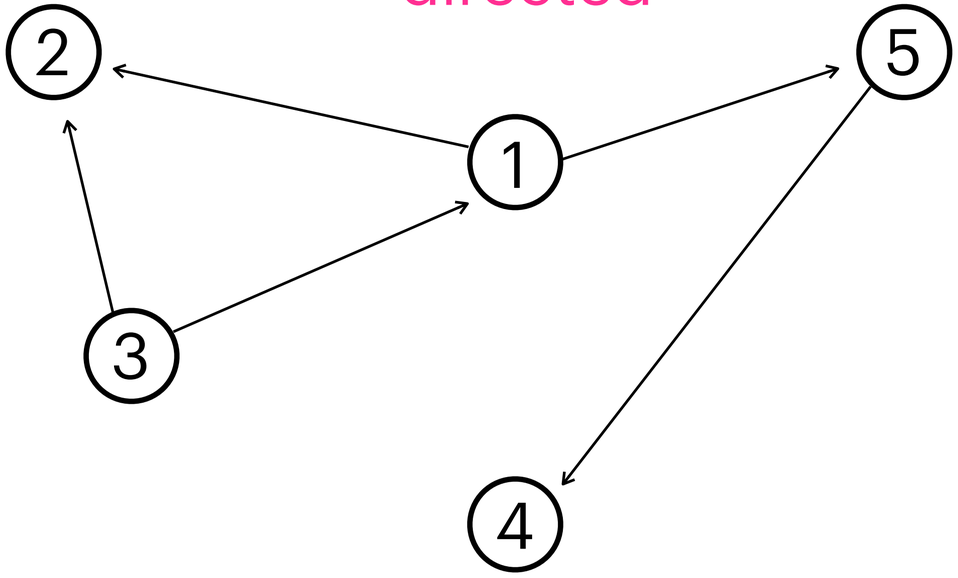
$E = \{e_1, ..., e_m\}, |E| = m$ the set of edges

Undirected / Directed Graph

undirected



directed



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

$E \subseteq V \times V$

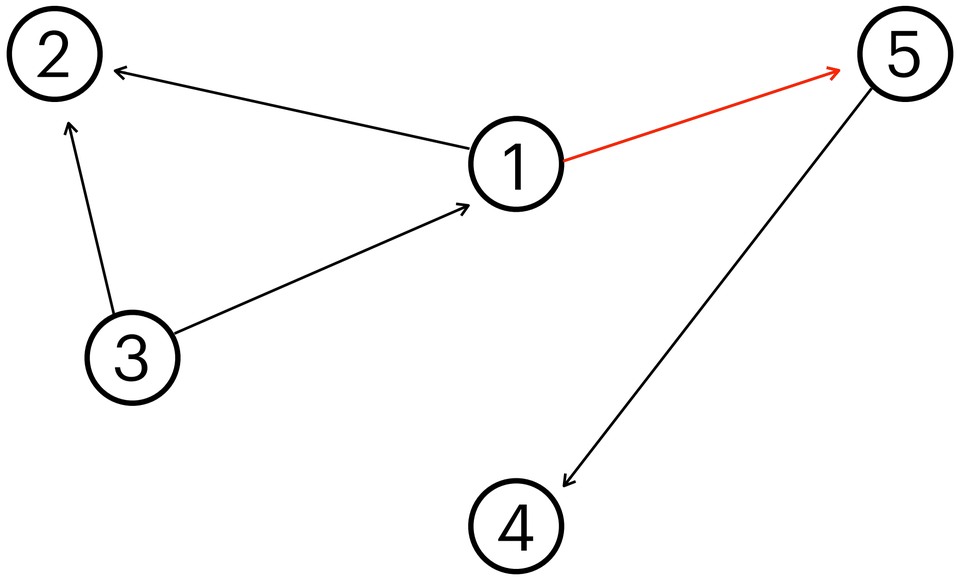
$e_k = \{v_i, v_j\}$

$e_k = (v_i, v_j)$

Graph

adjacent / incident

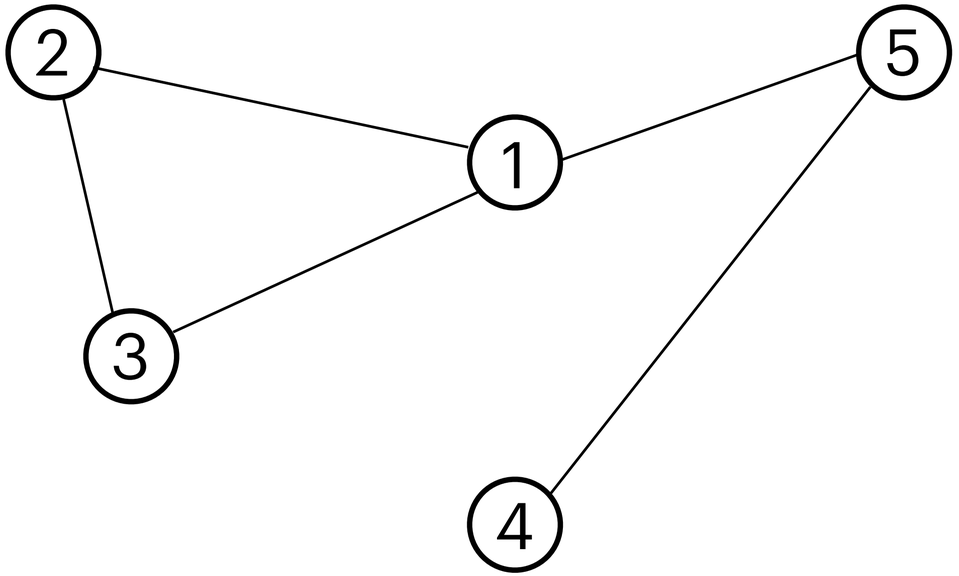
- v_i and v_j are adjacent iff : $\{v_i, v_j\} \in E$: $(v_i, v_j) \in E$:
- v_i and e_k are incident iff : $v_i \in e_k$.



Graph

Degree

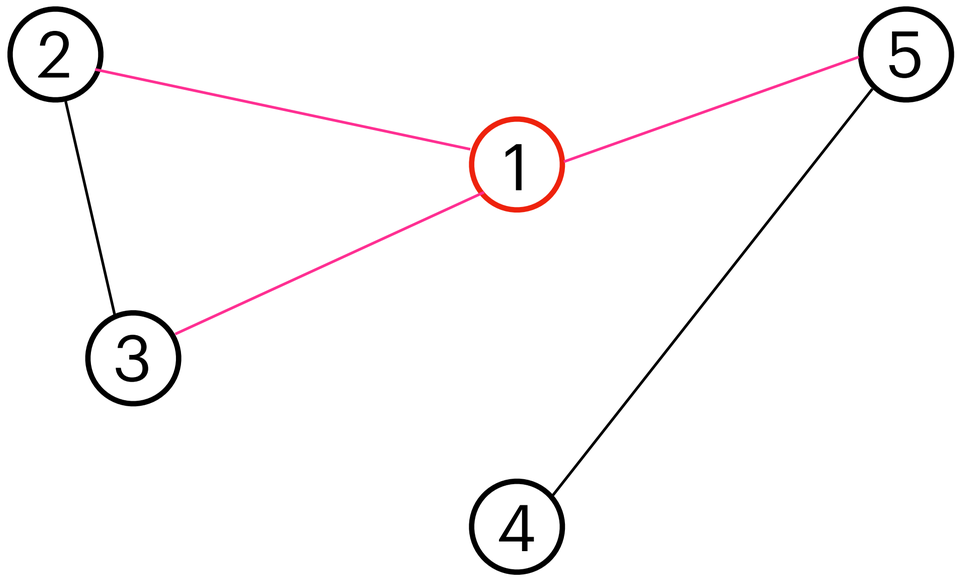
- deg(v) : The degree of a vertex v of a graph is the number of edges incident to this vertex, with loops counted twice.
- In directed graphs:
  - in-degree $\deg^-(v)$
  - out-degree $\deg^+(v)$



Graph

Degree

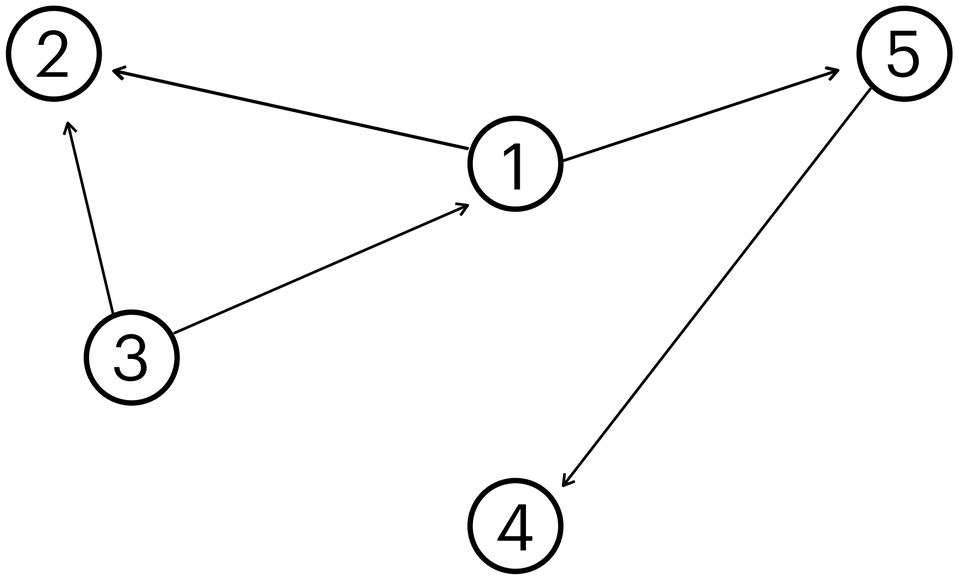
- deg(v) : The degree of a vertex v of a graph is the number of edges incident to this vertex, with loops counted twice.
- In directed graphs:
  - in-degree $\deg^-(v)$
  - out-degree $\deg^+(v)$



Graph

Degree

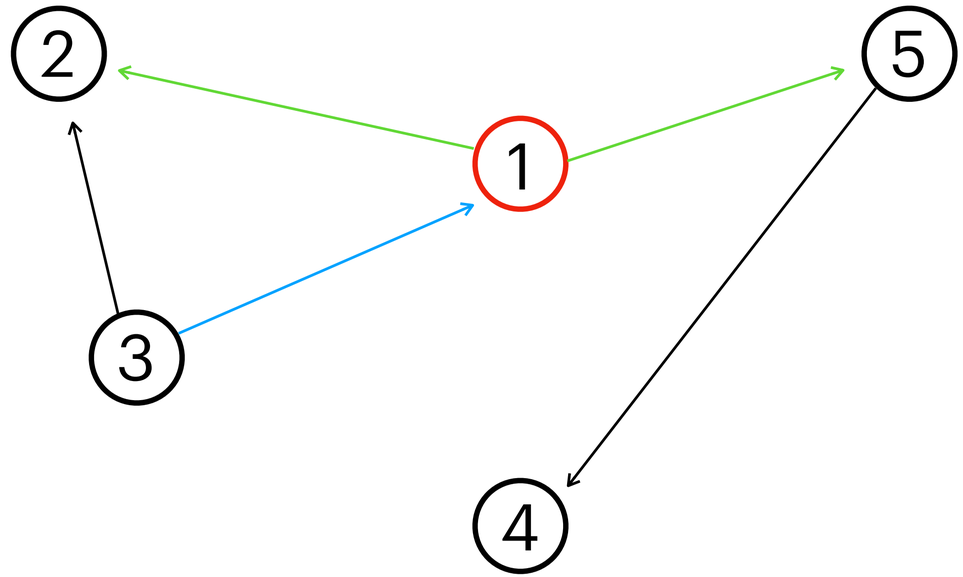
- deg(v) : The degree of a vertex v of a graph is the number of edges incident to this vertex, with loops counted twice.
- In directed graphs:
  - in-degree $\deg^-(v)$
  - out-degree $\deg^+(v)$



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 $\deg^+(v)$



Graph

Walk vs Path

walk

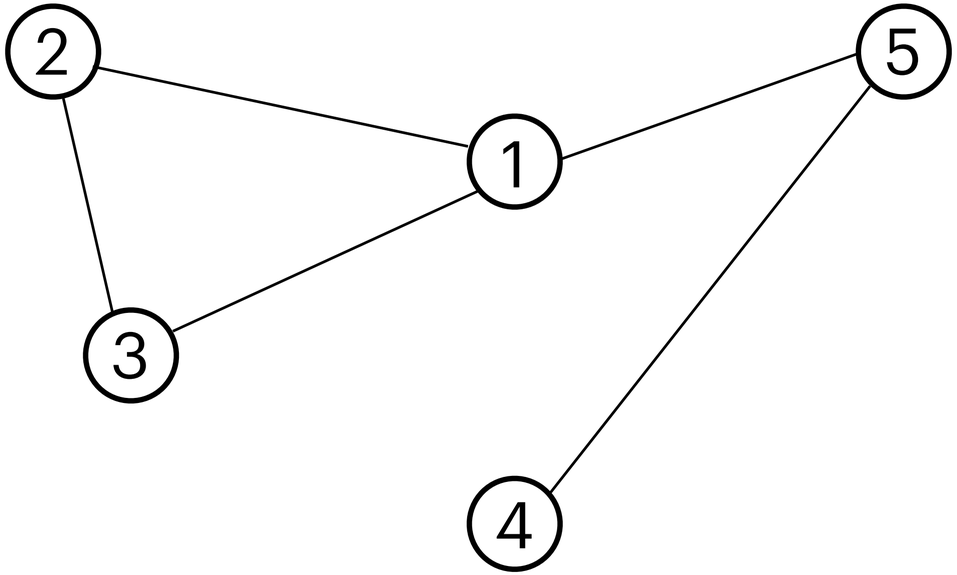
- A sequence of vertices $(v_0, v_1, \ldots, v_k)$ (with $v_i \in V$ for all i) is a **walk** (german "Weg") if $\{v_i, v_{i+1}\}$ is an edge for each $0 \leq i \leq k-1$. We say that v_0 and v_k are the **endpoints** (german "Startknoten" and "Endknoten") of the walk. The **length** of the walk $(v_0,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

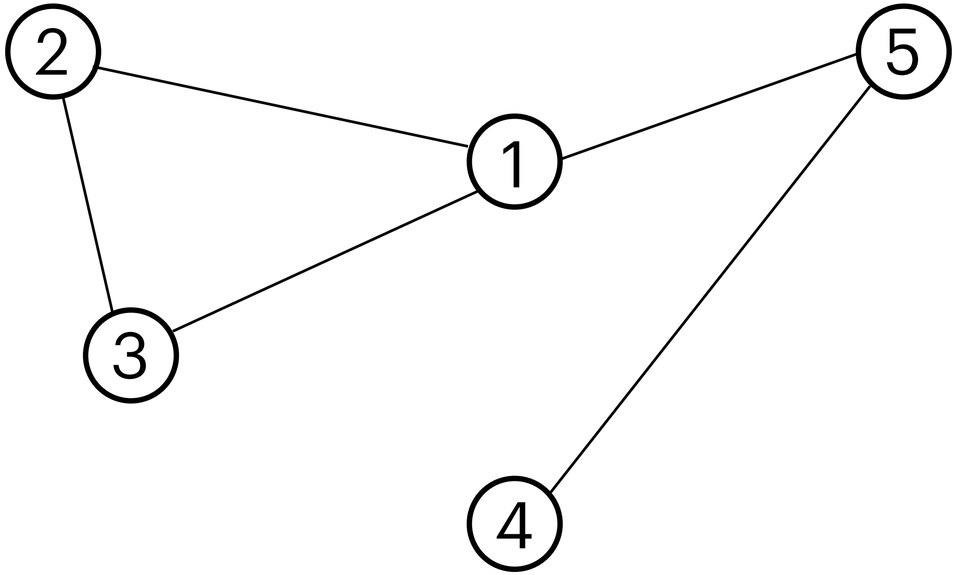
- A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **closed walk** (german "Zyklus") if it is a walk, $k \geq 2$ and $v_0 = v_k$.

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

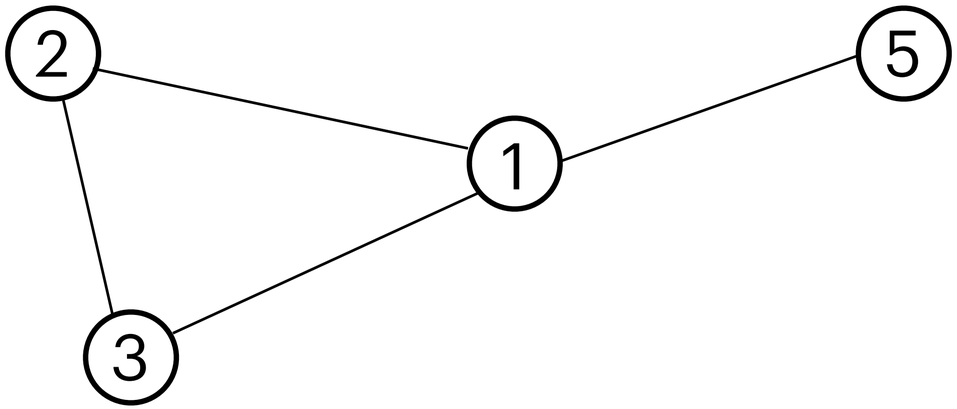
- A **Eulerian walk** (german "Eulerweg") is a walk that contains every edge exactly once.

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 ? ❌



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 $v_0 = v_k$.

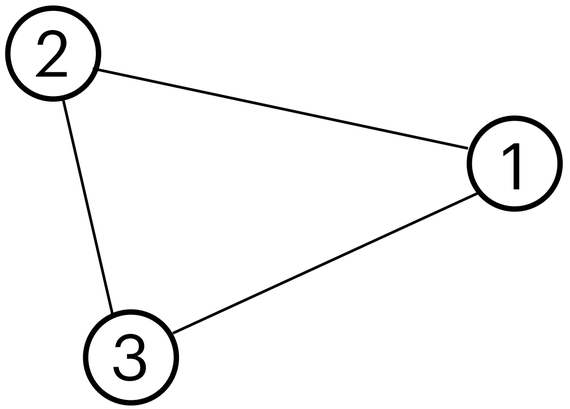
Eulerian Walk

- 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 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

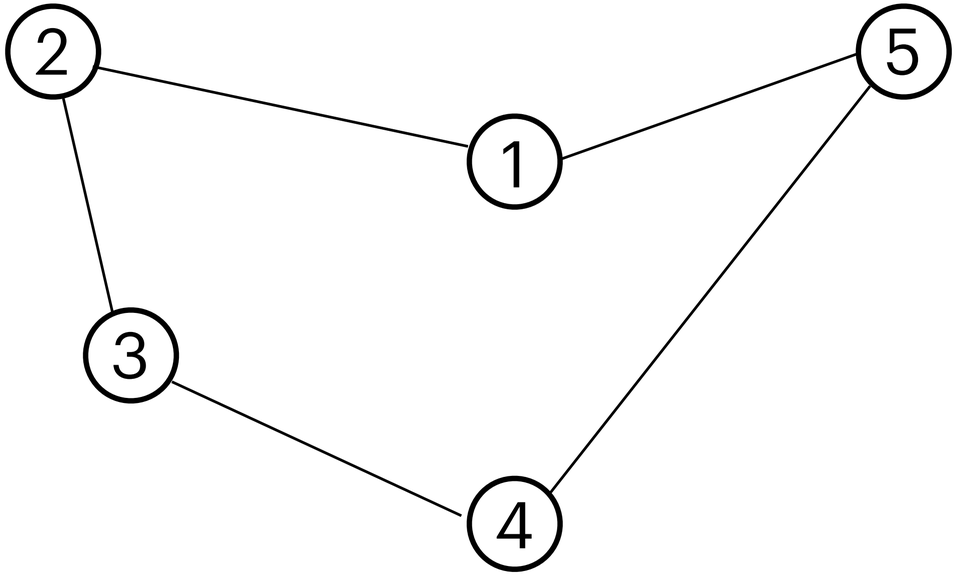
- A **Hamiltonian path** (german "Hamiltonpfad") is a path that contains every vertex.

Hamiltonian Cycle

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

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

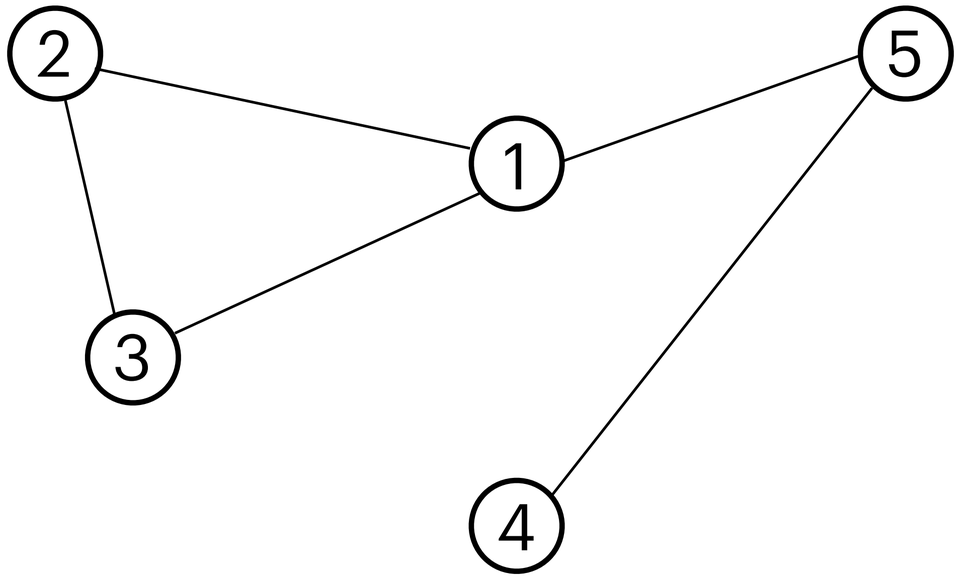
(5, 1, 2, 3, 4)	✅	❌
(5, 1, 2, 3, 4, 5)	❌	✅



Graph

Connectivity

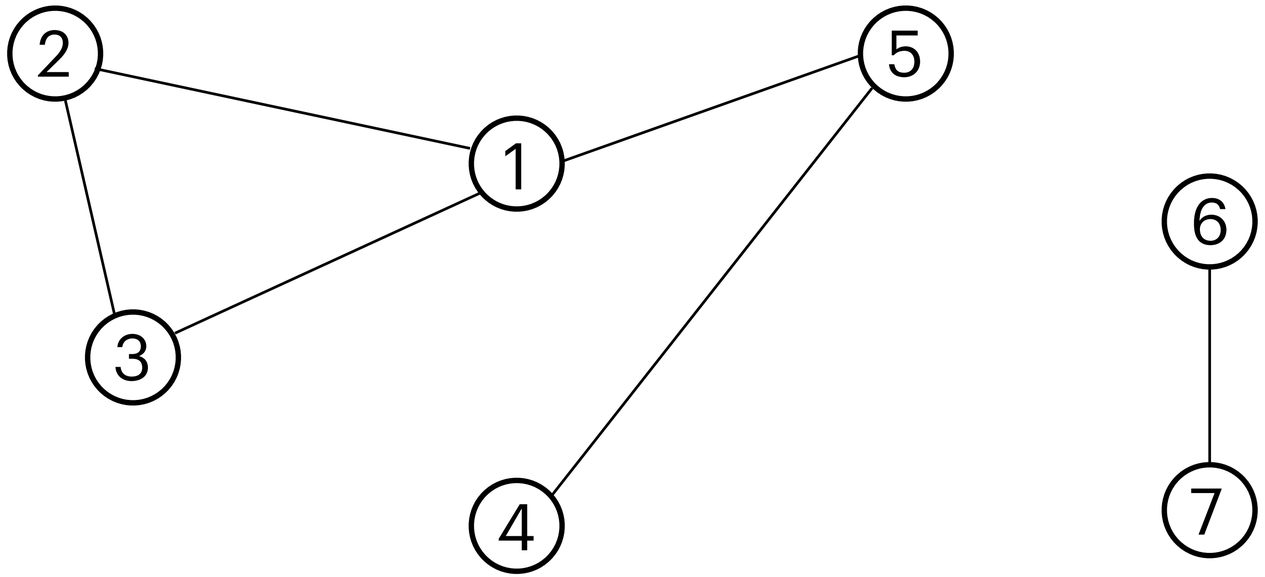
- For $u, v \in V$, we say u **reaches** v (or v is **reachable** from u; german "u erreicht v") if there exists a walk with endpoints u and v.
- A **connected component** of G is an equivalence class of the (equivalence) relation defined as follows: Two vertices $u, v \in V$ are equivalent if u reaches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

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.

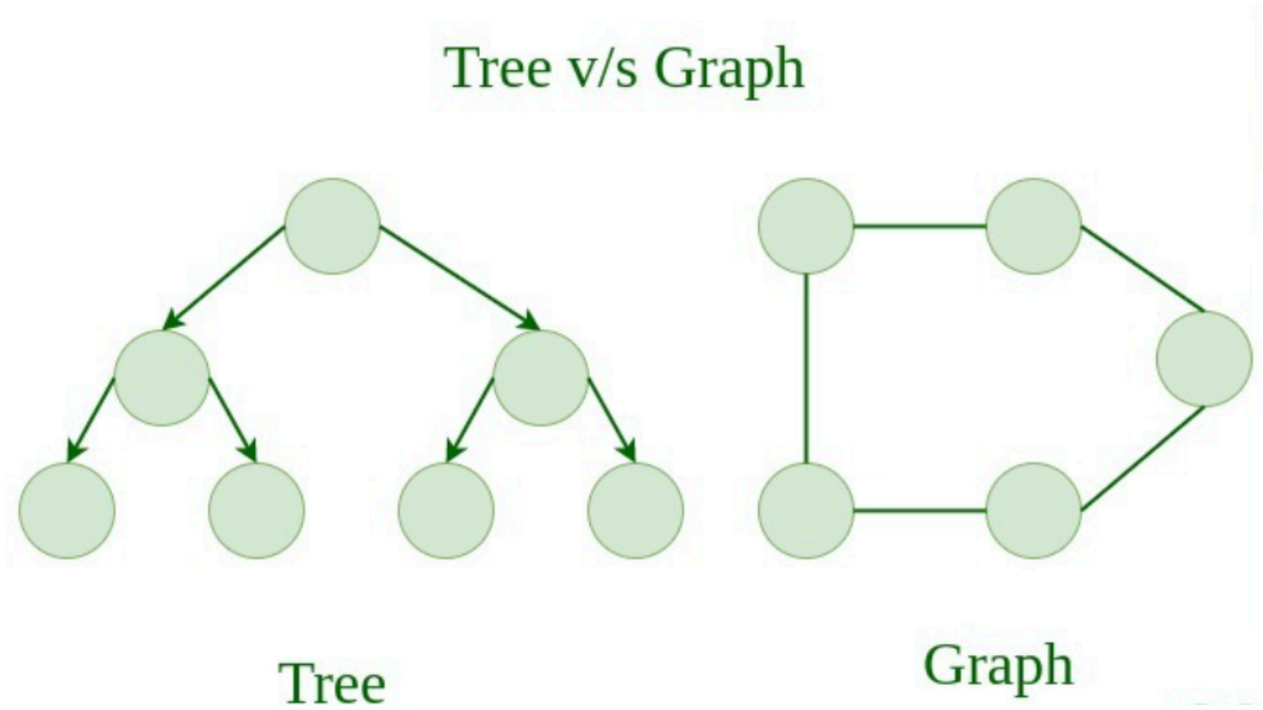


Graph

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.



Graph

Handshaking Lemma

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

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

Definitions

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

- For $v \in V$, the **degree** $\deg(v)$ of v (german "Knotengrad") is the number of edges that are incident to v.
- 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 $v_0 = v_k$.
- 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 **closed Eulerian walk** (german "Eulerzyklus") is a closed walk that contains every edge exactly once.
- A **Hamiltonian path** (german "Hamiltonpfad") is a path that contains every vertex.
- A **Hamiltonian cycle** (german "Hamiltonkreis") is a cycle that contains every vertex.
- 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.
- A graph G is a **tree** (german "Baum") if it is connected and has no cycles.

Let’s take a break

Graph

Short Proofs

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.

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.

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

(d) If every vertex of a non-empty graph G has degree at least 2, then G contains 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 $n - 1$).

(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).

Graph

Exam Question

/ 5 P

c) *Graph quiz:* For each of the following claims, state whether it is true or false. You get 1P for a correct answer, -1P for a wrong answer, 0P for a missing answer. You get at least 0 points in total.

As a reminder, here are a few definitions for a (directed) graph $G = (V, E)$:

For $k \geq 2$, a (directed) *walk* is a sequence of vertices $v_1, \ldots, v_k$ such that for every two consecutive vertices v_i, v_{i+1}, we have $\{v_i, v_{i+1}\} \in E$ (resp. $(v_i, v_{i+1}) \in E$ for a directed walk).

A (directed) *closed walk* is a (directed) walk with $v_1 = v_k$.

A (directed) *cycle* is a (directed) closed walk where $k \geq 3$ and all vertices (except v_1 and v_k) are distinct.

A (directed) *closed Eulerian walk* is a (directed) closed walk which traverses every edge in E exactly once.

For a vertex v in a directed graph $G = (V, E)$, the *in-degree* of v is the number of edges in E that end in v (i.e., of the form (w, v)), and the *out-degree* of v is the number of edges in E that start in v (i.e., of the form (v, w)).

Claim	**true**	**false**
A connected graph must contain a cycle.	☐	☐
A graph $G = (V, E)$ with $\lvert E\rvert \leq \lvert V\rvert - 1$ is a tree.	☐	☐
Let $G = (V, E)$ be a graph with $\lvert E\rvert \geq 4$, which contains a closed Eulerian walk. If we remove one edge from E, the resulting graph does not contain a closed Eulerian walk, no matter which edge we remove (the vertex set does not change).	☐	☐
Let $G = (V, E)$ be a *directed* graph. If the in-degree and out-degree of every vertex $v \in V$ is even, then G contains a *directed* closed Eulerian walk.	☐	☐
Let $G = (V, E)$ be an undirected graph. Then there is a way to direct the edges of G such that the resulting directed graph does not contain a directed cycle.	☐	☐

Graph Definitions

Exam Tipps

- T/F or a proof !
- Know all of the definitions
  - Don’t mix up similar ones, realise connections!
    - walk , closed walk , eulerian
    - path , cycle, hamiltonian
- Gain an intuition
  - Don’t rush ! Don’t gamble !
  - Do this by actually coming up with short proofs !
- Practice !

Next Week

Questions

Feedbacks , Recommendations

Nil Ozer

Additional Slides

Euler(G)

Euler(G): (finde Eulerzyklus in G wenn existent)

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

Euler Walk(u):

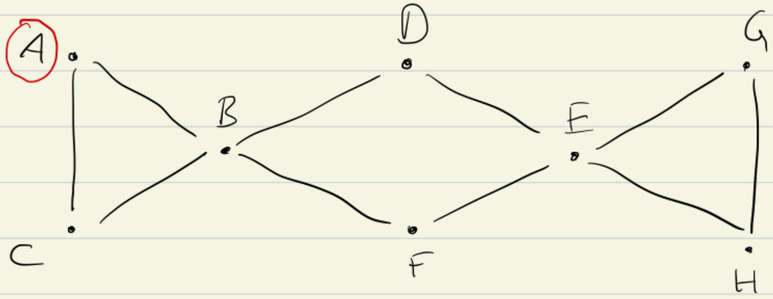
⚠ auch nicht in Rekursion markiert!

for $uv \in E$, nicht markiert

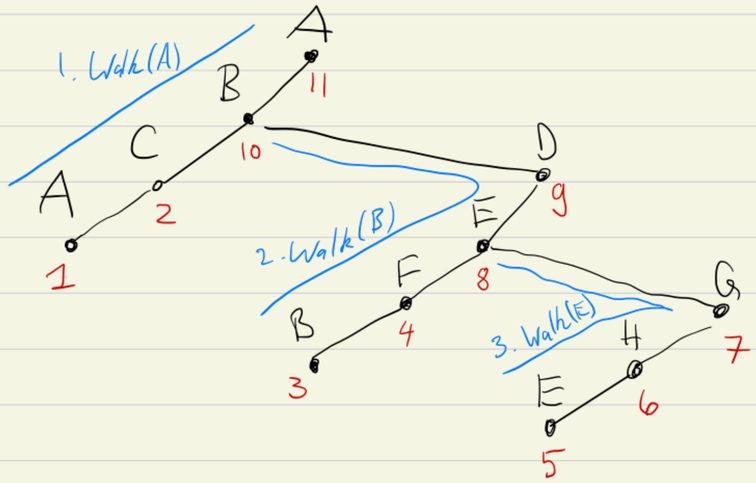
- markiere Kante uv
- EulerWalk(v)

$Z \leftarrow (Z, u)$

Beispiel:



Rekursionsbaum



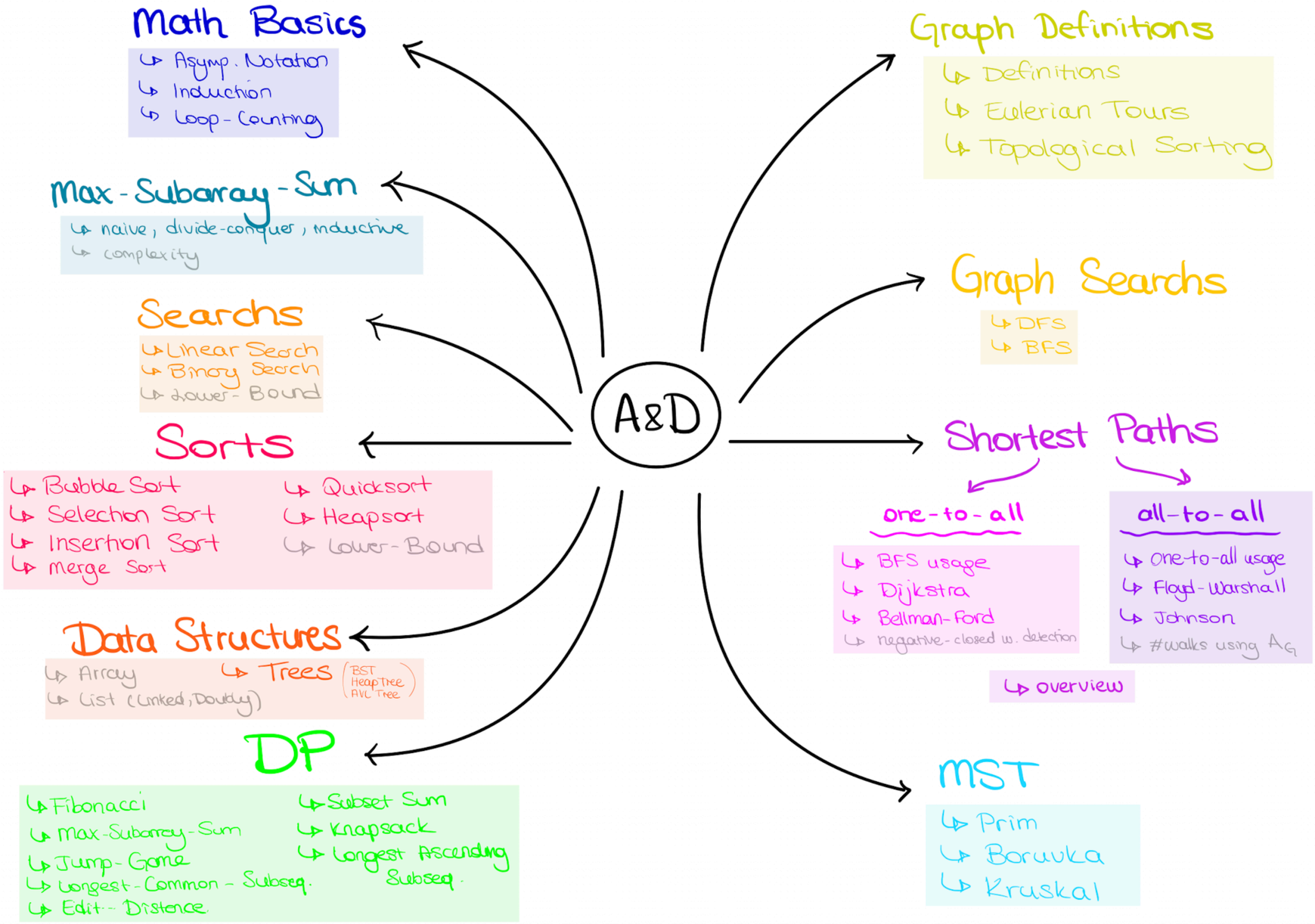
Z: A C B F E H G E D B A

A&D

Exercise Session 9

Nil Ozer

A&D Overview


A&D
Math Basics
↳ 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

Outline

- Quiz
- Exercise Sheets
- Questions from you

- Graph Definitions - Exam Question

- Graph Searchs - DFS
- Topological Sorting

- DP Mini Exam - Proof at the end

Quiz

Exercise Sheet 6

Bonus Feedback

- 6.1 :
  - Watch out for the comments !
  - Tree proof structure improved :) 👏
- 6.3 :
  - Indexes ☠️
  - DP structure !!
- 6.4 : 👏

Peergrading

- Exercise Sheet 8 peergrading
  - 8.1 this week
  - Emails will be sent

Questions from you

- **Asymptotic Bound Consistency in Algorithm Runtime Questions**
  - We specify explicitly what you need to prove
    - Often we say that you should create an algorithm with runtime at most O(x) and then you need to justify why your algorithm is in O(x)
  - If we don't specify, head-ta says that you should give a Theta bound or at least an O bound that is as tight as possible.

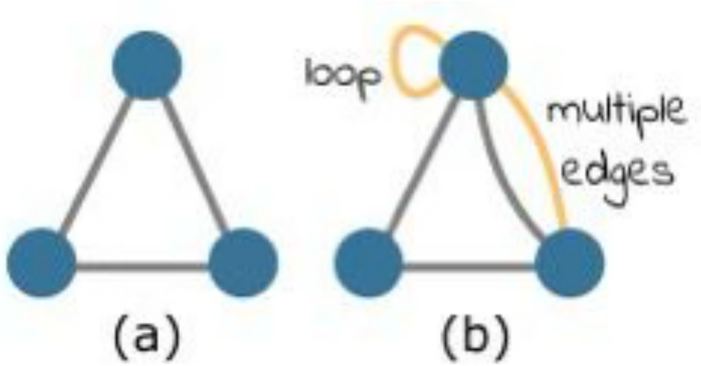
- **Is a tree a directed or undirected graph ?**
  - “tree”
  - “directed tree”

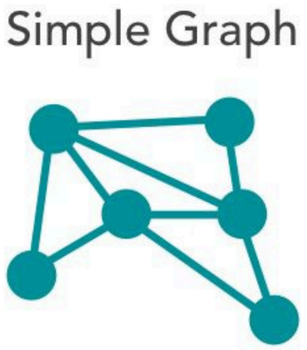
Questions from you

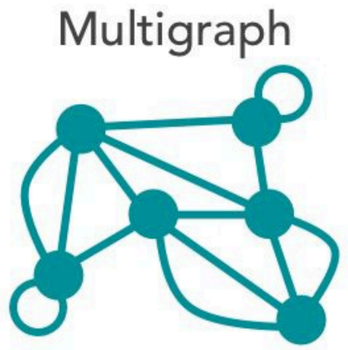
Valid Counterexamples for A&D !!

- For A&D we don’t consider multigraphs
  - unless explicitly specified otherwise.

- Simple Graph : no self loops or multiple edges between same vertices
- Multigraphs : self loops or multiple edges are allowed







Graph Definitions

Graph

Definitions

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

- For $v \in V$, the **degree** $\deg(v)$ of v (german "Knotengrad") is the number of edges that are incident to v.
- 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 $v_0 = v_k$.
- 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 **closed Eulerian walk** (german "Eulerzyklus") is a closed walk that contains every edge exactly once.
- A **Hamiltonian path** (german "Hamiltonpfad") is a path that contains every vertex.
- A **Hamiltonian cycle** (german "Hamiltonkreis") is a cycle that contains every vertex.
- 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.
- A graph G is a **tree** (german "Baum") if it is connected and has no cycles.

Graph

Exam Question

/ 5 P

c) *Graph quiz:* For each of the following claims, state whether it is true or false. You get 1P for a correct answer, -1P for a wrong answer, 0P for a missing answer. You get at least 0 points in total.

As a reminder, here are a few definitions for a (directed) graph $G = (V, E)$:

For $k \geq 2$, a (directed) *walk* is a sequence of vertices $v_1, \ldots, v_k$ such that for every two consecutive vertices v_i, v_{i+1}, we have $\{v_i, v_{i+1}\} \in E$ (resp. $(v_i, v_{i+1}) \in E$ for a directed walk).

A (directed) *closed walk* is a (directed) walk with $v_1 = v_k$.

A (directed) *cycle* is a (directed) closed walk where $k \geq 3$ and all vertices (except v_1 and v_k) are distinct.

A (directed) *closed Eulerian walk* is a (directed) closed walk which traverses every edge in E exactly once.

For a vertex v in a directed graph $G = (V, E)$, the *in-degree* of v is the number of edges in E that end in v (i.e., of the form (w, v)), and the *out-degree* of v is the number of edges in E that start in v (i.e., of the form (v, w)).

Claim	**true**	**false**
A connected graph must contain a cycle.	☐	☐
A graph $G = (V, E)$ with $\lvert E\rvert \leq \lvert V\rvert - 1$ is a tree.	☐	☐
Let $G = (V, E)$ be a graph with $\lvert E\rvert \geq 4$, which contains a closed Eulerian walk. If we remove one edge from E, the resulting graph does not contain a closed Eulerian walk, no matter which edge we remove (the vertex set does not change).	☐	☐
Let $G = (V, E)$ be a *directed* graph. If the in-degree and out-degree of every vertex $v \in V$ is even, then G contains a *directed* closed Eulerian walk.	☐	☐
Let $G = (V, E)$ be an undirected graph. Then there is a way to direct the edges of G such that the resulting directed graph does not contain a directed cycle.	☐	☐

Graph Definitions

Exam Tipps

- T/F or a proof !
- Know all of the definitions
  - Don’t mix up similar ones, realise connections!
    - walk , closed walk , eulerian
    - path , cycle, hamiltonian
- Gain an intuition
  - Don’t rush ! Don’t gamble !!!
  - Do this by actually coming up with short proofs !
- Practice, practice, practice!

Graph Searchs

DFS

Graph Searchs

DFS - with pre and post order

Algorithm 4 DFS(G)

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

Graph Searchs

DFS - with pre and post order

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

Algorithm 4 DFS(G)

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

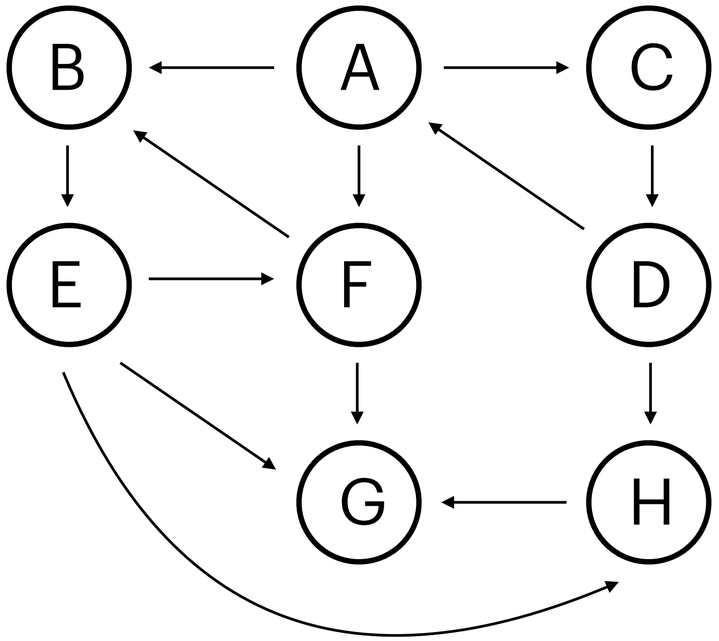
Algorithm 3 Visit(u)

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

Let’s take a break

Graph Searchs

DFS - Example



pre-order

post-order

tree edge

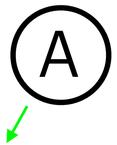
back edge

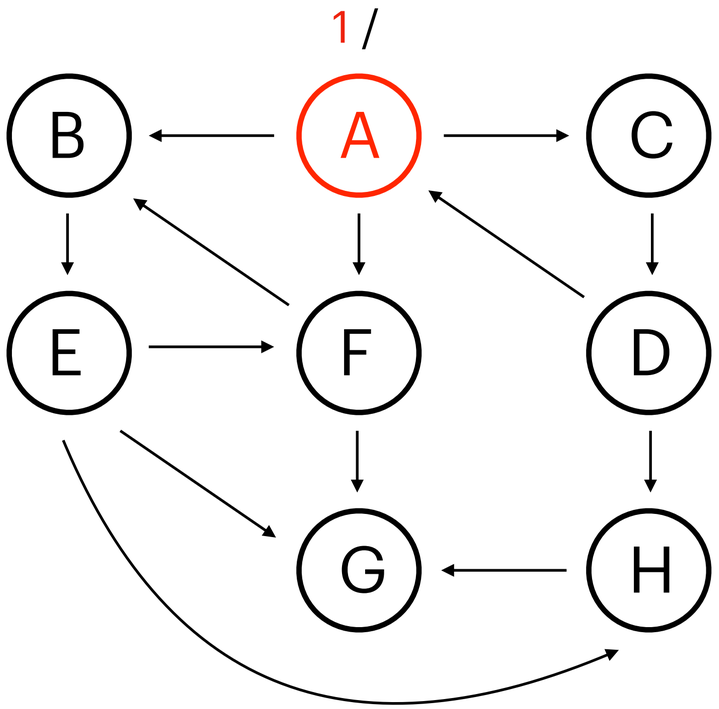
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

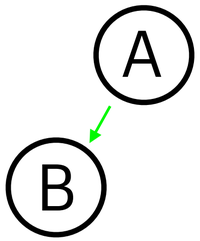
back edge

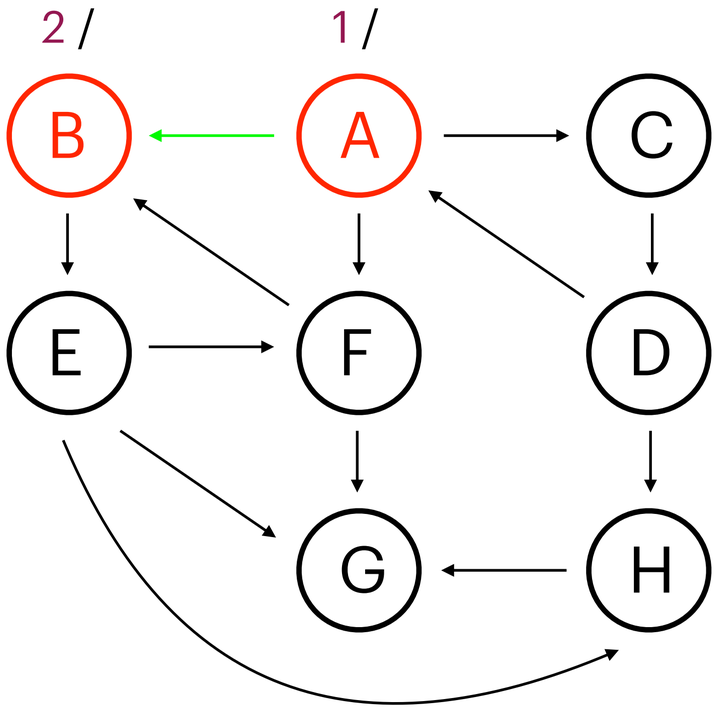
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

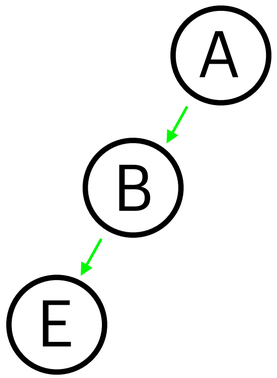
back edge

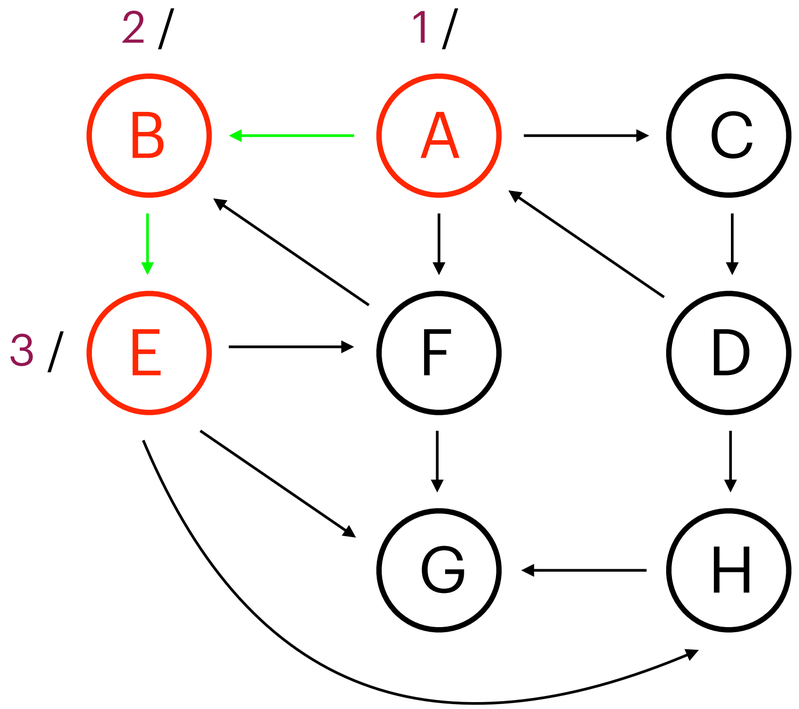
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

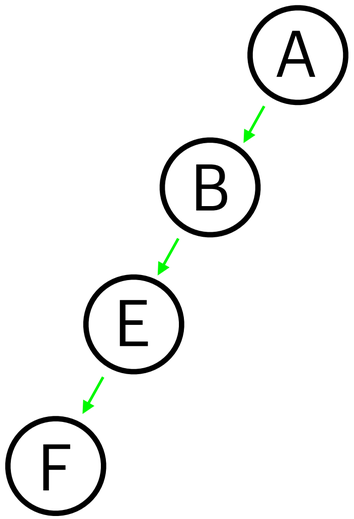
back edge

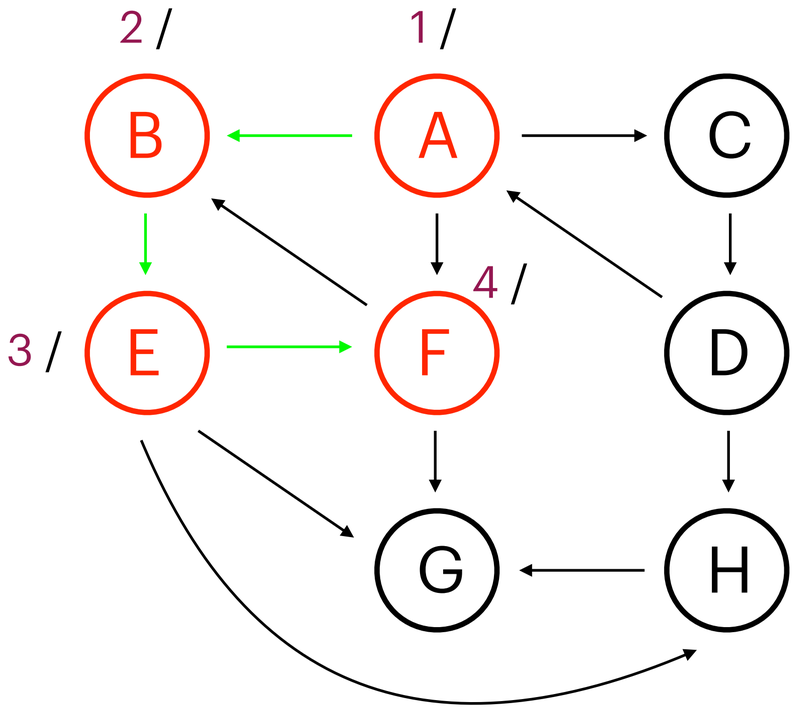
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

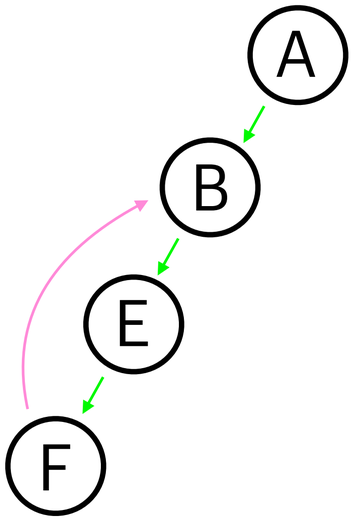
back edge

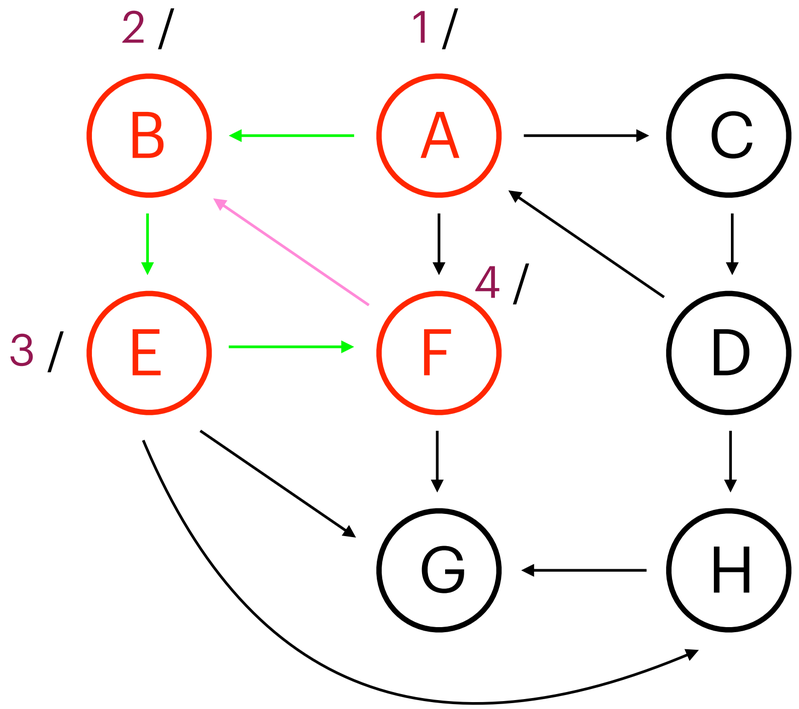
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

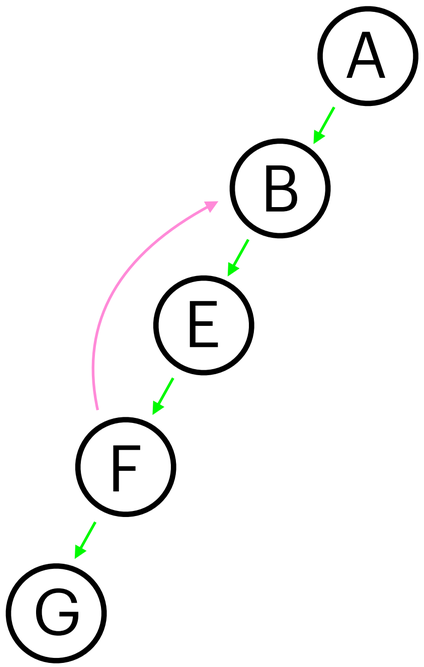
back edge

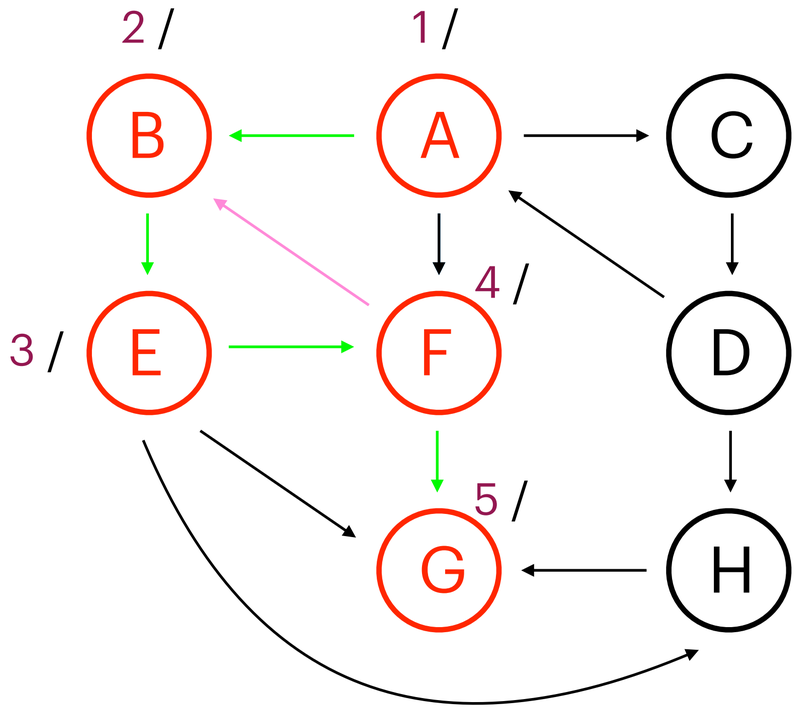
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

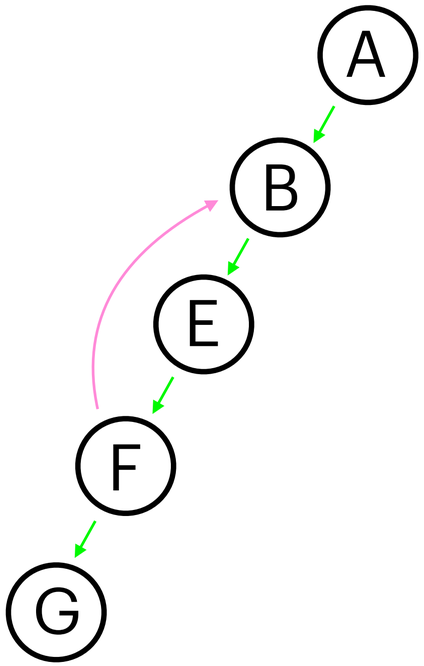
back edge

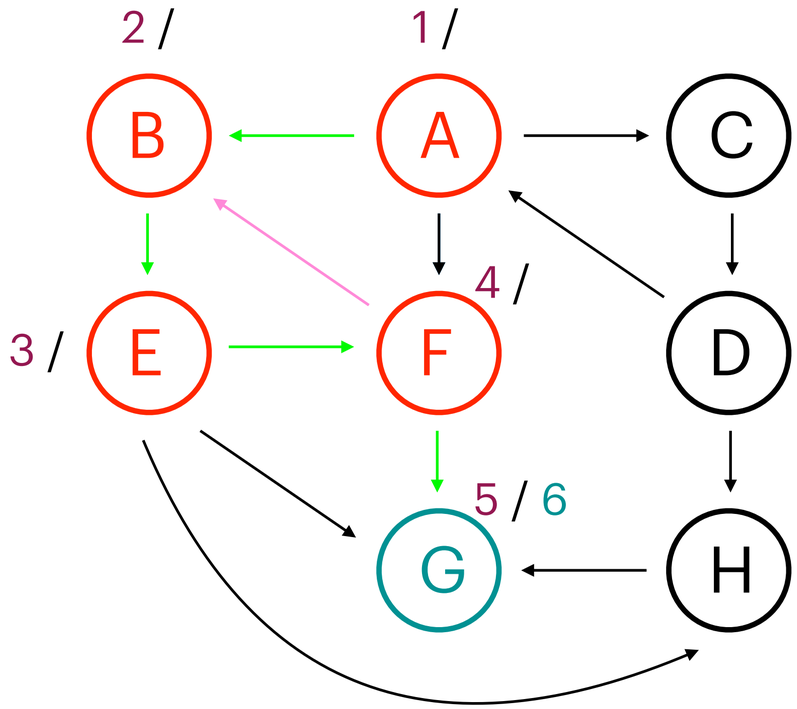
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

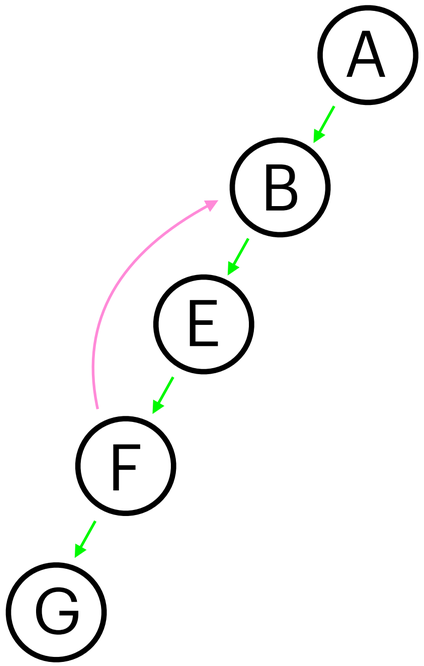
back edge

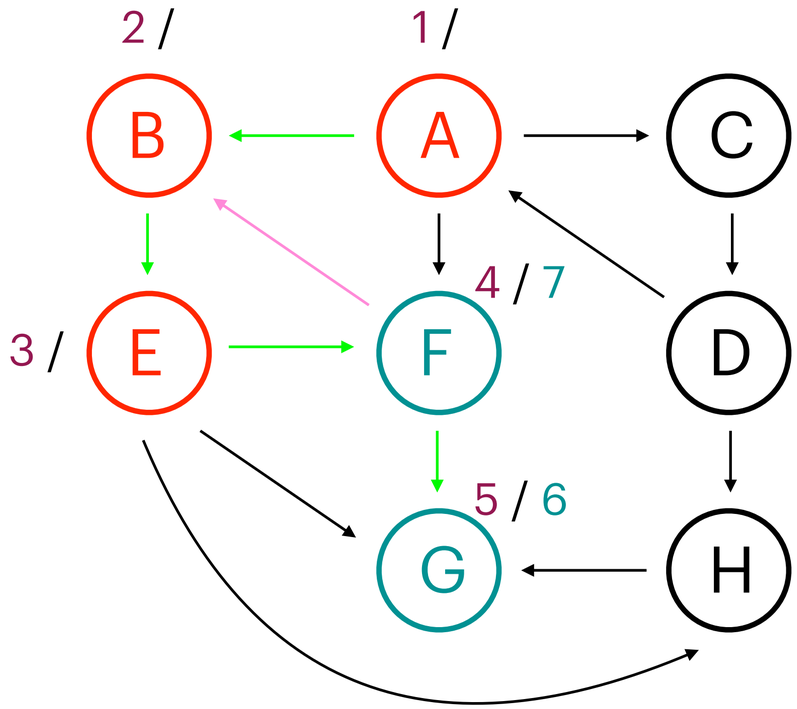
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

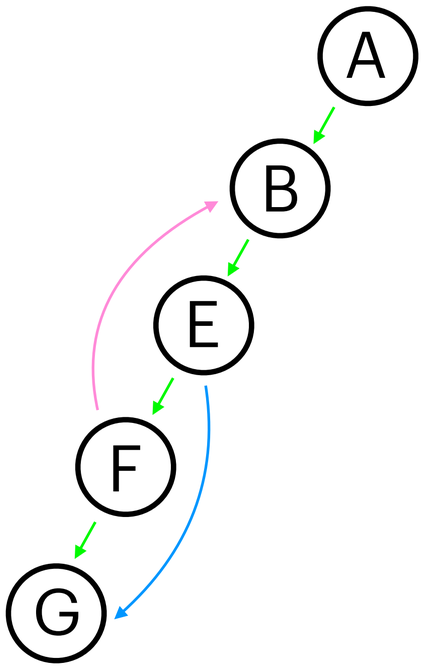
back edge

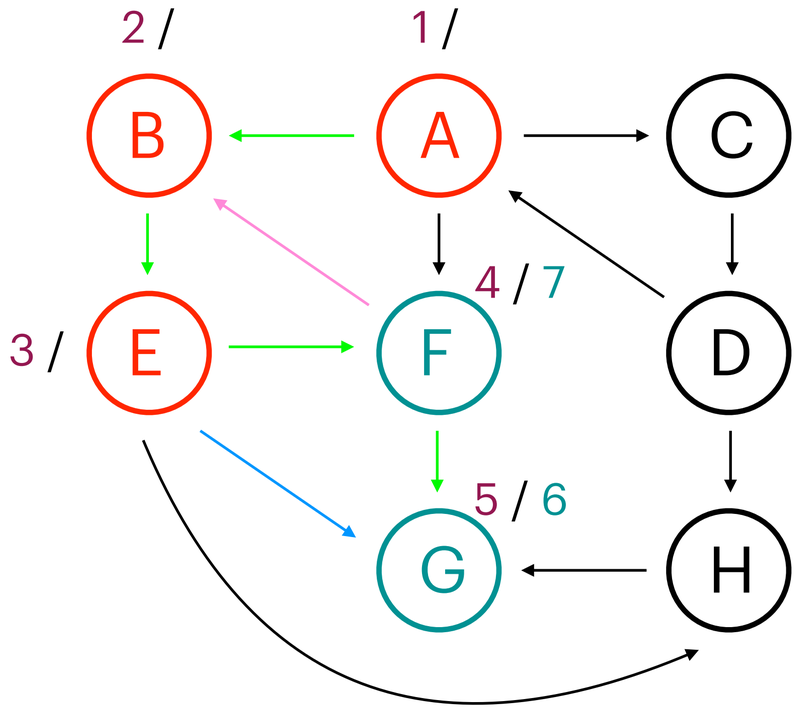
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

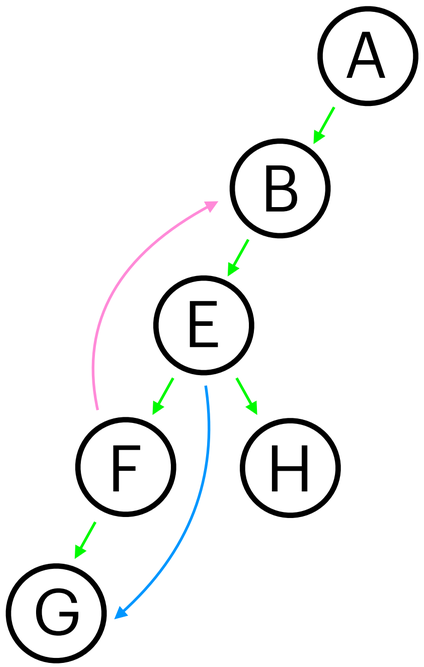
back edge

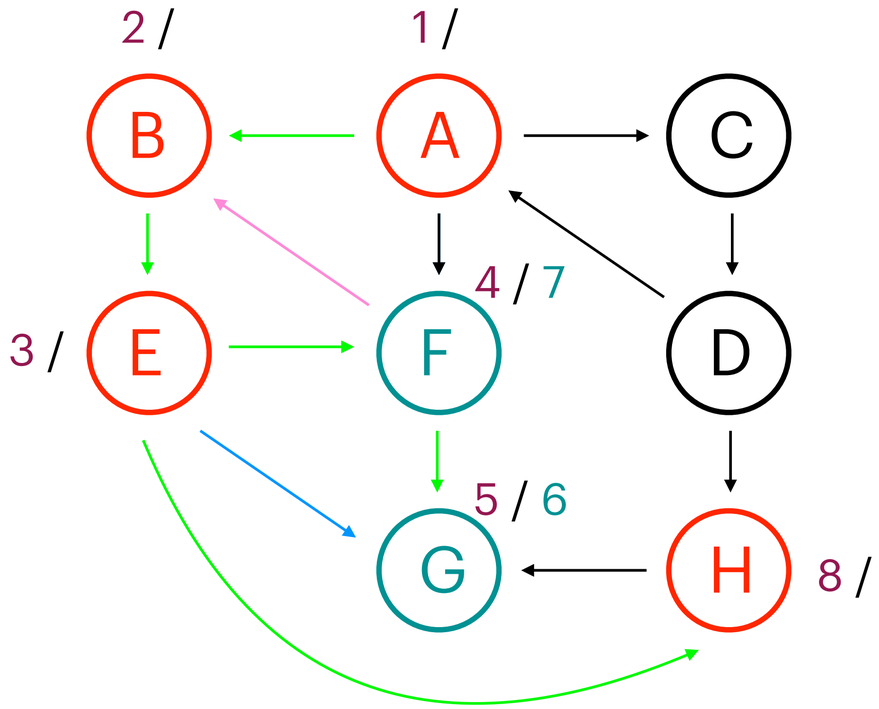
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

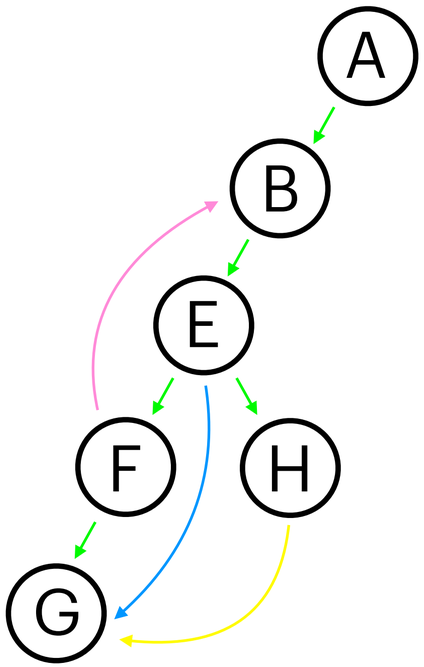
back edge

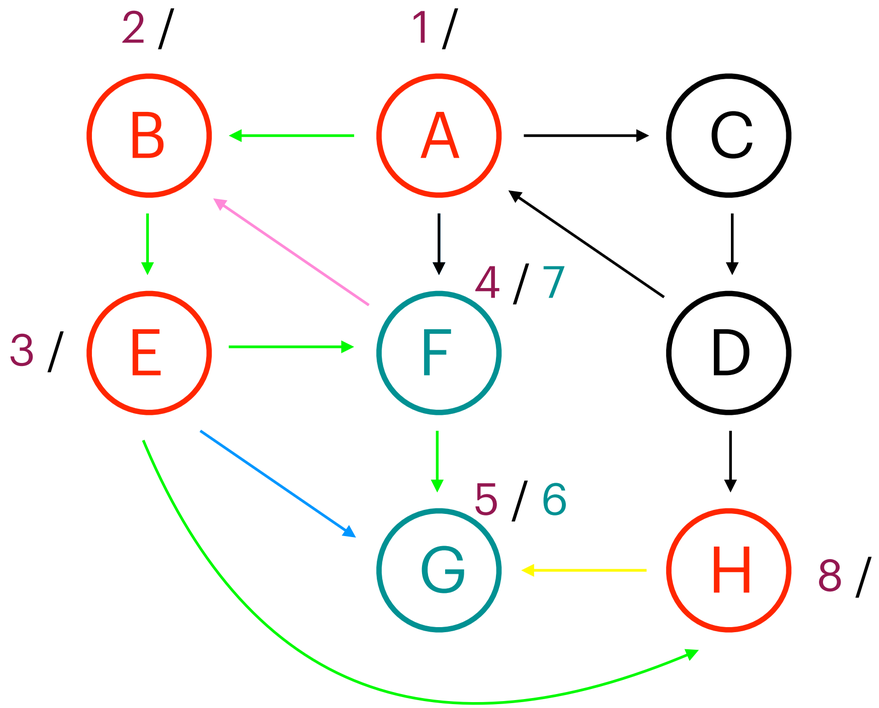
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

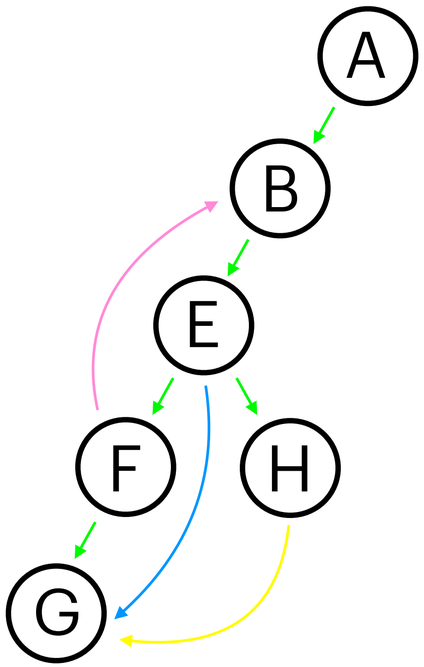
back edge

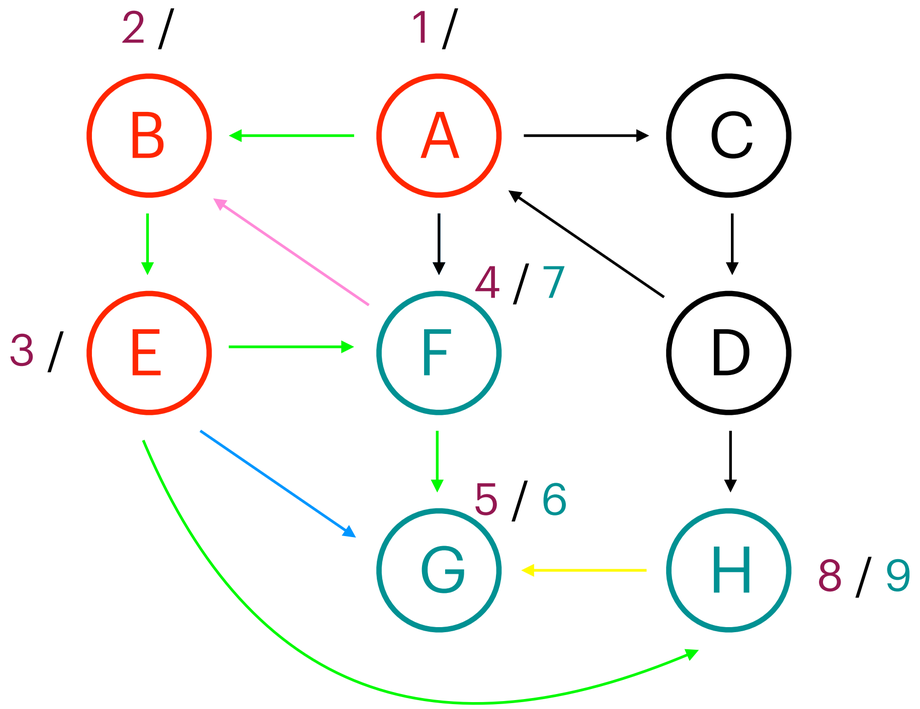
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

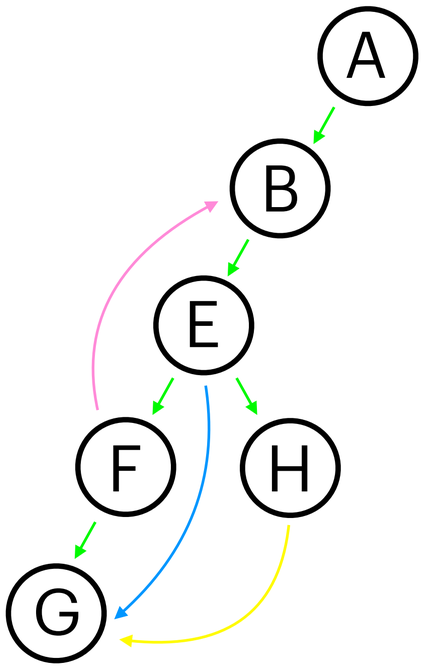
back edge

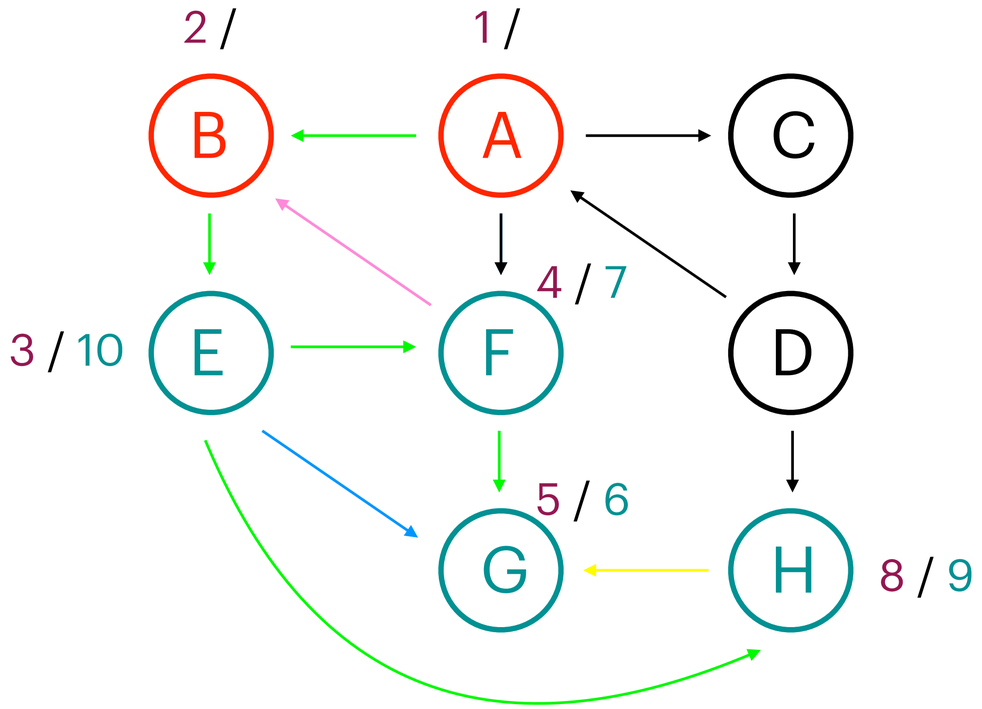
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

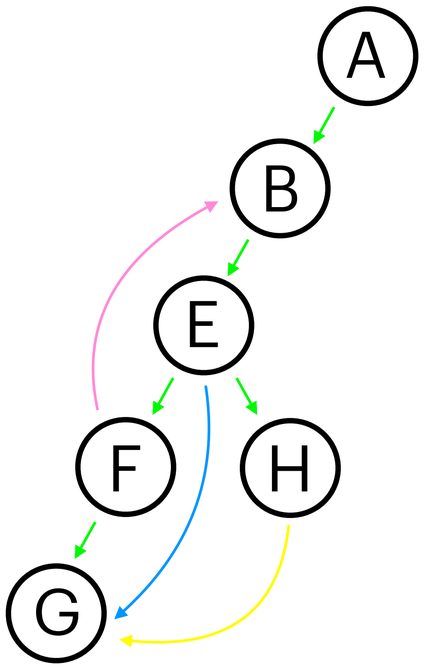
back edge

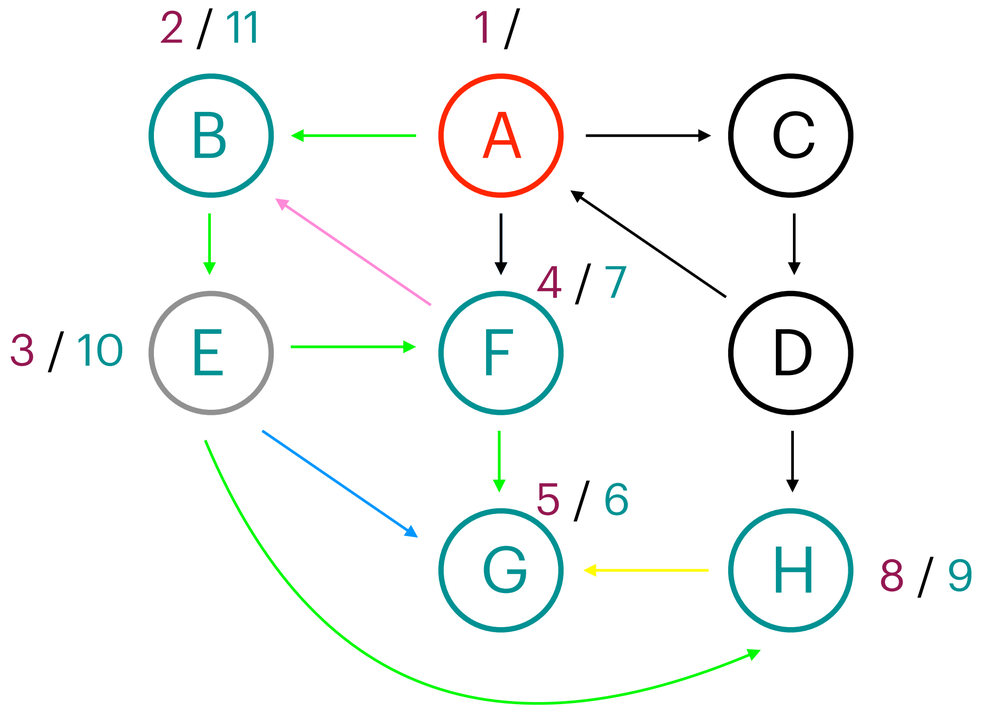
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

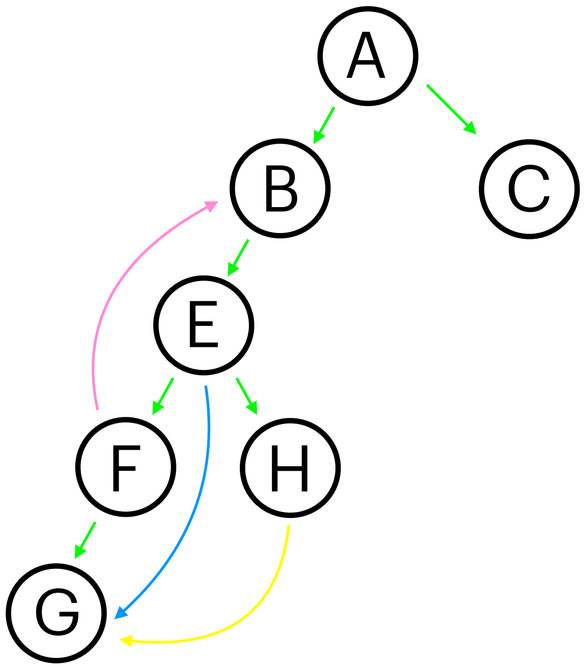
back edge

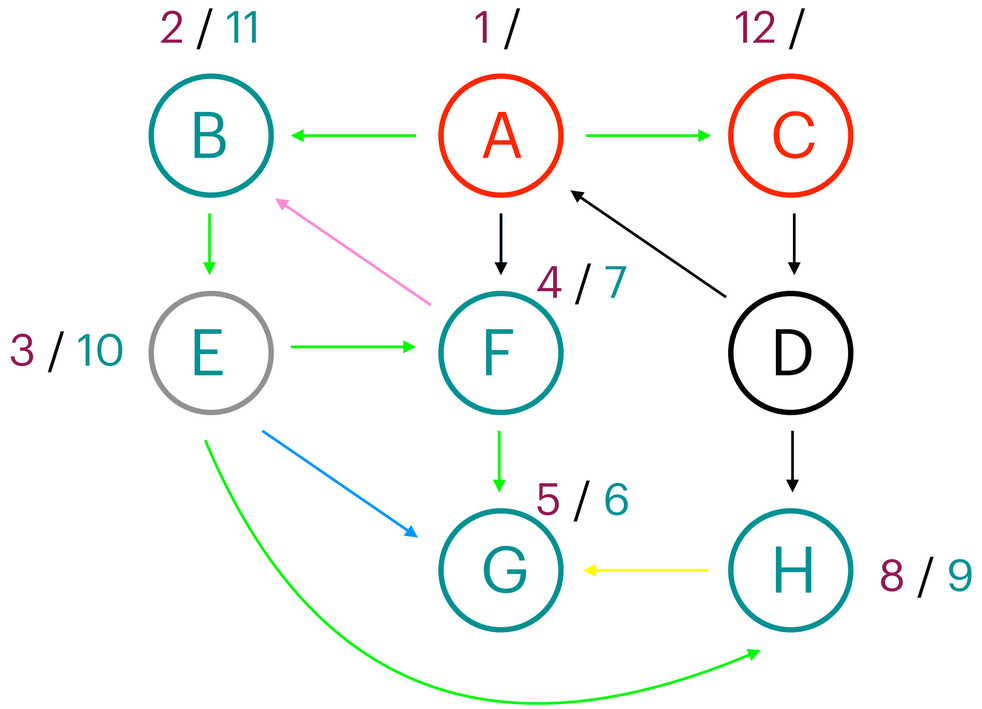
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

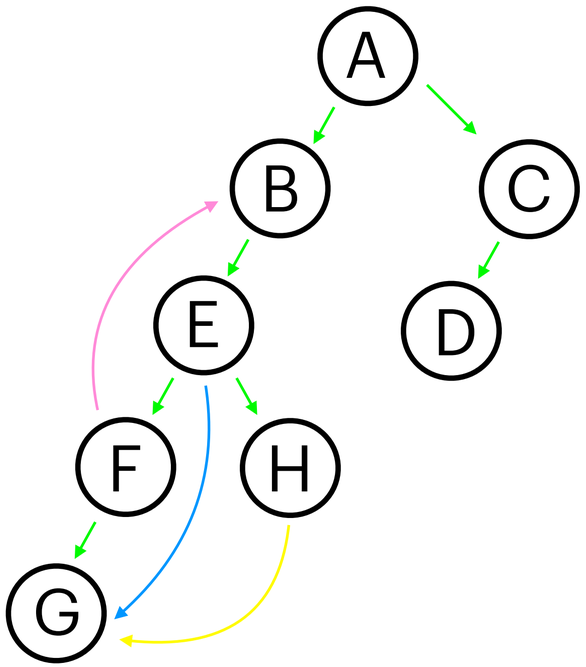
back edge

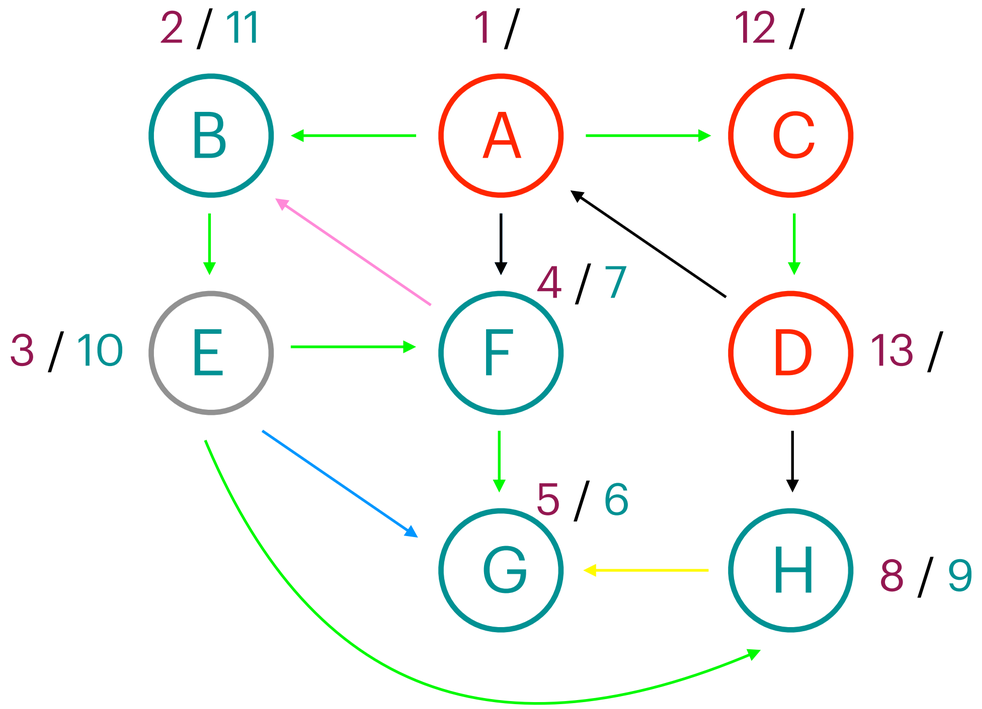
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

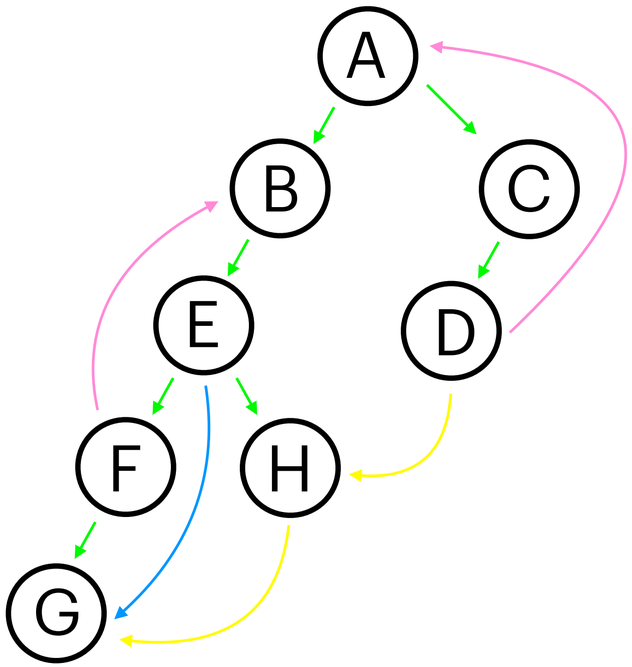
back edge

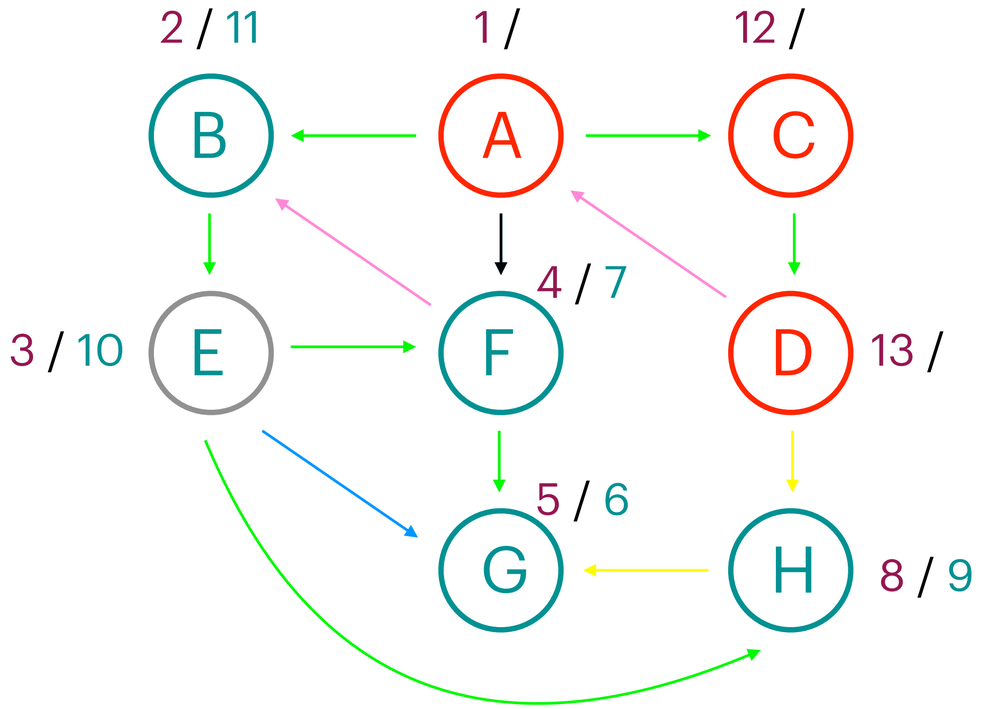
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

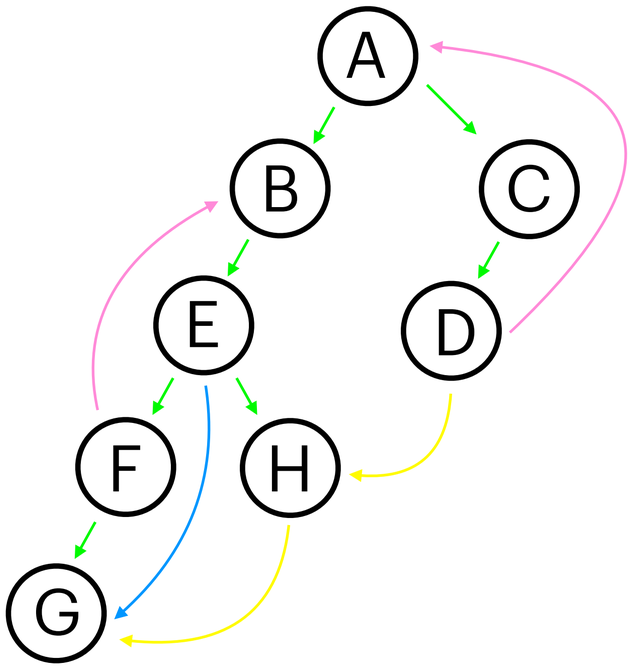
back edge

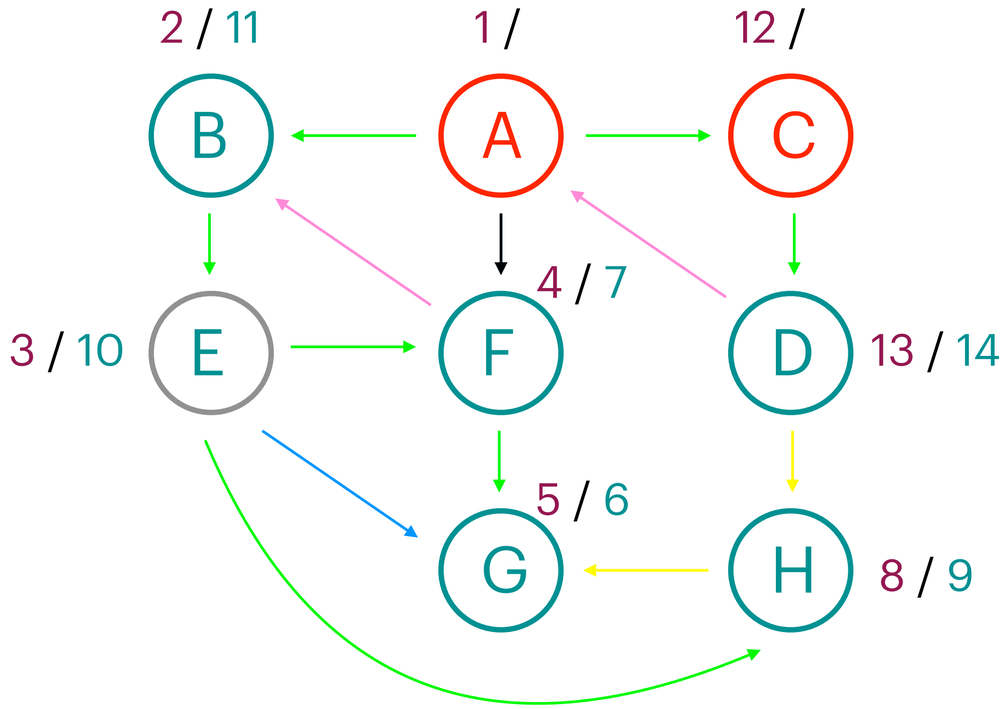
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

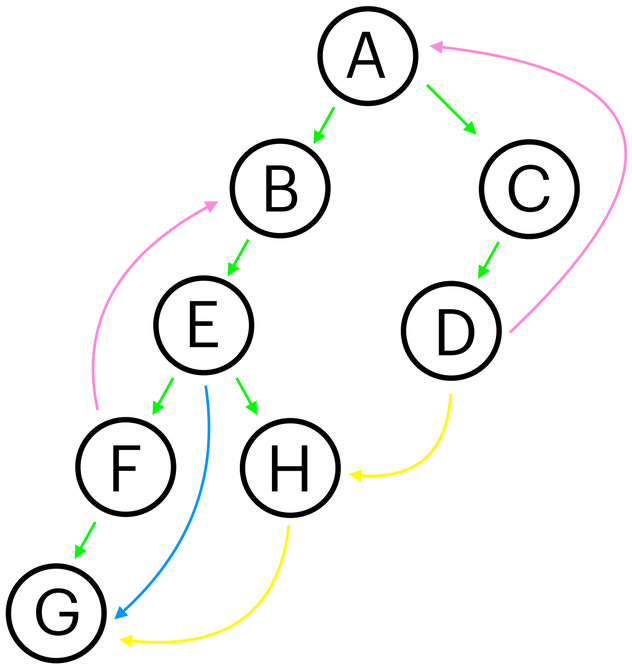
back edge

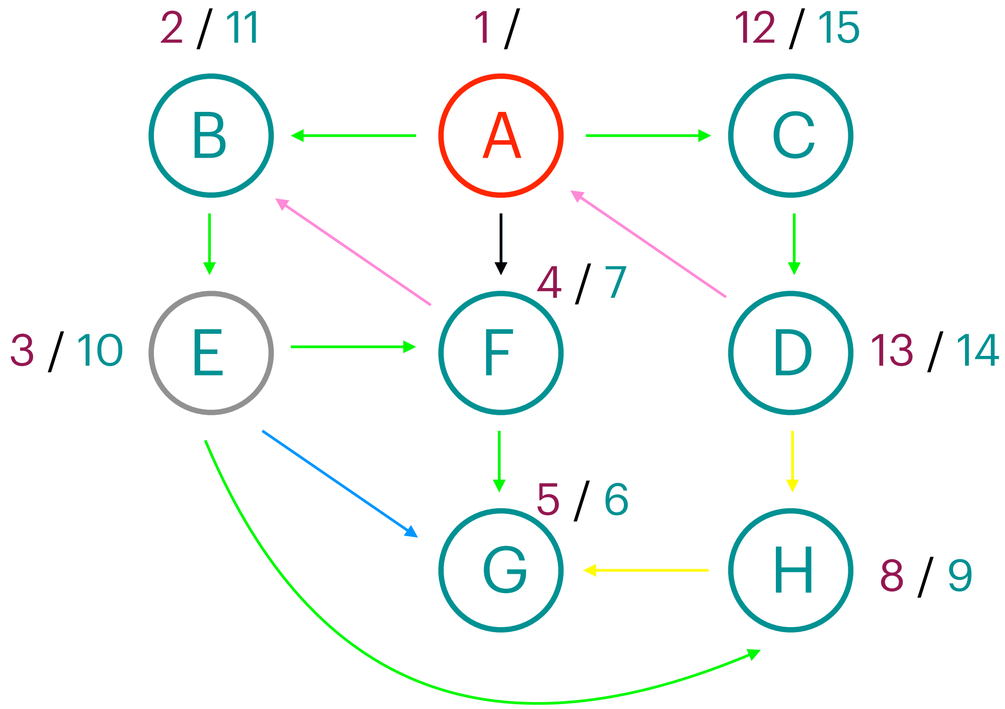
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

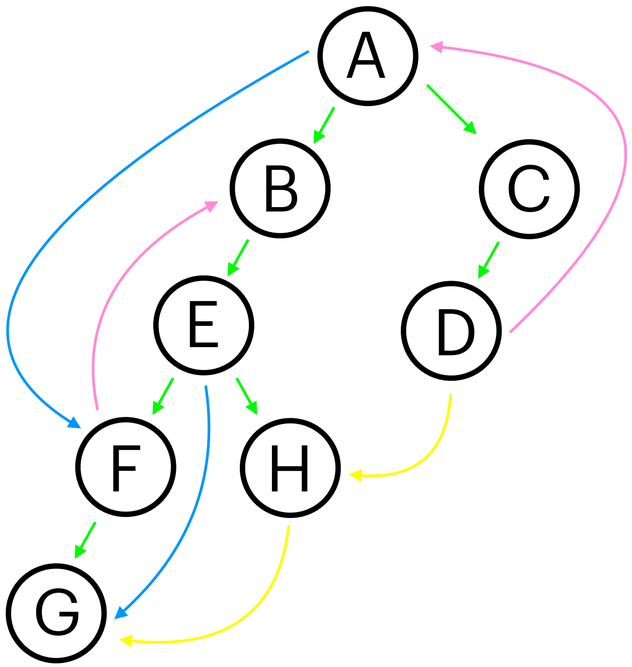
back edge

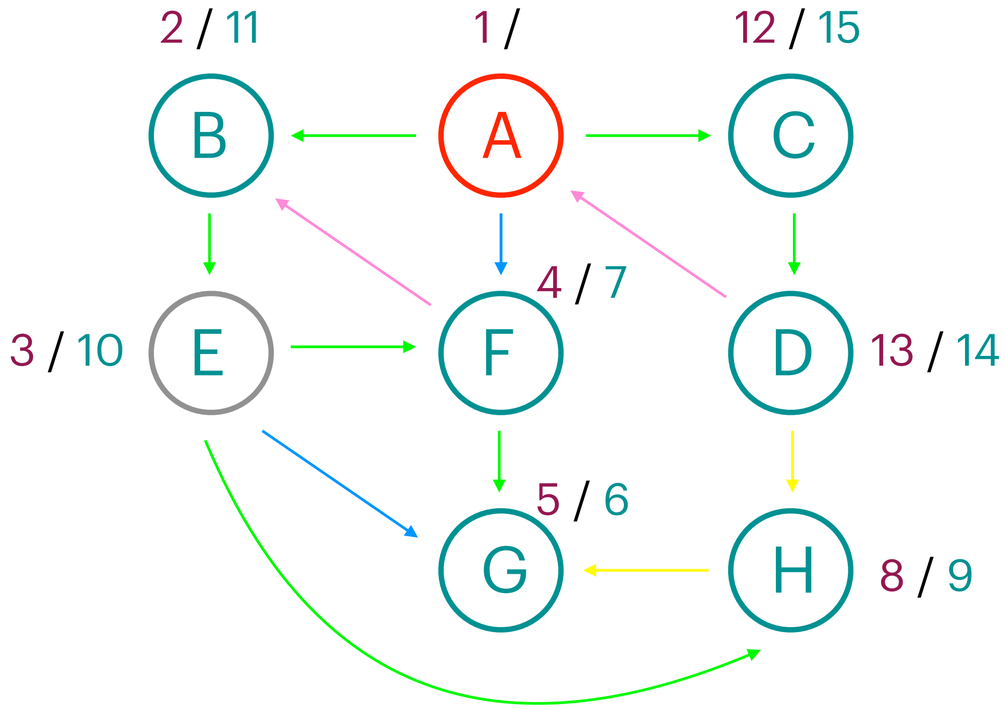
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

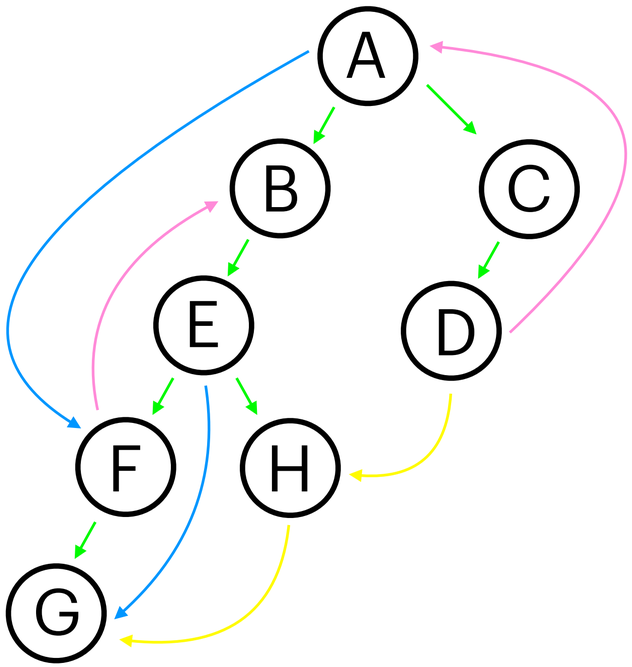
back edge

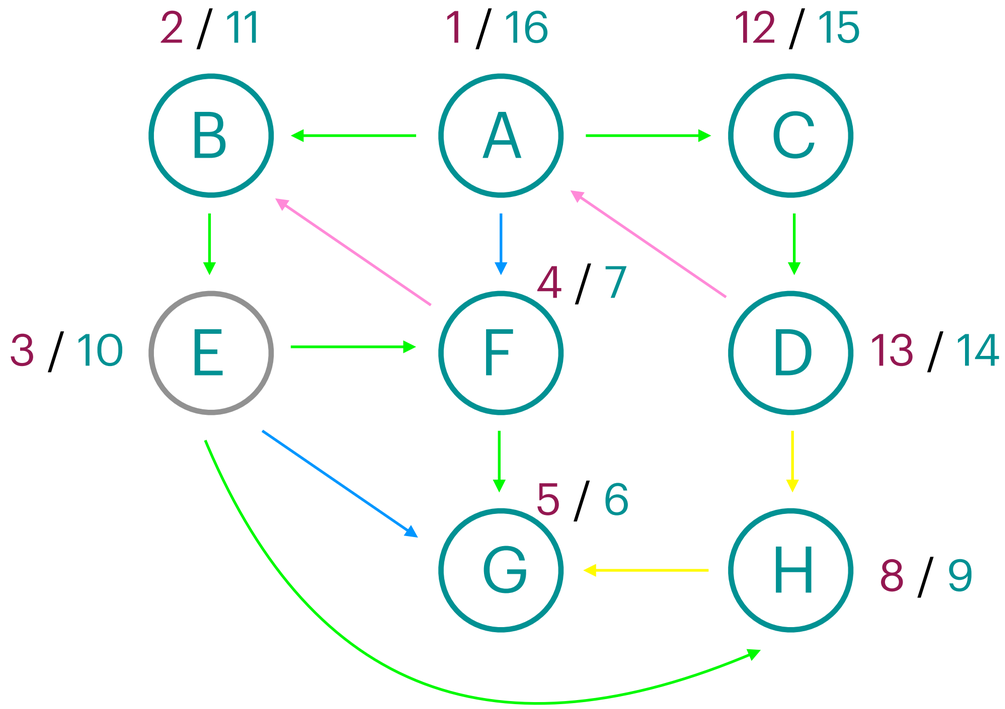
forward edge

cross edge

Graph Searchs

DFS - Example





pre-order

post-order

tree edge

back edge

forward edge

cross edge

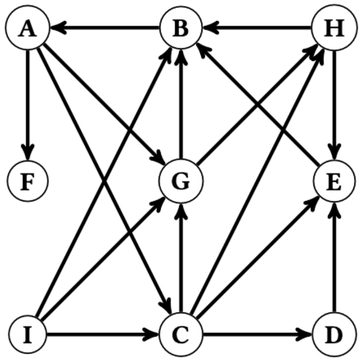
Graph Searchs

DFS - Exercise Sheet Question

Exercise 10.2 *Depth-first search* **(1 point)**.

