Graph Theory

Dr. Jens Dörpinghaus

doerpinghaus@uni-koblenz.de

Mathematisches Institut
Universität Koblenz-Landau
Campus Koblenz

Federal Institute for Vocational Education and Training (BIBB)
Robert-Schuman-Platz 3
53175 Bonn

Summer 2022

- In today's world, algorithms are becoming more and more important in our everyday lives.
- Whenever a computer accomplishes a task, the instruction of how to execute this task has to be implemented as an algorithm.

Definition 2.1

An *algorithm* is a sequence of instructions. It is typically used to solve a problem or a class of problems or to execute a calculation on a computer. The number of instructions has to be finite and, in the general understanding, an algorithm has to terminate after a finite number of steps. An algorithm gets a possibly empty input and always produces a non-empty output. Each action to be carried out is definite and unambiguously defined.

- We will see that given a problem there is in general more than one way, and therefore more than one algorithm, to solve this problem.
- Finding an algorithm with the *smallest possible running time* is one of the main goals in algorithm design and complexity theory, *minimizing the needed space* a second.
- Based upon that, we can classify given problems by the running time an algorithm solving this problem must at least have.
- Further, we are interested in classifying problems by their *complexity*.
- Thus we need a kind of measure for the running time.

- The Big O notation was invented by Paul Bachmann 1894 and Edmund Landau 1909.
- It is used to describe the limiting behavior or the growth rate of a function.
- In computer science, it is commonly used to measure the running time of an algorithm and to classify algorithms by their running time.
- We will start with the formal definition before we take a look at some examples.

Definition 2.2

Let $f : \mathbb{N}_0 \mapsto \mathbb{N}_0$ be any function, where $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ is the set of all natural numbers with 0 included. We define

$$\mathcal{O}(f) = \{g : \mathbb{N}_0 \mapsto \mathbb{N}_0 \mid \exists c, n_0 \in \mathbb{N}\ \forall n > n_0 : g(n) \leq c \cdot f(n)\} \tag{1}$$

- Roughly speaking, this means that $\mathcal{O}(f)$ is the set of all functions that do not grow faster than f.

Example 2.3

For example, we have that $4n \in \mathcal{O}(10n)$ because we get that $4n \leq c \cdot 10n$ holds for all $n > n_0$ if we choose e.g. $c = 1$ and $n_0 = 0$.
We therefore say $4n$ is in the order of $10n$.
But contrariwise, we also have $10n \in \mathcal{O}(4n)$ because it also holds that $10n \leq c \cdot 4n$ for all $n > n_0$ if we chose e.g. $c = 5$ and $n_0 = 0$.

- As we see, functions that only differ in a constant factor are of the same order.
- Consequently, we write $4n \in \mathcal{O}(n)$ and $10n \in \mathcal{O}(n)$.
- Also the addition of a constant value k does not affect the order of a function.
- Hence, also $4n + k \in \mathcal{O}(n)$.

So, what does affect the order of a function?

Example 2.4

For example, we have $g(n) = 4n^2 \notin \mathcal{O}(n)$.
The question is if we can find two constant values $c, n_o \in \mathbb{N}$ such that $4n^2 \leq cn$ for all $n > n_0$.
We can chose c arbitrarily large.
But no matter how large we chose c, we always get that $4c^2 > c \cdot c$ and also $4n^2 > c \cdot n$ for all $n > c$.
We are therefore unable to find a $n_0 \in \mathbb{N}$ such that the inequality from Definition 2.2 holds for $g(n) = 4n^2$ and $f(n) = n$.
On the other hand, it is easy to find such values if we chose $f(n) = n^2$.
Hence, $4n^2 \in \mathcal{O}(n^2)$.

As an overview, Table 1 shows the hierarchy of the complexity classes.

Complexity class	Description
$\mathcal{O}(1)$	Constant functions
$\mathcal{O}(\log(n))$	Logarithmic functions
$\mathcal{O}(n)$	Linear functions
$\mathcal{O}(n \cdot \log(n))$	Linear-logarithmic functions
$\mathcal{O}(n^2)$	Quadratic functions
$\mathcal{O}(n^k)$	polynomial functions (with $k \in \mathbb{N}$)
$\mathcal{O}(k^n)$	exponential functions (with $k \in \mathbb{N}$)

Table 1: The hierarchy of the complexity classes.

