Introduction to Computation Theory

Ronni Chang and Paige Zhu
Mentor: Zoe Xi

May 25, 2024

Abstract

Computation theory helps us understand which problems can be solved by algorithms and which cannot. This includes understanding concepts like decidable and undecidable problems, providing insight into the inherent limitations of computational systems.

In addition, algorithm analysis equips us with tools to analyze the efficiency of algorithms. The analysis assists us to identify the most efficient algorithms for solving problems and understanding the resources they require.

Computation theory also allows us to classify problems based on their computational difficulty. This classification helps in determining which problems are tractable (can be solved efficiently) and which are intractable (require impractical amounts of time/resources).

In this paper, we present fundamental concepts of automata, computability, and complexity theory. After defining Turing machines and exploring their variants, we shift toward time complexity analysis and explain big-O and small-o notation. Finally, we explore the importance of polynomial time algorithms and introduce complexity classes **P**, **NP**, and **NP-complete**.

1 Introduction

Computation Theory is the study of the fundamental principles underlying computation and the analysis of algorithms. It's often split into three parts: automata, computability, and complexity.

So what exactly is automata, computability, and complexity theory? Automata theory deals with abstract machines and formal languages, exploring the capabilities and limitations of computational models such as finite automata and pushdown automata. Computability theory investigates what problems can be solved algorithmically (i.e. with a Turing machine), focusing on the notion of computable functions and the halting problem. Complexity theory studies the resources required to solve computational problems, including time and space complexity, aiming to classify problems based on their inherent difficulty and identify efficient algorithms for solving them.

In general, computation theory aims to answer the question, *What are the fundamental capabilities and limitations of computers?*

In the following definitions, we will mostly follow Sipser [Sip96].

2 Preliminaries

Definition 2.1. *An **alphabet** is any non-empty finite set. Each item in the alphabet is known as a **symbol** of the alphabet. We use Σ to denote an alphabet. A **string** over Σ is a finite sequence of symbols from Σ, and a **language** over Σ is a set of strings.*

Definition 2.2. *Let A and B be languages. We can definition the following regular operations:*

- ***Union:*** $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
- ***Concatenation:*** $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$
- ***Star:*** $A^* = \{x_1 x_2 \cdots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

Definition 2.3. *A **finite automata** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ such that:*

1. *Q is a finite set of **states**,*
2. *Σ is the alphabet,*

3. $\delta : Q \times \Sigma \to Q$ *is the transition function,*

4. $q_0 \in Q$ *is the* ***start state****, and*

5. $F \subseteq Q$ *is the set of all accept states.*

3 Turing Machines

A **Turing machine**, or TM is one of the first models of our modern computer. It's very similar to **finite automata**, with one major difference: a TM uses an infinite tape with an unlimited memory. It uses a tape head to read and write symbols on the tape, as well as to move on the tape. The tape head will keep moving forever until it enters an accept or reject state, in which the accompanying will immediately halt.

Definition 3.1. *A* ***Turing machine*** *is a 7-tuple* $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ *where* Q, Σ, Γ *are all finite sets and*

1. Q *is the set of all states,*

2. Σ *is the input alphabet not containing the* ***blank symbol*** $\sqcup$*,*

3. Γ *is the tape alphabet, where* $\sqcup \in \Gamma$ *and* $\Sigma \subseteq \Gamma$*,*

4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *is the transition function,*

5. $q_0 \in Q$ *is the start state,*

6. $q_{accept} \in Q$ *is the accept state, and*

7. $q_{reject} \in Q$ *is the reject state, where* $q_{reject} \neq q_{accept}$*.*

The set of strings that a TM accepts is called the **language** of that TM, which is denoted as $L(TM)$. A TM is used to test membership in a language.

We can now define two new terms: **Turing-recognizable** and **Turing-decidable** to describe these languages.

Definition 3.2. *A language is* ***Turing-recognizable*** *if there is a Turing machine that can correctly identify and accept any string that belongs to the language.*

Definition 3.3. *A language is* ***Turing-decidable*** *or* ***decidable*** *if there exists a Turing machine that, for any input string, will always halt and either accept or reject that input.*

By this definition, we can also define an undecidable language.

Definition 3.4. *A language is* ***undecidable*** *if there does not exist a Turing machine that decides this language.*

To better understand how a Turing Machine works, let's try an example problem.