Execute a depth-first search (*Tiefensuche*) on the following graph. Use the algorithm presented in the lecture. Always do the calls to the function "visit" in alphabetical order, i.e. start the depth-first search

from A and once "visit(A)" is finished, process the next unmarked vertex in alphabetical order. When processing the neighbors of a vertex, also process them in alphabetical order.



(a) Mark the edges that belong to the depth-first forest (*Tiefensuchwald*) with a "T" (for tree edge).

(b) For each vertex in the depth-first forest, give its *pre-* and *post*-number.

(c) Give the vertex ordering that results from sorting the vertices by pre-number. Give the vertex ordering that results from sorting the vertices by post-number.

(d) Mark every forward edge (*Vorwärtskante*) with an "F", every backward edge (*Rückwärtskante*) with a "B", and every cross edge (*Querkante*) with a "C".

(e) Does the above graph have a topological ordering? If yes, write down the topological ordering we get from the above execution of depth-first search; if no, argue how we can use the above execution of depth-first search to find a directed cycle.

(f) Draw a scale from 1 to 18, and mark for every vertex v the interval I_v from pre-number to post-number of v. What does it mean if $I_u \subset I_v$ for two different vertices u and v?

(g) Consider the graph above where the edge from B to A is removed and an edge from F to I is added. How does the execution of depth-first search change? Does the graph have a topological ordering? If yes, write down the topological ordering we get from the execution of depth-first search; if no, argue how we can use the execution of depth-first search to find a directed cycle. If you sort the vertices by *pre-number*, does this give a topological sorting?

Graph Searchs

DFS - Lemmas, Facts

$\exists$ a back edge $\iff$ $\exists$ a directed cycle

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

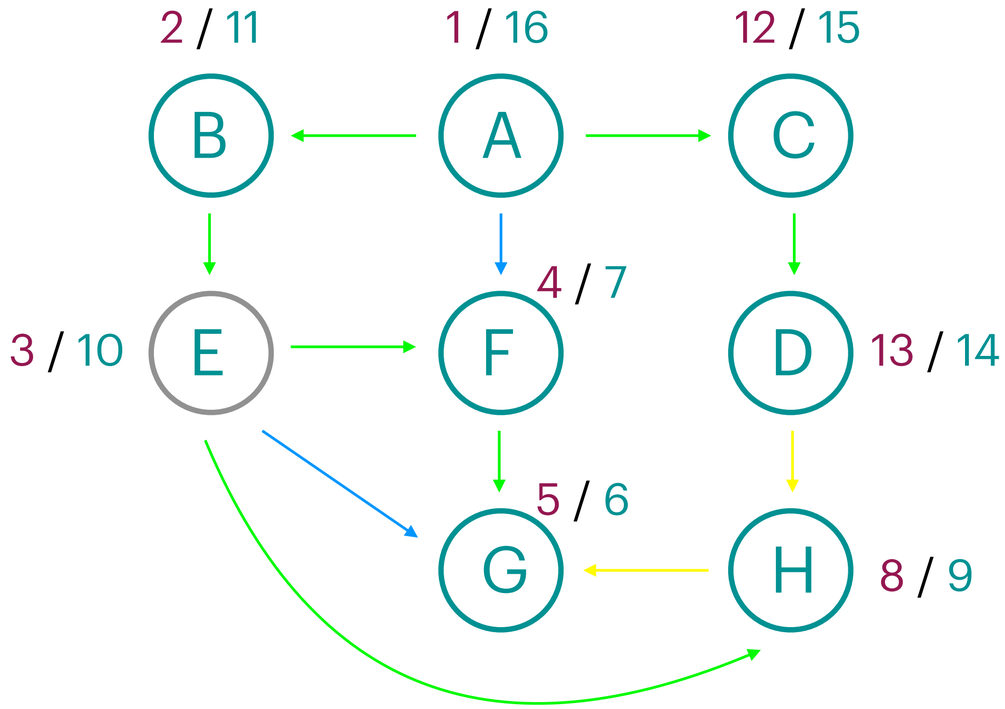
post-order :

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).

$\exists$ a topological sorting $\iff$ G is a DAG

(Directed Acyclic Graph)

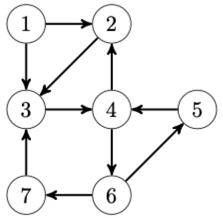
Topological Sorting doesn’t have to be unique, there can be multiple valid orders depending on the graph's structure.

DFS , Topological Sorting

HS23 , HS22

Exam Questions

/ 2 P d) *Depth-first search:* Consider the following directed graph:

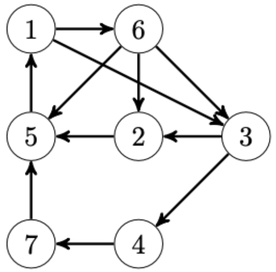


i) Draw the depth-first tree resulting from a depth-first search starting from vertex 1. Process the neighbors of a vertex in increasing order.

ii) Write out two edges e_1, e_2 such that the directed graph above has a topological ordering after removing e_1 and e_2 (the vertex set does not change).

Remark: There could be multiple valid solutions. In this case, you only need to write down one of them.

/ 2 P d) *Depth-first search:* Consider the following directed graph:



i) Draw the depth-first tree resulting from a depth-first search starting from vertex 1. Process the neighbors of a vertex in increasing order.

ii) Write out all the cross edges and all the back edges (specify which ones are cross edges, and which ones are back edges).

Topological Sorting

Exam Question

HS21

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.

Next Week...

BFS

DFS + BFS Code Example !!

DP Mini Exam - Proof

Questions

Feedbacks , Recommendations

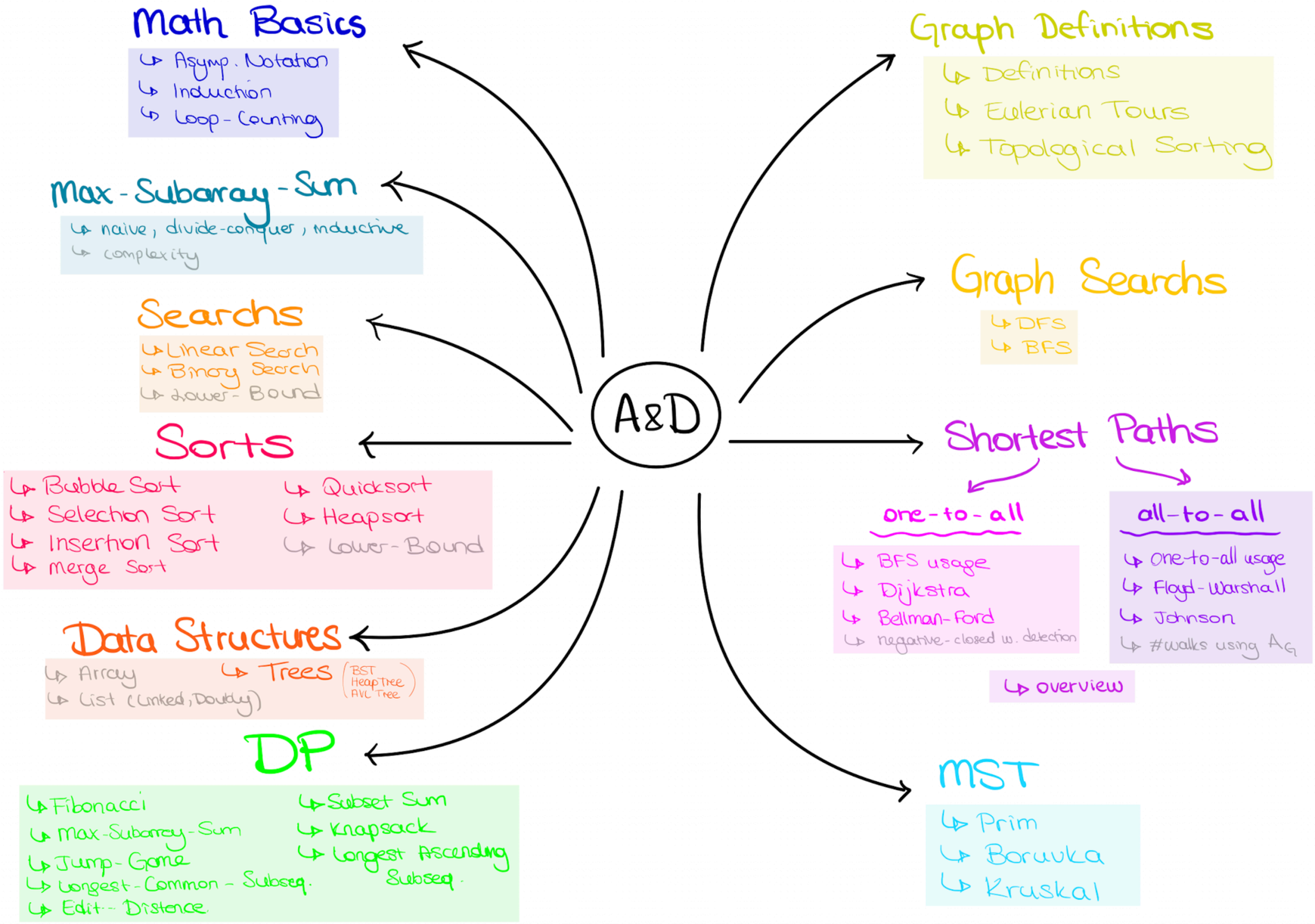
Nil Ozer

A&D

Exercise Session 10

Nil Ozer

A&D Overview


A&D
Math Basics
↳ 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

Outline

- Quiz
- Exercise Sheets

- BFS

- Code Expert - Graph Sets

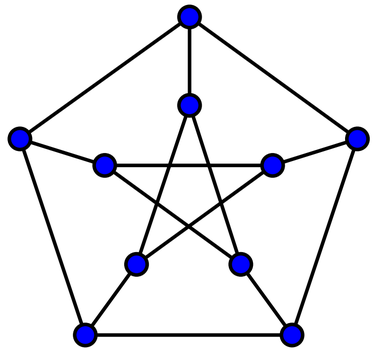
Quiz

Exercise Sheet 8

Bonus Feedback

- 8.1 : If it says translate the conditions, translate the conditions ! 👏
- 8.5 : 👏 👏 👏
- For all v ∃ hamiltonian path starting with v $\not\Rightarrow$ There is a hamiltonian cycle

  Example: Petersen Graph



- length of a path = #edges
  - n vertices in a path means that the path has a length of n -1

Peergrading and rest

- Exercise Sheet 9 peergrading
  - 9.4 this week
  - Emails will be sent

- New groups for sheet 10 !

- Sheet 7 grades coming up shortly

Graph Searchs

BFS

Graph Searchs

BFS - with pre and post order

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

Algorithm 5 BFS(s)

```
1: Q ← {s}
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
```

Q is a FIFO queue

Graph Searchs

BFS - with pre and post order + distances

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

Algorithm 5 BFS(s)

```
1: Q ← {s}
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
```

distance[s] = 0 ;

Q is a FIFO queue

distance[v] <- distance[u] + 1 ;

Graph Searchs

BFS - with pre and post order + distances

Algorithm 14 Breadth-first search

```
Q ← new queue ()
Q.PUSH (r)
D ← {r}
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}
```

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

Algorithm 5 BFS(s)

```
1: Q ← {s}
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
```

distance[s] = 0 ;

Q is a FIFO queue

distance[v] <- distance[u] + 1 ;

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

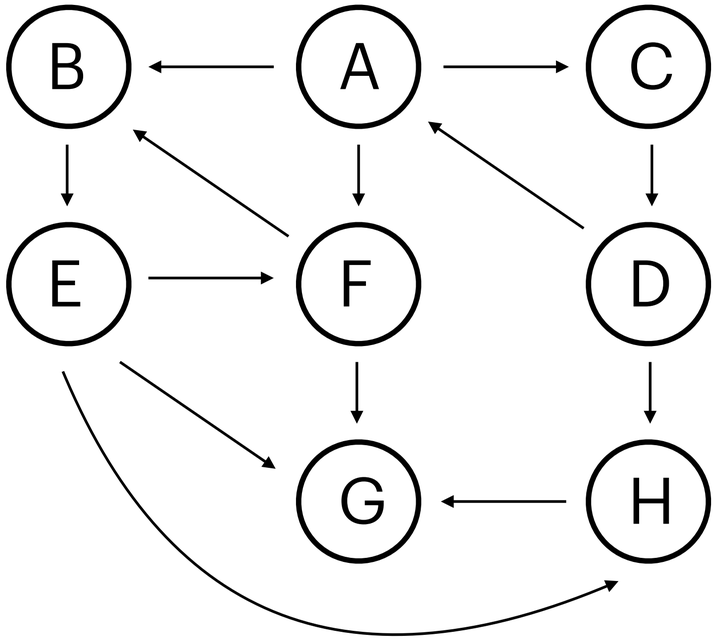
1: $Q \leftarrow \{s\}$
2: $\text{enter}[s] \leftarrow 0; \quad T \leftarrow 1$

distance[s] = 0 ;

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

Q : A

0 /



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							

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Q :

u = A

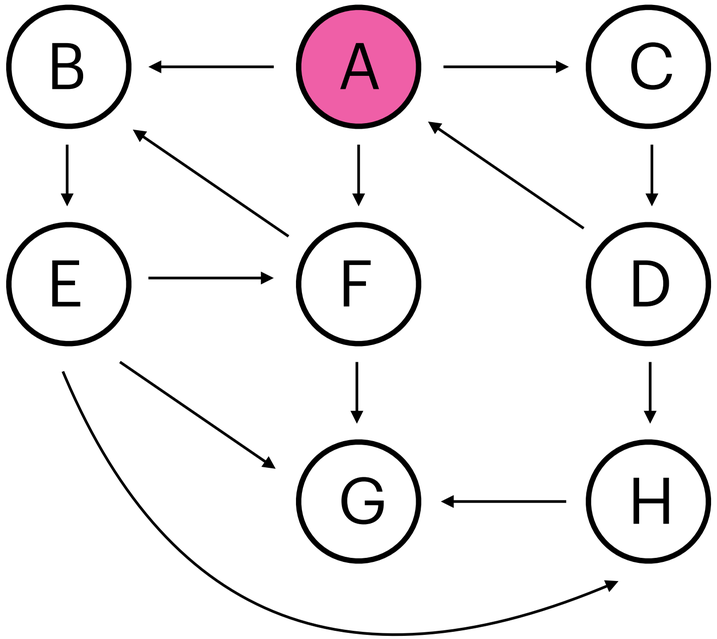
enter[] :

A	B	C	D	E	F	G	H
0							

leave[] :

A	B	C	D	E	F	G	H
1							

0 / 1



distance[] :

A	B	C	D	E	F	G	H
0							

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Q :

u = A

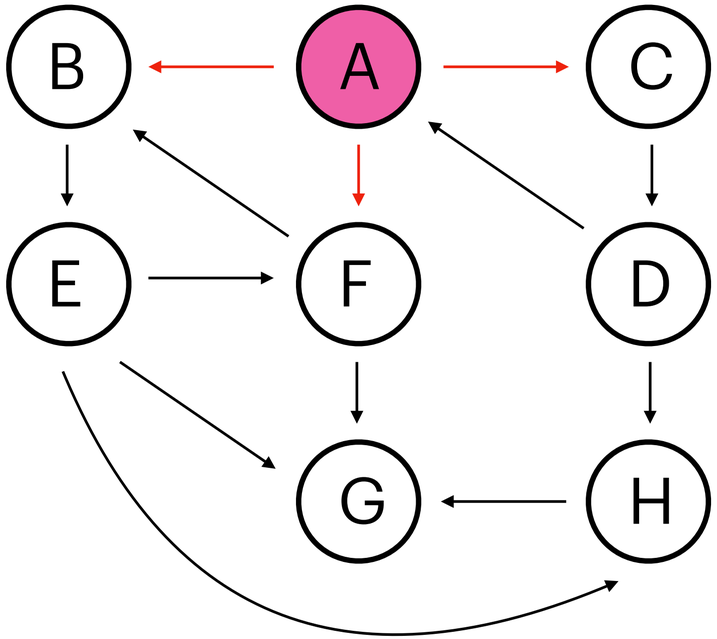
enter[] :

A	B	C	D	E	F	G	H
0							

leave[] :

A	B	C	D	E	F	G	H
1							

0 / 1



distance[] :

A	B	C	D	E	F	G	H
0							

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

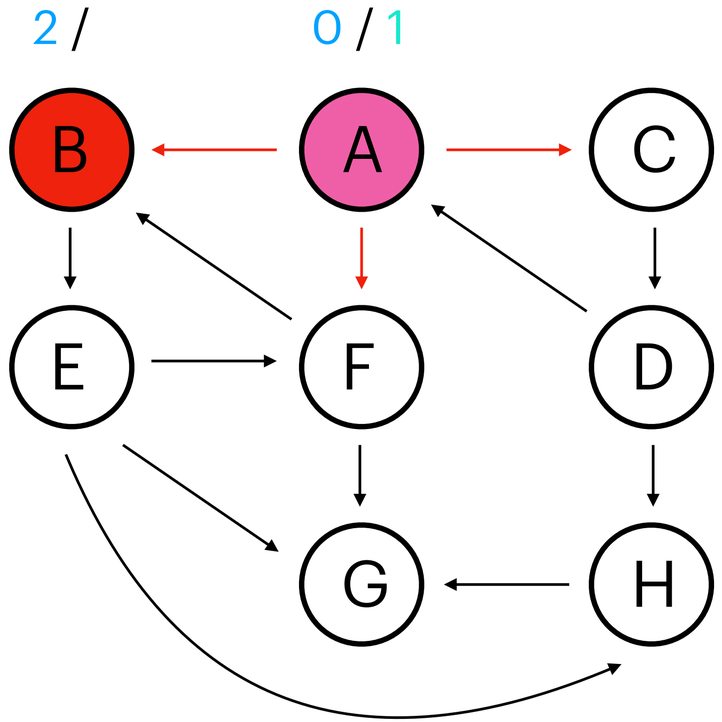
distance[s] = 0 ;

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

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						

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

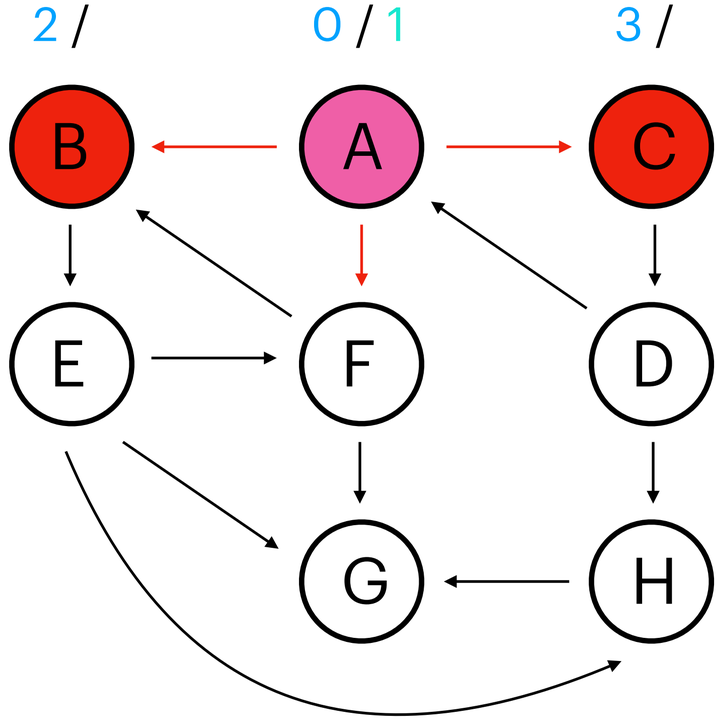
1: $Q \leftarrow \{s\}$
2: $\text{enter}[s] \leftarrow 0; \quad T \leftarrow 1$

distance[s] = 0 ;

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

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

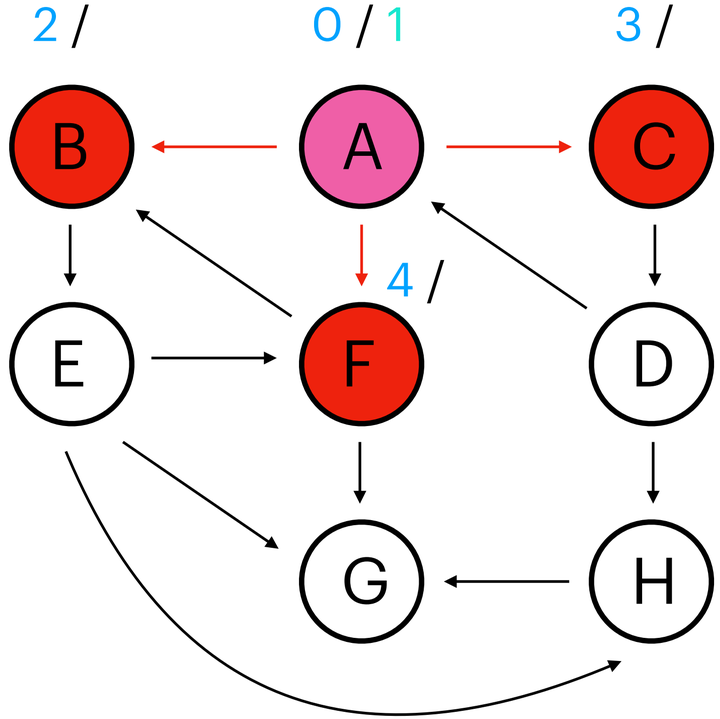
1: $Q \leftarrow \{s\}$
2: $\text{enter}[s] \leftarrow 0; \quad T \leftarrow 1$

distance[s] = 0 ;

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

Graph Searchs

BFS - Example

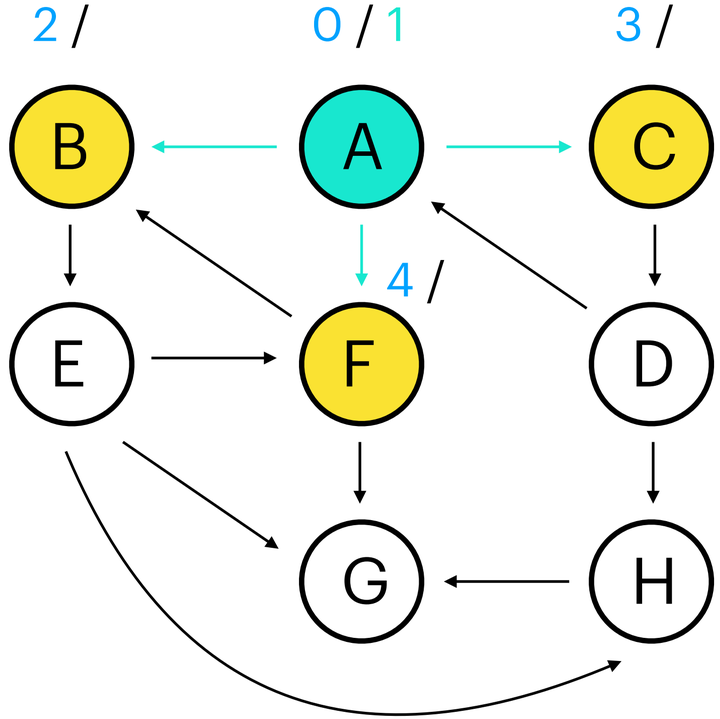
Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

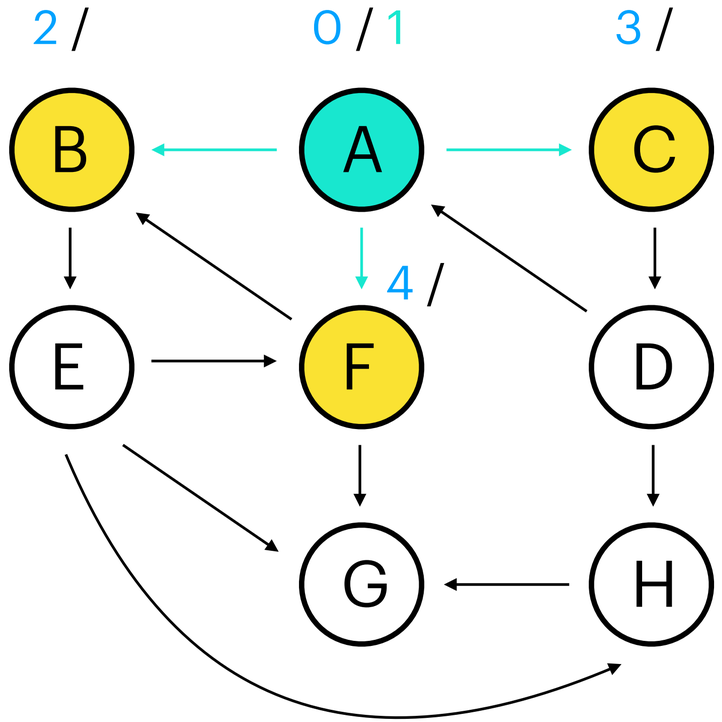
1: $Q \leftarrow \{s\}$
2: $\text{enter}[s] \leftarrow 0; \quad T \leftarrow 1$

distance[s] = 0 ;

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

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

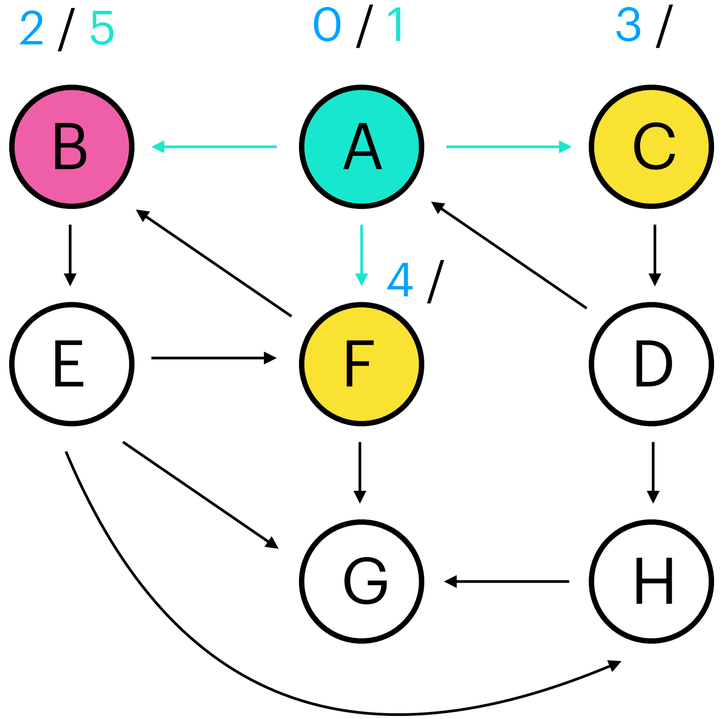
1: $Q \leftarrow \{s\}$
2: $\text{enter}[s] \leftarrow 0; \quad T \leftarrow 1$

distance[s] = 0 ;

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

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

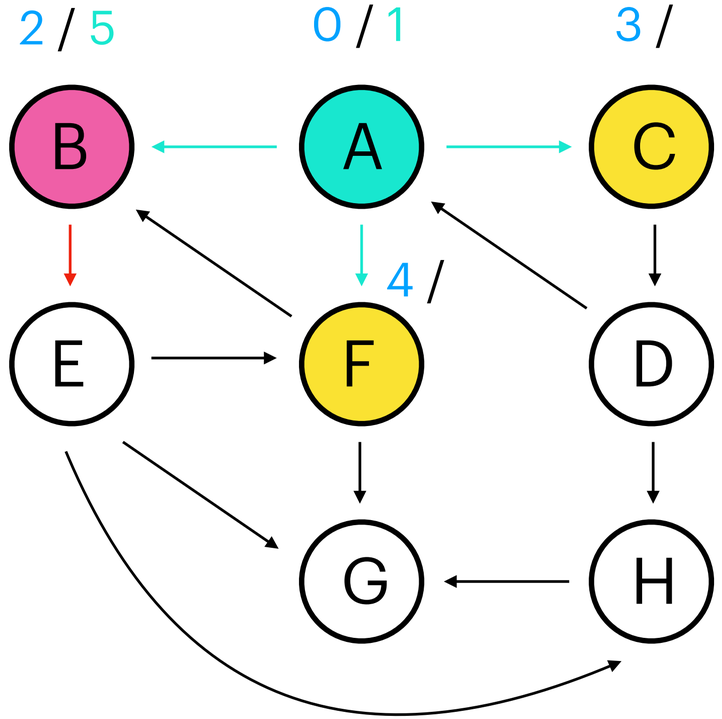
distance[s] = 0 ;

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

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		

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

distance[v] <- distance[u] + 1 ;

Q : C - F - E

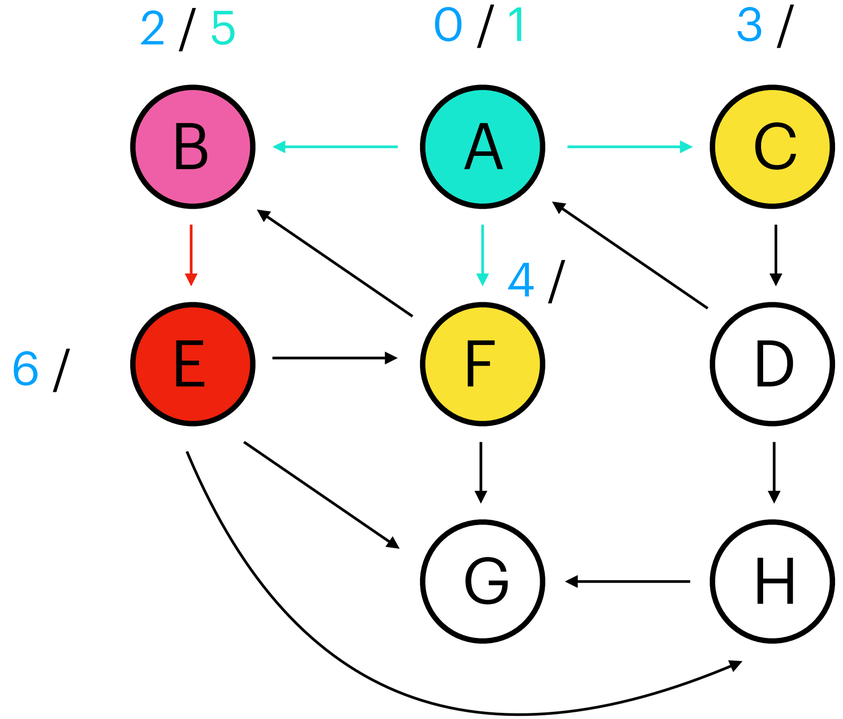
u = B

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		

Graph Searchs

BFS - Example

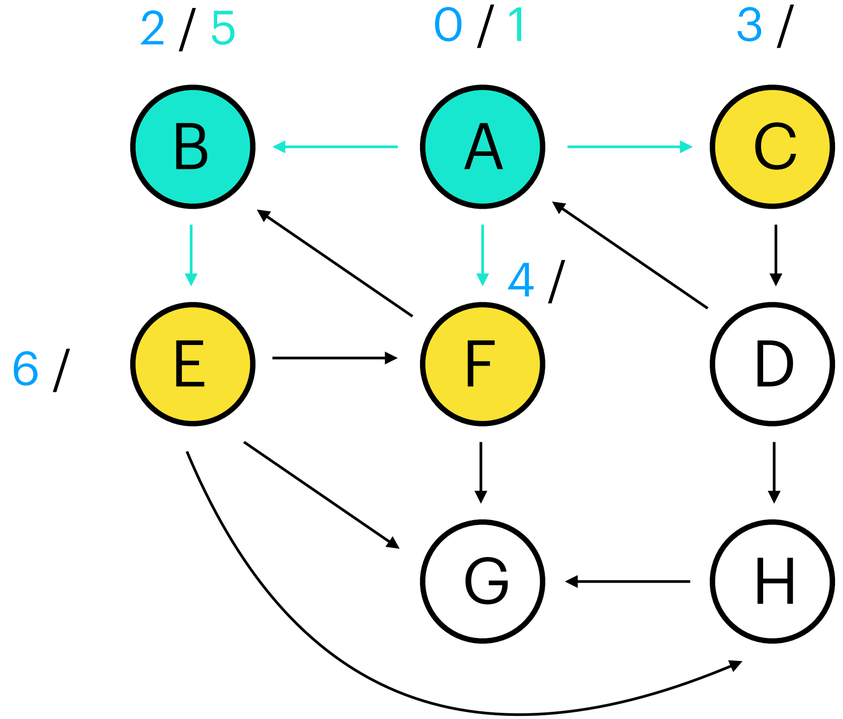
Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

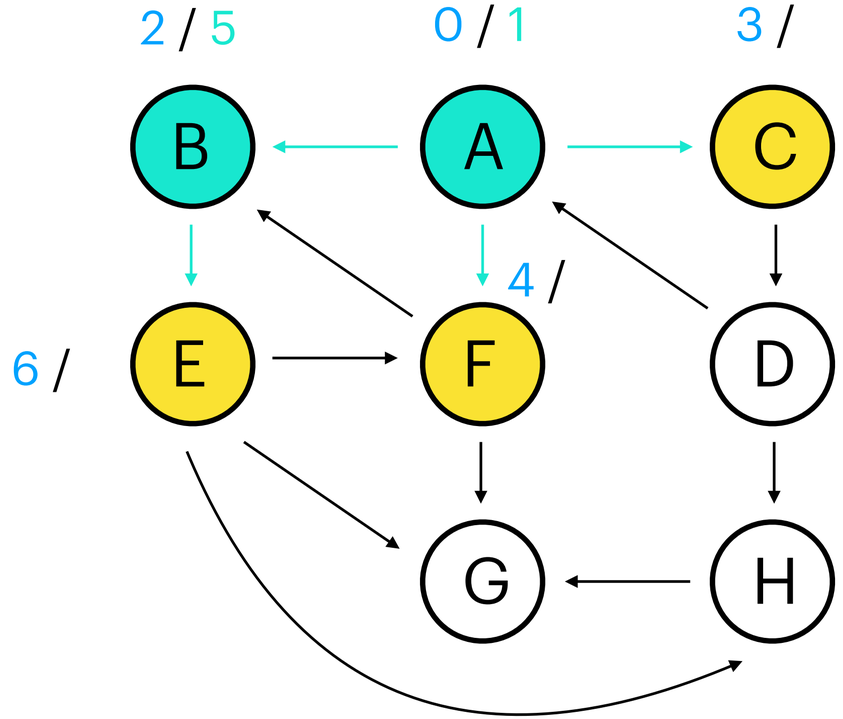
1: $Q \leftarrow \{s\}$
2: $\text{enter}[s] \leftarrow 0; \quad T \leftarrow 1$

distance[s] = 0 ;

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

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		

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

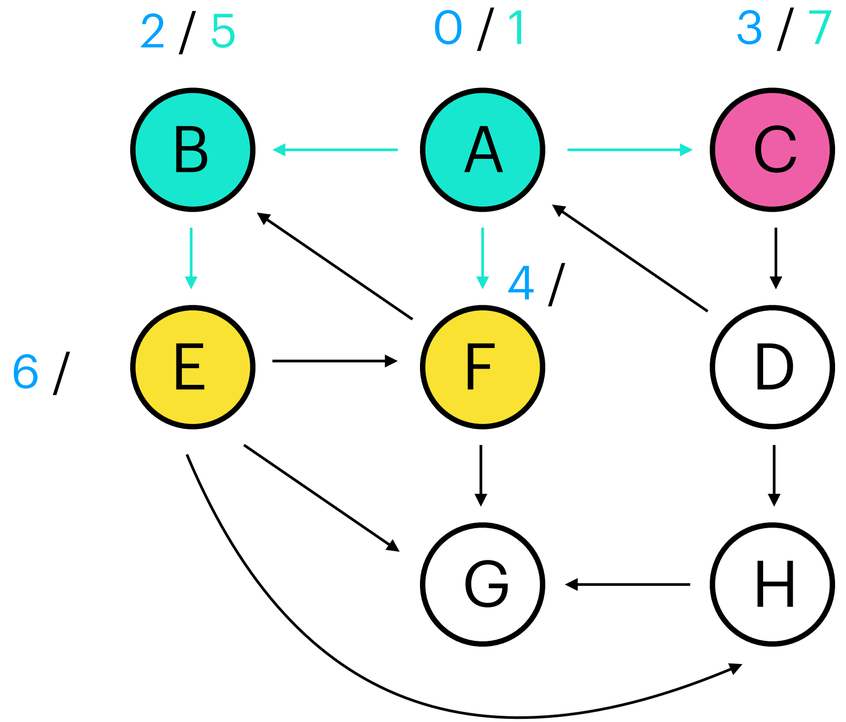
distance[s] = 0 ;

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

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		

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

distance[v] <- distance[u] + 1 ;

Q : F - E

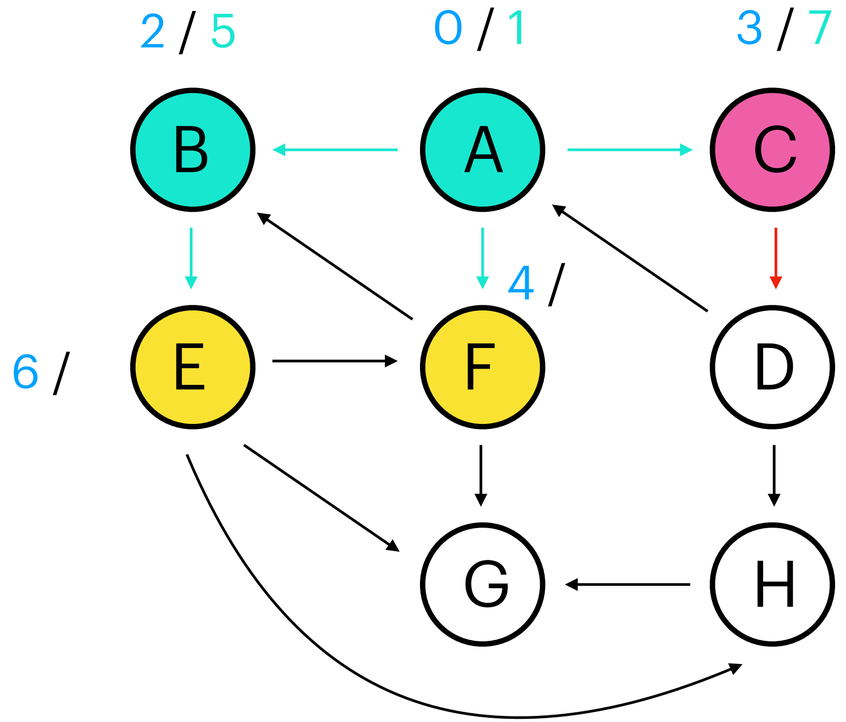
u = C

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		

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Q : F - E - D

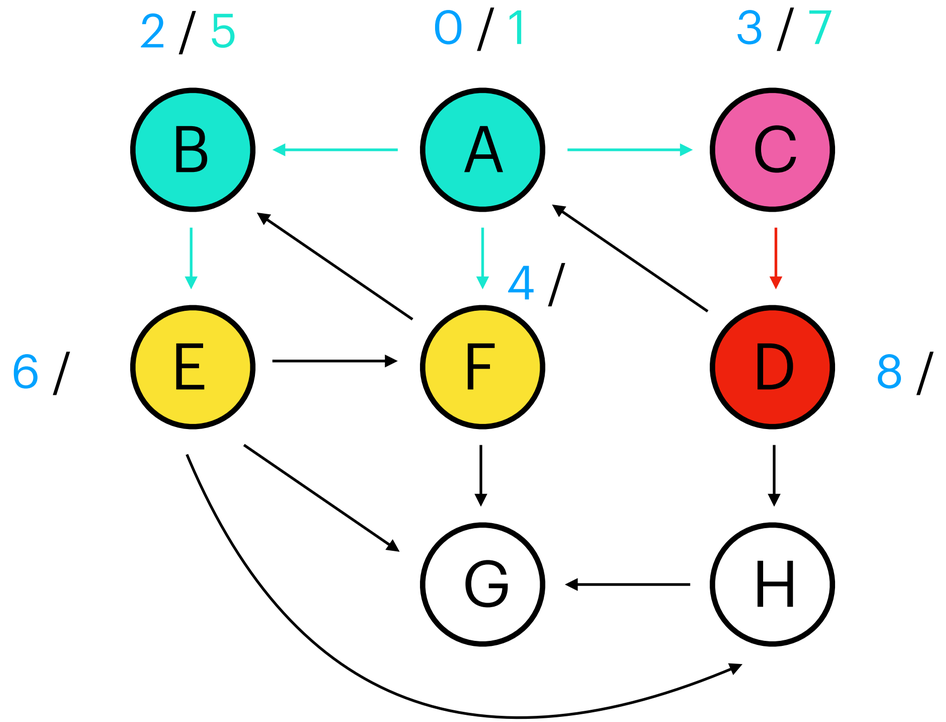
u = C

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		

Graph Searchs

BFS - Example

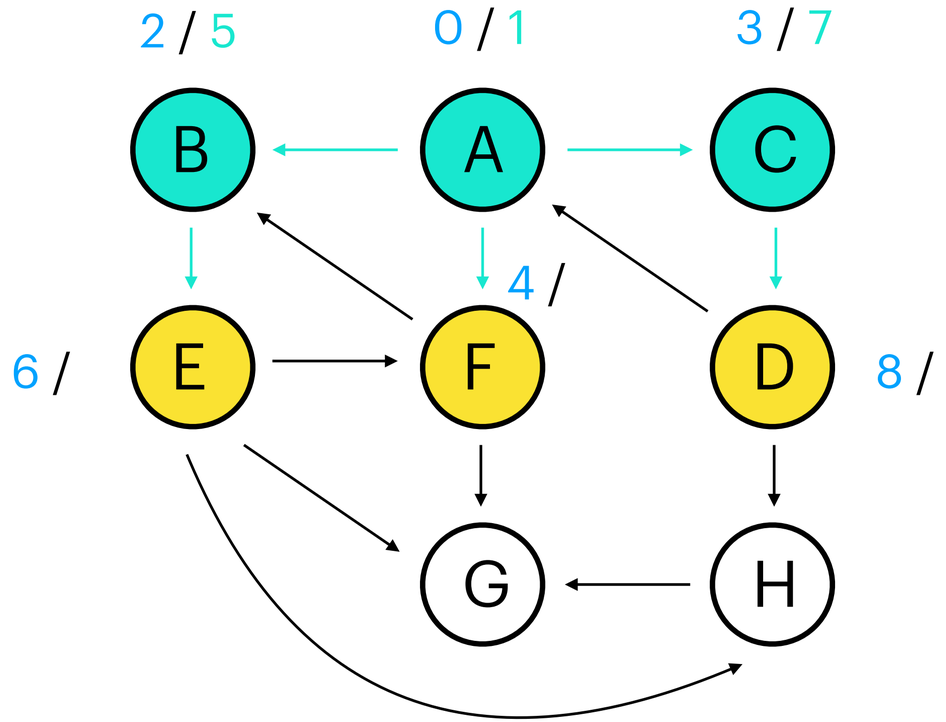
Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Graph Searchs

BFS - Example

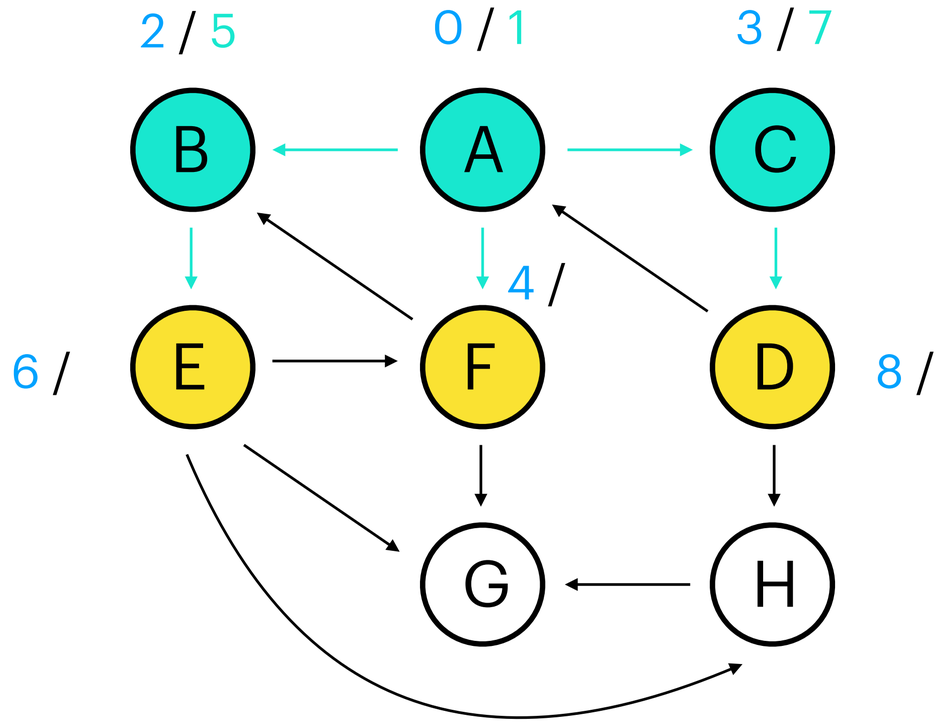
Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

distance[v] <- distance[u] + 1 ;

Q : E - D

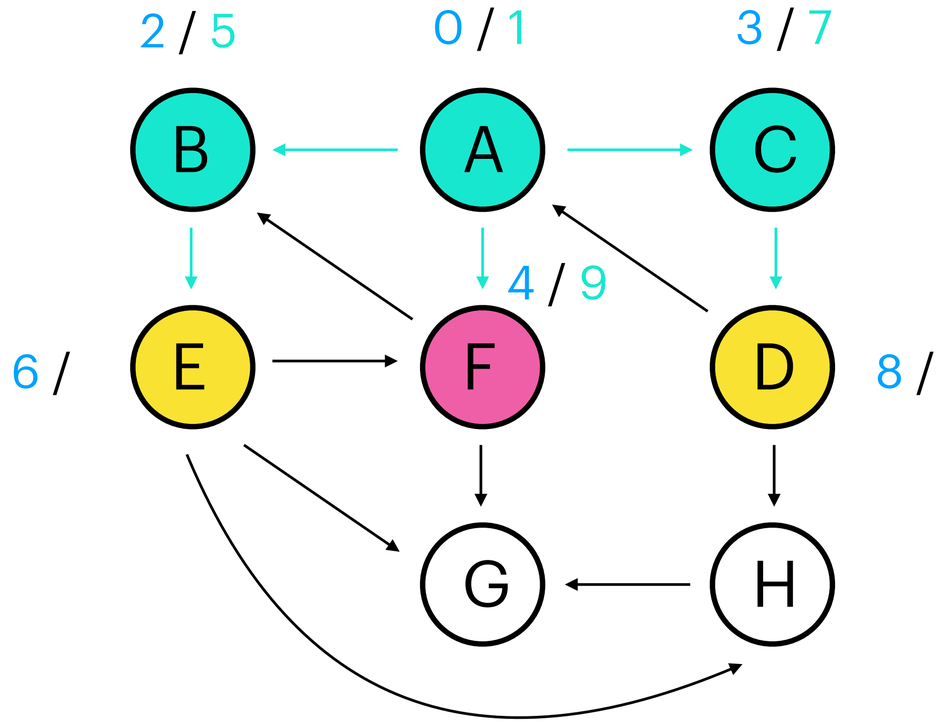
u = F

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		

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

distance[v] <- distance[u] + 1 ;

Q : E - D

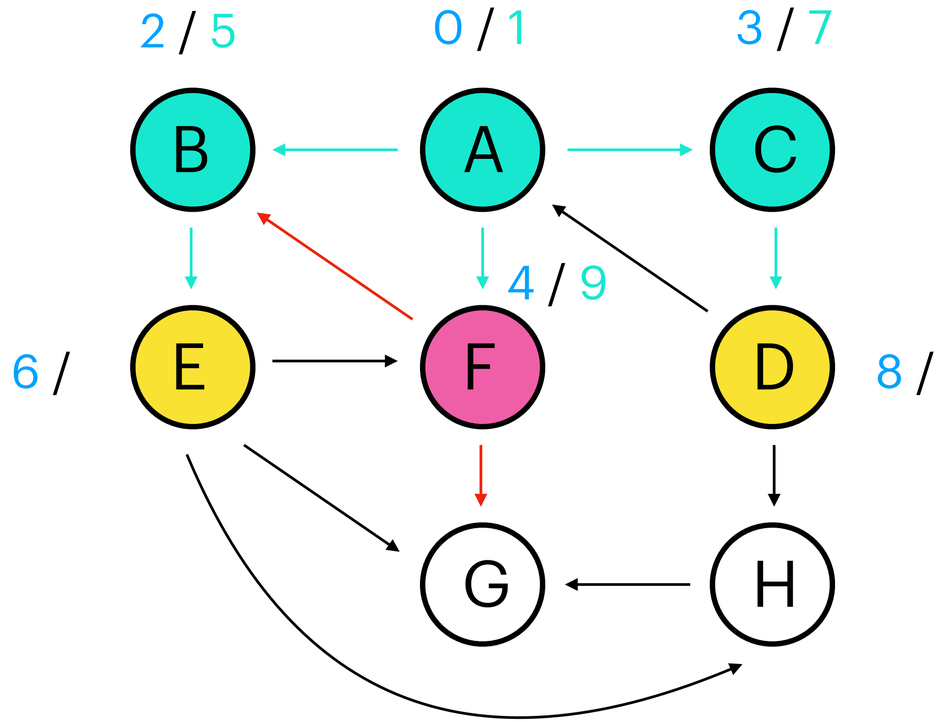
u = F

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		

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

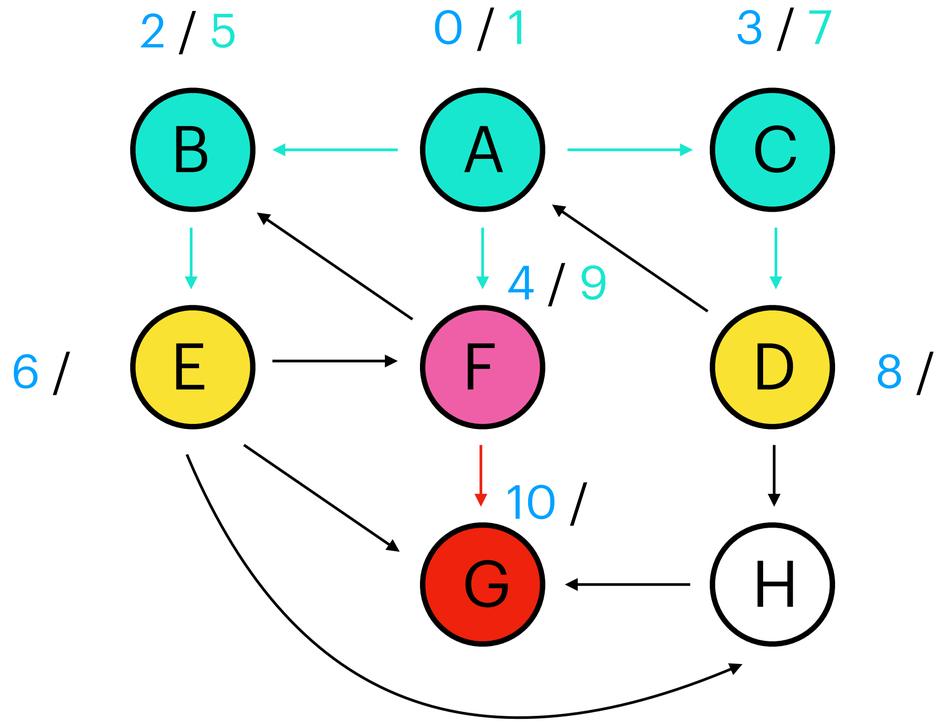
distance[s] = 0 ;

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

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	

Graph Searchs

BFS - Example

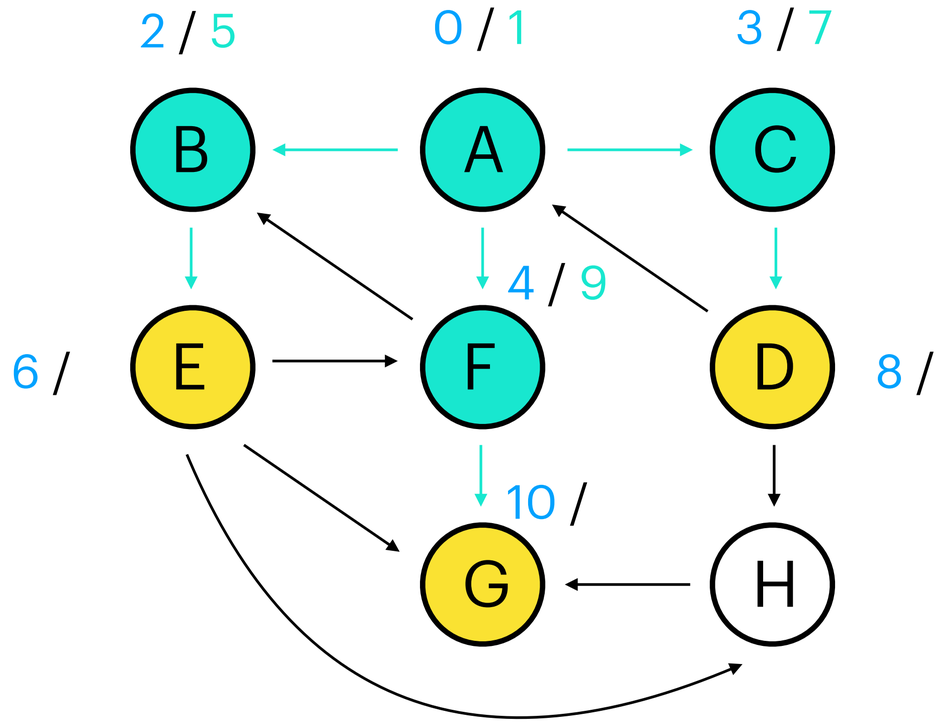
Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Q : E - D - G



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	

Graph Searchs

BFS - Example

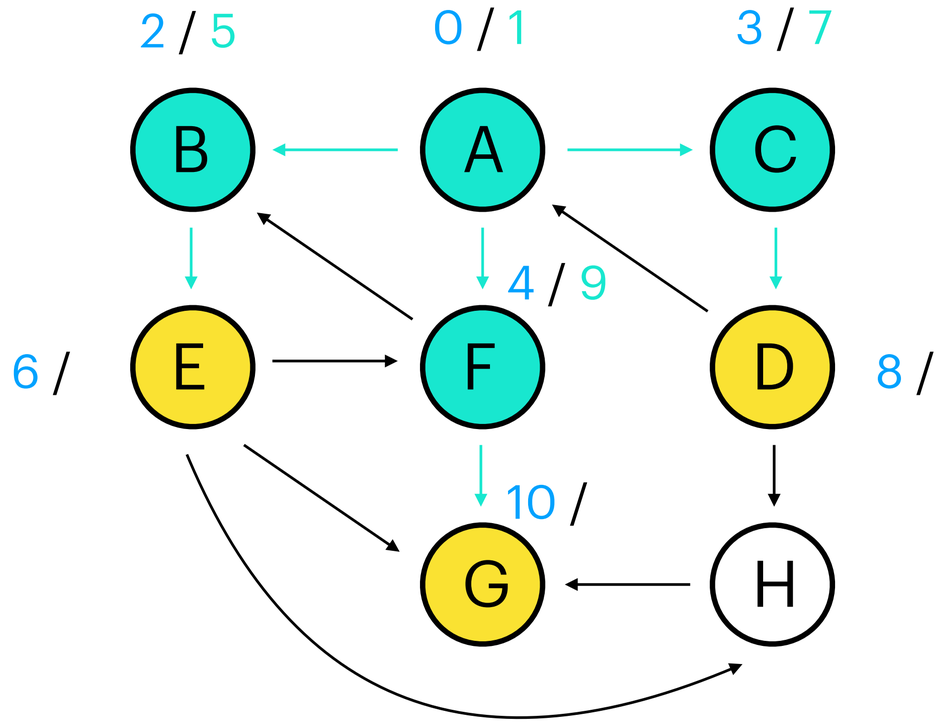
Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Q : E - D - G



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	

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Q : D - G

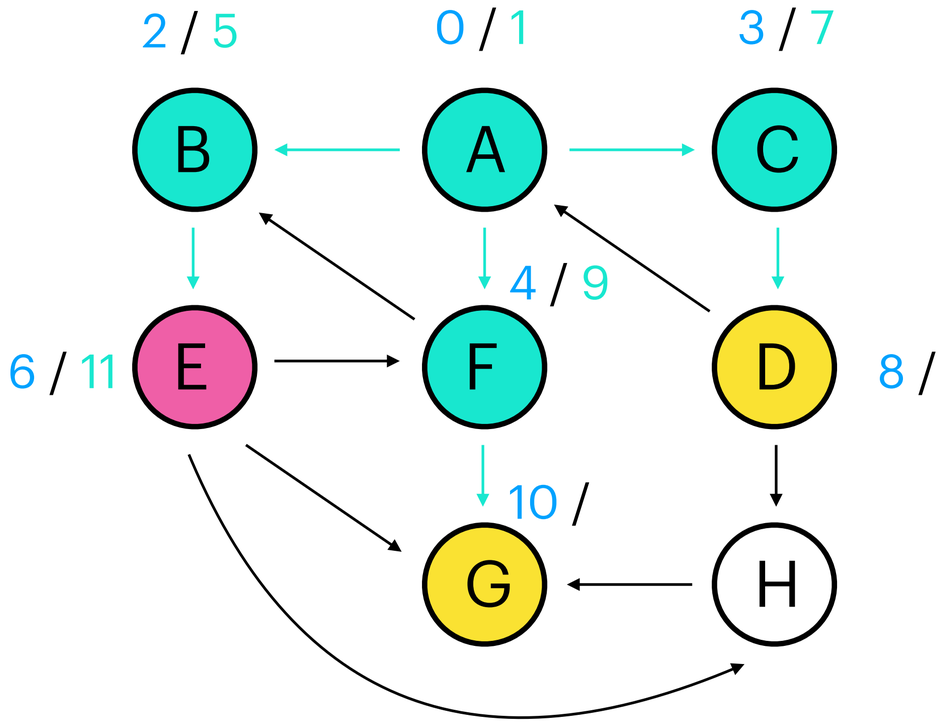
u = E

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	

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Q : D - G

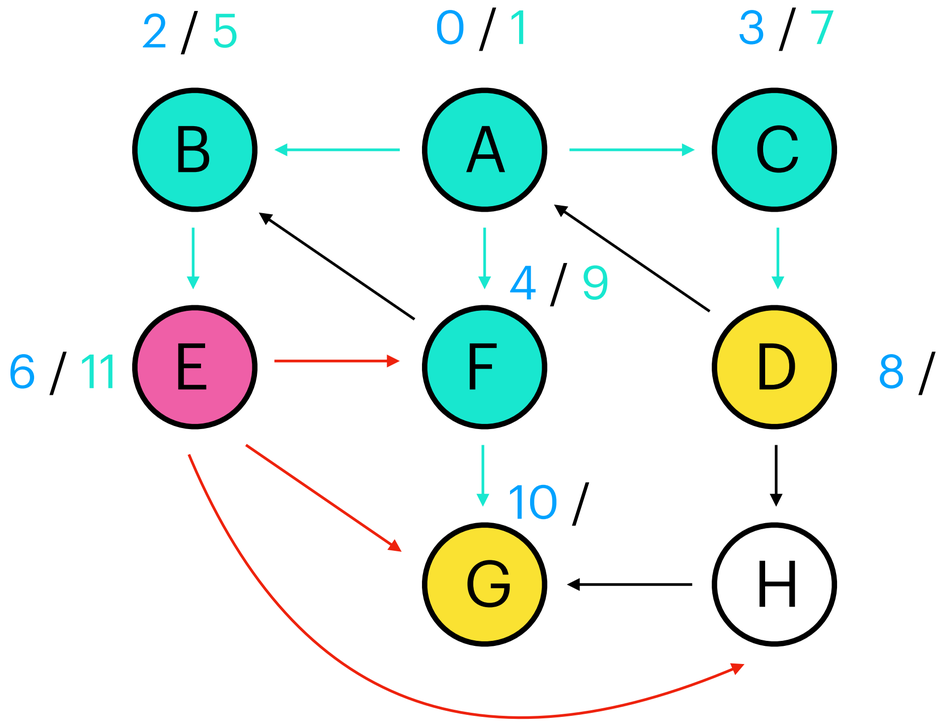
u = E

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	

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

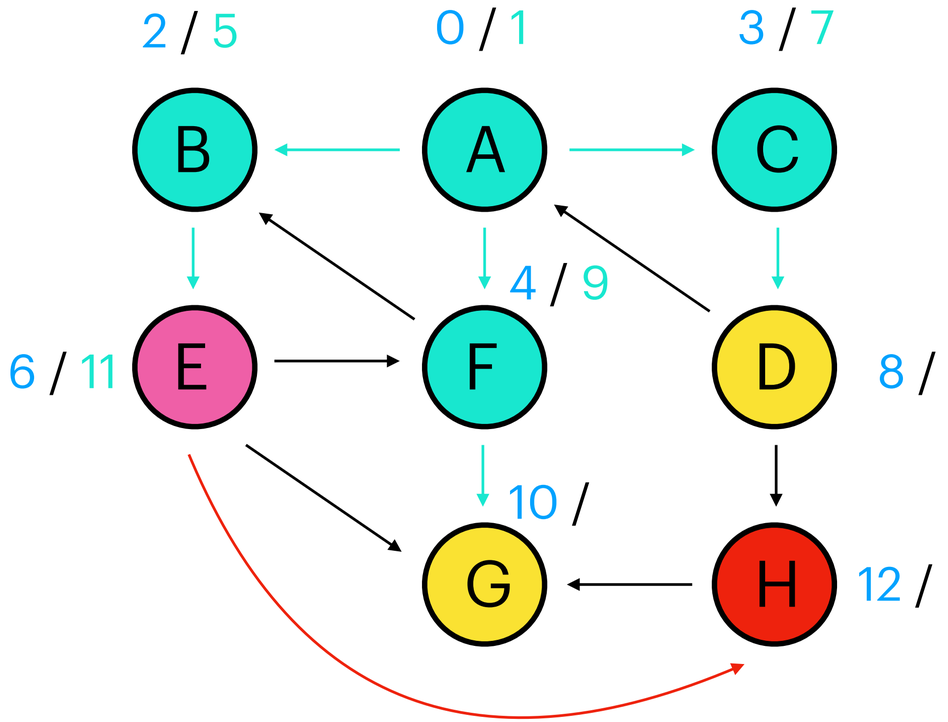
1: $Q \leftarrow \{s\}$
2: $\text{enter}[s] \leftarrow 0; \quad T \leftarrow 1$

distance[s] = 0 ;

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

Graph Searchs

BFS - Example

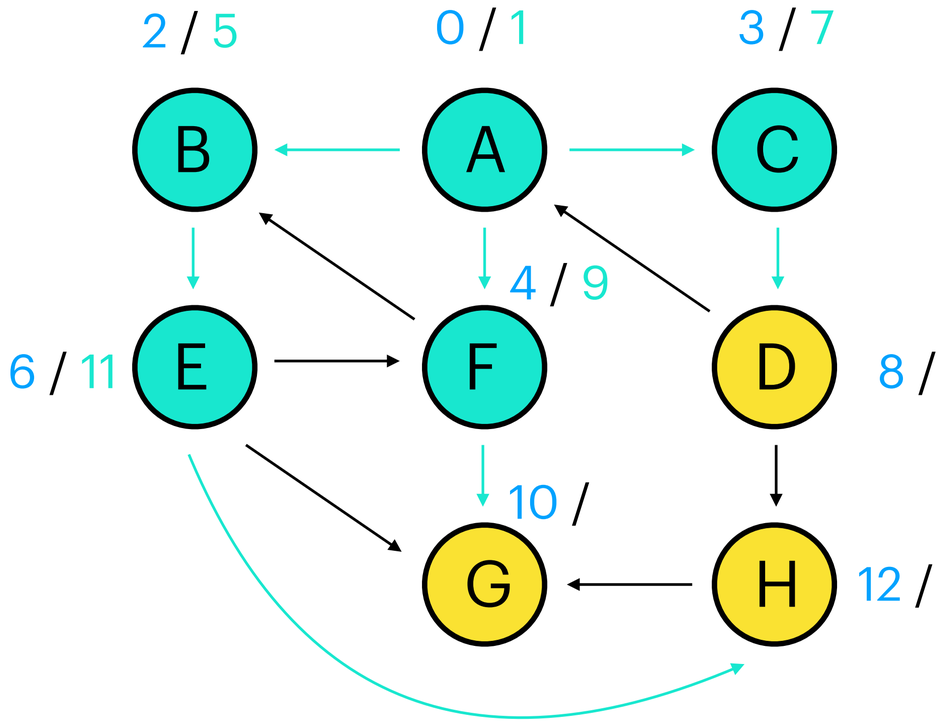
Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Graph Searchs

BFS - Example

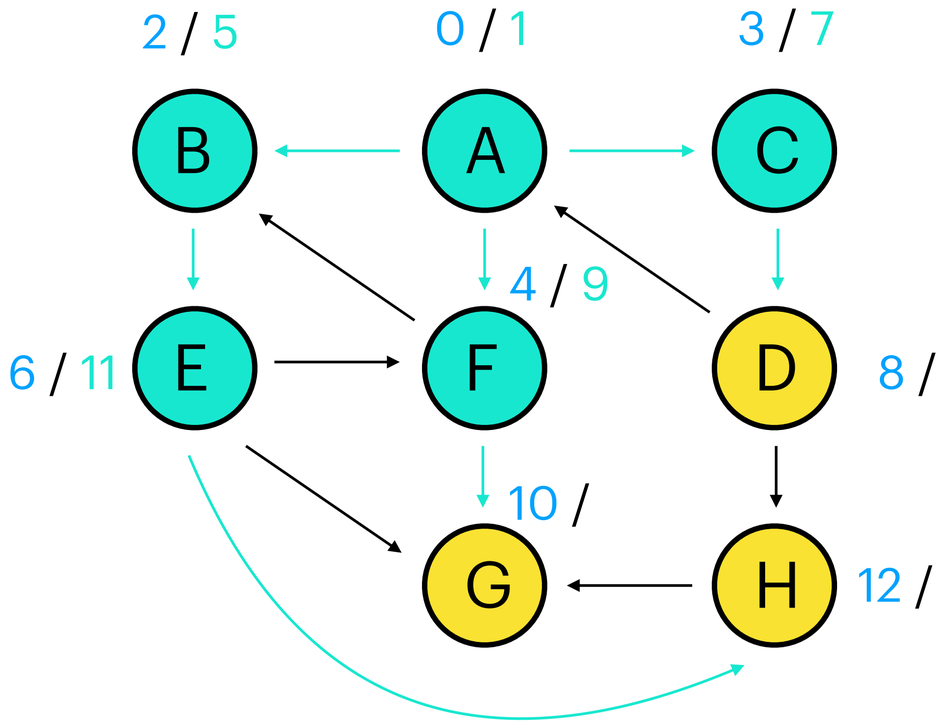
Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

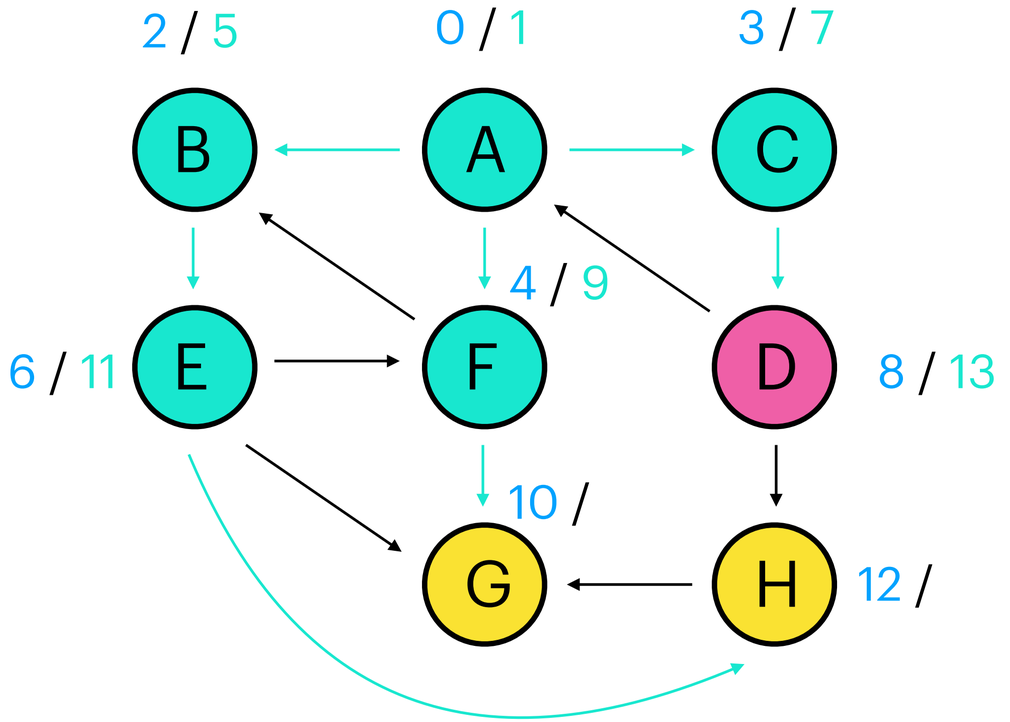
1: $Q \leftarrow \{s\}$
2: $\text{enter}[s] \leftarrow 0; \quad T \leftarrow 1$

distance[s] = 0 ;

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

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

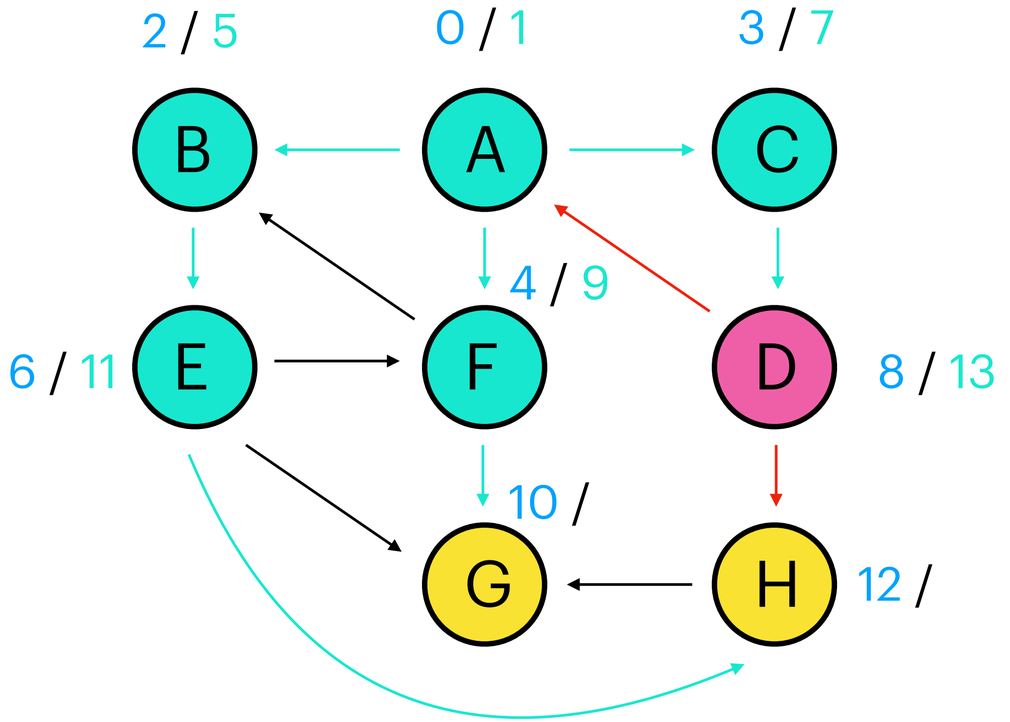
1: $Q \leftarrow \{s\}$
2: $\text{enter}[s] \leftarrow 0; \quad T \leftarrow 1$

distance[s] = 0 ;

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

Graph Searchs

BFS - Example

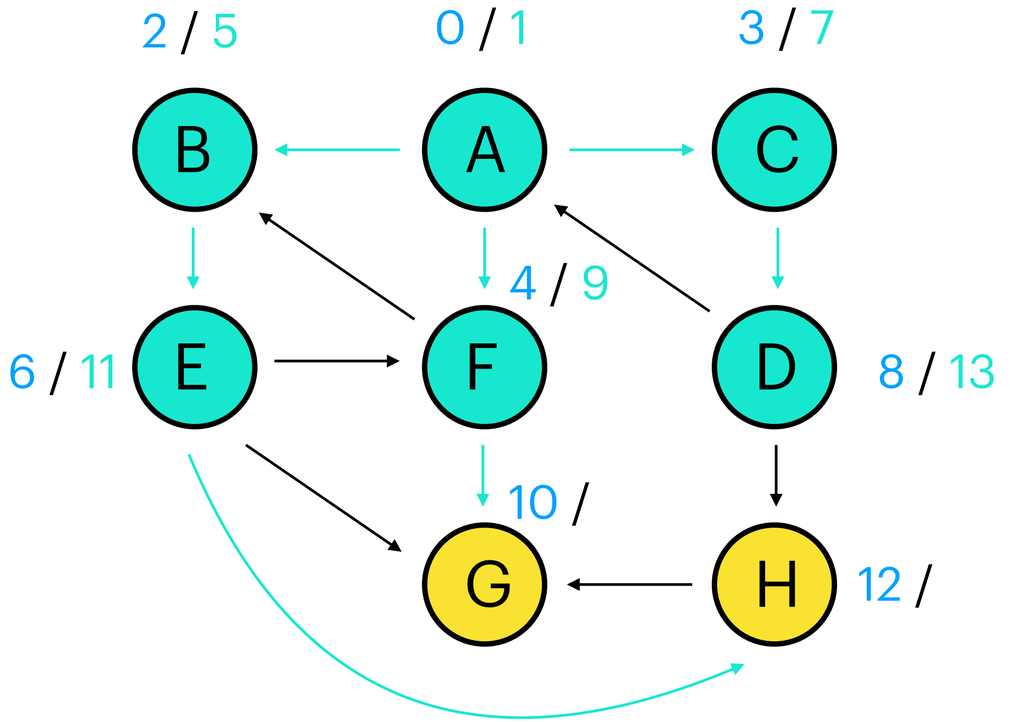
Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Q : 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	13	11	9		

distance[] :

A	B	C	D	E	F	G	H
0	1	1	2	2	1	2	3

Graph Searchs

BFS - Example

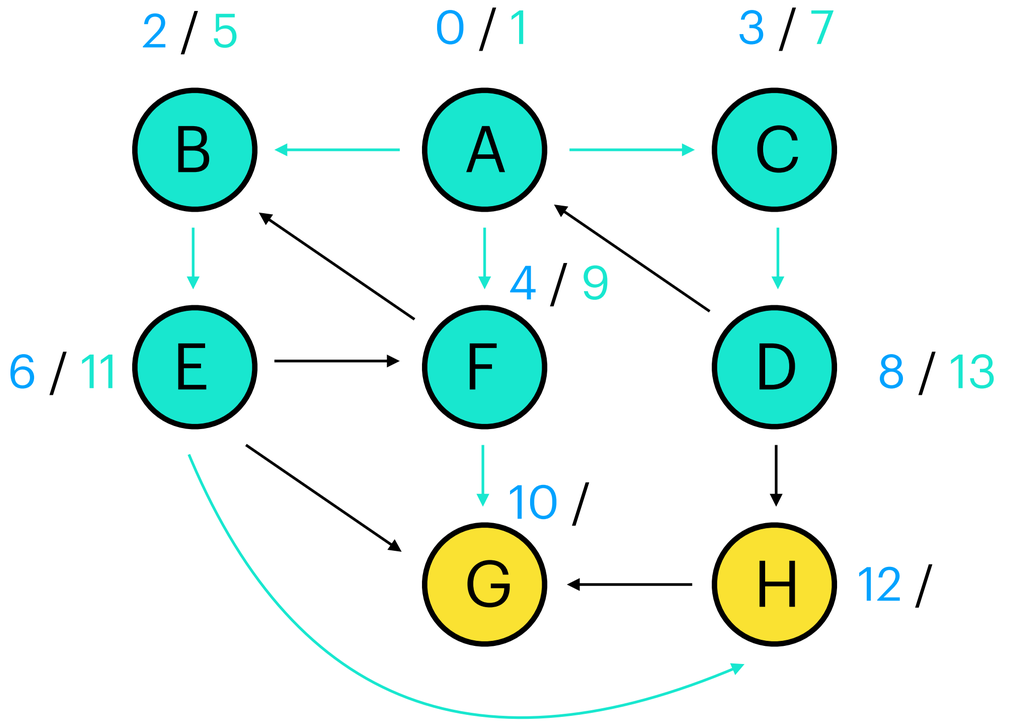
Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Q : 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	13	11	9		

distance[] :

A	B	C	D	E	F	G	H
0	1	1	2	2	1	2	3

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

distance[v] <- distance[u] + 1 ;

Q : H

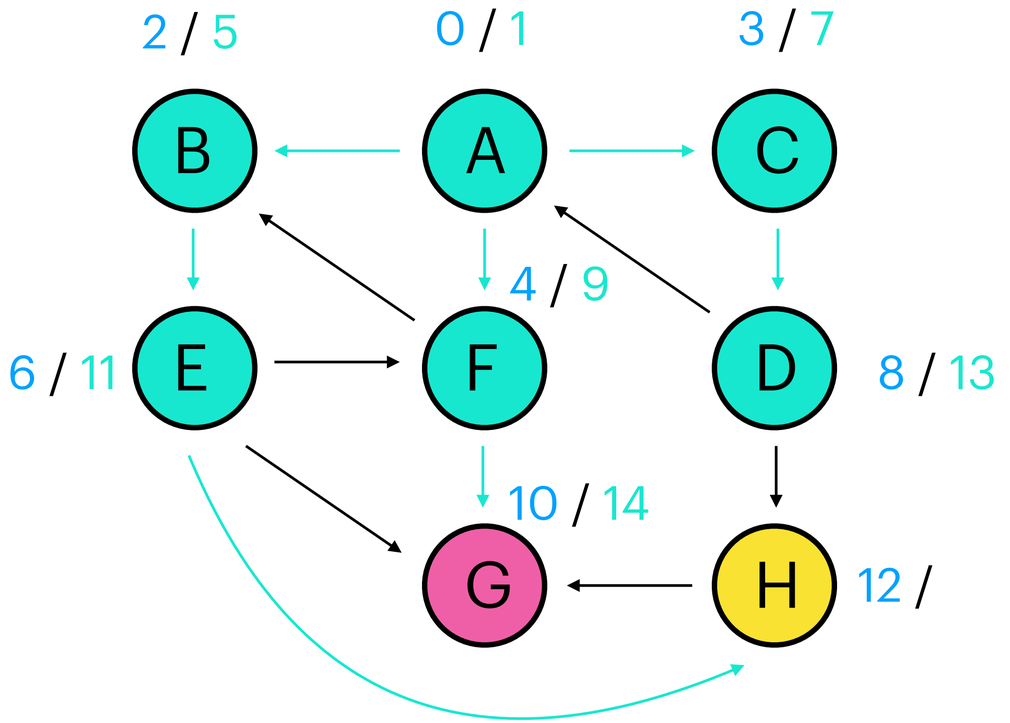
u = G

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

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

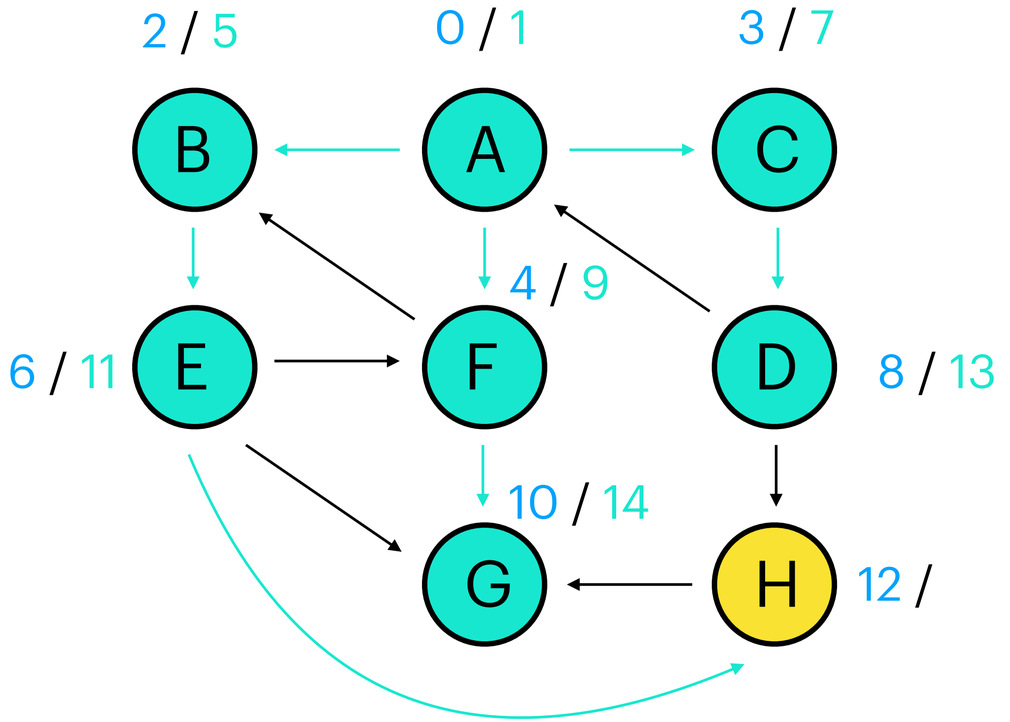
1: $Q \leftarrow \{s\}$
2: $\text{enter}[s] \leftarrow 0; \quad T \leftarrow 1$

distance[s] = 0 ;

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

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

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

distance[v] <- distance[u] + 1 ;

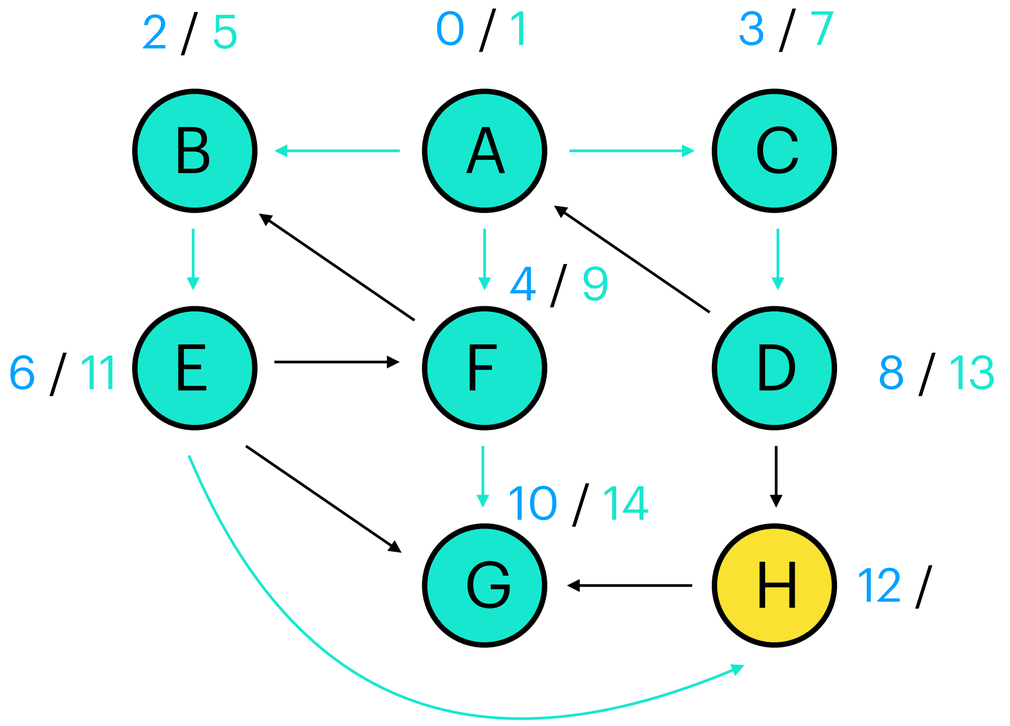
Q : H

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	



Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Q :

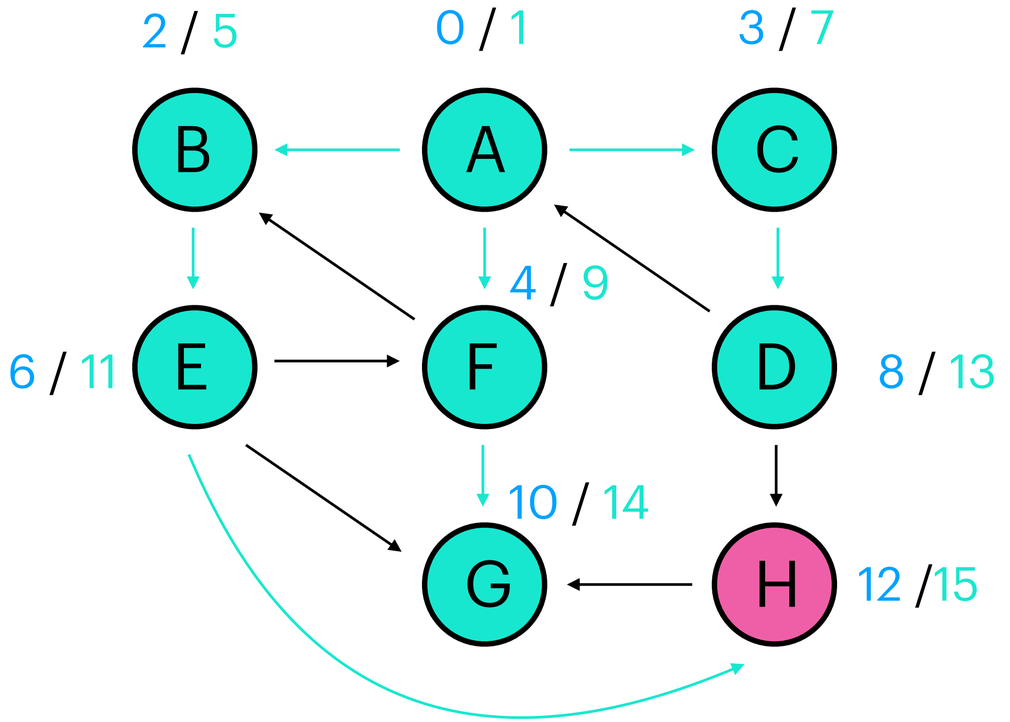
u = H

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

Graph Searchs

BFS - Example

Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

Q :

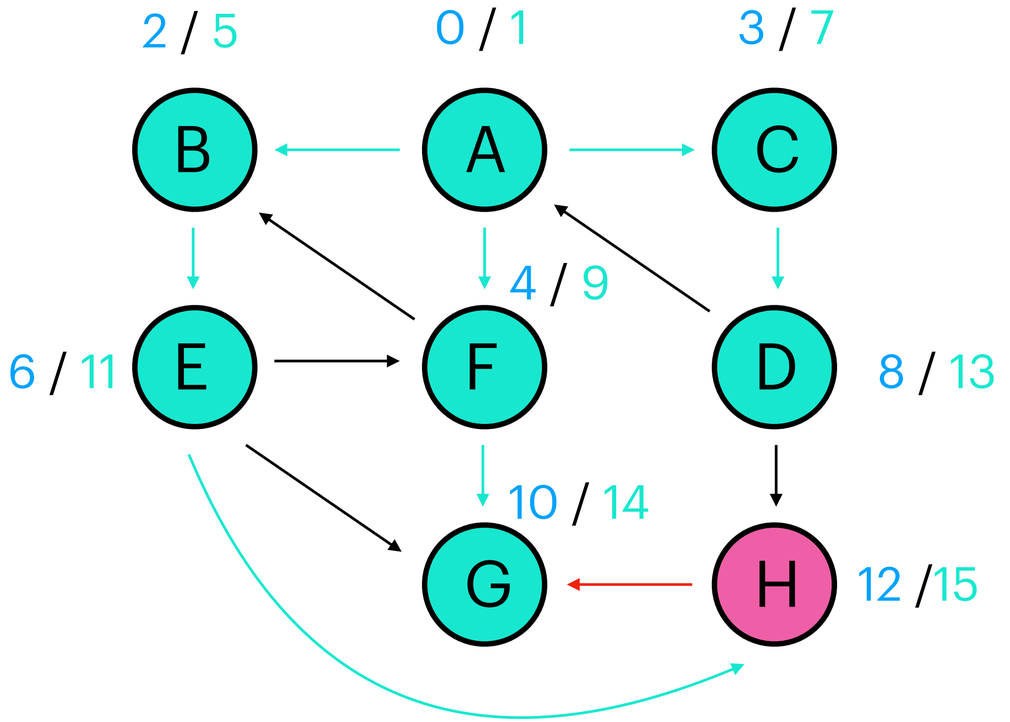
u = H

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

Graph Searchs

BFS - Example

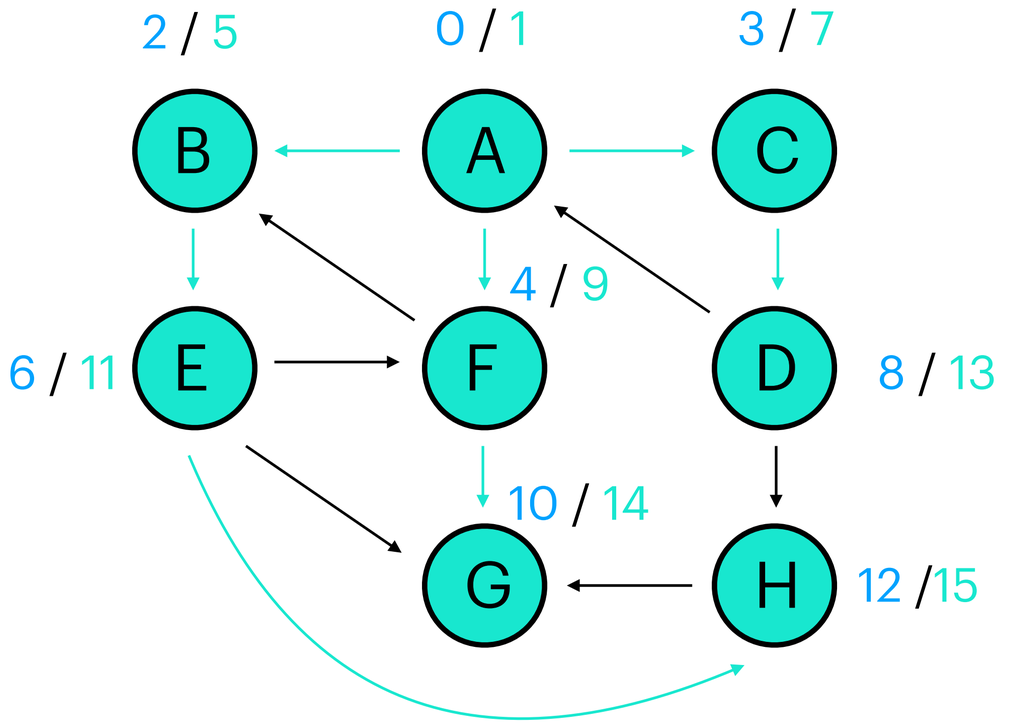
Algorithm 5 BFS(s)

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

distance[s] = 0 ;

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

BFS

Exam Question

HS21

/ 4 P

c) *Finding a shortest cycle*

Describe an algorithm which, given an unweighted directed graph $G = (V, E)$ and a vertex $v \in V$, finds a shortest cycle containing v. If there is no such cycle, the algorithm should report that v is not a vertex of any cycle. Faster algorithms are worth more points. To get full points, aim for $O(|V| + |E|)$ runtime.

DFS, BFS

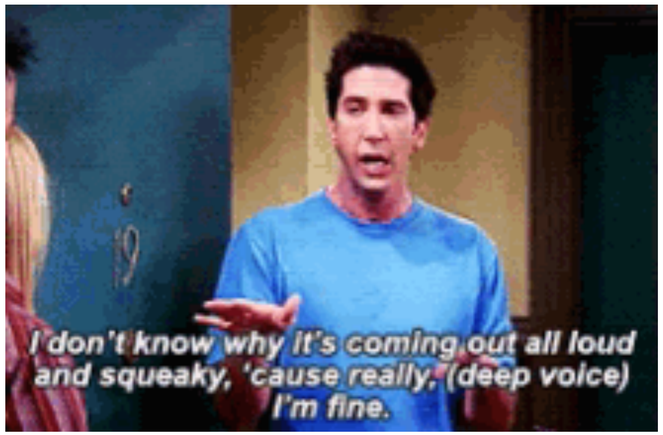
Exam Tipps

- DFS for **reachability** , finding components
- BFS for **distance** in unweighted Graph (Shortest Path in unweighted Graph)

- Theory
  - Usual new algorithm tipps
  - Watch out for the runtime !! O(|V| + |E|)
- Coding
  - Know how to implement both !! How ? ...

Let’s take a break

Code Expert - Graph Sets



Next Week ...