In practice, the running time of an algorithm is first of all determined by the number of loops and nested loops.

We consider algorithm A with running time $T_A(n)$.

$$\mathcal{O}(k \cdot T_A(n)) = \mathcal{O}(T_A(n)) \quad \text{for a constant value } k \in \mathbb{N}. \tag{2}$$
$$\mathcal{O}(T_A(n) + k) = \mathcal{O}(T_A(n)) \quad \text{for a constant value } k \in \mathbb{N}. \tag{3}$$

Example 2.5

We specify algorithm A's running time to

$$T_A(n) = n^2 + 5n = \mathcal{O}(n^2)$$

and consider a second algorithm B with running time

$$T_B(n) = 4n = \mathcal{O}(n)$$

We now assume that we want to execute these algorithms consecutively.
To determine the accumulated asymptotic running time of both algorithms, we first observe that the accumulated running time is simply the sum of their separate running times. Now, we can calculate the desired term $\mathcal{O}(T_A(n) + T_B(n))$ in the following way.

$$\begin{aligned}\mathcal{O}(T_A(n)) + \mathcal{O}(T_B(n)) &= \mathcal{O}(T_A(n) + T_B(n)) \\ &= \mathcal{O}(n^2 + 5n + 4n) = \mathcal{O}(n^2 + 9n) = \mathcal{O}(n^2) \\ &= \mathcal{O}(\max\{n^2, n\})\end{aligned}$$

From this example, we derive the **addition rule** for the Big O notation:

$$\mathcal{O}(T_A(n)) + \mathcal{O}(T_B(n)) = \mathcal{O}(\max\{T_A(n), T_B(n)\}) \tag{4}$$

Last, we have to bring in a **multiplication rule**:

$$\mathcal{O}(T_A(n)) \cdot \mathcal{O}(T_B(n)) = \mathcal{O}(T_A(n) \cdot T_B(n)) \tag{5}$$

Example 2.6

This is important when handling nested loops. We consider algorithm A and B from above. As we are interested in the multiplied asymptotic running time we have to calculate the term $\mathcal{O}(T_A(n)) \cdot \mathcal{O}(T_B(n))$. For this, we use the following **multiplication rule**:

$$\mathcal{O}(T_A(n)) \cdot \mathcal{O}(T_B(n)) = \mathcal{O}(T_A(n) \cdot T_B(n)) \tag{6}$$

and get

$$\begin{aligned}
\mathcal{O}(T_A(n)) \cdot \mathcal{O}(T_B(n)) &= \mathcal{O}(T_A(n) \cdot T_B(n)) \\
&= \mathcal{O}((n^2 + 5n) \cdot 4n) \\
&= \mathcal{O}(4n^3 + 20n^2) \\
&= \mathcal{O}(n^3).
\end{aligned}$$

- After these rather theoretical definitions, the next section comes up with some examples that show how to apply these rules and all the previous considerations to determine the asymptotic running time of an algorithm.
- For simple operations like basic arithmetic operations, the assignment of a certain value to a variable or an I/O-command, we already mentioned that the running time is constant and hence equals $\mathcal{O}(1)$.
- But things get more complicated if the algorithm A we want to analyze contains `if`-statements or loops and nested loops. Or if algorithm A calls another algorithm B.

- If A is a basic arithmetic operation, a value assignment or an I/O command it takes constant time:

$$\mathcal{O}(T_A) = \mathcal{O}(1). \tag{7}$$

- If A is a sequence of operations $A_1, \ldots, A_k$, we have to apply the addition rule:

$$\mathcal{O}(T_A) = \mathcal{O}(T_{A_1}) + \cdots + \mathcal{O}(T_{A_1}) \tag{8}$$

- If A is an `if`-command
  - "`if(C): B`":

$$\mathcal{O}(T_A) = \mathcal{O}(T_C) + \mathcal{O}(T_B) \tag{9}$$

- "`if(C): B, else D`":

$$\mathcal{O}(T_A) = \mathcal{O}(T_C) + \max\{\mathcal{O}(T_B), \mathcal{O}(T_D)\} \tag{10}$$

- If A is a `while`-loop "`while(C):B`":

$$\mathcal{O}(T_A) = (\mathcal{O}(T_C) + \mathcal{O}(T_B)) \cdot \#\mathit{iterations} \tag{11}$$