Example 3.5. We construct as follows a TM M that decides the language $A = \{0^{2^n} \mid n \geq 0\}$. We can give a high level description of M as follows:

$M =$"On input string ω:

- Go from left to right and cross off every other 0.
- If there was only 1 zero in the first step, *accept.*
- If there were more than 1 zero in the first step and there were an odd number of zeros, *reject.*
- Go back to the very left of the tape.
- Go back to the first step."

Every time our machine iterates, the number of 0s halves until there are an odd number of 0s left. When this happens, our machine checks how many 0s are left. If there is only one 0, we know that the original number of 0s must have been a power of 2, so the machine *accepts*. Otherwise, we know that

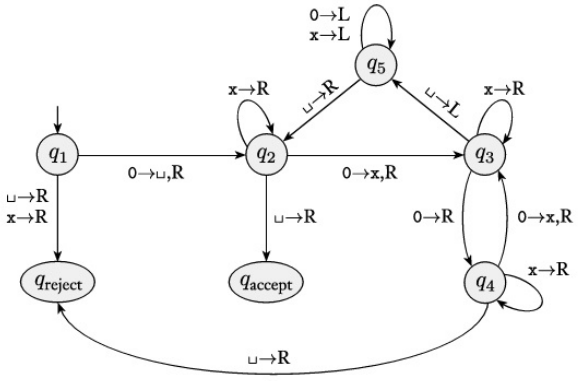
the original number could not have been a power of 2, so the machine *rejects.*

Let's now give a more formal description of this TM using our definition from before.

M :

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_{accept}, q_{reject}\}$
- $\Sigma = \{0\}$
- $\Gamma = \{0, x, \sqcup\}$
- δ as pictured in Figure 3
- The start state is q_1, the accept state is q_{accept}, and the reject state is q_{reject}.

Figure 1: State diagram for TM M (Sipser, pg. 172), and visualization of δ



This type of TM is termed a **single-tape Turing machine**. While there are other variations like **multi-tape Turing machines**, it's worth noting that single-tape TMs can simulate multi-tape TMs. Therefore, multi-tape TMs do not provide any additional computational power. Furthermore, there are **deterministic Turing machines** (DTMs) and **nondeterministic Turing machines** (NTMs). Each variant offers distinct capabilities, but NTMs cannot generate consistent results and thus are not a realistic computational model in practical problem solving. It is a big open problem whether NTMs truly provide more computational power than DTMs.

4 Time Complexity: Big-O $O(n)$ and Small-O $o(n)$ Notation

A computational apparatus will be used to perform algorithms for problem solving. In real world, computational resources are limited, so efficient algorithms are always desired. In this section, we analyze what it means for an algorithm to be efficient.

The concept of algorithmic efficiency is based on an algorithm's **running time** or **time complexity**, which is the time an algorithm takes to solve a problem.

Definition 4.1. *Let M be a deterministic TM that halts on all inputs. The **time complexity** of M is the function f, where $f : \mathcal{N} \to \mathcal{N}$, where $f(n)$ is the maximum number of steps M uses on any input of length n.*

When we are looking at the time complexity of different algorithms, the input could be very different. To simplify this down, and to be able to compare the time complexities of different algorithms, we can just compute the time complexity of an algorithm based purely on the length of the string representing the input.

Finding the exact amount of time that an algorithm spends is frequently a long and complicated process, so we often choose to estimate the complexity. Furthermore, we do not really care too much about the little details - as the algorithm grows with the size of the input, the smaller values (e.g., x^2 in $x^3 + x^2$) become nearly obsolete when the input size goes towards infinity. This is why we use **asymptotic bounds**. There are two ways to measure complexity that we will cover in this section: big O and small o.

Understanding algorithm time complexity is crucial for predicting performance as input sizes grow. Big O notation gives an upper bound on growth rates, while small o provides a stricter upper bound. You can think of big O notation as a "less than or equal to" function, while small o is more like "less than".

Definition 4.2. *Let f and g be functions $f, g : \mathcal{N} \to \mathcal{R}^+$. We can define* ***f(n) = O(g(n))*** *if positive integers c and n_0 exist such that for every integer $n \geq n_0$,*

$$f(n) \leq cg(n).$$

We say that $g(n)$ is the ***asymptotic upper bound*** *of $f(n)$.*

The following example of big O notation provides better understanding of the concept.