Shortest Path algorithms (one-to-all) ...

Questions

Feedbacks , Recommendations

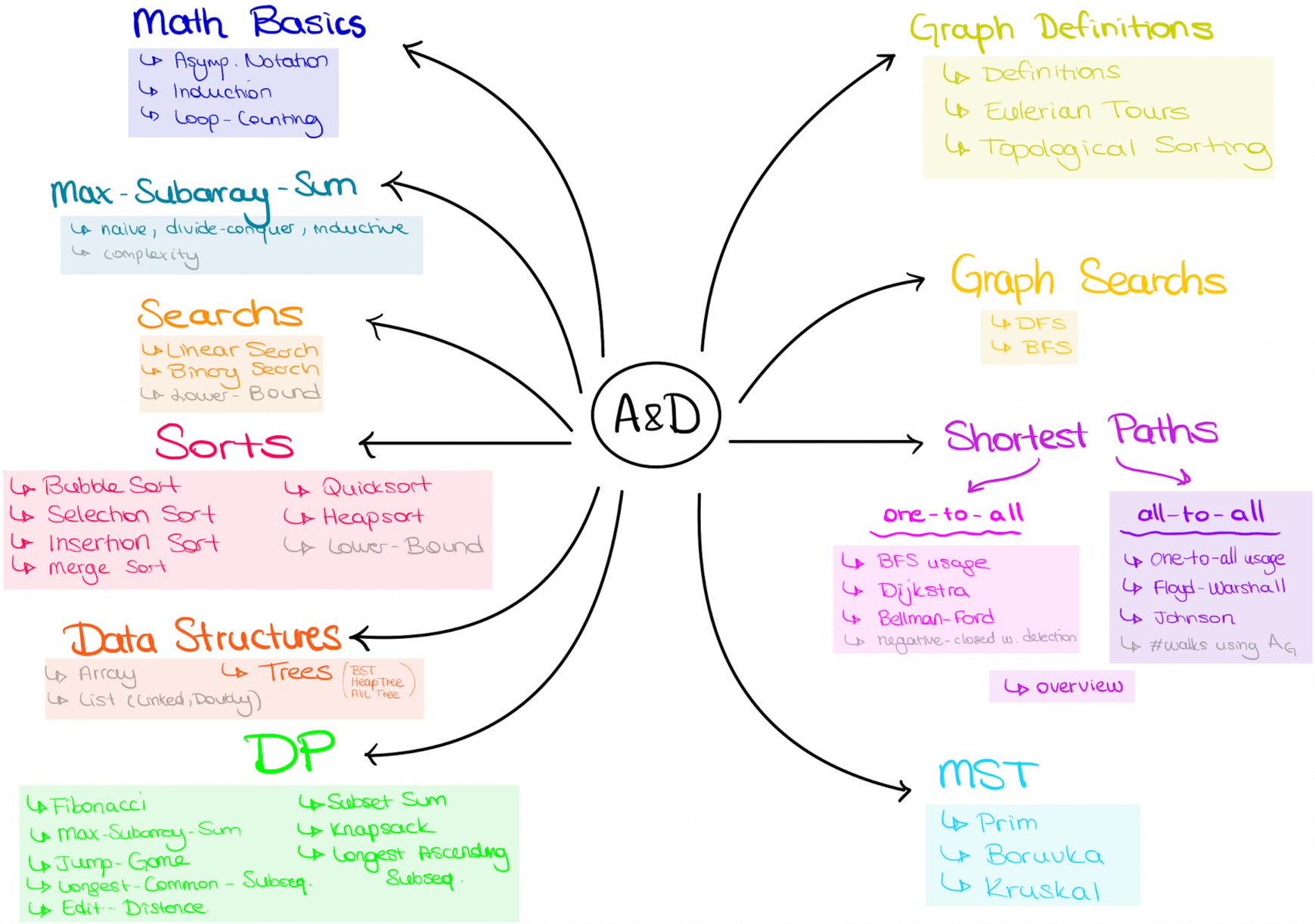
Nil Ozer

A&D

Exercise Session 11

Nil Ozer

A&D Overview


A&D
Math Basics
↳ 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

Outline

- Quiz
- Shortest Paths - one to all

- Code Expert Graph Sets

Quiz

Peergrading and rest

- Exercise Sheet 10 peergrading
  - 10.4 this week
  - Emails will be sent

- If urgent feedback is needed, send me an email !

Shortest Paths

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

Shortest Paths

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|)

Shortest Paths

BFS usage - with distances

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

Algorithm 5 BFS(s)

```
1: Q ← {s}
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
```

distance[s] = 0 ;

distance[v] <- distance[u] + 1 ;

Q is a FIFO queue

Shortest Paths

Dijkstra's Algorithm

Runtime : O ((|V| + |E|) * log n)

weighted , nonnegative edge weights
c(e) ≥ 0

Algorithm 6 Dijkstra(s)

1: $\mathrm{d}[s] \leftarrow 0$; $\mathrm{d}[v] \leftarrow \infty \ \ \forall v \in V \setminus \{s\}$
2: $S \leftarrow \varnothing$
3: $H \leftarrow$ make-heap(V); decrease-key($H, s, 0$)
4: **while** $S \neq V$ **do**
5: $\quad v^* \leftarrow$ extract-min(H)
6: $\quad S \leftarrow S \cup \{v^*\}$
7: $\quad$ **for** $(v^*, v) \in E,\ v \notin S$ **do**
8: $\quad\quad \mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[v^*] + c(v^*, v)\}$
9: $\quad\quad$ decrease-key($H, v, \mathrm{d}[v]$)

Shortest Paths

Dijkstra's Algorithm

Runtime : O ((|V| + |E|) * log n)

weighted , positive edge weights
c(e) ≥ 0

Algorithm 6 Dijkstra(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \ \ \forall v \in V \setminus \{s\}$
2: $S \leftarrow \varnothing$
3: $H \leftarrow \text{make-heap}(V)$; $\text{decrease-key}(H, s, 0)$
4: **while** $S \neq V$ **do**
5: $\quad v^* \leftarrow \text{extract-min}(H)$
6: $\quad S \leftarrow S \cup \{v^*\}$
7: $\quad$ **for** $(v^*, v) \in E, \ v \notin S$ **do**
8: $\quad\quad \mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[v^*] + c(v^*, v)\}$
9: $\quad\quad \text{decrease-key}(H, v, \mathrm{d}[v])$

d[] : distance array

S : visited set

H : min-heap

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

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])
```

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

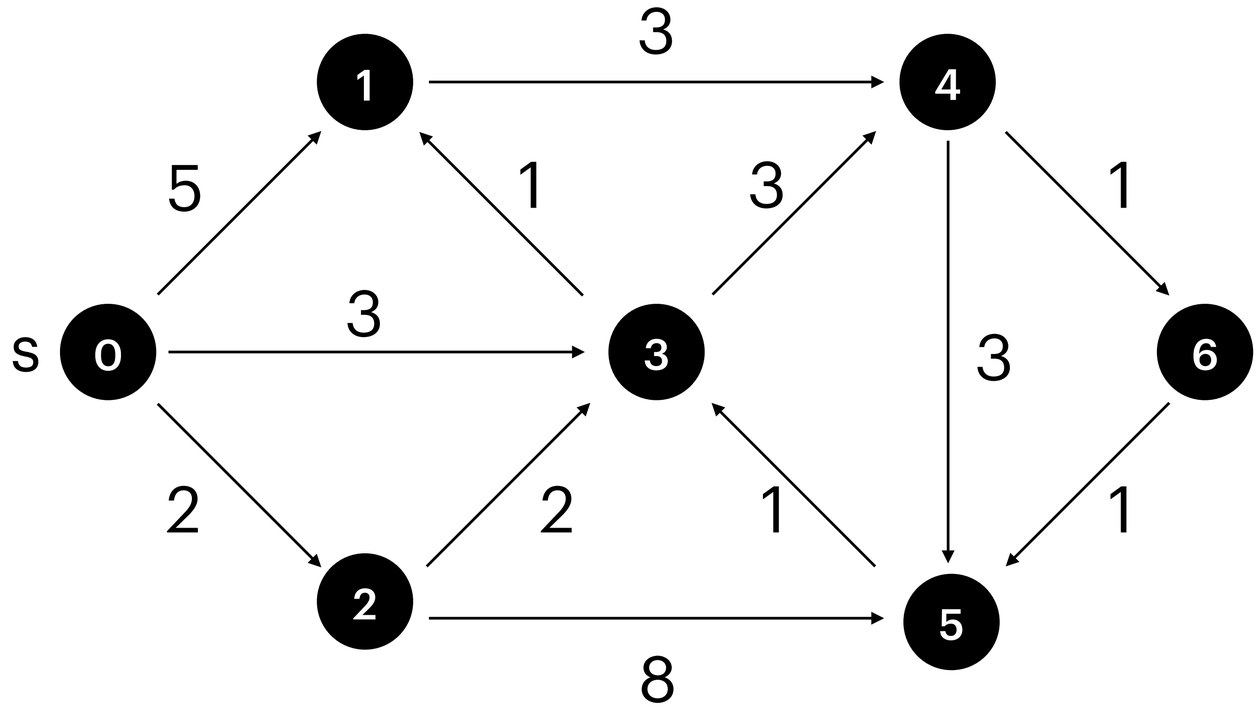
S :

d[] :

0	1	2	3	4	5	6

"Heap" :

0	1	2	3	4	5	6



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])
```

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

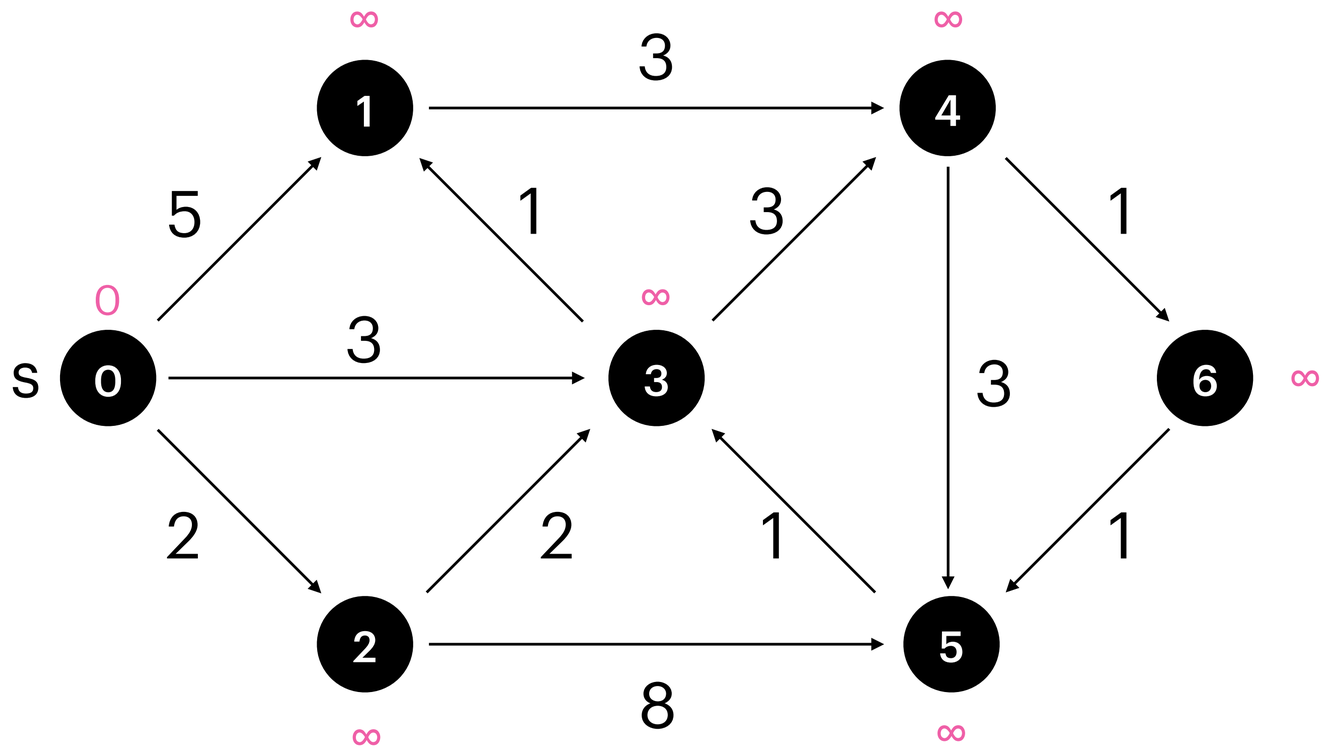
S : ∅

d[] :

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞

"Heap" :

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞



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])
```

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

S : ∅

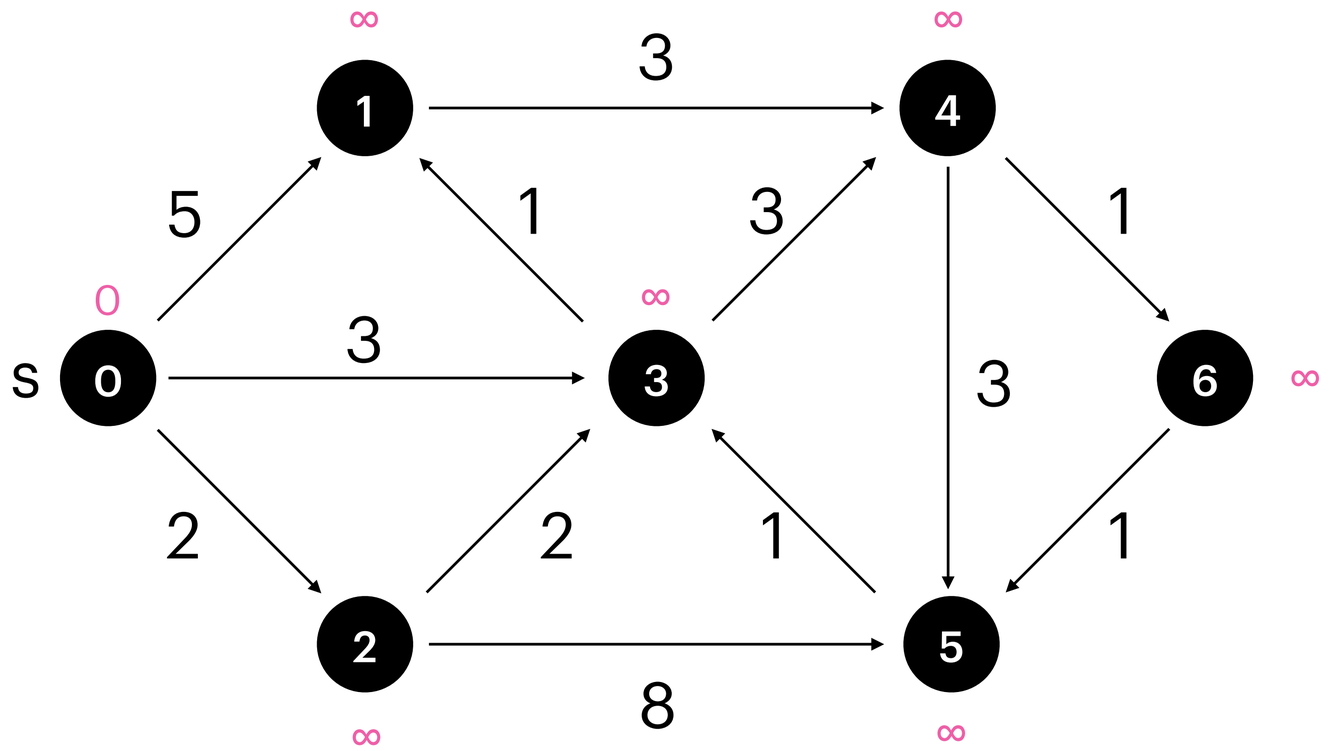
V* =

d[] :

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞

"Heap" :

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞



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])
```

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

S : ∅

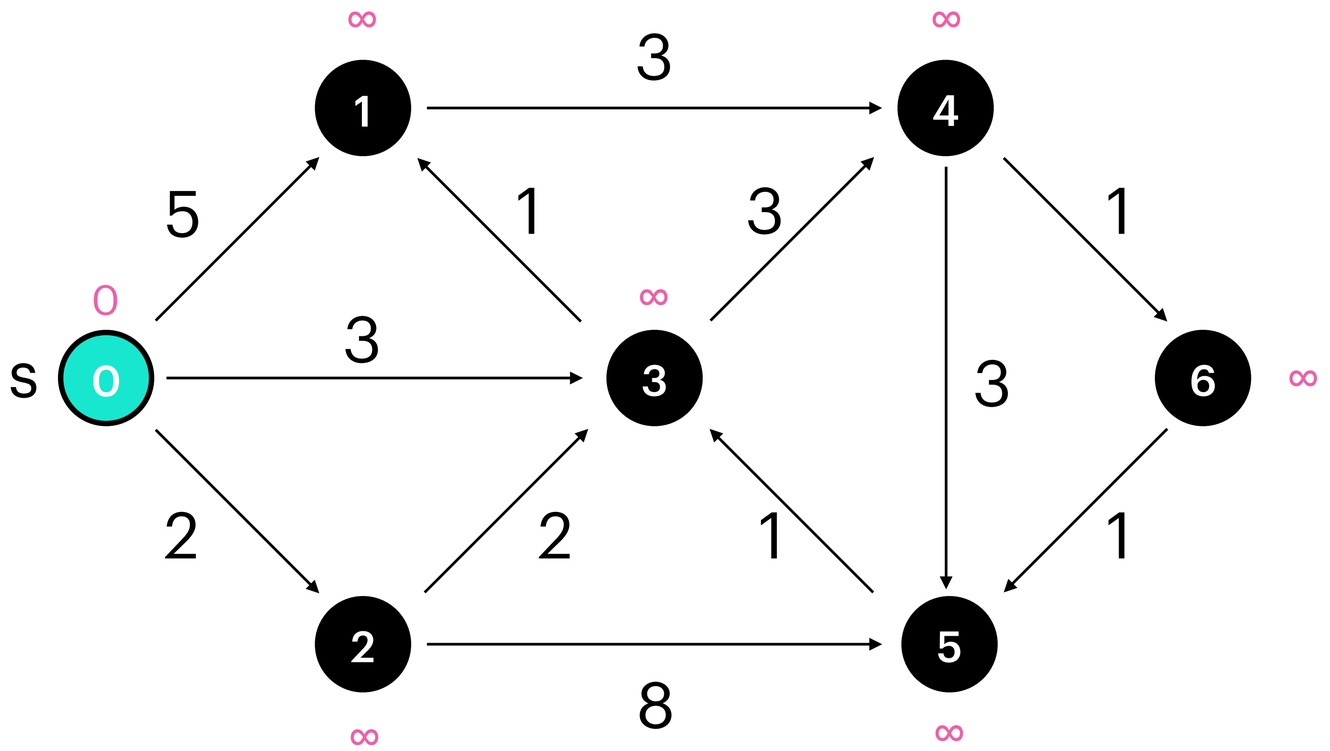
v* = 0

d[] :

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞

"Heap" :

1	2	3	4	5	6
∞	∞	∞	∞	∞	∞



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])
```

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

S : { 0 }

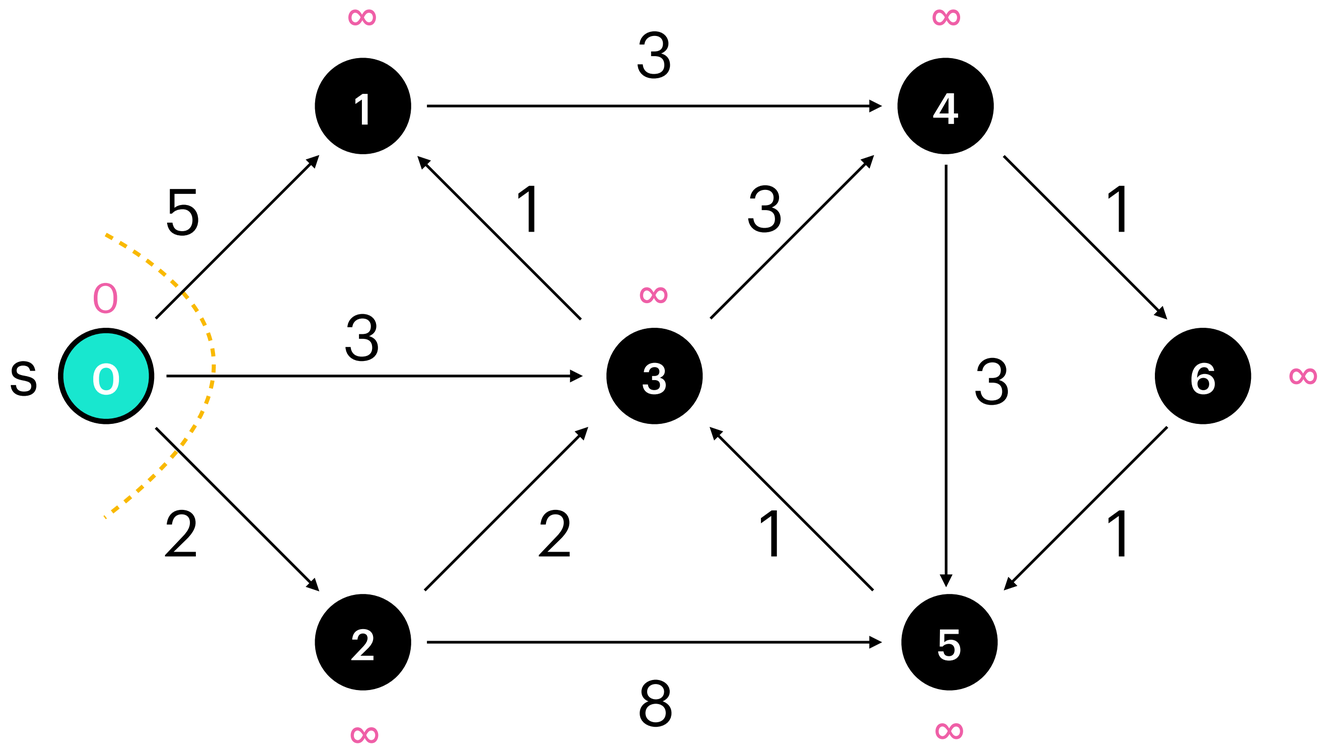
v* = 0

d[] :

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞

"Heap" :

1	2	3	4	5	6
∞	∞	∞	∞	∞	∞



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])
```

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

S : { 0 }

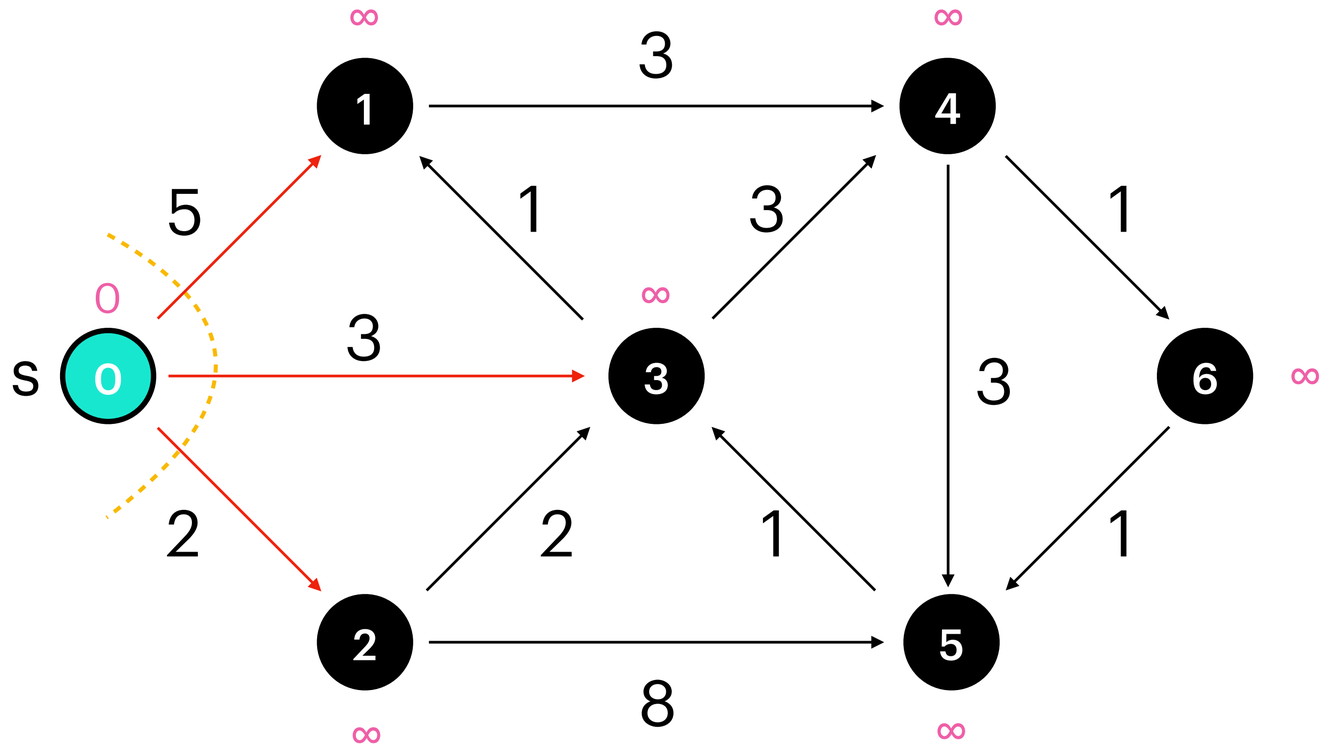
v* = 0

d[] :

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞

"Heap" :

1	2	3	4	5	6
∞	∞	∞	∞	∞	∞



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])
```

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

S : { 0 }

v* = 0

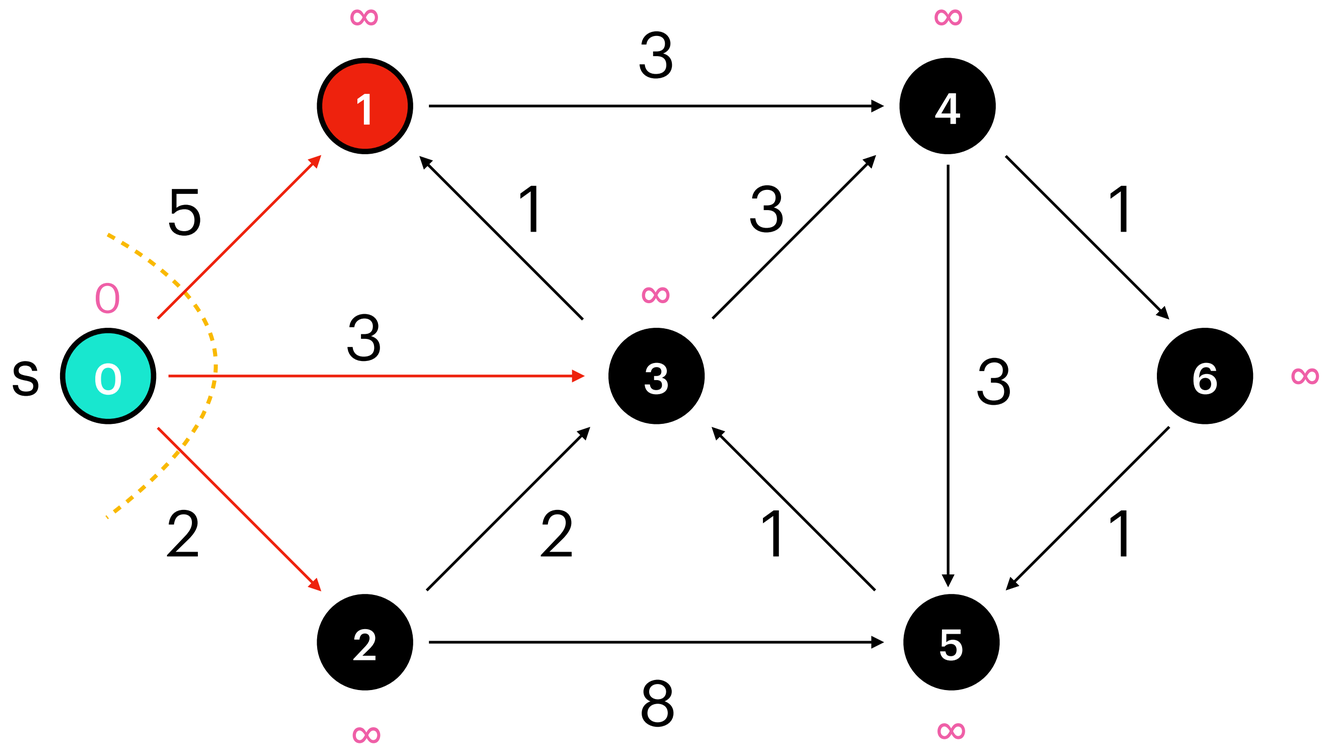
d[] :

0	1	2	3	4	5	6
0	5	∞	∞	∞	∞	∞

"Heap" :

1	2	3	4	5	6
5	∞	∞	∞	∞	∞

min {∞ , 0 + 5} = 5



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])
```

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

S : { 0 }

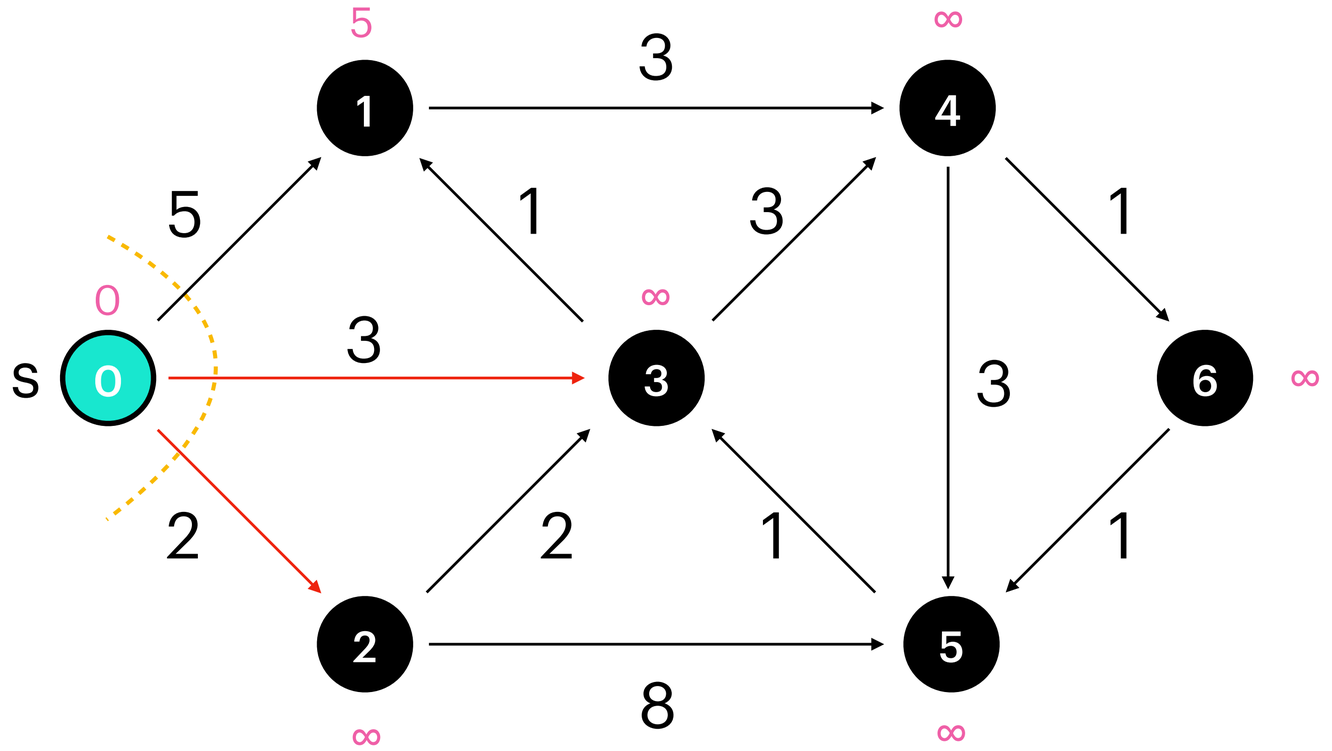
v* = 0

d[] :

0	1	2	3	4	5	6
0	5	∞	∞	∞	∞	∞

"Heap" :

1	2	3	4	5	6
5	∞	∞	∞	∞	∞



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])
```

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

S : { 0 }

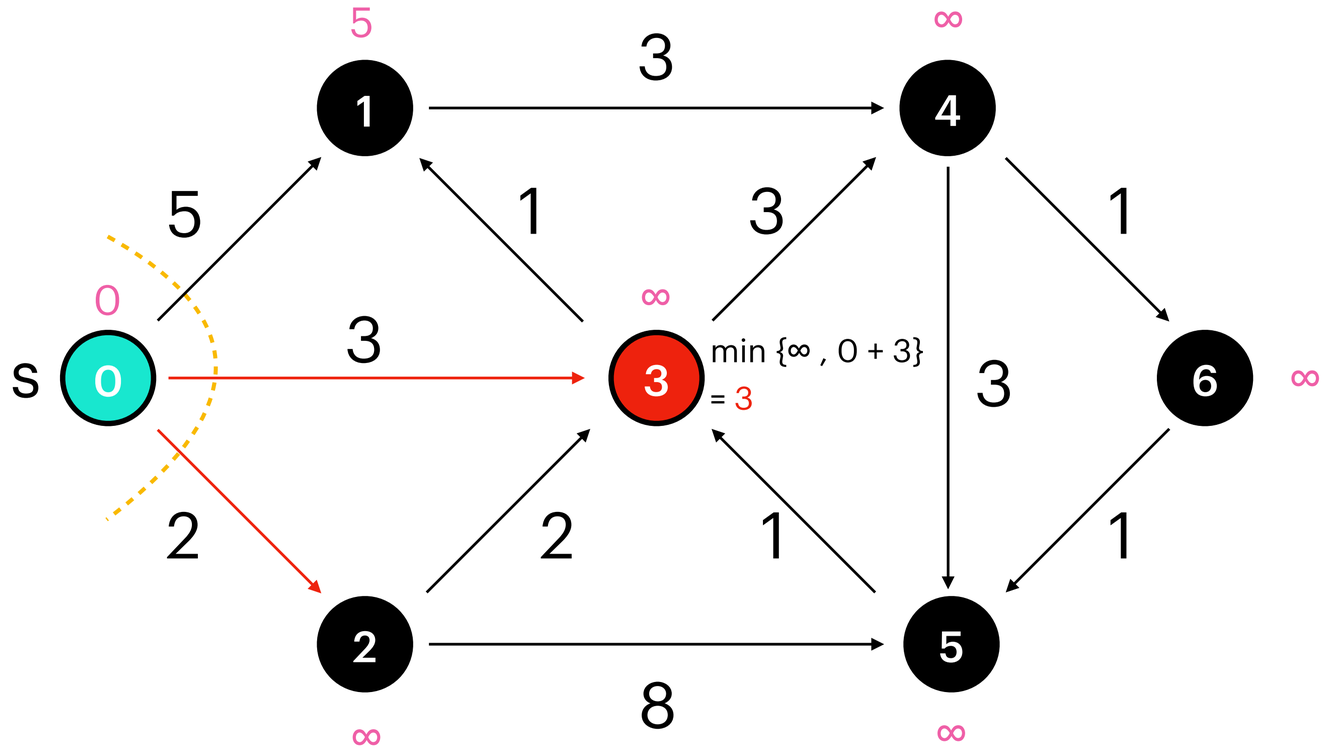
v* = 0

d[] :

0	1	2	3	4	5	6
0	5	∞	3	∞	∞	∞

"Heap" :

1	2	3	4	5	6
5	∞	3	∞	∞	∞



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])
```

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

S : { 0 }

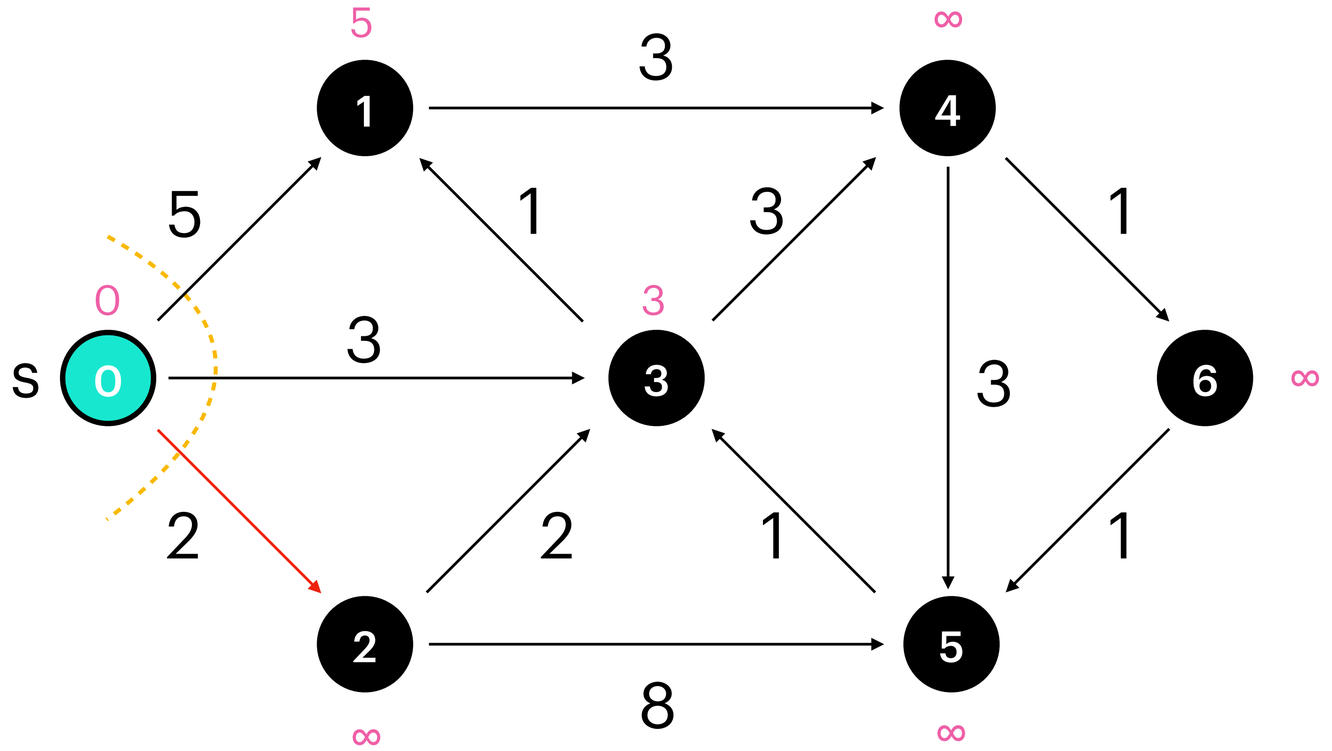
v* = 0

d[] :

0	1	2	3	4	5	6
0	5	∞	3	∞	∞	∞

"Heap" :

1	2	3	4	5	6
5	∞	3	∞	∞	∞



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])
```

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

S : { 0 }

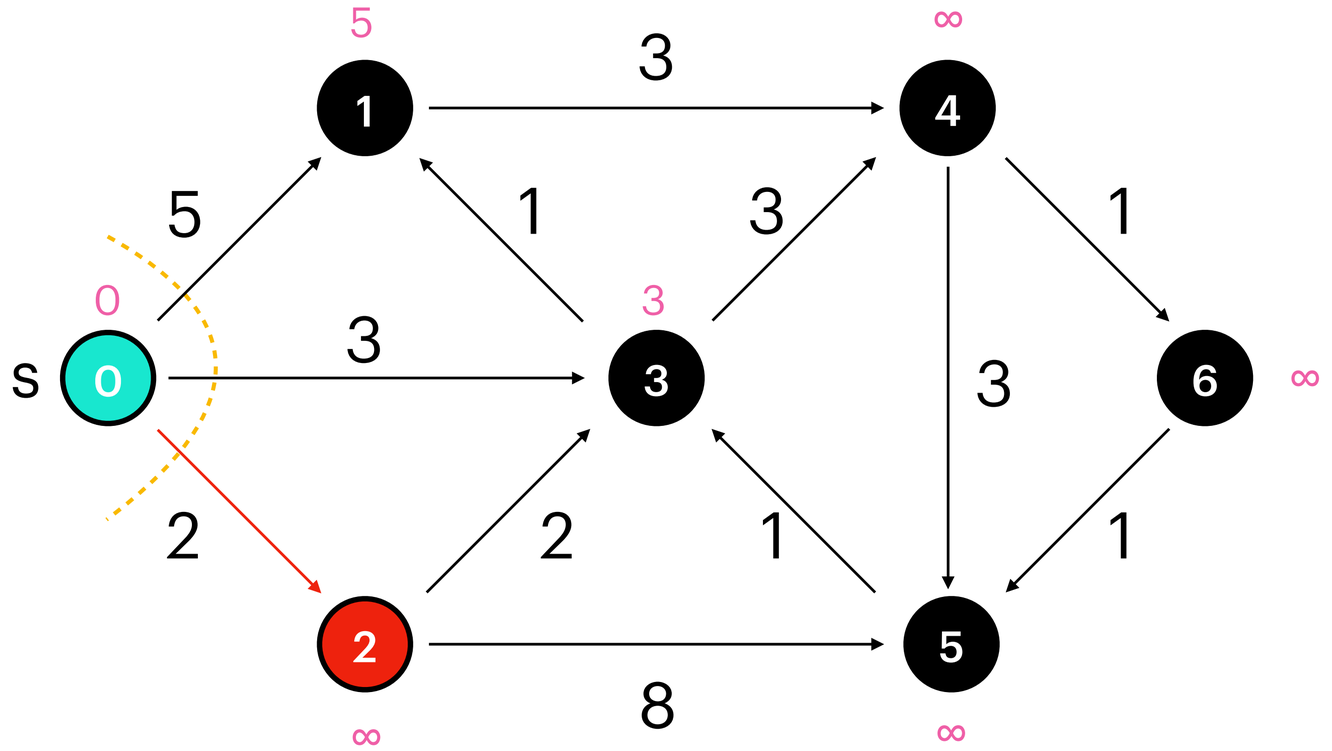
v* = 0

d[] :

0	1	2	3	4	5	6
0	5	2	3	∞	∞	∞

"Heap" :

1	2	3	4	5	6
5	2	3	∞	∞	∞



min {∞ , 0 + 2} = 2

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])
```

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

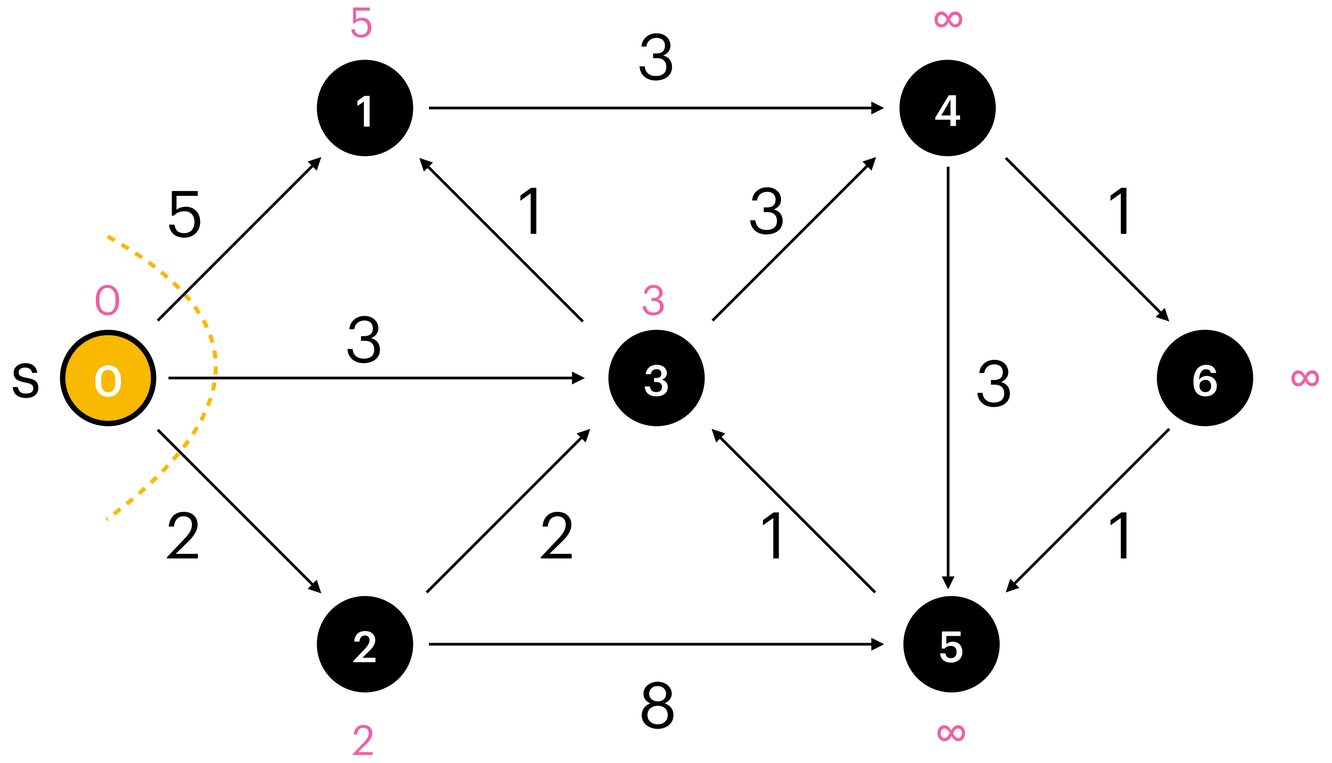
S : { 0 }

d[] :

0	1	2	3	4	5	6
0	5	2	3	∞	∞	∞

"Heap" :

1	2	3	4	5	6
5	2	3	∞	∞	∞



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])
```

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

S : { 0 }

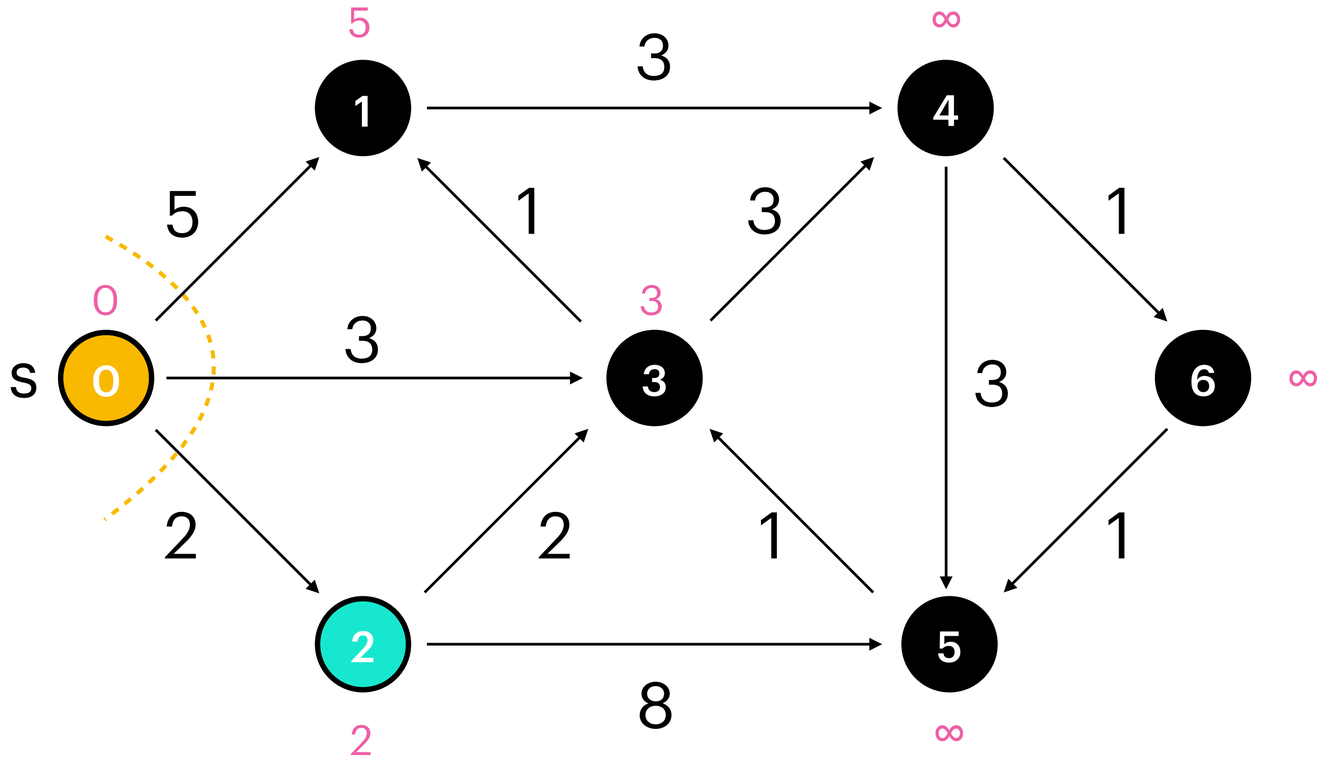
v* = 2

d[] :

0	1	2	3	4	5	6
0	5	2	3	∞	∞	∞

"Heap" :

1	3	4	5	6
5	3	∞	∞	∞



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])
```

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

S : { 0 , 2 }

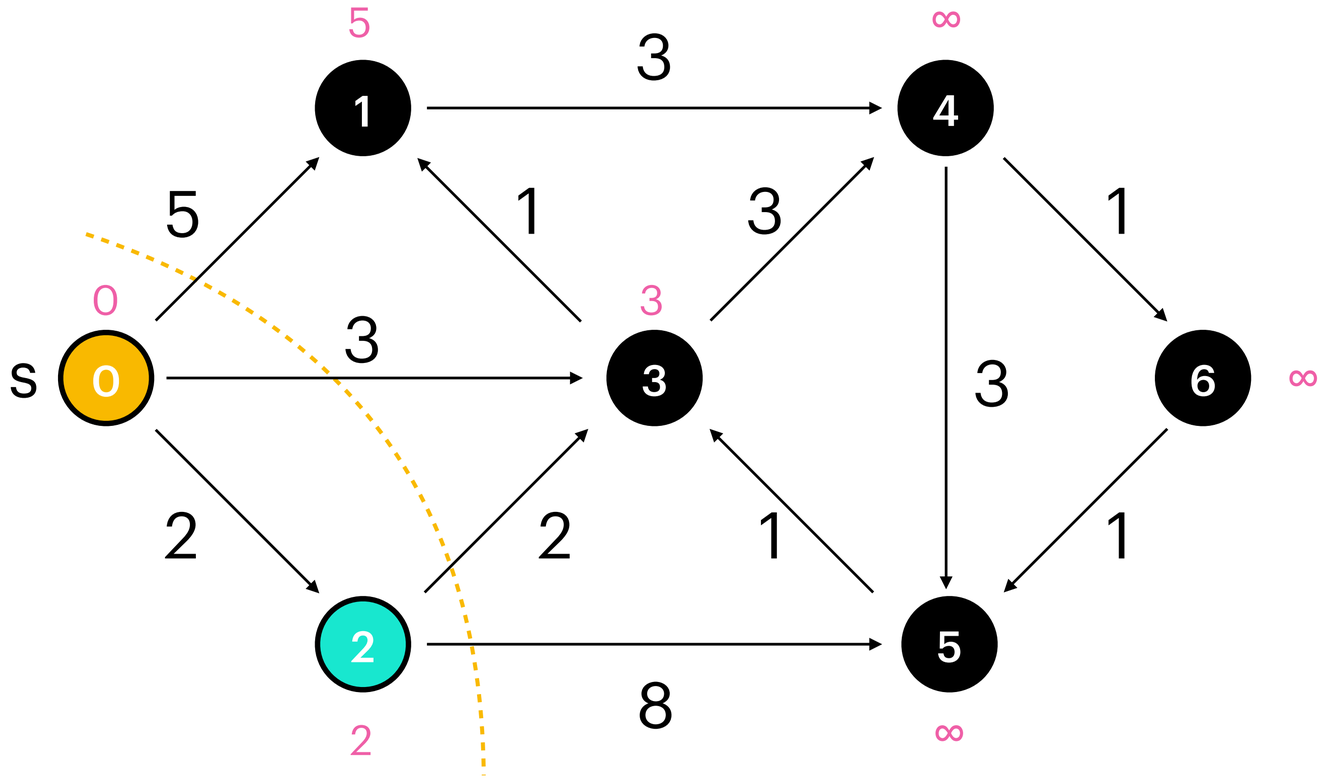
v* = 2

d[] :

0	1	2	3	4	5	6
0	5	2	3	∞	∞	∞

"Heap" :

1	3	4	5	6
5	3	∞	∞	∞



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])
```

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

S : { 0 , 2 }

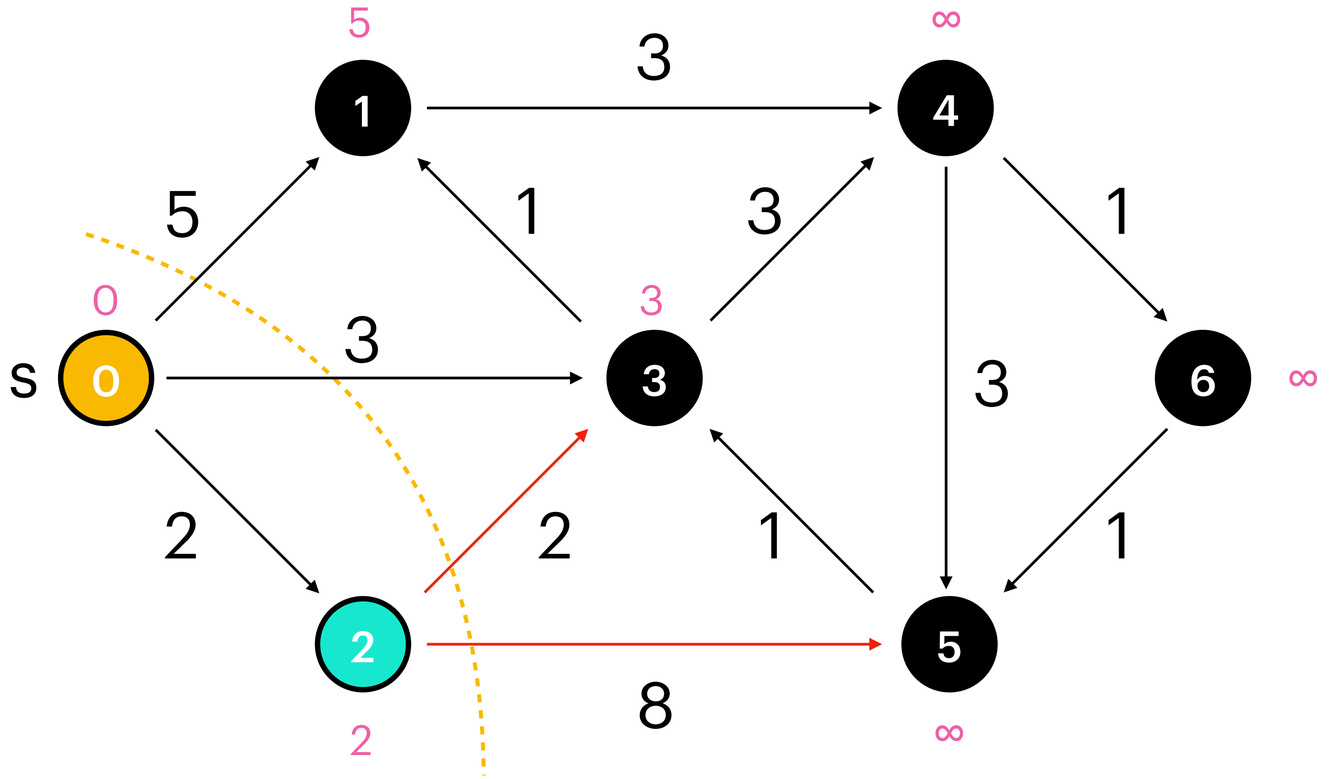
v* = 2

d[] :

0	1	2	3	4	5	6
0	5	2	3	∞	∞	∞

"Heap" :

1	3	4	5	6
5	3	∞	∞	∞



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])
```

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

S : { 0 , 2 }

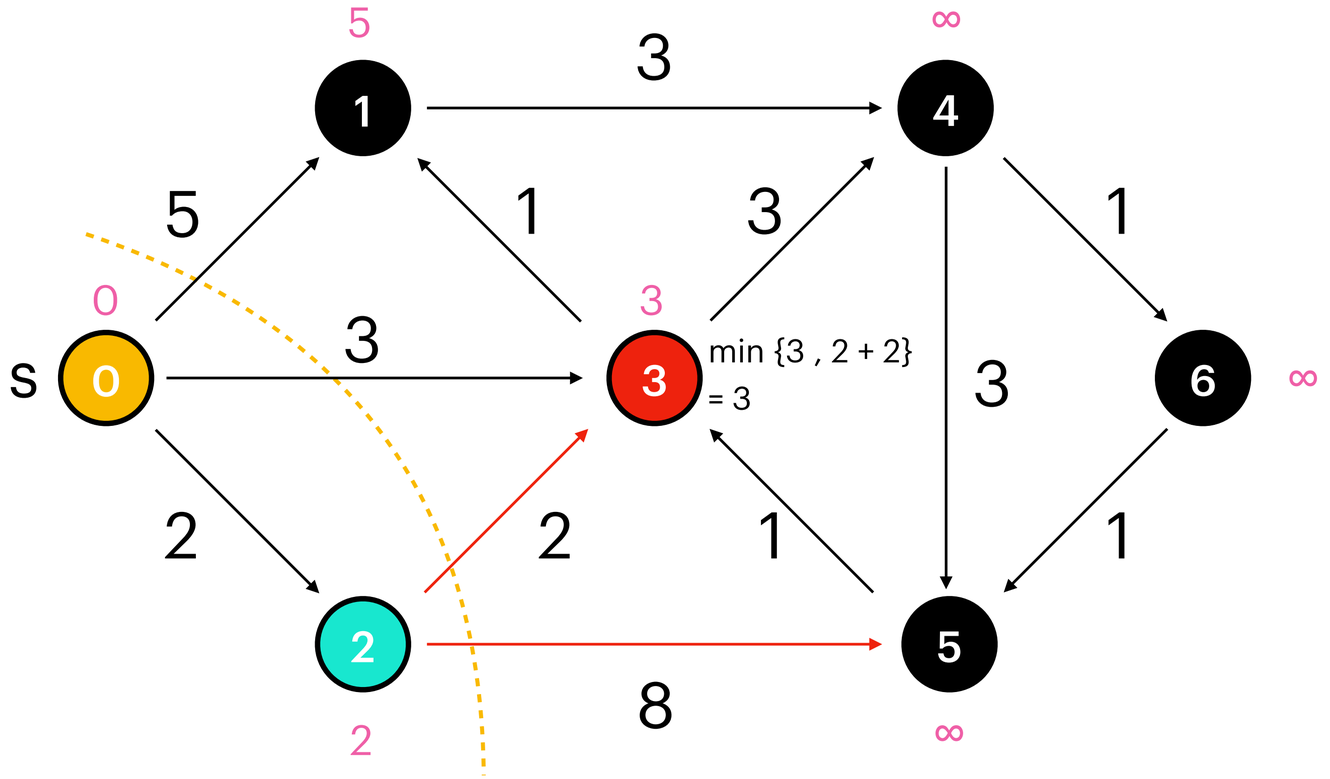
v* = 2

d[] :

0	1	2	3	4	5	6
0	5	2	3	∞	∞	∞

"Heap" :

1	3	4	5	6
5	3	∞	∞	∞



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])
```

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

S : { 0 , 2 }

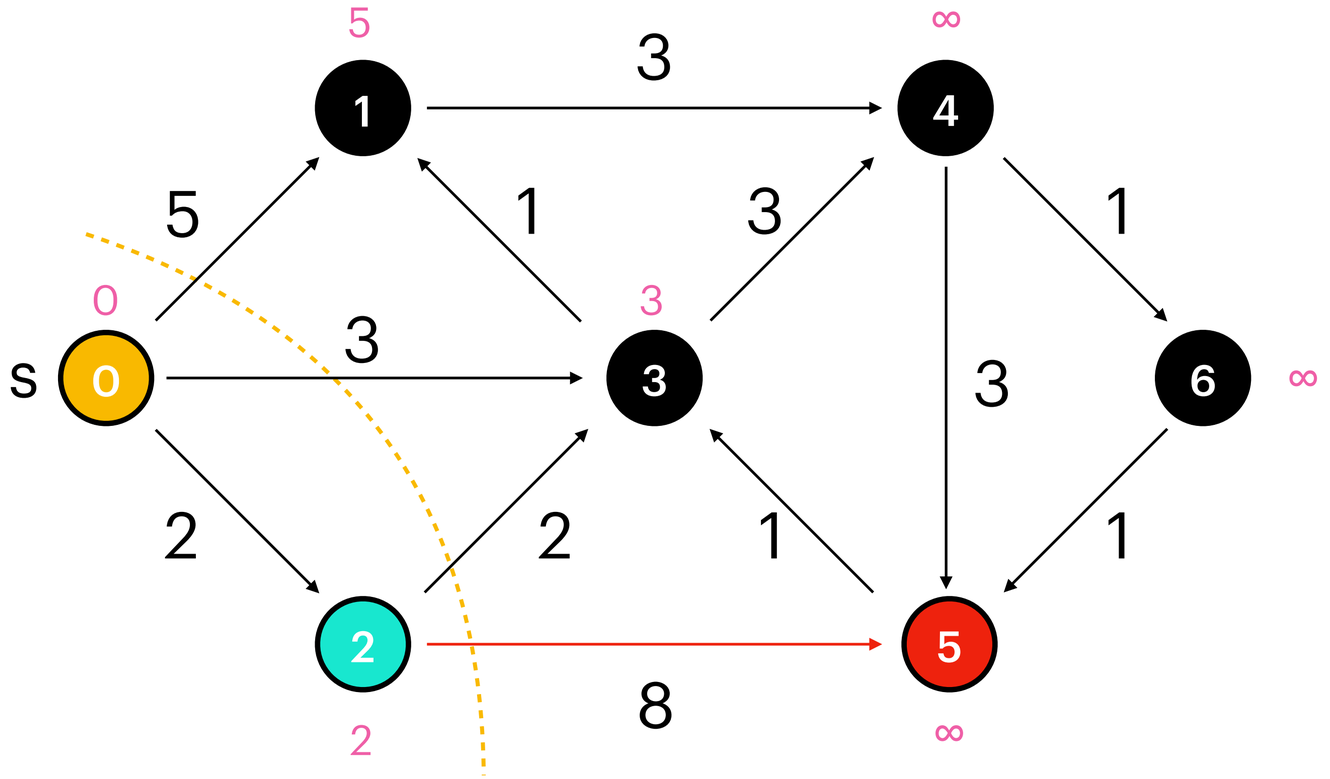
v* = 2

d[] :

0	1	2	3	4	5	6
0	5	2	3	∞	∞	∞

"Heap" :

1	3	4	5	6
5	3	∞	∞	∞



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])
```

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

S : { 0 , 2 }

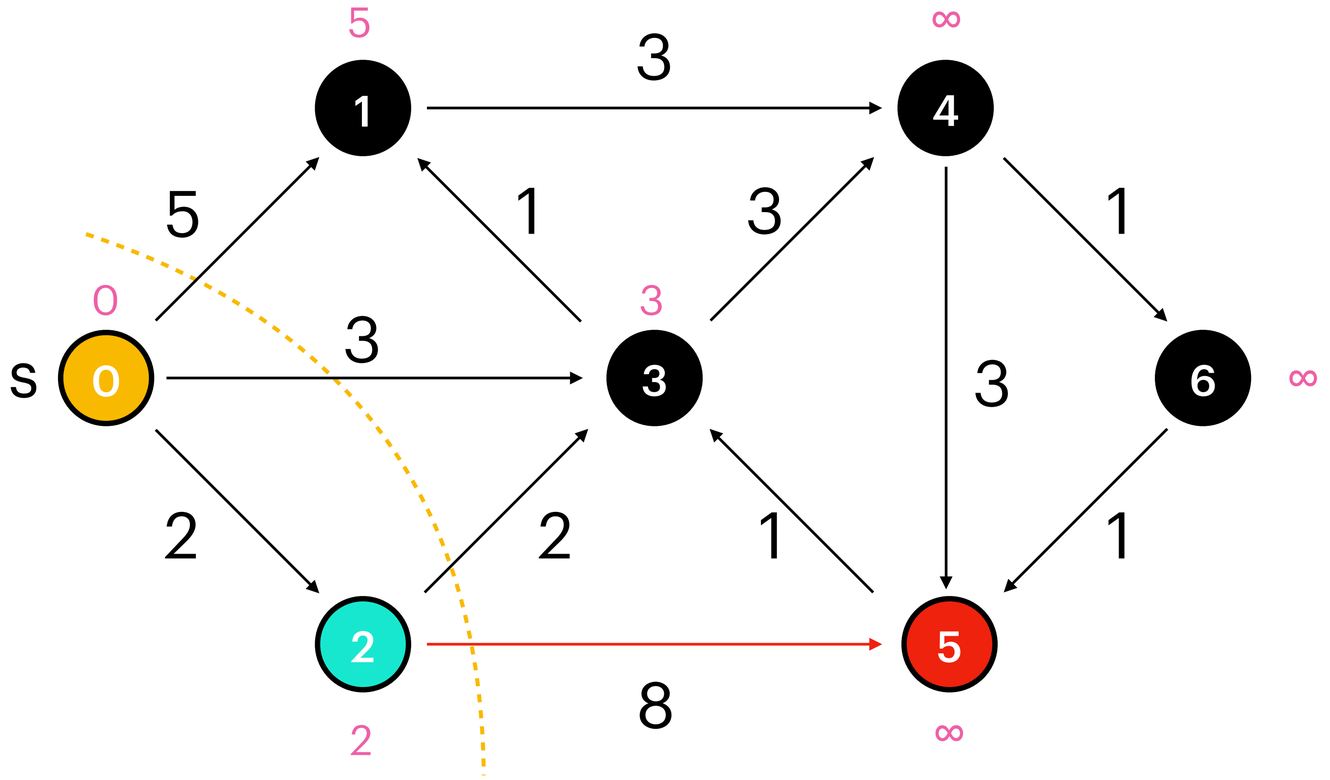
v* = 2

d[] :

0	1	2	3	4	5	6
0	5	2	3	∞	10	∞

"Heap" :

1	3	4	5	6
5	3	∞	10	∞



min {∞ , 2 + 8} = 10

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])
```

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

S : { 0 , 2 }

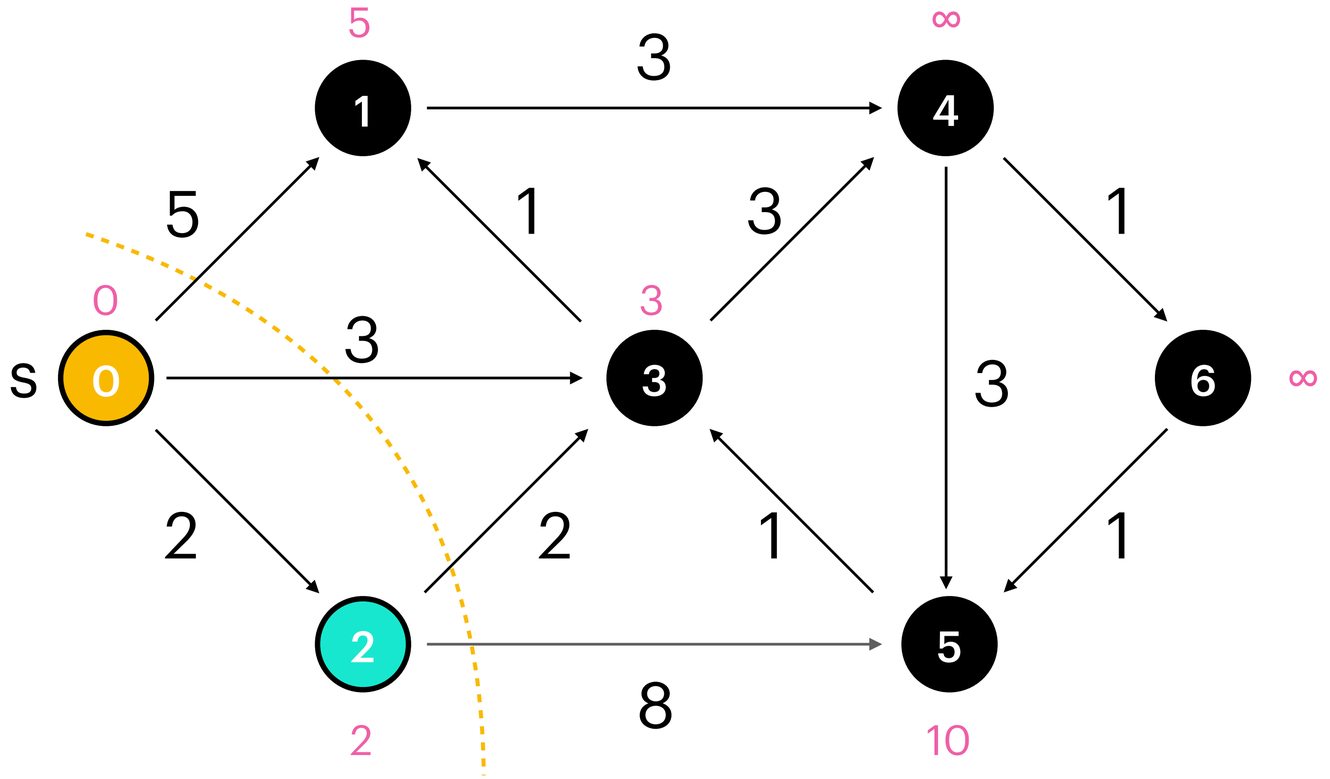
v* = 2

d[] :

0	1	2	3	4	5	6
0	5	2	3	∞	10	∞

"Heap" :

1	3	4	5	6
5	3	∞	10	∞



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])
```

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

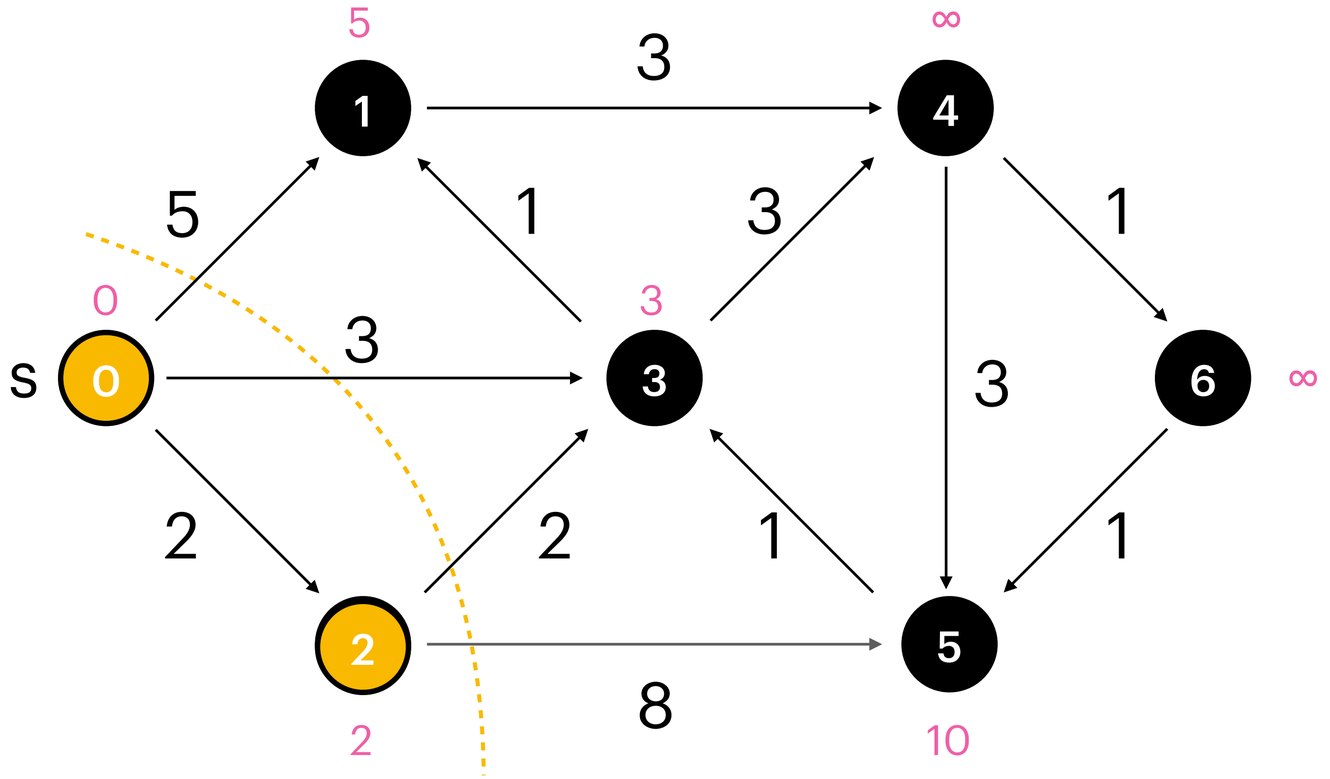
S : { 0 , 2 }

d[] :

0	1	2	3	4	5	6
0	5	2	3	∞	10	∞

"Heap" :

1	3	4	5	6
5	3	∞	10	∞



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])
```

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

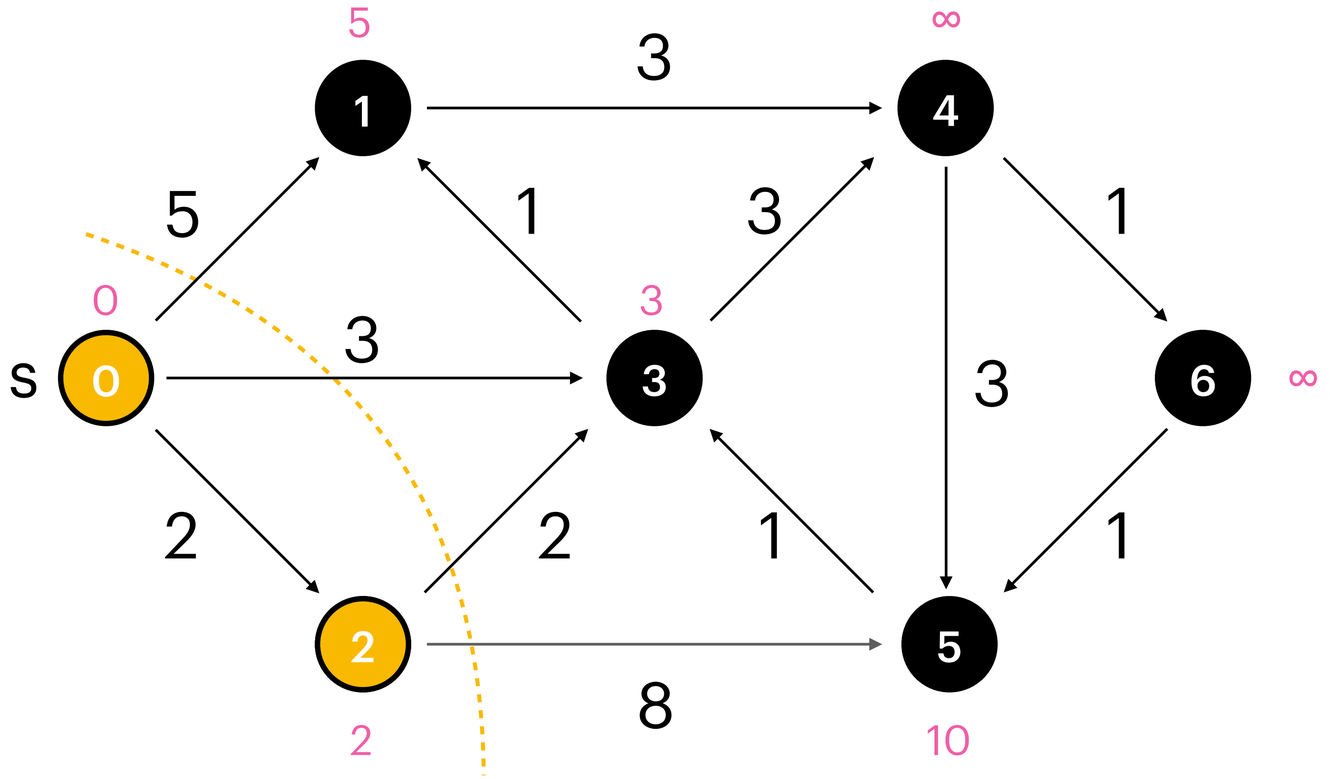
S : { 0 , 2 }

d[] :

0	1	2	3	4	5	6
0	5	2	3	∞	10	∞

"Heap" :

1	3	4	5	6
5	3	∞	10	∞



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])
```

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

S : { 0 , 2 }

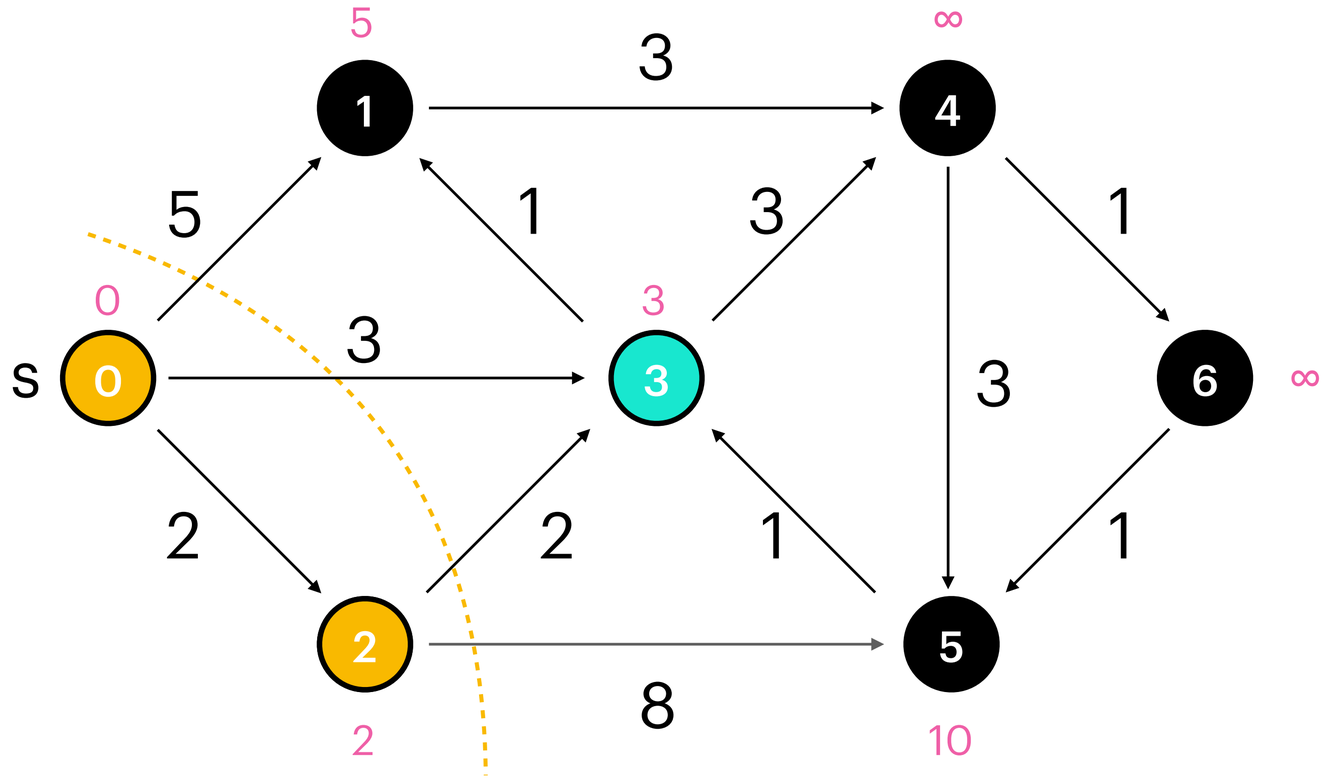
v* = 3

d[] :

0	1	2	3	4	5	6
0	5	2	3	∞	10	∞

"Heap" :

1	4	5	6
5	∞	10	∞



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])
```

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

S : { 0 , 2 , 3 }

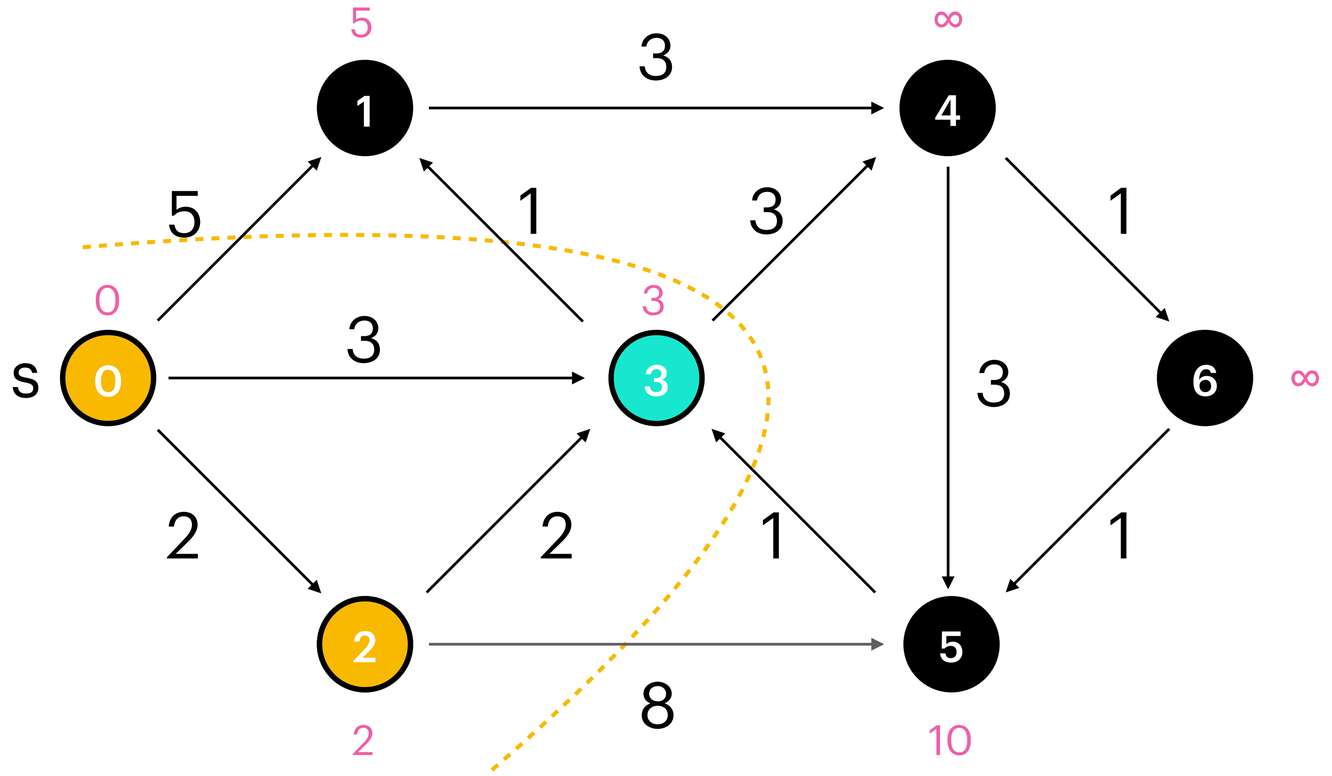
v* = 3

d[] :

0	1	2	3	4	5	6
0	5	2	3	∞	10	∞

"Heap" :

1	4	5	6
5	∞	10	∞



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])
```

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

S : { 0 , 2 , 3 }

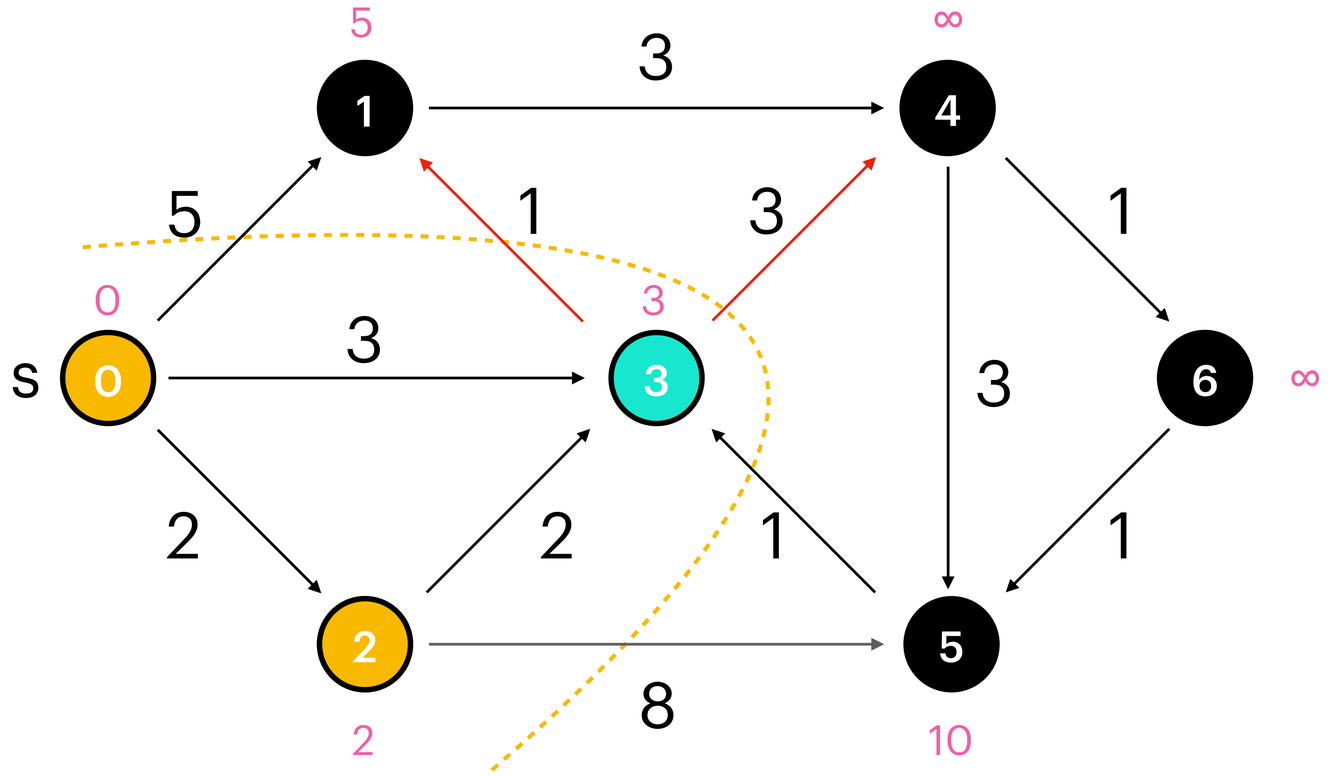
v* = 3

d[] :

0	1	2	3	4	5	6
0	5	2	3	∞	10	∞

"Heap" :

1	4	5	6
5	∞	10	∞



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])
```

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

S : { 0 , 2 , 3 }

v* = 3

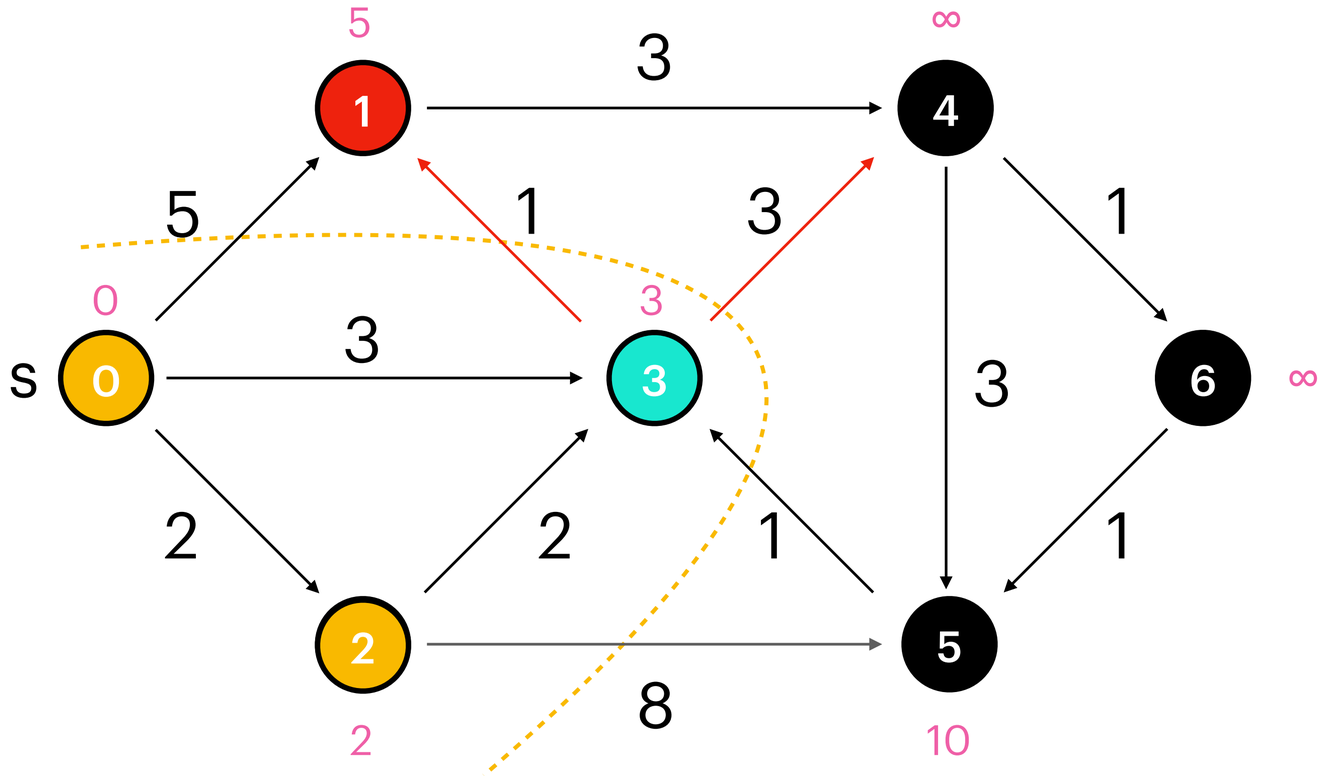
d[] :

0	1	2	3	4	5	6
0	4	2	3	∞	10	∞

"Heap" :

1	4	5	6
4	∞	10	∞

min {5 , 3 + 1} = 4



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])
```

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

S : { 0 , 2 , 3 }

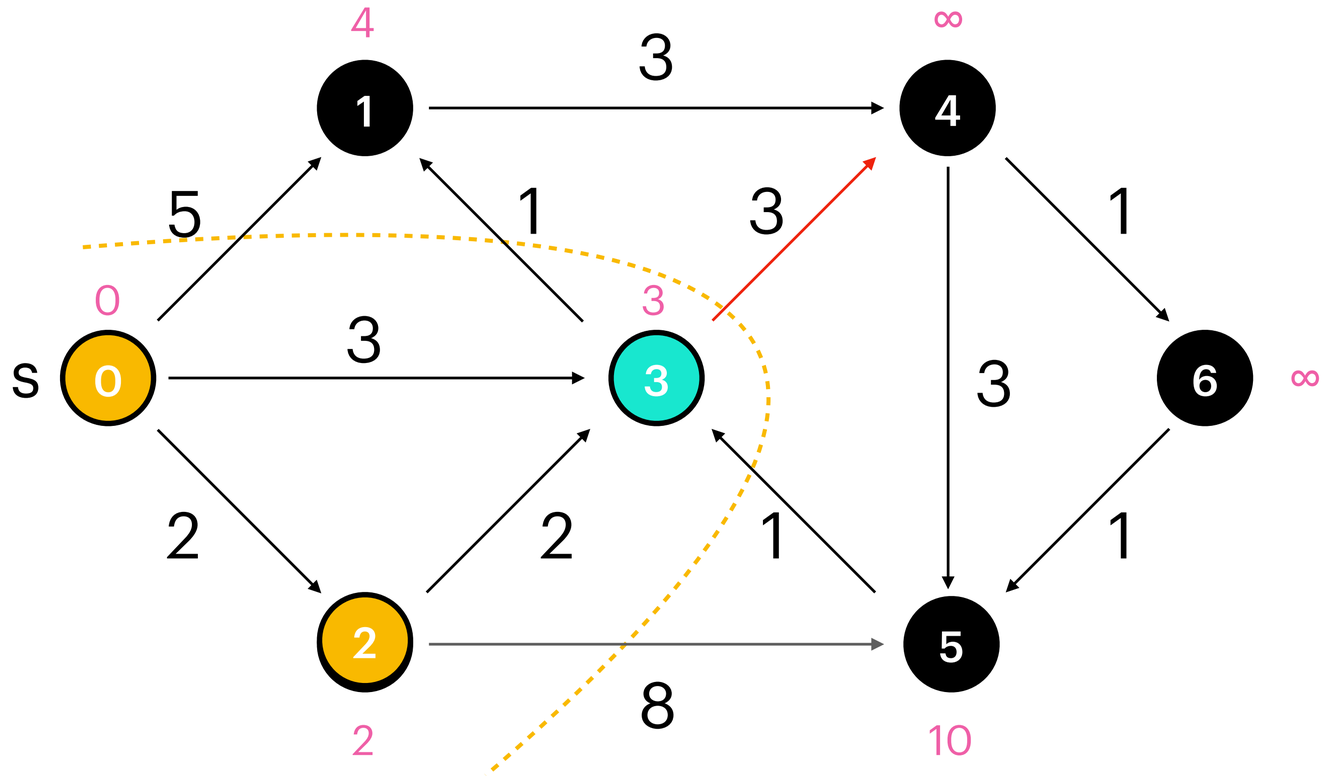
v* = 3

d[] :

0	1	2	3	4	5	6
0	4	2	3	∞	10	∞

"Heap" :

1	4	5	6
4	∞	10	∞



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])
```

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

S : { 0 , 2 , 3 }

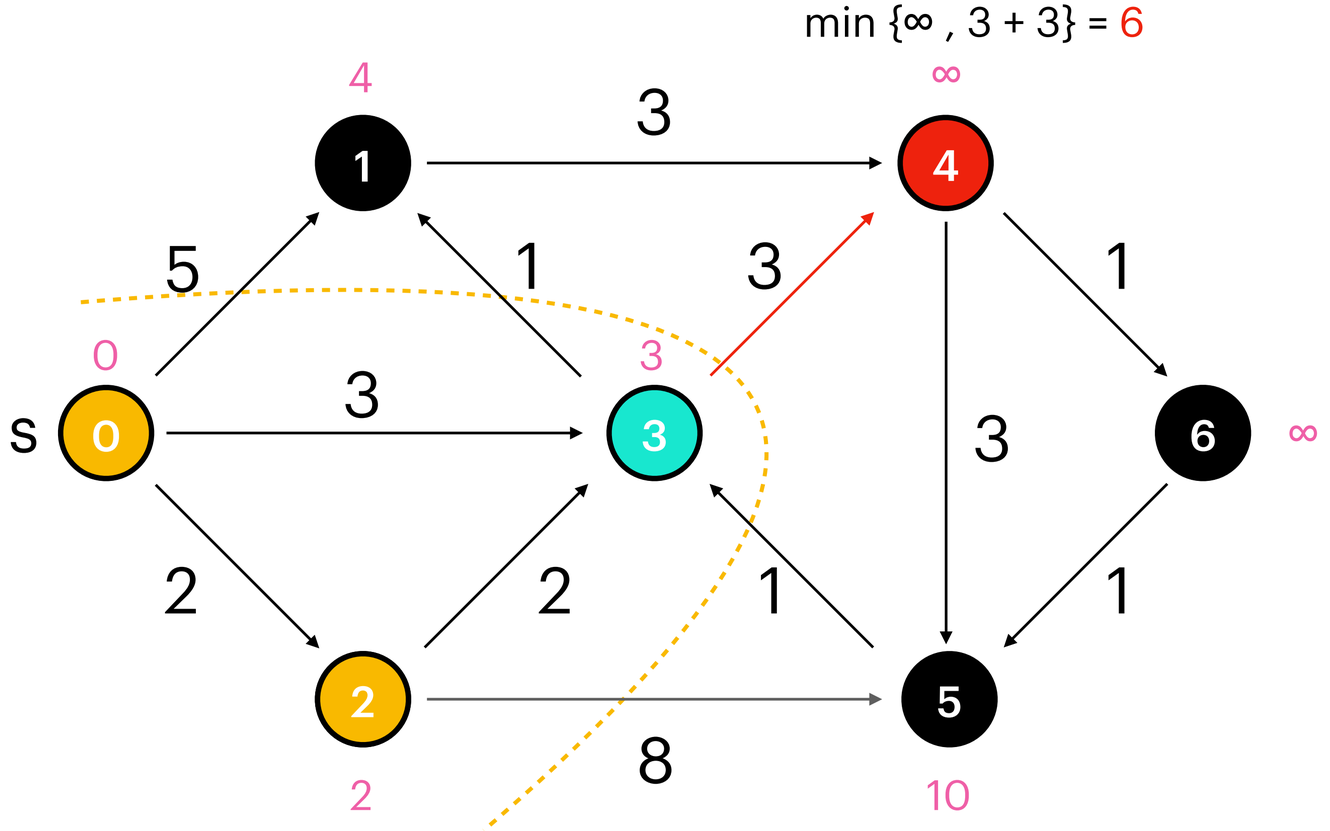
v* = 3

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	10	∞

"Heap" :

1	4	5	6
4	6	10	∞



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])
```

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

S : { 0 , 2 , 3 }

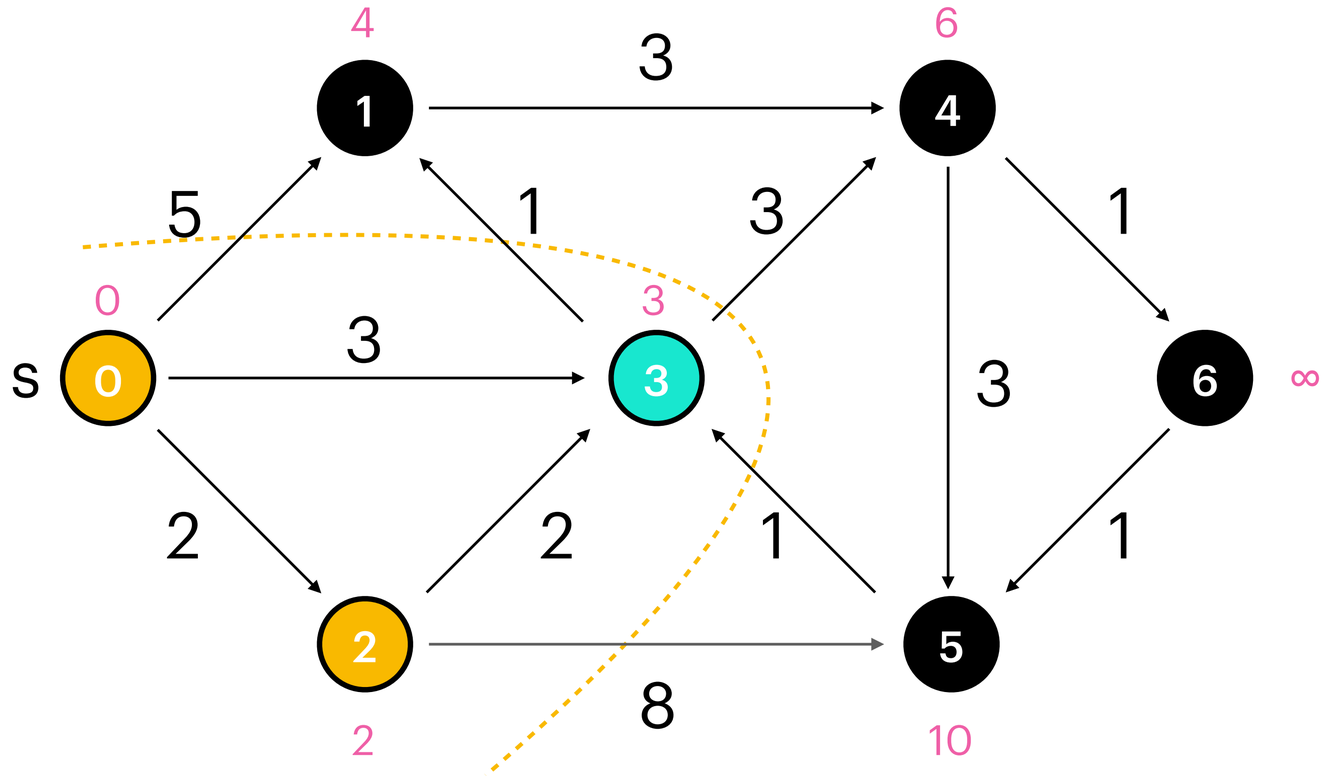
v* = 3

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	10	∞

"Heap" :