- If A is a `for`-loop "`for(...):B`":

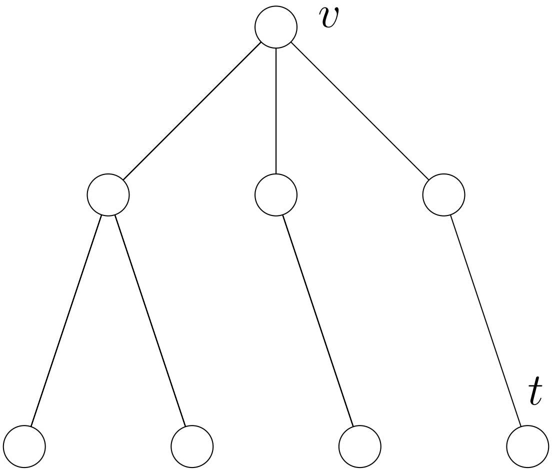
$$\mathcal{O}(T_A) = \mathcal{O}(T_B) \cdot \#\mathit{iterations} \tag{12}$$

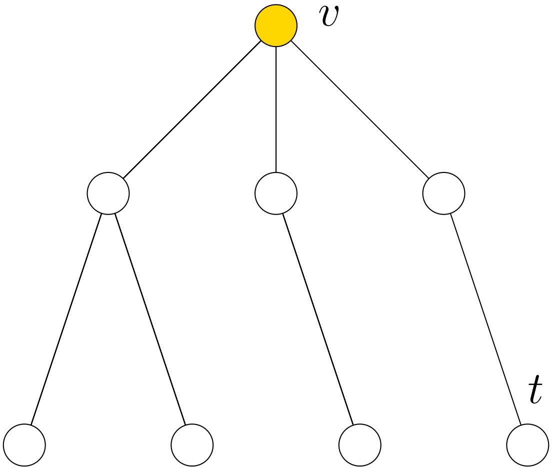
- Sometimes it is necessary to process every node in a graph (search, check, update, ...).
- This is called *graph traversal*.
- Two widely used approaches are Breadth-first search (BFS) and Depth-first search (DFS).

Algorithm 1 BFS

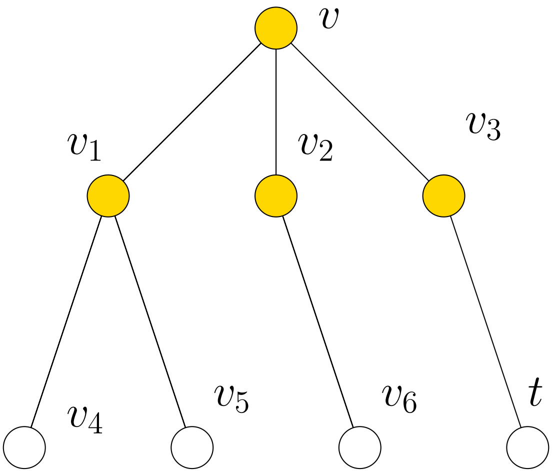
```
Ensure: Graph $G$, start node $v$, target node $t$
Ensure: Queue $Q$
 1: $visited.add(v)$
 2: $Q.enqueue(v)$
 3: while $Q \neq \emptyset$ do
 4:     $w := Q.dequeue()$
 5:     if $w = t$ then
 6:         $return(v)$
 7:     for $u \in N(w)$ do
 8:         if $u \notin visited$ then
 9:             $visited.add(u)$
10:             $Q.enqueue(u)$
```

- This approach uses a queue (FIFO) as data structure.