Example 4.3. Supposed that we are given a function $f(n) = 10n^3 + 7n^2 + 2n + 4$. By looking at only the highest order of this function and ignoring its coefficient, we can say that $f(n) = O(n^3)$.

Now, let us move onto the definition of small o.

Definition 4.4. *Let f and g be functions $f, g : \mathcal{N} \to \mathcal{R}^+$. Say that* ***f(n) = o(g(n))*** *if*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

.

Namely, $f(n) = o(g(n))$ means that for any real number $c > 0$, a number n_0 exists, where $f(n) < cg(n)$ for all $n \geq n_0$.

As usual, we use an example to better understand the concept of small o.

Example 4.5. Consider $n^2 = o(n^3)$. To show that $n^2 = o(n^3)$, we need to prove that $\lim_{n\to\infty} \frac{n^2}{n^3} = 0$. The evaluation of the limit is:

$$\lim_{n\to\infty} \frac{n^2}{n^3} = \lim_{n\to\infty} \frac{1}{n} = 0.$$

Since the limit is 0, this implies that n^2 grows asymptotically slower than n^3. Therefore, we conclude that $n^2 = o(n^3)$.

5 The Class P

In computational complexity theory, the complexity class **P**, standing for **Polynomial Time**, is the set of all decision problems (which are problems with a "yes" or "no" answer) that can be solved by a deterministic Turing machine in polynomial time. In other words, **P** contains all problems for which an algorithm exists that can solve the problem in a number of steps that is bounded by a polynomial function of the size of the input.

In general, problems in the **P** class are considered **efficiently solvable** or **tractable**; tractable means that the problems can be solved in theory as well as in practice. When being implemented, efficient algorithms of **P** problems can find their solutions relatively quickly, even when the input data is very large.

The reason why polynomial time is the center of discussions falls in two aspects:

- Growth Rates: Polynomial functions grow at a much slower rate compared to exponential or factorial functions. Namely, even if an algorithm for a problem has a high-degree polynomial time complexity (e.g., n^4 or n^5), it's likely still manageable for sufficiently large inputs compared to exponential solutions.
- Reasonable increases: If a task's runtime doubles when the input size doubles, we'd consider that acceptable. If the runtime increases a thousandfold, there's a problem. Polynomial time algorithms still exhibit a reasonable scaling behavior.

For instance, consider two growth rates, one of polynomial n^3 and the other of exponential such as 2^n. Let n be 100, which is a reasonable input size to an algorithm. The polynomial running time takes 1 billion steps, a large but manageable number, whereas the exponential running time 2^{100} is an enormous number. This simple example explains that polynomial time algorithms are fast enough for many purposes, but exponential time algorithms rarely are useful.

All reasonable deterministic computational models are **polynomially equivalent**. That is, any one of them can simulate another with only a polynomial increase in running time. Analyzing an algorithm to show that it runs in polynomial time includes two tasks.

First, we have to give a polynomial upper bound (usually in big-O notation) on the number of stages that the algorithm uses when it runs on an input of length n. Then, we have to examine the individual stages in the description of the algorithm to be sure that each can be implemented in polynomial time on a reasonable deterministic model. We choose the steps when describing the algorithm to make this second part of the analysis easy to do.

When both tasks have been completed, we can conclude that the algorithm runs in polynomial time because we have demonstrated that it runs for a polynomial number of stages, each of which can be done in polynomial time, and the composition of polynomials are polynomials.

Example 5.1. Given an array of n integers (e.g., $[5, 8, 3, 10, 6]$), let us develop an algorithm to find the maximum element in this array. This problem can be solved in linear time $O(n)$ by scanning through the array once and keeping track of the maximum element encountered so far. As each element is examined, we update the maximum element if the current element is greater than the current maximum. This algorithm has a time complexity that grows linearly with the size of the input array n, making it an example of a problem in the complexity class **P**.

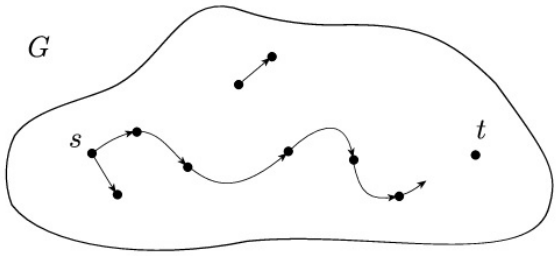
Example 5.2. Suppose we are given a directed graph G that contains vertices s and t, as shown in Figure 2. The PATH problem is to determine whether a directed path exists from s to t. Let

$$\mathsf{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}.$$

Theorem 5.3. *The language* PATH *is in* **P**.

