Handout

The Complexity Class P

Proseminar Theoretische Informatik WiSe 2020-21
Institut für Informatik
Freie Universität Berlin

valentinpi

8. Dezember 2020
(neueste Version)

Abstract

This handout gives an introduction to complexity theory and the complexity class P. It presents the most important definitions, discusses why polynomial computation time is of practical interest and then presents a handful of algorithmic problems and shows that they are in P.

Computability and Complexity Coming from the bachelor class "Foundations of Theoretical Computer Science", complexity theory has not been formally introduced yet.

- *Computability theory* discusses the *possibilities* of computational models
- *Complexity theory*, however, discusses the *efficiency* of computation

Efficiency can be viewed from multiple perspectives. For instance, time, space (memory usage) or encoding efficiency. Time might be the number of steps that the computational model has to take in order to decide a problem. For instance the number of left or right moves on a TM.

Definition 1. Let $f\colon \mathbb{N} \to \mathbb{R}^+$ be a function. The complexity class *TIME* of f is defined as follows:

$$\mathrm{TIME}(f(n)) \coloneqq \{\ L \mid \text{There is a deterministic TM that decides } L \text{ in } \mathcal{O}(f(n)) \text{ time for an input of size } n\ \}$$

We only consider decidable problems, as Turing machines of semi-decidable problems may not halt. There are problems that might require Turing machines running in exponential time or *polynomial time*.

- All versions of Turing machines (single-tape, multi-tape, multi-head, ...) are computationally, but also *polynomially equivalent*
- Polynomial differences are important, but in some cases avoiding exponential growth outweighs these differences, we merely focus on one part of the problem here

$\Rightarrow$ A polynomial time of e.g. n^{100} might not be practical, but focusing on finding polynomial solutions for exponential problems has proven to be useful for further reduction

- Most exponential approaches to problem solving involve iterating through exponentially many tests for a solution, also called *brute-force*

The Class P

Definition 2. P is the class of languages decidable in polynomial time on a deterministic single-tape TM.

$$\mathrm{P} \coloneqq \bigcup_{k \in \mathbb{N}} \mathrm{TIME}(n^k)$$

- P is invariant in terms of the Turing machine version which solves a problem in polynomial time
- Algorithms that utilize certain data structures (integers etc.) must have a representation that allows for polynomial reading, editing time (polynomial time encodings)

The PATH Problem We now take a look at the problem *PATH*. Let $G = (V, E)$ be a directed graph. Every edge $(u, v) \in E$ is an ordered touple representing a direction from one node to another. For two nodes s, t, we want to find an algorithm which decides whether a path, i.e. a sequence of edges that connect these points, exist.

$$\text{PATH} := \{\ \langle G, s, t\rangle \mid G \text{ is a directed graph and there is a path between nodes } s \text{ and } t\ \}$$

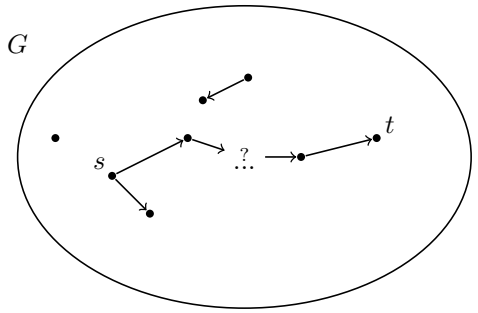

Figure 1: Small illustration of the PATH problem

Theorem 1. PATH $\in$ P

Proof. To prove the theorem, we need to find an algorithm that decides PATH in polynomial time.

To brute-force, we can attempt to calculate every possible path inside of the graph and then check for one that connects s and t. Any path is at most $m := |V|$ nodes long. If all nodes are connected, as in complete graphs, we can choose one of the m nodes for each of the $m - 1$ connections made. We do not always know whether the path satisfies the search. Therefore the runtime of this algorithm is $\mathcal{O}(m^m)$ and exponential.

Alternatively one can use a *breadth-first-search (BFS)* approach: We begin by placing a mark on the first node s, then mark its neighbors, then continue with those neighbors' neighbors and so on. By that, we ultimately mark each node that we can reach from s. If there are no further possible marks to be made our search is finished and we can check if t is marked, which leads to the following Turing machine M that decides PATH:

Input: $\langle G, s, t\rangle$, where G is a directed graph with nodes s, t.

Function:

1: Mark s
2: **while** nodes got marked **do**
3: Search all edges in E. If an edge (u, v) is found with u marked and v not marked, mark v.

Step 1 runs in $\mathcal{O}(1)$ time. The loop of 2 and 3 runs in $\mathcal{O}(m)$ time, since we can only mark as many times as we have nodes. For implementations, efficient data structures can store unmarked nodes and edges and thereby decrease runtime, but that is of no interest here. The runtime is $\mathcal{O}(m + 2)$, which is a polynomial runtime and therefore PATH $\in$ P. ■

As a practical example, consider BFS for finding a route between two points in a city.

