Basic Complexity Theory

FLOLAC 2023 Summer School

21 August – 1 September 2023

National Taiwan University

Taipei, Taiwan

Tony Tan

Department of Computer Science

University of Liverpool

Contents

1 Preliminaries 3
- 1.1 The big-Oh notations . . . 3
- 1.2 Turing machines . . . 3
- 1.3 Universal Turing machines . . . 4
- 1.4 Church-Turing thesis . . . 4

2 Basic complexity classes 5
- 2.1 Time complexity . . . 5
- 2.2 Space complexity . . . 6
- 2.3 Logarithmic space complexity . . . 7
- 2.4 Some basic relations between complexity classes . . . 7

3 NP-complete languages 8
- 3.1 An alternative definition of the class **NP** . . . 8
- 3.2 **NP**-complete languages . . . 8
- 3.3 More NP-complete problems . . . 12
- 3.4 **coNP**-complete languages . . . 12

4 The class NL and PSPACE 14
- 4.1 **NL**-complete languages . . . 14
- 4.2 **PSPACE**-complete languages . . . 16

5 P-complete languages 17

6 Alternating Turing machines 18
- 6.1 Definition . . . 18
- 6.2 Time and space complexity for ATM . . . 18

7 The polynomial hierarchy 20

8 The complexity of counting 21
- 8.1 The class **FP** . . . 21
- 8.2 The class ♯**P** . . . 21
- 8.3 The complexity of computing the permanent . . . 22
- 8.4 Reduction from 3-SAT to cycle cover . . . 23
- 8.5 Reduction from matrices over $\mathbb{Z}$ to matrices over $\{0,1\}$ . . . 26
- 8.6 ♯**P**-hardness of PERM – Putting all the pieces together . . . 26

9 Diagonalization on the class NP 27
- 9.1 Ladner's theorem: **NP**-intermediate language . . . 27
- 9.2 Limit of diagonalization . . . 29

A The notion of computable functions 30

B Time and space constructible functions 30

C Hardness via log space reduction 31

1 Preliminaries

1.1 The big-Oh notations

Let $\mathbb{N}$ denote the set of natural numbers $\{0, 1, 2, \ldots\}$. Let f and g be functions from $\mathbb{N}$ to $\mathbb{N}$. We will use the following notations.

- $f = O(g)$ means that there is c and n_0 such that for every $n \geqslant n_0$, $f(n) \leqslant c \cdot g(n)$.

  It is usually phrased as "there is c such that for (all) sufficiently large n," $f(n) \leqslant c \cdot g(n)$.
- $f = \Omega(g)$ means $g = O(f)$.
- $f = \Theta(g)$ means $g = O(f)$ and $f = O(g)$.
- $f = o(g)$ means for every $c > 0$, $f(n) \leqslant c \cdot g(n)$ for sufficiently large n.

  Equivalently, $f = o(g)$ means $f = O(g)$ and $g \neq O(f)$.

  Another equivalent definition is $f = o(g)$ means $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- $f = \omega(g)$ means $g = o(f)$.

To emphasize the input parameter, we will write $f(n) = O(g(n))$. The same for the Ω, o, ω notations. We also write $f(n) = \mathsf{poly}(n)$ to denote that $f(n) = c \cdot n^k$ for some c and $k \geqslant 1$.

Throughout the course, for an integer $n \geqslant 0$, we will denote by $\lfloor n \rfloor$ the binary representation of n. Likewise, $\lfloor G \rfloor$ the binary encoding of a graph G. In general, we write $\lfloor X \rfloor$ to denote the encoding/representation of an object X as a binary string, i.e., a 0-1 string. To avoid clutter, we often write X instead of $\lfloor X \rfloor$.

We usually use Σ to denote a finite input alphabet. Often $\Sigma = \{0, 1\}$. Recall also that for a word $w \in \Sigma^*$, $|w|$ denotes the length of w. For a DTM/NTM $\mathcal{M}$, we write $L(\mathcal{M})$ to denote the language $\{w : \mathcal{M} \text{ accepts } w\}$.

We often view a language $L \subseteq \Sigma^*$ as a boolean function, i.e., $L : \Sigma^* \to \{\mathsf{true}, \mathsf{false}\}$, where $L(x) = \mathtt{true}$ if and only if $x \in L$, for every $x \in \Sigma^*$.

1.2 Turing machines

Let $k \geqslant 1$. A *k-tape Turing machine* is a system $\mathcal{M} = \langle \Sigma, \Gamma, Q, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}, \delta \rangle$,

- Σ is a finite alphabet, called the *input* alphabet, where $\sqcup, \triangleleft \notin \Sigma$.
- Γ is a finite alphabet, called the *tape* alphabet, where $\Sigma \subseteq \Gamma$ and $\sqcup, \triangleleft \in \Gamma$.
- Q is a finite set of states.
- $q_0 \in Q$ is the initial state.
- $q_{\mathsf{acc}}, q_{\mathsf{rej}} \in Q$ are two special states called the *accept* and *reject* states, respectively.
- $\delta : (Q - \{q_{\mathsf{acc}}, q_{\mathsf{rej}}\}) \times \Gamma^k \to Q \times \Gamma^k \times \{\mathtt{Left}, \mathtt{Right}\}^k$ is the transition function.