Proof. **The high-level idea.** We prove this theorem by presenting a polynomial time algorithm that decides PATH. Before describing that algorithm,

Figure 2: The PATH problem (Sipser, pg. 287)



let's observe that a brute-force algorithm for this problem isn't fast enough. A brute-force algorithm for PATH proceeds by examining all potential paths in G and determining whether any of them is a directed path from s to t.

A potential path is a sequence of vertices in G having a length of at most m, where m is the number of vertices in G. (If any directed path exists from s to t, one having a length of at most m exists because repeating a vertex never is necessary.) But the number of such potential paths is roughly m^m, which is exponential in the number of vertices in G. Therefore, this brute-force algorithm uses exponential time.

To get a polynomial time algorithm for PATH, we must do something that avoids brute force. One way is to use a graph-searching method such as breadth-first search. Here, we successively mark all vertices in G that are reachable from s by directed paths of length 1, then 2, then 3, through m. We can now bound the running time of this strategy by a polynomial.

A poly-time algorithm for PATH. A polynomial time algorithm M for PATH operates as follows.

$M =$"On input $\langle G, s, t\rangle$, where G is a directed graph with vertices s and t:

1. Place a mark on vertex s.

2. Repeat the following until no additional vertices are marked:

    - Go over every edge (a, b) of all the edges of G; if an edge (a, b)

is found going from a marked vertex a to an unmarked vertex b, mark vertex b.

3. If t is marked, *accept*. Otherwise, *reject*."

Now we analyze this algorithm to show that it runs in polynomial time. Obviously, stages 1 and 3 are executed only once. Stage 2 runs at most m times because each time except the last it marks an additional vertex in G. Thus, the total number of stages used is at most $1 + 1 + m$, giving a polynomial in the size of G.

Stages 1 and 3 of M are easily implemented in polynomial time on any reasonable deterministic model. Stage 2 involves a scan of the input and a test of whether certain vertices are marked, which also is easily implemented in polynomial time. Hence M is a polynomial time algorithm for PATH.

□

6 The Class NP

The examples in the previous section demonstrate that many problems do not have to be solved by exhaustively searching through a space of solutions, called **brute-force search**.

There are problems, however, for which we are unable to find polynomial time algorithms; the reason could be that a polynomial solution has not been discovered yet, or that a polynomial algorithm can not exist due to the intrinsic difficulties of the problems. Instead, we may be given a proposed solution, and our task is simply to check whether or not the proposed solution is correct. Verifying the correctness of the proposed solution may be much easier than determining its existence. Simply put, a problem is said to have **polynomial verifiability** if there exists a polynomial-time verifier for its solutions.

An example of a problem with polynomial verifiability is the Hamiltonian Path problem: given a directed graph G and two vertices specified s

and t, is there a directed path from s to t that visits each vertex exactly once?

Let $\mathsf{HAMPATH} = \{\langle G, s, t\rangle \mid G$ is a directed graph with a Hamiltonian path from s to $t\}$. Verifying a proposed solution involves checking whether it is indeed a valid path that satisfies the conditions, and this verification can be done in polynomial time.

Another polynomially verifiable problem is compositeness. Recall that a natural number is *composite* if it is the product of two integers greater than 1 (i.e., a composite number is one that is not a prime number). Let

$$\mathsf{COMPOSITES} = \{x \mid x = pq, \text{for integers } p, q > 1\}.$$

We can easily verify that a number is composite—all that is needed is a divisor of that number.

On the other hand, some problems may not be polynomially verifiable, such as the complement of the Hamiltonian Path problem. Even if we could somehow determine that a graph did not have a Hamiltonian path, we do not know of a way for someone else to verify its nonexistence without using the same exponential time algorithm for making the determination in the first place.

Definition 6.1. *A **verifier** for a language L is an algorithm V, where*

$$L = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

When evaluating the time of a verifier, we only consider the length of the input string w. A **polynomial time verifier** is a verifier that runs in polynomial time relative to the length of w. A language L is considered **polynomially verifiable** if it has a polynomial time verifier.