Relatively Prime Integers If two natural integers have the greatest common divisor (GCD) 1, they are called *relatively prime*. This leads us to the next algorithmic problem.

$$\text{RELPRIME} \coloneqq \{ \langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime} \}$$

Theorem 2. RELPRIME $\in$ P

Proof. We can try to compute all possible divisors of x and y. If the numbers are encoded in binary, for instance, with the MSB (Most Significant Bit) of k, we would need to compute the divisibility test for at least 2^k values. This also applies to any other base b. The brute-force approach is exponential and therefore not feasible.

The *Euclidian Algorithm* gives us a polynomial solution to this problem. The correctness will not be proven here, but the algorithm complexity will be analyzed. The idea is that for any $x, y \in \mathbb{N}$ it holds that $\gcd(x, y) = \gcd(y, x \bmod y)$ and when y reaches zero, the GCD is calculated. Using the euclidian algorithm, we directly construct a Turing machine M which decides RELPRIME:

Input: $\langle x, y \rangle$, where $x, y \in \mathbb{N}$.

Function:

1: **while** $y > 0$ **do**
2: $\quad x \leftarrow x \bmod y$
3: $\quad$ Swap x and y
4: If $x = 1$, accept. Otherwise, reject.

Step 4 runs in $\mathcal{O}(1)$ time. For the loop in 1 with steps 2, 3, we are going to show that each iteration cuts the value of x in at least a half. If $x < y$ initially, the algorithm essentially swaps x and y, so we start off with $x > y$ after this possible initial swap. After 2 is executed, it holds $x < y$ and after the swap in 3 $x > y$ again. We take a look at $\frac{x}{2}$:
First case: $\frac{x}{2} \geq y$. Then by the laws of the modulo operation, $x \bmod y < y \leq \frac{x}{2}$ and x is cut by at least half.
Second case: $\frac{x}{2} < y$. Then either $x = y$ or $2y > x > y$. Either way $x \bmod y = x - y$ per application of the modulo, but also $x - y < \frac{x}{2}$. Therefore, the value of x gets cut by at least half.
Since x and y get swapped in each iteration, both of them get reduced by at least half every time. It is not important which of the variables gets cut first and by that the maximum number of iterations is $\min(2 \log_2 x, 2 \log_2 y)$, which is polynomial due to $\mathcal{O}(n)$ being an upper bound of these complexities. ■

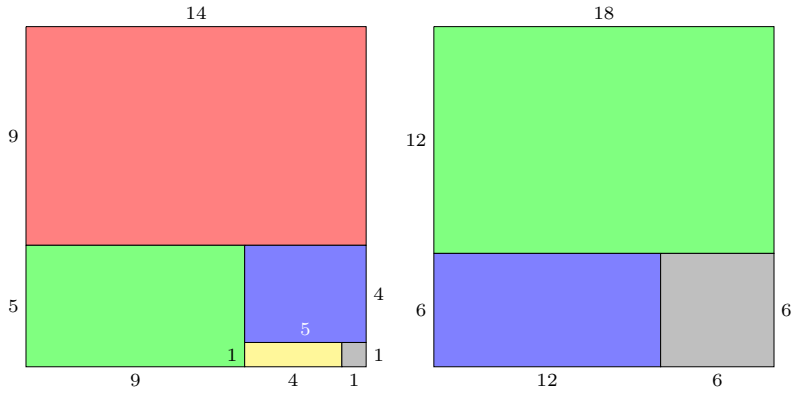

Figure 2: Rectangle visualisation of the Euclidian algorithm for $\gcd(14, 9)$ and $\gcd(18, 12)$

Context-Free Languages It has been shown that every context free language is decidable (CYK-algorithm). The following analysis shows that due to this method all of them are also in P.

Theorem 3. Every context-free language $L \in \mathrm{CFL}$ is in P.

Proof. Let G be a CNF grammar (Chomsky Normal Form) for L with variables V and start variable $S \in V$. G only consists of productions in the form $A \to a, A \to BC$ with $S \to \varepsilon$ allowed as an exception case. We use the CYK algorithm. The idea is to setup a table of grammar variables in the form factor $n \times n$. If $w = \sigma_1 ... \sigma_n \neq \varepsilon$ is the word, then each entry holds $table(i,j) = \{A \in V | A \to^* \sigma_i ... \sigma_j\}$. So if $S \in table(1,n)$, the word is derivable from G.
The following Turing machine for L decides it in polynomial time using the CYK algorithm:

Input: $w \in \Sigma^*$, for $w \neq \varepsilon$ write $w = \sigma_1 \sigma_2 ... \sigma_n$.