1	4	5	6
4	6	10	∞



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])
```

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

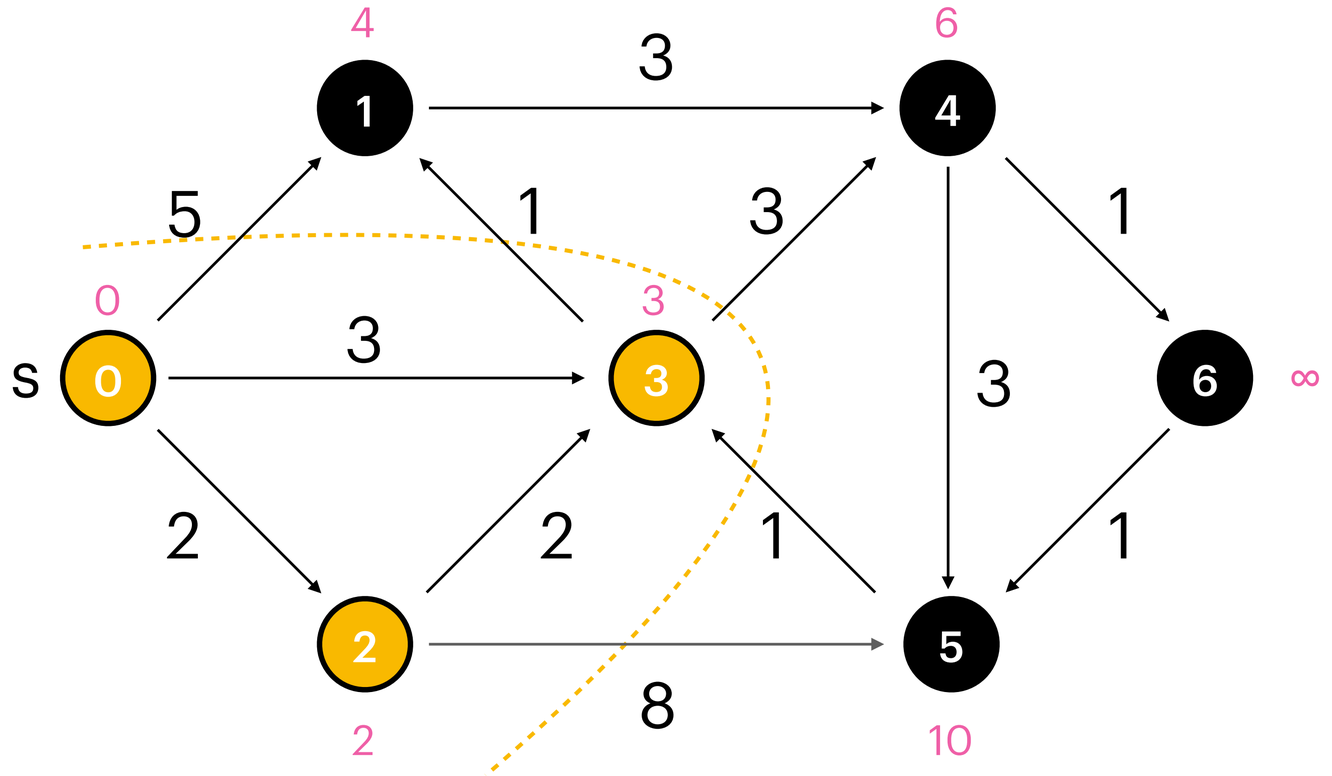
S : { 0 , 2 , 3 }

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	10	∞

"Heap" :

1	4	5	6
4	6	10	∞



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])
```

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

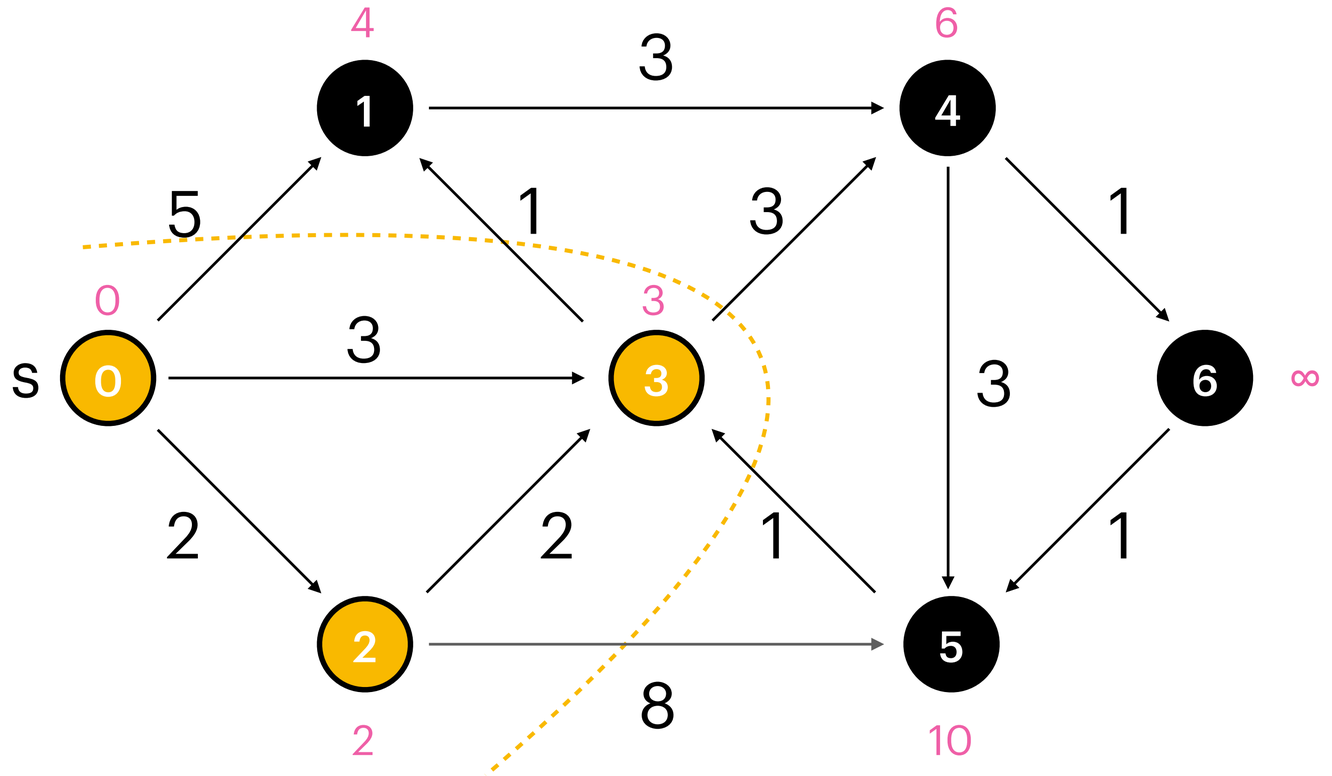
S : { 0 , 2 , 3 }

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	10	∞

"Heap" :

1	4	5	6
4	6	10	∞



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])
```

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

S : { 0 , 2 , 3 }

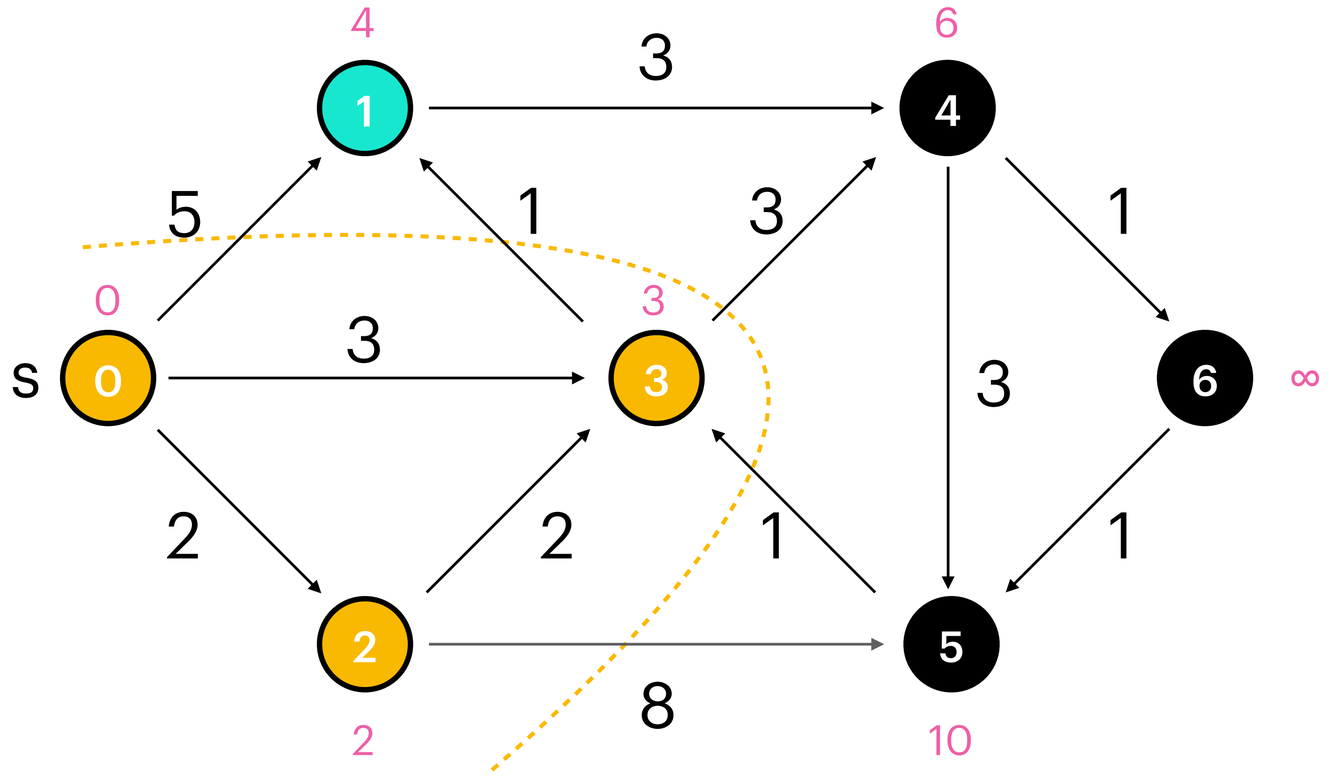
v* = 1

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	10	∞

"Heap" :

4	5	6
6	10	∞



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])
```

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

S : { 0 , 2 , 3 , 1 }

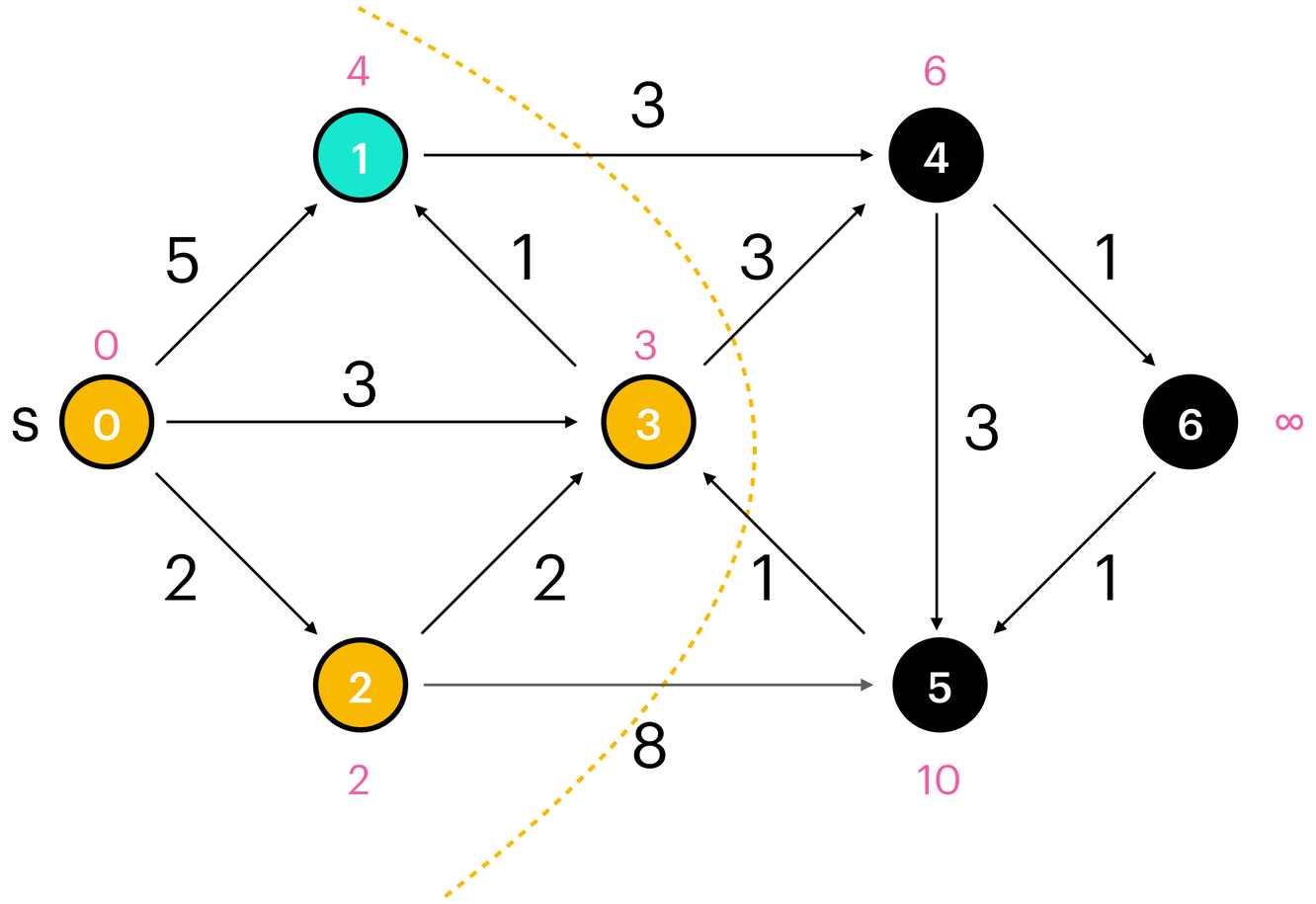
v* = 1

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	10	∞

"Heap" :

4	5	6
6	10	∞



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])
```

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

S : { 0 , 2 , 3 , 1 }

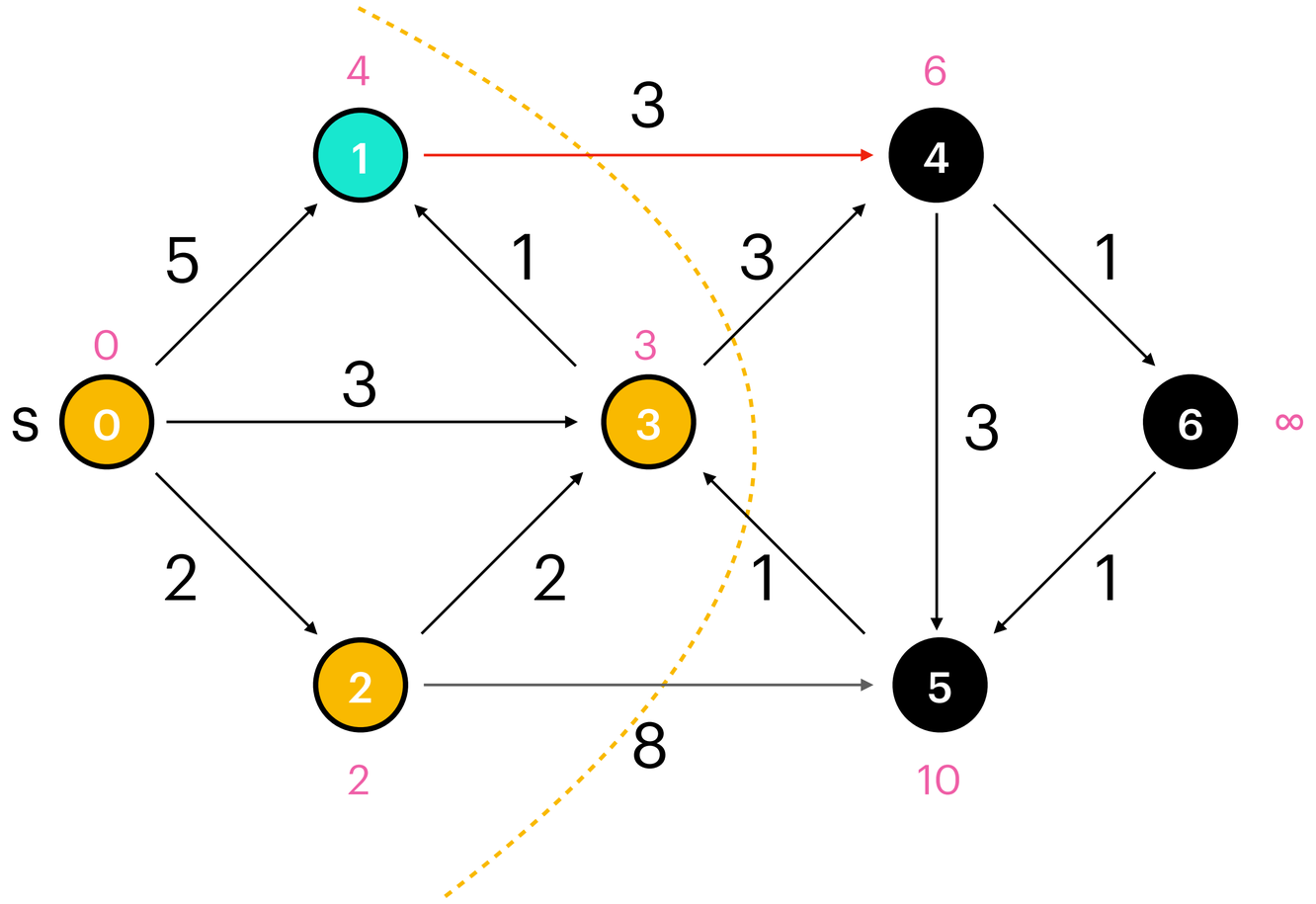
v* = 1

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	10	∞

"Heap" :

4	5	6
6	10	∞



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])
```

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

S : { 0 , 2 , 3 , 1 }

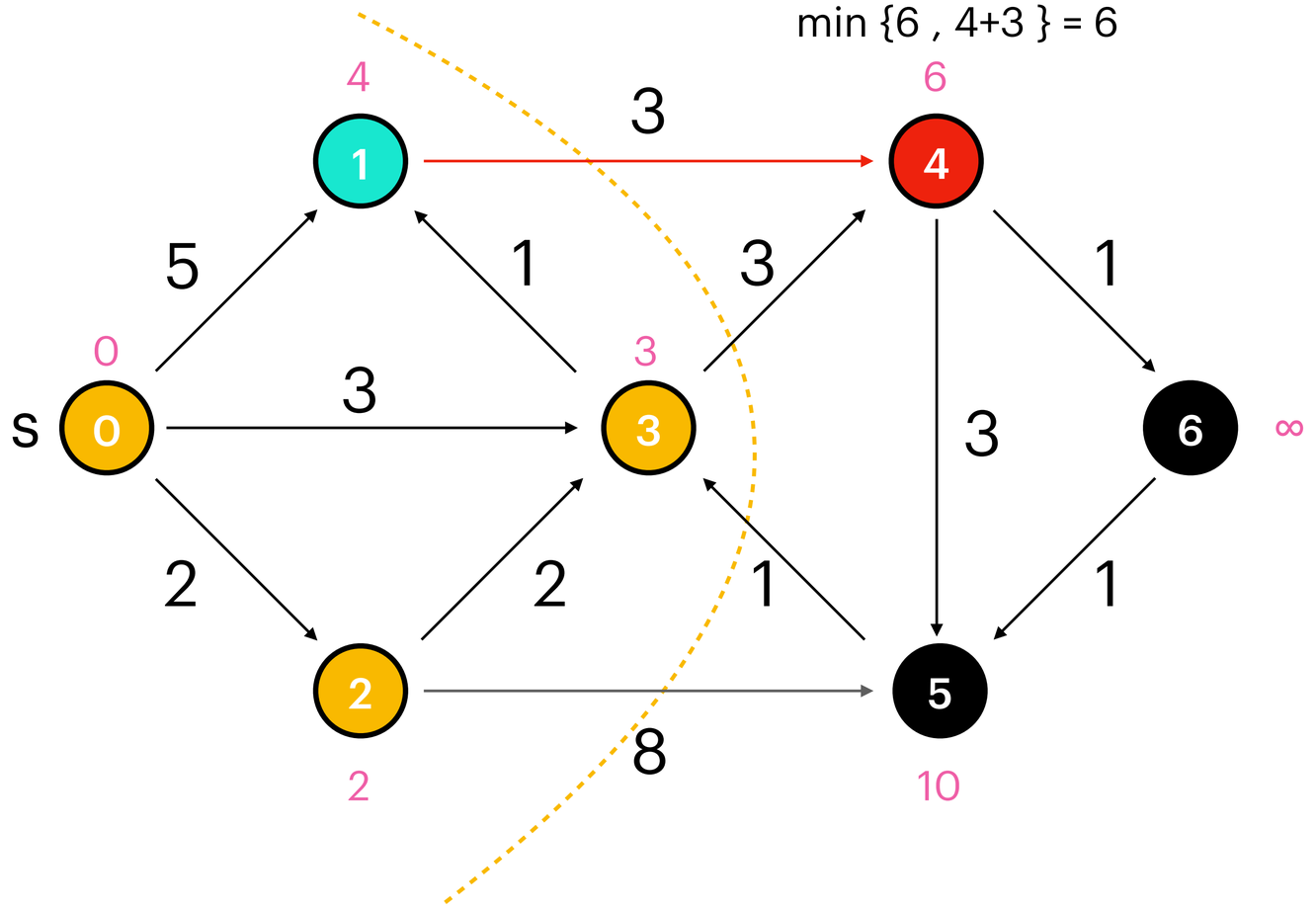
v* = 1

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	10	∞

"Heap" :

4	5	6
6	10	∞



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])
```

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

S : { 0 , 2 , 3 , 1 }

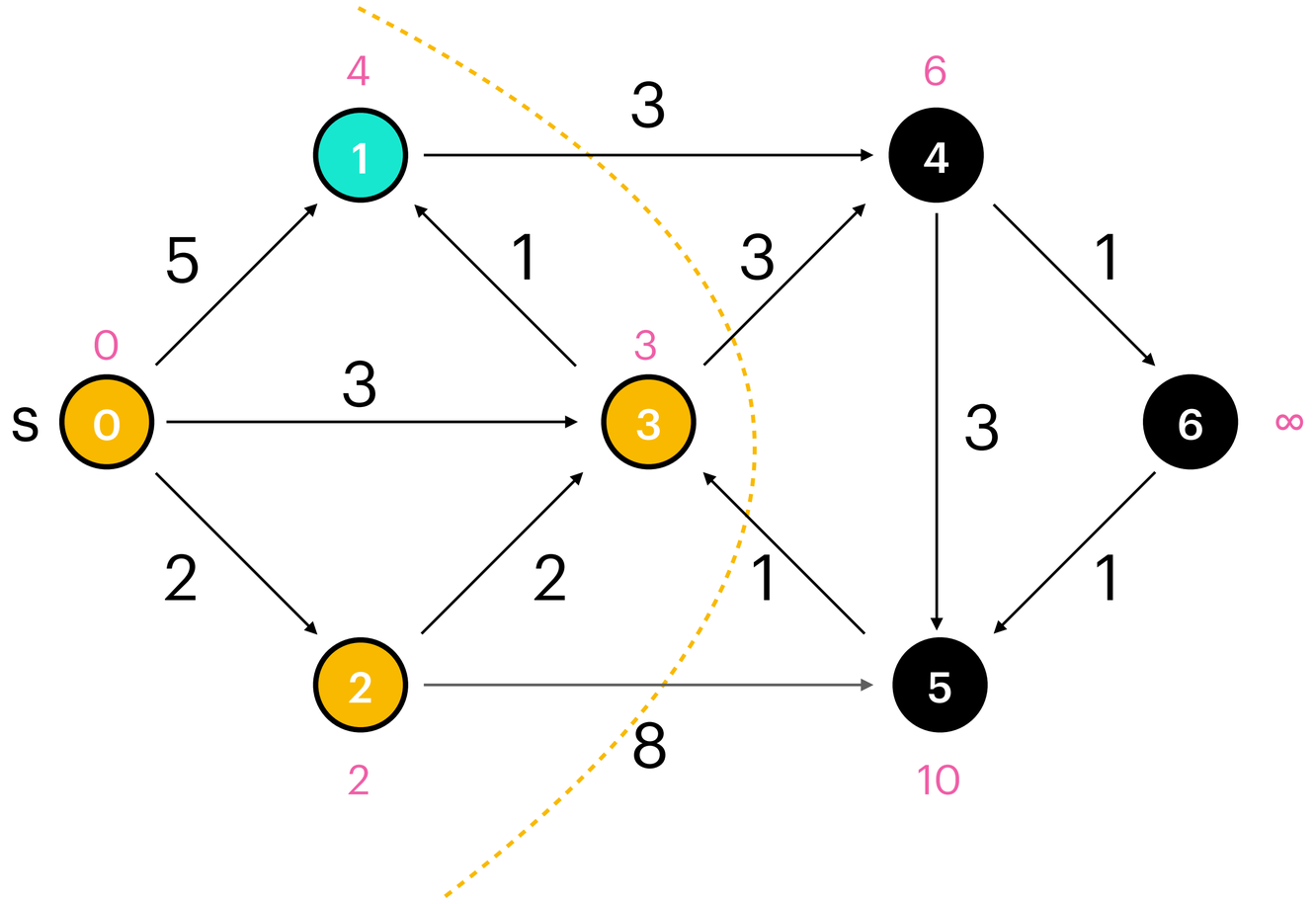
v* = 1

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	10	∞

"Heap" :

4	5	6
6	10	∞



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])
```

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

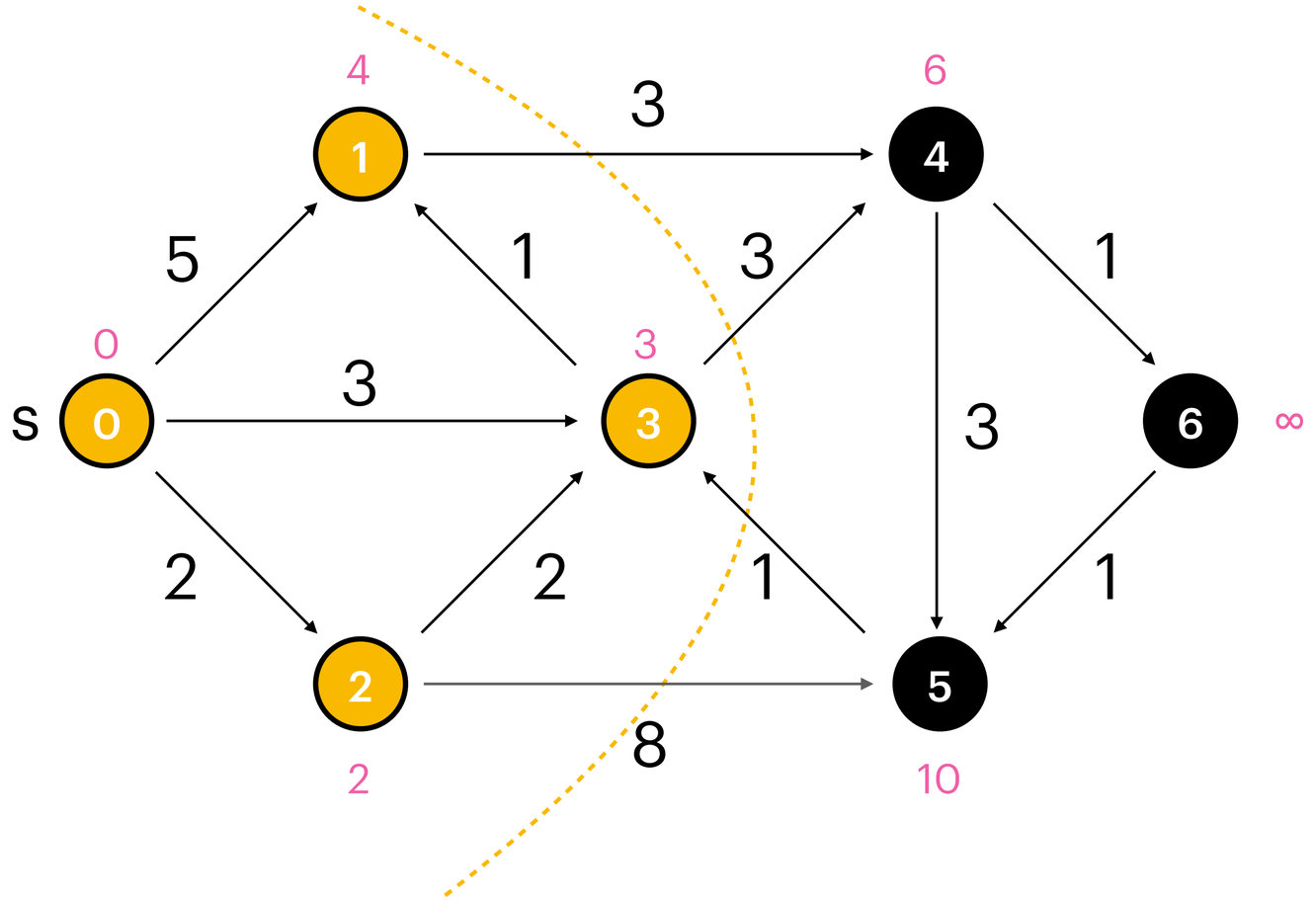
S : { 0 , 2 , 3 , 1 }

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	10	∞

"Heap" :

4	5	6
6	10	∞



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])
```

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

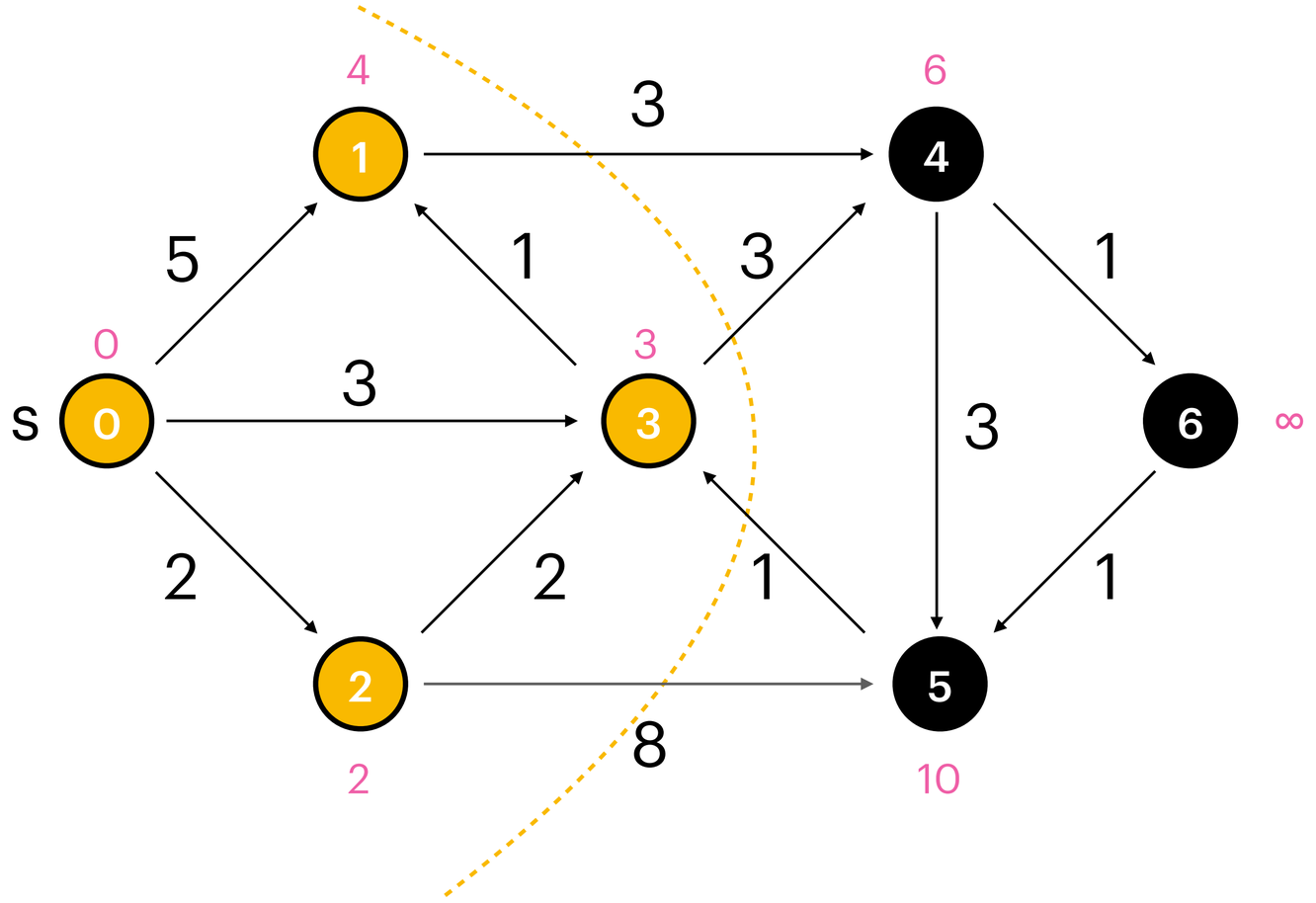
S : { 0 , 2 , 3 , 1 }

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	10	∞

"Heap" :

4	5	6
6	10	∞



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])
```

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

S : { 0 , 2 , 3 , 1 }

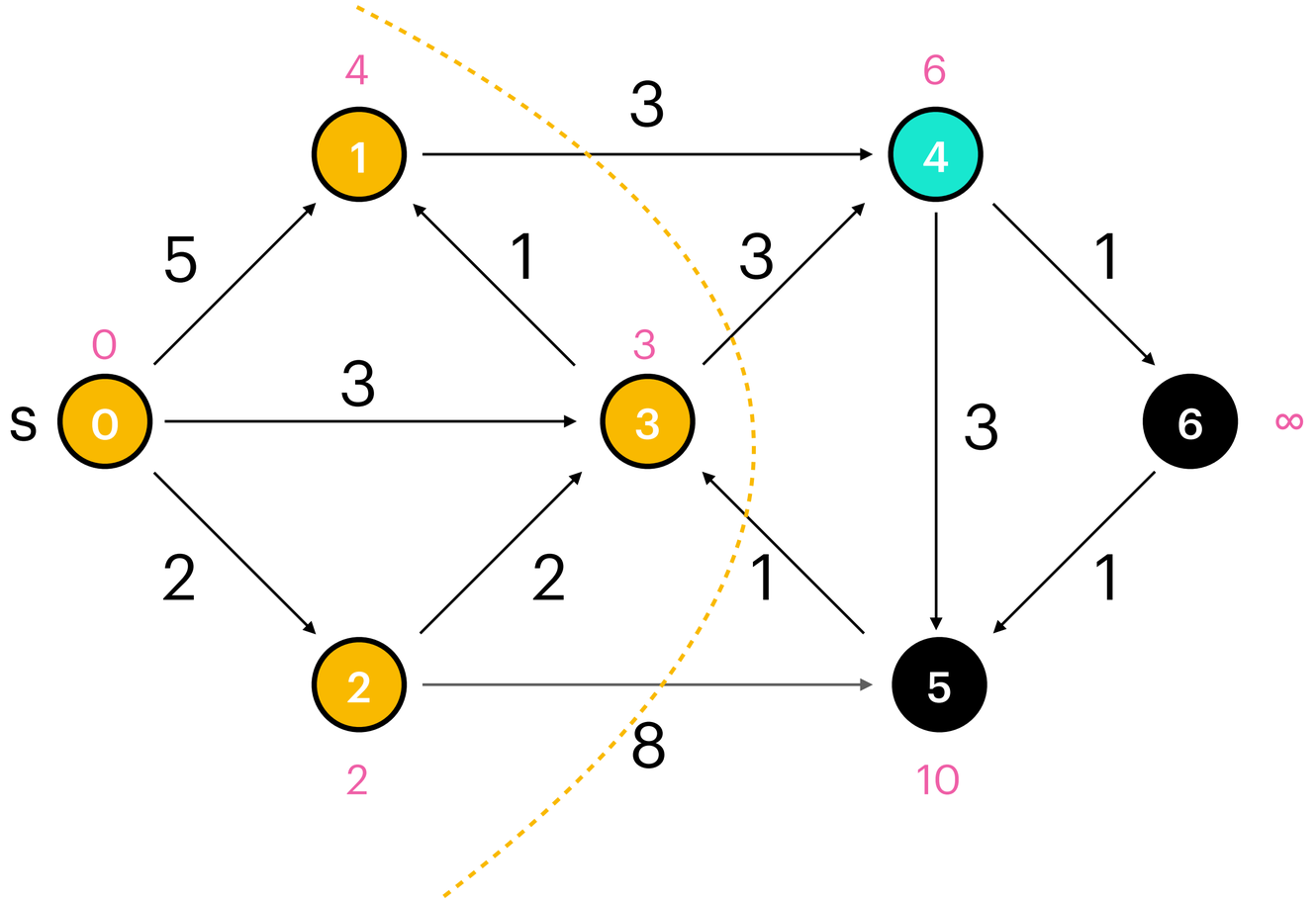
v* = 4

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	10	∞

"Heap" :

5	6
10	∞



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])
```

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

S : { 0 , 2 , 3 , 1 , 4 }

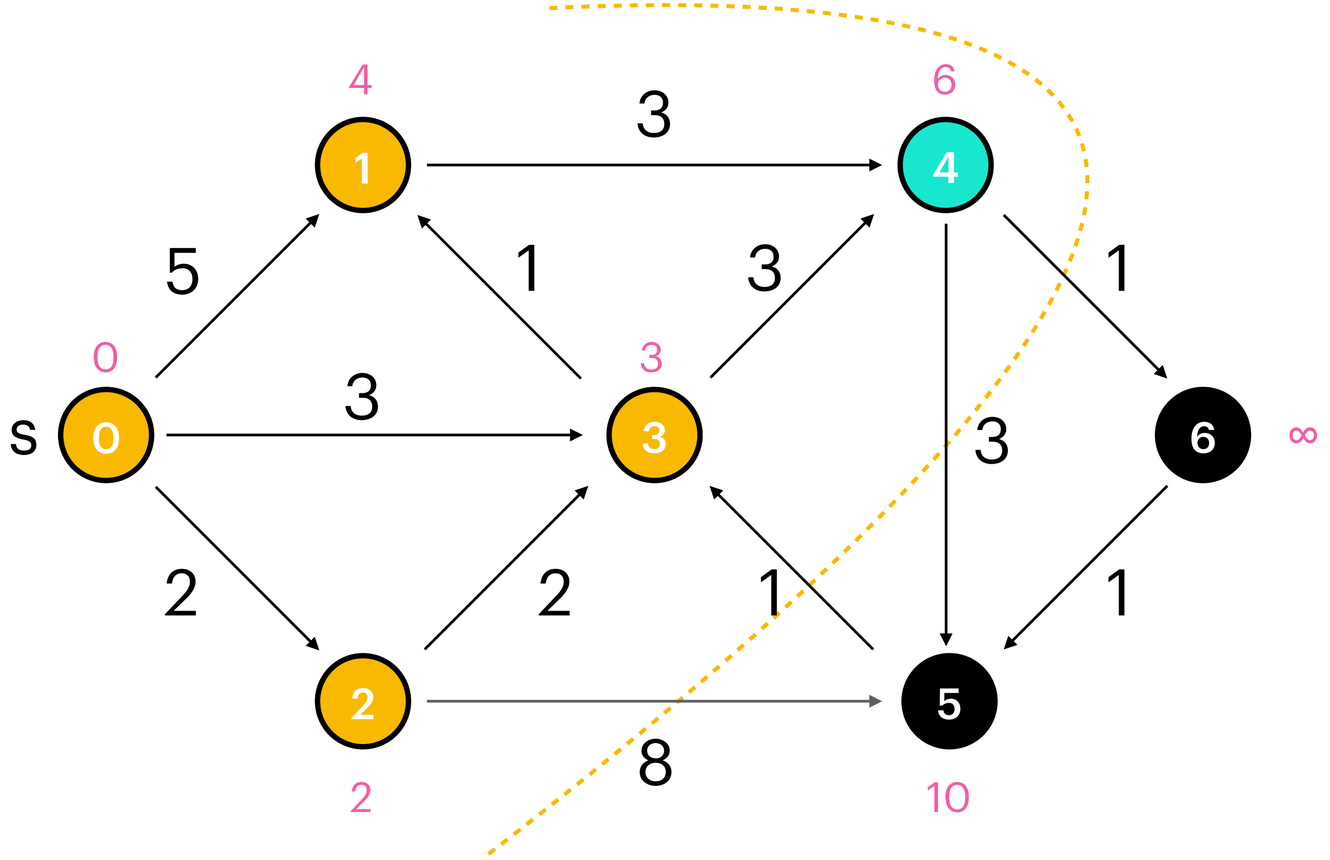
v* = 4

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	10	∞

"Heap" :

5	6
10	∞



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])
```

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

S : { 0 , 2 , 3 , 1 , 4 }

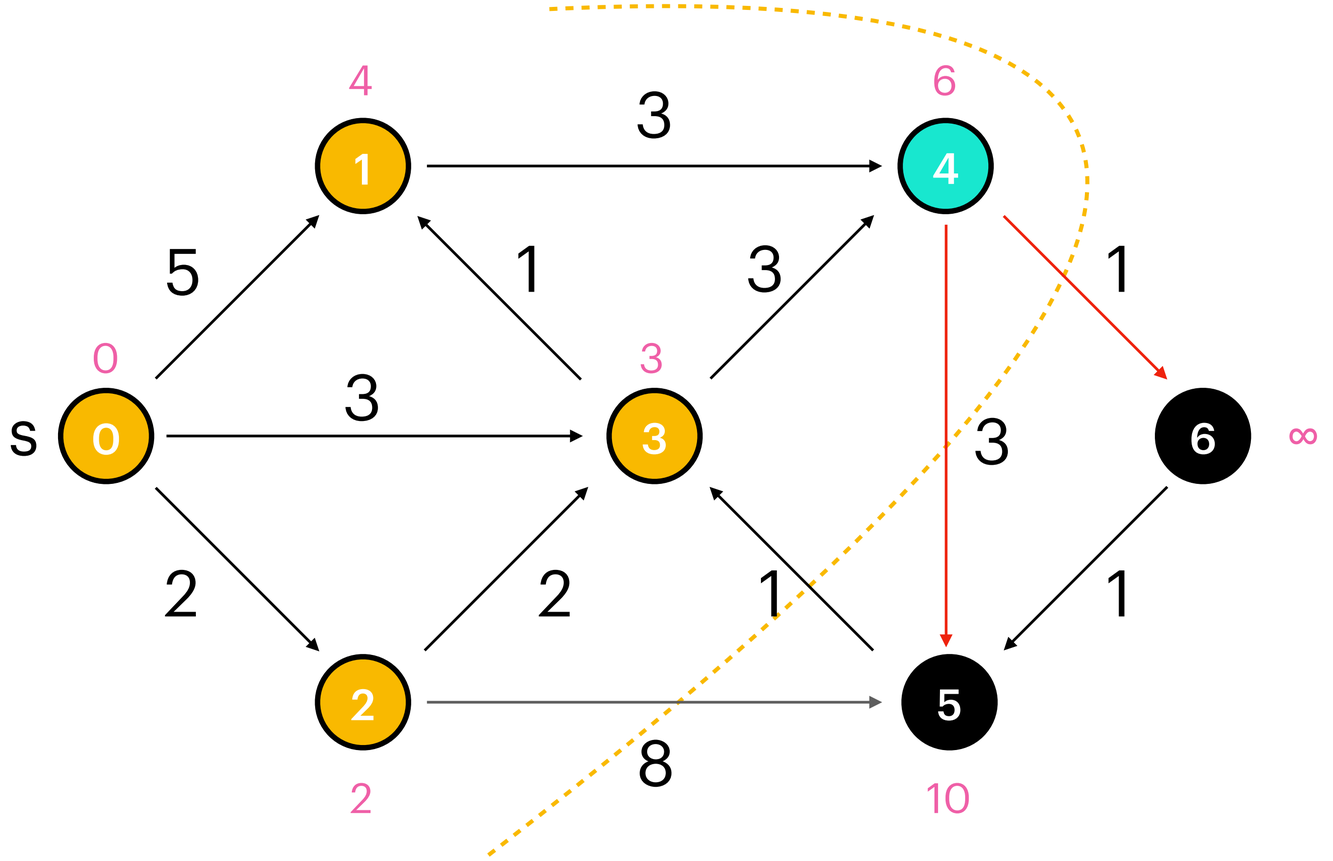
v* = 4

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	10	∞

"Heap" :

5	6
10	∞



Shortest Paths

Dijkstra's Algorithm

Algorithm 6 Dijkstra(s)

1: $d[s] \leftarrow 0$; $\quad d[v] \leftarrow \infty \ \ \forall v \in V \setminus \{s\}$
2: $S \leftarrow \varnothing$
3: $H \leftarrow$ make-heap(V); decrease-key($H, s, 0$)
4: **while** $S \neq V$ **do**
5: $\quad v^* \leftarrow$ extract-min(H)
6: $\quad S \leftarrow S \cup \{v^*\}$
7: $\quad$ **for** $(v^*, v) \in E,\ v \notin S$ **do**
8: $\quad\quad d[v] \leftarrow \min\{d[v], d[v^*] + c(v^*, v)\}$
9: $\quad\quad$ decrease-key($H, v, d[v]$)

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

S : { 0 , 2 , 3 , 1 , 4 }

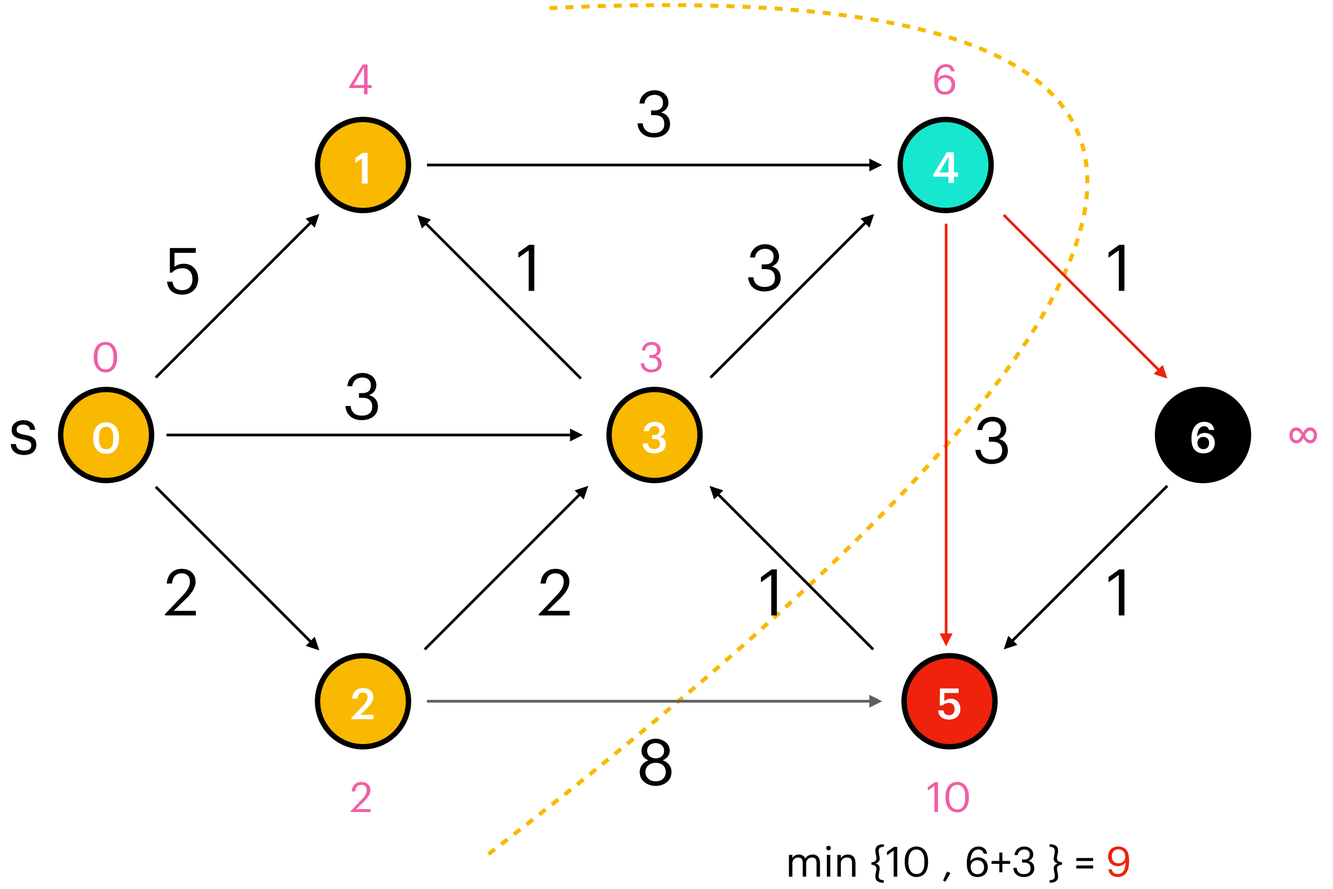
v* = 4

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	9	∞

"Heap" :

5	6
9	∞



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])
```

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

S : { 0 , 2 , 3 , 1 , 4 }

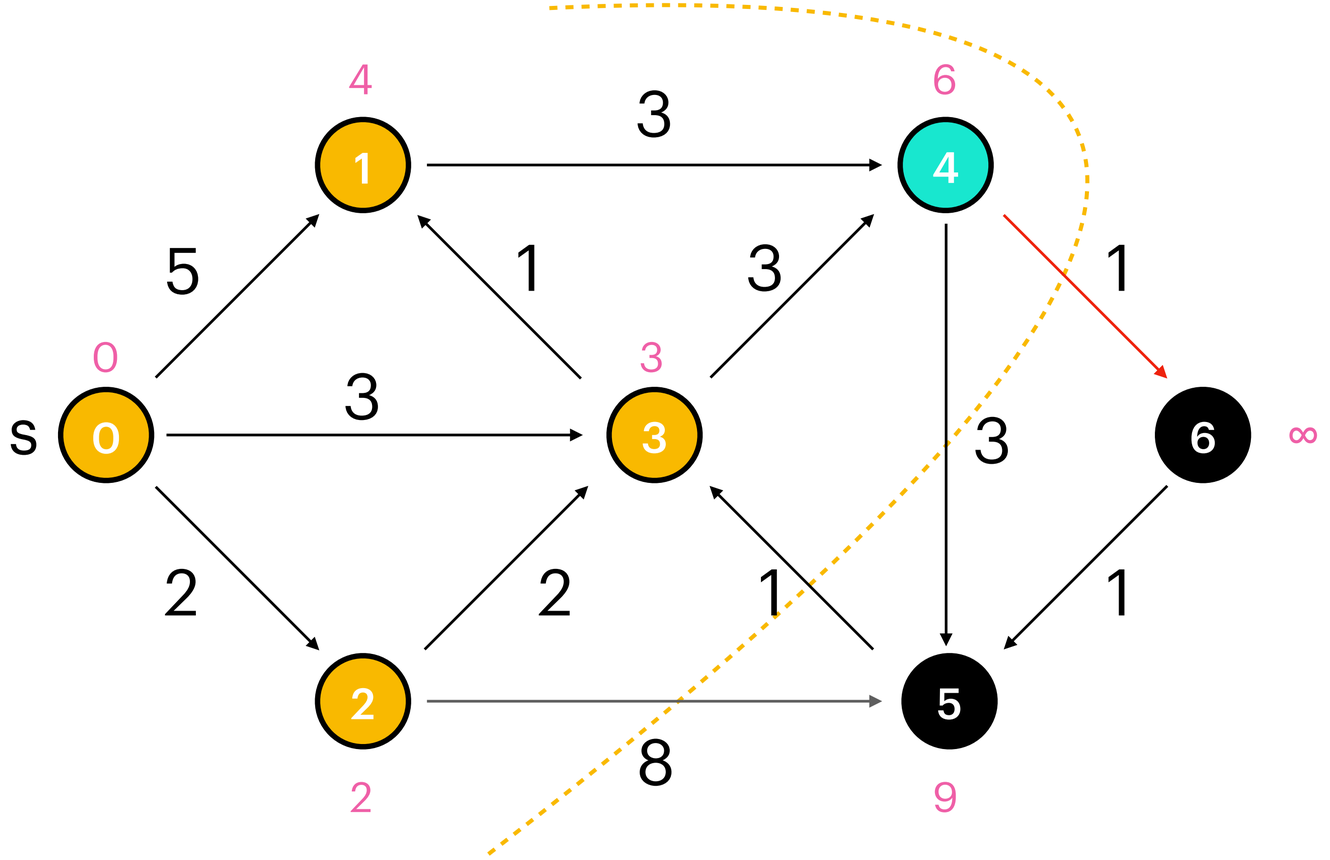
v* = 4

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	9	∞

"Heap" :

5	6
9	∞



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])
```

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

S : { 0 , 2 , 3 , 1 , 4 }

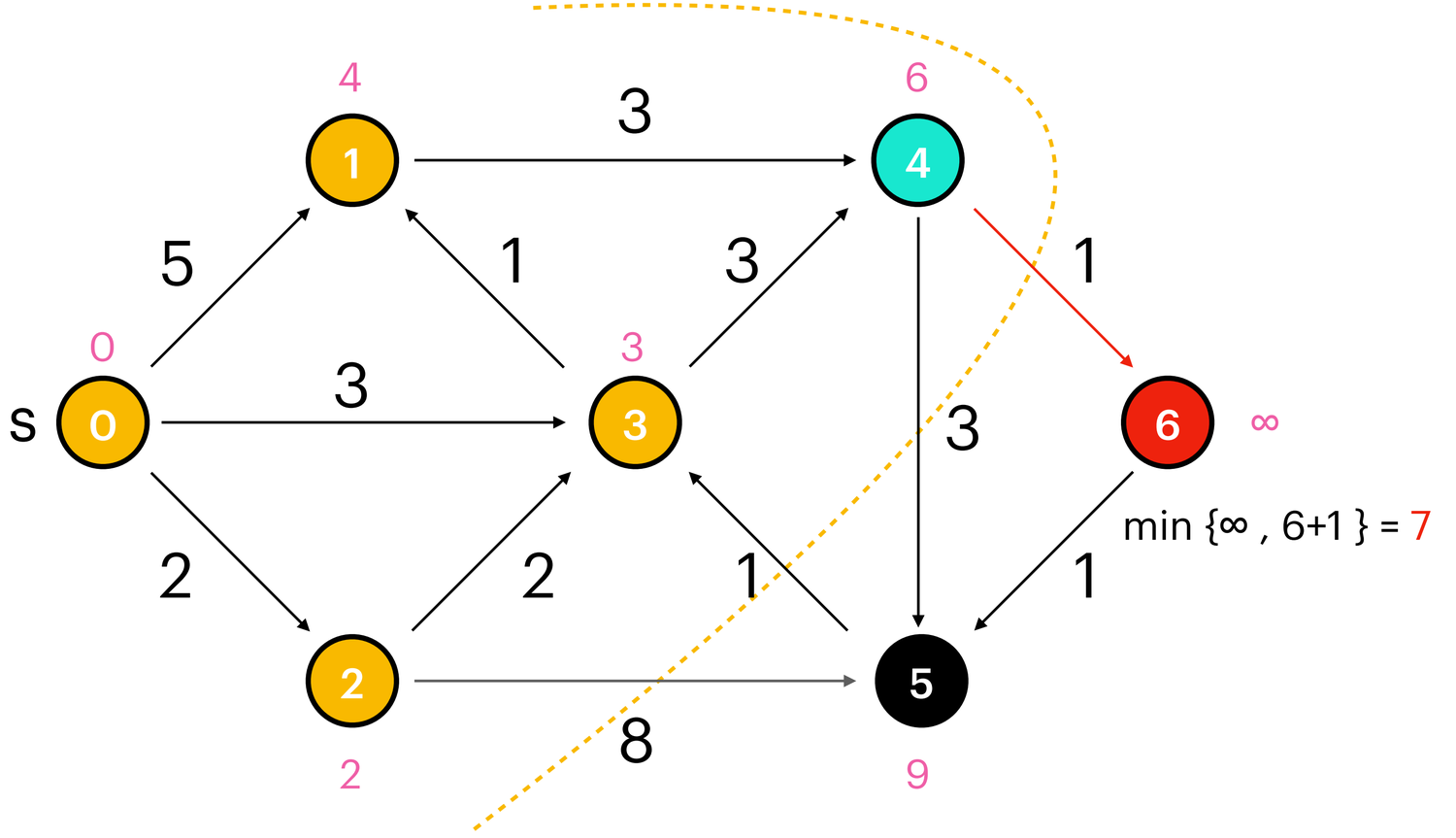
v* = 4

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	9	7

"Heap" :

5	6
9	7



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])
```

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

S : { 0 , 2 , 3 , 1 , 4 }

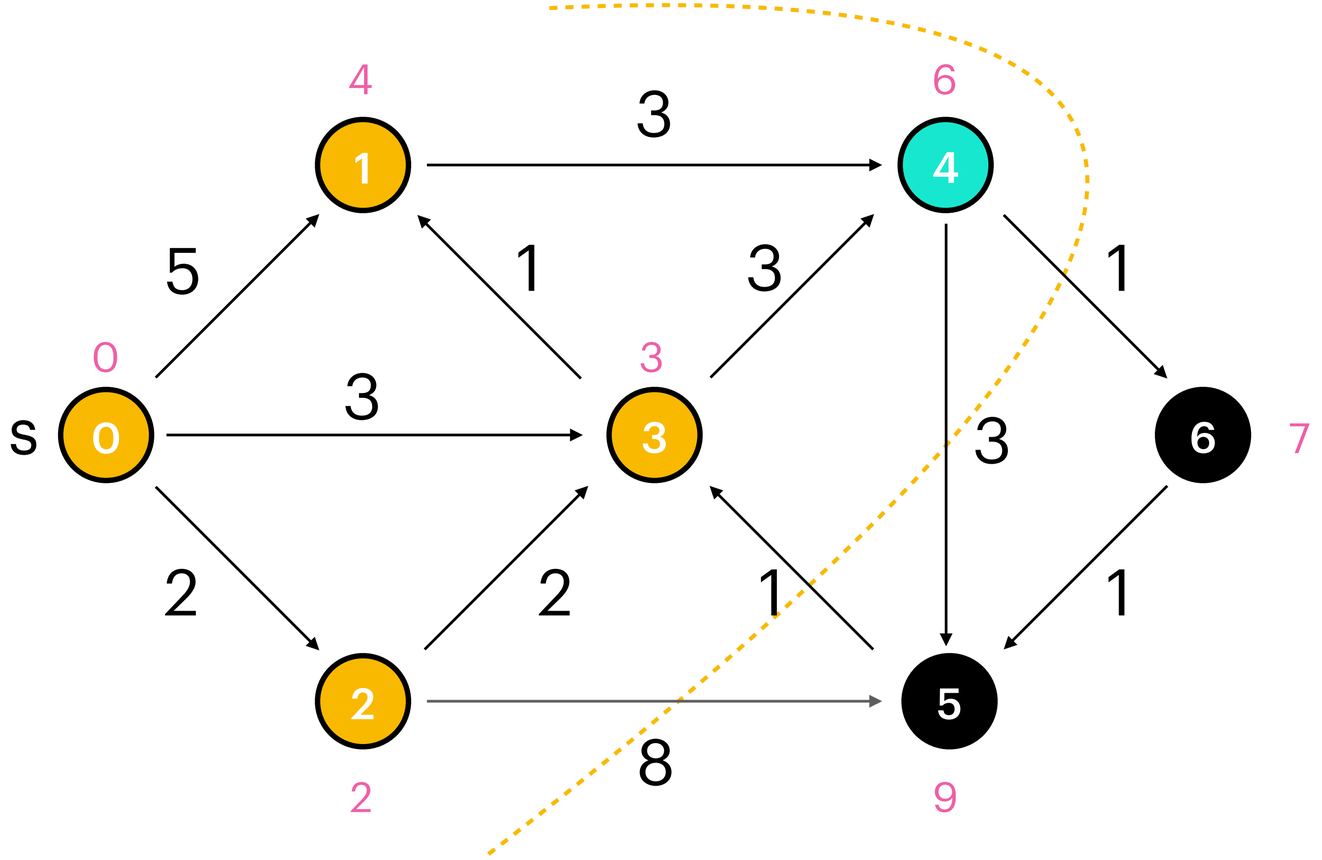
v* = 4

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	9	7

"Heap" :

5	6
9	7



Shortest Paths

Dijkstra's Algorithm

Algorithm 6 Dijkstra(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \ \ \forall v \in V \setminus \{s\}$
2: $S \leftarrow \varnothing$
3: $H \leftarrow$ make-heap(V); decrease-key($H, s, 0$)
4: **while** $S \neq V$ **do**
5: $\quad v^* \leftarrow$ extract-min(H)
6: $\quad S \leftarrow S \cup \{v^*\}$
7: $\quad$ **for** $(v^*, v) \in E,\ v \notin S$ **do**
8: $\quad\quad \mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[v^*] + c(v^*, v)\}$
9: $\quad\quad$ decrease-key($H, v, \mathrm{d}[v]$)

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

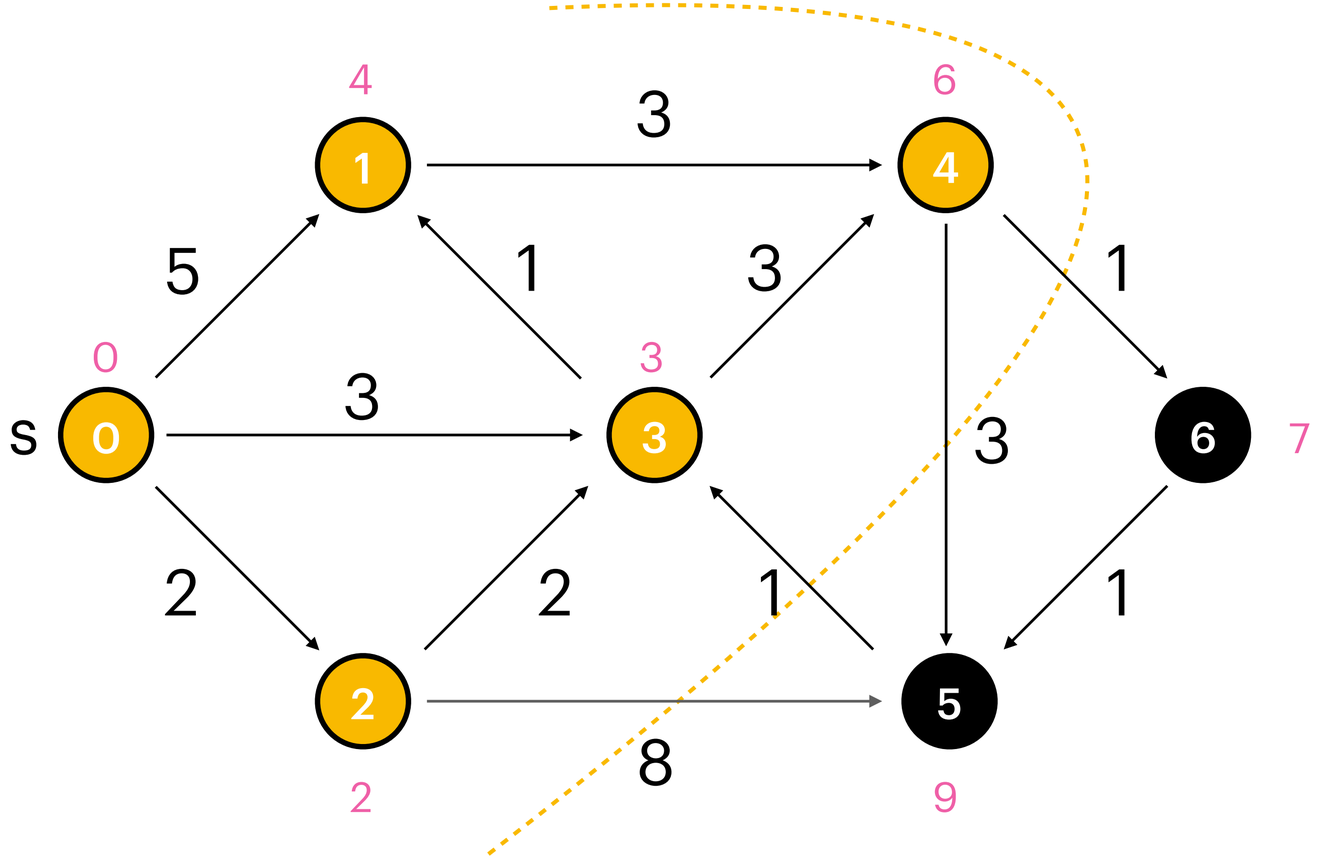
S : { 0 , 2 , 3 , 1 , 4 }

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	9	7

"Heap" :

5	6
9	7



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])
```

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

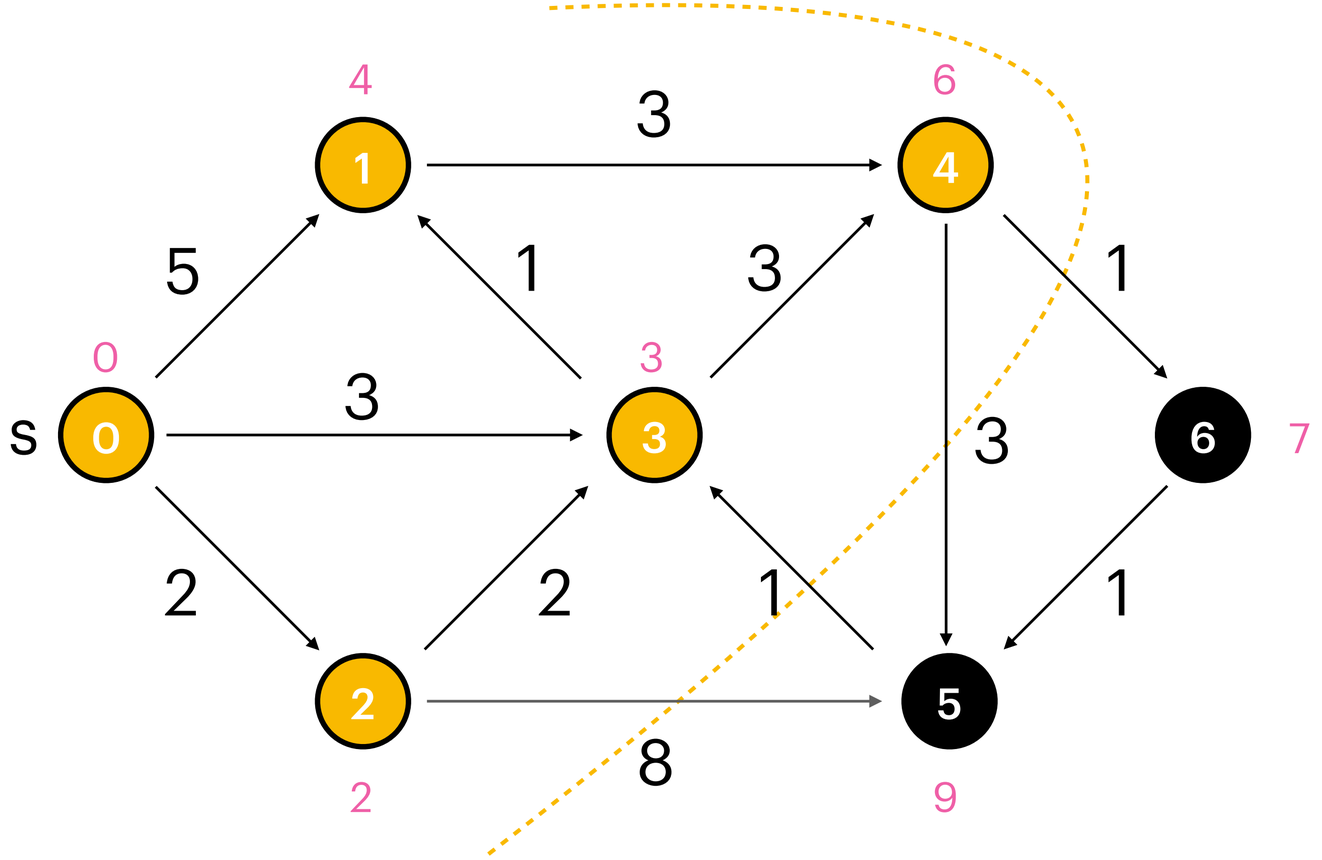
S : { 0 , 2 , 3 , 1 , 4 }

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	9	7

"Heap" :

5	6
9	7



Shortest Paths

Dijkstra's Algorithm

Algorithm 6 Dijkstra(s)

1: $d[s] \leftarrow 0$; $\quad d[v] \leftarrow \infty \ \ \forall v \in V \setminus \{s\}$
2: $S \leftarrow \varnothing$
3: $H \leftarrow$ make-heap(V); decrease-key($H, s, 0$)
4: **while** $S \neq V$ **do**
5: $\quad v^* \leftarrow$ extract-min(H)
6: $\quad S \leftarrow S \cup \{v^*\}$
7: $\quad$ **for** $(v^*, v) \in E,\ v \notin S$ **do**
8: $\quad\quad d[v] \leftarrow \min\{d[v], d[v^*] + c(v^*, v)\}$
9: $\quad\quad$ decrease-key($H, v, d[v]$)

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

S : { 0 , 2 , 3 , 1 , 4 }

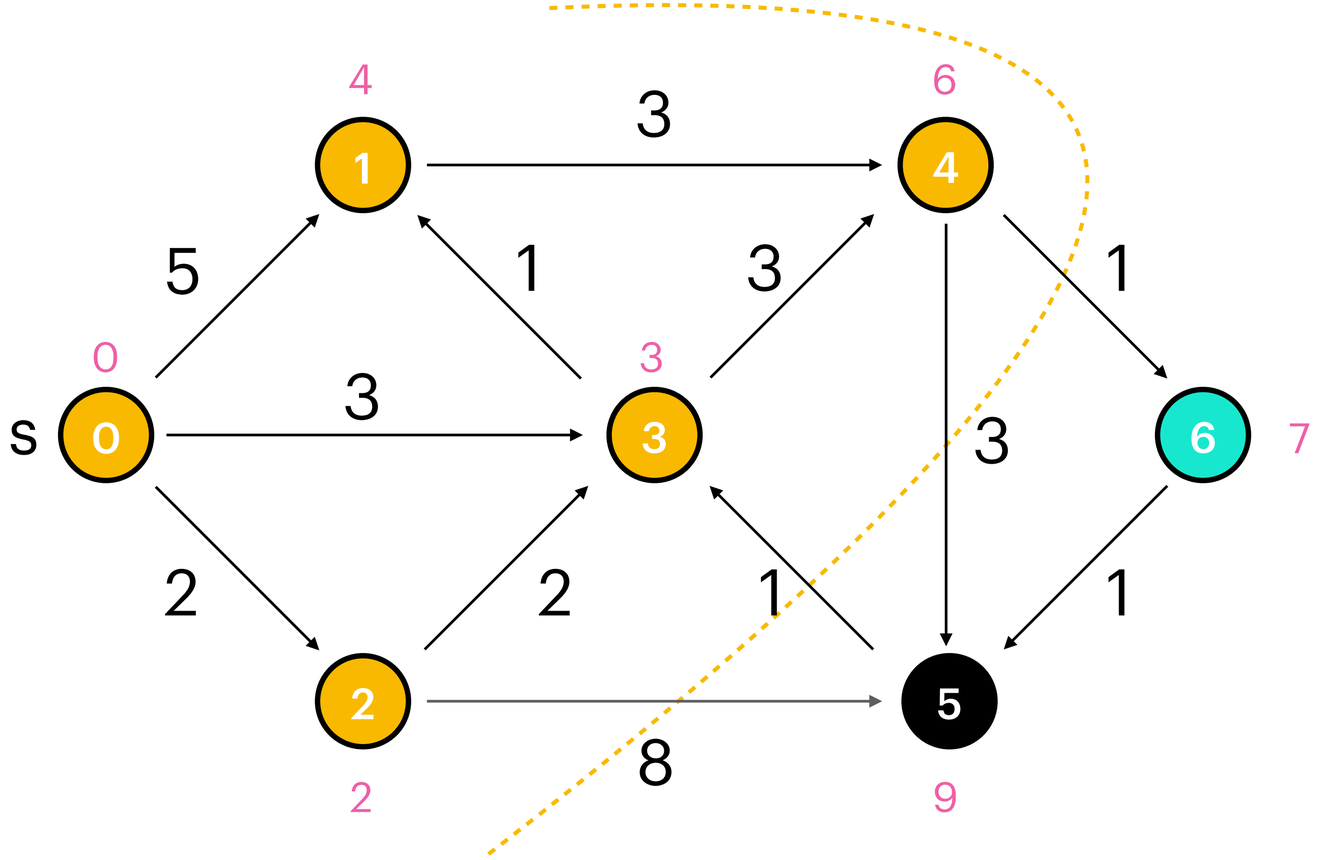
v* = 6

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	9	7

"Heap" :

5
9



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])
```

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

S : { 0 , 2 , 3 , 1 , 4 , 6 }

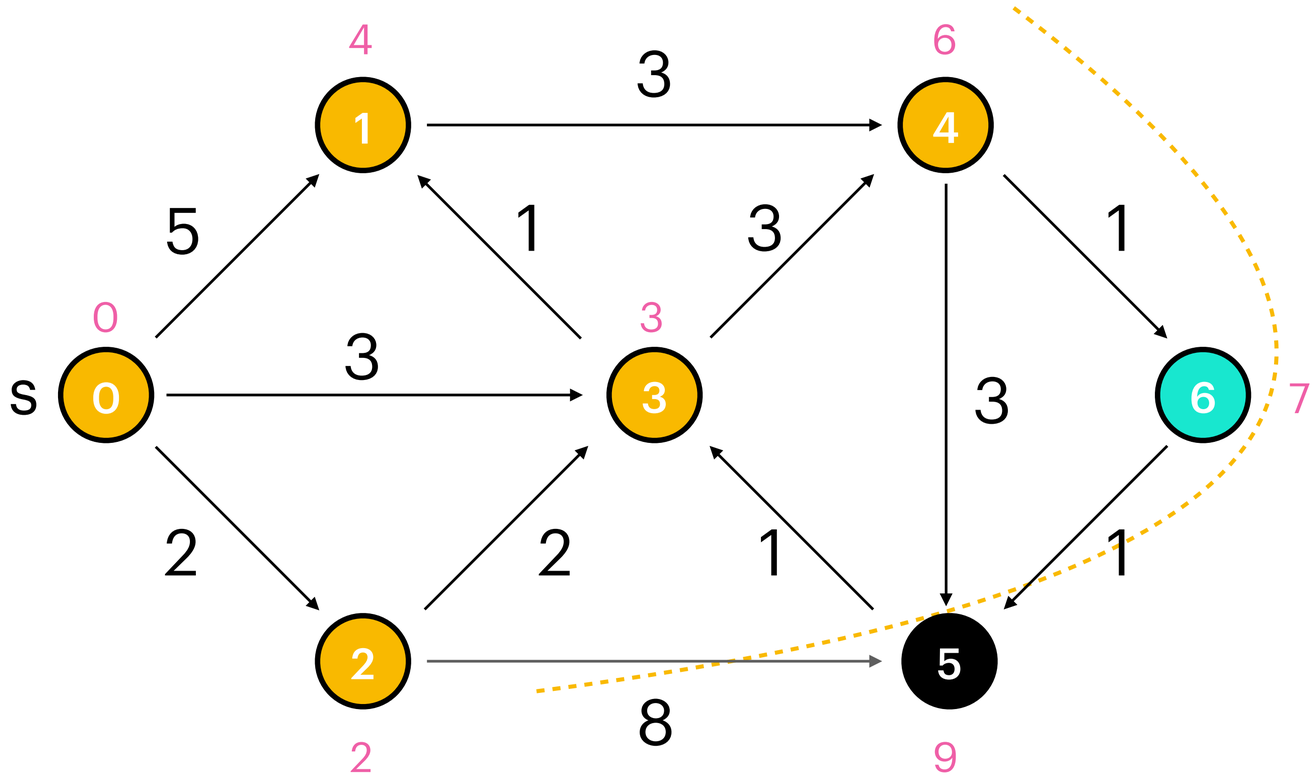
v* = 6

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	9	7

"Heap" :

5
9



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])
```

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

S : { 0 , 2 , 3 , 1 , 4 , 6 }

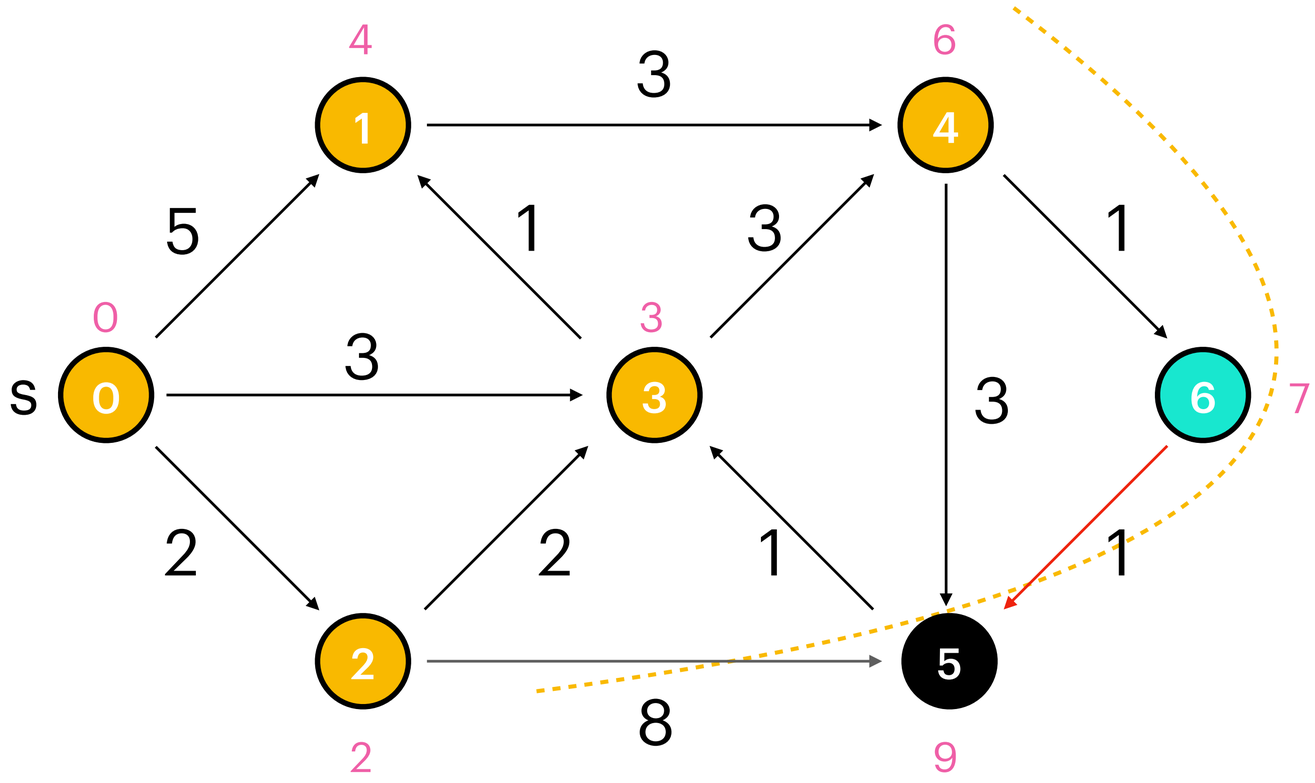
v* = 6

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	9	7

"Heap" :

5
9



Shortest Paths

Dijkstra's Algorithm

Algorithm 6 Dijkstra(s)

1: $d[s] \leftarrow 0$; $\quad d[v] \leftarrow \infty \ \ \forall v \in V \setminus \{s\}$
2: $S \leftarrow \varnothing$
3: $H \leftarrow$ make-heap(V); decrease-key($H, s, 0$)
4: **while** $S \neq V$ **do**
5: $\quad v^* \leftarrow$ extract-min(H)
6: $\quad S \leftarrow S \cup \{v^*\}$
7: $\quad$ **for** $(v^*, v) \in E,\ v \notin S$ **do**
8: $\quad\quad d[v] \leftarrow \min\{d[v], d[v^*] + c(v^*, v)\}$
9: $\quad\quad$ decrease-key($H, v, d[v]$)

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

S : { 0 , 2 , 3 , 1 , 4 , 6 }

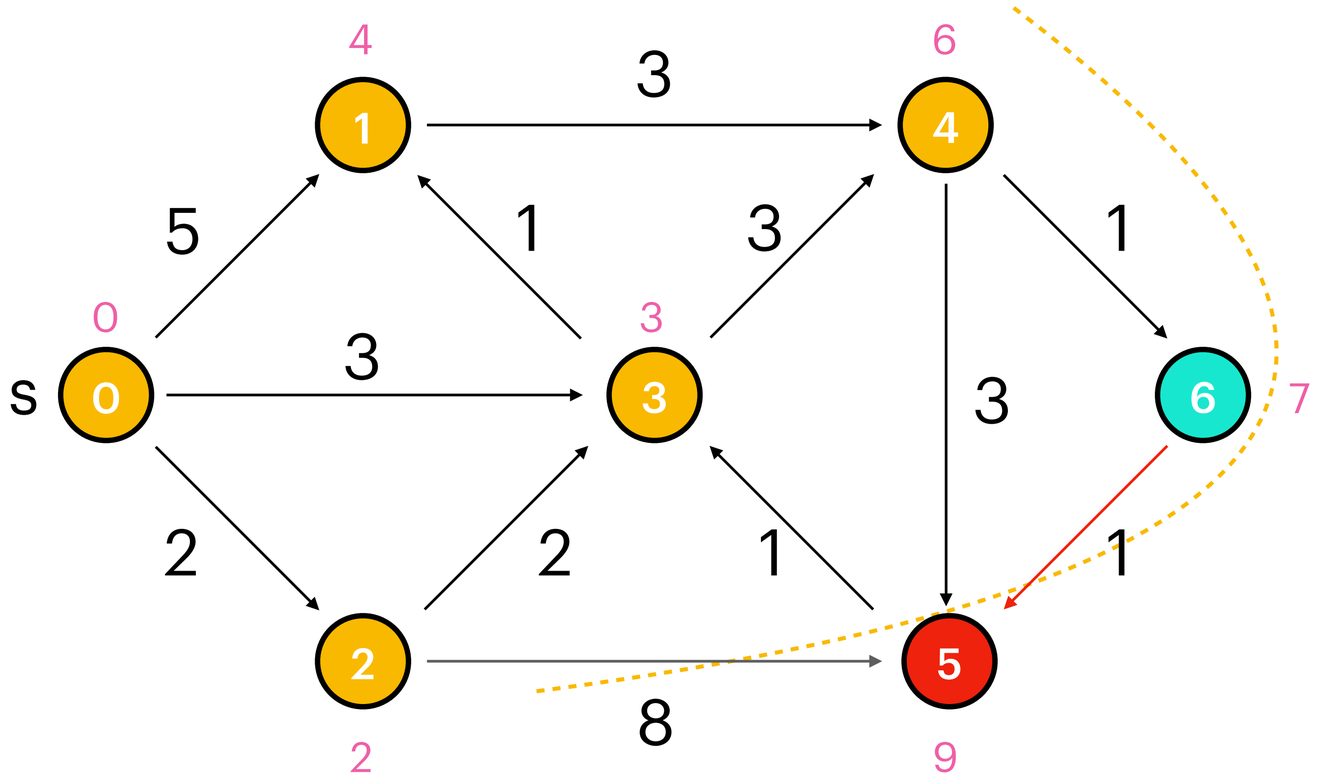
v* = 6

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	8	7

"Heap" :

5
8



min {9 , 7+1 } = 8

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])
```

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

S : { 0 , 2 , 3 , 1 , 4 , 6 }

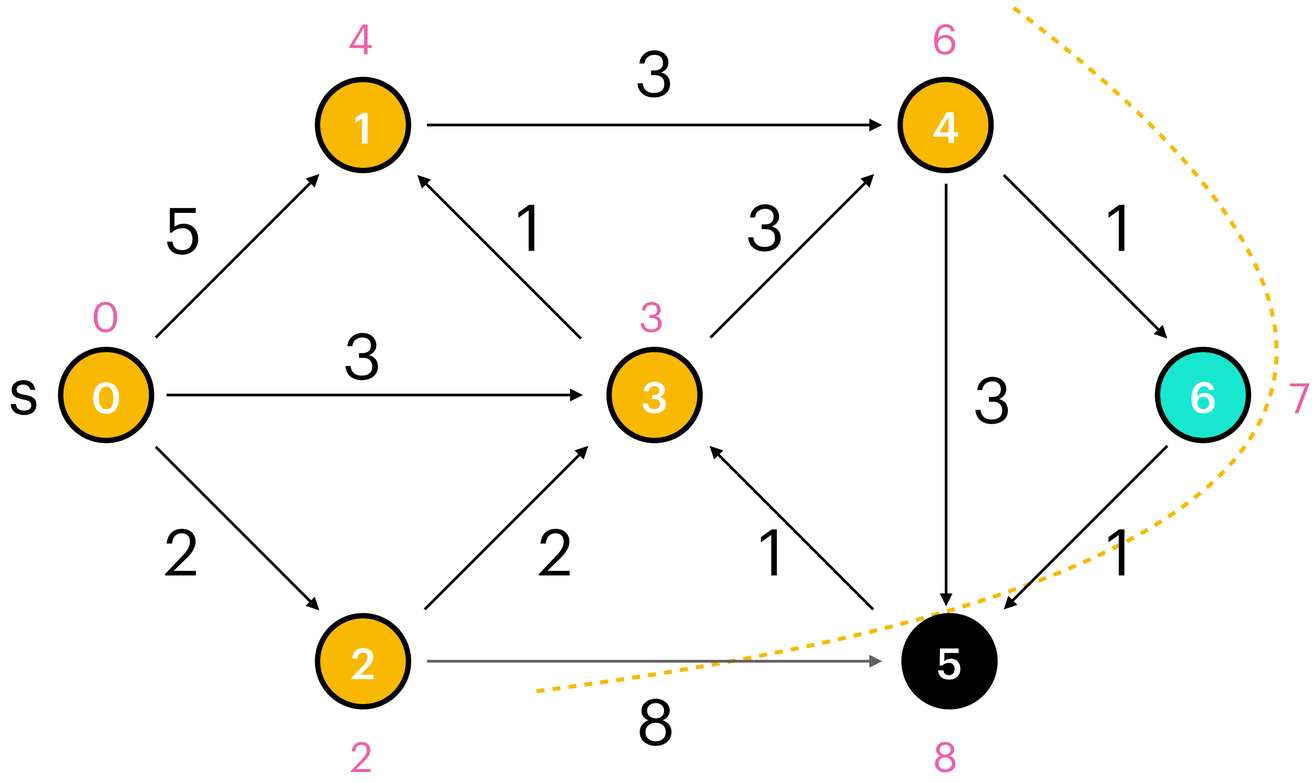
v* = 6

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	8	7

"Heap" :

5
8



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])
```

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

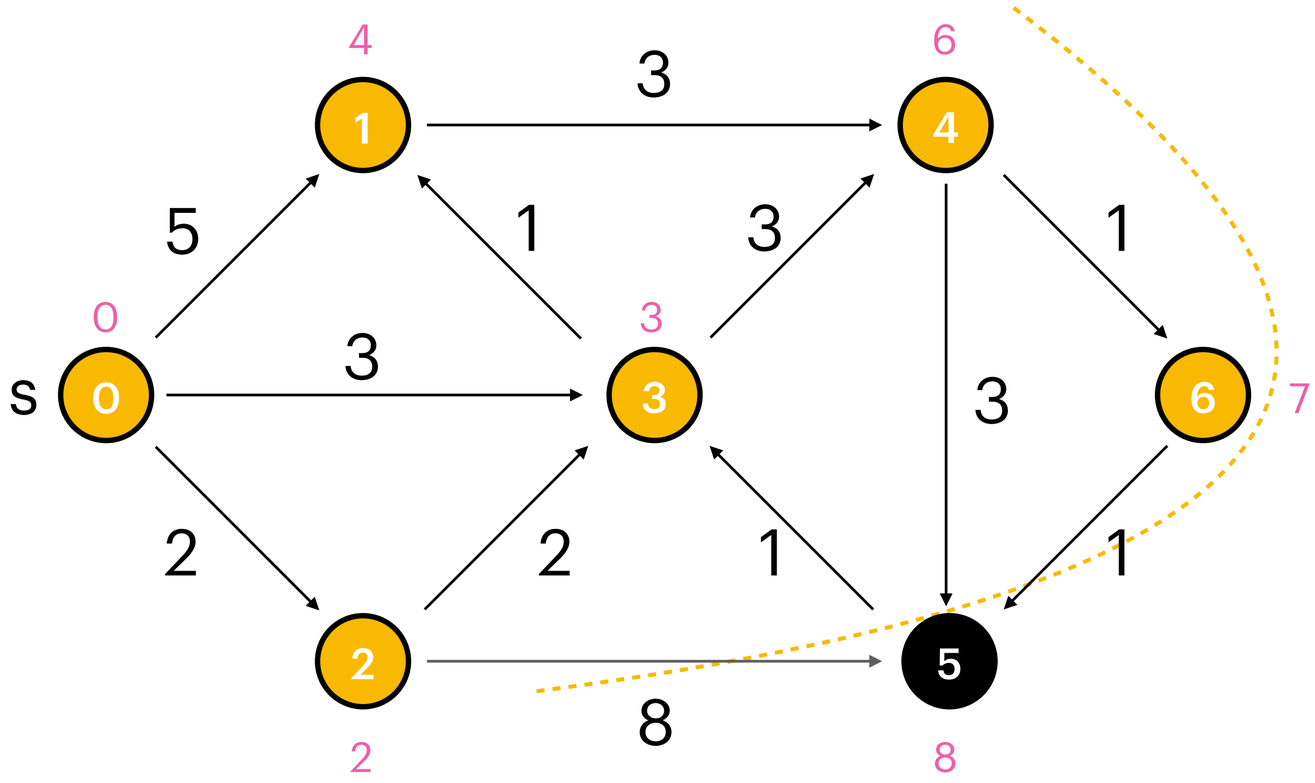
S : { 0 , 2 , 3 , 1 , 4 , 6 }

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	8	7

"Heap" :

5
8



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])
```

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

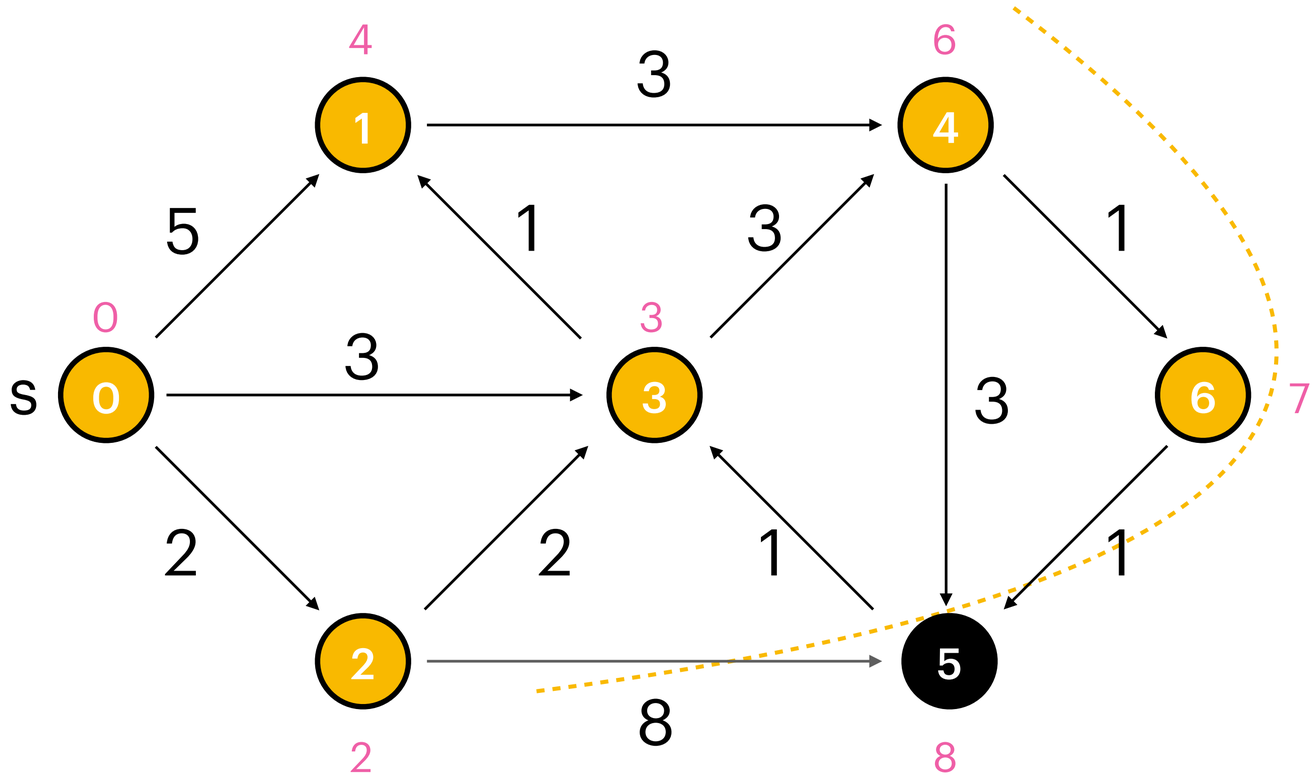
S : { 0 , 2 , 3 , 1 , 4 , 6 }

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	8	7

"Heap" :

5
8



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])
```

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

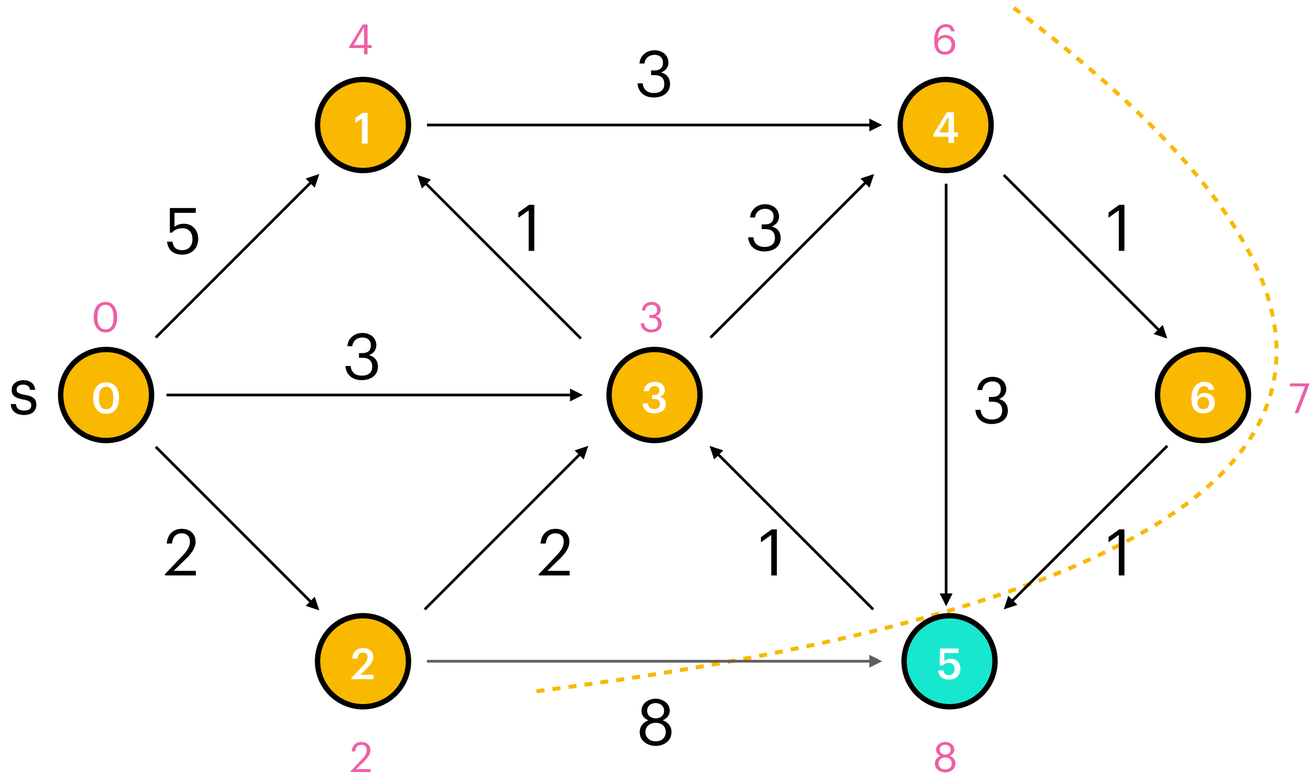
S : { 0 , 2 , 3 , 1 , 4 , 6 }

v* = 5

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	8	7

"Heap" :