It's important to note that a verifier uses additional information, represented by the string c in the definition, to verify whether the string w is a member of L. This information c is also called a **certificate**, or **proof**, of membership in L. For polynomial verifiers, the certificate also has polynomial length, in the length of w, since that is the only amount of information the verifier can access within its time bound.

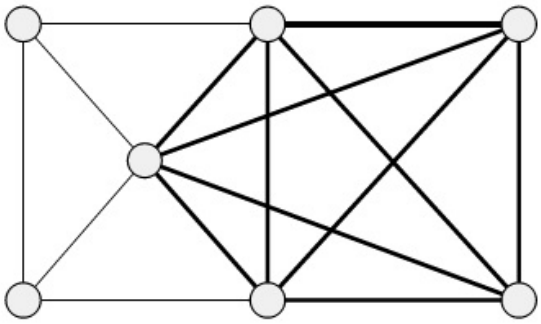
Definition 6.2. ***NP*** *is the class of languages that have polynomial time verifiers.*

The term **NP** stands for *Non-deterministic Polynomial Time* and it comes from an alternative representation, by using nondeterministic polynomial time Turing machines (NTMs) that can be deemed as "guess" machines. Essentially, if a problem is in **NP**, given a "candidate" solution to the problem, one can check whether the candidate is a correct solution in polynomial time. It's important to note that while all problems in **P** are also in **NP** (since a solution can be verified in polynomial time if it can be found in polynomial time), not all **NP** problems are known to be in **P**.

Example 6.3. In an undirected graph, a **clique** is a subgraph wherein every two vertices are connected by an edge. A k**-clique** is a clique that contains k vertices. Figure 3 illustrates a graph with a 5-clique. The clique problem is to determine whether a graph contains a clique of a specified size k. Let

$$\mathsf{CLIQUE} = \{\langle G, k\rangle | G \text{ is an undirected graph with a } k\text{-}clique\}.$$

Figure 3: A graph with 5-clique. (Sipser, pg. 296)



Theorem 6.4. CLIQUE *is in* **NP**.

Proof. The proof idea is to show the clique the certificate. The following is a verifier V for CLIQUE.

V = "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a subgraph with k vertices in G.

2. Test whether G contains all edges connecting vertices in c.

3. If both pass, *accept*; otherwise, *reject*."

Since the running time of the first two steps are proportional to the numbers of vertices and edges, the verifier V can be implemented in polynomial time.

□

Example 6.5. Given a set of positive integers and a target sum, is there a subset of the integers that adds up to the target sum?

For instance, suppose we are given a set of positive integers: {2, 4, 7, 9, 11, 15} and a target sum of 20. The subset sum problem asks whether there exists a subset of these numbers that adds up to the target sum of 20.

If someone claims to have found a subset (e.g., $\{4, 7, 9\}$) that adds up to the target sum, we can quickly verify it by summing the numbers in the subset to check if they indeed add up to 20. However, finding such a subset (if it exists) may require trying all possible combinations of numbers, which can be exponential in the number of integers.

Example 6.6. Let us consider the traveling salesman problem (TSP). Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the original city?

For instance, suppose we are given a set of cities: $\{A, B, C, D\}$; the distances between each pair of cities as follows:

- Distance from A to B: 10 units
- Distance from A to C: 15 units
- Distance from A to D: 20 units
- Distance from B to C: 35 units
- Distance from B to D: 25 units

- Distance from C to D: 30 units

The TSP asks for the shortest route that visits each city exactly once and returns to the original city.

If someone claims to have found a route as a solution, we can quickly verify it by summing the distances along the route and checking if it visits each city exactly once and returns to the original city. However, finding such a route (if it exists) may require trying all possible permutations of the cities, which grows factorially with the number of cities.

7 NP-Complete class

NP-complete problems constitute a subset within the category of **NP** problems. A problem is **NP-complete** if every other problem in **NP** can be reduced to it in polynomial time and it itself is also a **NP** problem. **NP-complete** problems are significant because they serve as a litmus test for the potential polynomial-time solvability of all **NP** problems. The existence of a polynomial-time algorithm for any **NP-complete** problem implies that every single **NP** problem is also has a polynomial-time algorithm, enabling the efficient resolution of all **NP** problems.