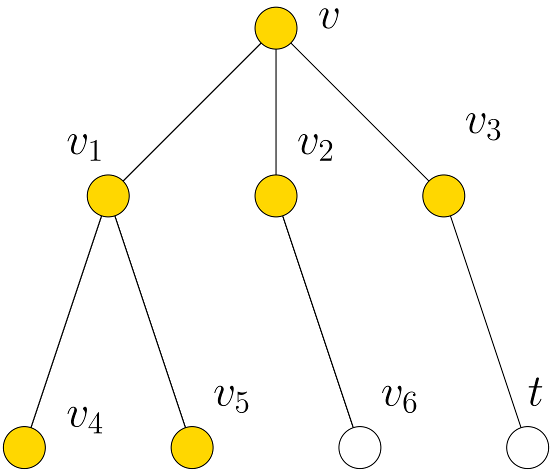

v
t



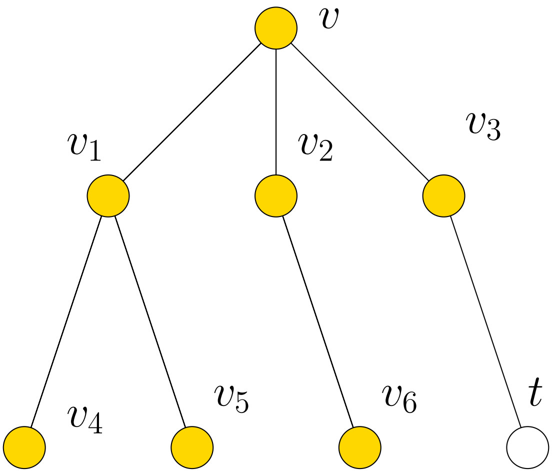
$$Q = \{v\}$$



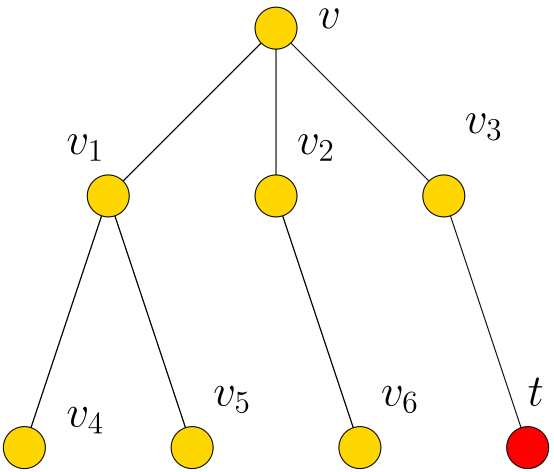
$$Q = \{v_1, v_2, v_3\}$$



$$Q = \{v_2, v_3, v_4, v_5\}$$



$$Q = \{v_3, v_4, v_5, v_6\}$$



$$Q = \{v_4, v_5, v_6\}$$

Runtime:

- While-Loop [3]: maximum $|V|$
- For Loop [7]: maximum $|E|$
- Average Runtime: $\mathcal{O}(|V| + |E|)$

Runtime:

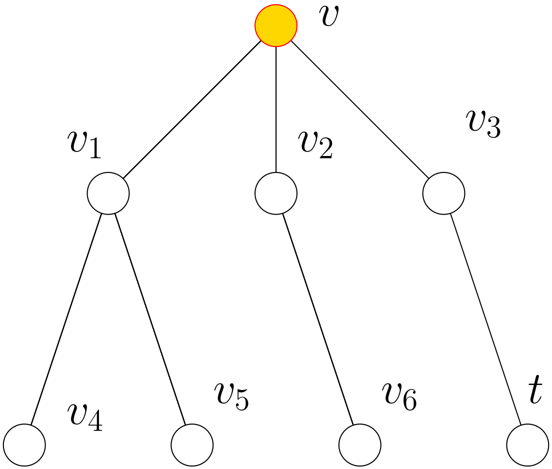
- While-Loop [3]: maximum $|V|$
- For Loop [7]: maximum $|E|$
- Average Runtime: $\mathcal{O}(|V| + |E|)$

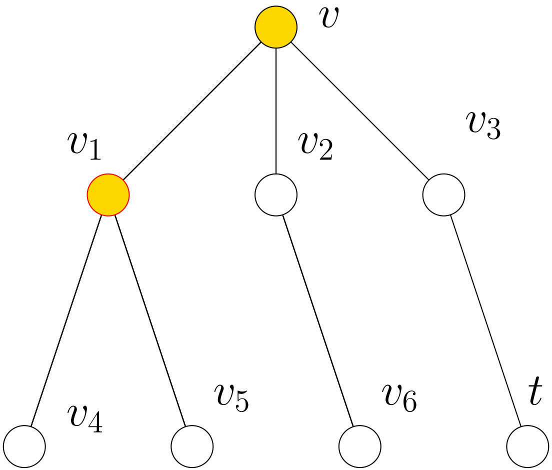
Applications:

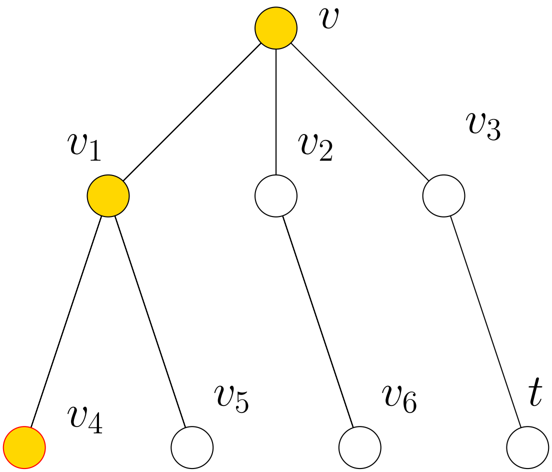
- BFS can be used to find shortest paths.
- Test the bipartiteness of a graph.

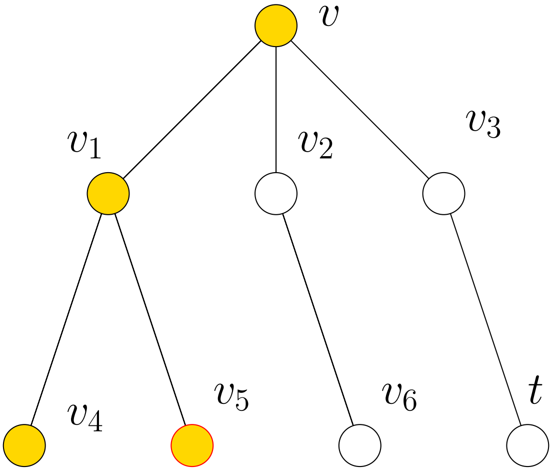
Algorithm 2 Recursive DFS

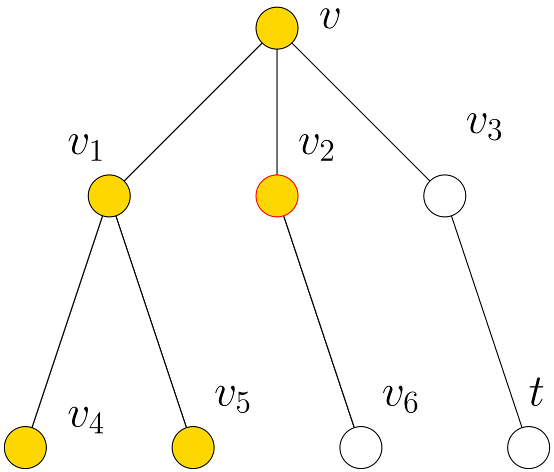
```
1: Function{DFS}{$G$,$v$)}
2: $visited.add(v)$
3: for $w \in N(v)$ do
4:     if $w \notin visted$ then
5:         DFS($G, w$)
6: EndFunction
```