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])
```

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

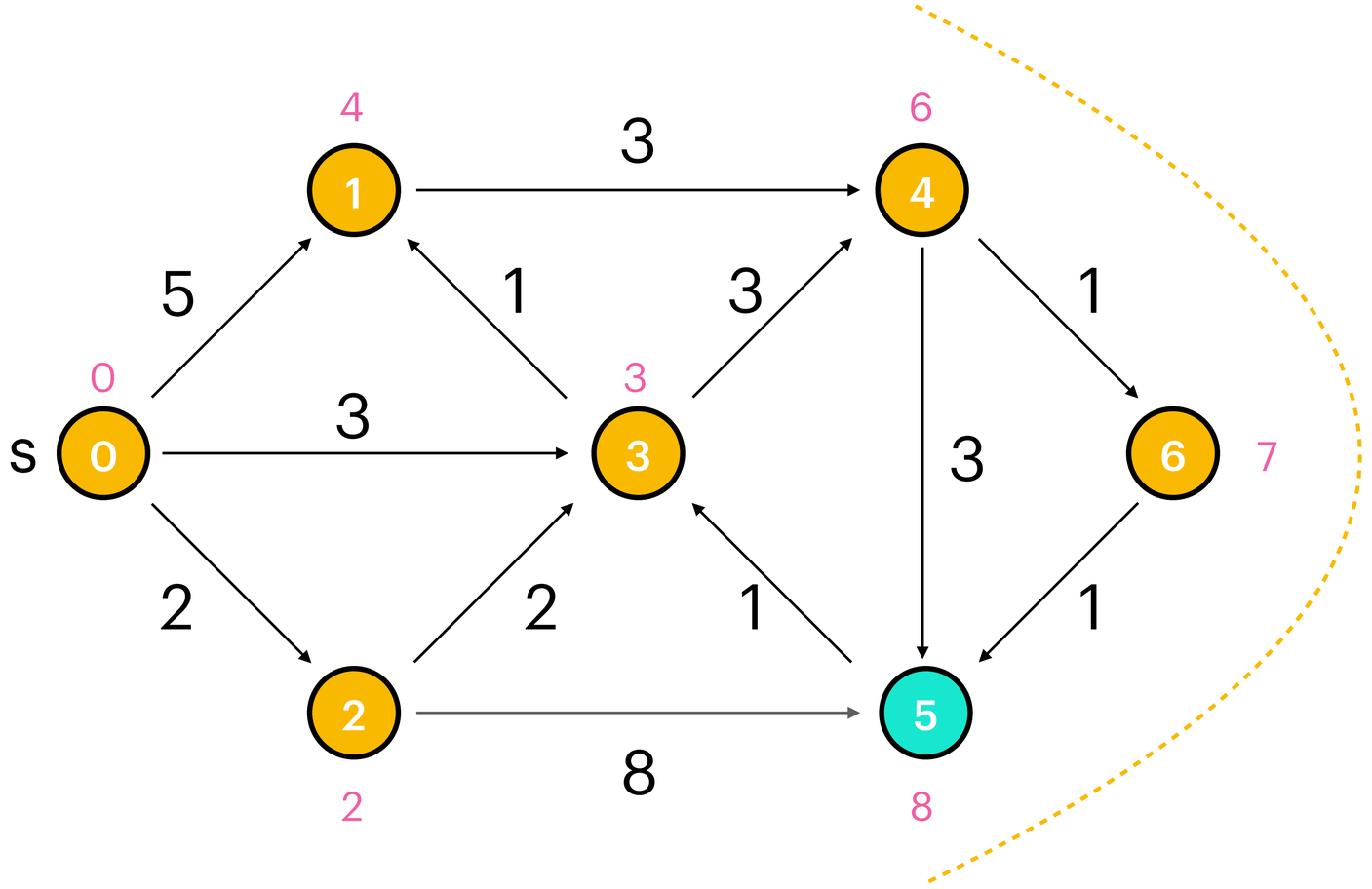
S : { 0 , 2 , 3 , 1 , 4 , 6 , 5 }

v* = 5

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	8	7

"Heap" :



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])
```

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) :

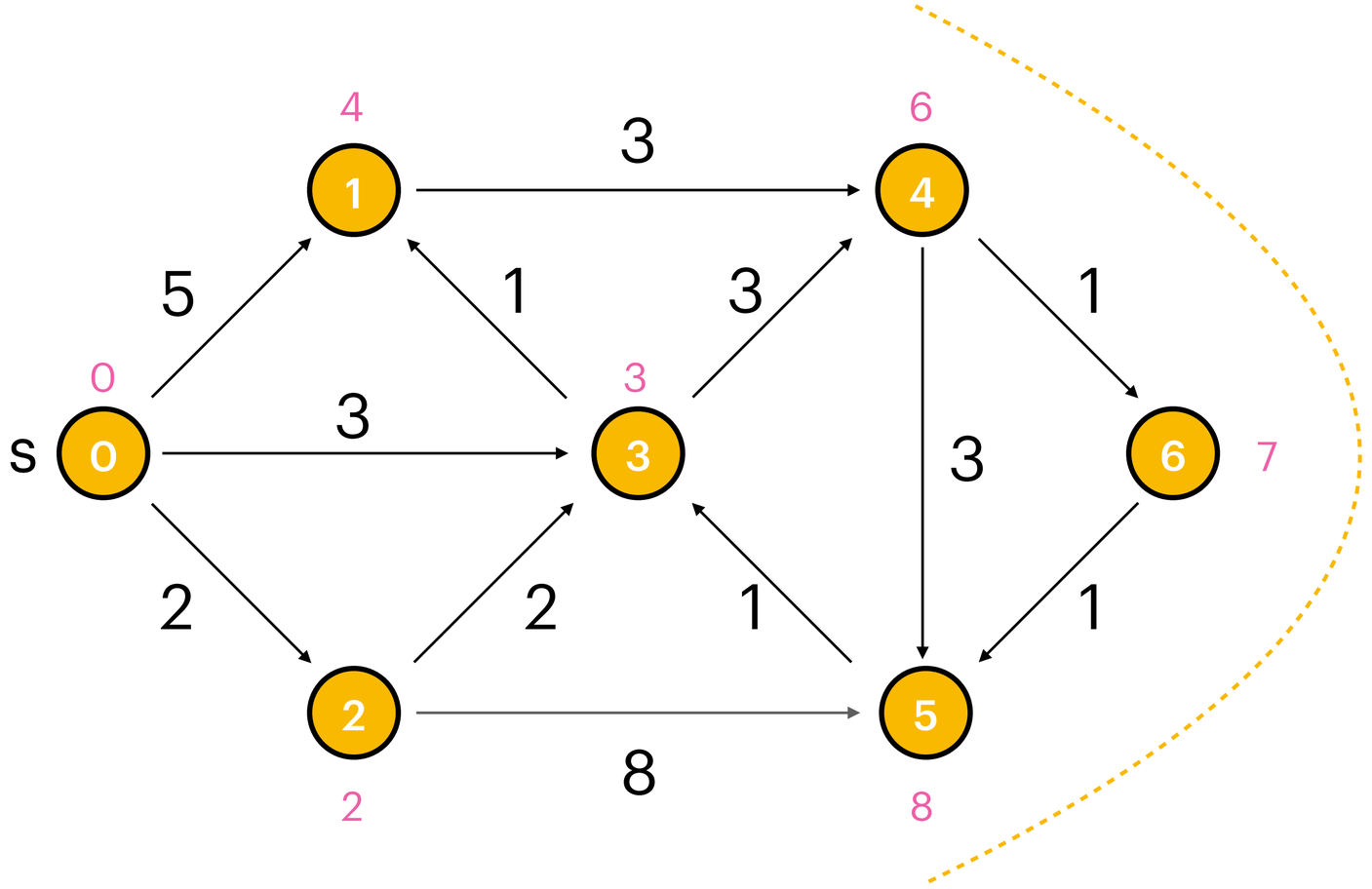
Update the distance of v in heap H to the key k

S : { 0 , 2 , 3 , 1 , 4 , 6 , 5 }

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	8	7

"Heap" :