From a practical perspective, **NP-completeness** acts as a warning signal, discouraging attempts to find efficient polynomial-time solutions, which are likely to be fruitless endeavors. Although there is no formal proof yet, the widely accepted belief that $\mathbf{P} \neq \mathbf{NP}$ guides this pragmatic approach. Therefore, identifying a problem as **NP-complete** provides strong evidence that it cannot be solved by any polynomial-time algorithm.

A central component of demonstrating **NP-completeness** is **polynomial time reducibility**. When problem A is reducible to problem B, it means if we can find a solution to B, it can be used to solve A as well. In other words, when problem A is efficiently reducible to problem B, an efficient solution to B can be used to solve A efficiently.

Definition 7.1. *A function $f : \Sigma^* \to \Sigma^*$ is a **polynomial time computable function** if some polynomial time Turing machine M exists that*

halts with just $f(w)$ on its tape, when started on input w.

Definition 7.2. *Language A is **polynomial time mapping reducible**, or simply **polynomial time reducible**, to language B, written $A \leq_P B$, if a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ exists, where for every w,*

$$w \in A \Leftrightarrow f(w) \in B.$$

*The function f is called the **polynomial time reduction** of A to B.*

A polynomial-time reduction of A to B provides an efficient method to convert testing of membership in A to testing of membership in B. To test whether $w \in A$, we use the polynomial-time reduction function f to map w to $f(w)$ and then test whether $f(w) \in B$. If one language is polynomial time reducible to another language which is already known to have an existing polynomial time solution, then we can find a polynomial time solution to the original language, as stated in the following theorem.

Theorem 7.3. *If $A \leq_P B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.*

Proof. Let M be the polynomial time algorithm deciding B and f be the polynomial time reduction from A to B. We describe a polynomial time algorithm N deciding A as follows.

$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

We have $w \in A$ whenever $f(w) \in B$ because f is a reduction from A to B. Thus, M accepts $f(w)$ whenever $w \in A$. Moreover, N runs in polynomial time because each of its two stages runs in polynomial time. Note that stage 2 runs in polynomial time because the composition of two polynomials is a polynomial.

□

Before we move onto formal definition of **NP-completeness**, let us introduce the first **NP-complete** problem: the SAT problem. Recall that there are two Boolean variables, TRUE and FALSE, which are typically represented by 1 and 0, respectively. The Boolean operations AND, OR, and NOT are represented by the symbols $\wedge$, $\vee$, and $\neg$, respectively; we use the overbar as a shorthand for the $\neg$ symbol, so $\bar{x}$ means $\neg x$. The following lists these operations on two Boolean variables.

$$\begin{array}{lll} 0 \wedge 0 = 0 & 0 \vee 0 = 0 & \bar{0} = 1 \\ 0 \wedge 1 = 0 & 0 \vee 1 = 1 & \bar{1} = 0 \\ 1 \wedge 0 = 0 & 1 \vee 0 = 1 & \\ 1 \wedge 1 = 1 & 1 \vee 1 = 1 & \end{array}$$

A Boolean formula is a logical statement that combines Boolean variables (TRUE or FALSE) using Boolean operations (AND, OR, and NOT). For example,

$$\varphi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$$

is a Boolean formula. A Boolean formula is **satisfiable** if some expression assignment of 0s and 1s makes the formula evaluate to 1 (TRUE). The formula above is satisfiable because there is at least one assignment $x = 0$, $y = 1$, and $z = 0$ that makes φ evaluate to 1. We say this specific assignment satisfies φ. The satisfiability problem is to decide whether a given Boolean formula is satisfiable, denoted by

$$\mathsf{SAT} = \{\langle\varphi\rangle | \varphi \text{ is a satisfiable Boolean formula}\}.$$

Now, let's consider the problem 3SAT, which is a special case of the satisfiability problem where all the Boolean formulas are in a particular form. A **literal** is a Boolean variable or a negated Boolean variable, as in x or $\bar{x}$. A **clause** is several literals connected with $\vee$s, as in $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$. A Boolean formula is in **conjunctive normal form**, called a **cnf-formula**, if it comprises several clauses connected with $\wedge$s, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6}).$$

The following is a **3cnf-formula** because all the clauses have three literals:

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6).$$

Let $\mathsf{3SAT} = \{\langle\varphi\rangle | \varphi \text{ is a satisfiable 3cnf-formula}\}$. An assignment that satisfies a cnf-formula is one where each clause must contain at least one literal that evaluates to 1.

