Computational Theory and Complexity Theory

Prof. Dr.-Ing. Sebastian Schlesinger

Berlin School for Economics and Law

October 17, 2025

Word Problem

- Given a formal language L over an alphabet Σ and a word $w \in \Sigma^*$
- Decide whether $w \in L$
- Example: $L = \{a^n b^n | n \geq 0\}$, $w = aaabbb$
- $w \in L$?

Decidability and Semidecidability

- A language L is **decidable** if there exists a Turing machine that halts on every input and accepts exactly the words in L
- A language L is **semidecidable** if there exists a Turing machine that accepts exactly the words in L and may either reject or run forever on words not in L
- Every decidable language is also semidecidable
- Example: The Halting Problem is semidecidable but not decidable

Computability

- A function $f : \Sigma^* \to \Sigma^*$ is **computable** if there exists a Turing machine that, given any input $w \in \Sigma^*$, halts and outputs $f(w)$
- Example: The function that maps a binary number to its successor is computable
- Not all functions are computable (e.g., the Halting Problem)

Completeness and Decidability

The propositional logic is decidable. However, first-order logic is not decidable, but semi-decidable. However, first-order logic is complete, i.e., every valid formula can be proven. Hence, although there must be a proof for every valid formula, there is no algorithm that can find it for every formula in finite time.

Decidability of the Chomsky Hierarchy

- Type 3 (Regular Languages): Decidable
- Type 2 (Context-Free Languages): Decidable
- Type 1 (Context-Sensitive Languages): Decidable
- Type 0 (Recursively Enumerable Languages): Semi-decidable

Recursively Enumerable vs. Recursive Languages

- **Recursive Languages**: Languages for which there exists a Turing machine that halts on all inputs and decides membership.
- **Recursively Enumerable Languages**: Languages for which there exists a Turing machine that will enumerate all valid strings but may not halt on invalid strings.

Primitive Recursive Functions

- A class of functions that can be defined using basic functions (zero, successor, projection) and closed under composition and primitive recursion.
- All primitive recursive functions are total and computable.
- Example: Addition, multiplication, and factorial are primitive recursive functions.

Mu-Recursive Functions

- An extension of primitive recursive functions that includes the minimization operator.
- Mu-recursive functions can express a wider range of computations, including some that are not primitive recursive.
- All mu-recursive functions are computable, but not all are total.

Church-Turing Thesis

- The Church-Turing Thesis posits that any function that can be computed by an algorithm can be computed by a Turing machine.
- This thesis provides a foundation for the field of computability theory.
- It implies that Turing machines are a powerful model of computation, equivalent to other models (e.g., lambda calculus, recursive functions).

Computability vs. Complexity

- **Computability** is concerned with what can be computed (i.e., the existence of an algorithm).
- **Complexity** is concerned with how efficiently a problem can be solved (i.e., the resources required).

Complexity Classes

- **P**: Class of decision problems solvable by a deterministic Turing machine in polynomial time.
- **NP**: Class of decision problems solvable by a nondeterministic Turing machine in polynomial time. Equivalently, problems for which a given solution can be verified in polynomial time.
- **PSPACE**: Class of decision problems solvable by a Turing machine using polynomial space.
- **EXPTIME**: Class of decision problems solvable by a deterministic Turing machine in exponential time.

P vs NP Problem

- One of the most important open problems in computer science.
- Question: Is P equal to NP?
- If P = NP, then every problem for which a solution can be verified quickly can also be solved quickly.
- Most experts believe that P $\neq$ NP, but it remains unproven.

NP-Hard and NP-Complete

- A problem is **NP-hard** if every problem in NP can be reduced to it in polynomial time.
- A problem is **NP-complete** if it is both in NP and NP-hard.
- If any NP-complete problem can be solved in polynomial time, then P = NP.
- Examples of NP-complete problems: Traveling Salesman Problem, Boolean Satisfiability Problem (SAT), Knapsack Problem