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])
```

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

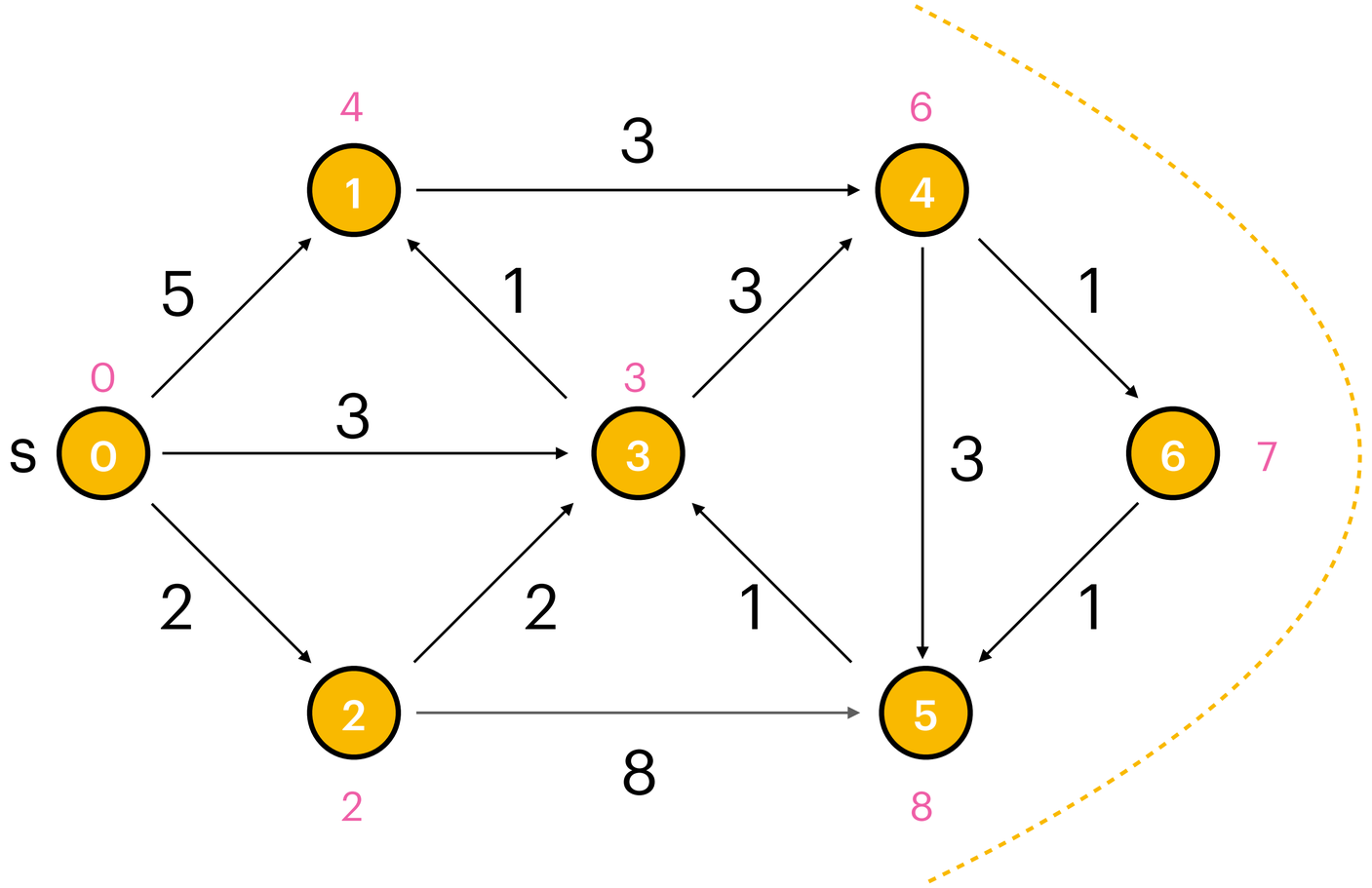
S : { 0 , 2 , 3 , 1 , 4 , 6 , 5 }

S = V

d[] :

0	1	2	3	4	5	6
0	4	2	3	6	8	7

"Heap" :

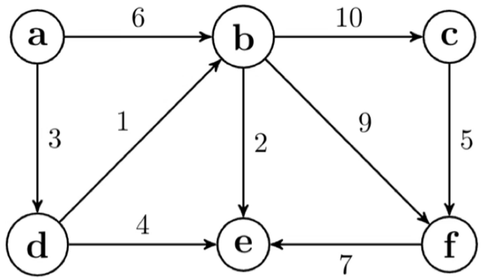


Dijkstra

Exam Questions

/ 2 P

f) *Shortest Path Tree:* Consider the following graph:



i) Highlight the edges that are part of the shortest-path tree rooted at vertex a (i.e., the output of Dijkstra's algorithm if we were to start from vertex a).

ii) Does the above graph have a topological ordering? If yes, write down one topological ordering. If no, give an argument.

Dijkstra

Exam Tipps

- Don’t rush to the solution
  - Update the distances just like dijkstra’s algo

- Don’t mix up with MST ! It’s not the same !

Let’s take a break

Shortest Paths

Bellman Ford

weighted, positive and (possibly) negative edge weights ,
(possibly) negative closed walks
$c(e) \in \mathbb{R}$

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

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

relax the edges for n-1 times

Shortest Paths

Bellman Ford

weighted, positive and (possibly) negative edge weights ,
(possibly) negative closed walks
$c(e) \in \mathbb{R}$

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

Algorithm 7 Bellman-Ford(s)

1: $d[s] \leftarrow 0; \quad d[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

relax the edges for n-1 times

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

Shortest Paths

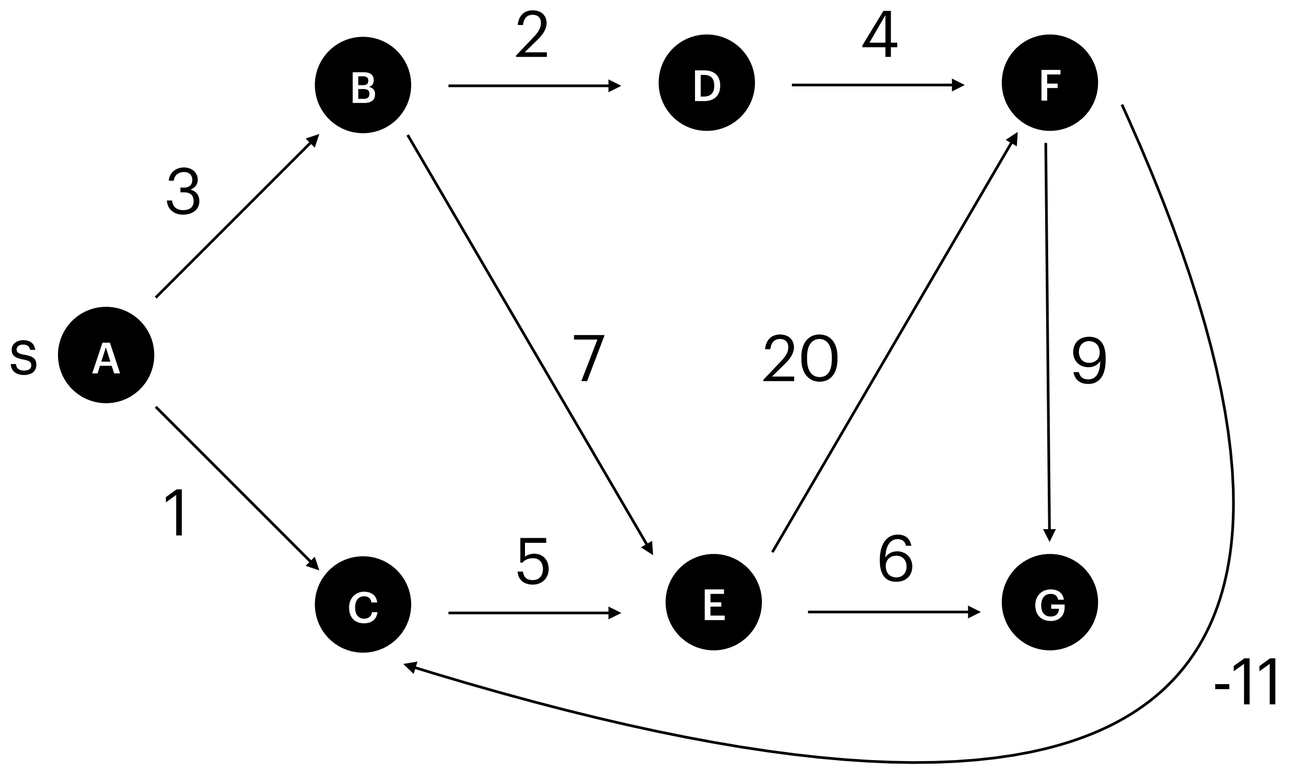
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0							
1							
2							
3							
4							
5							
6							



Shortest Paths

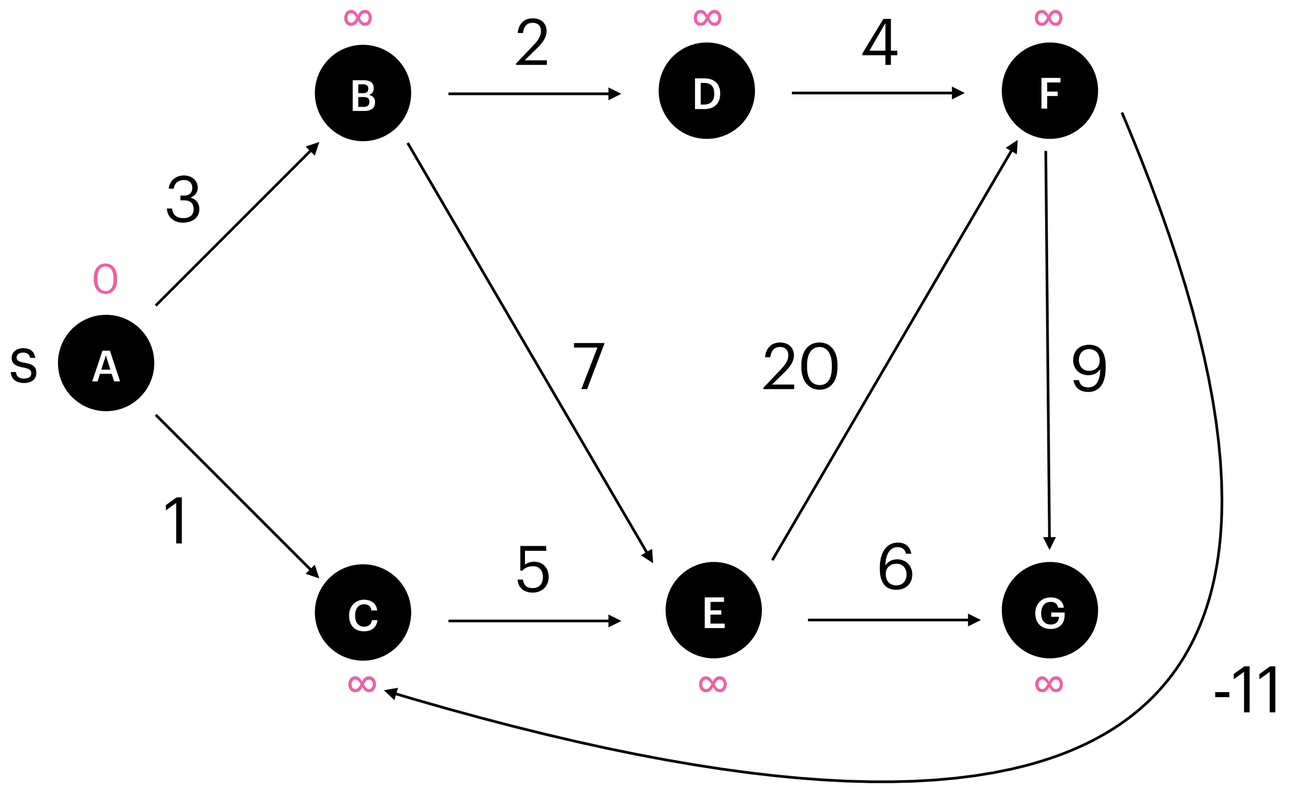
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1							
2							
3							
4							
5							
6							



Shortest Paths

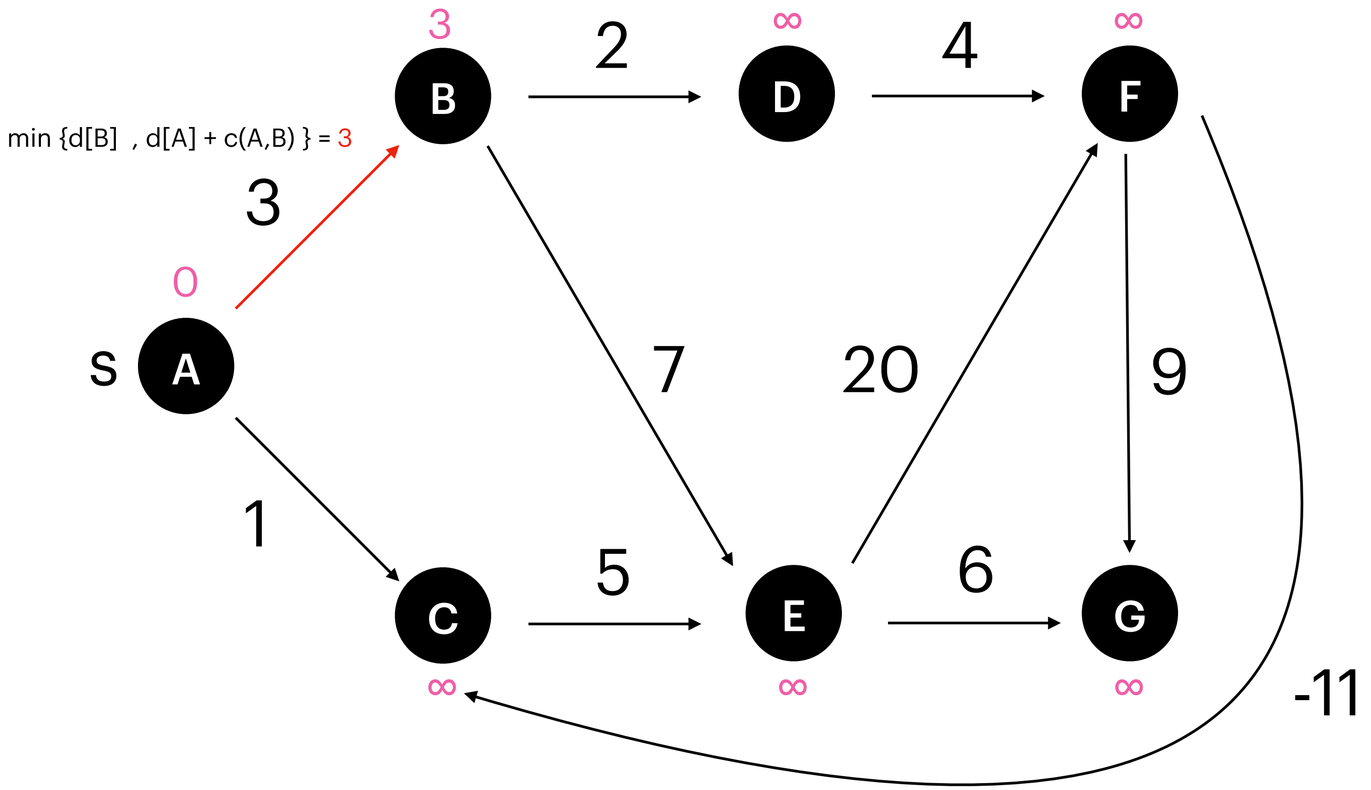
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3					
2							
3							
4							
5							
6							



Shortest Paths

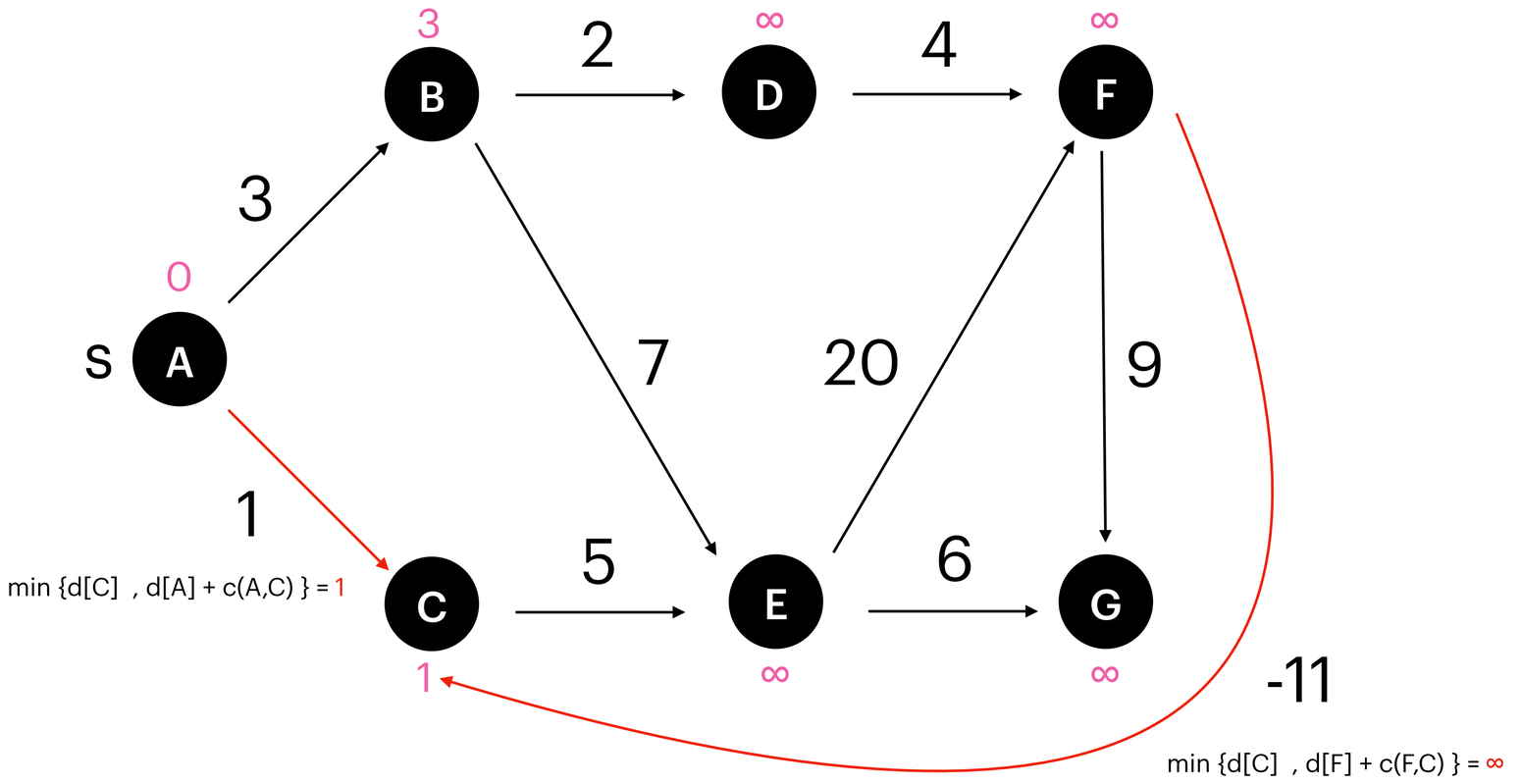
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1				
2							
3							
4							
5							
6							



Shortest Paths

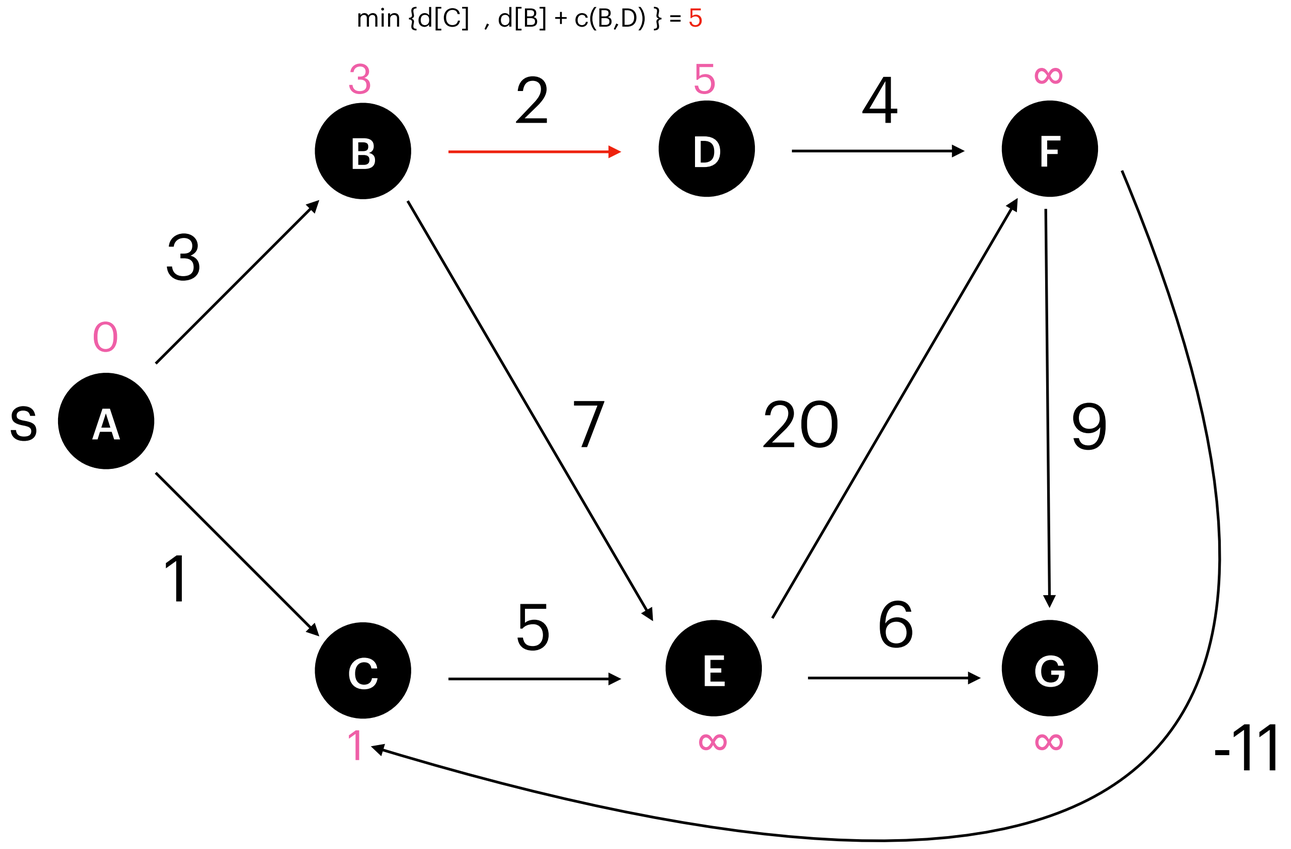
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5			
2							
3							
4							
5							
6							



Shortest Paths

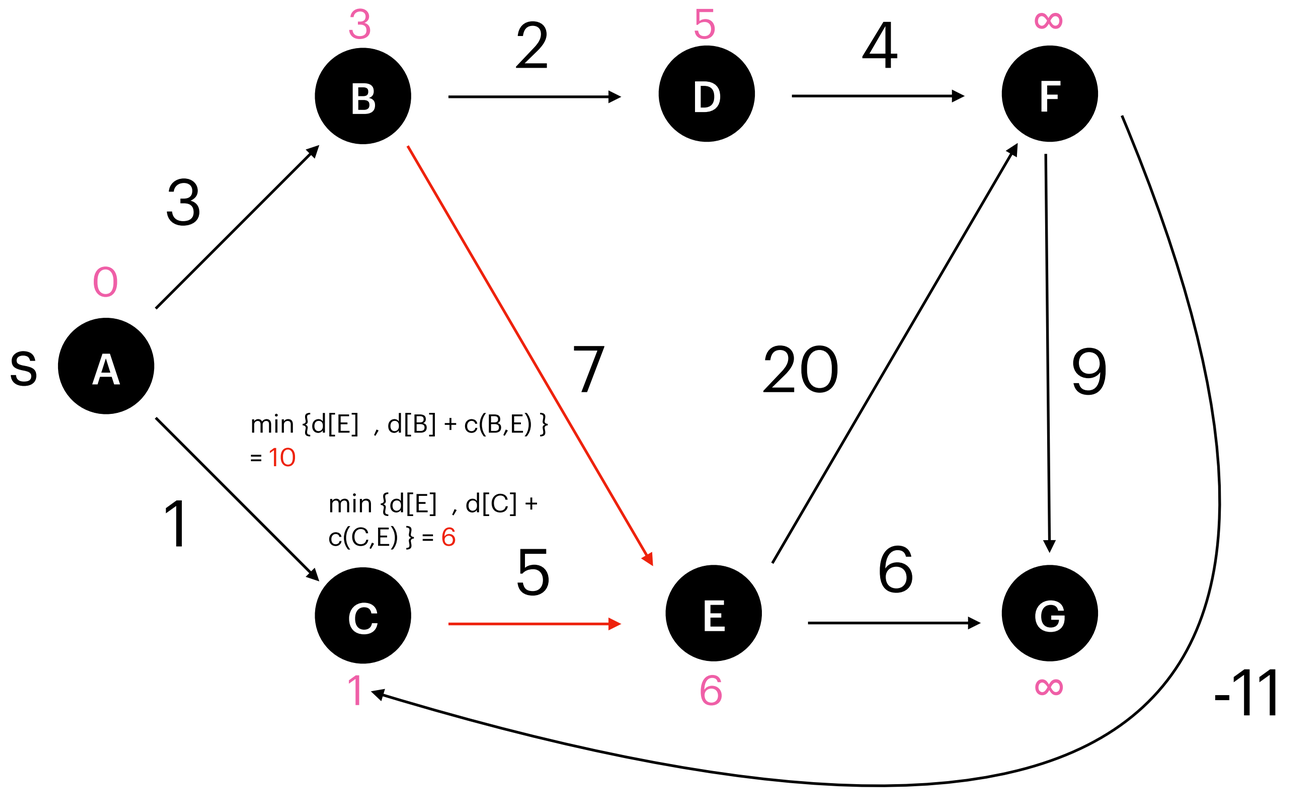
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5	6		
2							
3							
4							
5							
6							



Shortest Paths

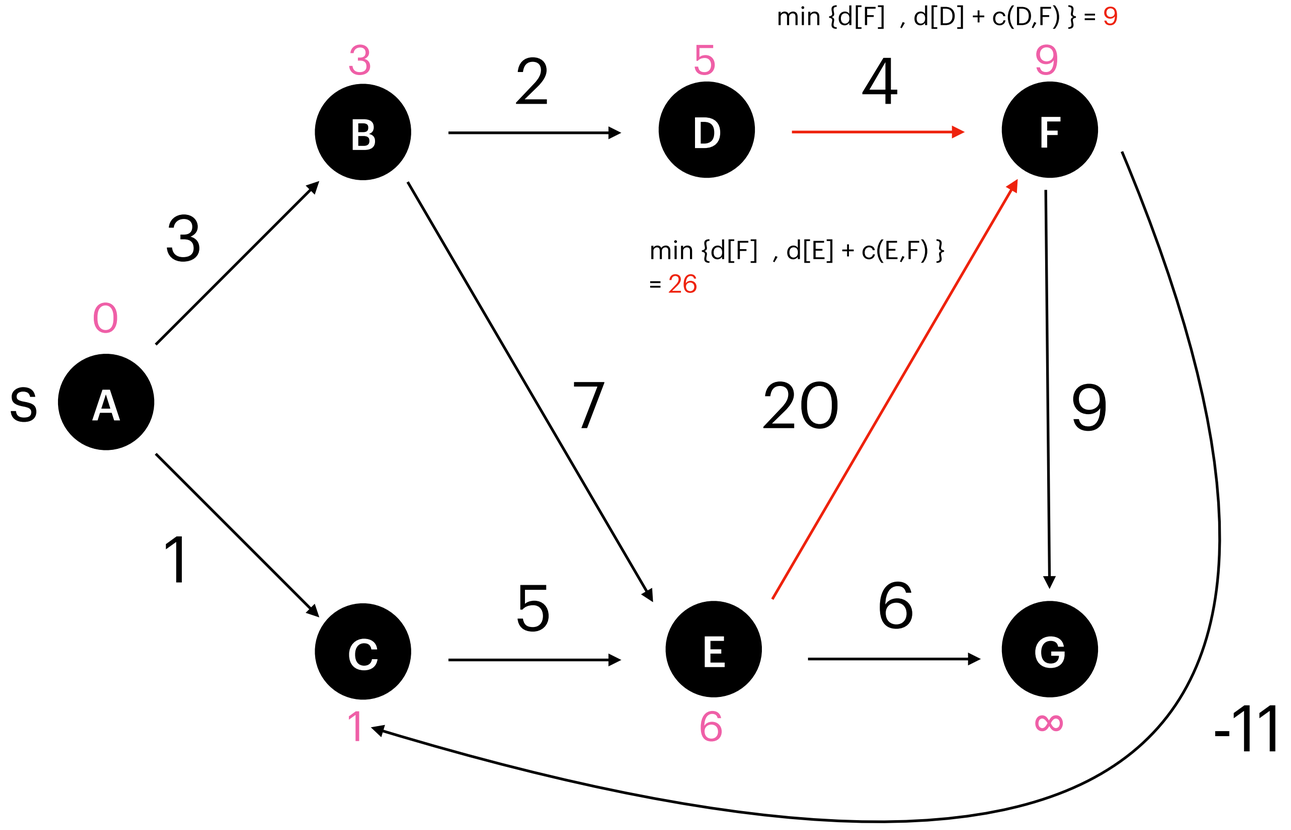
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5	6	9	
2							
3							
4							
5							
6							



Shortest Paths

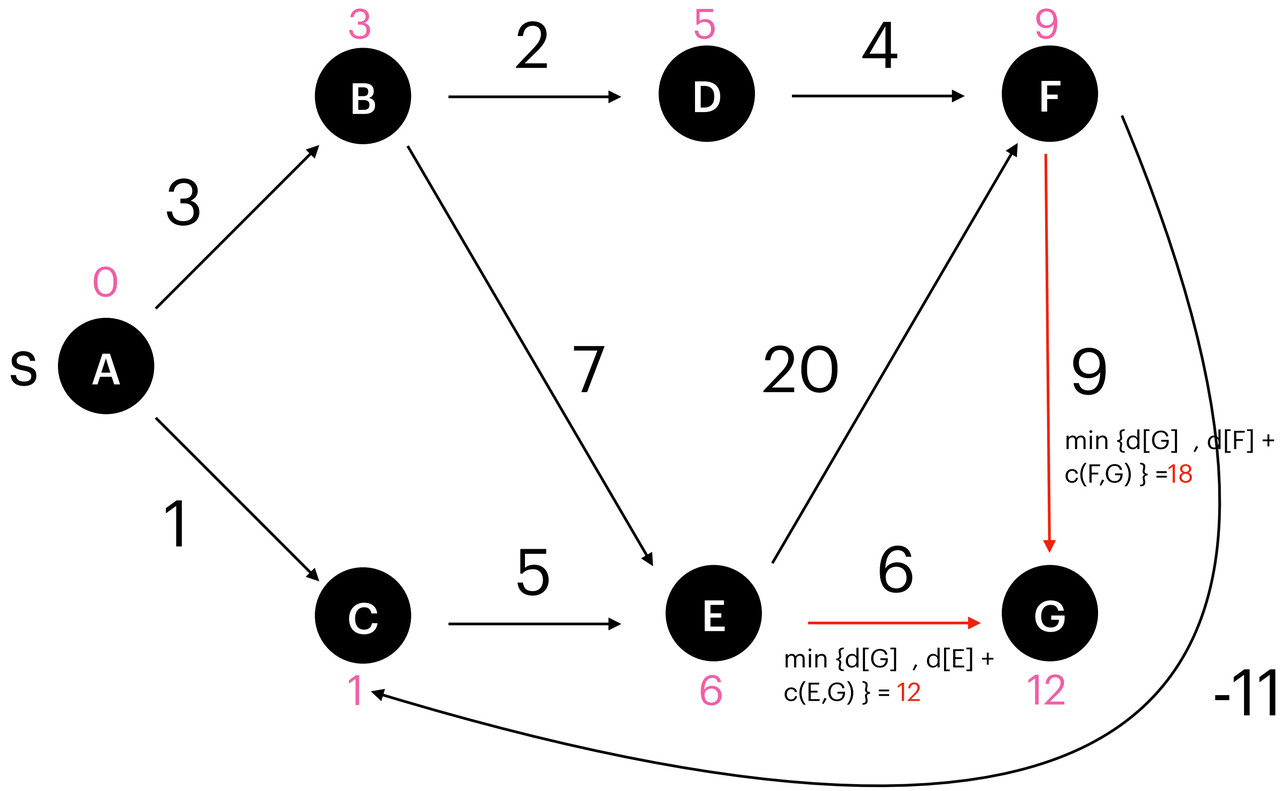
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

d[] :

i	A	B	C	D	E	F	G
0	0	∞	∞	∞	∞	∞	∞
1	0	3	1	5	6	9	12
2							
3							
4							
5							
6							



Shortest Paths

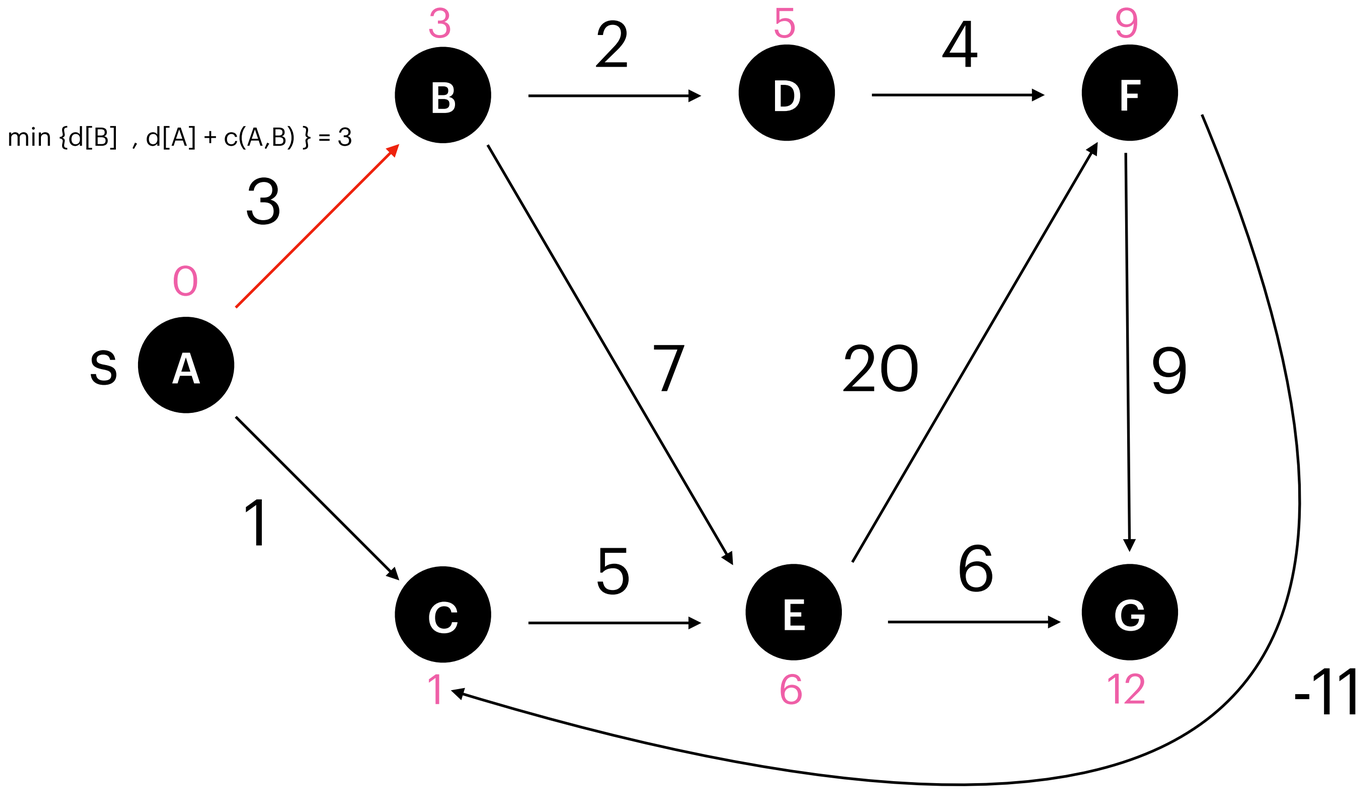
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

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							



Shortest Paths

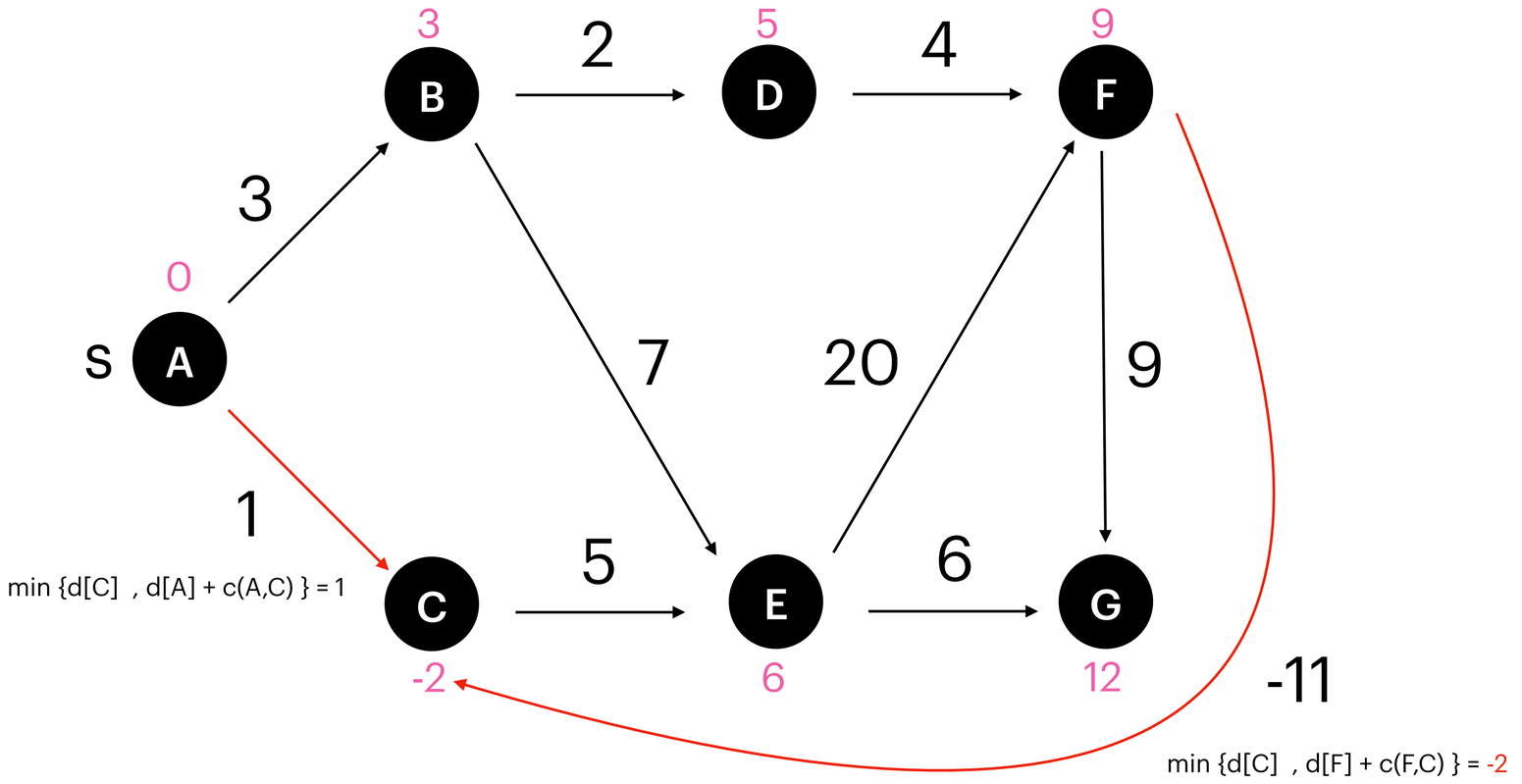
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

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							



Shortest Paths

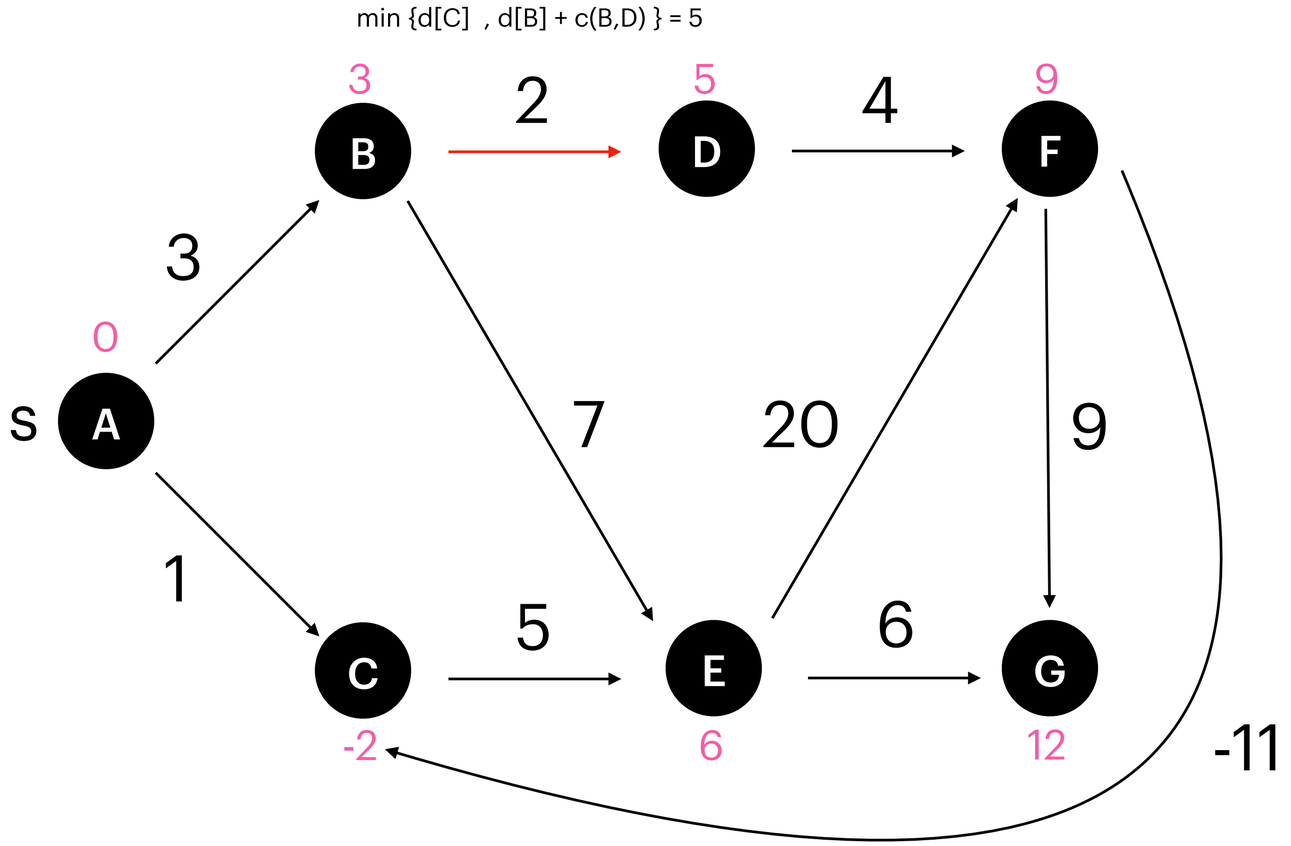
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

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							



Shortest Paths

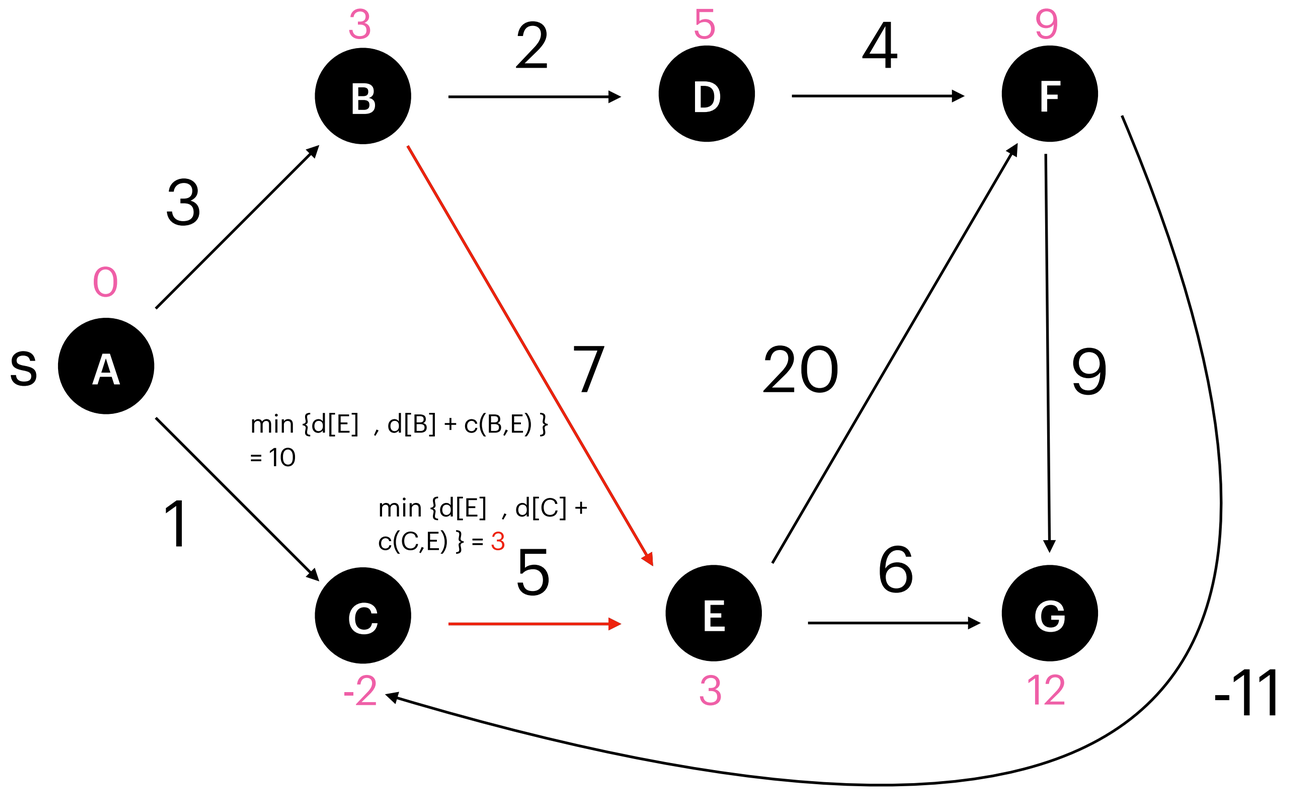
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

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							



Shortest Paths

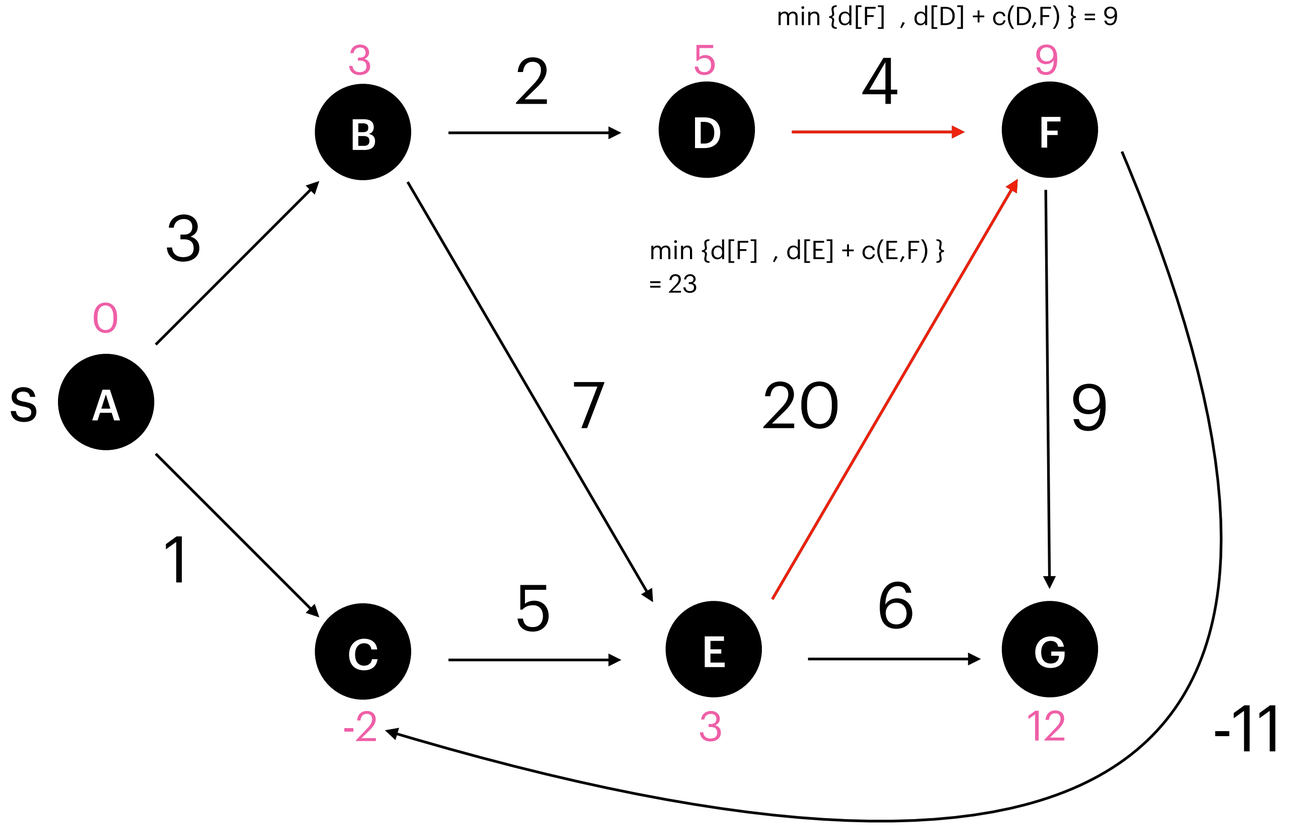
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

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							



Shortest Paths

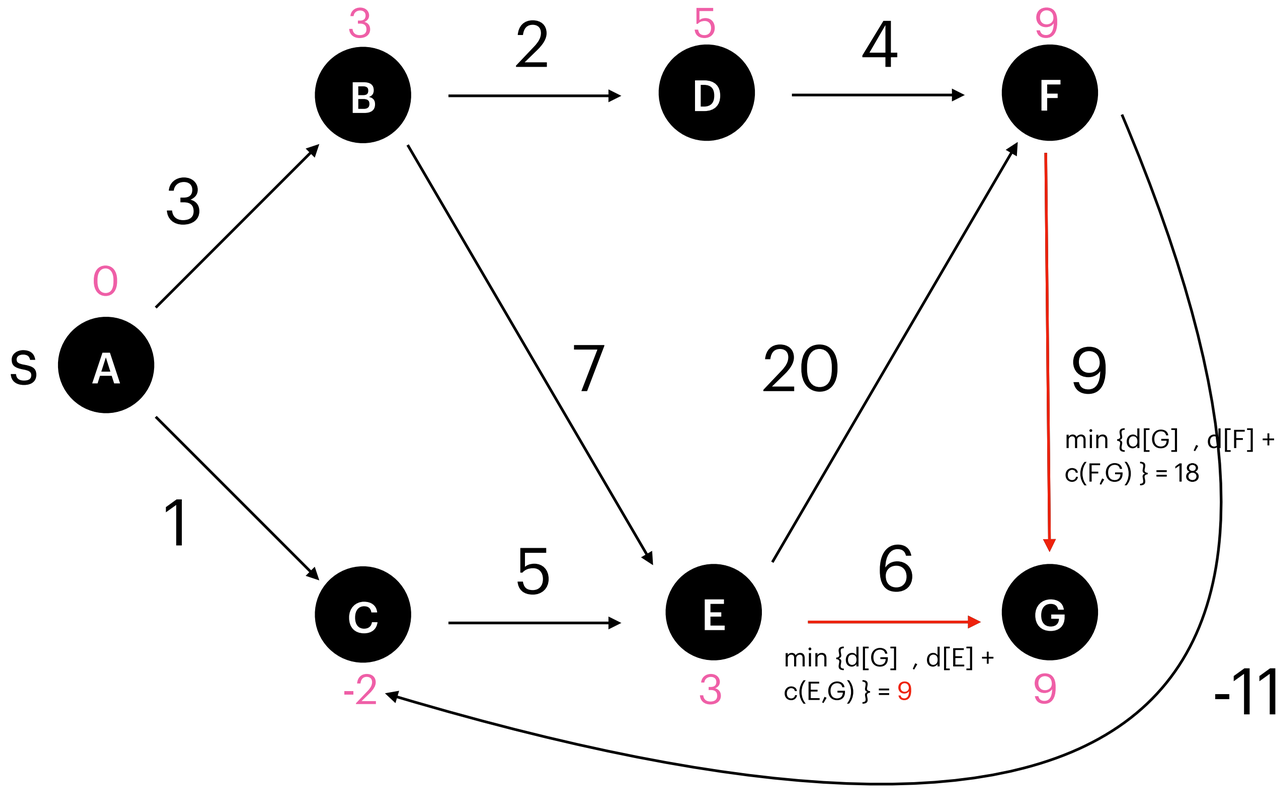
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

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							



Shortest Paths

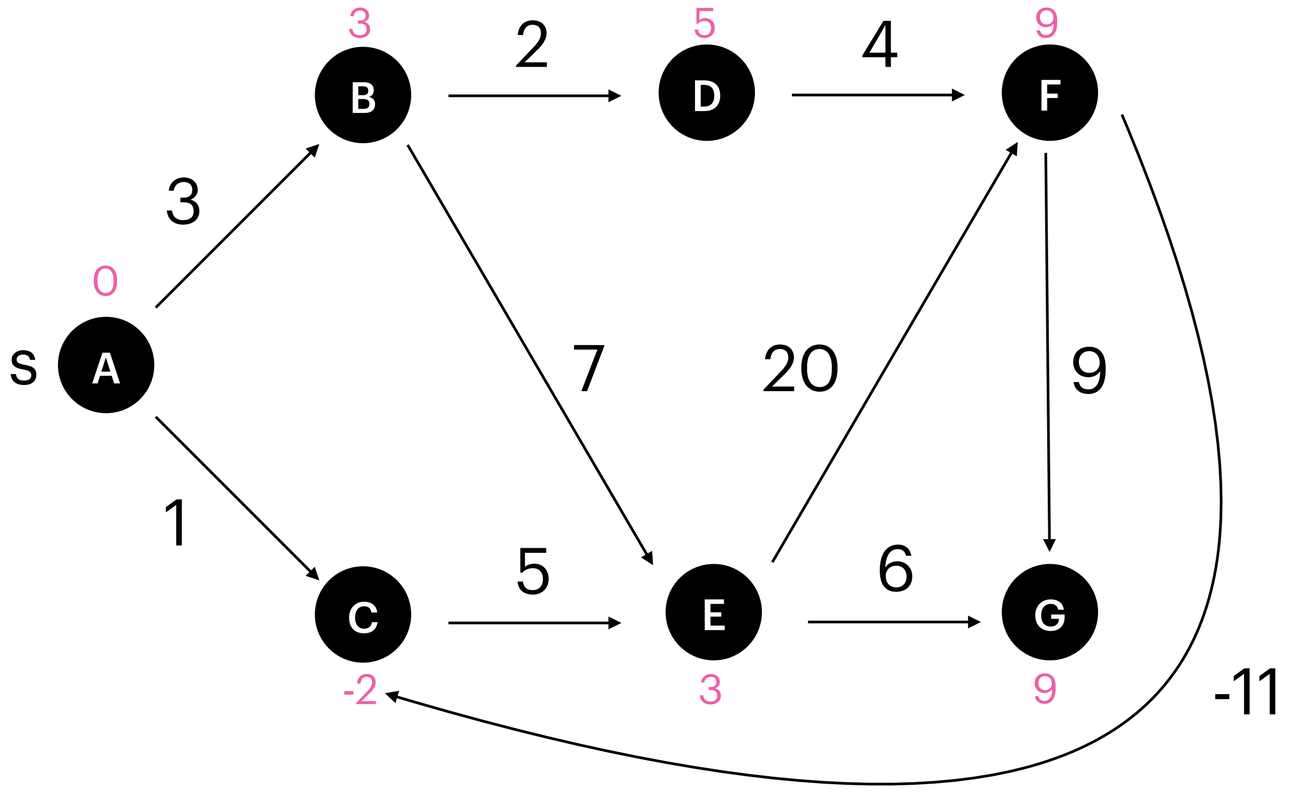
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

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							



Shortest Paths

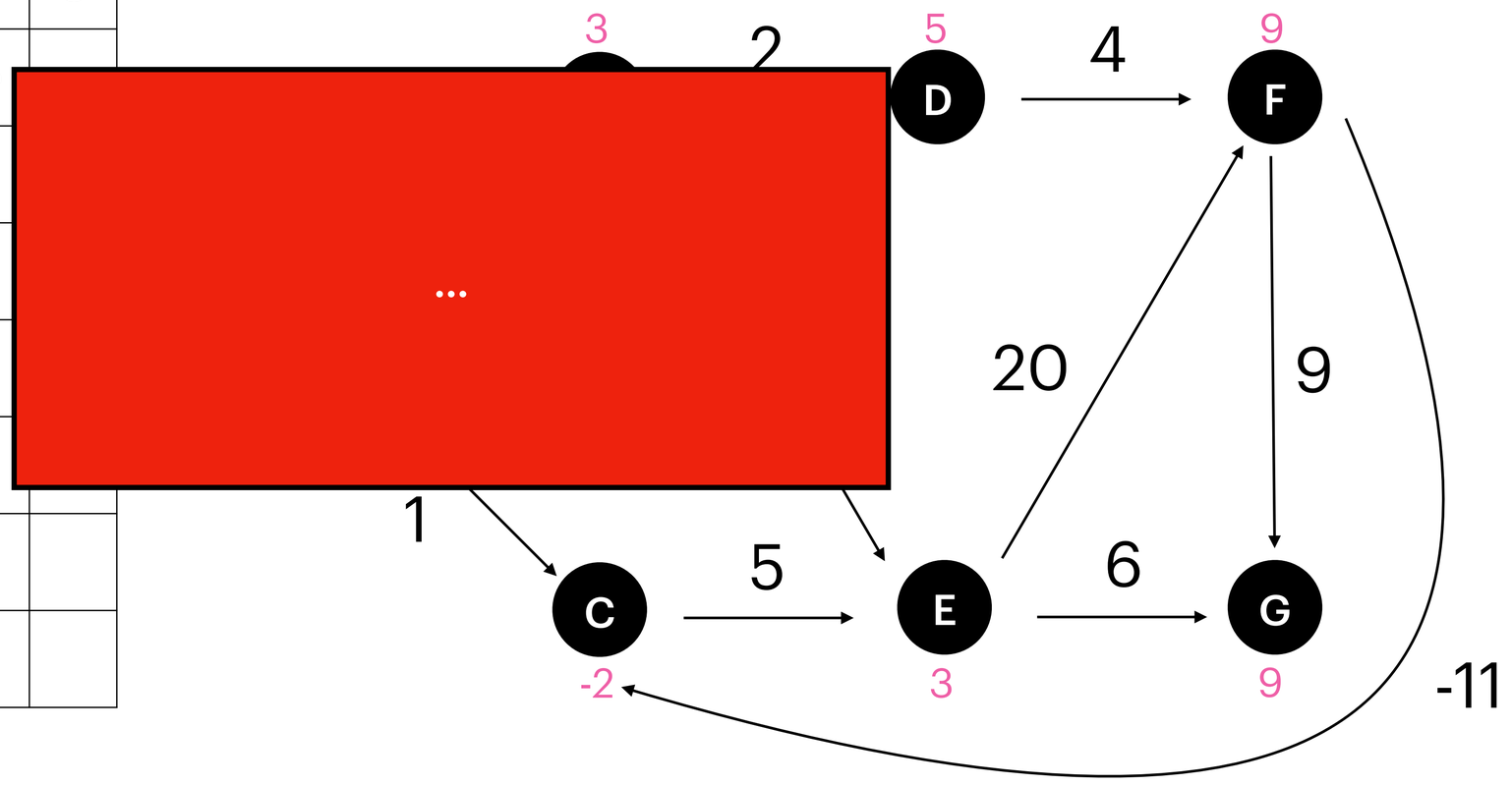
Bellman Ford

Algorithm 7 Bellman-Ford(s)

1: $\mathrm{d}[s] \leftarrow 0; \quad \mathrm{d}[v] \leftarrow \infty \; \forall v \in V \setminus \{s\}$
2: **for** $i \in \{1, \ldots n-1\}$ **do**
3: **for** $(u, v) \in E$ **do**
4: $\mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[u] + c(u, v)\}$

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							



Shortest Paths

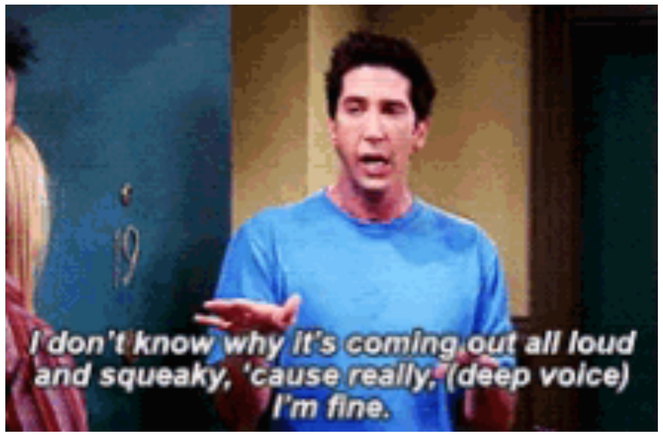
one - to - all

G (directed/undirected)	Algorithm	Runtime
unweighted , all nodes with the same positive weight	BFS usage	O (|V| + |E|)
weighted , positive 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|)

Graph Modelling

Exam Question

Code Expert - Graph Sets



Next Week ...

MST

Questions

Feedbacks , Recommendations

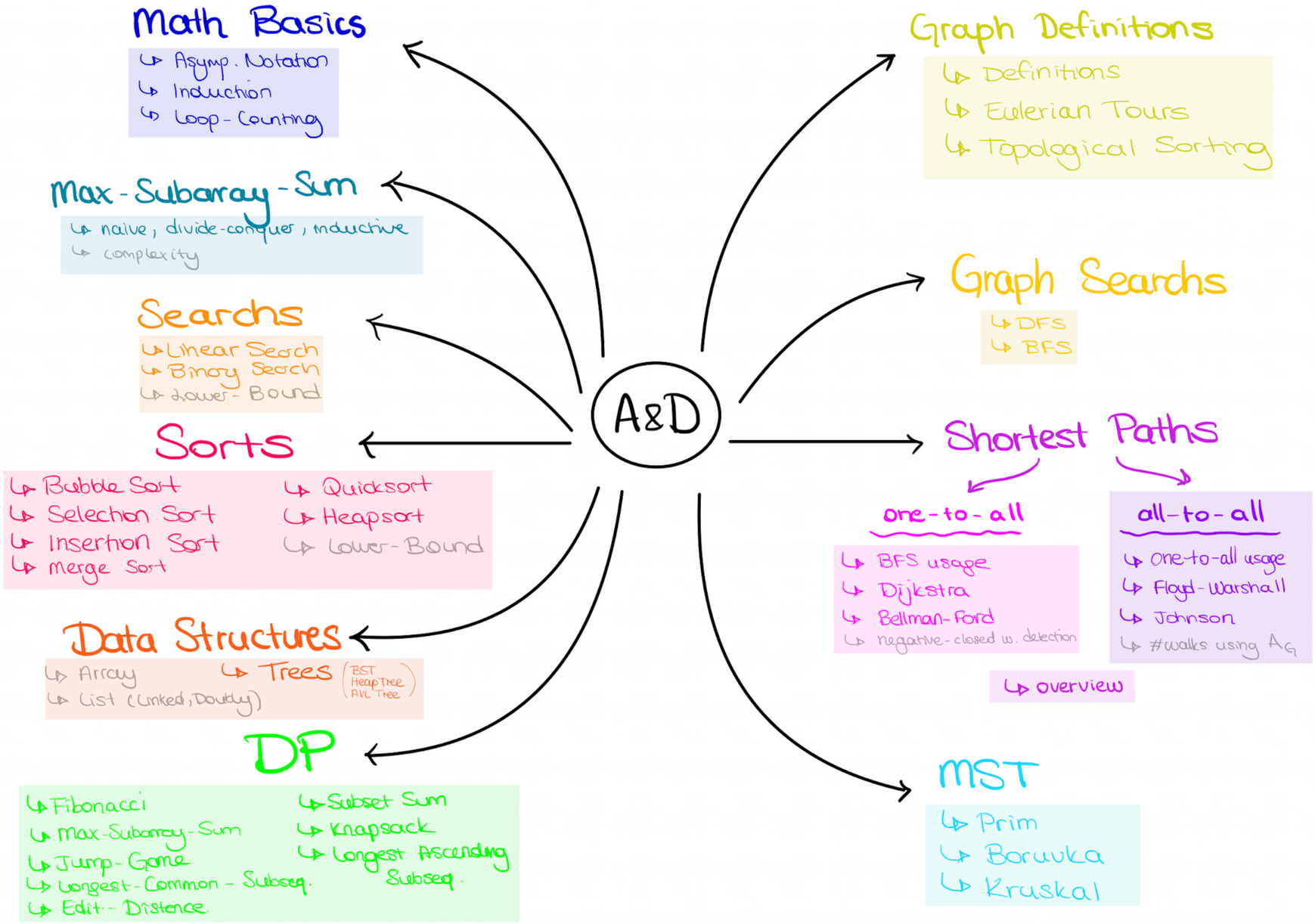
Nil Ozer

A&D

Exercise Session 12

Nil Ozer

A&D Overview


A&D
Math Basics
↳ 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

Outline

- Quiz
- Exercise Sheets

- MST

- Last week organization

Quiz

Exercise Sheet 7

Bonus Feedback

- 7.1 : 👏👏👏
- 7.4 : 👏👏👏
- 7.5 : 👏👏👏

- Recursion Justification :(

Exercise Sheet 9

Bonus Feedback

- 9.2 : short questions about graphs 👏👏👏
- 9.4 : 👏
  - Recursion Justification :(
  - Watch out for the calculation order :
    - path starting at i : n to 1
    - path ending at i : 1 to n
- 9.5 : 👏👏👏
  - Pre-order , revision from the dfs slides

Peergrading and rest

- Exercise Sheet 11 peergrading
  - 11.1 this week
  - Emails will be sent

- If urgent feedback is needed, send me an email !

MST

MST

Definition

Tree

- no cycles
- A tree with k vertices has k-1 edges

MST

Definition

Spanning Tree

- Spans all the vertices in the graph
- Every vertex is included in the tree.

- no cycles
- |V| - 1 edges

MST

Definition

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

Recap

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

MST

Prim's Algorithm - Dijkstra approach

Runtime : O ((|V| + |E|) * log n)

Algorithm 6 Dijkstra(s)

1: $\mathrm{d}[s] \leftarrow 0$; $\mathrm{d}[v] \leftarrow \infty \ \ \forall v \in V \setminus \{s\}$
2: $S \leftarrow \varnothing$
3: $H \leftarrow$ make-heap(V); decrease-key($H, s, 0$)
4: **while** $S \neq V$ **do**
5: $\quad v^* \leftarrow$ extract-min(H)
6: $\quad S \leftarrow S \cup \{v^*\}$
7: $\quad$ **for** $(v^*, v) \in E,\ v \notin S$ **do**
8: $\quad\quad \mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], \mathrm{d}[v^*] + c(v^*, v)\}$
9: $\quad\quad$ decrease-key($H, v, \mathrm{d}[v]$)

Algorithm 10 Prim(G, s) (mit min-heap)

1: $H \leftarrow$ make-heap(V, ∞), $S \leftarrow \varnothing$
2: $\mathrm{d}[s] \leftarrow 0$; $\mathrm{d}[v] \leftarrow \infty \ \ \forall v \in V \setminus \{s\}$
3: decrease-key($H, s, 0$)
4: **while** $H \neq \varnothing$ **do**
5: $\quad v^* \leftarrow$ extract-min(H)
6: $\quad S \leftarrow S \cup \{v^*\}$
7: $\quad$ **for** $v^*v \in E,\ \ v \notin S$ **do**
8: $\quad\quad \mathrm{d}[v] \leftarrow \min\{\mathrm{d}[v], c(v^*, v)\}$
9: $\quad\quad$ decrease-key($H, v, \mathrm{d}[v]$)

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: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

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: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : edges of the MST

S : connected component set

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: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

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

MST

Prim's Algorithm

F : edges of the MST

S : connected component set

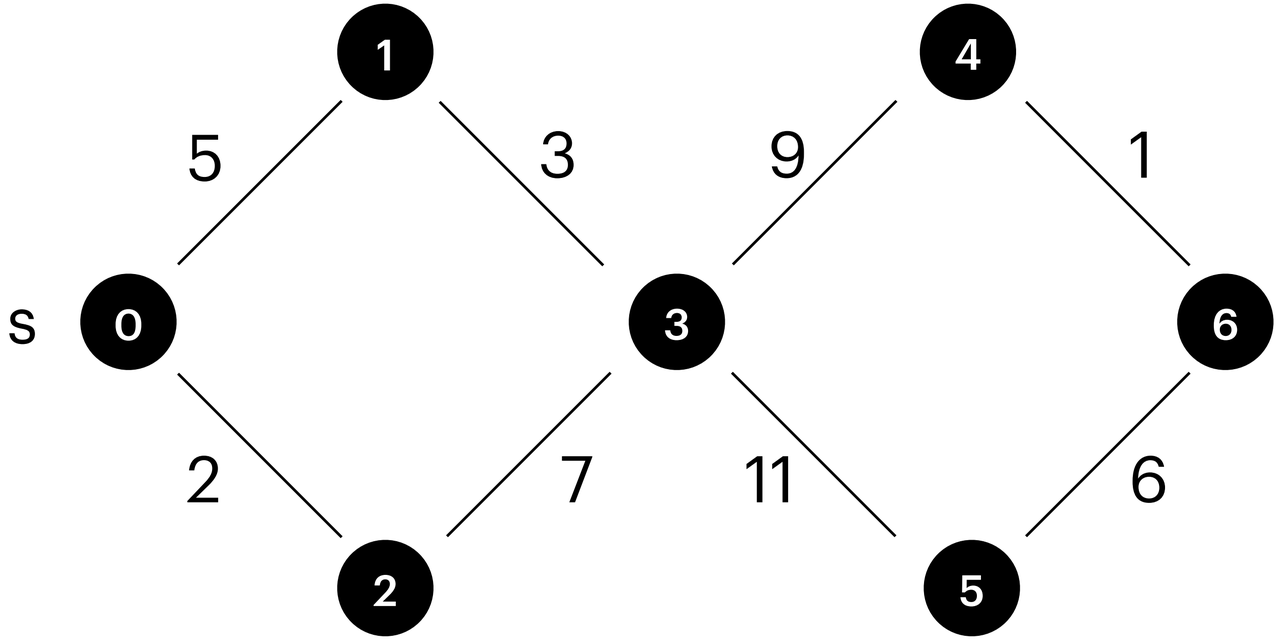
4 : find the minimum edge {u*,v*} s.t.
u* is in S but v* is not

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F :

S :



MST

Prim's Algorithm

F : edges of the MST

S : connected component set

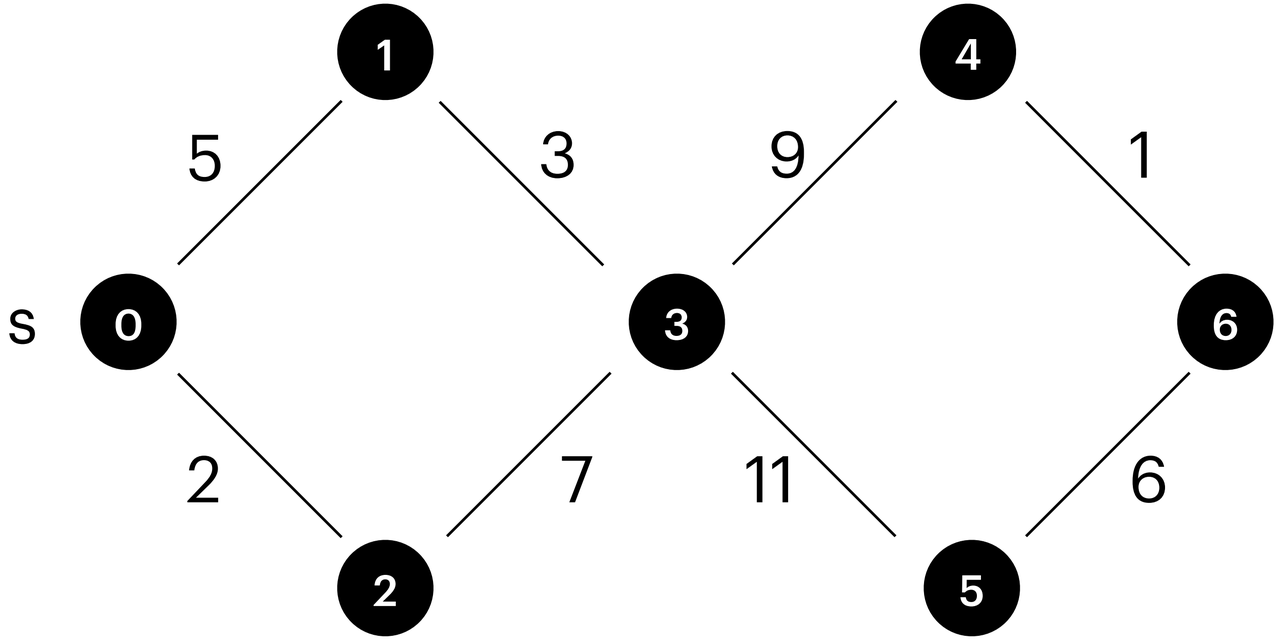
4 : find the minimum edge {u*,v*} s.t.
u* is in S but v* is not

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : ∅

S : { 0 }



MST

Prim's Algorithm

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

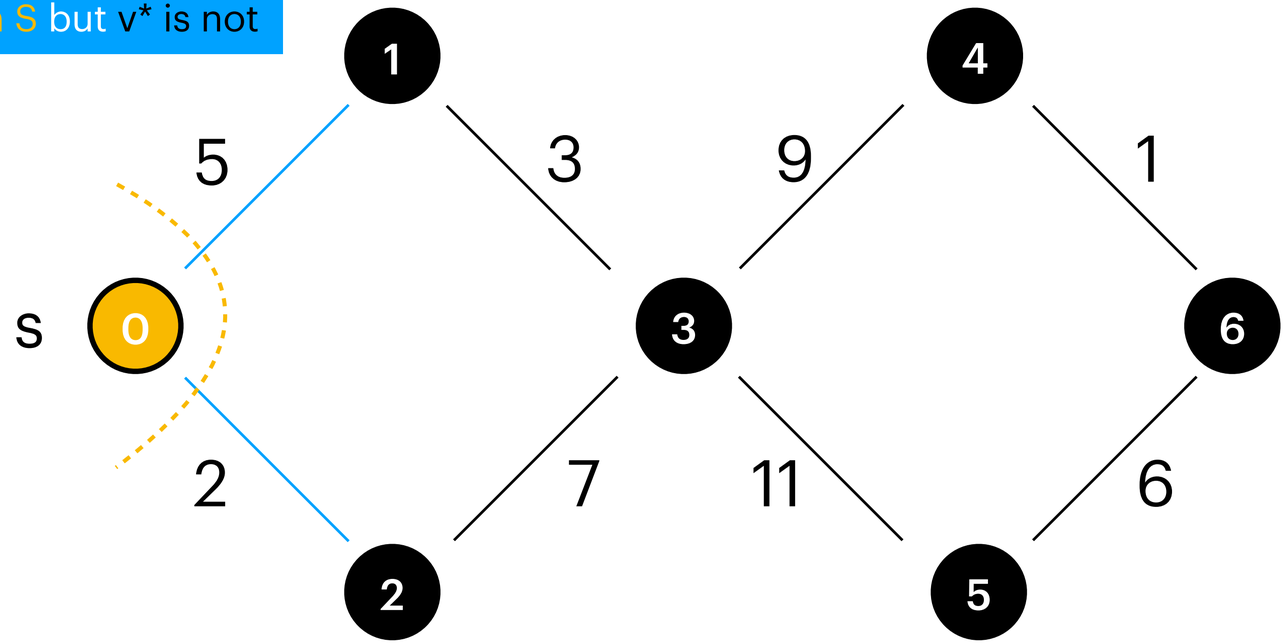
Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : ∅

S : { 0 }

edges {u* v*} s.t.
u* is in S but v* is not



MST

Prim's Algorithm

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

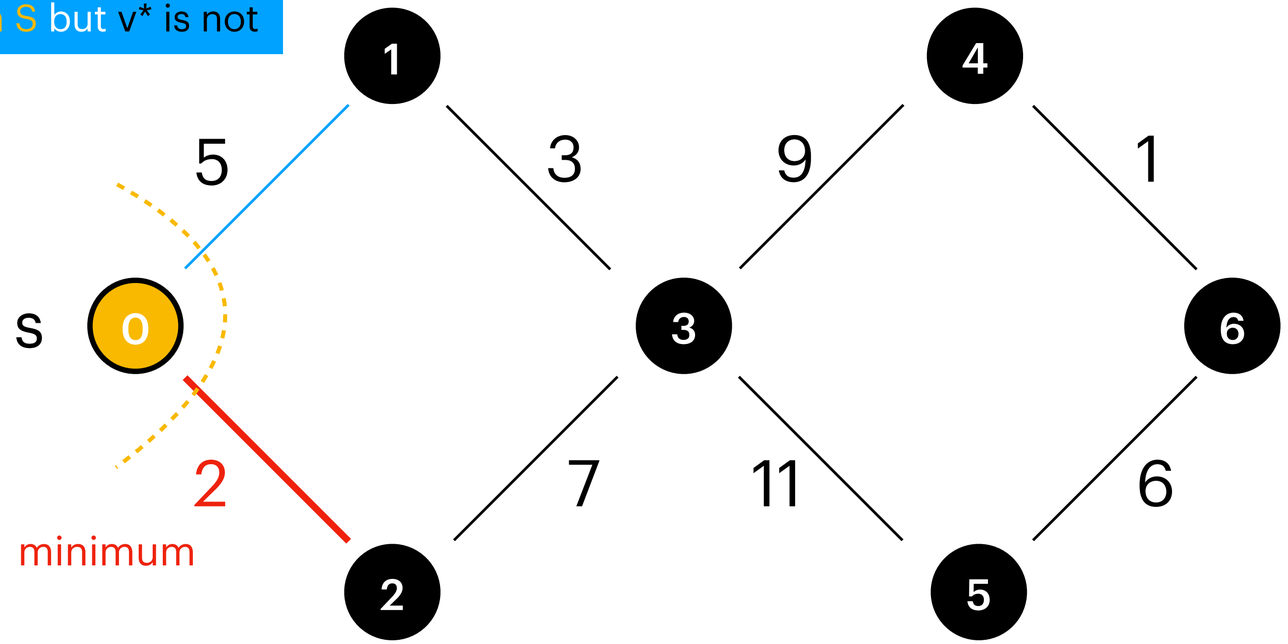
Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : ∅

S : { 0 }

edges {u* v*} s.t.
u* is in S but v* is not



MST

Prim's Algorithm

F : edges of the MST

S : connected component set

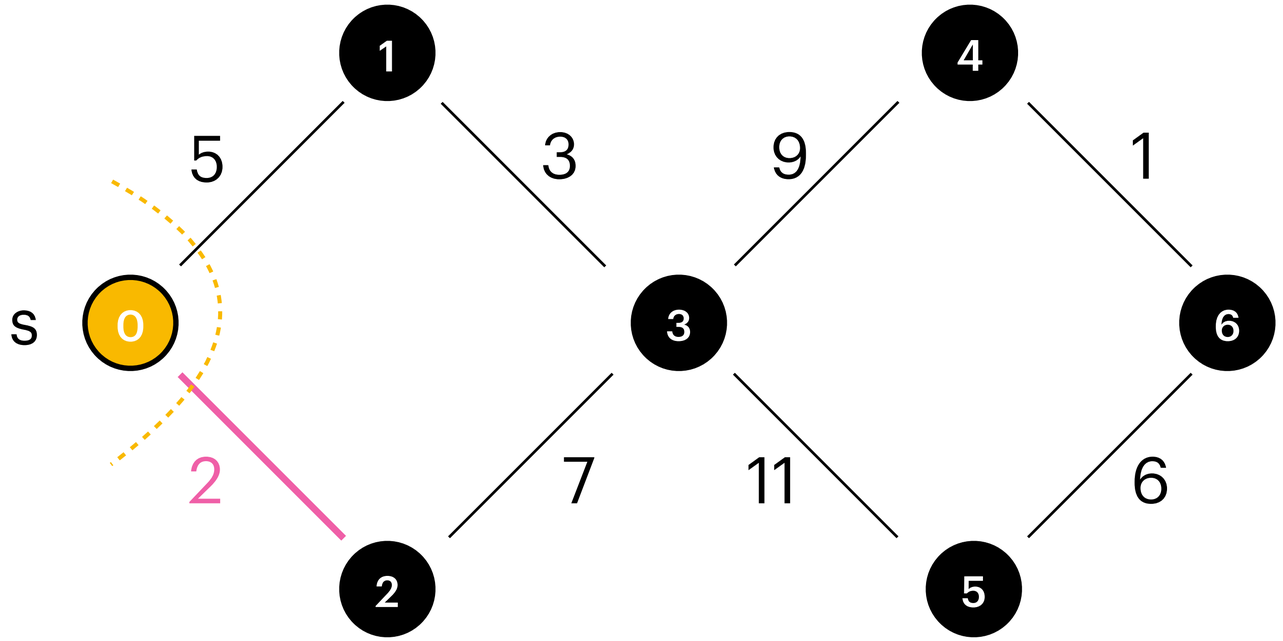
4 : find the minimum edge {u*,v*} s.t.
u* is in S but v* is not

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : { {0,2} }

S : { 0 }



MST

Prim's Algorithm

F : edges of the MST

S : connected component set

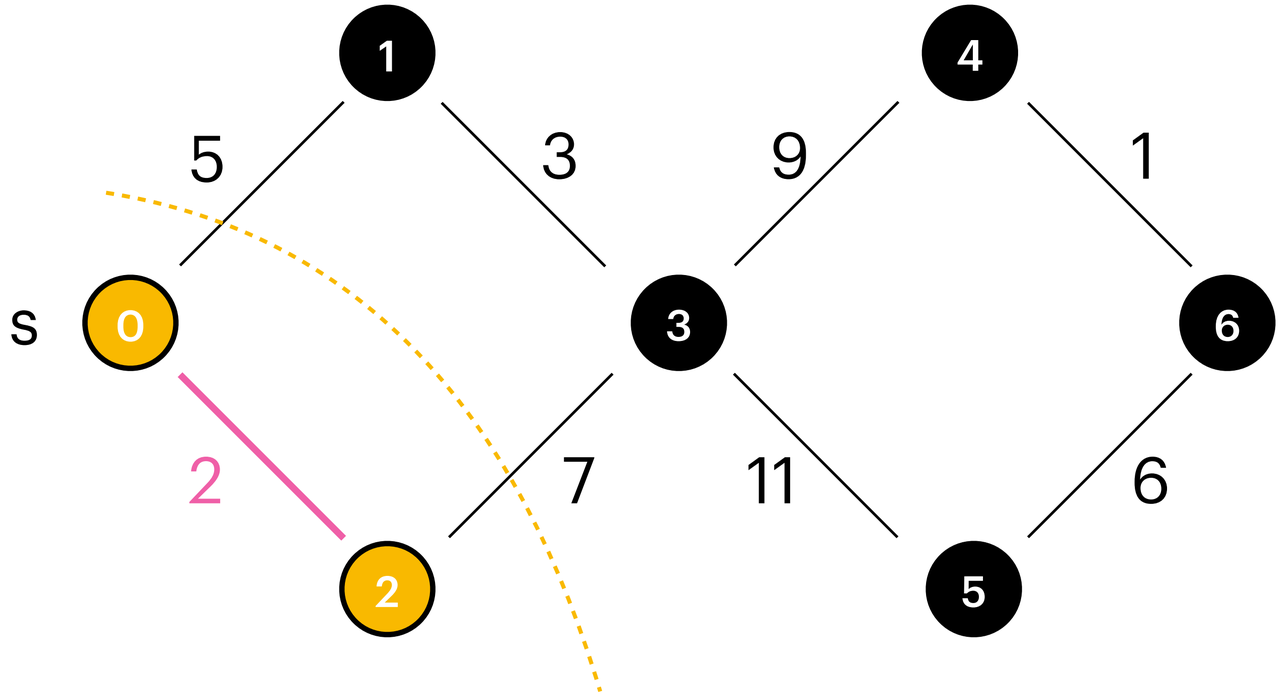
4 : find the minimum edge {u*,v*} s.t.
u* is in S but v* is not

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : { {0,2} }

S : { 0 , 2 }



MST

Prim's Algorithm

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

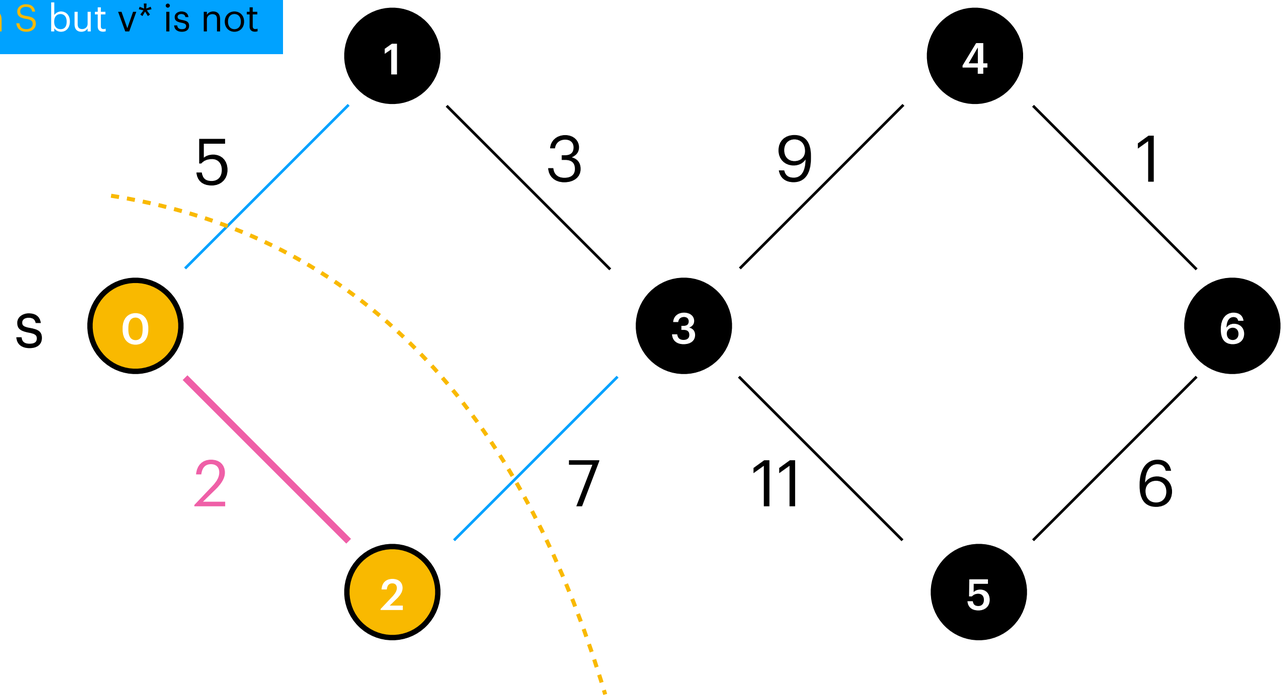
Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : { {0,2} }

S : { 0 , 2 }

edges {u* v*} s.t.
u* is in S but v* is not



MST

Prim's Algorithm

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

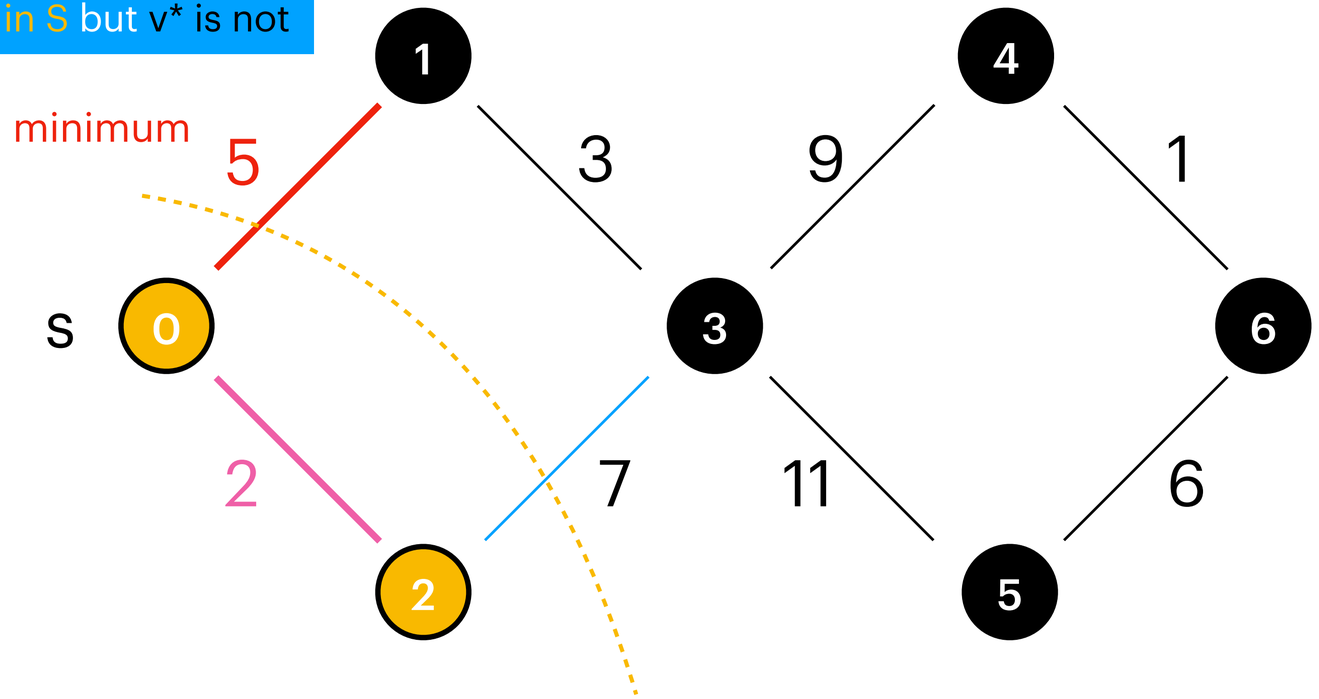
Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : { {0,2} }

S : { 0 , 2 }

edges {u* v*} s.t.
u* is in S but v* is not



MST

Prim's Algorithm

F : edges of the MST

S : connected component set

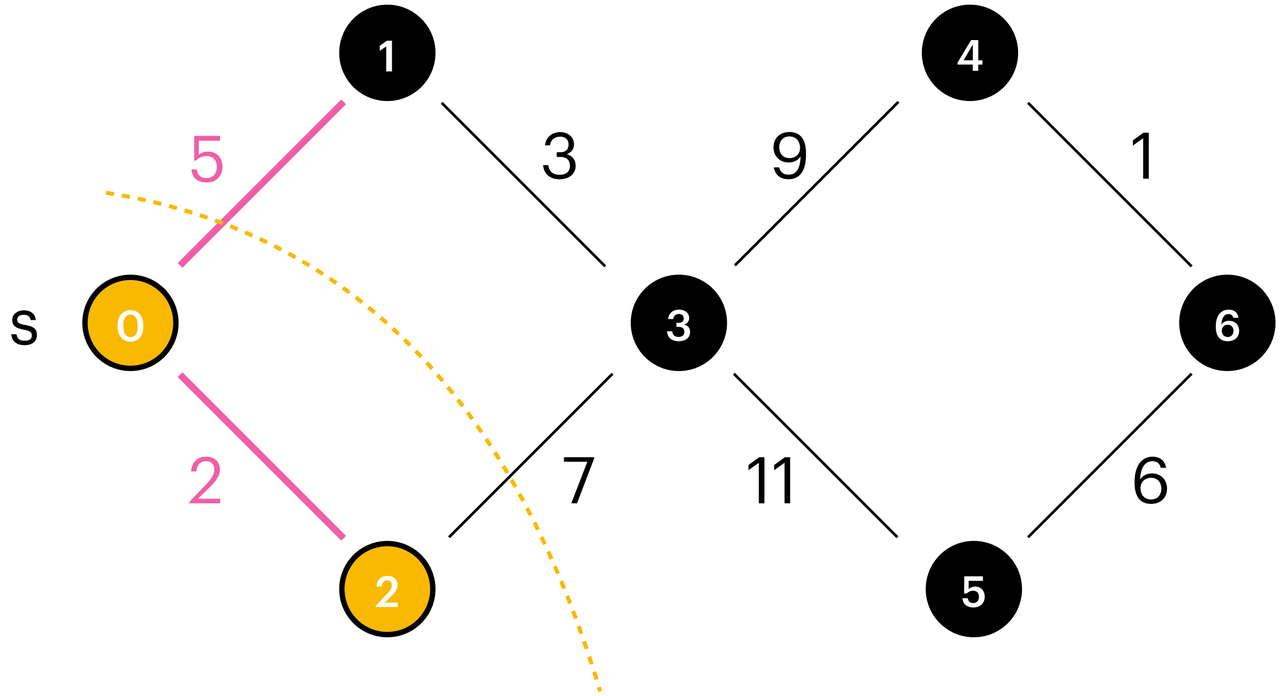
4 : find the minimum edge {u*,v*} s.t.
u* is in S but v* is not

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : { {0,2} , {0,1} }

S : { 0 , 2 }



MST

Prim's Algorithm

F : edges of the MST

S : connected component set

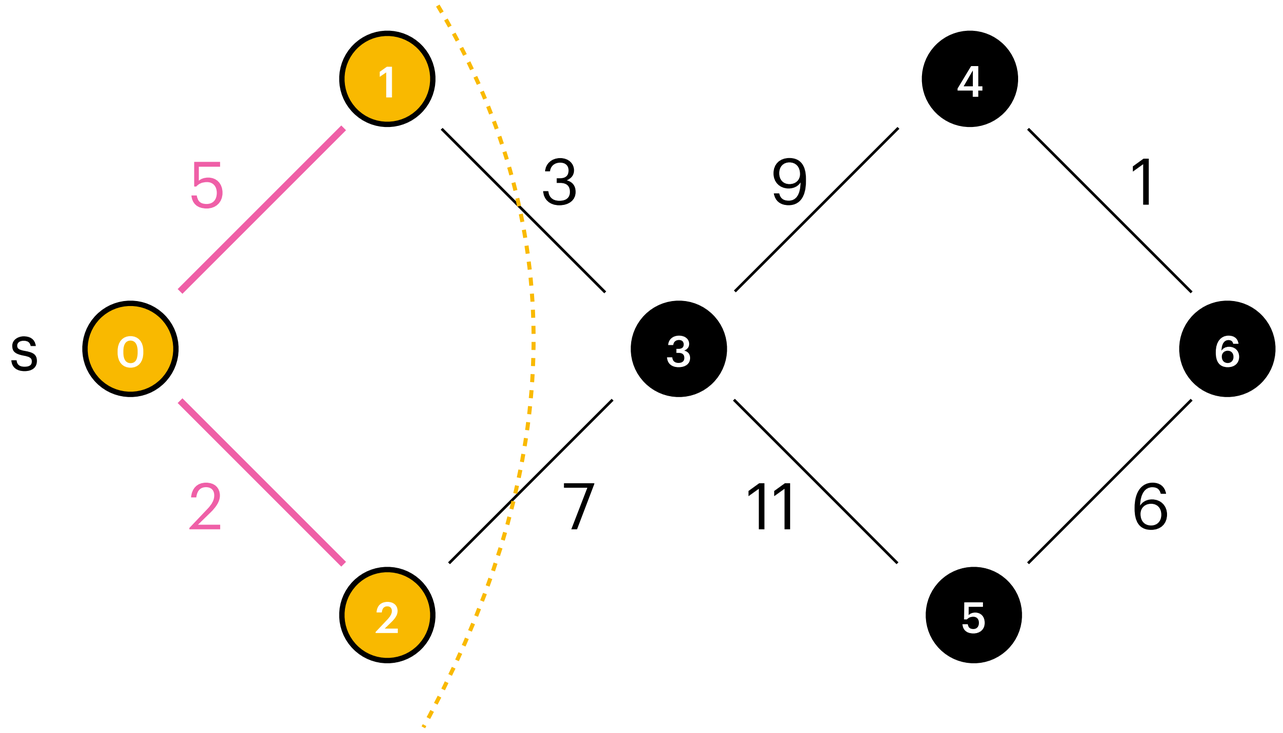
4 : find the minimum edge {u*,v*} s.t.
u* is in S but v* is not

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : { {0,2} , {0,1} }

S : { 0 , 2 , 1 }



MST

Prim's Algorithm

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

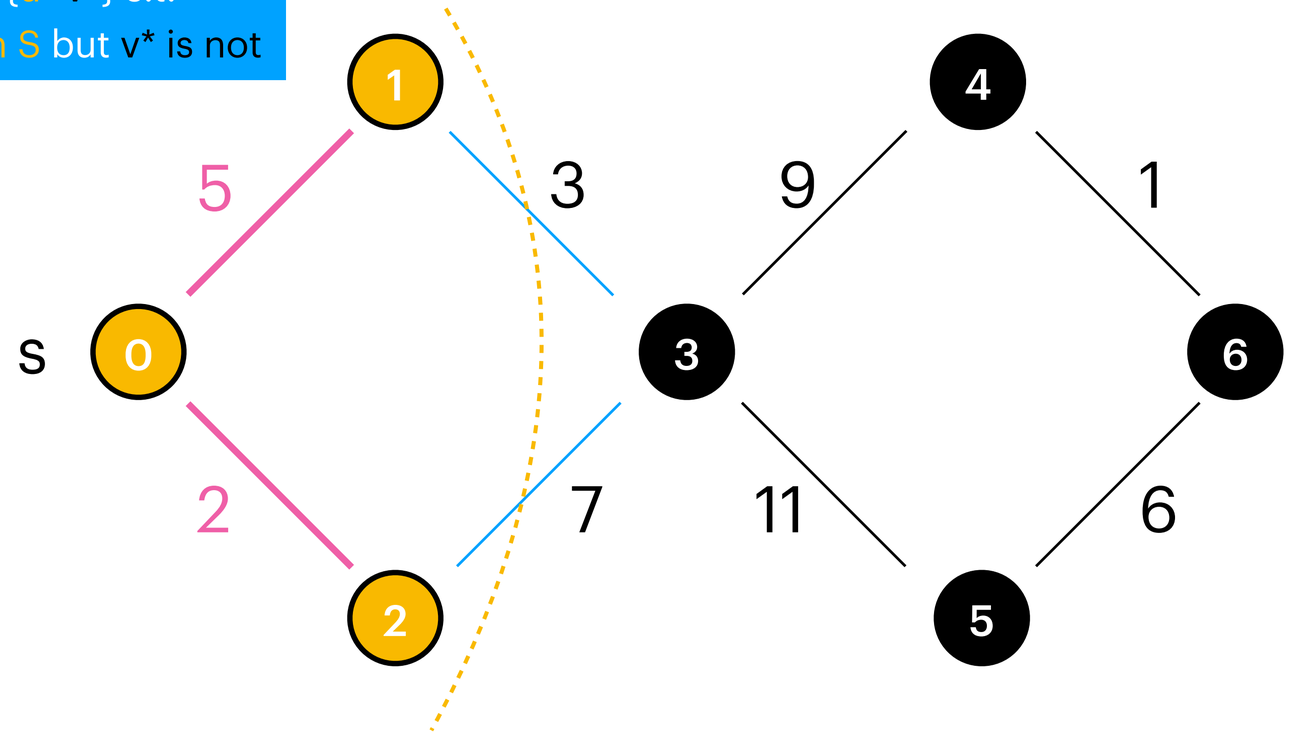
Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : { {0,2} , {0,1} }

S : { 0 , 2 , 1 }

edges {u* v*} s.t.
u* is in S but v* is not



MST

Prim's Algorithm

F : edges of the MST

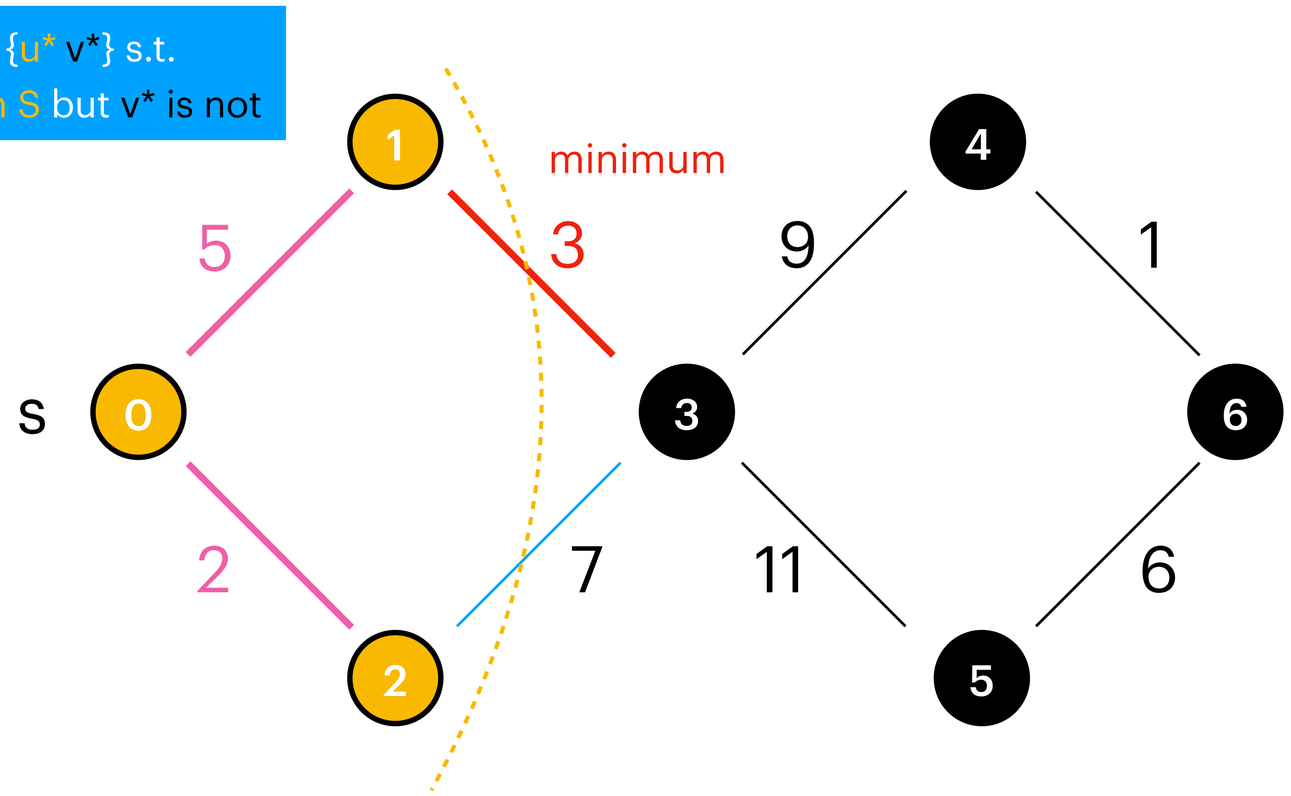
S : connected component set

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: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

edges {u* v*} s.t.
u* is in S but v* is not



F : { {0,2} , {0,1} }

S : { 0 , 2 , 1 }

MST

Prim's Algorithm

F : edges of the MST

S : connected component set

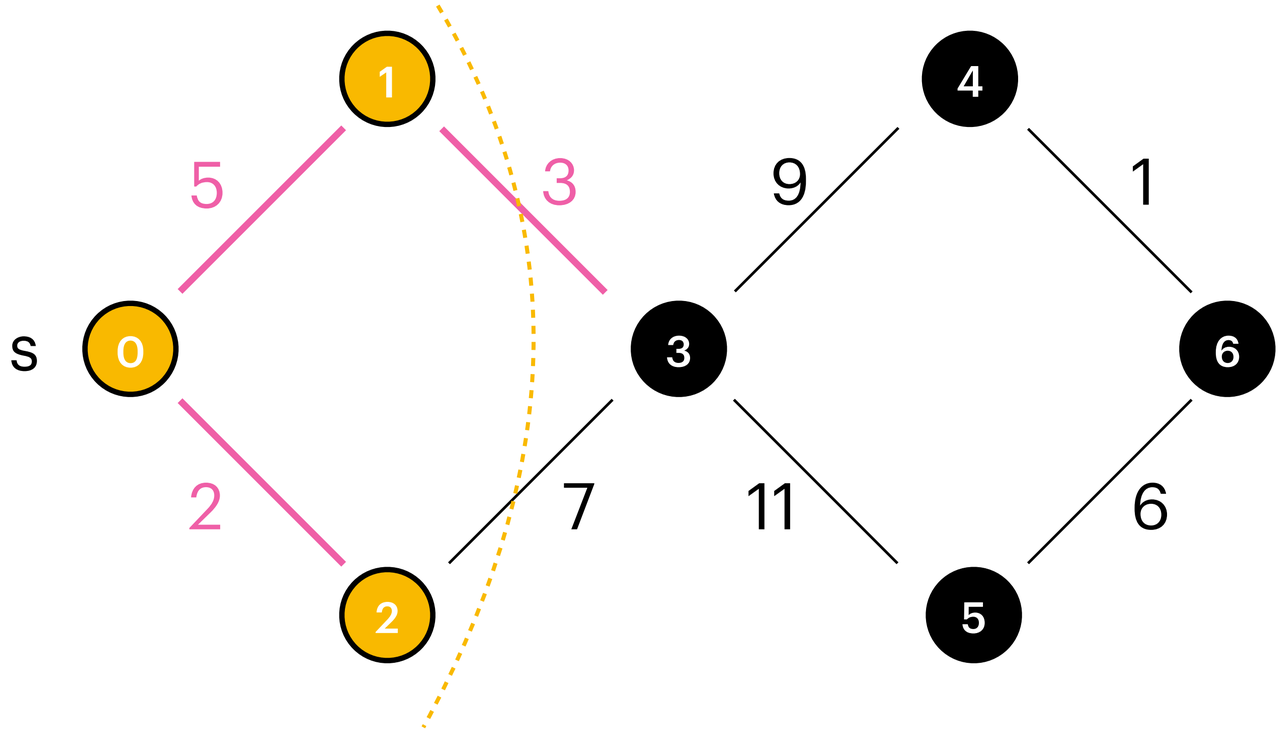
4 : find the minimum edge {u*,v*} s.t.
u* is in S but v* is not

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : { {0,2} , {0,1} , {1,3} }

S : { 0 , 2 , 1 }



MST

Prim's Algorithm

F : edges of the MST

S : connected component set

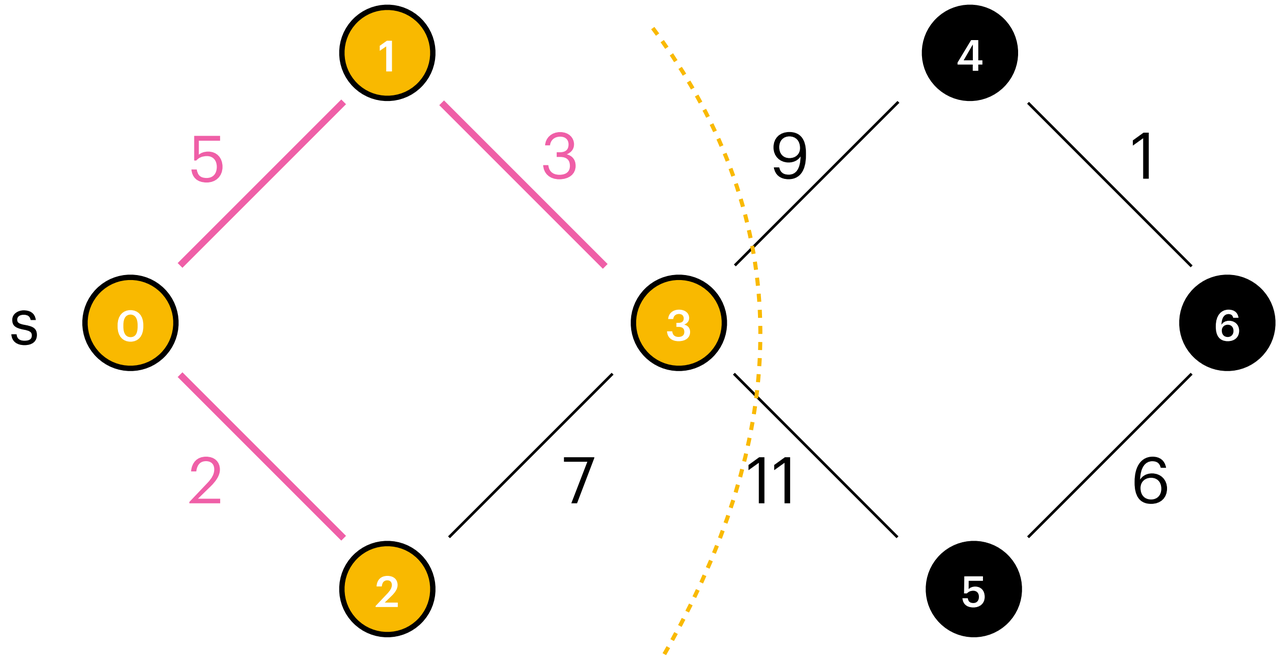
4 : find the minimum edge {u*,v*} s.t.
u* is in S but v* is not

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : { {0,2} , {0,1} , {1,3} }

S : { 0 , 2 , 1 , 3}



MST

Prim's Algorithm

F : edges of the MST

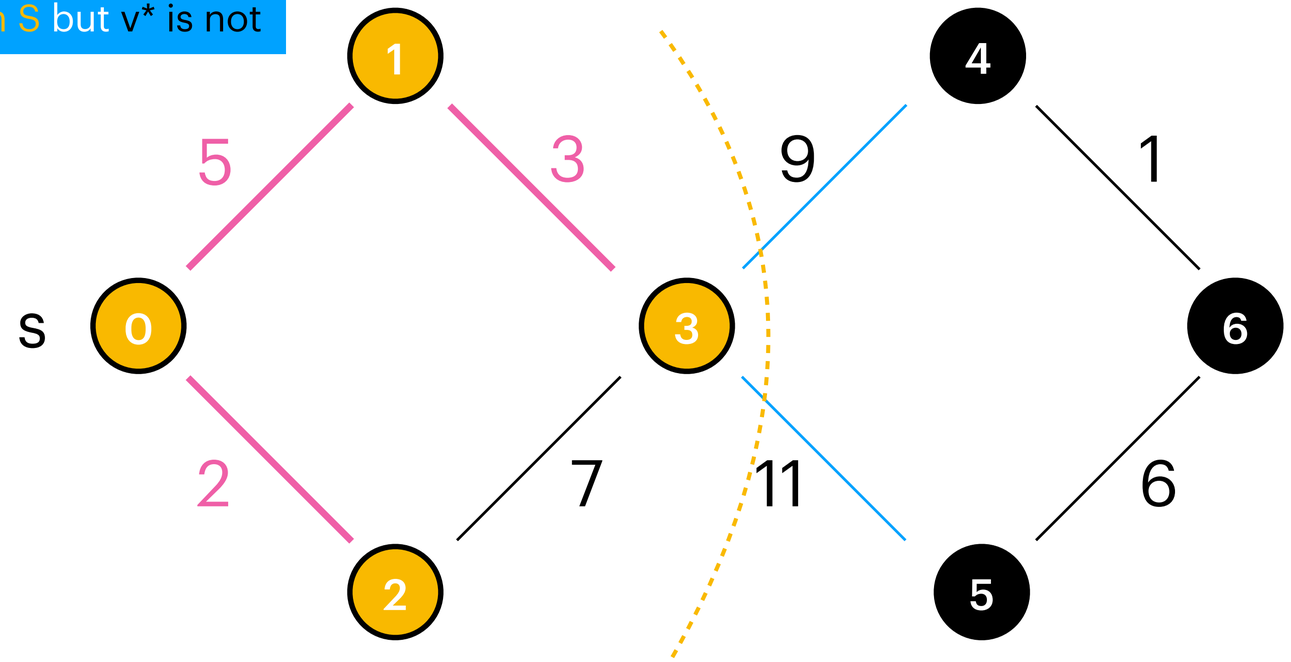
S : connected component set

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: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

edges {u* v*} s.t.
u* is in S but v* is not



F : { {0,2} , {0,1} , {1,3} }

S : { 0 , 2 , 1 , 3}

MST

Prim's Algorithm

F : edges of the MST

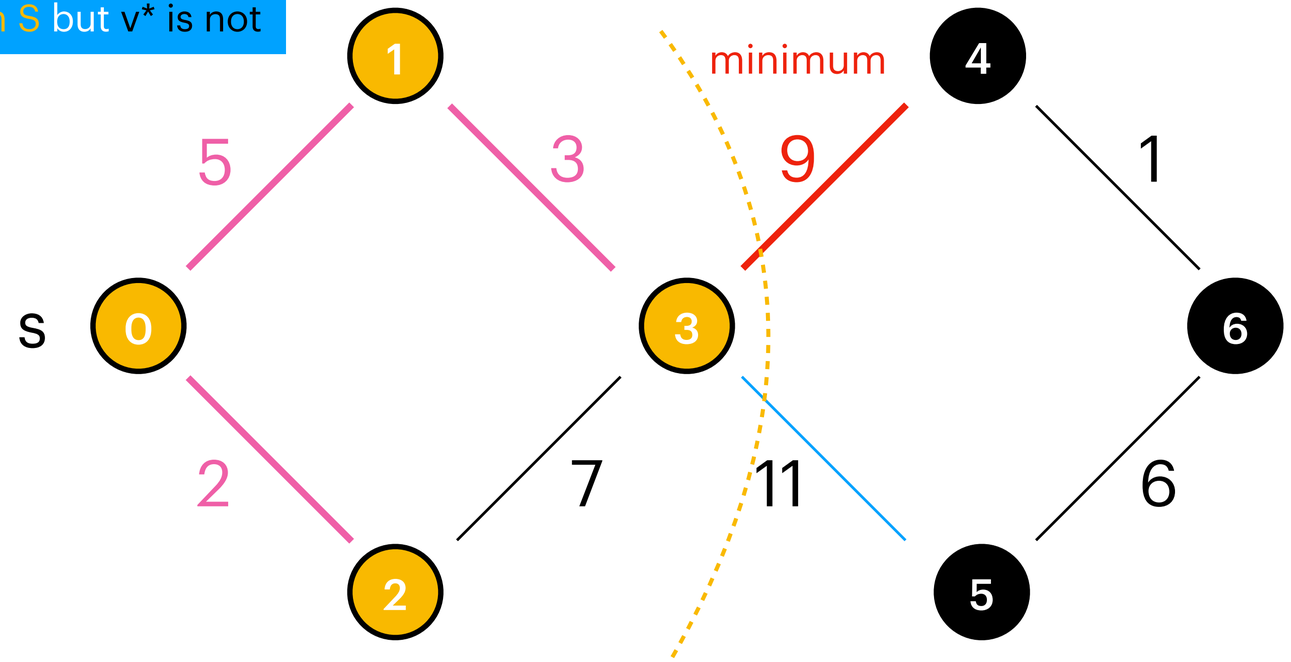
S : connected component set

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: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

edges {u* v*} s.t.
u* is in S but v* is not



F : { {0,2} , {0,1} , {1,3} }

S : { 0 , 2 , 1 , 3}

MST

Prim's Algorithm

F : edges of the MST

S : connected component set

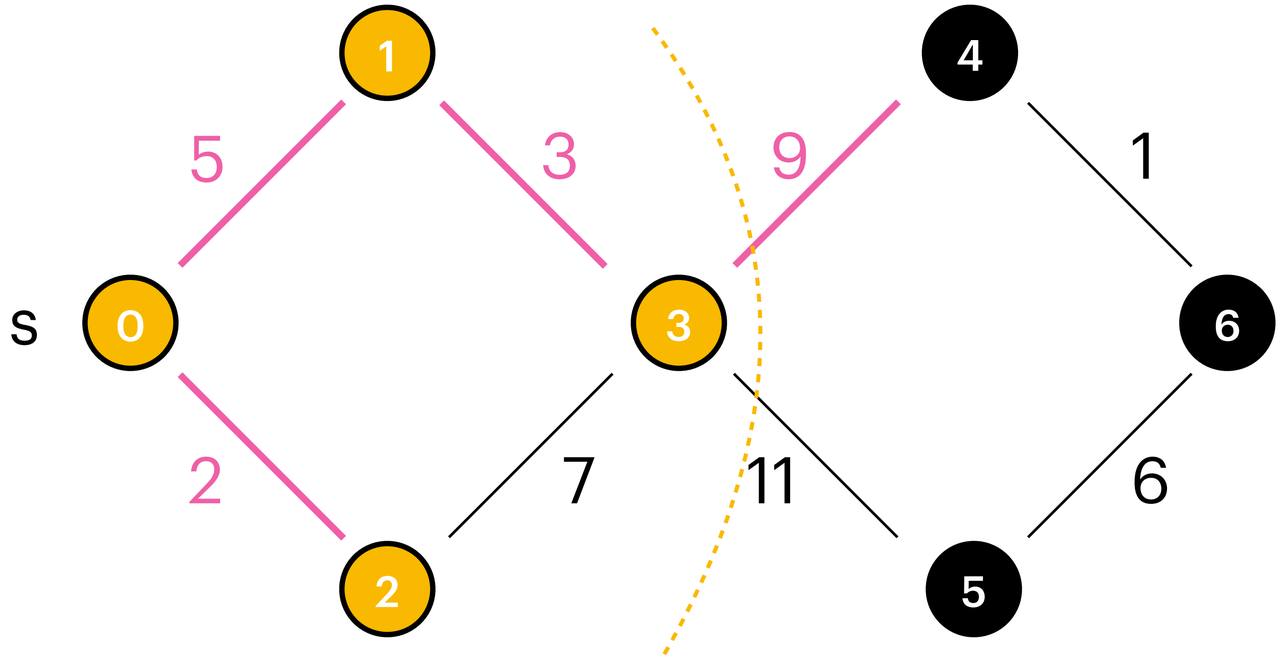
4 : find the minimum edge {u*,v*} s.t.
u* is in S but v* is not

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : { {0,2} , {0,1} , {1,3} , {3,9} }

S : { 0 , 2 , 1 , 3}



MST

Prim's Algorithm

F : edges of the MST

S : connected component set

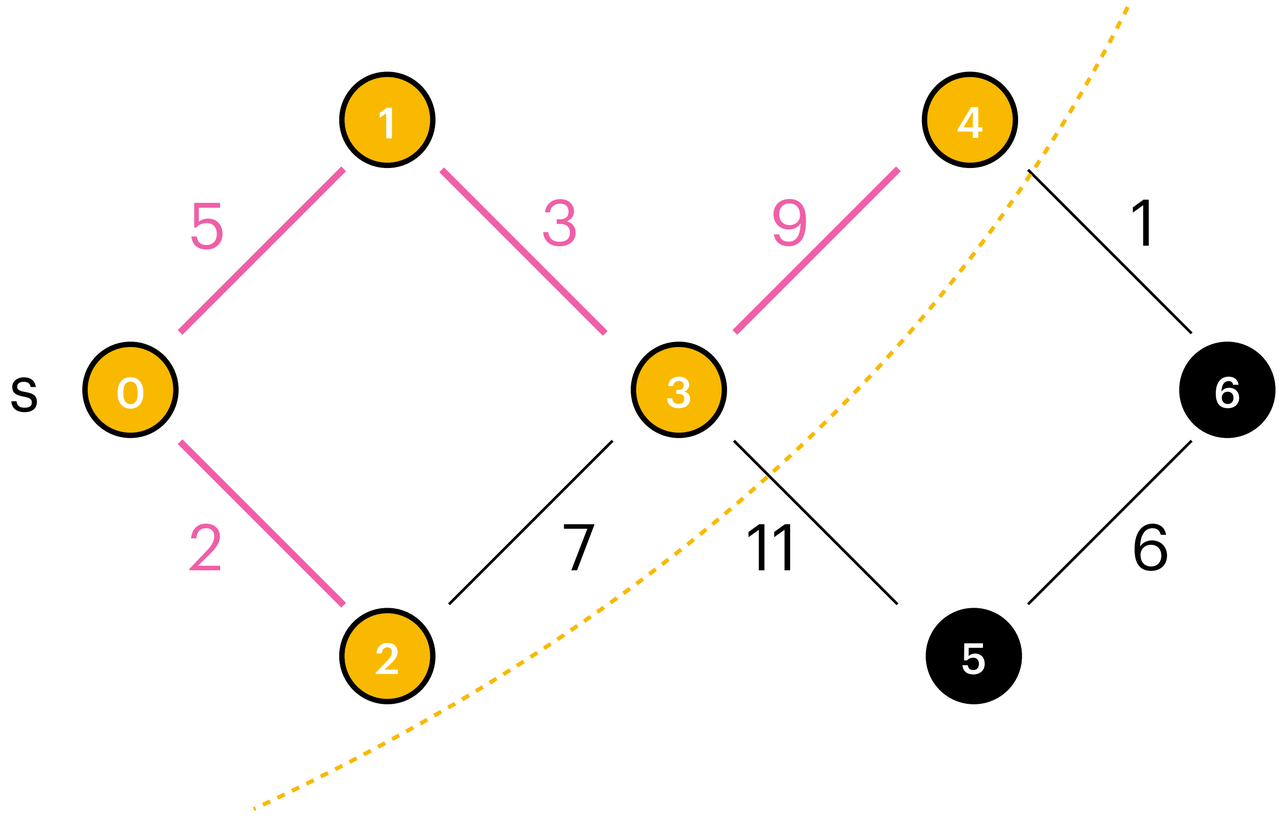
4 : find the minimum edge {u*,v*} s.t.
u* is in S but v* is not

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : { {0,2} , {0,1} , {1,3} , {3,9} }

S : { 0 , 2 , 1 , 3 , 4 }



MST

Prim's Algorithm

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

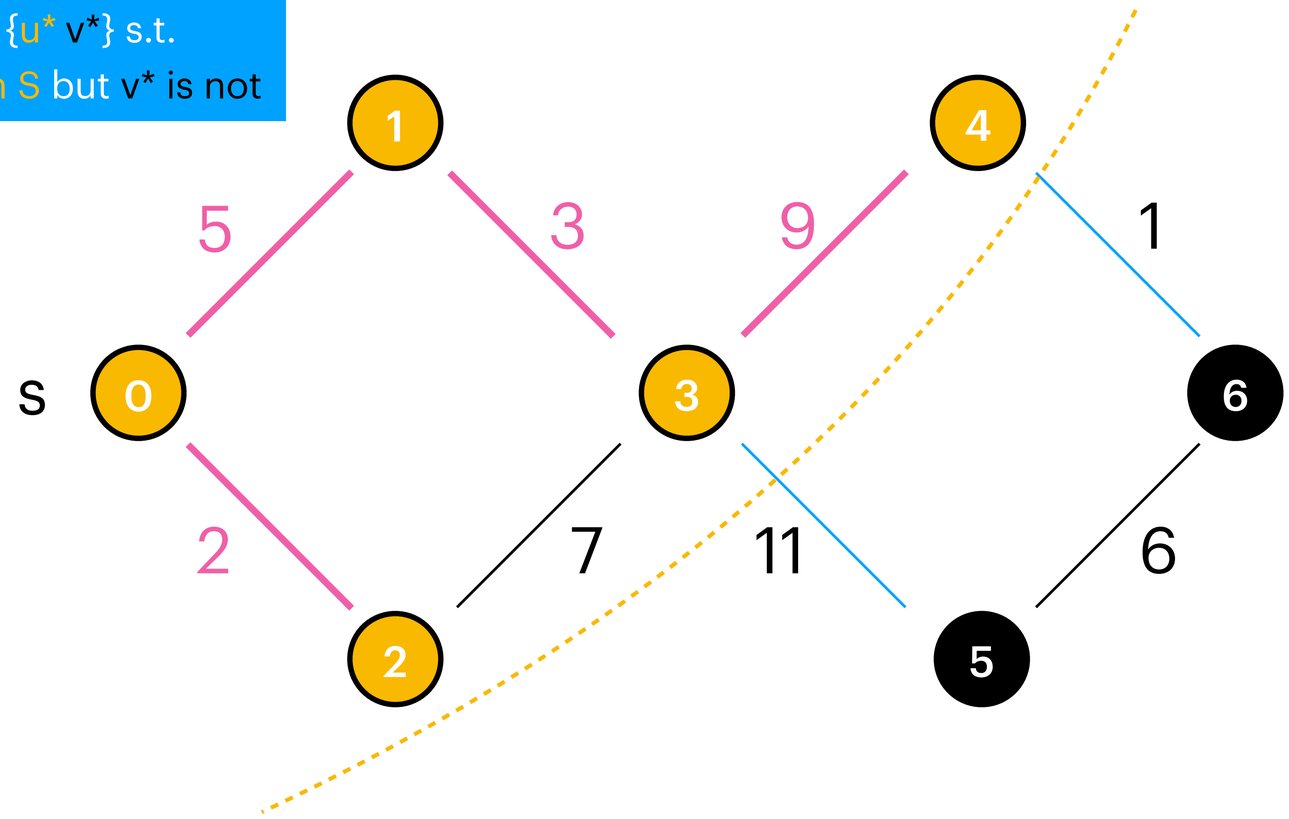
Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

edges {u* v*} s.t.
u* is in S but v* is not

F : { {0,2} , {0,1} , {1,3} , {3,9} }

S : { 0 , 2 , 1 , 3 , 4 }



MST

Prim's Algorithm

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

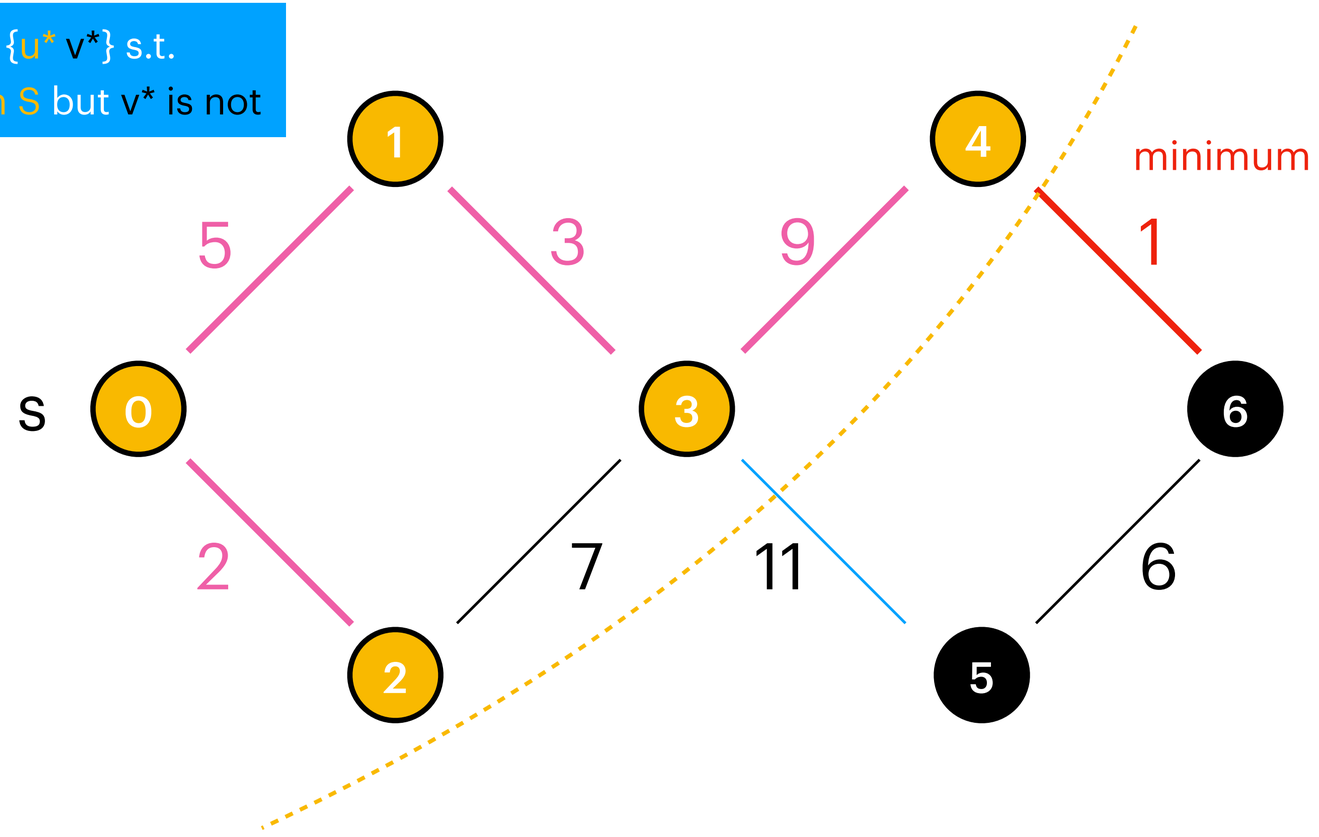
Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

edges {u* v*} s.t.
u* is in S but v* is not

F : { {0,2} , {0,1} , {1,3} , {3,9} }

S : { 0 , 2 , 1 , 3 , 4 }



MST

Prim's Algorithm

F : edges of the MST

S : connected component set

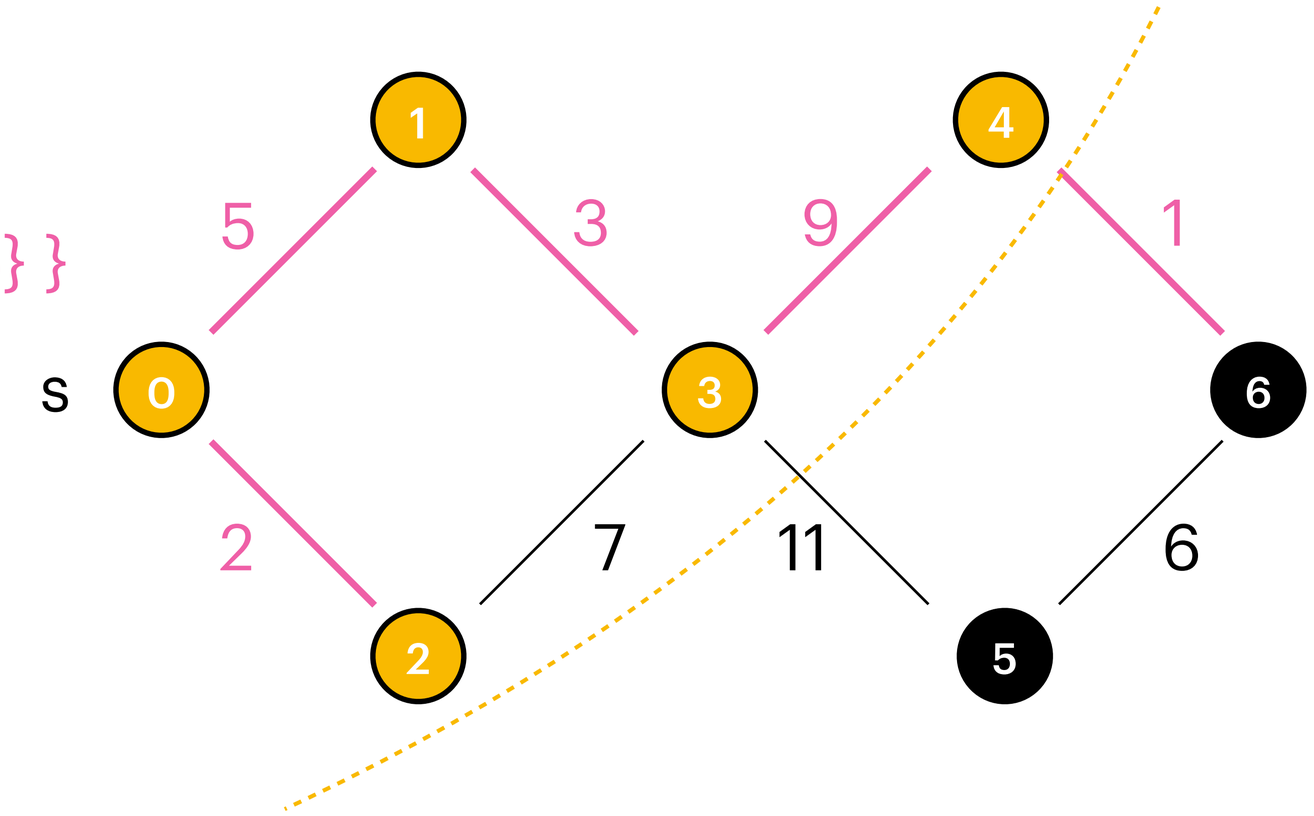
4 : find the minimum edge {u*,v*} s.t.
u* is in S but v* is not

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} }

S : { 0 , 2 , 1 , 3 , 4 }



MST

Prim's Algorithm

F : edges of the MST

S : connected component set

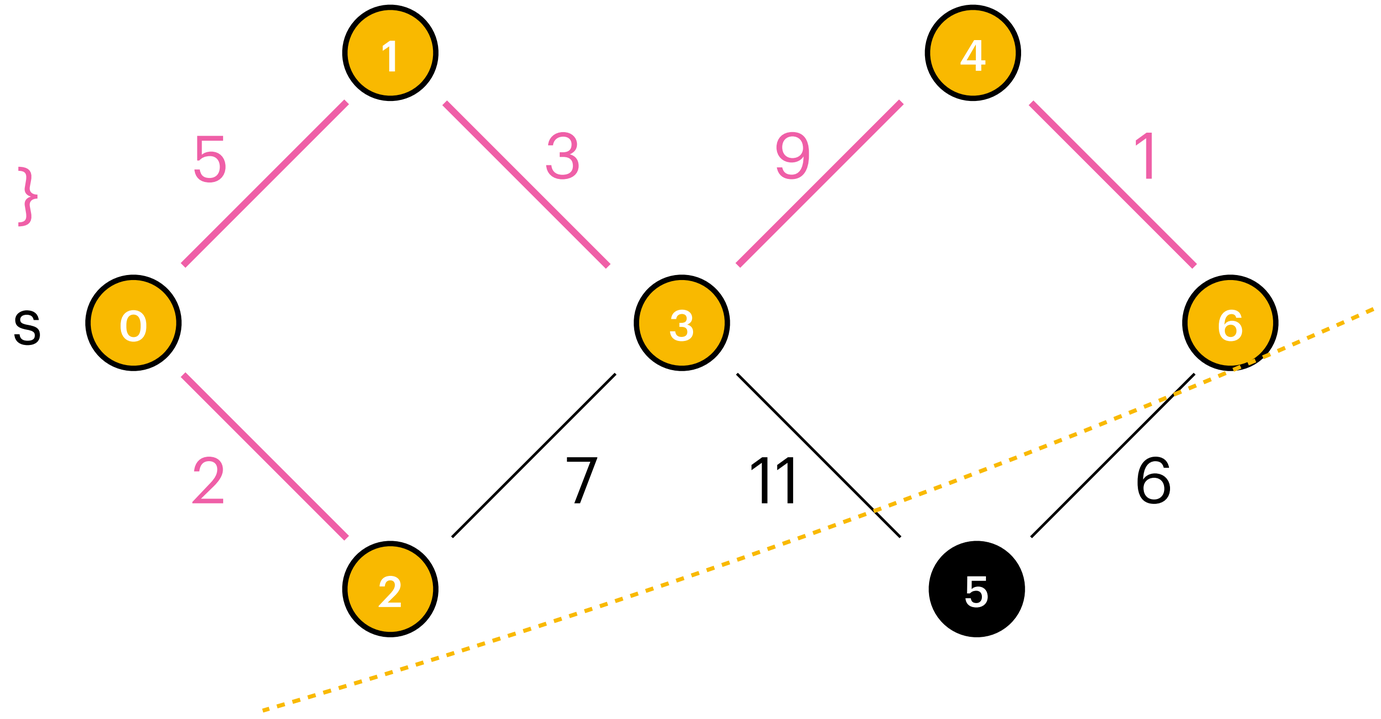
4 : find the minimum edge {u*,v*} s.t.
u* is in S but v* is not

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} }

S : { 0 , 2 , 1 , 3 , 4 , 6 }



MST

Prim's Algorithm

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

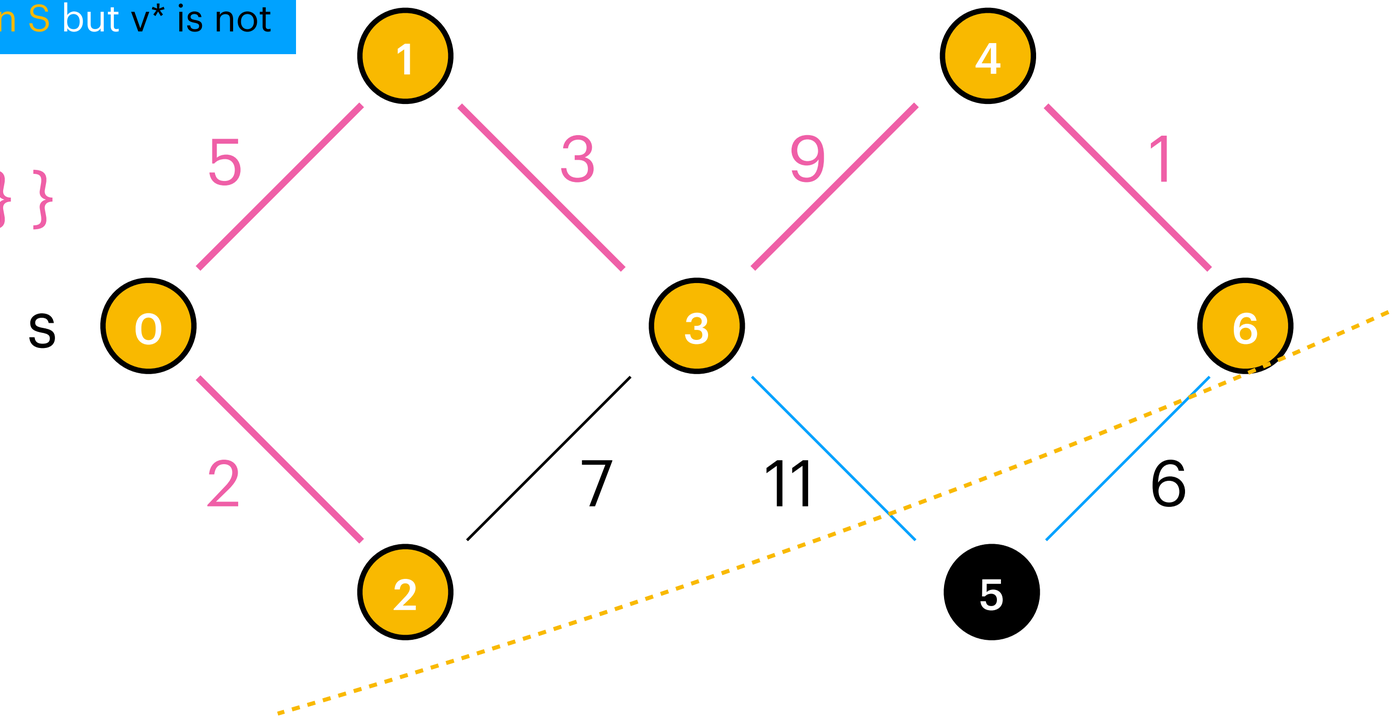
Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

edges {u* v*} s.t.
u* is in S but v* is not

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} }

S : { 0 , 2 , 1 , 3 , 4 , 6 }



MST

Prim's Algorithm

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

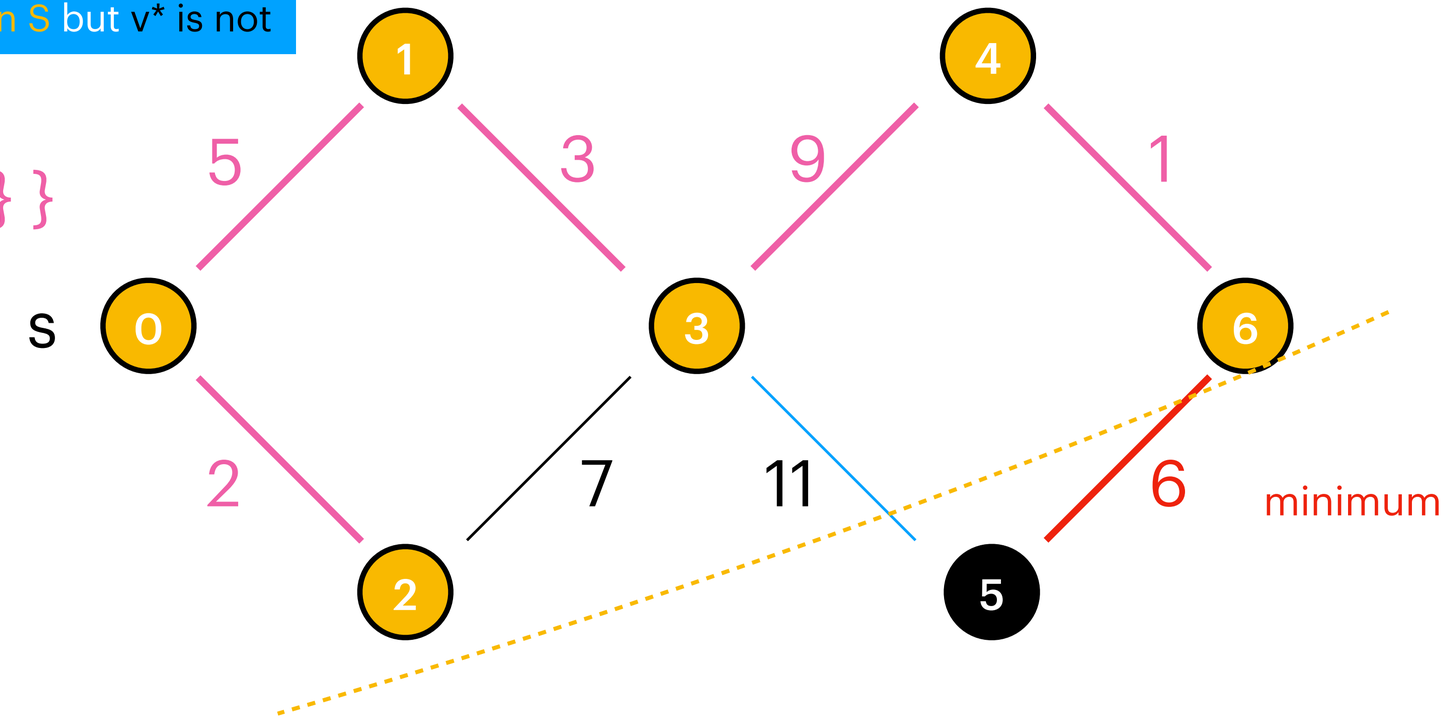
Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

edges {u* v*} s.t.
u* is in S but v* is not

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} }

S : { 0 , 2 , 1 , 3 , 4 , 6 }



MST

Prim's Algorithm

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

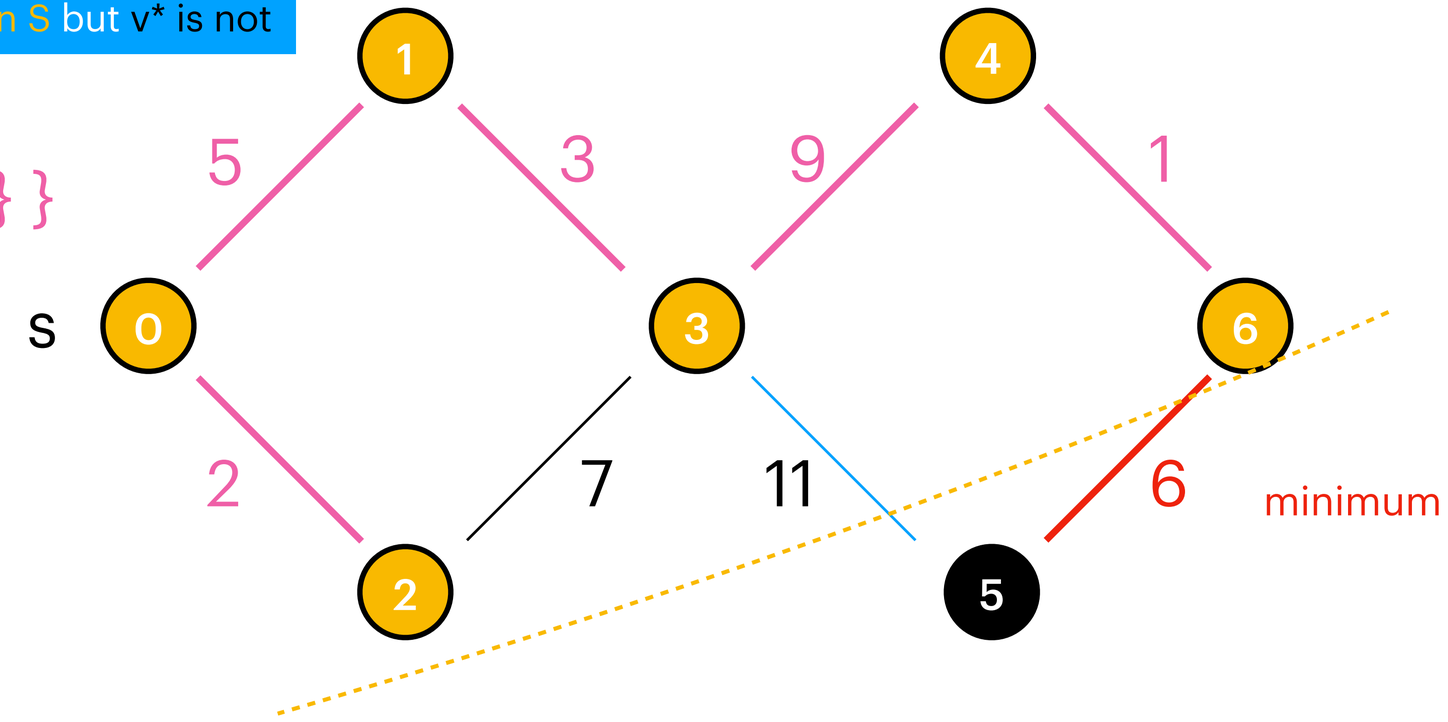
Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

edges {u* v*} s.t.
u* is in S but v* is not

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} }

S : { 0 , 2 , 1 , 3 , 4 , 6 }



MST

Prim's Algorithm

F : edges of the MST

S : connected component set

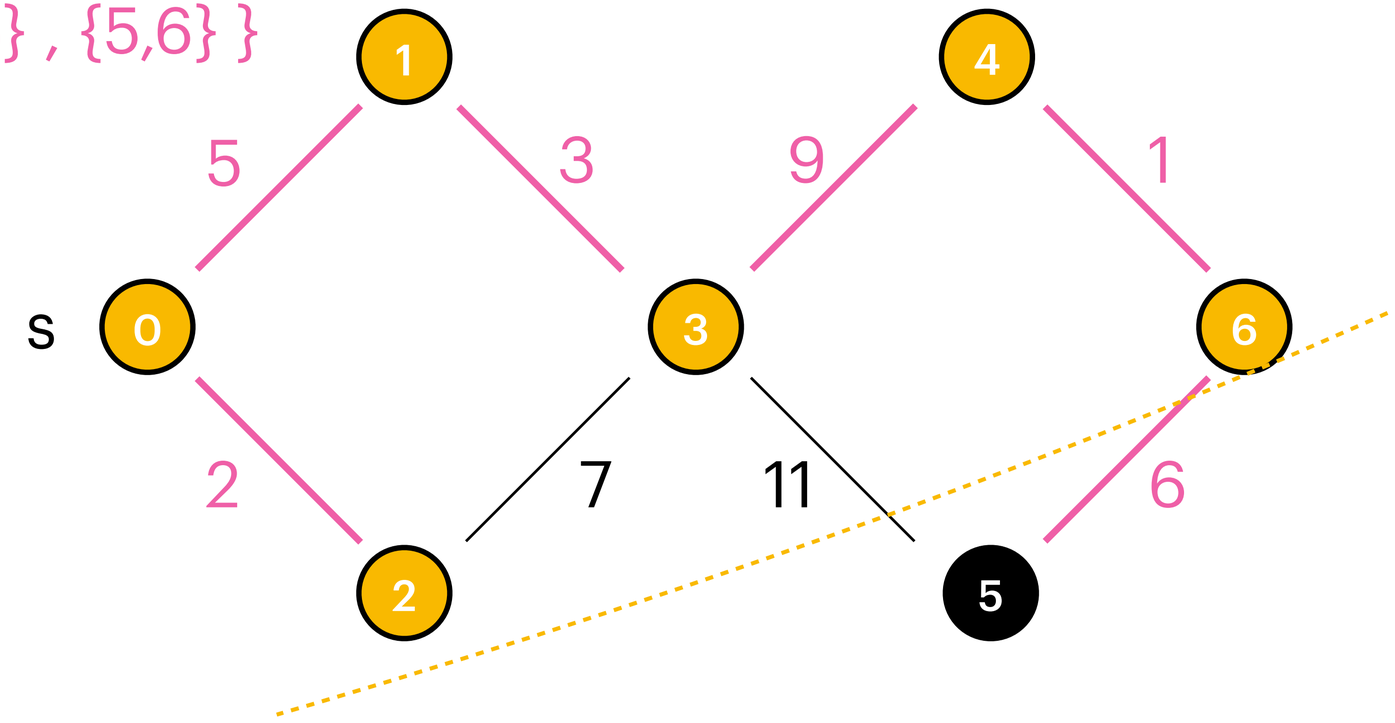
4 : find the minimum edge {u*,v*} s.t.
u* is in S but v* is not

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} , {5,6} }

S : { 0 , 2 , 1 , 3 , 4 , 6 }



MST

Prim's Algorithm

F : edges of the MST

S : connected component set

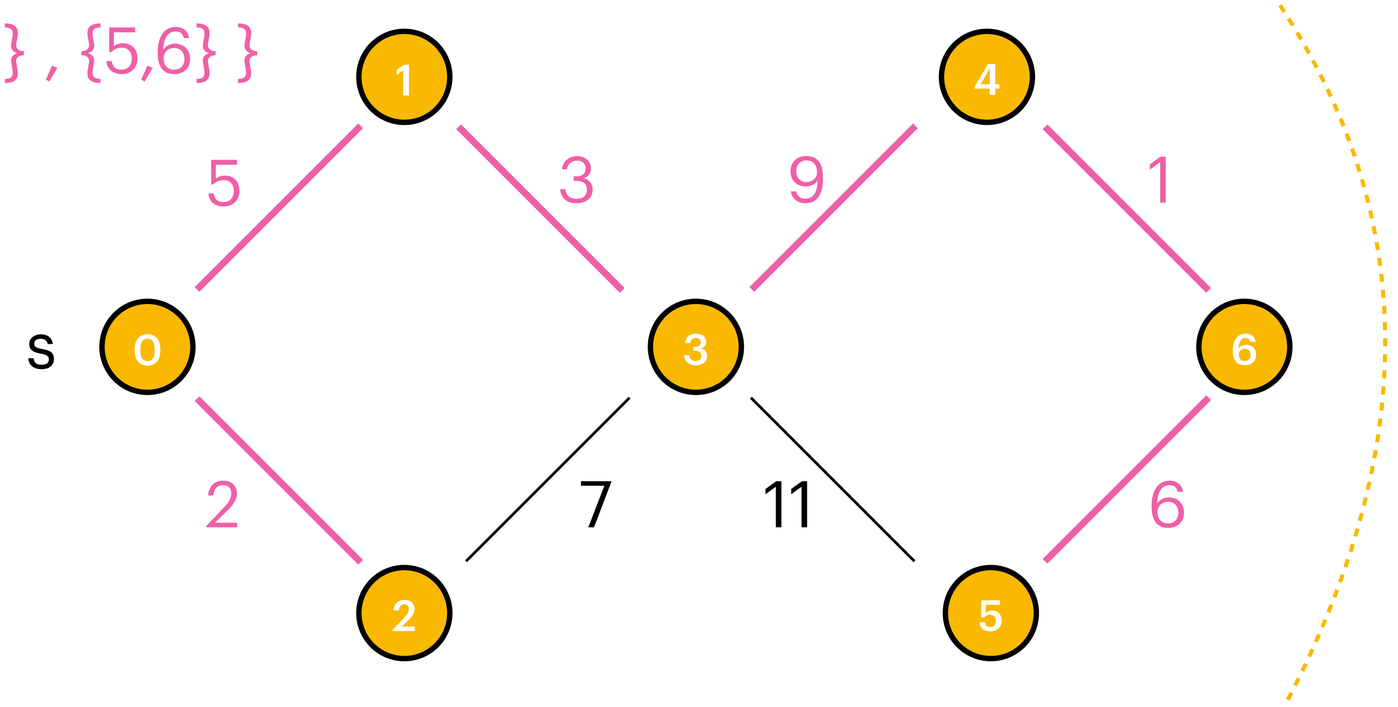
4 : find the minimum edge {u*,v*} s.t.
u* is in S but v* is not

Algorithm 9 Prim(G, s) (allgemeine Form)

1: $F \leftarrow \varnothing$
2: $S \leftarrow \{s\}$
3: **while** F nicht Spannbaum **do**
4: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

F : { {0,2} , {0,1} , {1,3} , {3,9} , {4,6} , {5,6} }

S : { 0 , 2 , 1 , 3 , 4 , 6 , 5 }



MST

Prim's Algorithm

F : edges of the MST

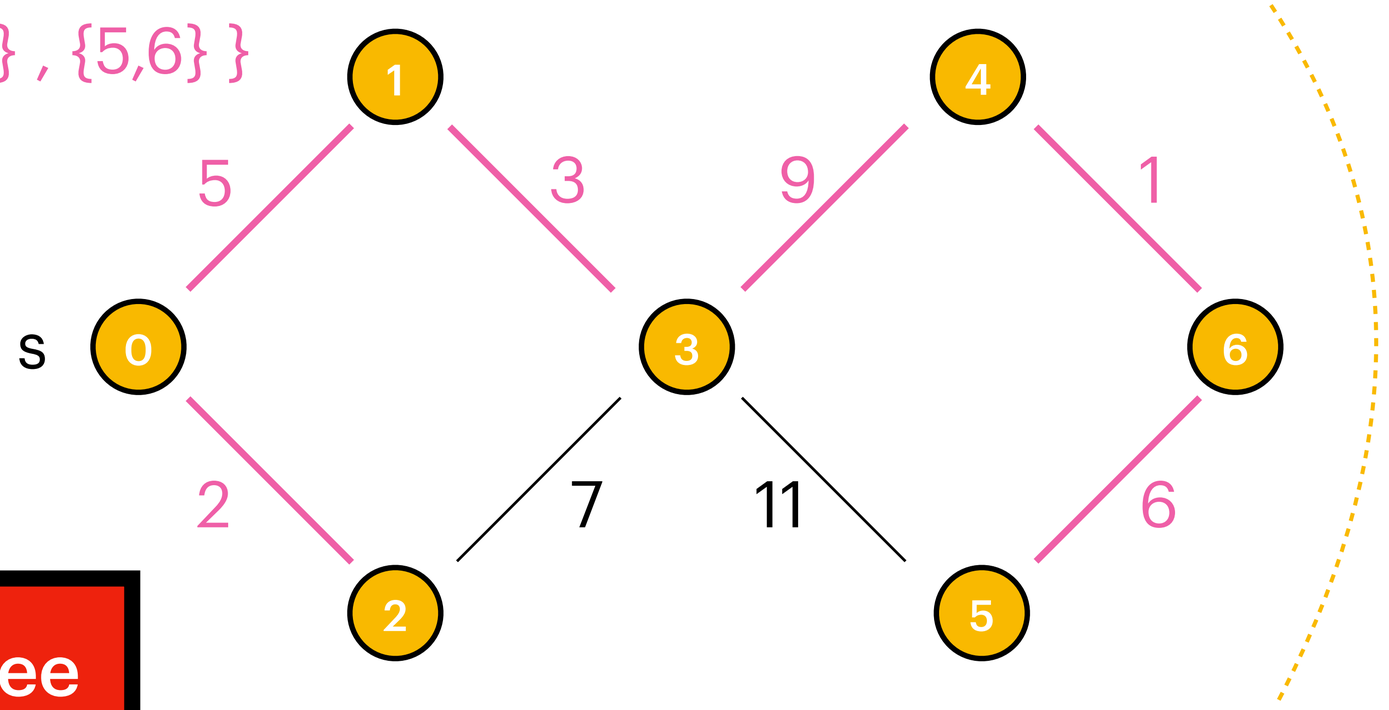
S : connected component set

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: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

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

MST

Prim's Algorithm

F : edges of the MST

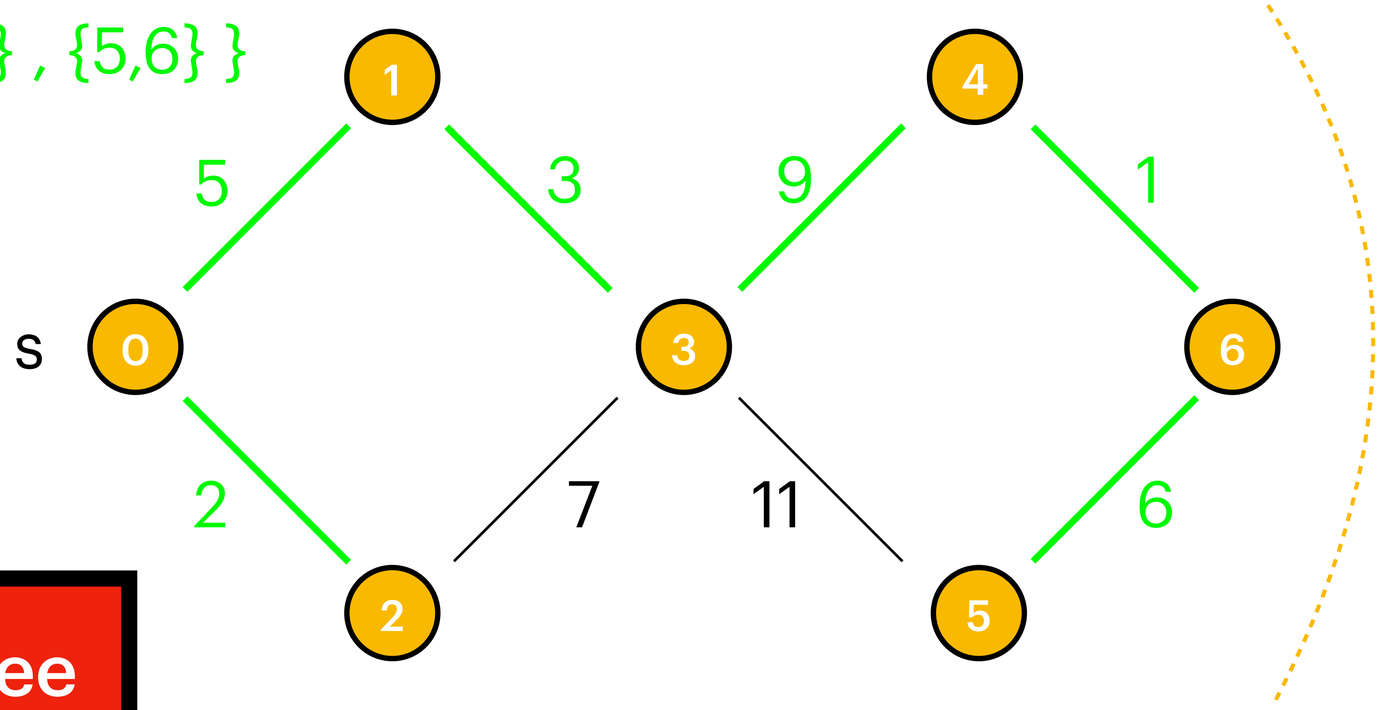
S : connected component set

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: $\quad u^*v^* \leftarrow$ minimale Kante an $S \quad (u^* \in S, v^* \notin S)$
5: $\quad F \leftarrow F \cup \{u^*v^*\}$
6: $\quad S \leftarrow S \cup \{v^*\}$

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

MST

Boruvka's Algorithm

Runtime : O ((|V| + |E|) * log n)

Algorithm 8 Boruvka(G)

1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F connected component of 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

MST

Boruvka's Algorithm

Runtime : O ((|V| + |E|) * log n)

Algorithm 8 Boruvka(G)

1: $F \leftarrow \emptyset$
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: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

find with DFS only using the edges in F !

connected components with edges from F

minimum edges near Ss

F : edges of the MST

MST

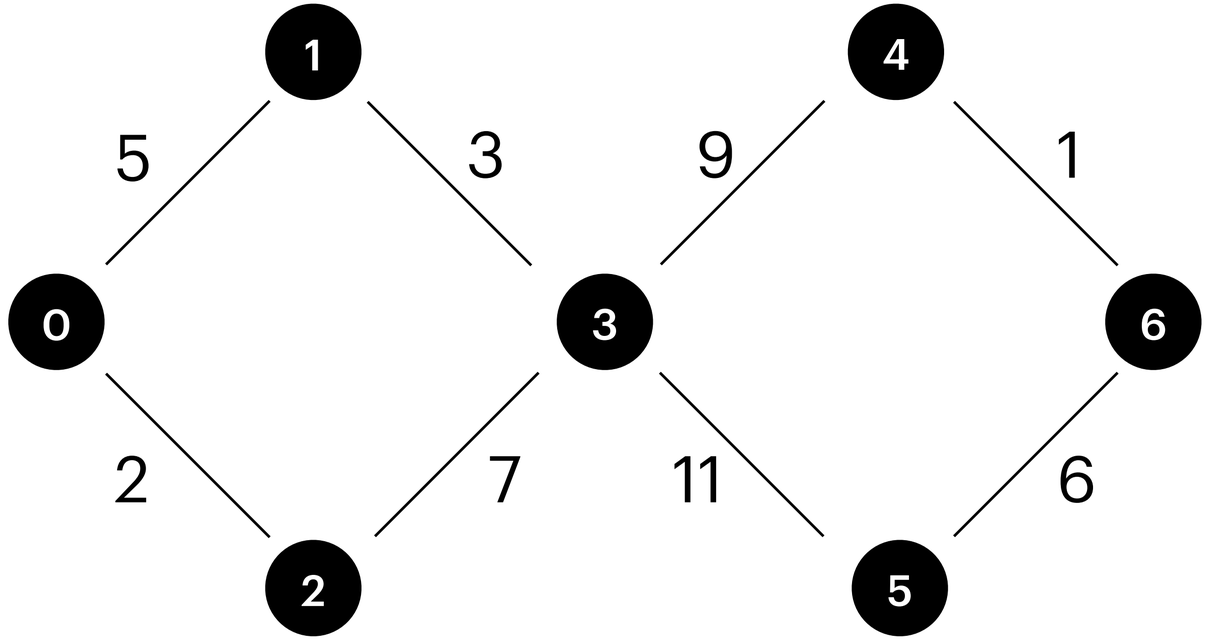
Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)

1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $\quad (S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $\quad (e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $\quad F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F :



MST

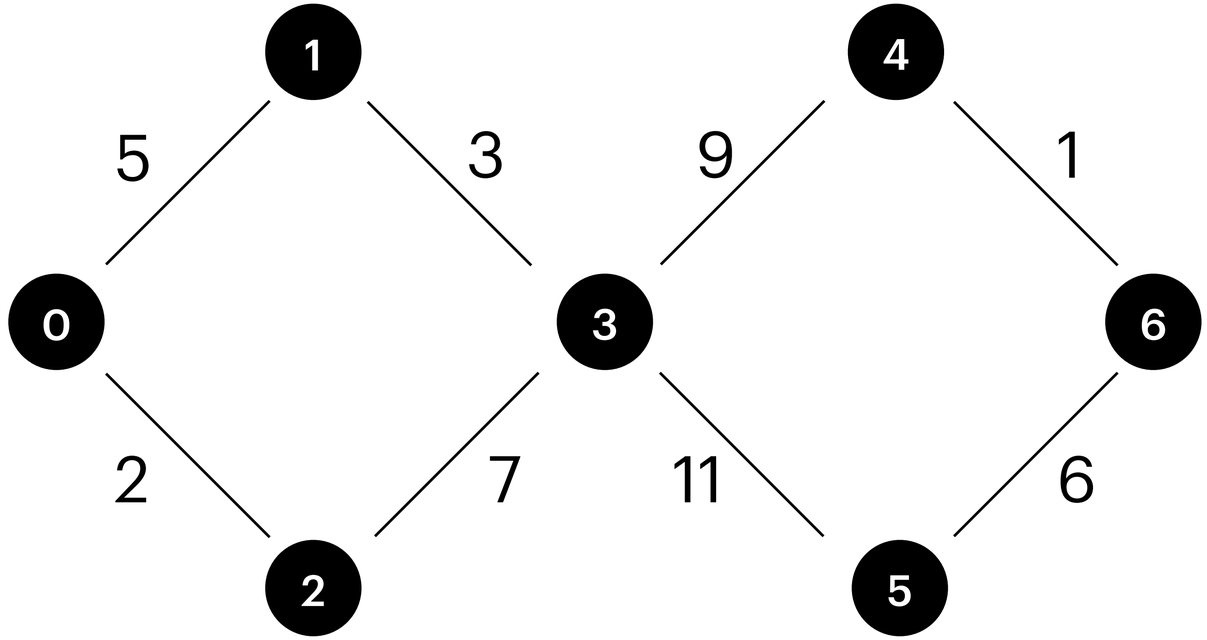
Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)

1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $\quad (S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $\quad (e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $\quad F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : ∅



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 : ∅

$S_1 = \{0\}$

$S_2 = \{1\}$

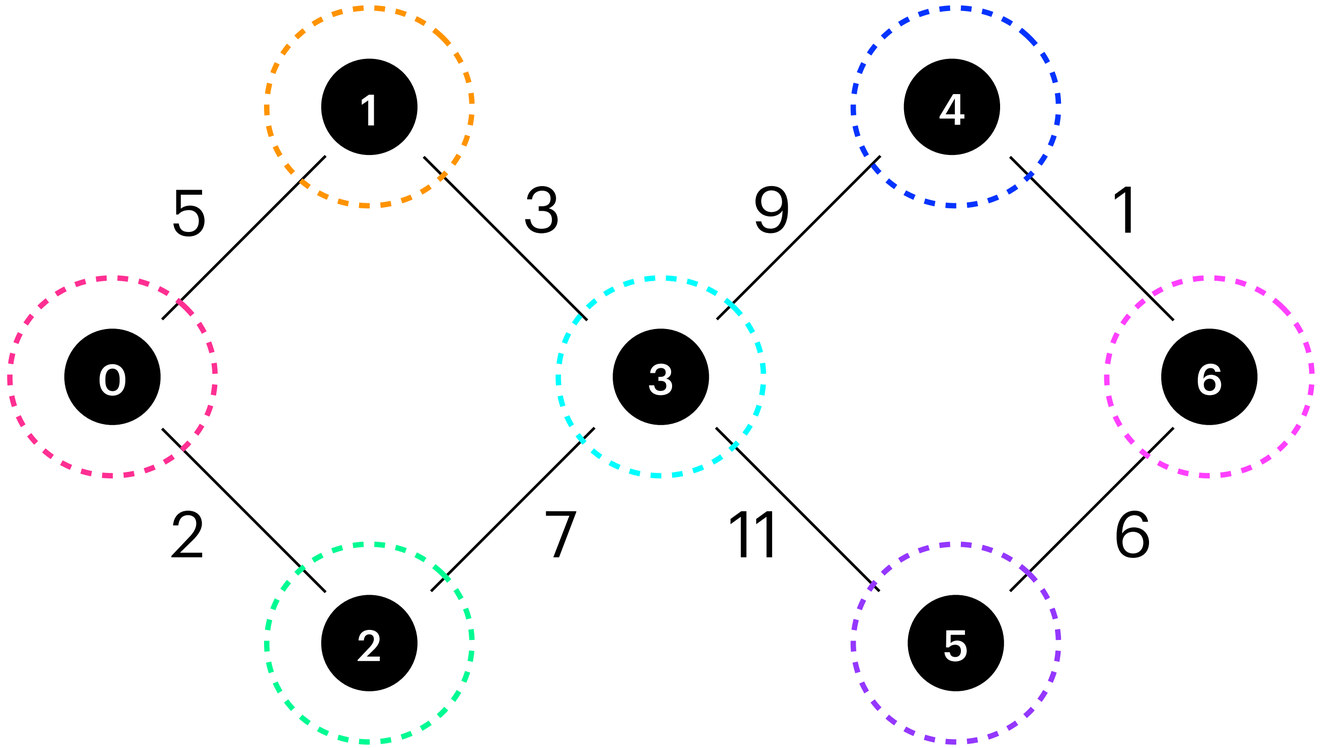
$S_3 = \{2\}$

$S_4 = \{3\}$

$S_5 = \{4\}$

$S_6 = \{5\}$

$S_7 = \{6\}$



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 : ∅

$S_1 = \{0\}$

$S_2 = \{1\}$

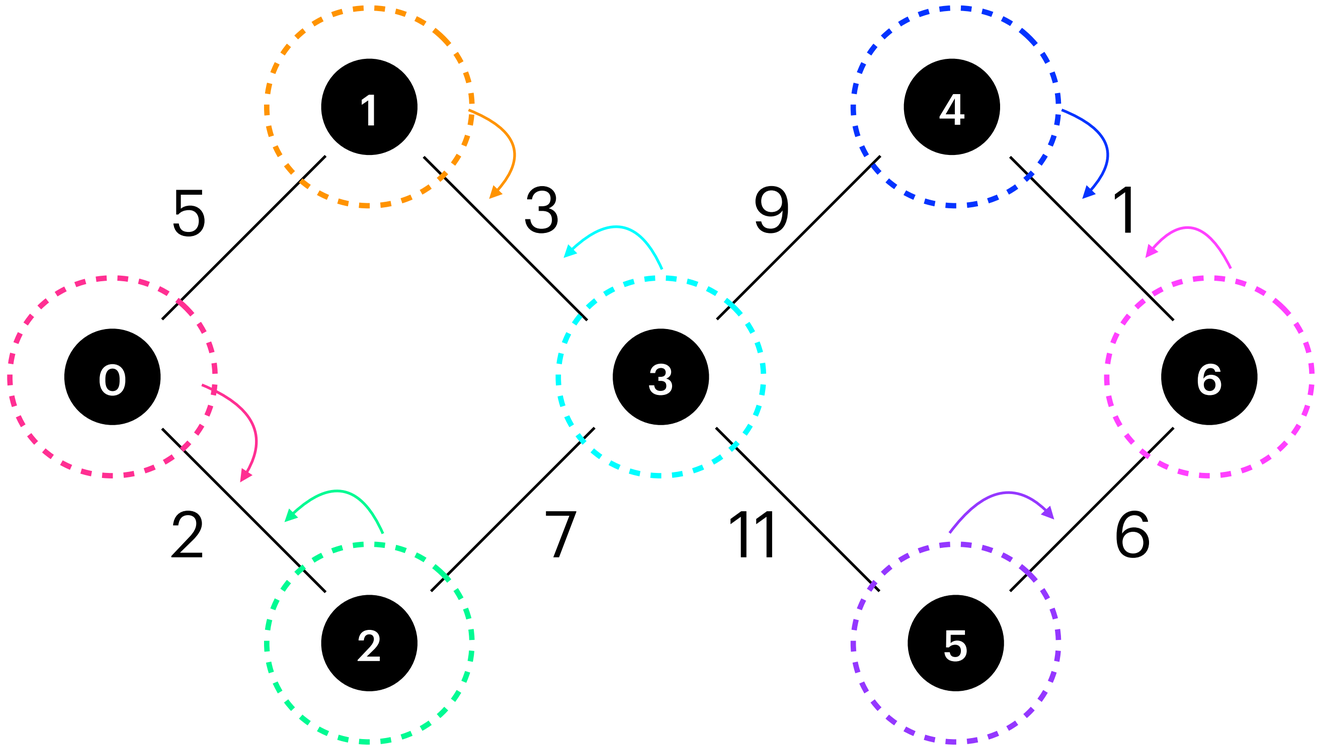
$S_3 = \{2\}$

$S_4 = \{3\}$

$S_5 = \{4\}$

$S_6 = \{5\}$

$S_7 = \{6\}$



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 : ∅

$S_1 = \{0\}$

$S_2 = \{1\}$

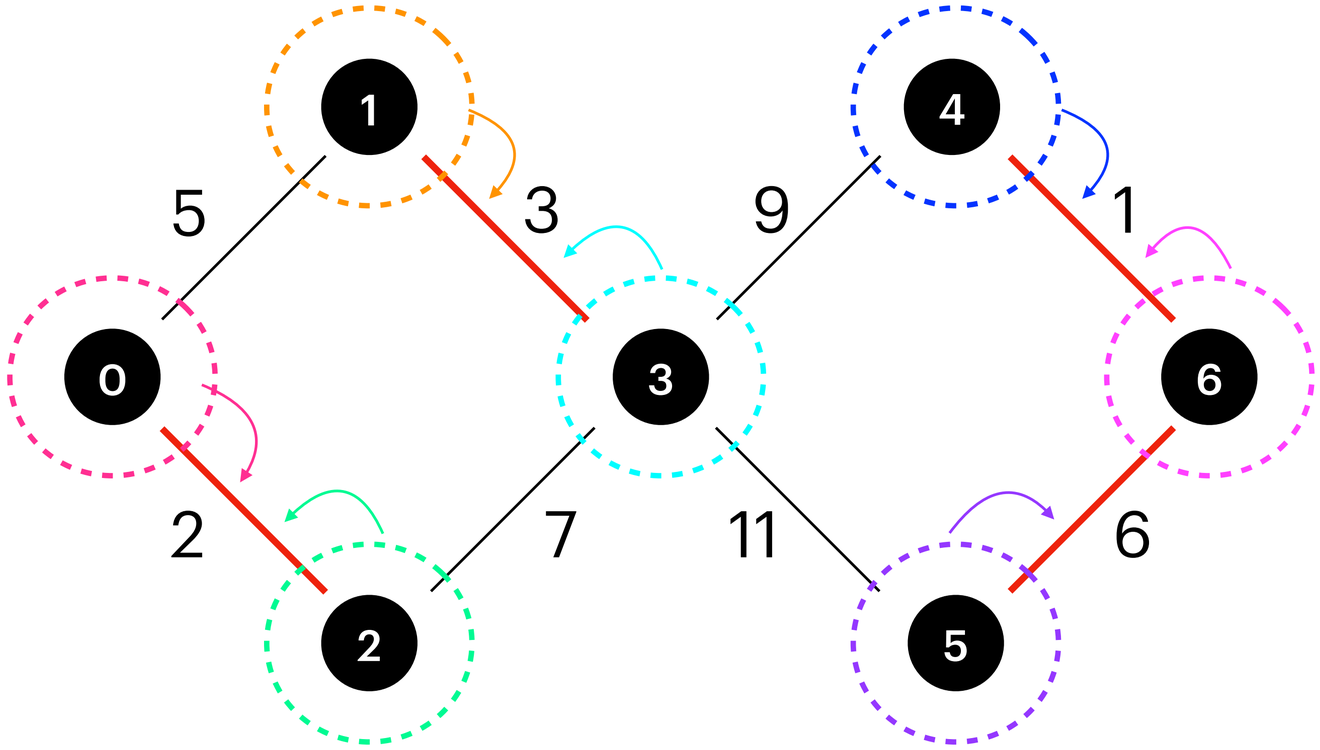
$S_3 = \{2\}$

$S_4 = \{3\}$

$S_5 = \{4\}$

$S_6 = \{5\}$

$S_7 = \{6\}$



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} }

$S_1 = \{0\}$

$S_2 = \{1\}$

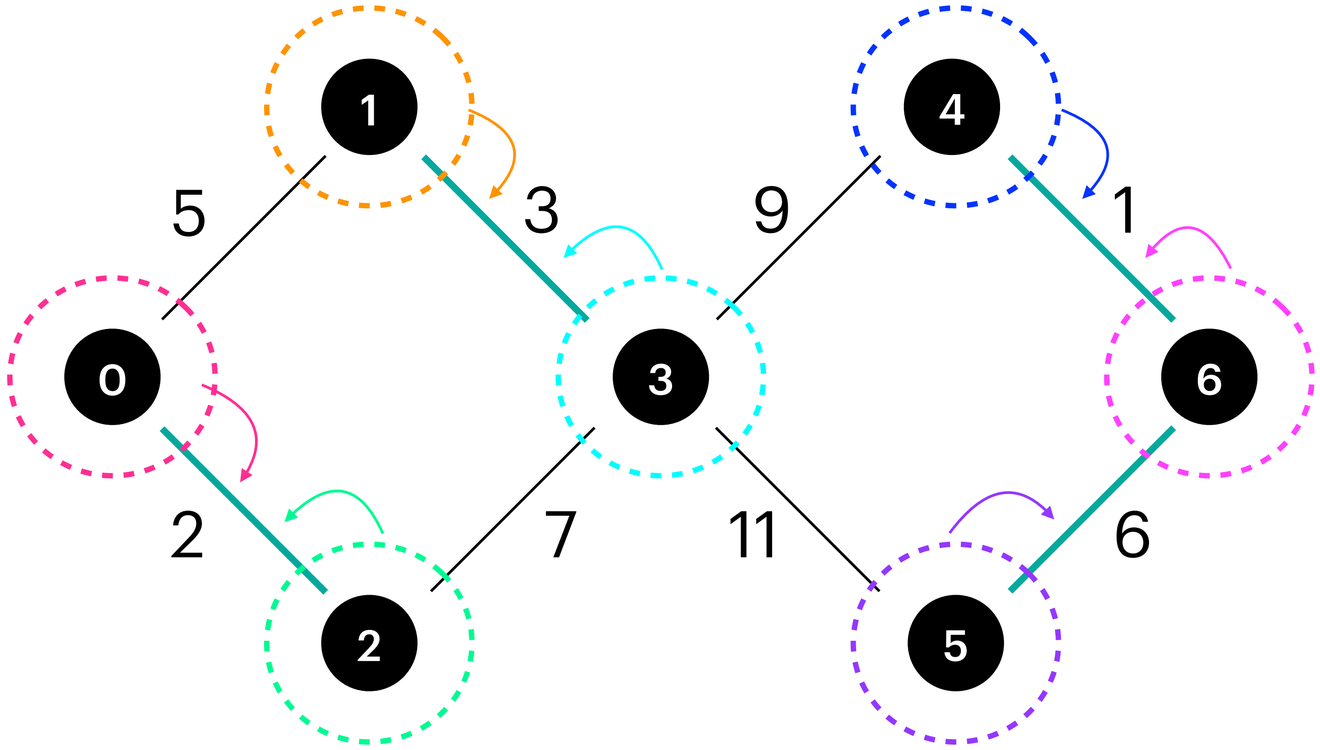
$S_3 = \{2\}$

$S_4 = \{3\}$

$S_5 = \{4\}$

$S_6 = \{5\}$

$S_7 = \{6\}$



MST

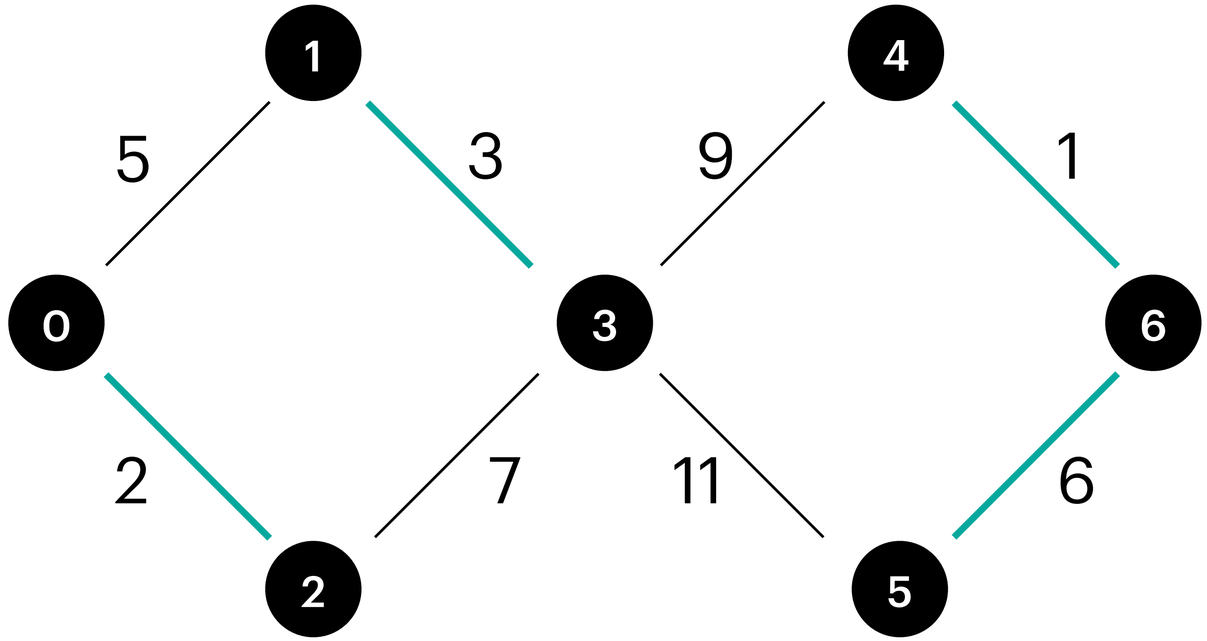
Boruvka's Algorithm