The following theorem presents a polynomial time reduction from the 3SAT problem to the CLIQUE problem.

Theorem 7.4. 3SAT *is polynomial time reducible to* CLIQUE.

Proof. The polynomial time reduction f that we demonstrate from 3SAT to CLIQUE converts formulas to graphs. In the constructed graphs, cliques of a specified size correspond to satisfying assignments of the formula. Structures within the graph are designed to mimic the behavior of the variables and clauses.

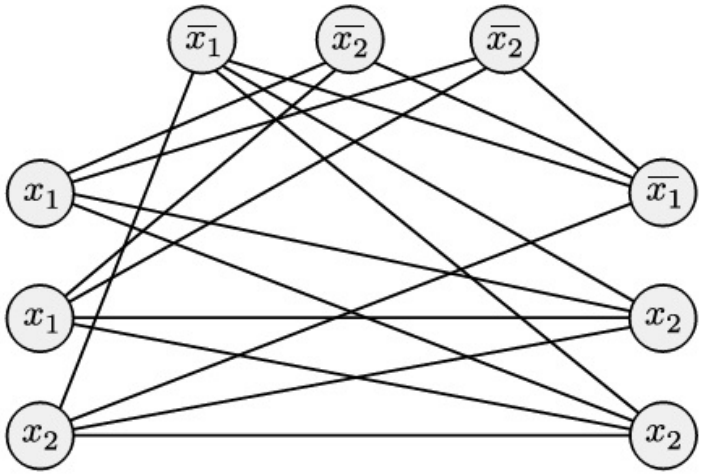
Let φ be a formula with k clauses such as

$$\varphi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k).$$

The reduction f generates the string $\langle G, k\rangle$, where G is an undirected graph defined as follows. The vertices in G are organized into k groups of three vertices each called the **triples**, $t_1, \ldots, t_k$. Each triple corresponds to one of the clauses in φ, and each vertex in a triple corresponds to a literal in the associated clause. Label each vertex of G with its corresponding literal in φ. The edges of G connect all but two types of pairs of vertices in G. No edge is present between vertices in the same triple, and no edge is present between two vertices with contradictory labels, as in x_2 and $\neg x_2$. Figure 4 illustrates this construction when $\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.

Now we demonstrate why this construction works. We show that φ is satisfiable if and only if G has a k-clique.

Figure 4: The graph reduction of the 3SAT problem. (Sipser, pg. 303)



Suppose that φ has a satisfying assignment. In that satisfying assignment, at least one literal is true in every clause. In each triple of G, we select one vertex corresponding to a true literal in the satisfying assignment. If more than one literal is true in a particular clause, we choose one of the true literals arbitrarily. The Vertices just selected form a k-clique. The number of vertices selected is k because we chose one for each of the k triples. Each pair of selected vertices is joined by an edge because no pair fits one of the exceptions described previously. They could not be from the same triple because we selected only one vertex per triple. They could not have contradictory labels because the associated literals were both true in the satisfying assignment. Therefore, G contains a k-clique.

Suppose that G has a k-clique. No two of the clique's vertices occur in the same triple because vertices in the same triple are not connected by edges. Therefore, each of the k triples contains exactly one of the k clique vertices. We assign truth values to the variables of φ so that each literal labeling a clique Vertex is made true. Doing so is always possible because two vertices labeled in a contradictory way are not connected by an edge and hence both can't be in the clique. This assignment to the variables satisfies φ because each triple contains a clique vertex and hence each clause contains a literal that is assigned TRUE. Therefore, φ is satisfiable.

□

This example shows that if CLIQUE is solvable in polynomial time, then so is 3SAT. At first sight, this connection between these two problems is surprising because, they appear to be quite different on the surface. However, the concept of polynomial-time reducibility allows us to find a link between the computational complexities of these two seemingly unrelated problems. Now we proceed to give a definition that will allow us to similarly link the complexities of an entire class of problems.

Definition 7.5. *A language B is **NP-complete** if it satisfies two conditions:*

1. *B is in* **NP**, *and*
2. *every A in* **NP** *is polynomial time reducible to B.*

Theorem 7.6. *If B is NP-complete and $B \in \mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$.*

Proof. This theorem follows directly from the definition of polynomial time reducibility.