Function:

1: For $w = \varepsilon$, if $S \to \varepsilon$ is production, accept. Otherwise, reject.
2: **for** $i = 1$ to n **do**
3: **for** each variable A **do**
4: If $A \to \sigma_i$ is a rule, place A in $table(i,i)$.
5: **for** $l = 2$ to n **do** ▷ Substring length
6: **for** $i = 1$ to $n - l + 1$ **do** ▷ Starting position
7: $j \leftarrow i + l - 1$ ▷ End position
8: **for** $k = i$ to $j - 1$ **do** ▷ Split position
9: **for** each rule $A \to BC$ **do**
10: If $B \in table(i,k)$ and $C \in table(k+1,j)$, put A in $table(i,j)$.
11: If $S \in table(1,n)$, accept. Else, reject.

Steps 1 and 11 run in $\mathcal{O}(1)$ time. Let $v = |V|$ be the number of variables, then the steps 2, 3, 4 run in $nv \in \mathcal{O}(n)$ time, since v is a fixed constant for G. Stage 5 runs $n - 1 \in \mathcal{O}(n)$ times, stages 6 and 7 also, as well as stages 8-10. Stages 9-10 however, run v times. We conclude a runtime of $\mathcal{O}(n^3)$, since these loops are nested. Therefore the runtime is $\mathcal{O}(n^3)$, which is polynomial. ■

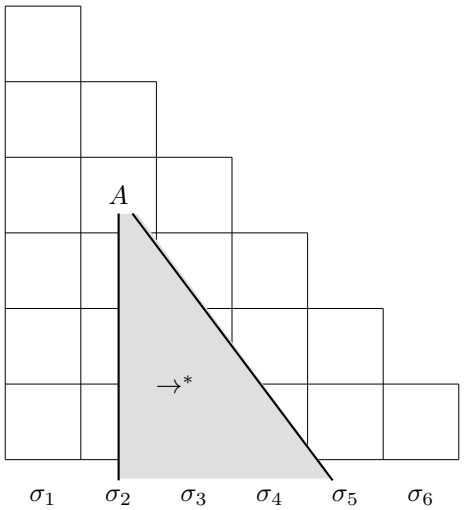

Figure 3: *table* and the concept of the CYK subproblems, $n = 6$ for illustration

Conclusion The landscape of decidable languages has changed since the last time we saw a diagram of it. This sketch shows the new classes and problems.

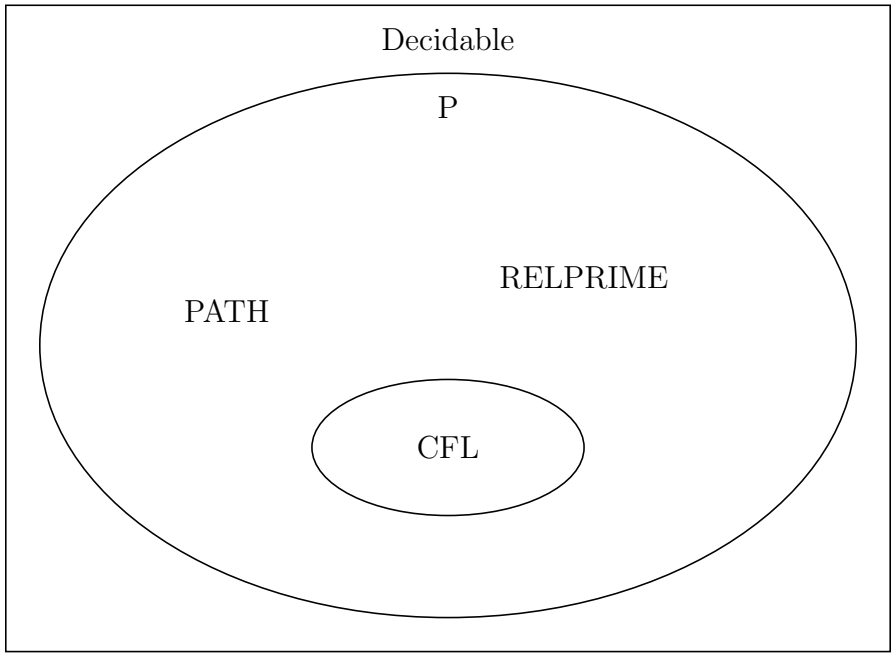

Figure 4: The landscape of decidable languages by now

References

[1] Michael Sipser. *Introduction to the Theory of Computation, Third Edition.* Cengage Learning, 2012.

[2] David Wees. Visualizing euclid's algorithm. https://www.geogebra.org/m/ztbesvsd, 11.11.2020, 21:30 (last visited).

All the illustrations were made by myself using *LaTex/Tikz*.
The webpage I linked was inspiration for the GCD visualisation.