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} }



MST

Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)

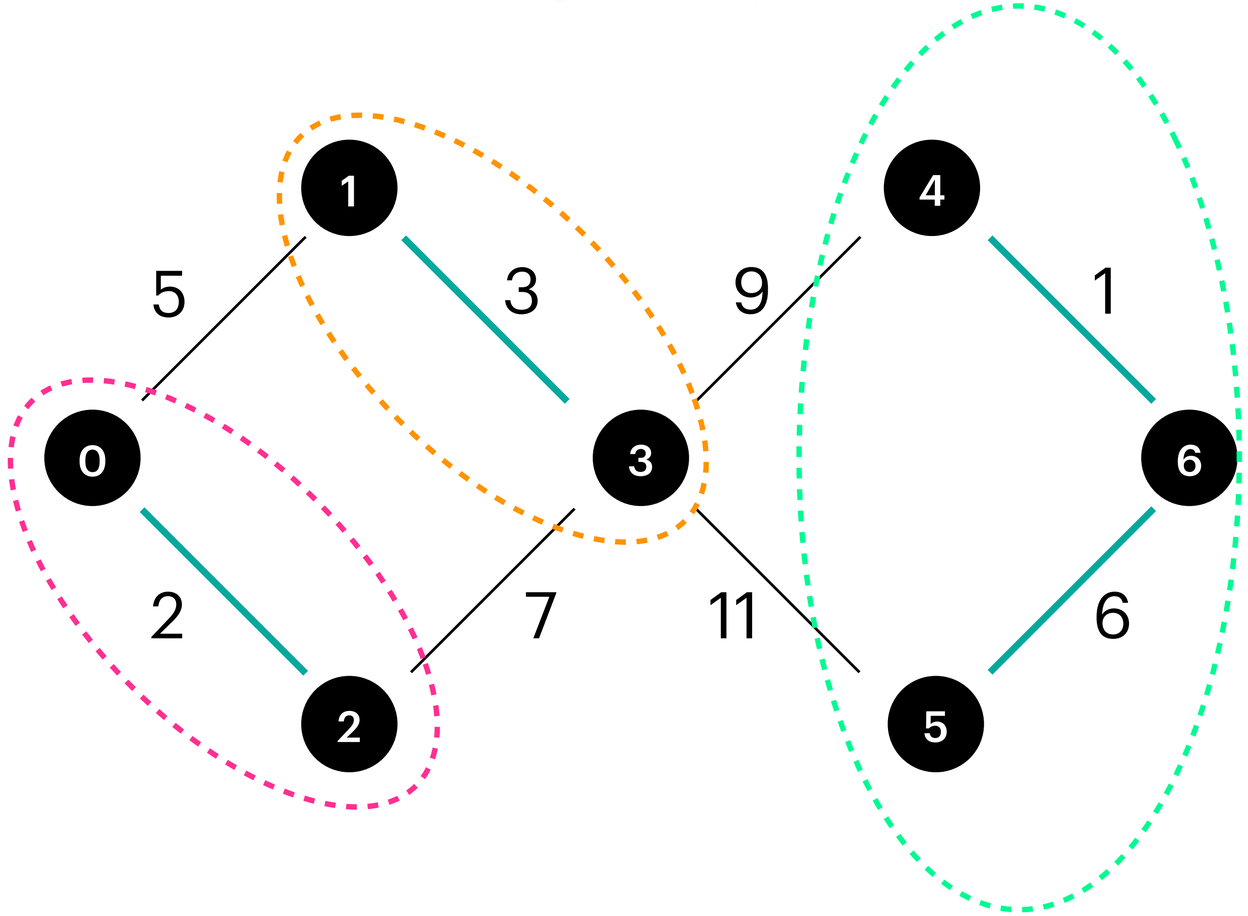
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : { {0,2} , {1,3} , {4,6} , {5,6} }

S_1 = {0,2}

S_2 = {1,3}

S_3 = {4,5,6}



MST

Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)

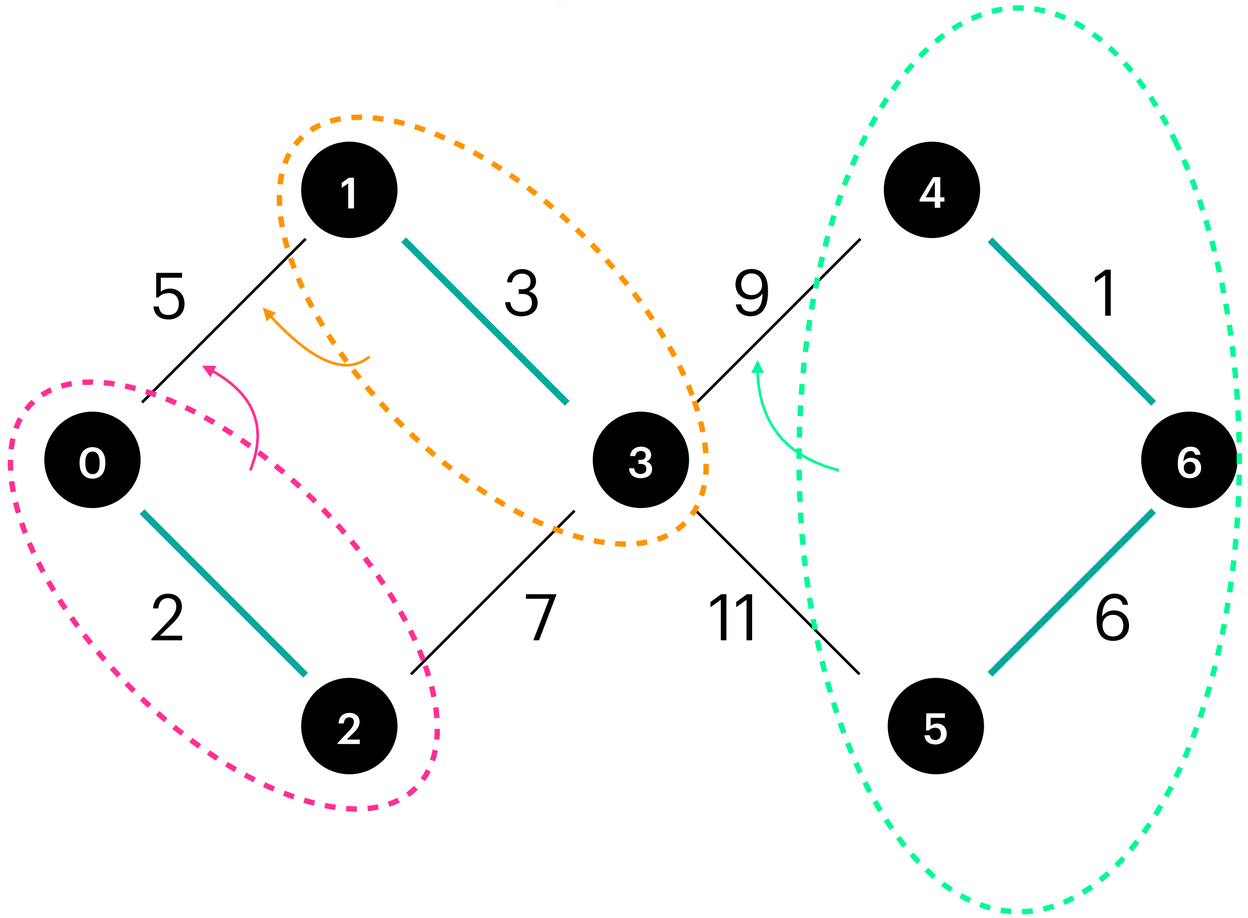
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : { {0,2} , {1,3} , {4,6} , {5,6} }

S_1 = {0,2}

S_2 = {1,3}

S_3 = {4,5,6}



MST

Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)

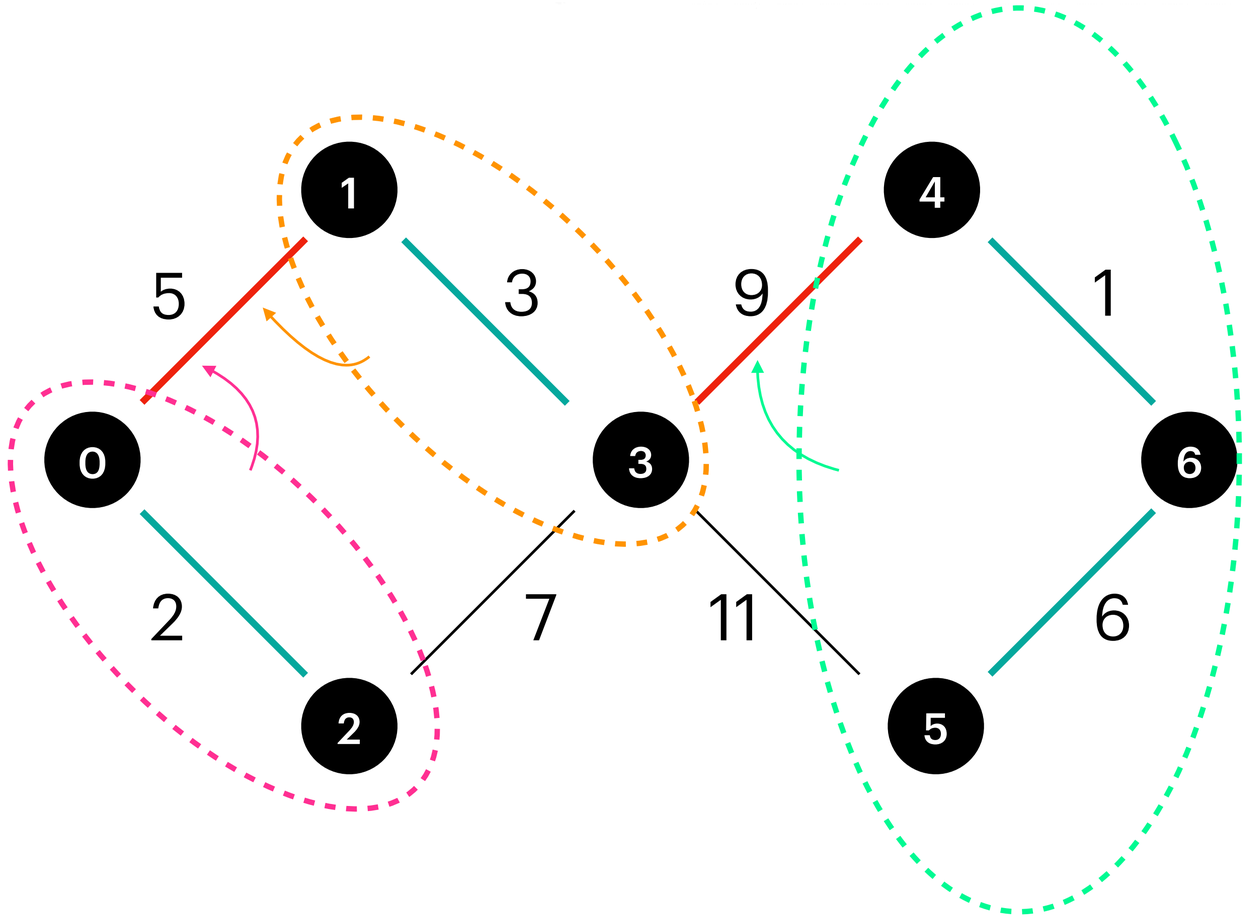
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

F : { {0,2} , {1,3} , {4,6} , {5,6} }

S_1 = {0,2}

S_2 = {1,3}

S_3 = {4,5,6}



MST

Boruvka's Algorithm

F : edges of the MST

Algorithm 8 Boruvka(G)

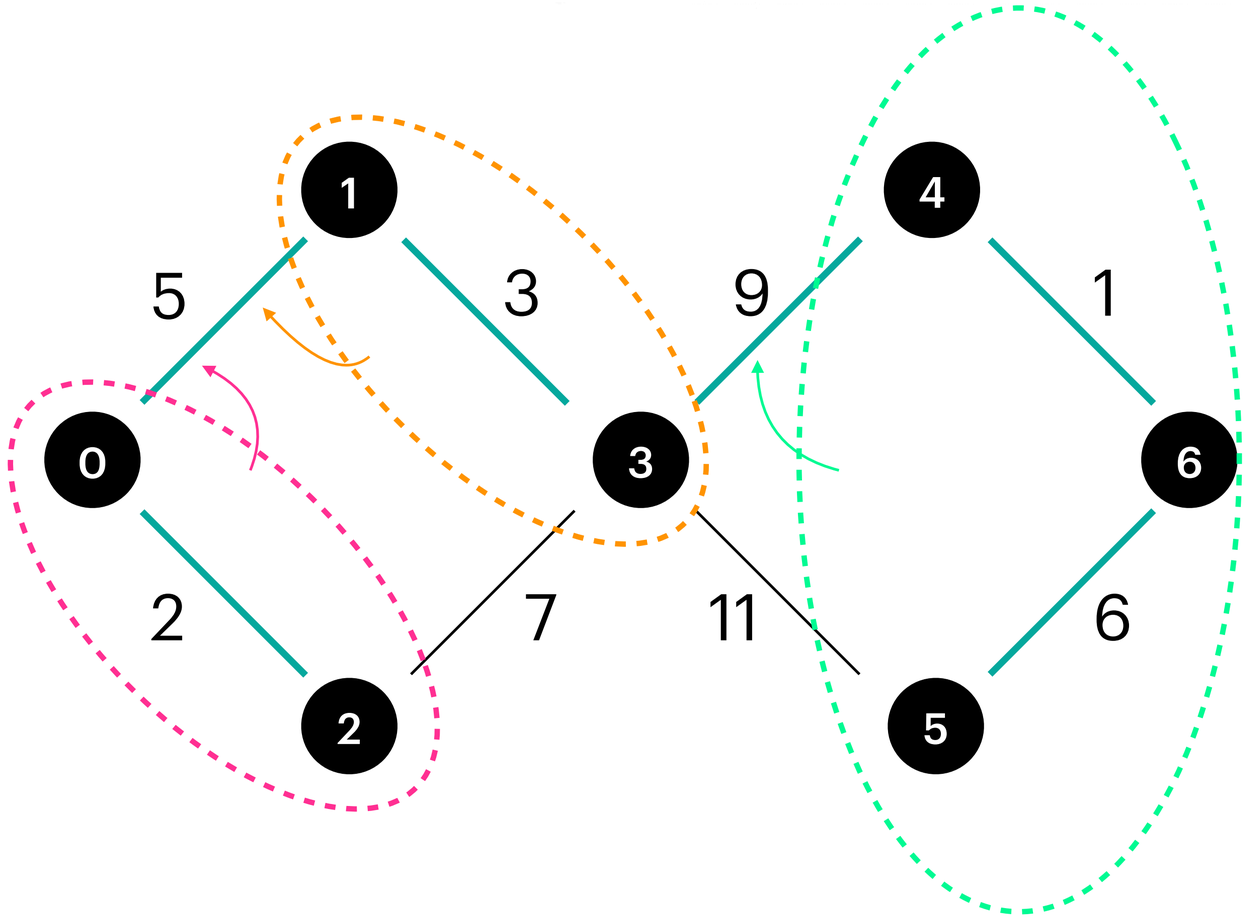
1: $F \leftarrow \varnothing$
2: **while** F nicht Spannbaum **do**
3: $(S_1, \ldots, S_k) \leftarrow$ ZHKs von F
4: $(e_1, \ldots, e_k) \leftarrow$ minimale Kanten an $S_1, \ldots, S_k$
5: $F \leftarrow F \cup \{e_1, \ldots, e_k\}$

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}



MST

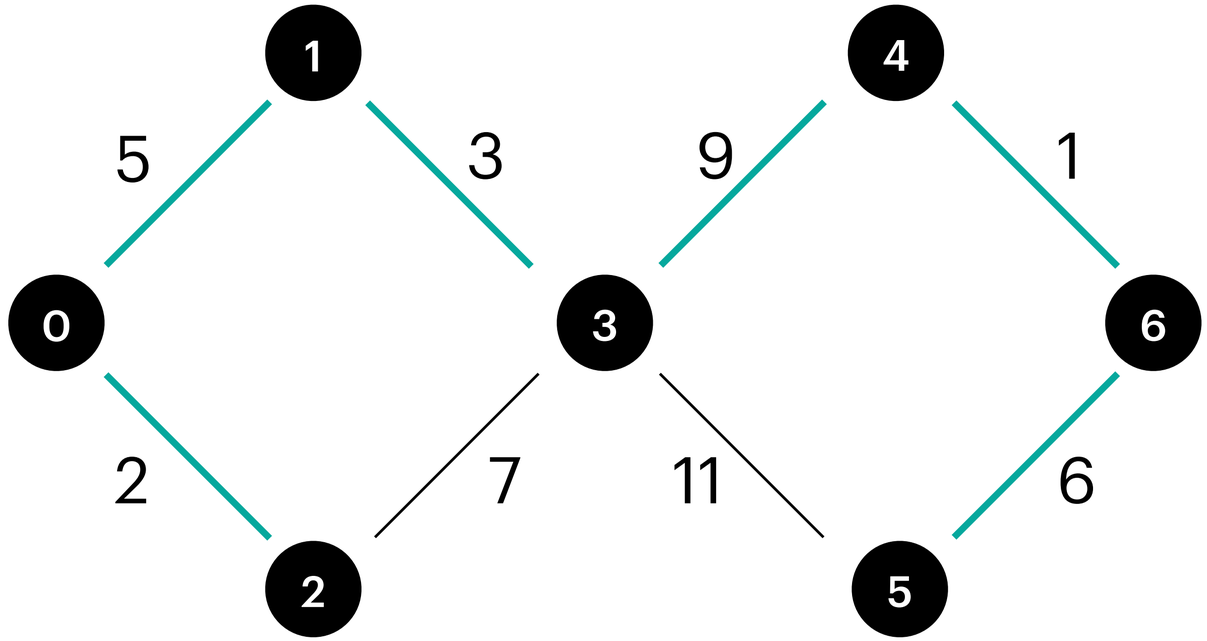
Boruvka's Algorithm

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} }



MST

Boruvka's Algorithm

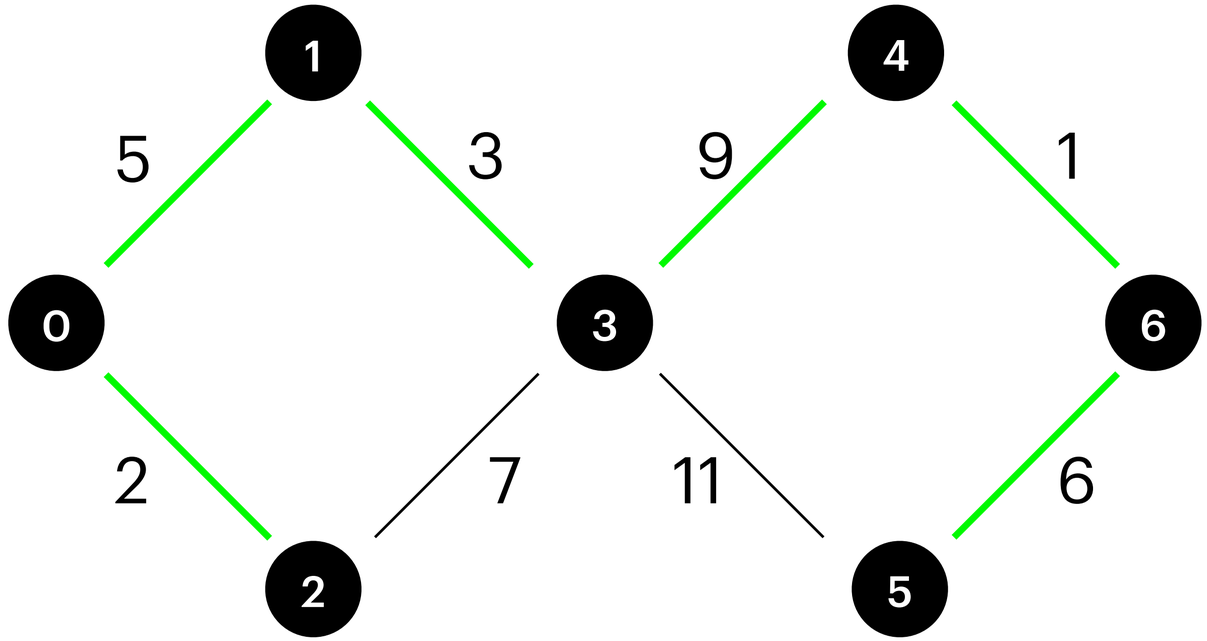
F : edges of the MST

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



MST

Kruskal's Algorithm

rep[v] : unique representative of ConComp(v)

members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 11 Union-Find(G)

Implementierung:

MAKE(V): $\text{rep}[v] \leftarrow v \quad \forall v \in V$ members[v] <- {v} ▷ $\mathcal{O}(n)$

SAME(u,v): teste ob $\text{rep}[u] = \text{rep}[v]$ ▷ $\mathcal{O}(1)$

UNION(u,v): ▷ $\mathcal{O}(|\text{ZHK}(u)|)$

for $x \in \text{members}[\text{rep}[u]]$ **do**

    $\text{rep}[x] \leftarrow \text{rep}[v]$

    $\text{members}[\text{rep}[v]] \leftarrow \text{members}[\text{rep}[v]] \cup \{x\}$

MST

Kruskal's Algorithm

Runtime : O ((|V| + |E|) * log n)

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$ edges of the MST
2: $UF \leftarrow$ MAKE(V) UF : Union Find
3: SORT(E) sort the edges in increasing order
4: **for** $uv \in E$, aufsteigend sortiert **do**
5: **if** SAME(u,v) = false **then** if its not in the same concomp
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])

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

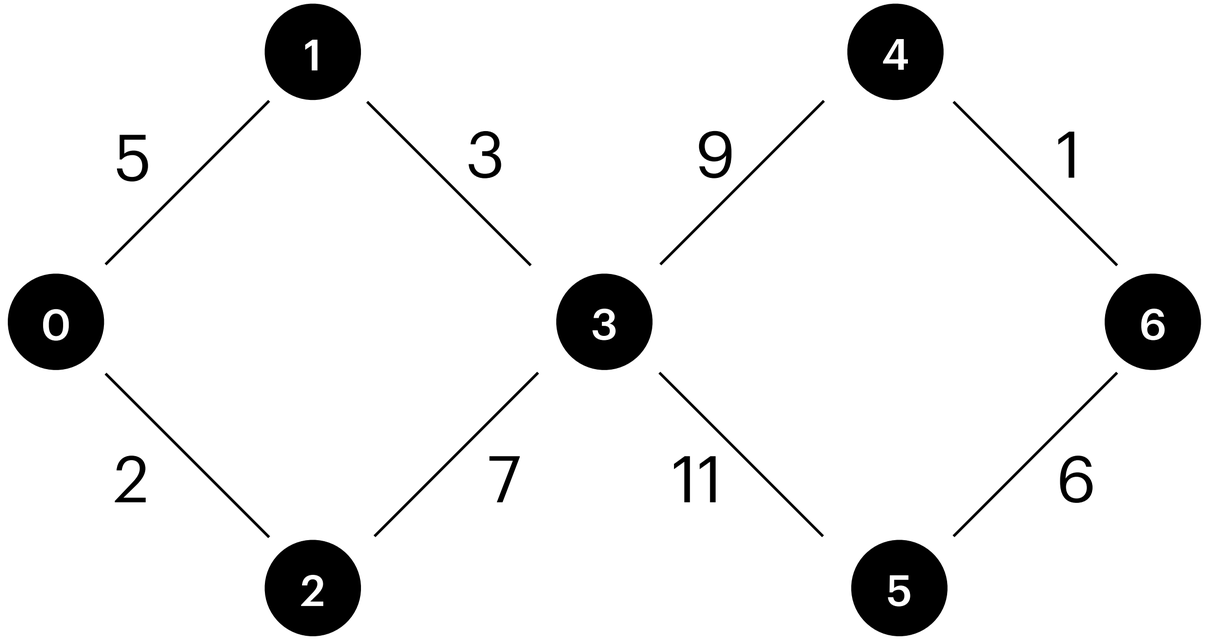
F :

rep[] :

0	1	2	3	4	5	6

members[] :

0	
1	
2	
3	
4	
5	
6	



MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

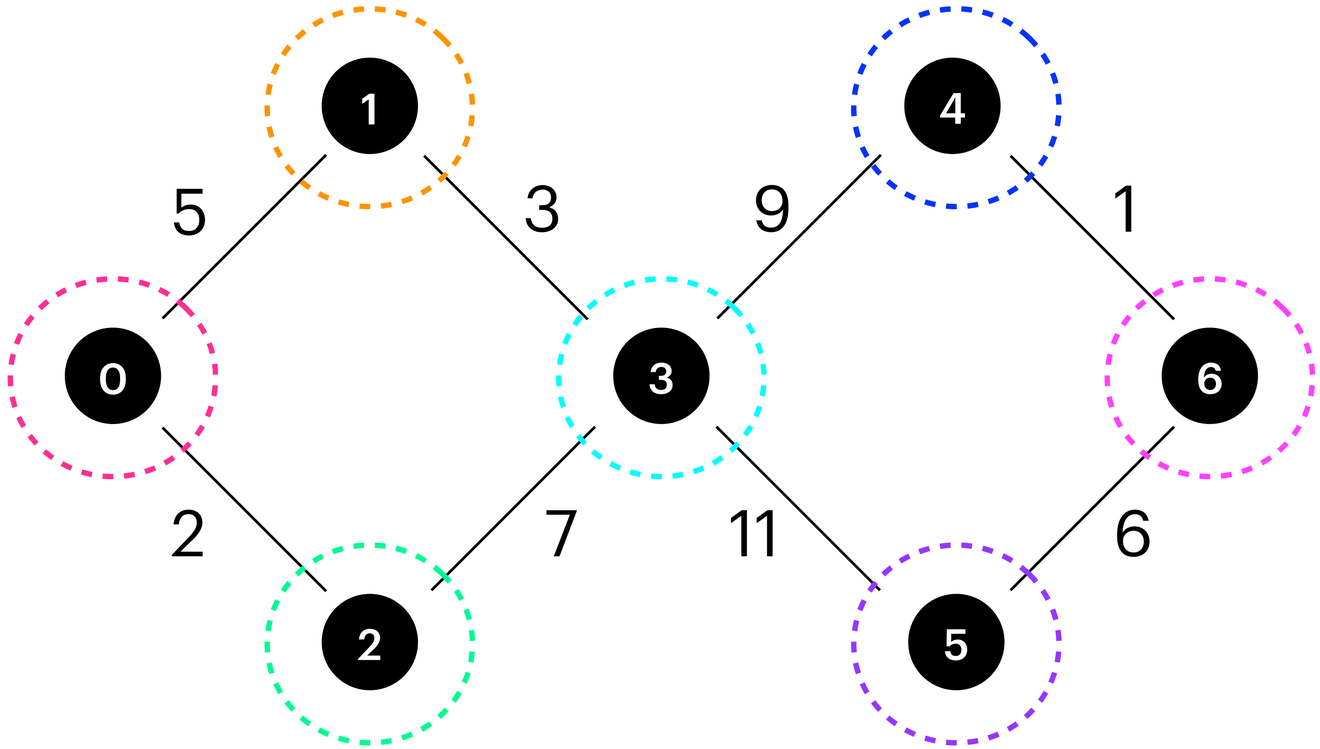
F : ∅

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} }

MST

Kruskal's Algorithm

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: $\text{members}[\text{rep}[v]] \leftarrow \text{members}[\text{rep}[v]] \cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
2: $UF \leftarrow \text{MAKE}(V)$
3: $\text{SORT}(E)$
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)

F : edges of the MST

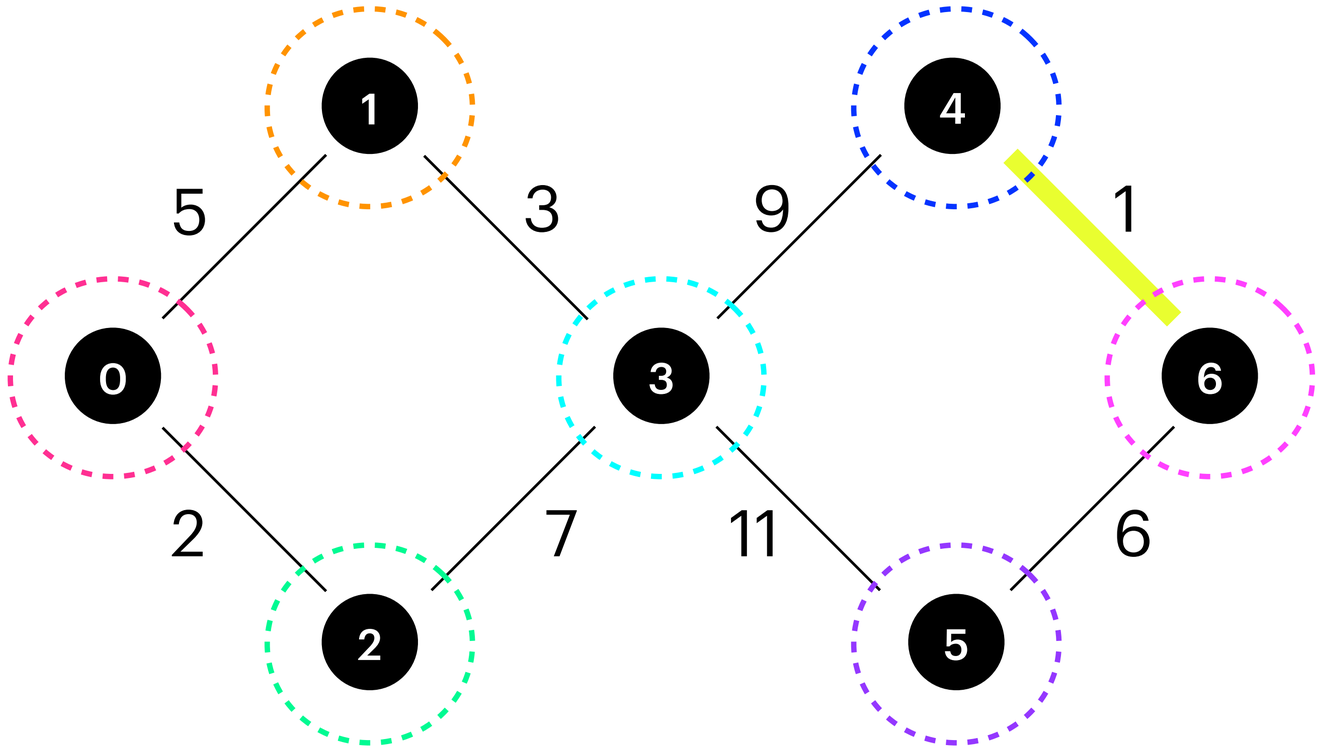
F : ∅

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

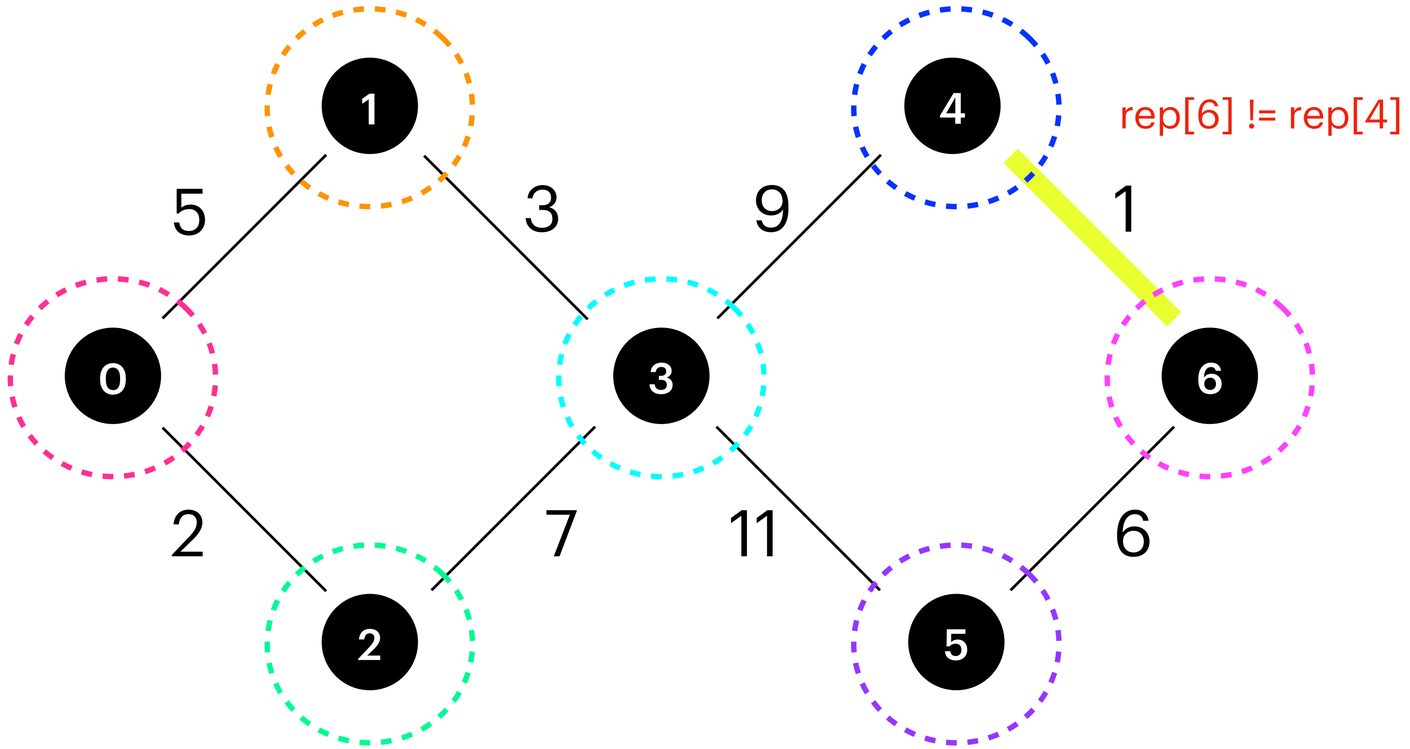
F : ∅

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

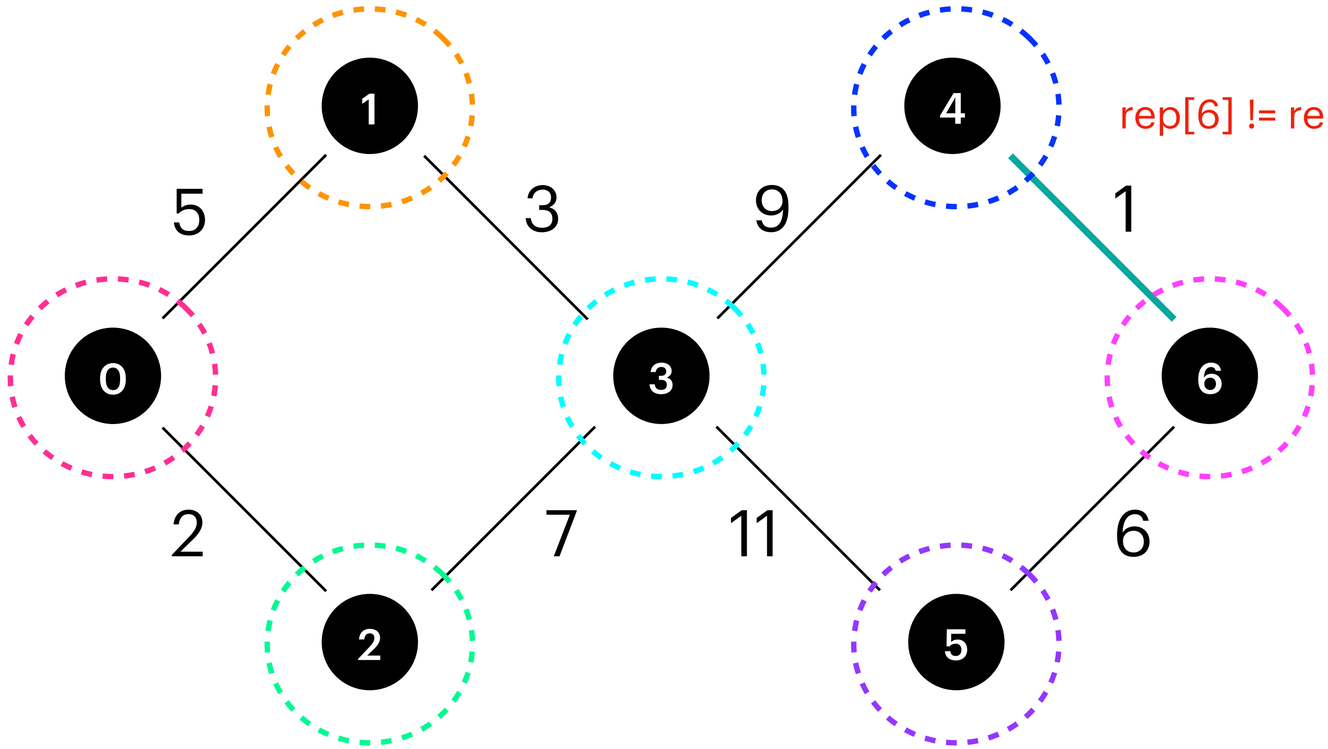
F : { {6,4} }

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 }



rep[6] != rep[4]

SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

F : { {6,4} }

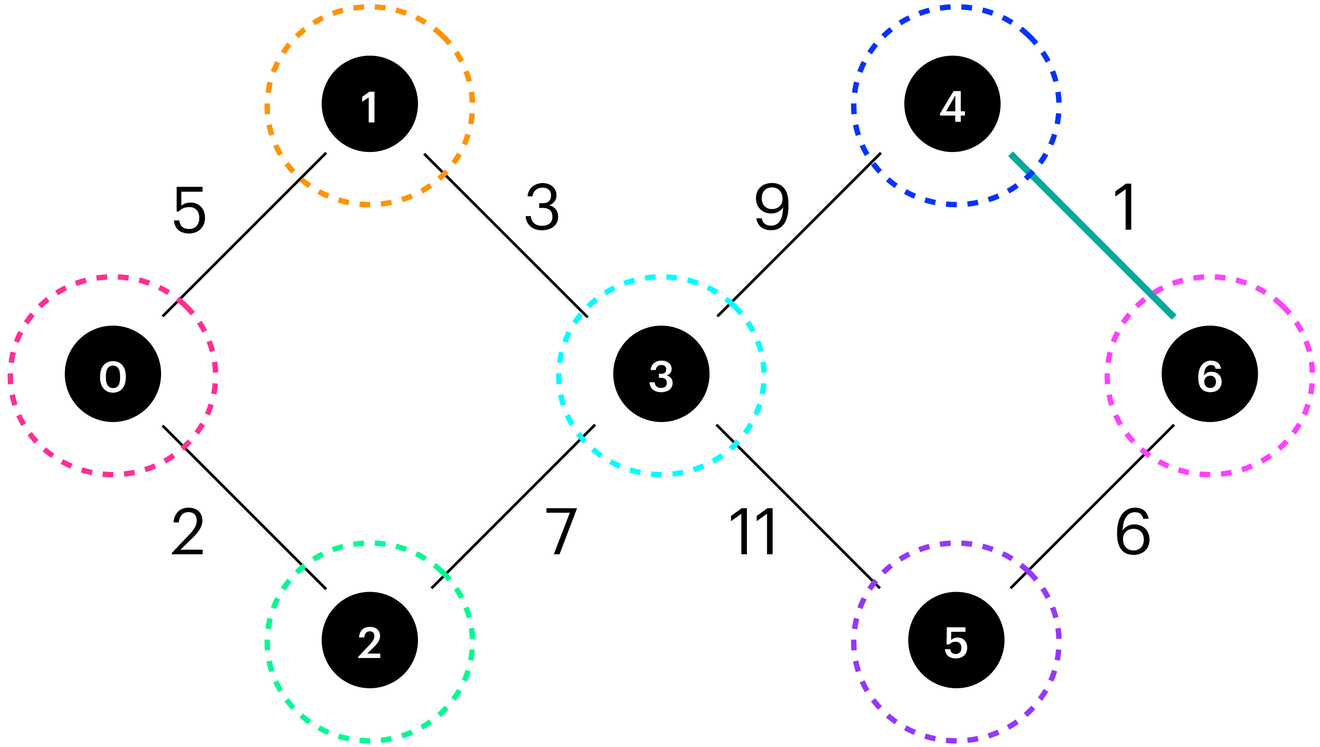
UNION(6,4)

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

F : { {6,4} }

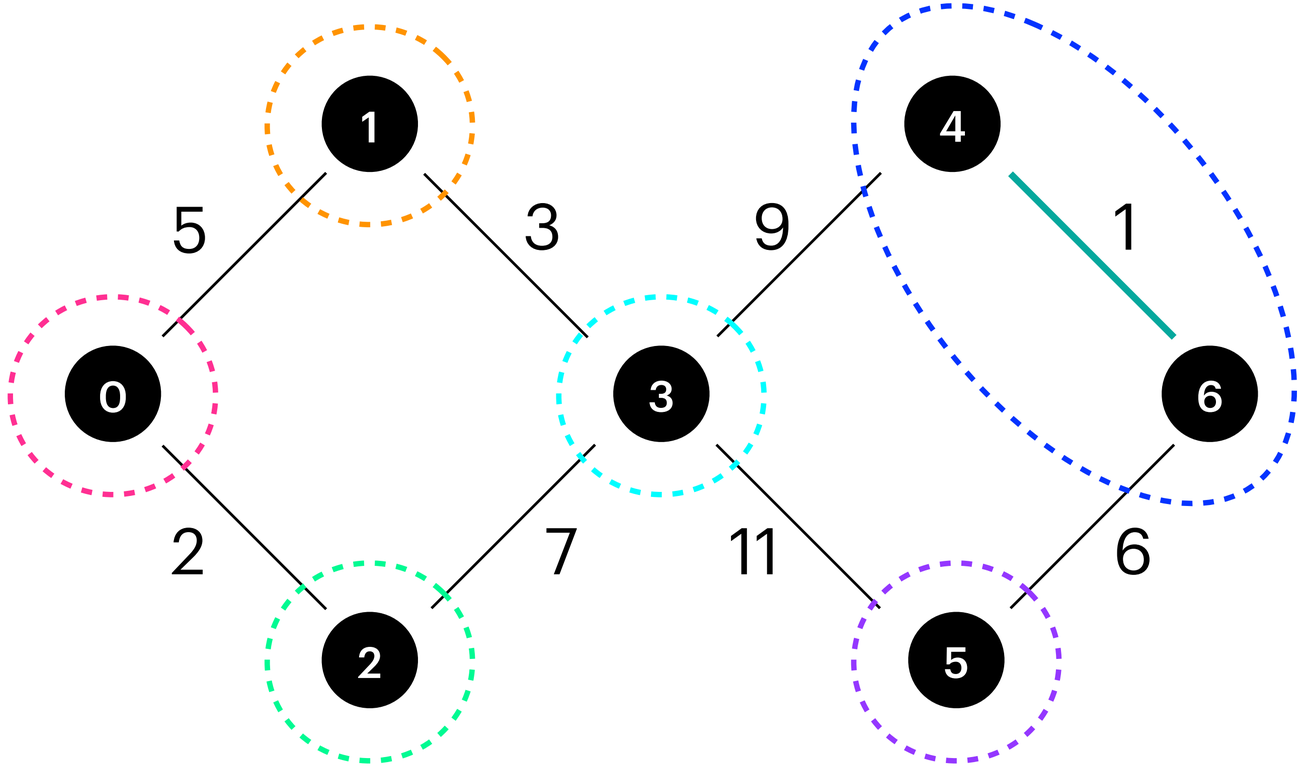
UNION(6,4)

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

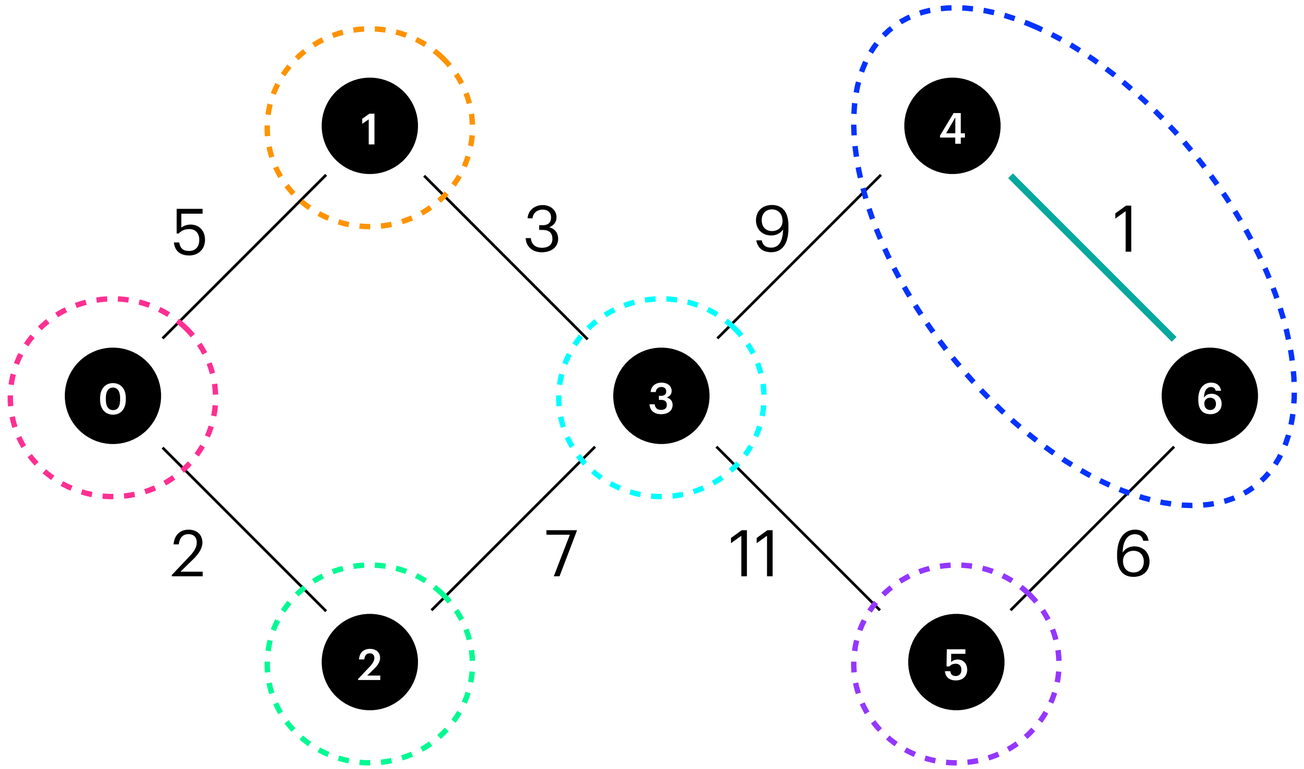
F : { {6,4} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

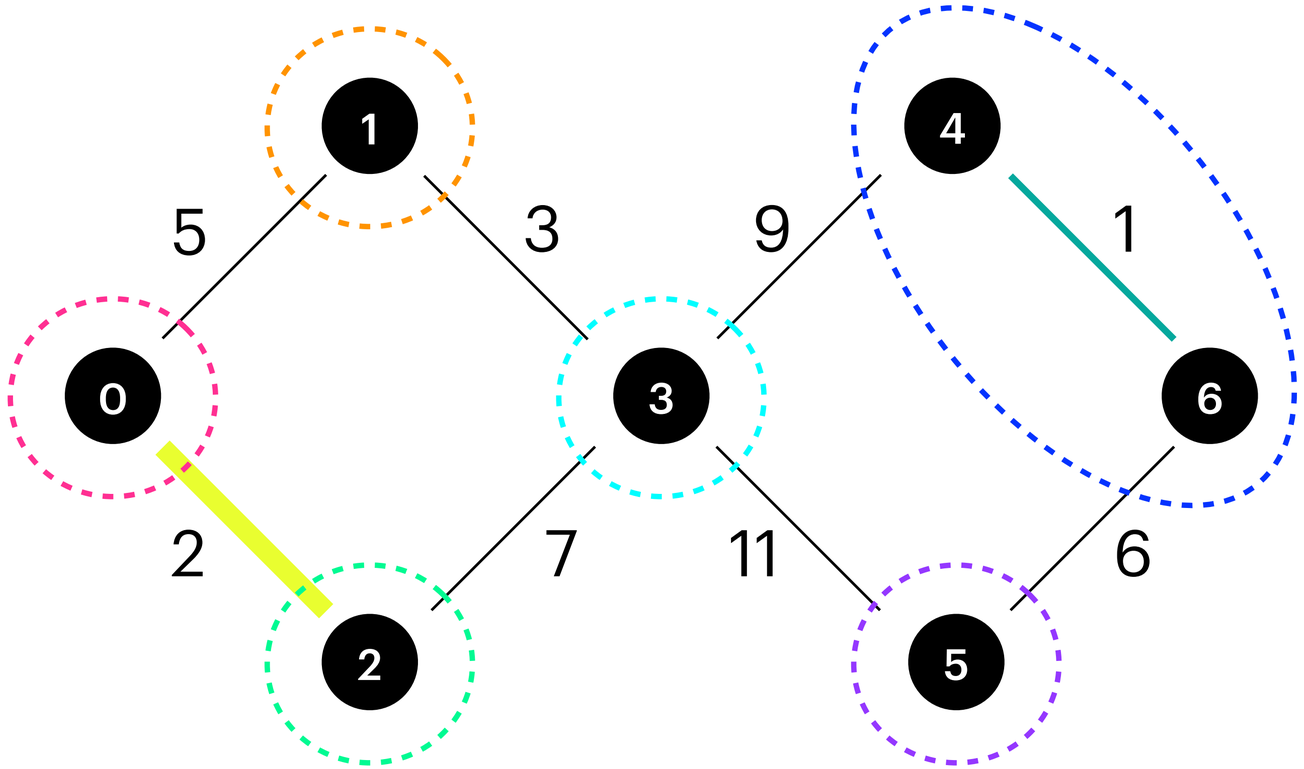
F : { {6,4} }

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} }

MST

Kruskal's Algorithm

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: $\text{members}[\text{rep}[v]] \leftarrow \text{members}[\text{rep}[v]] \cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
2: $UF \leftarrow \text{MAKE}(V)$
3: $\text{SORT}(E)$
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)

F : edges of the MST

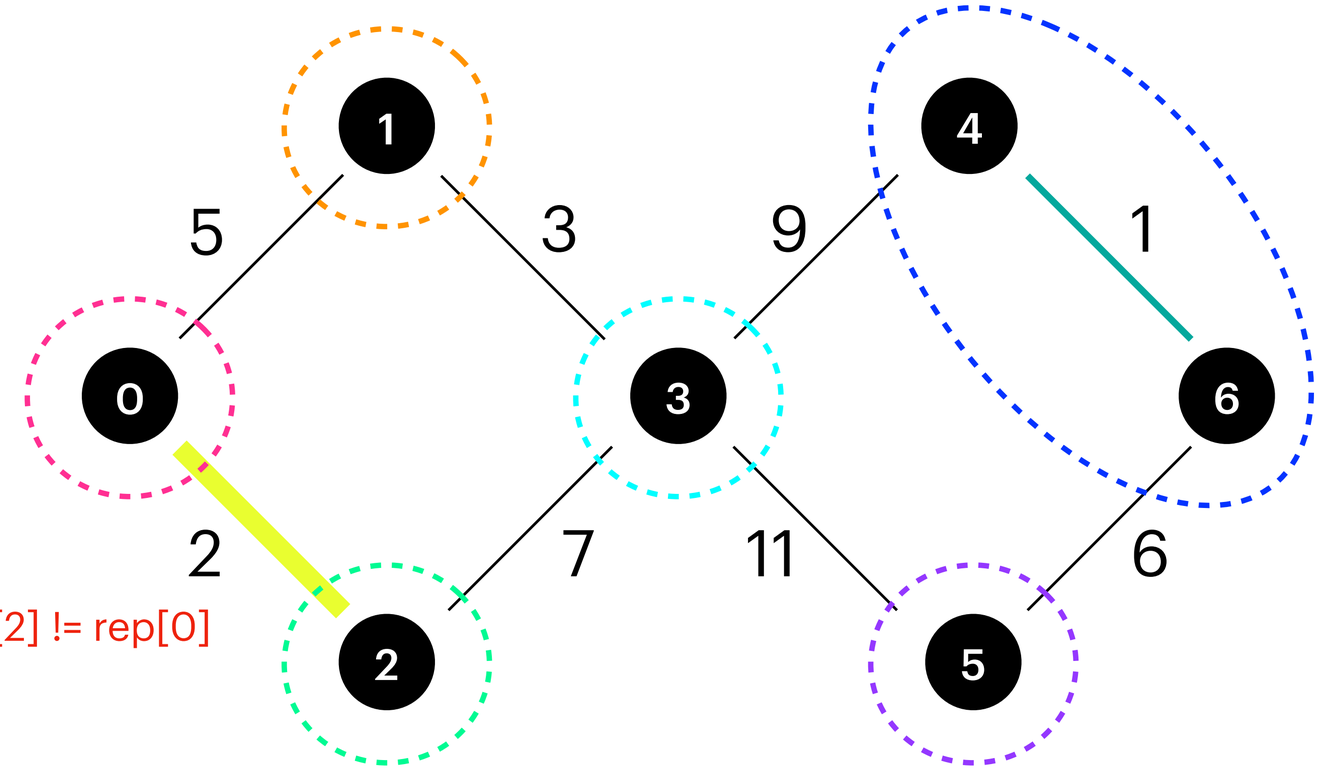
F : { {6,4} }

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 }



rep[2] != rep[0]

SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

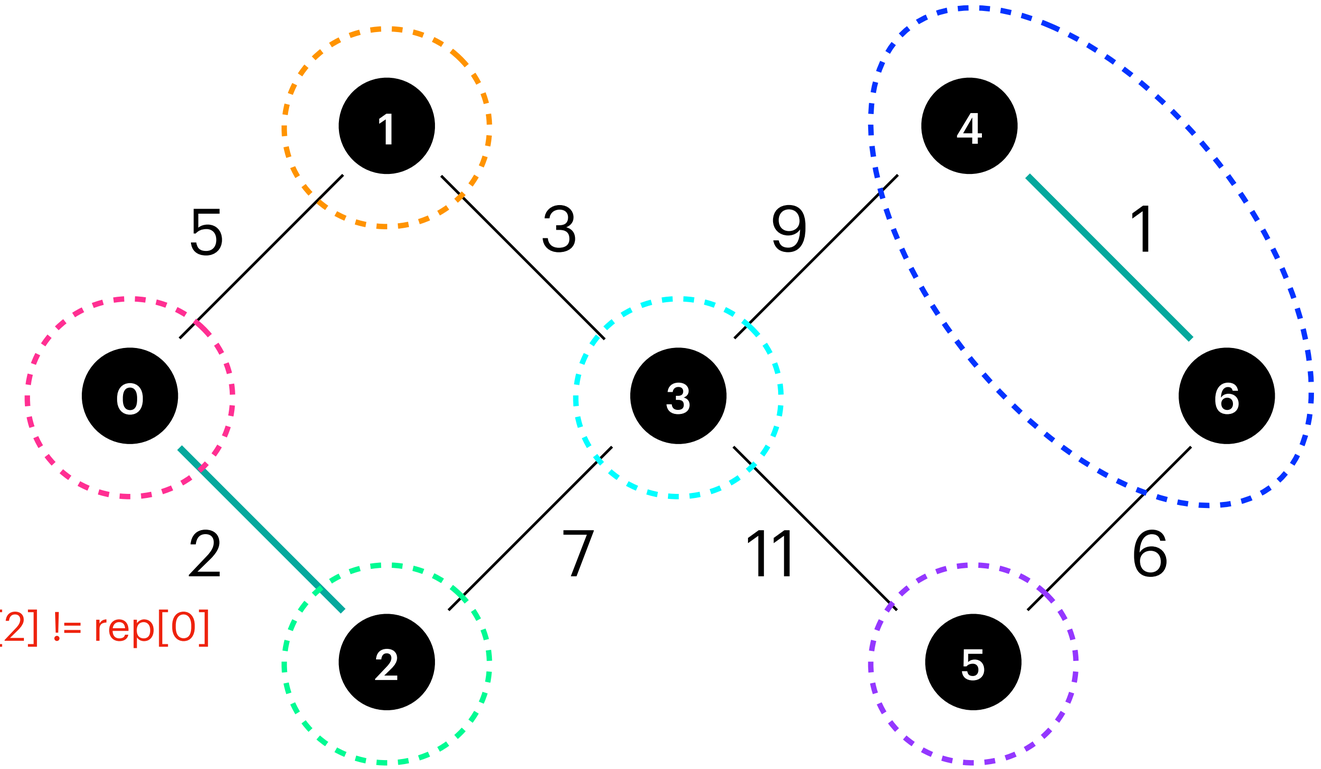
F : { {6,4} , {2,0} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

F : { {6,4} , {2,0} }

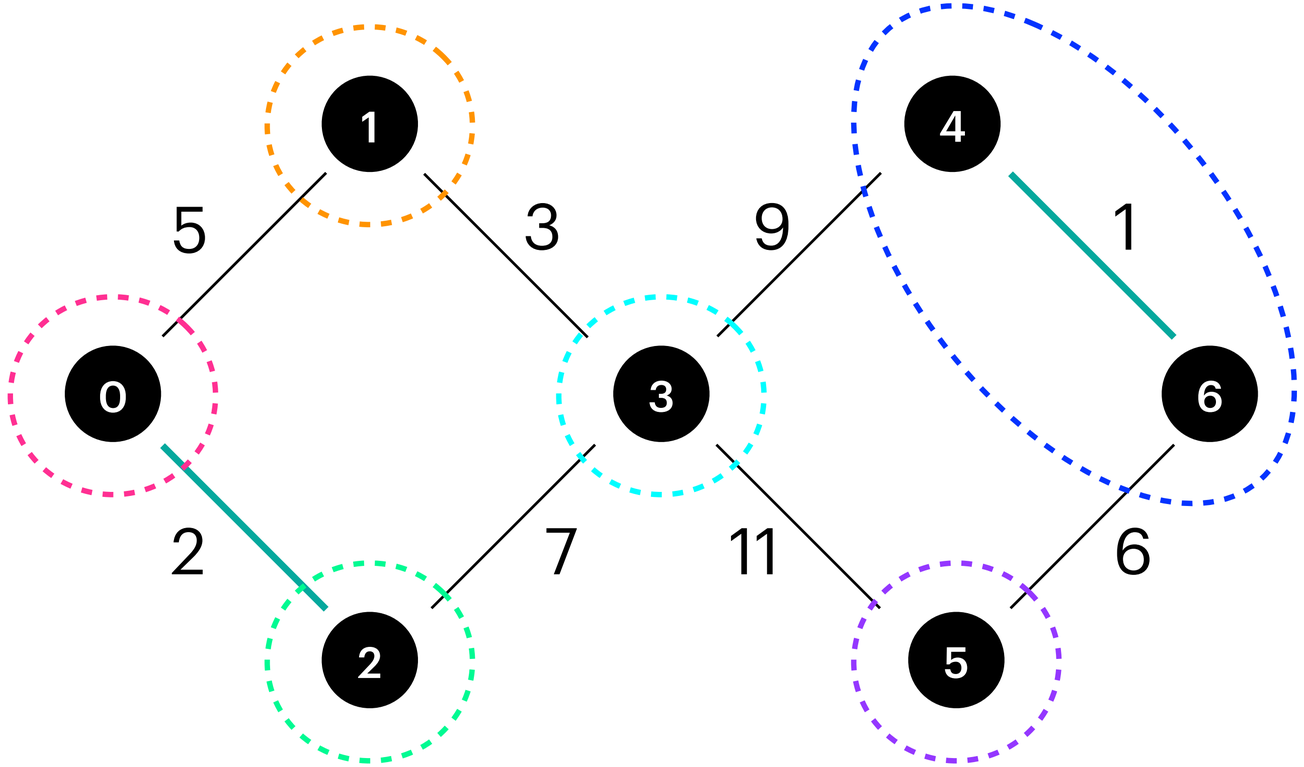
UNION(2,0)

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

F : { {6,4} , {2,0} }

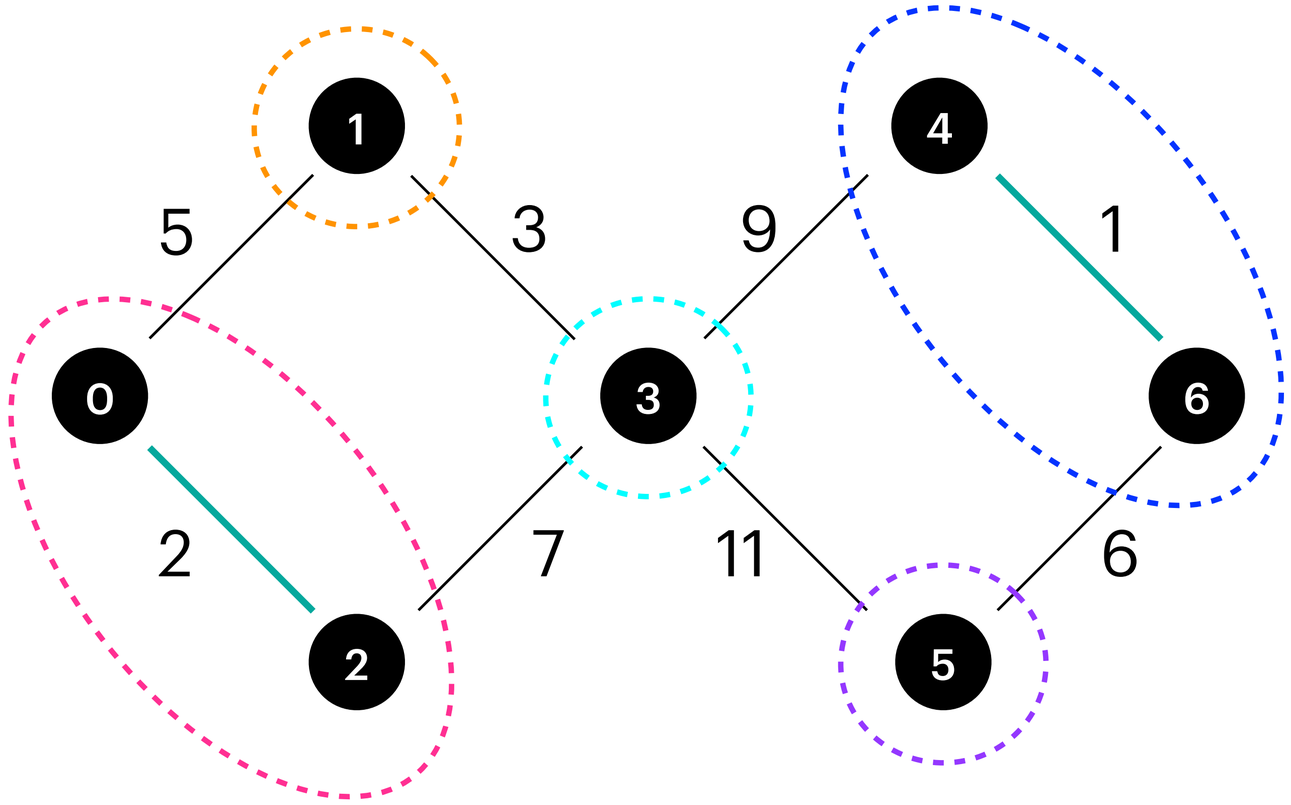
UNION(2,0)

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

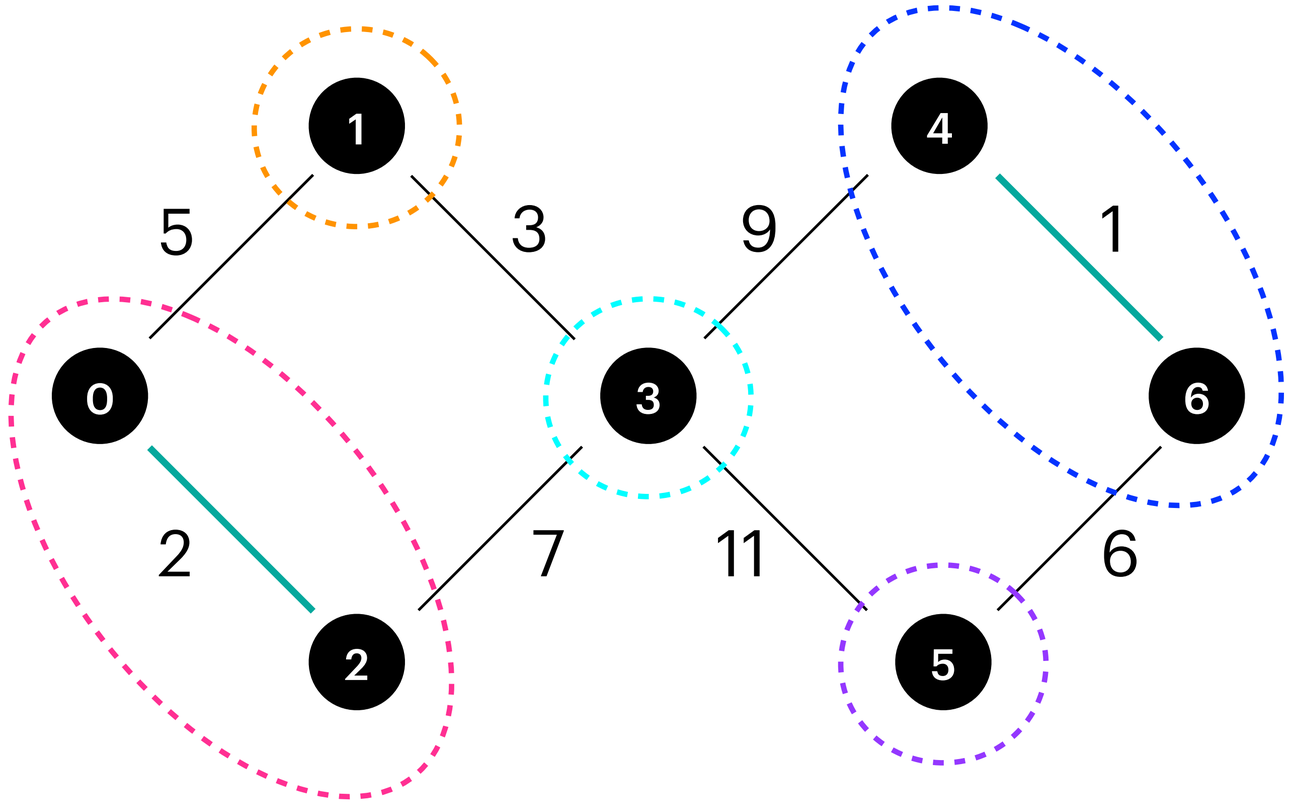
F : { {6,4} , {2,0} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

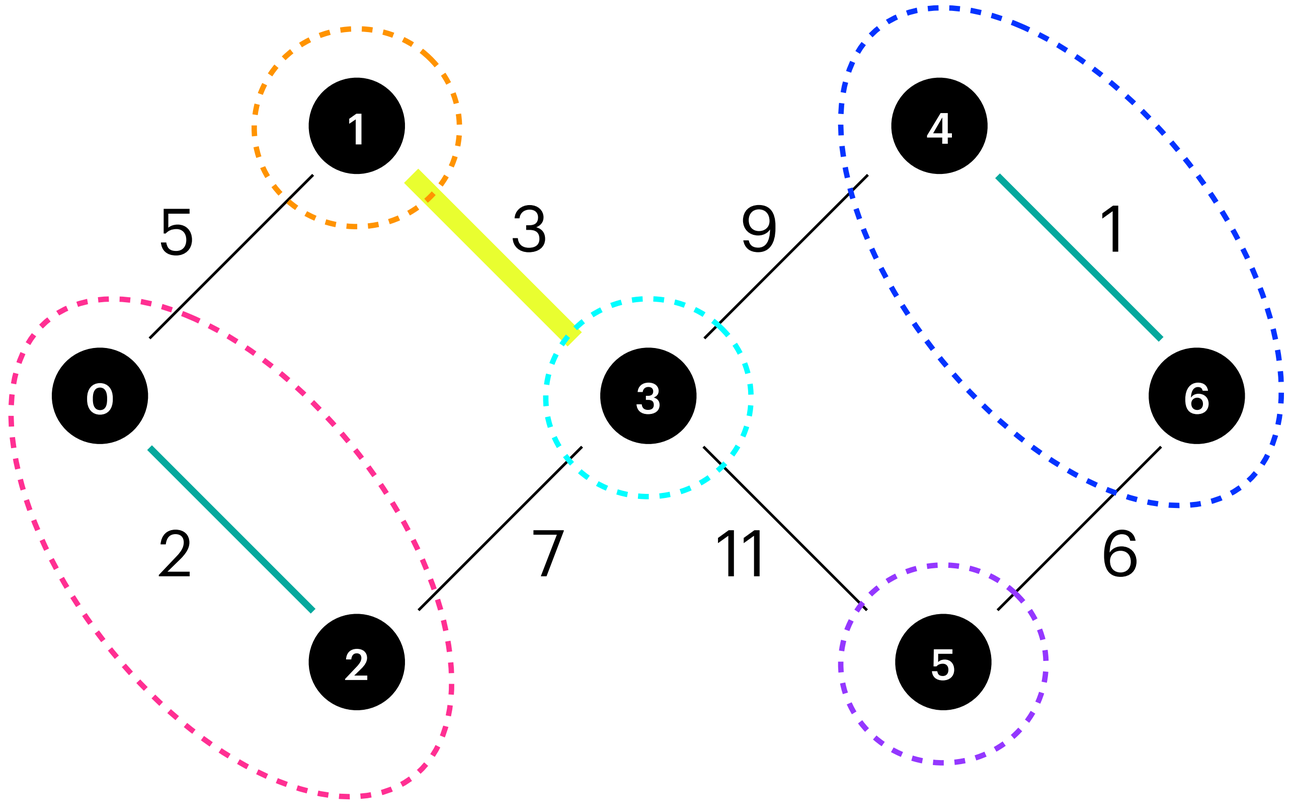
F : { {6,4} , {2,0} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

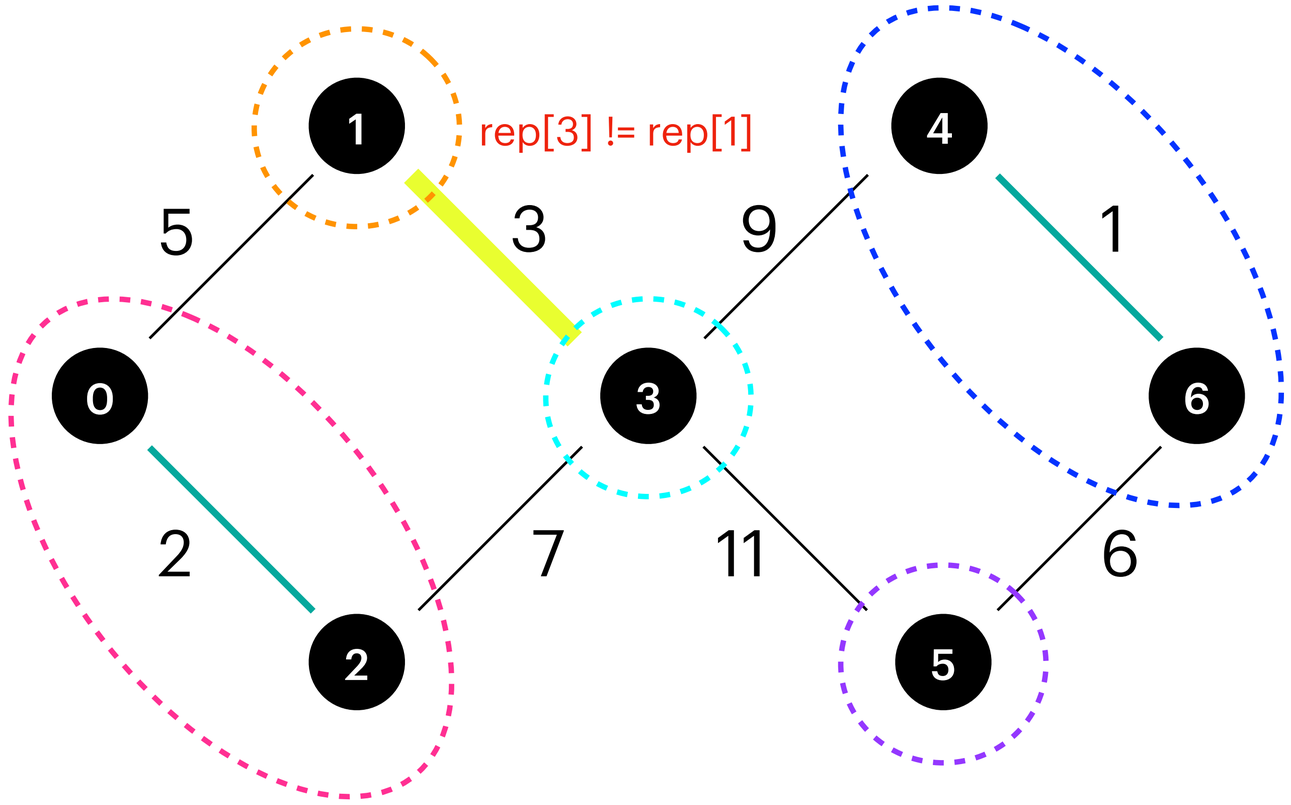
F : { {6,4} , {2,0} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

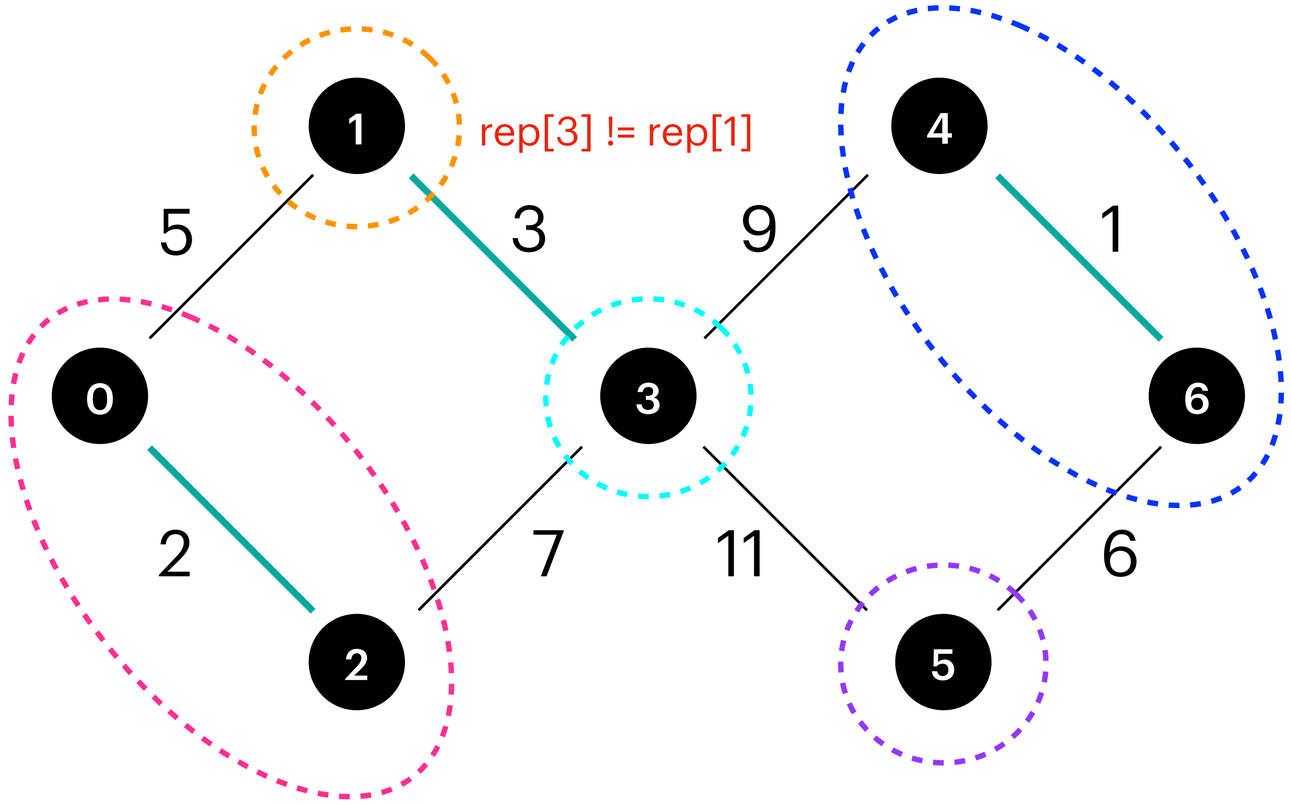
F : { {6,4} , {2,0} , {3,1} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

F : { {6,4} , {2,0} , {3,1} }

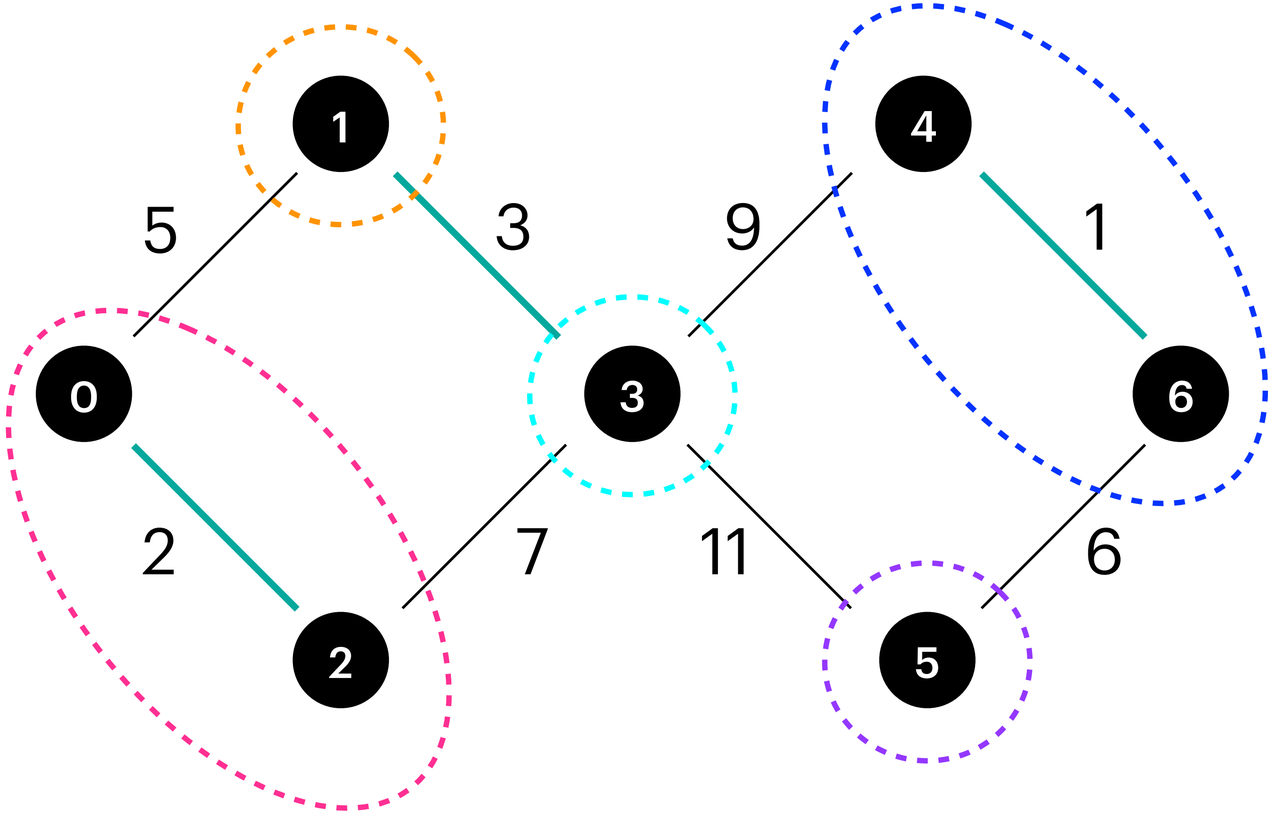
UNION(3,1)

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

F : { {6,4} , {2,0} , {3,1} }

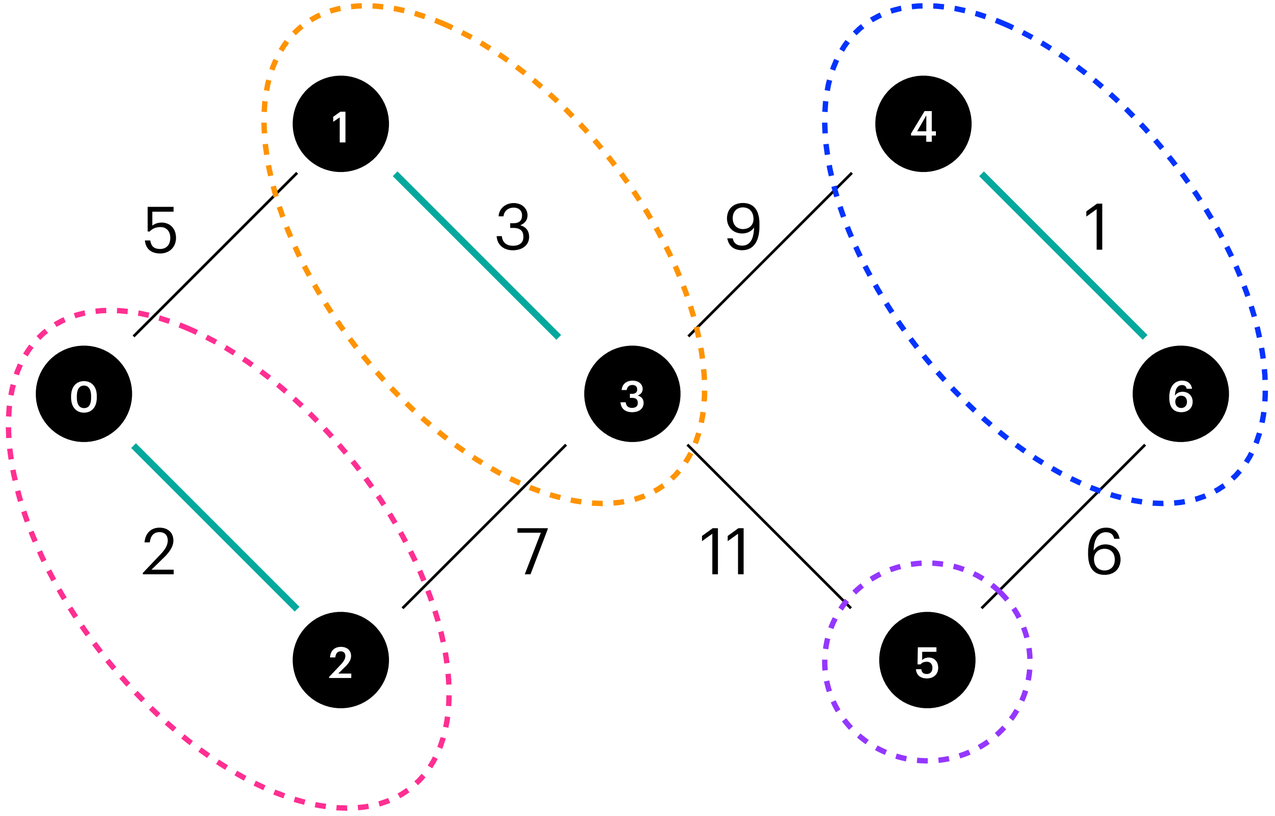
UNION(3,1)

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

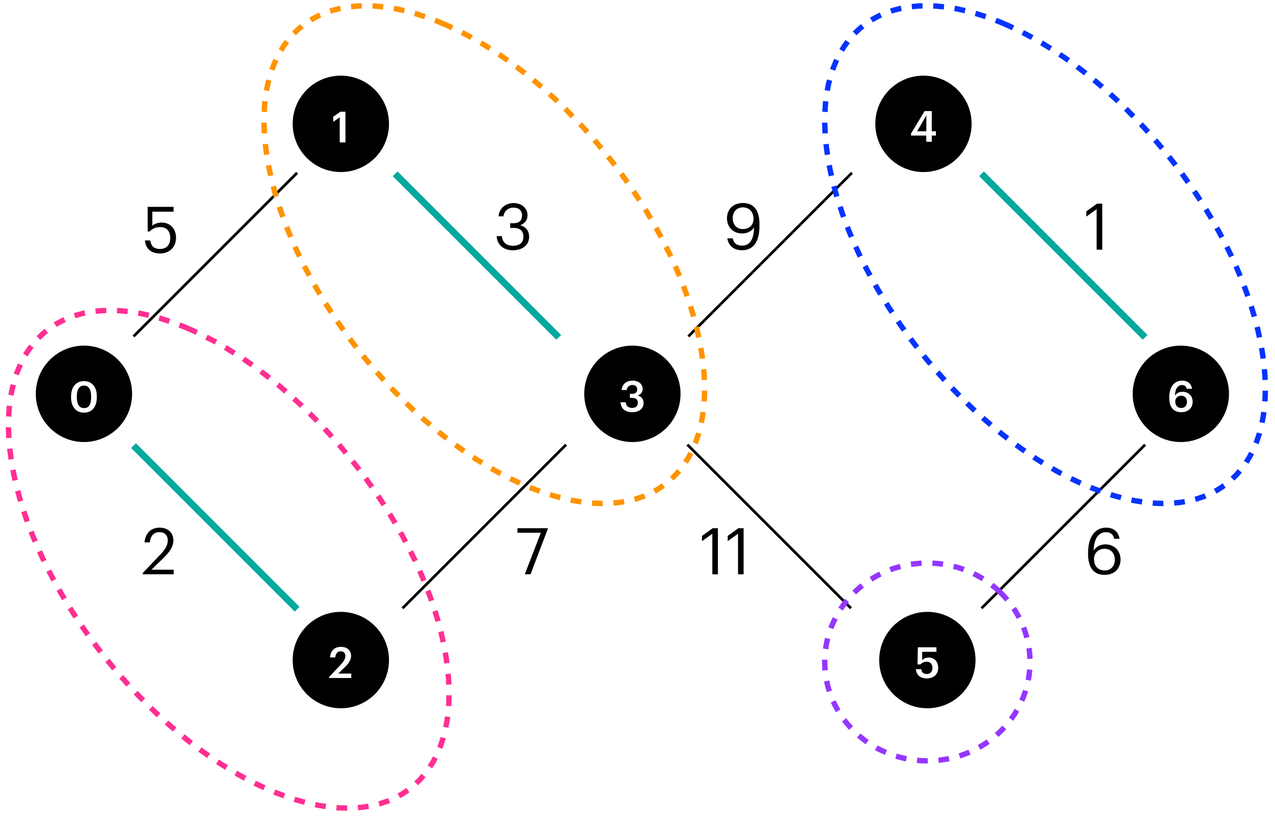
F : { {6,4} , {2,0} , {3,1} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

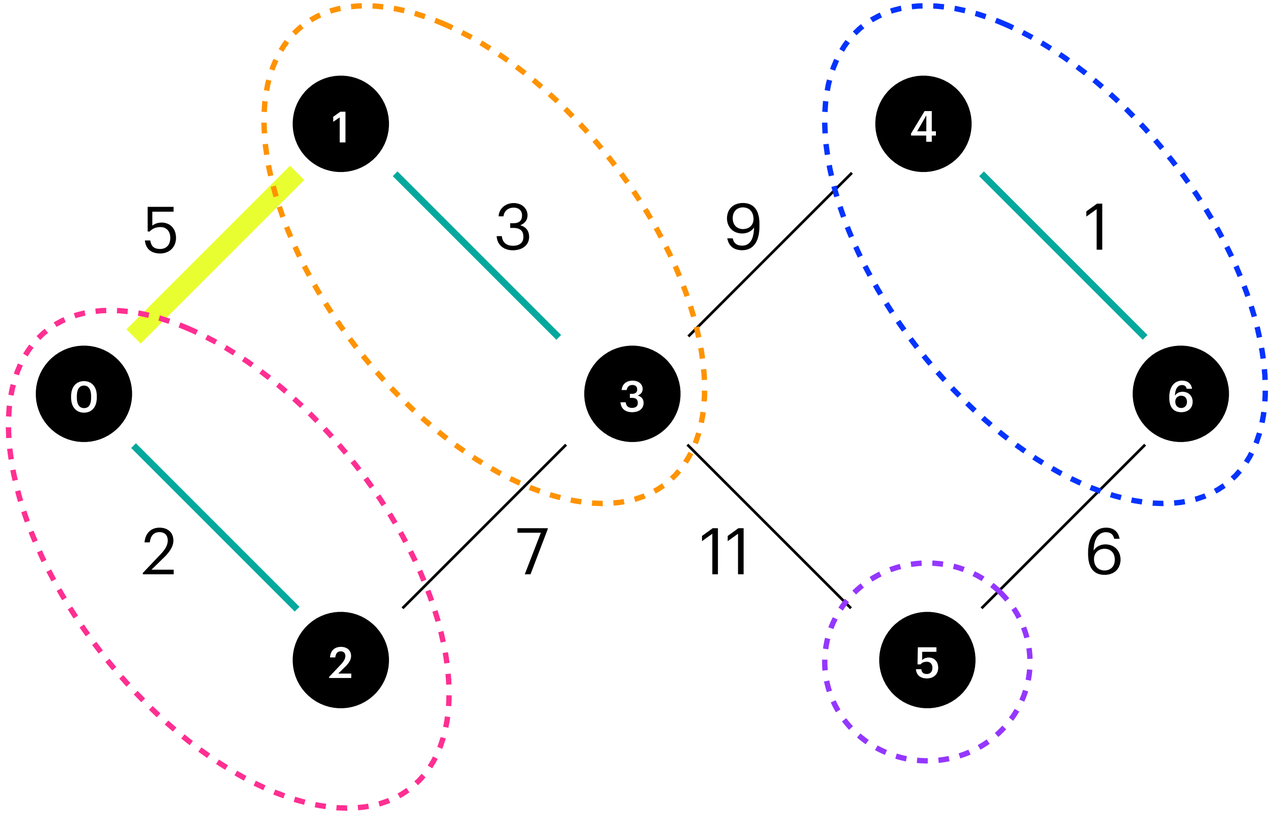
F : { {6,4} , {2,0} , {3,1} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

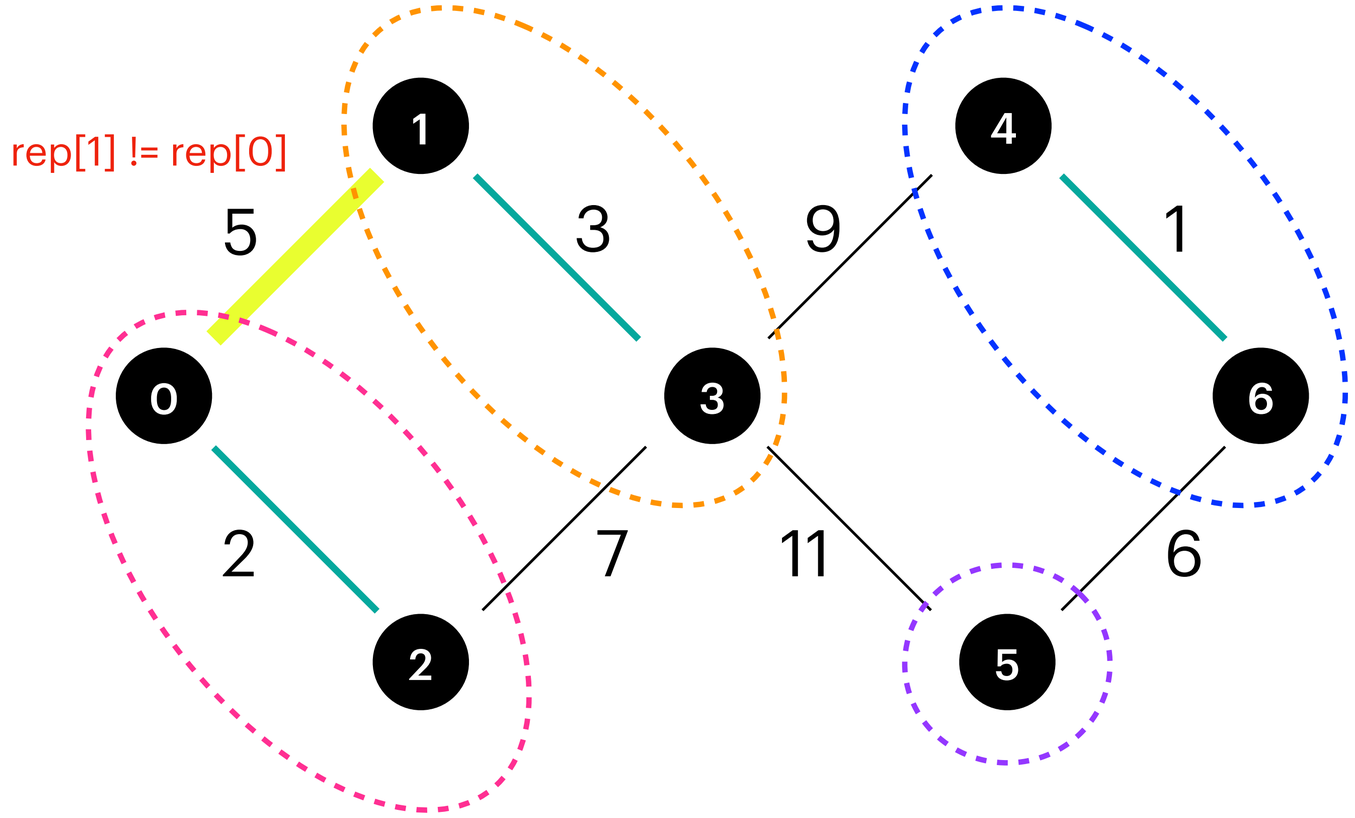
F : { {6,4} , {2,0} , {3,1} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

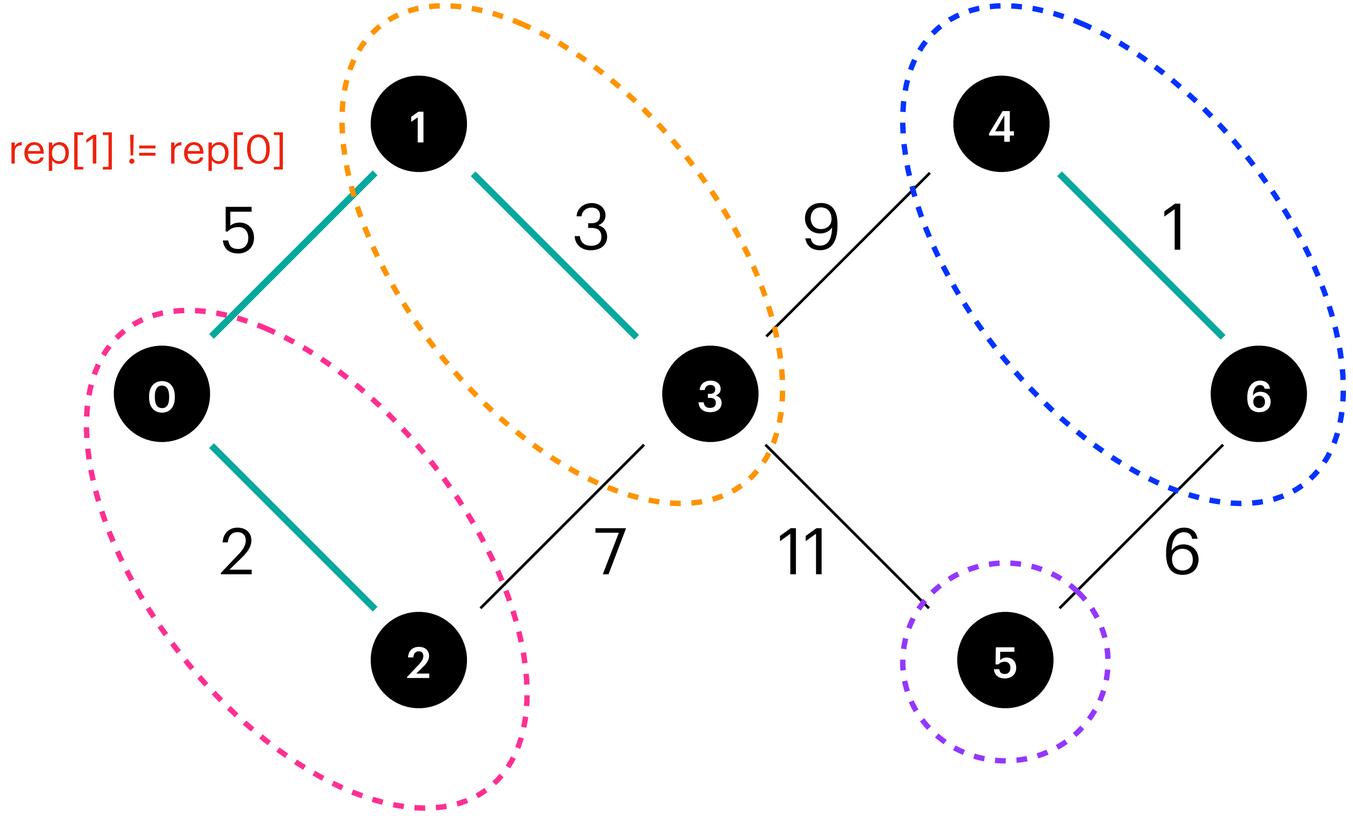
F : { {6,4} , {2,0} , {3,1} , {1,0} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

F : { {6,4} , {2,0} , {3,1} , {1,0} }

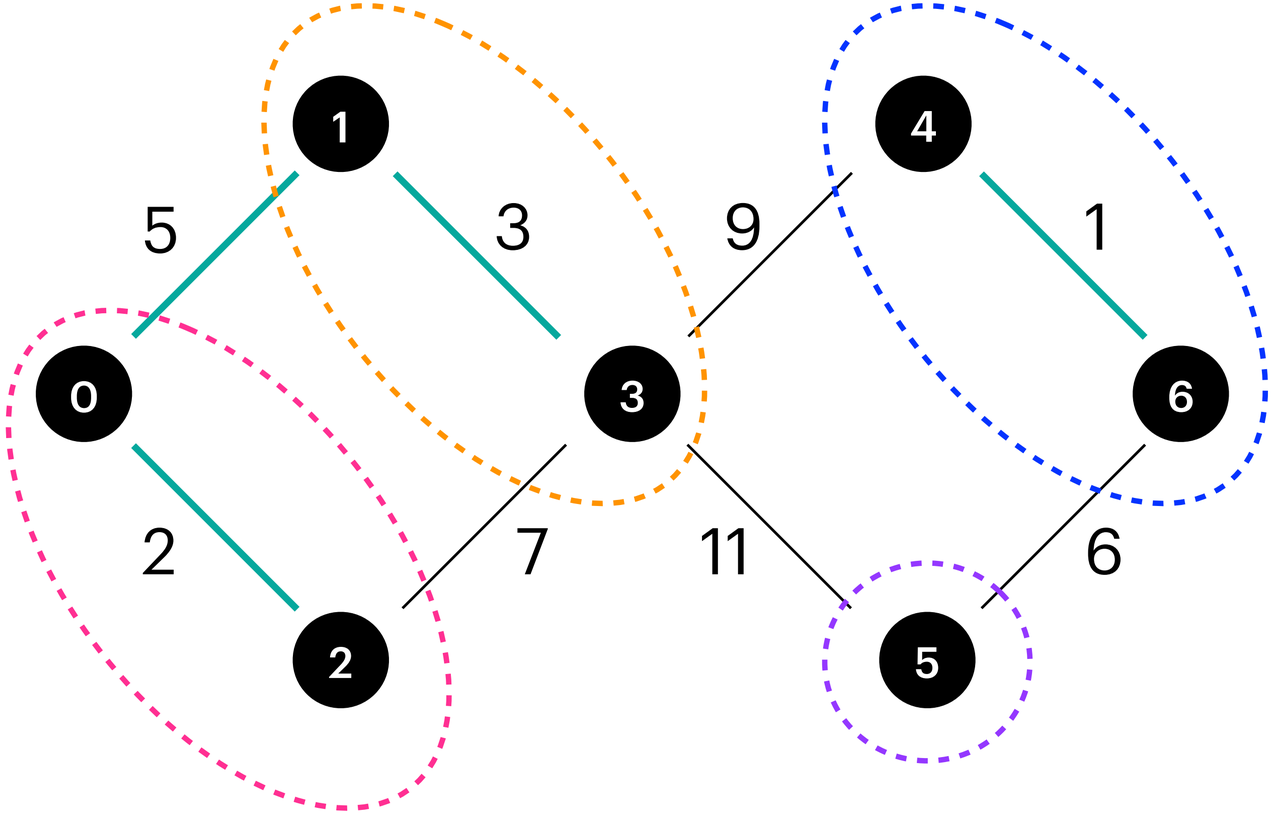
UNION(1,0)

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

F : { {6,4} , {2,0} , {3,1} , {1,0} }

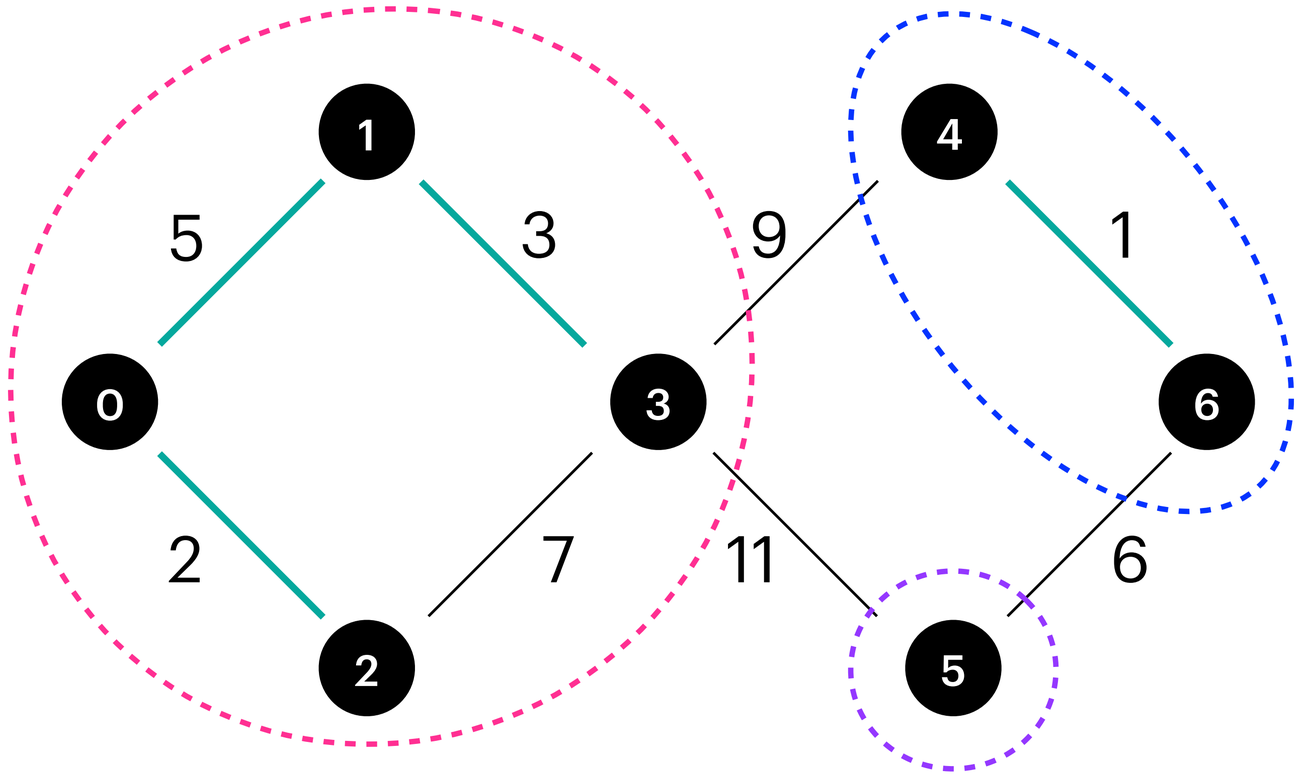
UNION(1,0)

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

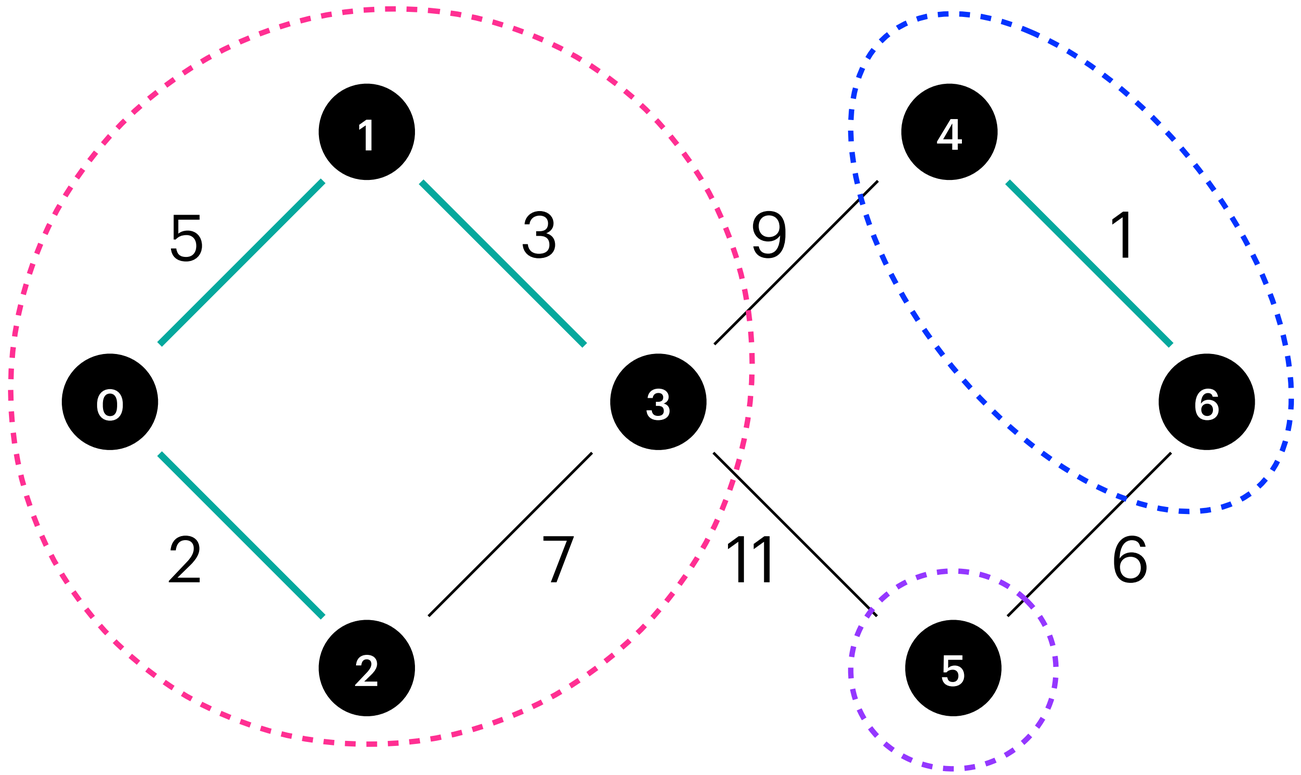
F : { {6,4} , {2,0} , {3,1} , {1,0} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

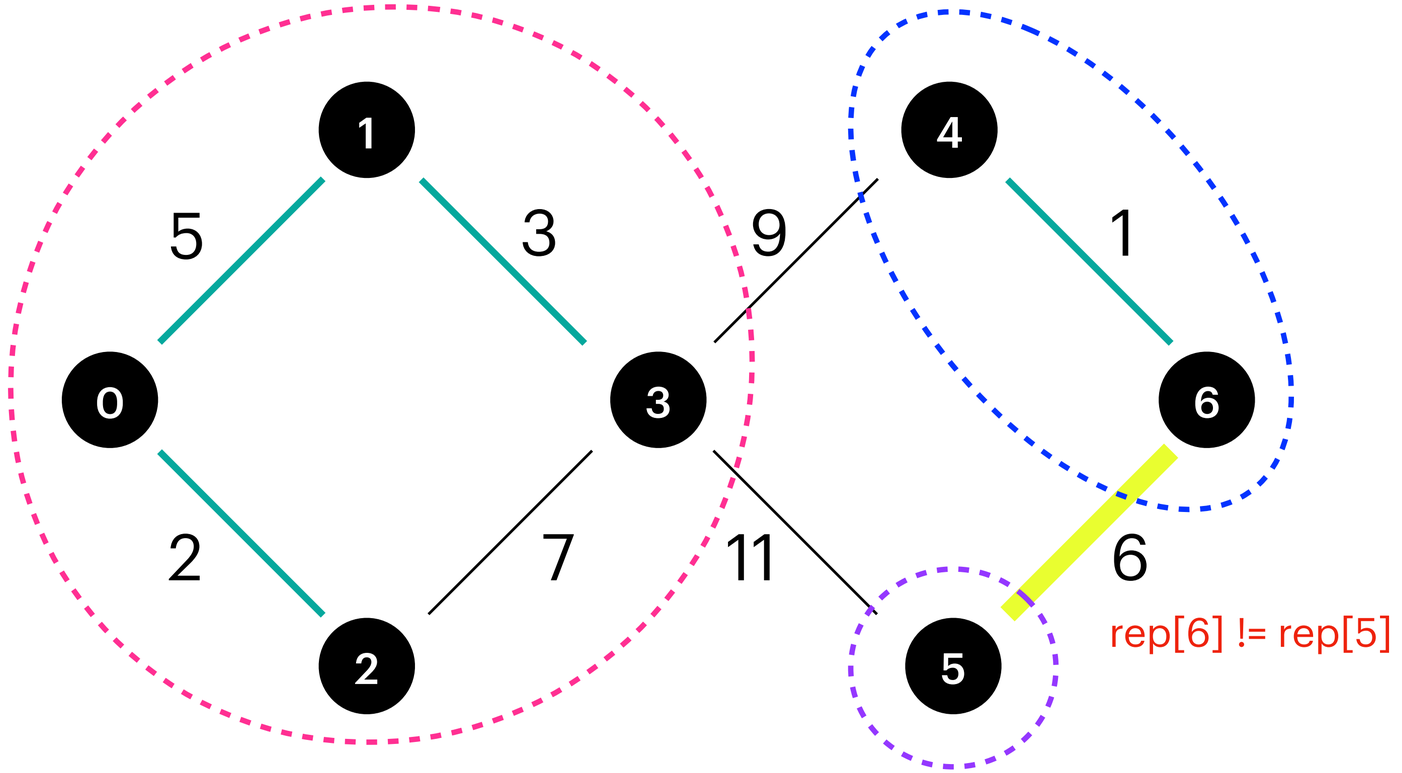
F : { {6,4} , {2,0} , {3,1} , {1,0} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

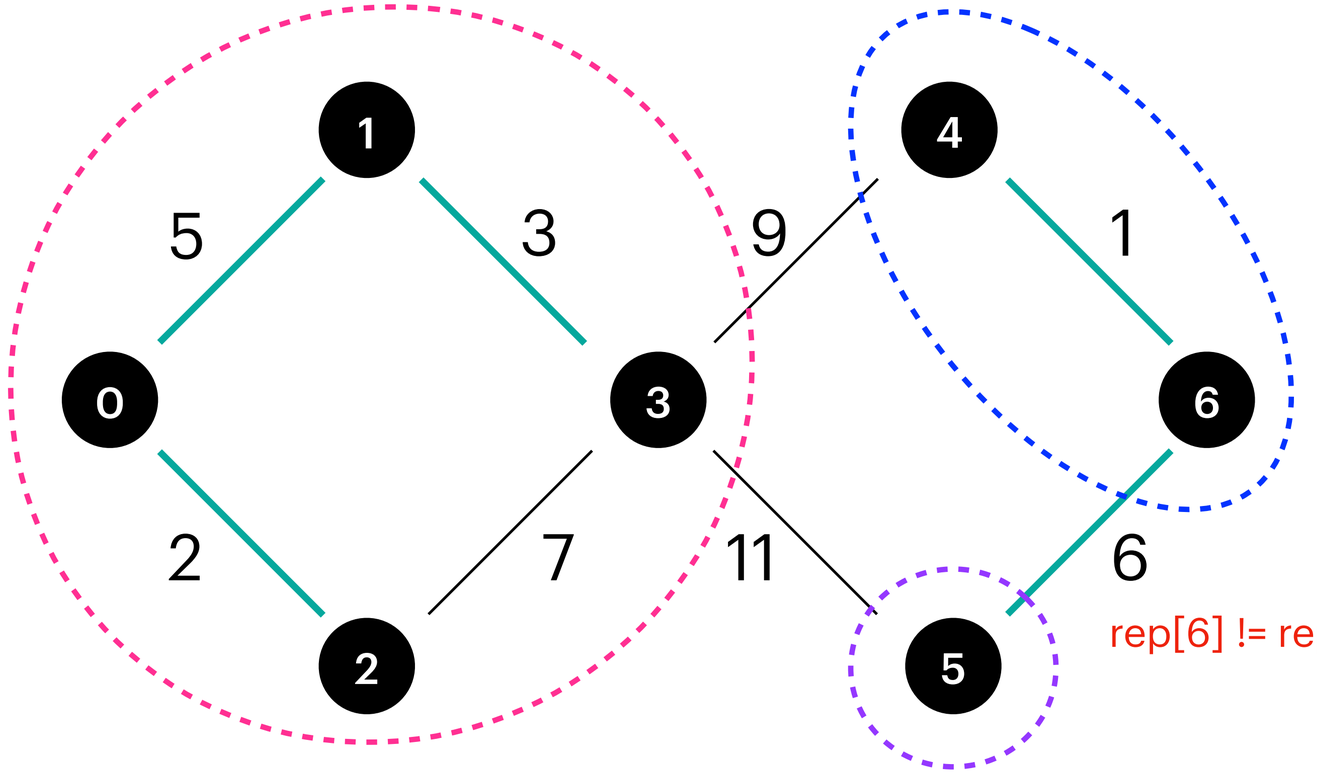
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

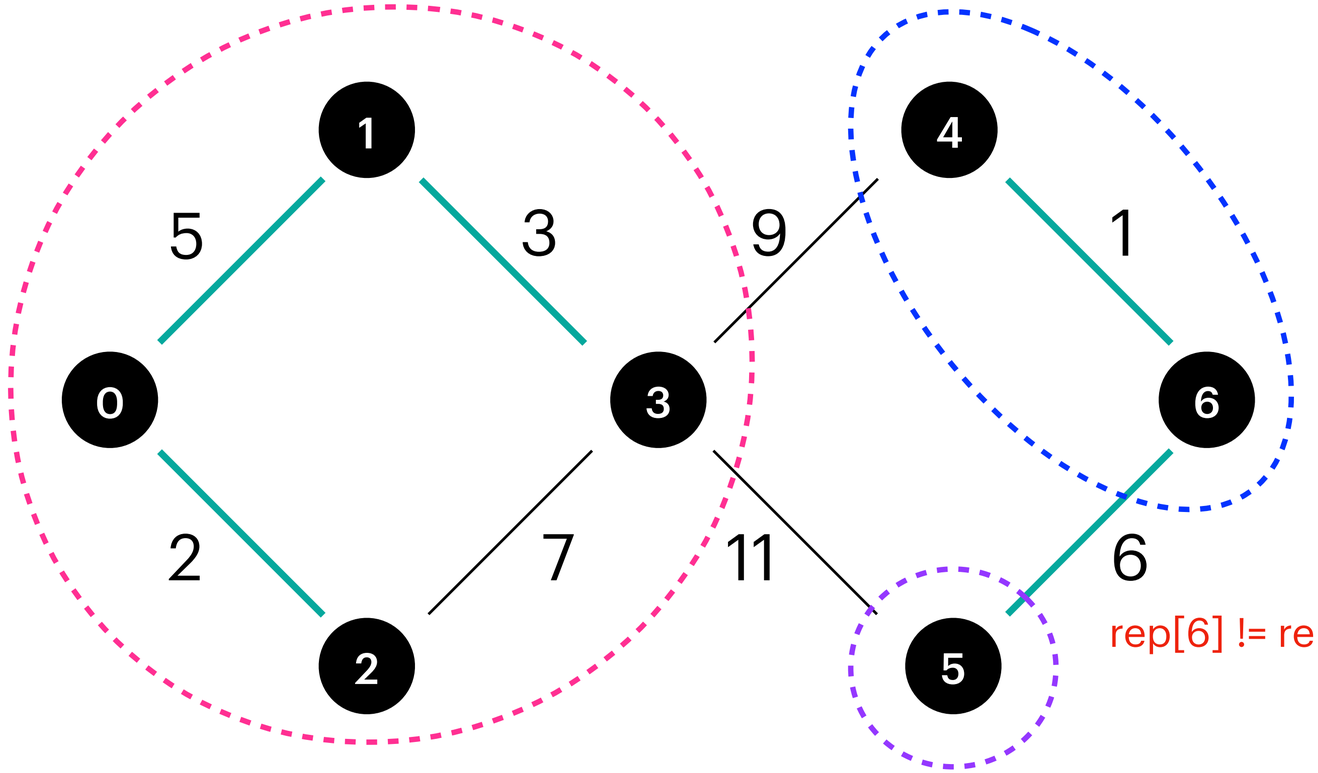
UNION(6,5)

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 }
6	{ 6 }



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

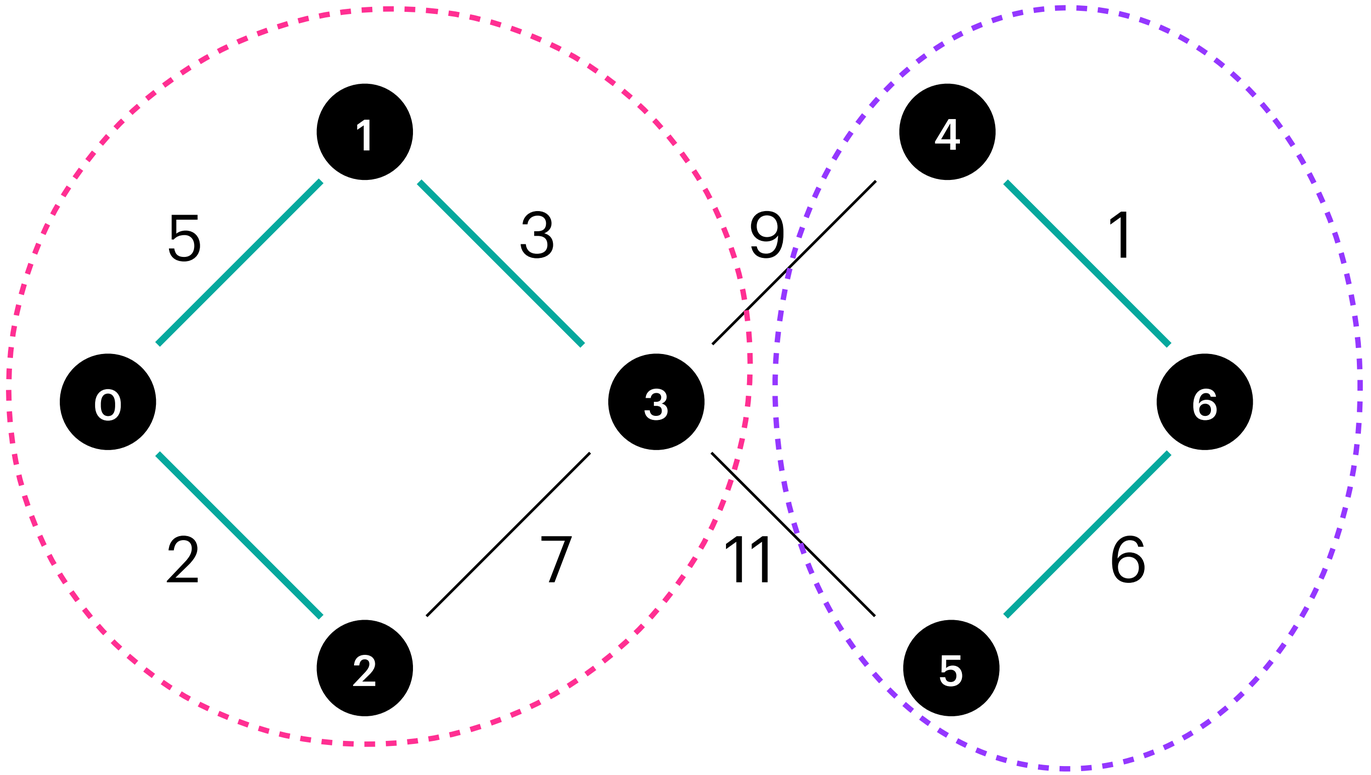
UNION(6,5)

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

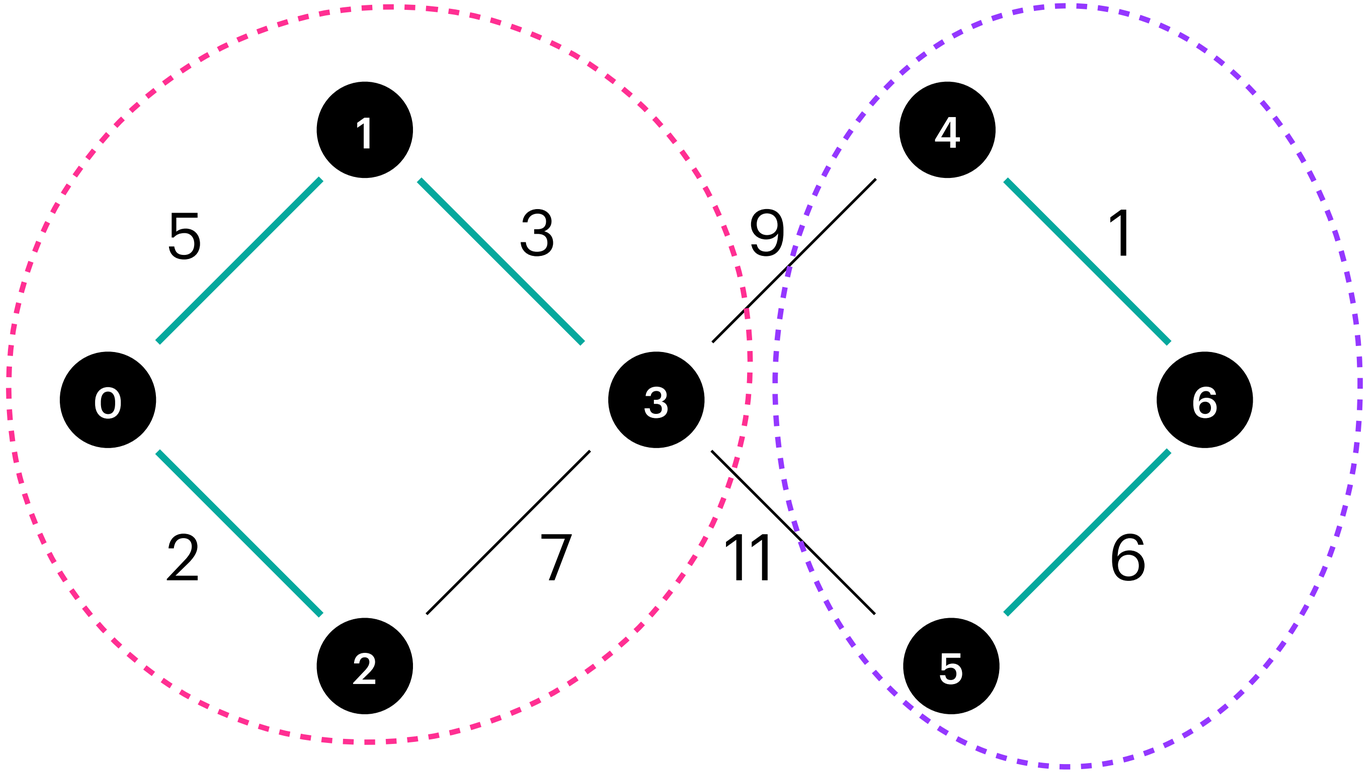
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

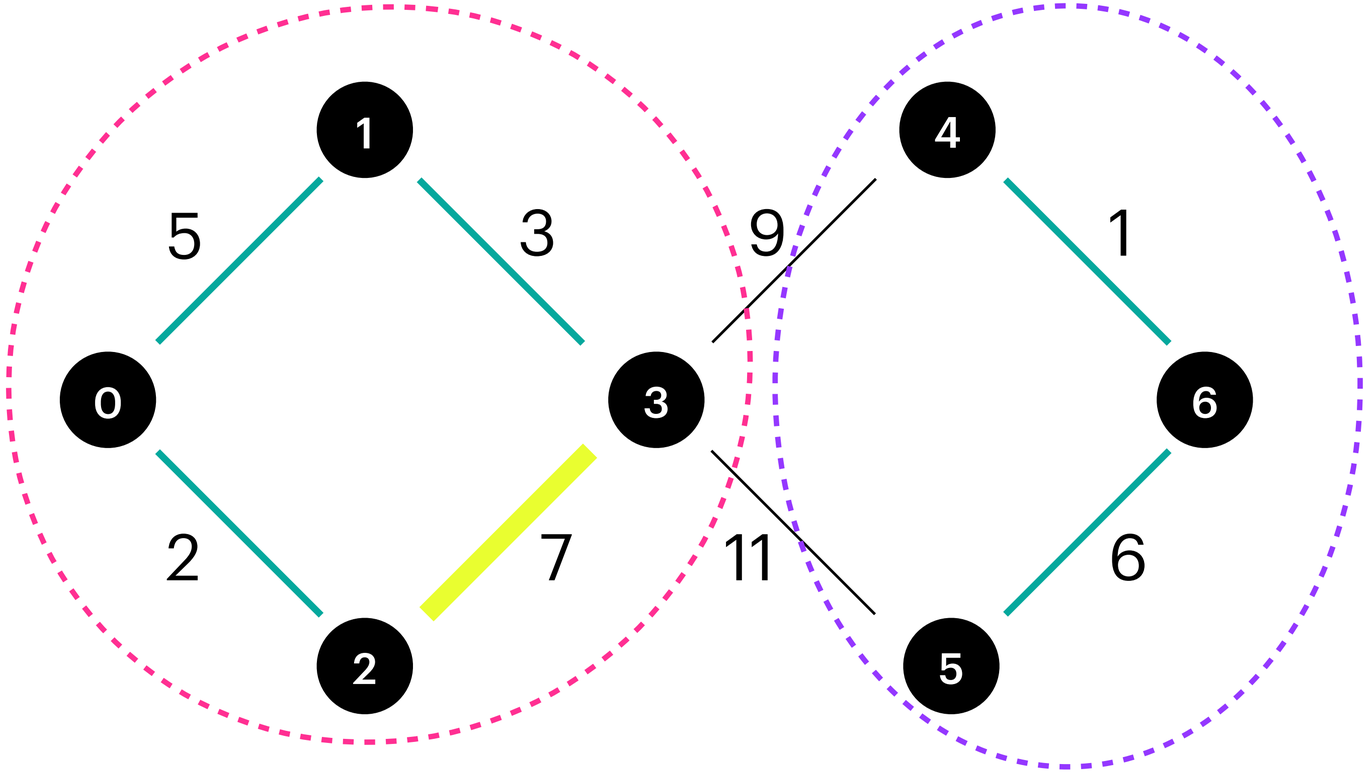
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

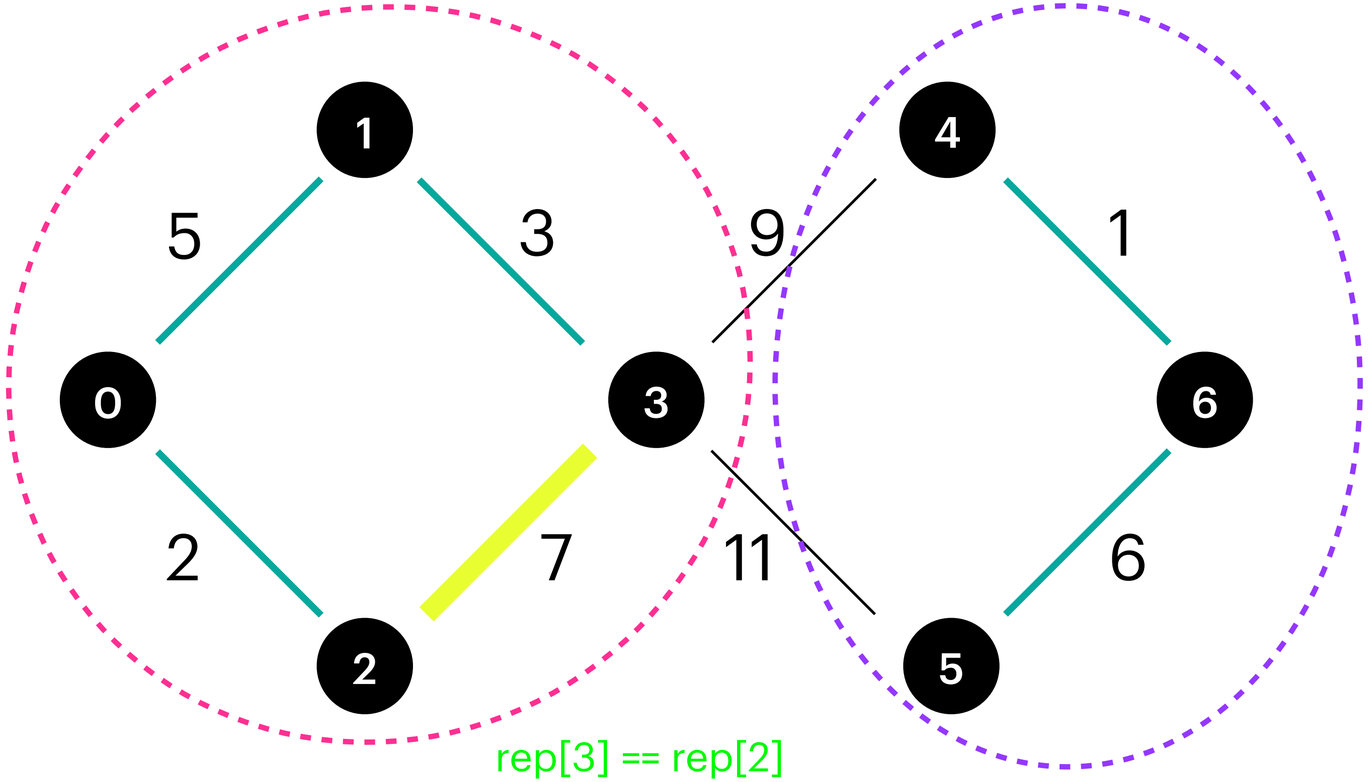
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

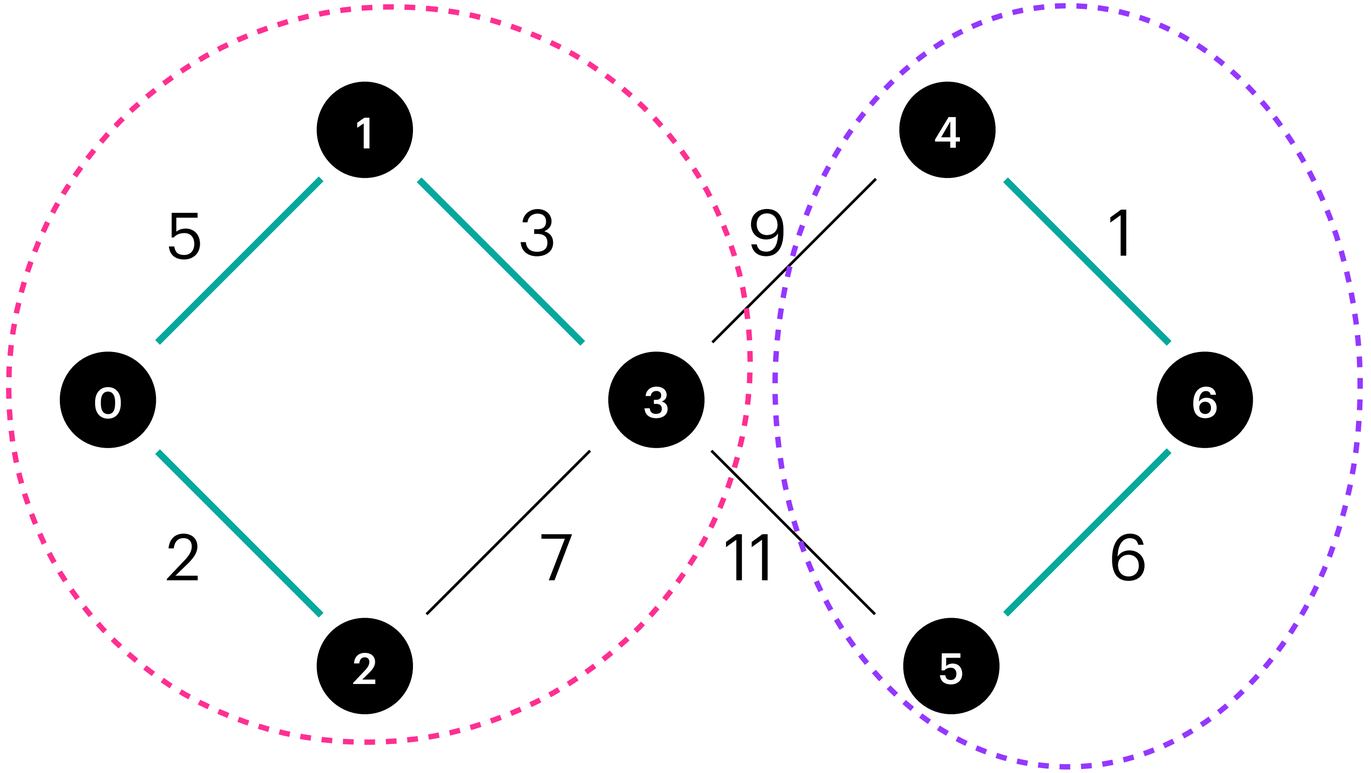
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

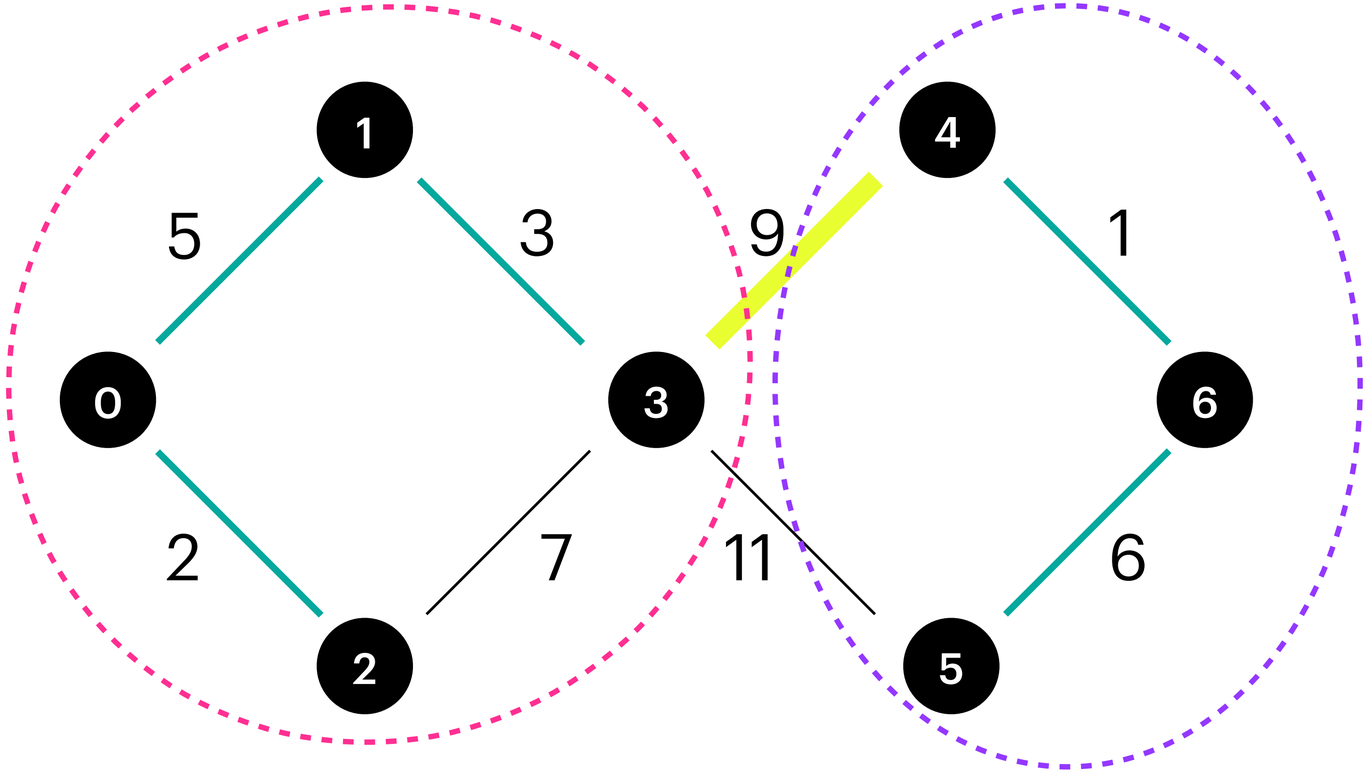
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} }

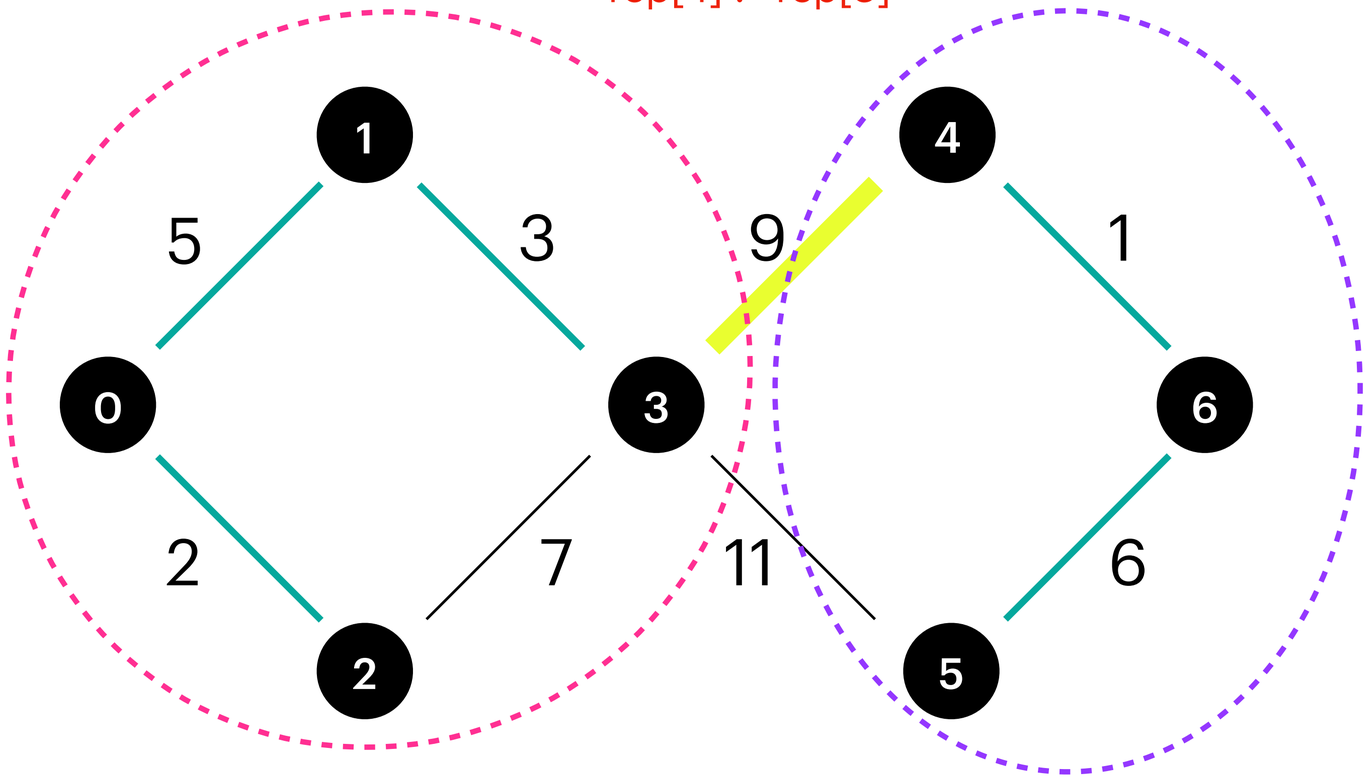
rep[] :

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 }

rep[4] != rep[3]



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

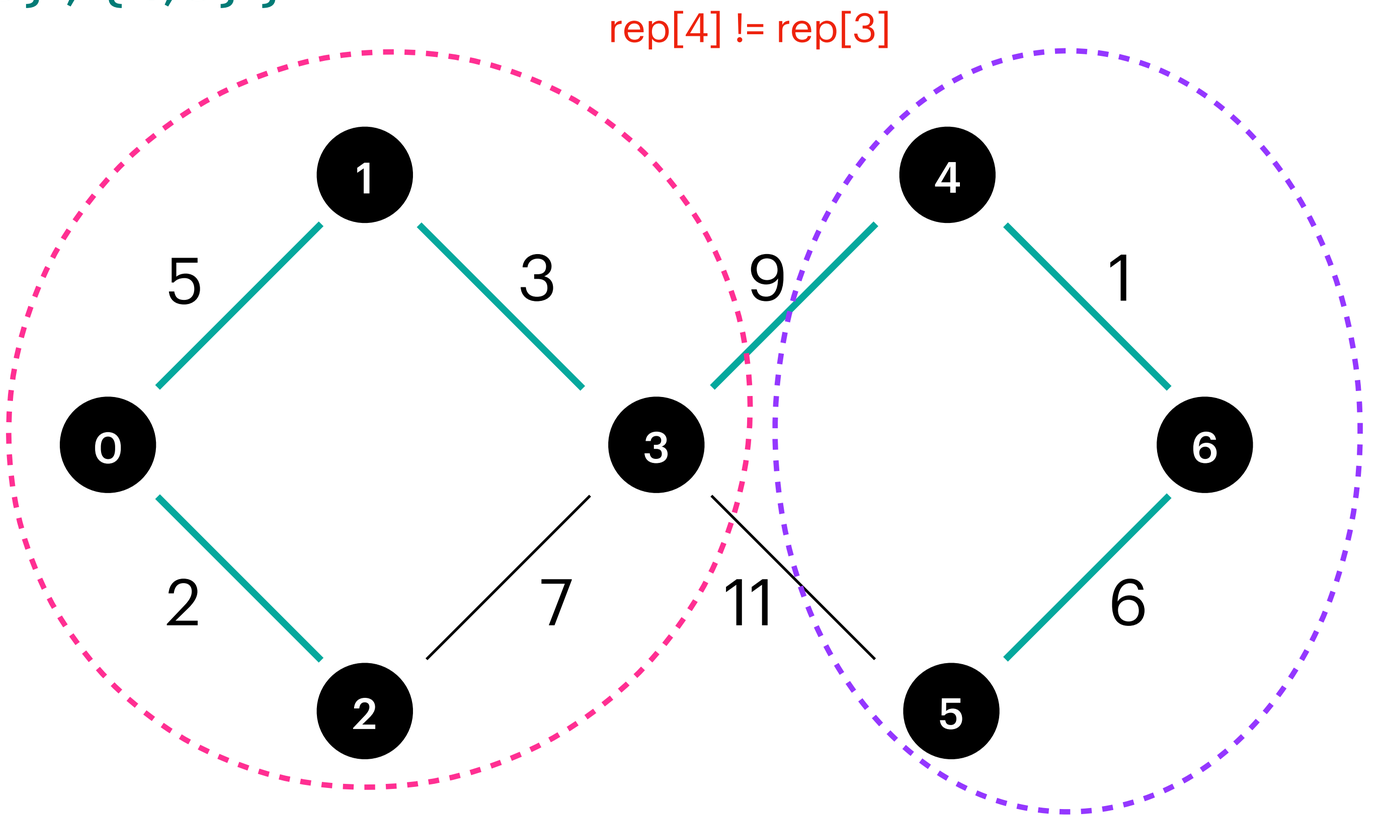
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {4,3} }

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} }

MST

Kruskal's Algorithm

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: $\text{members}[\text{rep}[v]] \leftarrow \text{members}[\text{rep}[v]] \cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
2: $UF \leftarrow \text{MAKE}(V)$
3: $\text{SORT}(E)$
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)

F : edges of the MST

F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {4,3} }

UNION(4,3)

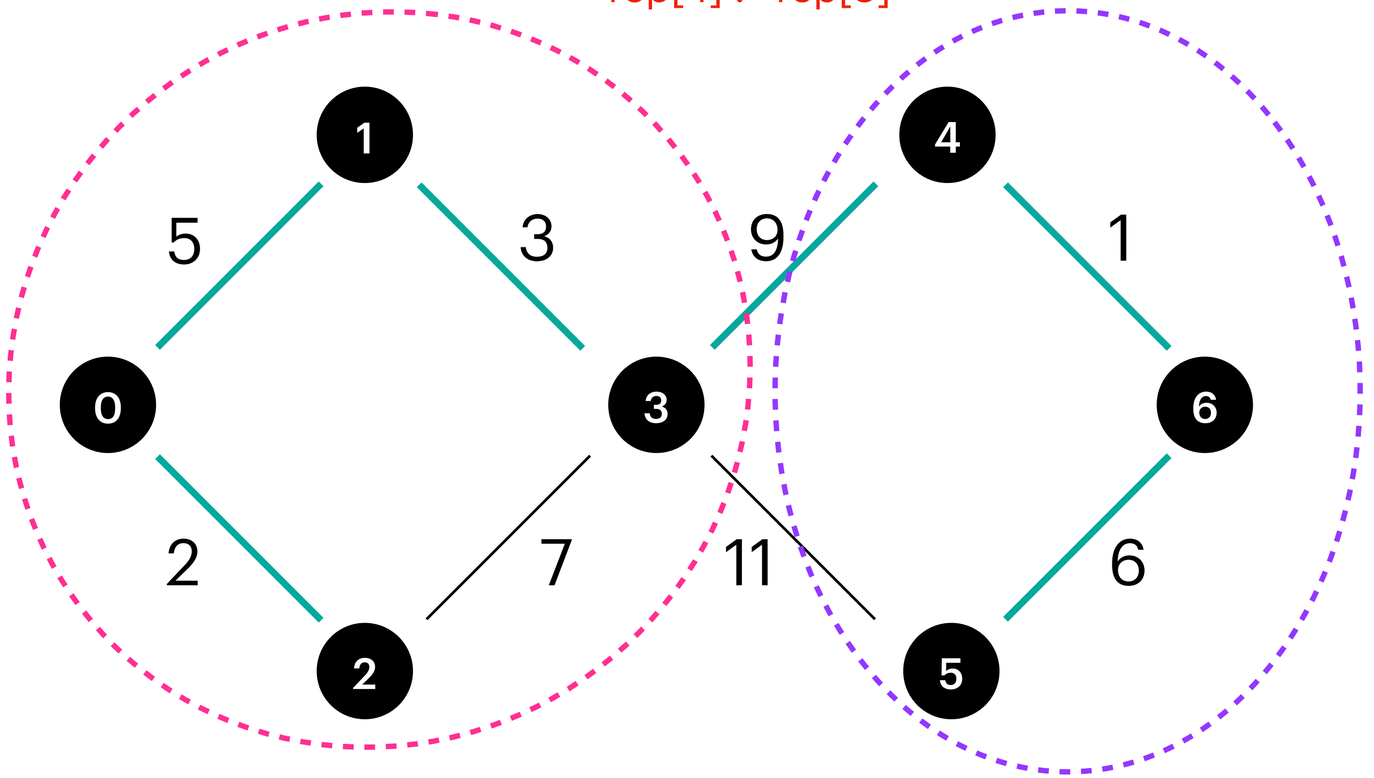
rep[] :

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 , 4 , 6 }
6	{ 6 }

rep[4] != rep[3]



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {4,3} }

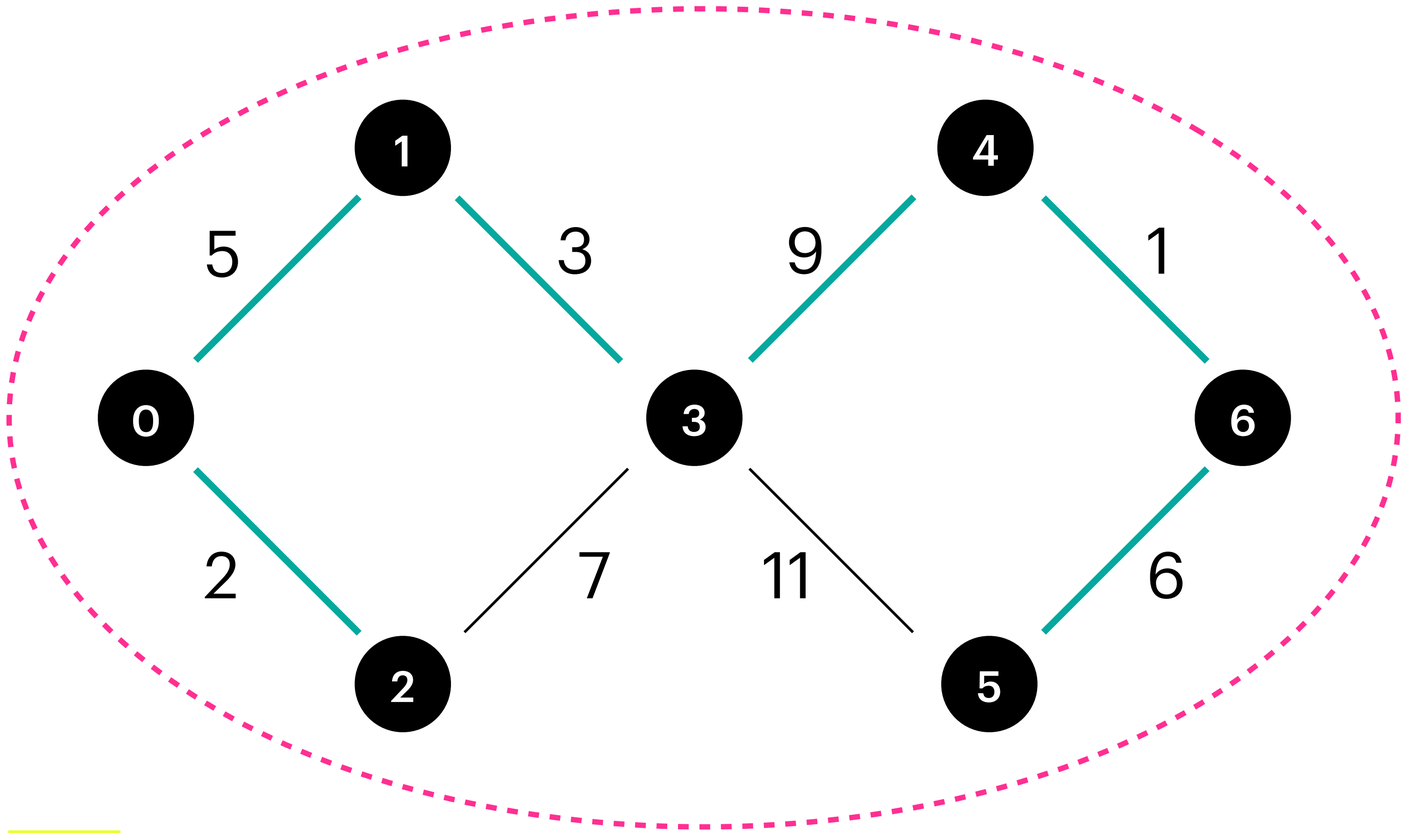
UNION(4,3)

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 , 4 , 6 }
6	{ 6 }



SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

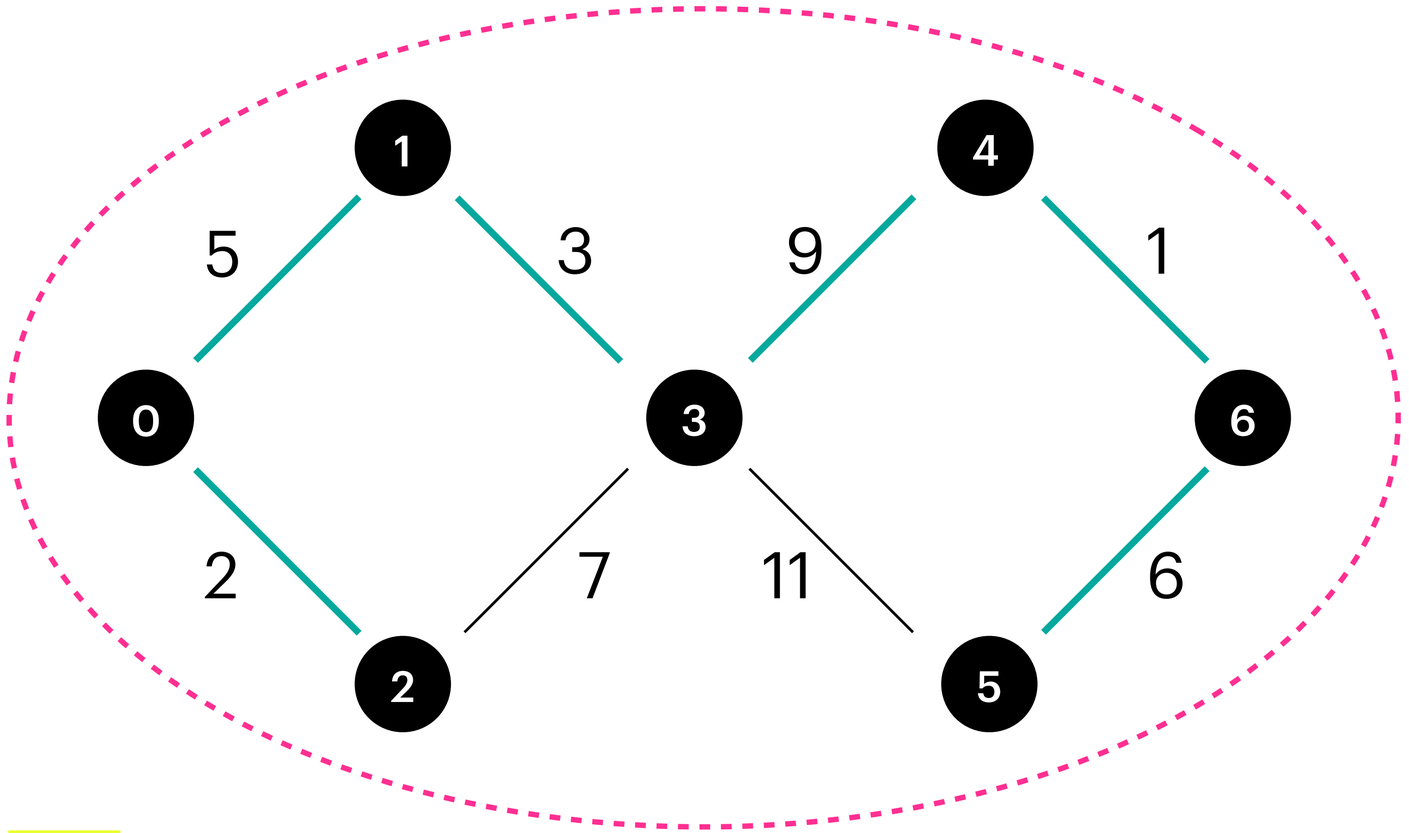
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {4,3} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

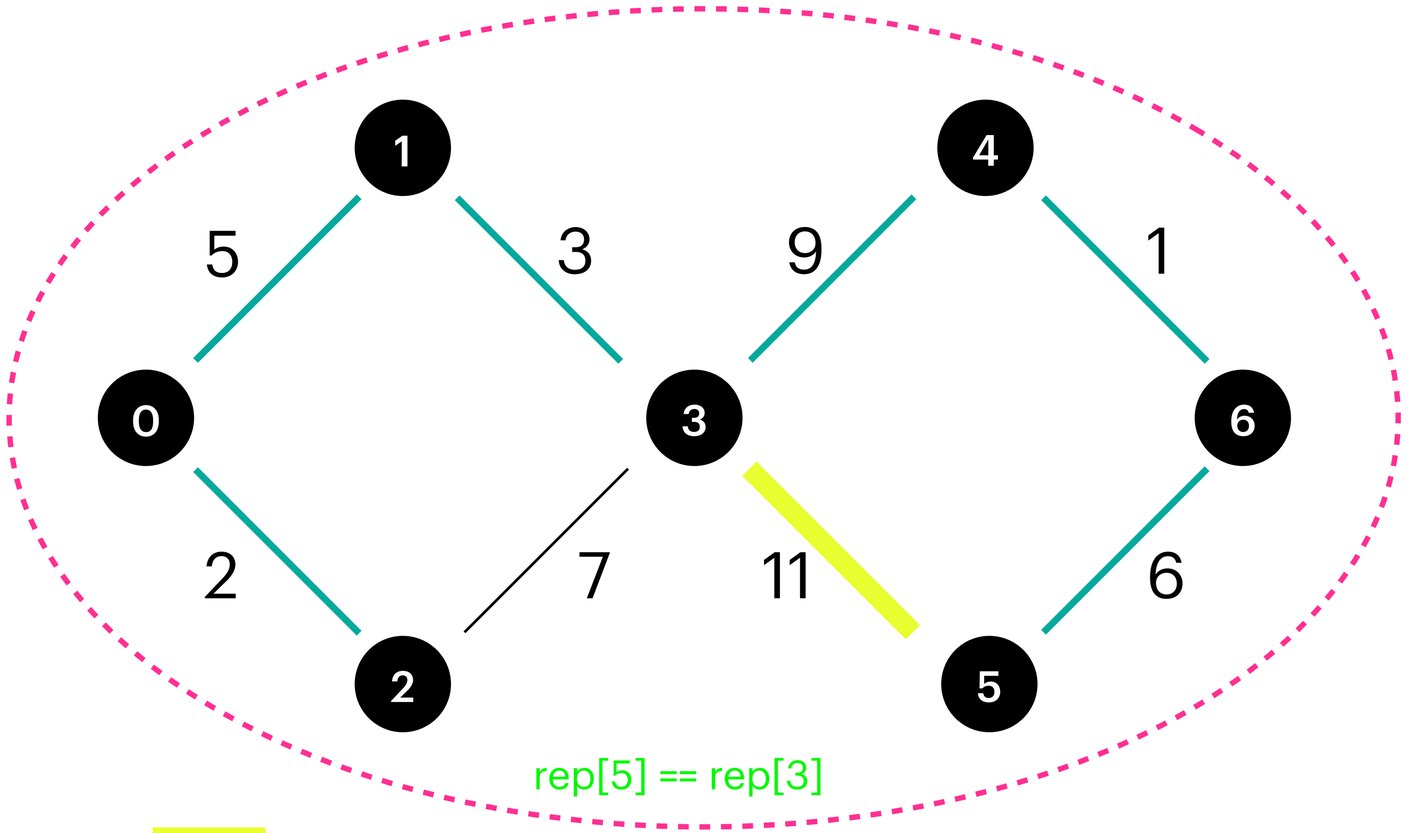
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {4,3} }

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} }

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

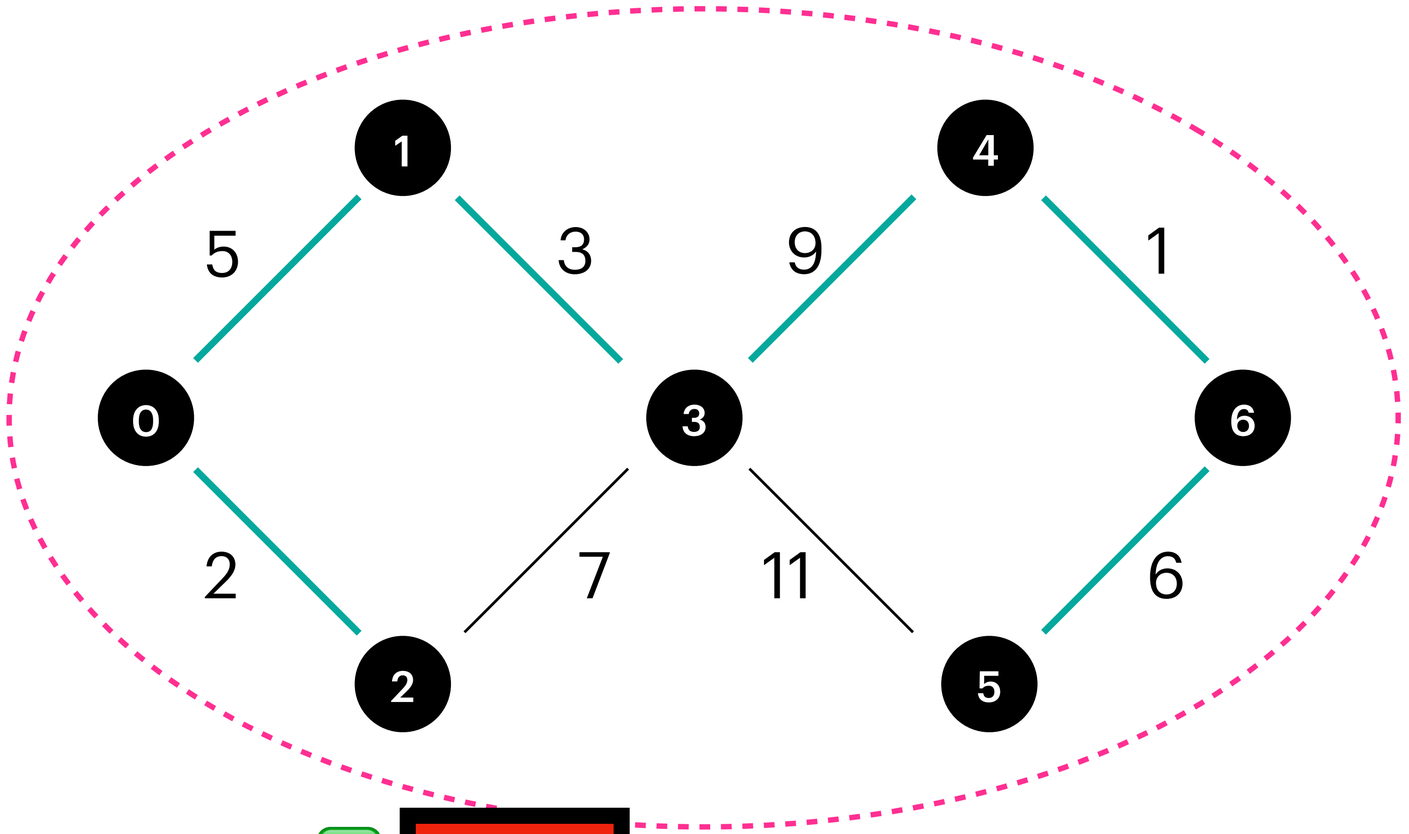
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {4,3} }

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

MST

Kruskal's Algorithm

Algorithm 11 Union-Find(G)

1: **Implementierung:**
2: MAKE(V): rep[v] $\leftarrow v$ $\forall v \in V$
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] $\leftarrow$ rep[v]
9: members[rep[v]] $\leftarrow$ members[rep[v]] $\cup \{x\}$

rep[v] : unique representative of ConComp(v)
members[rep[v]] : list of the nodes in ConComp(rep[v])

Algorithm 12 Kruskal(G) (mit UF-Datenstruktur)

1: $F \leftarrow \varnothing$
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)

F : edges of the MST

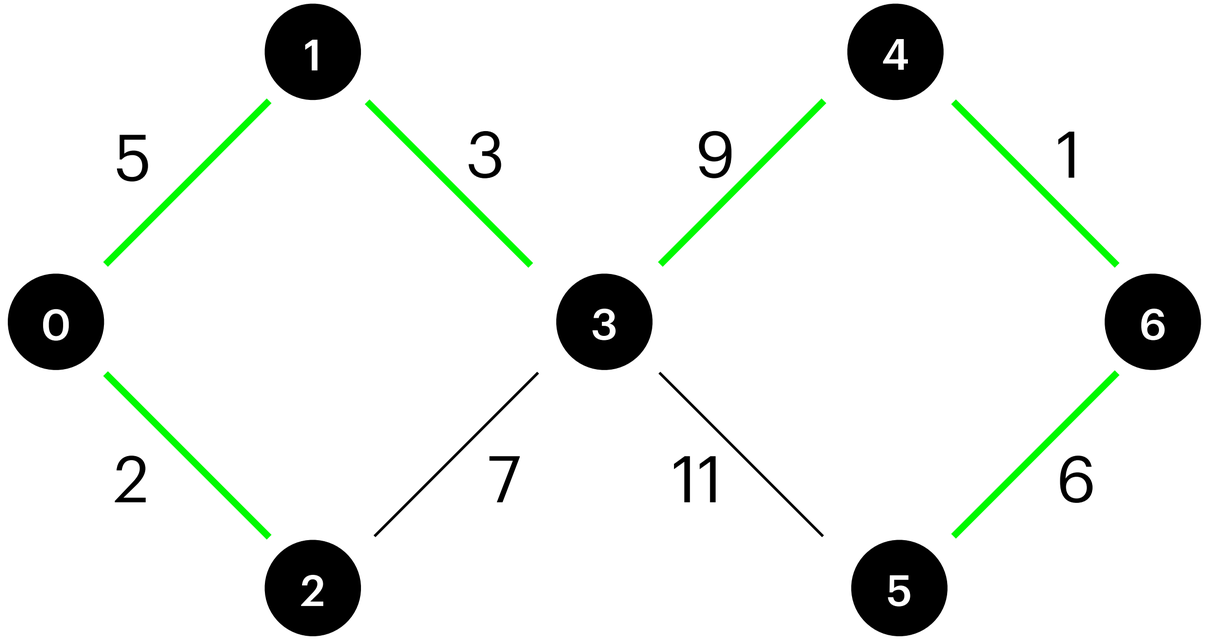
F : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {4,3} }

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 }



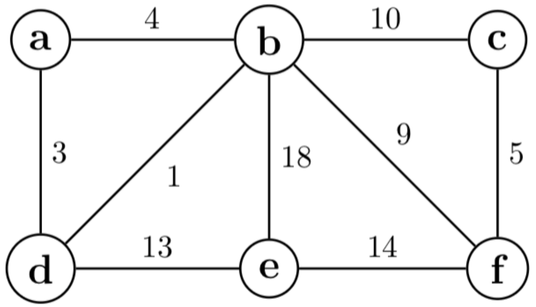
SORT(E) : { {6,4} , {2,0} , {3,1} , {1,0} , {6,5} , {3,2} , {4,3} , {5,3} } ✅ All edges are done

MST

Exam Question

/ 2 P

e) *Minimum Spanning Tree:* Consider the following graph:



i) Highlight the edges that are part of the minimum spanning tree. (Either in the picture above, or you can recreate the graph below).

ii) Write out all positive integers x such that if we replace the weight 1 of edge $\{b, d\}$ in the above graph with x, then edge $\{b, d\}$ would be in at least one minimum spanning tree of the resulting graph.

My To-Do List

Remaining things from sessions

- Graph Sets Code Expert
- Graph Modelling exam question

- Exercise Sheet Corrections
- Quiz Templates

Last Weeks

Organization

- Extra session on friday 13 Dec 14:15 - 17:00 , **CAB H52**
  - All-to-all paths
  - Exercise Correction
- Last session on monday 16 Dec
  - Exam Preparation Session
    - Exam tipps, lernphase tipps, mock exam
  - Recap topics
    - Let me know your specific topics till 23:59 today !
  - Additional things let me know till 23:59 today

Please fill the poll !!!

Questions

Feedbacks , Recommendations

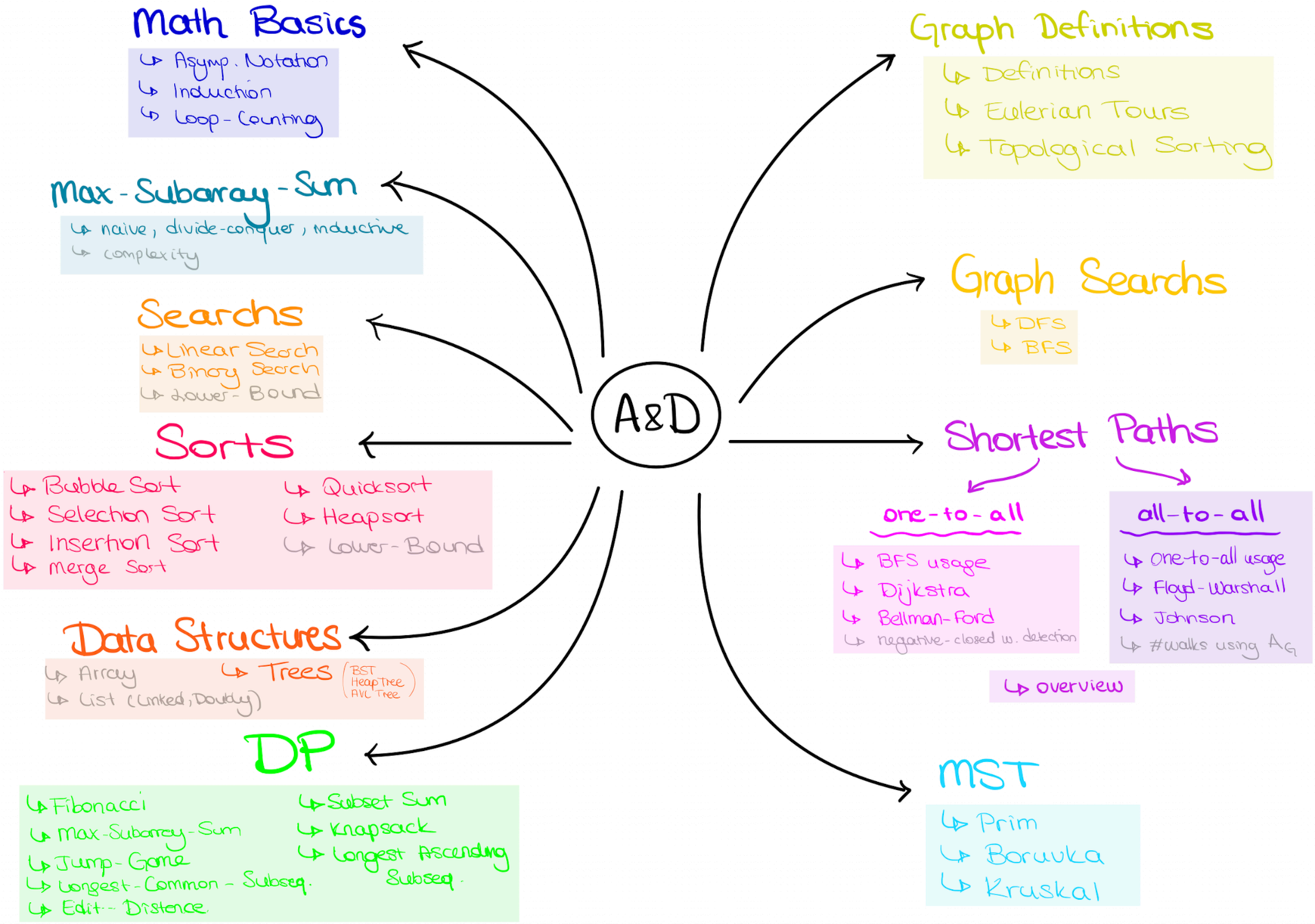
Nil Ozer

A&D

Exercise Session 13

Nil Ozer

A&D Overview


A&D
Math Basics
↳ 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

Outline

- Shortest Paths - all to all
- DP Recap
- Exam preparation session organization

Shortest Paths

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

Recap

Shortest Paths (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|)

Shortest Paths

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	$O(\|V\| * (\|V\| + \|E\|) * \log(\|V\|))$ $O(\|V\|*\|E\| + \|V\|^2 \log(\|V\|))$	with Fibonacci-Heap
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	$O(\|V\| * (\|V\| + \|E\|) * \log n)$ $O(\|V\|*\|E\| + \|V\|^2 \log(\|V\|))$	with Fibonacci-Heap

All-to-all Shortest Paths

Floyd-Warshall

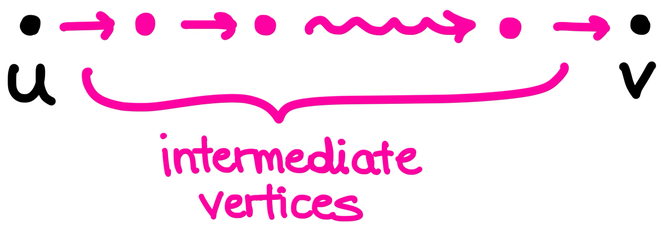
Idea : Two things can happen about a vertex i, considering a walk from vertex u to v

- 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 $DP[0][u][v] \leftarrow \begin{cases} 0 & \text{falls } u = v \text{ ist,} \\ c(u,v) & \text{falls } (u,v) \in E \text{ ist,} \\ \infty & \text{sonst} \end{cases}$

no need to walk, we're already there

just walk that edge, you don't use any intermediate vertices

you can't reach v without using intermediate vertices

2 **for** $i \leftarrow 1, \ldots, n$ **do**

3 **for** $u \leftarrow 1, \ldots, n$ **do**

4 **for** $v \leftarrow 1, \ldots, n$ **do**

5 $DP[i][u][v] \leftarrow \min(DP[i-1][u][v], DP[i-1][u][i] + DP[i-1][i][v])$

don't use vertex i

use vertex i

6 **return** DP

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 $DP[0][u][v] \leftarrow \begin{cases} 0 & \text{falls } u = v \text{ ist,} \\ c(u,v) & \text{falls } (u,v) \in E \text{ ist,} \\ \infty & \text{sonst} \end{cases}$

no need to walk, we're already there

just walk that edge, you don't use any intermediate vertices

you can't reach v without using intermediate vertices

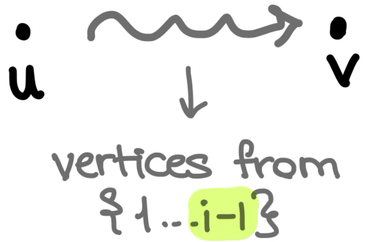
```
2 for i ← 1,...,n do
3     for u ← 1,...,n do
4         for v ← 1,...,n do
5             DP[i][u][v] ← min(DP[i−1][u][v], DP[i−1][u][i]+DP[i−1][i][v])
6 return DP
```

don't use vertex i

use vertex i

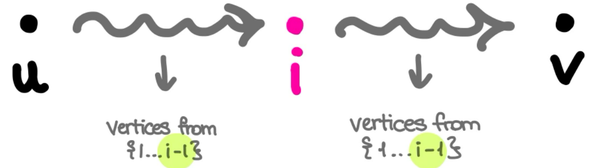
DP [i-1][u][v]

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

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 $DP[0][u][v] \leftarrow \begin{cases} 0 & \text{falls } u = v \text{ ist,} \\ c(u,v) & \text{falls } (u,v) \in E \text{ ist,} \\ \infty & \text{sonst} \end{cases}$

no need to walk, we're already there

just walk that edge, you don't use any intermediate vertices

you can't reach v without using intermediate vertices

2 **for** $i \leftarrow 1, \ldots, n$ **do**
3 **for** $u \leftarrow 1, \ldots, n$ **do**
4 **for** $v \leftarrow 1, \ldots, n$ **do**
5 $DP[i][u][v] \leftarrow \min(DP[i-1][u][v], DP[i-1][u][i] + DP[i-1][i][v])$
6 **return** DP

don't use vertex i

use vertex i

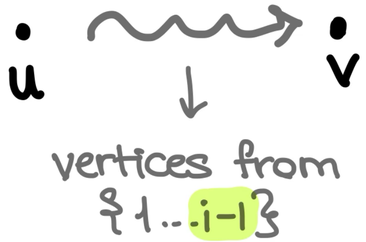
Runtime: $O(n^3)$

Solution at: DP[n][][]

"using all vertices"

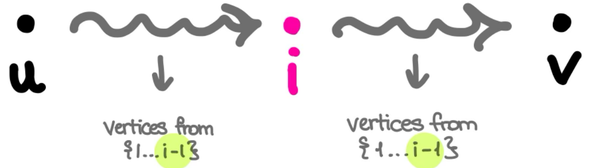
DP [i-1][u][v]

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

$\exists$ a negative closed walk $\iff$ $\exists$ v with DP[n][v][v] < 0

DP[n][v][v] : The shortest walk from v to v using {1...n}

All-to-all Shortest Paths

Johnson

Problem is the negative edges ! (we can’t use dijkstra)

Idea :

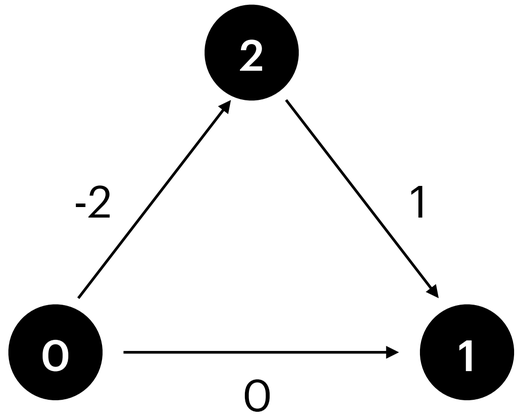
- Make all edge weights ≥ 0
- n x Dijkstra

DOES NOT WORK WITH NEGATIVE CYCLES !

We know Dijkstra, how can we make all edge weights ≥ 0 ?

All-to-all Shortest Paths

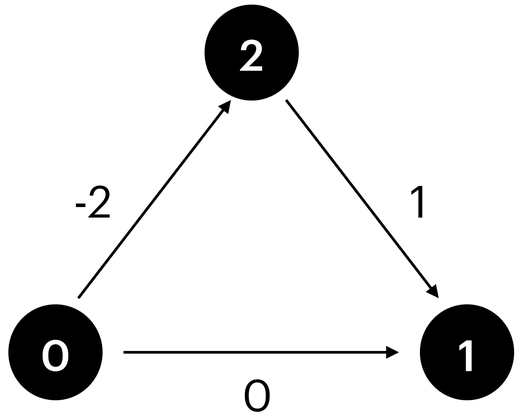
Johnson - Making all edge weights ≥ 0



All-to-all Shortest Paths

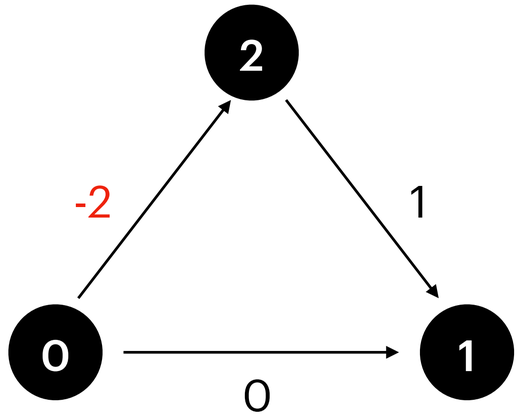
Johnson - Making all edge weights ≥ 0

c(0,2) = -2 c(2,1) = 1 c(0,1) = 0



All-to-all Shortest Paths

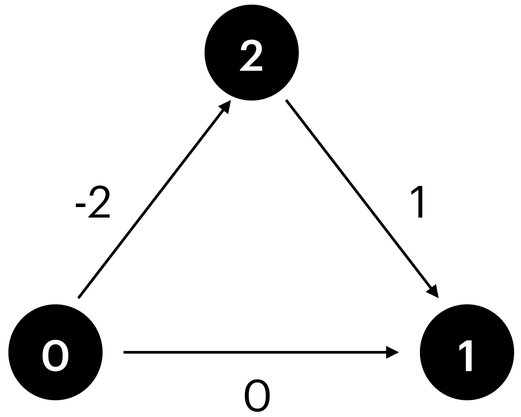
Johnson - Making all edge weights ≥ 0



All-to-all Shortest Paths

Johnson - Making all edge weights ≥ 0

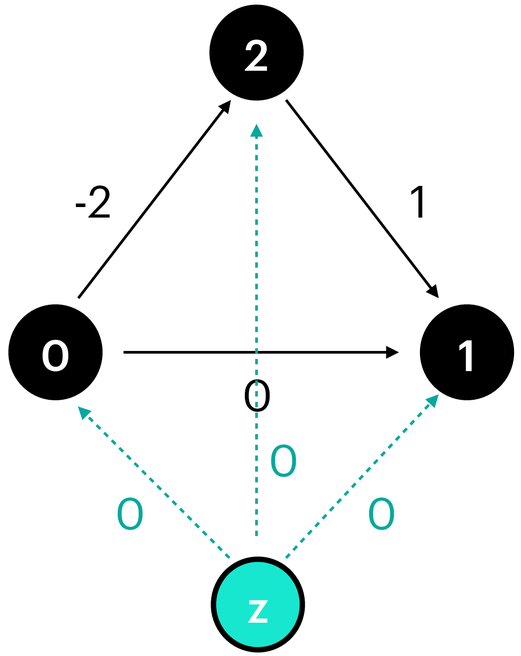
- Add a new vertex z , and connect it to every vertex in the original G with and edge with cost 0



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



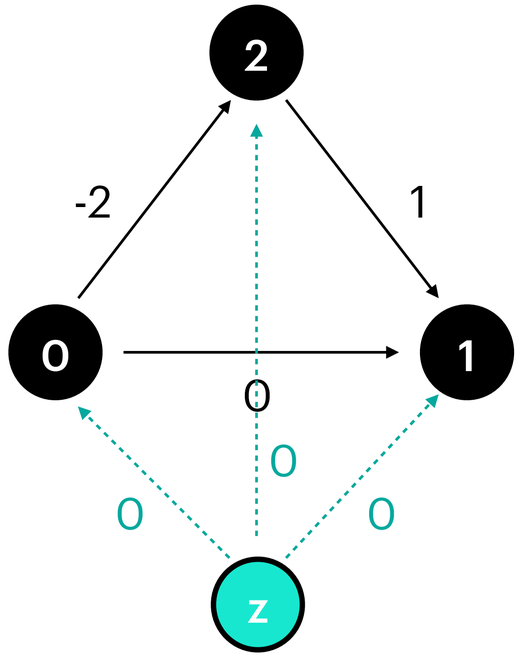
All-to-all Shortest Paths

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

  with Bellman-Ford x1 from z

  h(u) := length of the shortest path from z to u



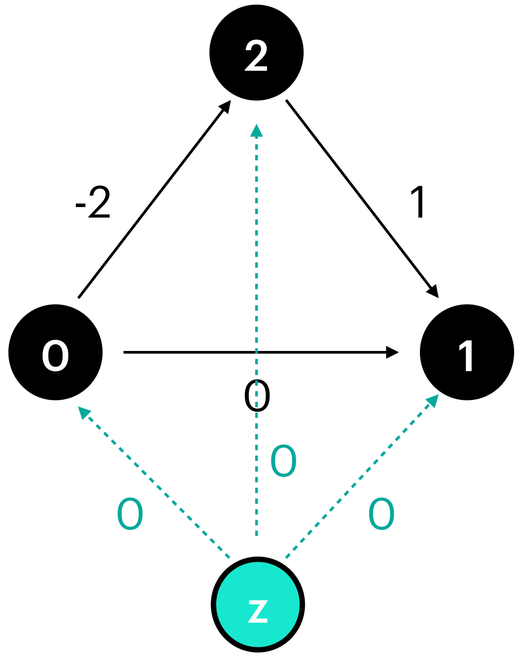
All-to-all Shortest Paths

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

  with Bellman-Ford x1 from z

  h(u) := length of the shortest path from z to u



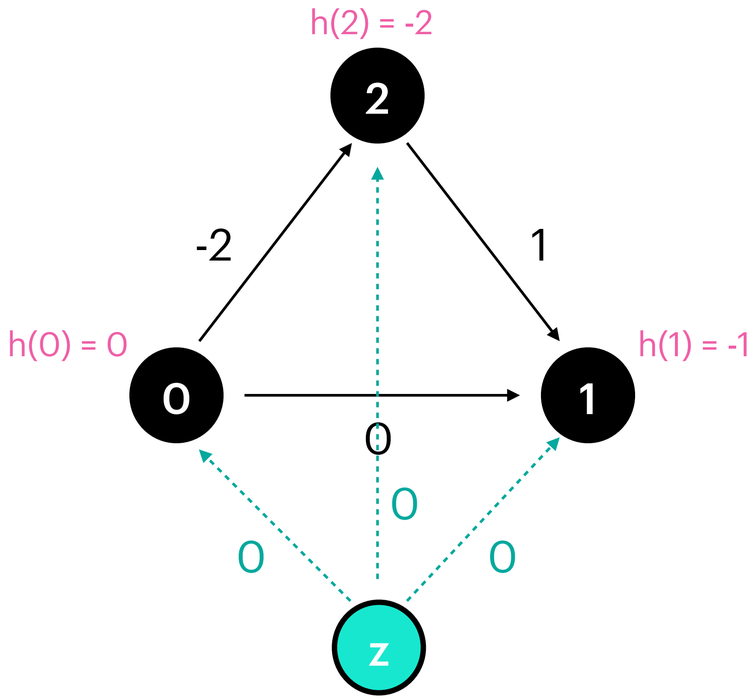
All-to-all Shortest Paths

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



All-to-all Shortest Paths

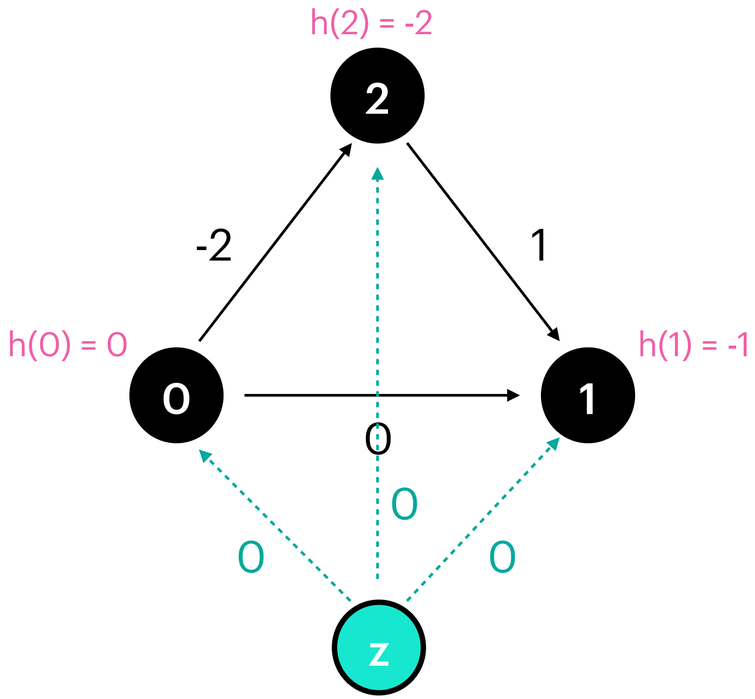
Johnson - Making all edge weights ≥ 0

- 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

  with Bellman-Ford x1 from z

  h(u) := length of the shortest path from z to u
- Calculate c'(u,v) for every edge

  c'(u,v) := c(u,v) + h(u) - h(v)



All-to-all Shortest Paths

Johnson - Making all edge weights ≥ 0

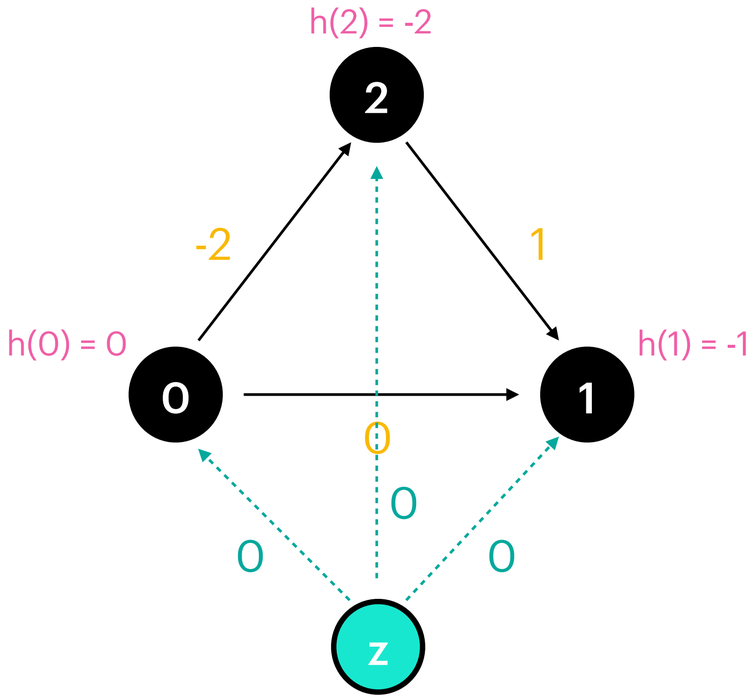
- Add a new vertex z , and connect it to every vertex in the original G with and edge with cost 0
- Find h(u) for every u

  with Bellman-Ford x1 from z

  h(u) := length of the shortest path from z to u
- 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

All-to-all Shortest Paths

Johnson - Making all edge weights ≥ 0

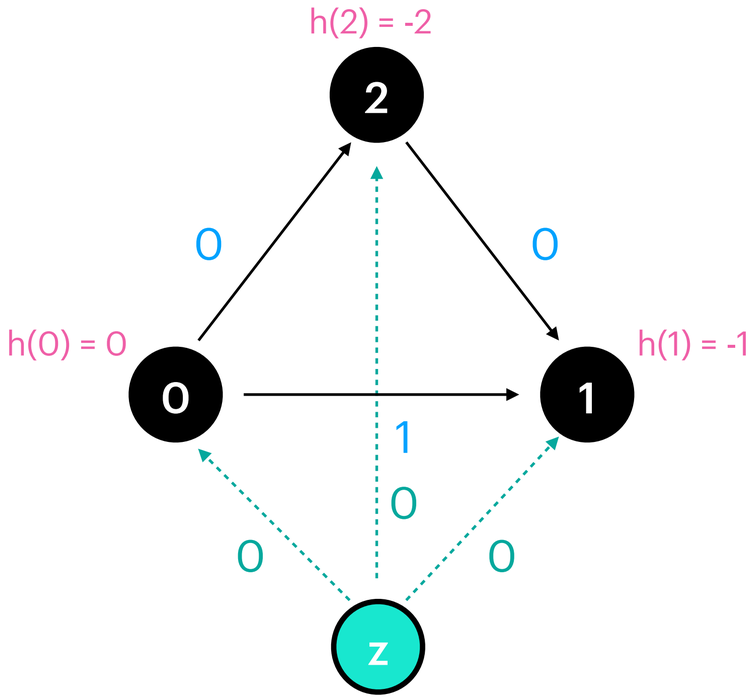
- Add a new vertex z , and connect it to every vertex in the original G with and edge with cost 0
- Find h(u) for every u

  with Bellman-Ford x1 from z

  h(u) := length of the shortest path from z to u
- 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

All-to-all Shortest Paths

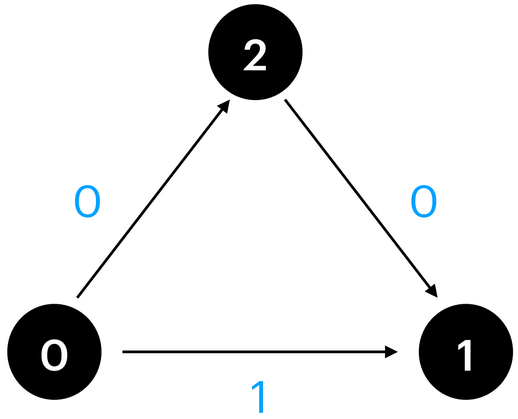
Johnson - Making all edge weights ≥ 0

- 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

  with Bellman-Ford x1 from z

  h(u) := length of the shortest path from z to u
- Calculate c’(u,v) for every edge

  c’(u,v) := c(u,v) + h(u) - h(v)



All-to-all Shortest Paths

Johnson

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 !

Shortest Paths

Overview

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

Shortest Paths

Exam Tipps

- Know every detail of the “overview”
- Graph Modelling
  - Model the problem correctly, define G,V,E,w
  - Know which algorithm to apply for that particular graph problem
    - BFS,DFS
    - Shortest Paths
    - MST

- Practice, practice, practice !!!

Recap

DP

Recap

Theory Task T3. / 9 P

Let $A = [a_1, a_2, \ldots, a_n]$ be an array of non-negative integers. Given A and a non-negative integers $S \geq 0$, we want to determine whether S can be written as a (non-repeating) sum of elements of A, where we are allowed to take the square of elements. Formally, we want to determine if there exist $I, J \subseteq \{1, 2, \ldots, n\}$ with $I \cap J = \emptyset$, $I \cup J = \{1, 2, \ldots, n\}$ such that:

$$S = \sum_{i \in I} a_i + \sum_{j \in J} a_j^2.$$

Provide a *dynamic programming* algorithm that outputs `True` if this is possible, and `False` otherwise. For example,