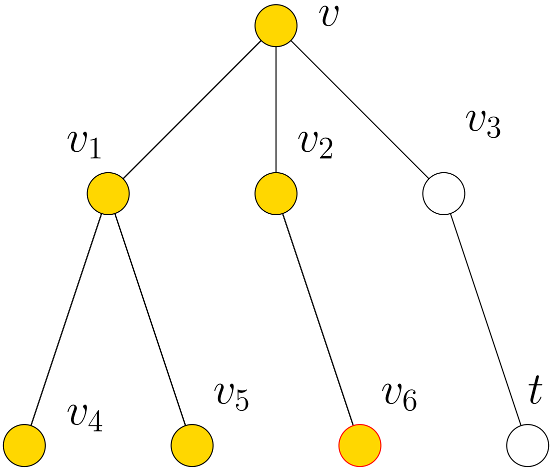

v
v_1
v_2
v_3
v_4
v_5
v_6
t

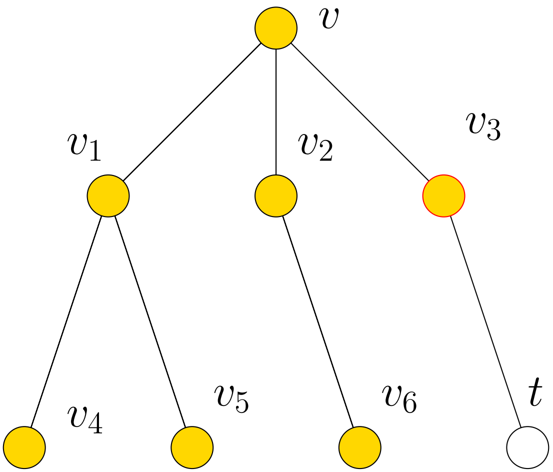

v
v_1
v_2
v_3
v_4
v_5
v_6
t

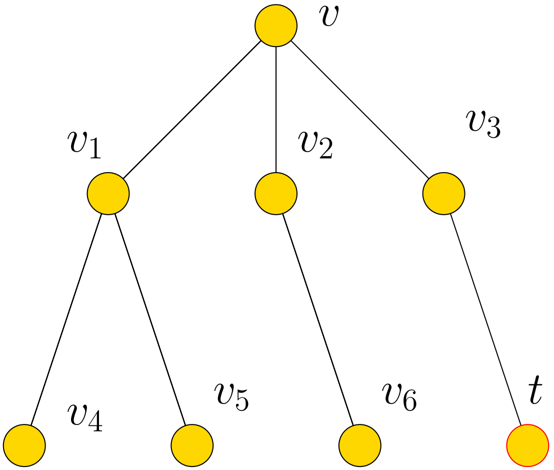

v
v_1
v_2
v_3
v_4
v_5
v_6
t


v
v_1
v_2
v_3
v_4
v_5
v_6
t


v
v_1
v_2
v_3
v_4
v_5
v_6
t


v
v_1
v_2
v_3
v_4
v_5
v_6
t


v
v_1
v_2
v_3
v_4
v_5
v_6
t


v
v_1
v_2
v_3
v_4
v_5
v_6
t

Runtime:

- Exercise.

Runtime:

- Exercise.

Applications:

- DFS can be used to check if a graph is connected,
- and find (strongly) connected components.
- Check if a graph is planar.
- Some sorting approaches use DFS.
- ...

- Based upon that, we have the needed tool to classify given problems by the running time an algorithm solving this problem must at least have.
- Further, we are interested in classifying problems by their *complexity*. For this, we have the classification scheme of the classes $\mathcal{P}$ and $\mathcal{NP}$.

- The class $\mathcal{P}$ contains all the (decision) problems that can be solved in polynomial time.
- These problems are also considered as the “easy” problems.
- On the other hand, there is the class $\mathcal{NP}$ that contains $\mathcal{P}$.
- $\mathcal{NP}$ stands for “**n**on-deterministic **p**olynomial”.
- This means that every problem K in this class has the following property:

  Given the (decision) problem K and an alleged solution S, we can check in polynomial time whether S indeed solves K.

- Additionally, there is the class of the $\mathcal{NP}$*-hard* problems.
- A problem K_1 is said to be NP-hard if any problem K_2 in the class $\mathcal{NP}$ can be *reduced* to K_1 in polynomial time.
- Reducing a problem K_2 to a problem K_1 means to find a way to transform a solution of K_1 into a solution of K_2.
- Once this transformation is found, it follows that we can solve K_2 whenever we are able to solve K_1.
- Because, if we can not solve K_2 directly, we can first solve K_1 and then transform this solution into a solution of K_2.
- In other words, this means that K_2 is not harder to solve than K_1 or, the other way around, K_1 is at least as hard to solve as K_2.
- Thus, a problem in the class of the $\mathcal{NP}$-hard problems is at least as hard to solve as any other problem in the class $\mathcal{NP}$.

- Finally, a problem is called $\mathcal{NP}$*-complete* if it is both in $\mathcal{NP}$ and $\mathcal{NP}$-hard.
- Figure 1 shows the hierarchy of the different complexity classes.
- In the following chapters, but also in the exercises, we will consider an example to make things more clear.

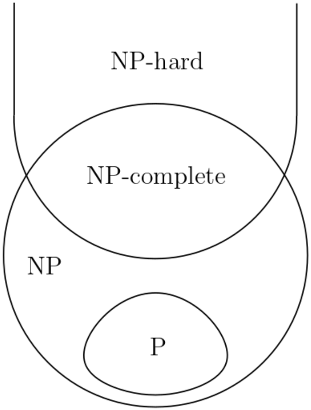


Figure 1: The hierachy of the different complexity classes.