□

Theorem 7.7. *If B is NP-complete and $B \leq_P C$ for C in* **NP**, *then C is NP-complete.*

Proof. We already know that C is in **NP**, so we must show that every A in **NP** is polynomial time reducible to C. Because B is NP-complete, every language in **NP** is polynomial time reducible to B, and B in turn is polynomial time reducible to C. Polynomial time reductions compose; that is, if A is polynomial time reducible to B and B is polynomial time reducible to C, then A is polynomial time reducible to C. Hence every language in **NP** is polynomial time reducible to C.

□

Let's now look at another example.

Example 7.8. Graph coloring is a classic example of an **NP-complete** problem. In the graph coloring problem, we are given an undirected graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. The task is to assign colors to the vertices of the graph such that no two adjacent

vertices share the same color, using the fewest possible number of colors. This property makes graph coloring an important problem with applications in scheduling, register allocation in compilers, and network design, among others.

Formally, a graph coloring problem can be stated as follows: Given an undirected graph $G = (V, E)$, find a function $f : V \to \mathcal{N}$ such that for any edge $(u, v) \in E$, $f(u) \neq f(v)$.

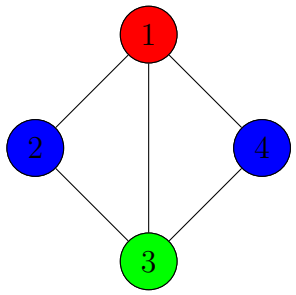


Figure 5: A graph coloring problem with three colors

Figure 5 illustrates an example of the graph coloring problem. In this coloring, each vertex is assigned a color (red, blue, or green) such that no two adjacent vertices have the same color. This is a valid coloring of the graph.

To show that the graph coloring problem is **NP-complete**, we need to demonstrate two tasks.

First, the graph coloring problem is in the class **NP**, meaning that given a potential coloring of the graph, we can verify in polynomial time whether it is a valid coloring.

For the second task, we can reduce the well-known **NP-complete** problem SAT (Boolean satisfiability problem) to the graph coloring problem. This reduction is typically done by constructing a graph from a given SAT instance, such that if the SAT instance is satisfiable, the graph can be colored with a certain number of colors, and if the SAT instance is not satisfiable, the graph

cannot be colored with a certain number of colors.

For simplicity, we use three colors for the explanation. Here's a brief outline of the reduction:

Step 1: Construct a Boolean formula from the graph. We construct a Boolean formula that is SAT if and only if the graph can be 3-colored.

- **Variables**: Create Boolean variables $x_{v,c}$ for each vertex $v \in V$ and each color $c \in \{R, G, B\}$. The variable $x_{v,c}$ will be true if and only if vertex v is colored with color c.
- **Clauses**: Construct the following types of clauses to ensure a valid 3-coloring:
  - **Each vertex must be colored with exactly one color**:
    - For each vertex v, create the clause:

      $$(x_{v,R} \vee x_{v,G} \vee x_{v,B})$$

      This clause ensures that v is colored with at least one color.
    - For each pair of colors $c_1 \neq c_2$, create the clause:

      $$(\neg x_{v,c_1} \vee \neg x_{v,c_2})$$

      This clause ensures that v is not colored with two colors simultaneously.
  - **No two adjacent vertices can share the same color**:
    - For each edge $(u, v) \in E$ and each color c, create the clause:

      $$(\neg x_{u,c} \vee \neg x_{v,c})$$

      This clause ensures that if u is colored with color c, then v cannot be colored with color c, and vice versa.

Step 2: Formulate the complete Boolean formula. Combine all the clauses constructed in the previous step into a single cnf-formula Φ. The formula Φ is satisfiable if and only if the graph G can be colored with three

colors.

Step 3: Polynomial-time transformation. Each vertex generates a constant number of clauses (specifically, four clauses: one for ensuring the vertex is colored and three for ensuring it is not colored with more than one color). Each edge generates three clauses. Therefore, the number of clauses and the size of the formula are polynomial in the size of the input graph.

Step 4: Conclusion. Since the SAT problem is known to be **NP-complete**, and we have shown a polynomial-time reduction from the 3-coloring problem to the SAT problem, it follows that the 3-coloring problem is also **NP-complete**.

References

[Sip96] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.