- The inputs $A = [2, 4, 4]$, $S = 34$ should result in `True`, since $34 = 2 + 4^2 + 4^2$.
- The inputs $A = [2, 4, 4]$, $S = 35$ should result in `False`.
- The inputs $A = [2, 4, 3, 22]$, $S = 21$ should result in `False`.

In order to obtain full points, your algorithm should run in time $O(n^2 \cdot S)$. Address the following aspects of 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) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n and S, and justify your answer.

Theory Task T3. / 9 P

You are given an array of n natural numbers $a_1, \ldots, a_n \in \mathbb{N}$, and two natural numbers $A, B \in \mathbb{N}$. You want to determine whether there is a subset $I \subseteq \{1, \ldots, n\}$ satisfying

$$\sum_{i \in I} a_i = A \quad \text{and} \quad \sum_{i \in I} a_i^2 = B.$$

For example,

- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1, 4, 5, 3]$, $A = 8$ and $B = 30$ is *yes* because the set of indices $I = \{1, 4, 6\}$, which corresponds to $(a_i)_{i \leq I} = [2, 1, 5]$, yields the *sum* $2 + 1 + 5 = 8$ and the *sum-of-squares* $2^2 + 1^2 + 5^2 = 30$.
- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1]$, $A = 6$ and $B = 15$ is *no*.

Provide a *dynamic programming* algorithm that determines whether such a subset I exists. In order to get full points, your algorithm should have an $O(n \cdot A \cdot B)$ runtime. 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 ?

Theory Task T3. / 9 P

ou are given an array of n natural numbers $a_1, \ldots, a_n \in \mathbb{N}$ summing to $A := \sum_{i=1}^{n} a_i$, which is a ultiple of 3. You want to determine whether it is possible to partition $\{1, \ldots, n\}$ into three disjoint bsets I, J, K such that the corresponding elements of the array yield the same sum, i.e.

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{A}{3}.$$

ote that I, J, K form a partition of $\{1, \ldots, n\}$ if and only if $I \cap J = I \cap K = J \cap K = \emptyset$ and $\cup J \cup K = \{1, \ldots, n\}$.

or example, the answer for the input $[2, 4, 8, 1, 4, 5, 3]$ is *yes*, because there is the partition $\{3, 4\}$, $2, 6\}$, $\{1, 5, 7\}$ (corresponding to the subarrays $[8, 1]$, $[4, 5]$, $[2, 4, 3]$, which are all summing to 9). n the other hand, the answer for the input $[3, 2, 5, 2]$ is *no*.

rovide a *dynamic programming* algorithm that determines whether such a partition exists. Your lgorithm should have an $\mathcal{O}(nA^2)$ runtime to get full points. Address the following aspects in your olution:

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

be com
lo not d
es be co
s ?
lution be
our algo

Theory Task T3. / 8 P

An array of non-negative integers $A = [a_1, \ldots, a_n]$ is called *summy* if and only if, for all $i \in \{2, \ldots, n\}$, there exists a (possibly empty) set $I \subseteq \{1, \ldots, i-1\}$ such that $a_i = \sum_{j \in I} a_j$. In other terms, every integer in the array except the first one must be the sum of (distinct) integers that precede it in the array.

For example,

- The array $[2, 2, 4, 6, 0, 12]$ is summy, because $2 = 2$, $4 = 2 + 2$, $6 = 2 + 4$, $12 = 2 + 4 + 6$.
- The array $[2, 2, 4, 6, 0, 13]$ is not summy, since 13 can not be written as a sum of integers from $\{2, 2, 4, 6, 0\}$.

Provide a *dynamic programming* algorithm that, given an array A of length n, returns `True` if the array is summy, and `False` otherwise. In order to obtain full points, your algorithm should have an $O(n \cdot \max A)$ runtime (where $\max A$ means the maximum value of entries in A). 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) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n and $\max A$, and justify your answer.

Theory Task T3. / 9 P

Let m, r be two integers satisfying $m \geq 2$ and $0 \leq r < m$. We say that a finite set $A \subset \mathbb{N}$ of natural numbers is (m, r)-aligned if

$$\left(\sum_{x \in A} x\right) \bmod m = r\,.$$

Note that for $A = \varnothing$, we adopt the convention that $\sum_{x \in A} x = 0$. Hence, the empty set is $(m, 0)$-aligned for every $m > 0$.

Given three integers m, r, n such that $0 \leq r < m < n$ and $m \geq 2$, we would like to determine the number of subsets of $\{1, 2, \ldots, n\}$ which are (m, r)-aligned.

For example,

- If $r = 1$, $m = 2$ and $n = 3$, the subsets of $\{1, 2, 3\}$ that are $(3, 1)$-aligned are $\{1\}$, $\{3\}$, $\{1, 2\}$ and $\{2, 3\}$. Hence, the answer is 4.

Provide a *dynamic programming* algorithm that solves the problem. In order to get full points, your algorithm should have an $O(n \cdot m)$ runtime. 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) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n, m and r, and justify your answer.

n an entry be computed from the values of other entries?
ries that do not depend on others.

can entries be computed so that values needed for each vious steps?

ıe final solution be extracted once the table has been filled?

g time of your algorithm? Provide it in Θ-notation in terms er.

DP

Recap

Theory Task T3. / 9 P

Let $A = [a_1, a_2, \ldots, a_n]$ be an array of non-negative integers. Given A and a non-negative integers $S \geq 0$, we want to determine whether S can be written as a (non-repeating) sum of elements of A, where we are allowed to take the square of elements. Formally, we want to determine if there exist $I, J \subseteq \{1, 2, \ldots, n\}$ with $I \cap J = \emptyset$, $I \cup J = \{1, 2, \ldots, n\}$ such that:

$$S = \sum_{i \in I} a_i + \sum_{j \in J} a_j^2.$$

Provide a *dynamic programming* algorithm that outputs `True` if this is possible, and `False` otherwise. For example,

- The inputs $A = [2, 4, 4]$, $S = 34$ should result in `True`, since $34 = 2 + 4^2 + 4^2$.
- The inputs $A = [2, 4, 4]$, $S = 35$ should result in `False`.
- The inputs $A = [2, 4, 3, 22]$, $S = 21$ should result in `False`.

In order to obtain full points, your algorithm should run in time $O(n^2 \cdot S)$. Address the following aspects of your solution:

1) *Definition of the DP table*: What are the dimensions
of each entry?
2) *Computation of an entry*: How can an entry be co
Specify the base cases, i.e., the entries that do not
3) *Calculation order*: In which order can entries be c
entry have been determined in previous steps?
4) *Extracting the solution*: How can the final solution b
5) *Running time*: What is the running time of your alg
of n and S, and justify your answer.

Theory Task T3. / 8 P

An array of non-negative integers $A = [a_1, \ldots, a_n]$ is called *summy* if and only if, for all $i \in \{2, \ldots, n\}$, there exists a (possibly empty) set $I \subseteq \{1, \ldots, i-1\}$ such that $a_i = \sum_{j \in I} a_j$. In other terms, every integer in the array except the first one must be the sum of (distinct) integers that precede it in the array.

For example,

- The array $[2, 2, 4, 6, 0, 12]$ is summy, because $2 = 2$, $4 = 2 + 2$, $6 = 2 + 4$, $12 = 2 + 4 + 6$.
- The array $[2, 2, 4, 6, 0, 13]$ is not summy, since 13 can not be written as a sum of integers from $\{2, 2, 4, 6, 0\}$.

Provide a *dynamic programming* algorithm that, given an array A of length n, returns `True` if the array is summy, and `False` otherwise. In order to obtain full points, your algorithm should have an $O(n \cdot \max A)$ runtime (where $\max A$ means the maximum value of entries in A). 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) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n and $\max A$, and justify your answer.

Theory Task T3. / 9 P

You are given an array of n natural numbers $a_1, \ldots, a_n \in \mathbb{N}$, and two natural numbers $A, B \in \mathbb{N}$. You want to determine whether there is a subset $I \subseteq \{1, \ldots, n\}$ satisfying

$$\sum_{i \in I} a_i = A \quad \text{and} \quad \sum_{i \in I} a_i^2 = B.$$

For example,

- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1, 4, 5, 3]$, $A = 8$ and $B = 30$ is *yes* because the set of indices $I = \{1, 4, 6\}$, which corresponds to $(a_i)_{i \leq I} = [2, 1, 5]$, yields the *sum* $2 + 1 + 5 = 8$ and the *sum-of-squares* $2^2 + 1^2 + 5^2 = 30$.
- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1]$, $A = 6$ and $B = 15$ is *no*.

Provide a *dynamic programming* algorithm that determines whether such a subset I exists. In order to get full points, your algorithm should have an $O(n \cdot A \cdot B)$ runtime. 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 ?

be com
lo not d
es be co
s ?
lution be
our algo

Theory Task T3. / 9 P

Let m, r be two integers satisfying $m \geq 2$ and $0 \leq r < m$. We say that a finite set $A \subset \mathbb{N}$ of natural numbers is (m, r)-aligned if

$$\left(\sum_{x \in A} x\right) \bmod m = r\,.$$

Note that for $A = \varnothing$, we adopt the convention that $\sum_{x \in A} x = 0$. Hence, the empty set is $(m, 0)$-aligned for every $m > 0$.

Given three integers m, r, n such that $0 \leq r < m < n$ and $m \geq 2$, we would like to determine the number of subsets of $\{1, 2, \ldots, n\}$ which are (m, r)-aligned.

For example,

- If $r = 1$, $m = 2$ and $n = 3$, the subsets of $\{1, 2, 3\}$ that are $(3, 1)$-aligned are $\{1\}$, $\{3\}$, $\{1, 2\}$ and $\{2, 3\}$. Hence, the answer is 4.

Provide a *dynamic programming* algorithm that solves the problem. In order to get full points, your algorithm should have an $O(n \cdot m)$ runtime. 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) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n, m and r, and justify your answer.

heory Task T3. / 9 P

ou are given an array of n natural numbers $a_1, \ldots, a_n \in \mathbb{N}$ summing to $A := \sum_{i=1}^{n} a_i$, which is a ultiple of 3. You want to determine whether it is possible to partition $\{1, \ldots, n\}$ into three disjoint ubsets I, J, K such that the corresponding elements of the array yield the same sum, i.e.

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{A}{3}.$$

ote that I, J, K form a partition of $\{1, \ldots, n\}$ if and only if $I \cap J = I \cap K = J \cap K = \emptyset$ and $\cup J \cup K = \{1, \ldots, n\}$.

or example, the answer for the input $[2, 4, 8, 1, 4, 5, 3]$ is *yes*, because there is the partition $\{3, 4\}$, $2, 6\}$, $\{1, 5, 7\}$ (corresponding to the subarrays $[8, 1]$, $[4, 5]$, $[2, 4, 3]$, which are all summing to 9). n the other hand, the answer for the input $[3, 2, 5, 2]$ is *no*.

rovide a *dynamic programming* algorithm that determines whether such a partition exists. Your lgorithm should have an $\mathcal{O}(nA^2)$ runtime to get full points. Address the following aspects in your olution:

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

n an entry be computed from the values of other entries? ries that do not depend on others.

can entries be computed so that values needed for each vious steps?

ıe final solution be extracted once the table has been filled?

g time of your algorithm? Provide it in Θ-notation in terms er.

DP

Σ

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

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

DP

Σ

elements a_i according to indexes i

$$I \subseteq \{1, \ldots, n\}$$

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

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

sum of the elements a_i

indexes i from Index-set I

DP

Σ

elements a_i according to indexes i

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

sum of the elements a_i

indexes i from Index-set 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

DP

Σ

elements a_i according to indexes i

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

sum of the elements a_i

indexes i from Index-set 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

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

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

$$S = \sum_{i \in I} a_i + \sum_{j \in J} a_j^2.$$

$$\sum_{i \in I} a_i = A \quad \text{and} \quad \sum_{i \in I} a_i^2 = B.$$

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{A}{3}.$$

$$\left(\sum_{x \in A} x \right) \bmod m = r\,.$$

DP

Practice

Theory Task T3. / 9 P

Let $A = [a_1, a_2, \ldots, a_n]$ be an array of non-negative integers. Given A and a non-negative integers $S \geq 0$, we want to determine whether S can be written as a (non-repeating) sum of elements of A, where we are allowed to take the square of elements. Formally, we want to determine if there exist $I, J \subseteq \{1, 2, \ldots, n\}$ with $I \cap J = \emptyset$, $I \cup J = \{1, 2, \ldots, n\}$ such that:

$$S = \sum_{i \in I} a_i + \sum_{j \in J} a_j^2.$$

Provide a *dynamic programming* algorithm that outputs `True` if this is possible, and `False` otherwise. For example,

- The inputs $A = [2, 4, 4]$, $S = 34$ should result in `True`, since $34 = 2 + 4^2 + 4^2$.
- The inputs $A = [2, 4, 4]$, $S = 35$ should result in `False`.
- The inputs $A = [2, 4, 3, 22]$, $S = 21$ should result in `False`.

In order to obtain full points, your algorithm should run in time $O(n^2 \cdot S)$. Address the following aspects of 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 co
   Specify the base cases, i.e., the entries that do not
3) *Calculation order*: In which order can entries be c
   entry have been determined in previous steps?
4) *Extracting the solution*: How can the final solution b
5) *Running time*: What is the running time of your alg
   of n and S, and justify your answer.

Theory Task T3. / 8 P

An array of non-negative integers $A = [a_1, \ldots, a_n]$ is called *summy* if and only if, for all $i \in \{2, \ldots, n\}$, there exists a (possibly empty) set $I \subseteq \{1, \ldots, i-1\}$ such that $a_i = \sum_{j \in I} a_j$. In other terms, every integer in the array except the first one must be the sum of (distinct) integers that precede it in the array.

For example,

- The array $[2, 2, 4, 6, 0, 12]$ is summy, because $2 = 2$, $4 = 2 + 2$, $6 = 2 + 4$, $12 = 2 + 4 + 6$.
- The array $[2, 2, 4, 6, 0, 13]$ is not summy, since 13 can not be written as a sum of integers from $\{2, 2, 4, 6, 0\}$.

Provide a *dynamic programming* algorithm that, given an array A of length n, returns `True` if the array is summy, and `False` otherwise. In order to obtain full points, your algorithm should have an $O(n \cdot \max A)$ runtime (where $\max A$ means the maximum value of entries in A). 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) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n and $\max A$, and justify your answer.

Theory Task T3. / 9 P

You are given an array of n natural numbers $a_1, \ldots, a_n \in \mathbb{N}$, and two natural numbers $A, B \in \mathbb{N}$. You want to determine whether there is a subset $I \subseteq \{1, \ldots, n\}$ satisfying

$$\sum_{i \in I} a_i = A \quad \text{and} \quad \sum_{i \in I} a_i^2 = B.$$

For example,

- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1, 4, 5, 3]$, $A = 8$ and $B = 30$ is *yes* because the set of indices $I = \{1, 4, 6\}$, which corresponds to $(a_i)_{i \leq I} = [2, 1, 5]$, yields the *sum* $2 + 1 + 5 = 8$ and the *sum-of-squares* $2^2 + 1^2 + 5^2 = 30$.
- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1]$, $A = 6$ and $B = 15$ is *no*.

Provide a *dynamic programming* algorithm that determines whether such a subset I exists. In order to get full points, your algorithm should have an $O(n \cdot A \cdot B)$ runtime. 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 ?

be com
lo not d
es be co
s ?
lution be
our algo

Theory Task T3. / 9 P

Let m, r be two integers satisfying $m \geq 2$ and $0 \leq r < m$. We say that a finite set $A \subset \mathbb{N}$ of natural numbers is (m, r)-aligned if

$$\left(\sum_{x \in A} x\right) \bmod m = r\,.$$

Note that for $A = \varnothing$, we adopt the convention that $\sum_{x \in A} x = 0$. Hence, the empty set is $(m, 0)$-aligned for every $m > 0$.

Given three integers m, r, n such that $0 \leq r < m < n$ and $m \geq 2$, we would like to determine the number of subsets of $\{1, 2, \ldots, n\}$ which are (m, r)-aligned.

For example,

- If $r = 1$, $m = 2$ and $n = 3$, the subsets of $\{1, 2, 3\}$ that are $(3, 1)$-aligned are $\{1\}$, $\{3\}$, $\{1, 2\}$ and $\{2, 3\}$. Hence, the answer is 4.

Provide a *dynamic programming* algorithm that solves the problem. In order to get full points, your algorithm should have an $O(n \cdot m)$ runtime. 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) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n, m and r, and justify your answer.

heory Task T3. / 9 P

ou are given an array of n natural numbers $a_1, \ldots, a_n \in \mathbb{N}$ summing to $A := \sum_{i=1}^{n} a_i$, which is a ultiple of 3. You want to determine whether it is possible to partition $\{1, \ldots, n\}$ into three disjoint bsets I, J, K such that the corresponding elements of the array yield the same sum, i.e.

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{A}{3}.$$

ote that I, J, K form a partition of $\{1, \ldots, n\}$ if and only if $I \cap J = I \cap K = J \cap K = \emptyset$ and $\cup J \cup K = \{1, \ldots, n\}$.

or example, the answer for the input $[2, 4, 8, 1, 4, 5, 3]$ is *yes*, because there is the partition $\{3, 4\}$, $2, 6\}$, $\{1, 5, 7\}$ (corresponding to the subarrays $[8, 1]$, $[4, 5]$, $[2, 4, 3]$, which are all summing to 9). n the other hand, the answer for the input $[3, 2, 5, 2]$ is *no*.

rovide a *dynamic programming* algorithm that determines whether such a partition exists. Your lgorithm should have an $\mathcal{O}(nA^2)$ runtime to get full points. Address the following aspects in your olution:

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

n an entry be computed from the values of other entries? ries that do not depend on others.

can entries be computed so that values needed for each vious steps?

ıe final solution be extracted once the table has been filled?

g time of your algorithm? Provide it in Θ-notation in terms er.

Last Session

Organization

- Last session on monday 16 Dec
  - Exam Preparation Session
    - Exam tipps, lernphase tipps, mock exam
  - Recap topics
    - The rest will be covered during mock exam
  - Semester-end celebration !!!

Questions

Feedbacks , Recommendations

Nil Ozer