A transition in δ is written in the form:

$$(q, a_1, \ldots, a_k) \to (p, b_1, \ldots, b_k, \alpha_1, \ldots, \alpha_k).$$

Intuitively, it means that if the TM is in state q, and on each $i = 1, \ldots, k$, the head on tape i is reading a_i, then it enters state p, and for $i = 1, \ldots, k$, the head on tape i writes the symbol b_i and moves according to α_i. If α_i is `Left`, it moves left. If α_i is `Right`, it moves right.

A *configuration* of $\mathcal{M}$ is a string of the form $(q, u_1, \ldots, u_k)$, where $q \in Q$, each u_i is a string over $\Gamma \cup \{\bullet\}$ and the symbol $\bullet$ appears exactly once in each u_i. The symbol $\bullet$ denotes the position of the head.

The *initial configuration* of $\mathcal{M}$ on input w is $(q_0, \triangleleft \bullet w, \triangleleft\bullet, \ldots, \triangleleft\bullet)$, i.e., the first tape initially contains the input word and all the other tapes are initially blank. A configuration is *accepting*, if the state is the accept state and *rejecting*, if the state is the reject state. It is a *halting* configuration, if it is accepting or rejecting configuration.

A configuration C yields another configuration C', denoted by $C \vdash C'$, if C' is the configuration obtained after applying the transition function on C. On input word w, a *run* of $\mathcal{M}$ on w is a sequence $C_0 \vdash C_1 \vdash C_2 \vdash \cdots$ where C_0 is the initial configuration on w. It is an accepting run, if it ends with an accepting configuration.

Remark 1.1

- We will often assume that the input alphabet of a Turing machine is $\Sigma = \{0, 1\}$ and the tape alphabet is $\Gamma = \{0, 1, \sqcup\}$.
- $\lfloor \mathcal{M} \rfloor$ denotes the encoding of a TM $\mathcal{M}$.

1.3 Universal Turing machines

Definition 1.2 A *Universal Turing machine* (UTM) is a k-tape DTM $\mathcal{U}$, for some $k \geqslant 1$, such that $L(\mathcal{U}) = \{\lfloor \mathcal{M} \rfloor \$ w \mid \mathcal{M} \text{ accepts } w \text{ and } w \in \{0,1\}^*\}$.

The following theorem will be useful, but we will not prove it in the class.

Theorem 1.3 *There is a UTM $\mathcal{U}$ such that for every DTM $\mathcal{M}$ and every word w, if $\mathcal{M}$ decides w in time t, then $\mathcal{U}$ decides $\lfloor \mathcal{M} \rfloor \$ w$ in time $(\alpha \cdot t \cdot \log t)$, where α does not depends $|w|$, but on size of the tape alphabet of $\mathcal{M}$ as well as the number of tapes and states of $\mathcal{M}$.*

1.4 Church-Turing thesis

Church-Turing thesis states that any "algorithm" is equivalent to a Turing machine. To be more concrete, we can view an algorithm is just a C++ program. Note that a C++ program can only use a fixed number of variables and each variable can only store a string. Multi-tape Turing machines and C++ programs are equivalent in the sense that a k-tape Turing machine can be viewed as a C++ program that uses k variables, and conversely, a C++ program that uses k variables can be viewed as a k-tape Turing machine. In this note we will often use the terms *Turing machine* and *algorithm* interchangeably depending on our convenience. When we try to show that a certain problem is in certain class, we often simply describe its algorithm. However, in some proofs, e.g., when we try to prove SAT is **NP**-complete, it will be easier to use Turing machines.

2 Basic complexity classes

2.1 Time complexity

Definition 2.4 Let $\mathcal{M}$ be a DTM/NTM, $w \in \Sigma^*$, $t \in \mathbb{N}$ and let $f : \mathbb{N} \to \mathbb{N}$ be a function.

- $\mathcal{M}$ *decides* w *in time* t *(or, in* t *steps)*, if every run of $\mathcal{M}$ on w has length at most t. That is, for every run of $\mathcal{M}$ on w:

$$C_0 \vdash C_1 \vdash \cdots \vdash C_m \qquad \text{where } C_m \text{ is a halting configuration,}$$

we have $m \leqslant t$.

- $\mathcal{M}$ *runs in time* $O(f(n))$, if there is $c > 0$ such that for sufficiently long word w, $\mathcal{M}$ decides w in time $c \cdot f(|w|)$.
- $\mathcal{M}$ *decides/accepts a language* L *in time* $O(f(n))$, if $L(\mathcal{M}) = L$ and $\mathcal{M}$ runs in time $O(f(n))$.
- $\textsc{Dtime}[f(n)] \stackrel{\text{def}}{=} \{L : \text{there is a DTM } \mathcal{M} \text{ that decides } L \text{ in time } O(f(n))\}$.
- $\textsc{Ntime}[f(n)] \stackrel{\text{def}}{=} \{L : \text{there is an NTM } \mathcal{M} \text{ that decides } L \text{ in time } O(f(n))\}$.

Note that Definition 2.4 applies in similar manner for both DTM and NTM. The only difference is that a DTM has one run for each input word w, whereas NTM can have many runs for each input word w.

We say that $\mathcal{M}$ runs in *polynomial* and *exponential time*, if there is $f(n) = \mathsf{poly}(n)$ such that $\mathcal{M}$ runs in time $O(f(n))$ and $O(2^{f(n)})$, respectively. In this case we also say that $\mathcal{M}$ is a polynomial/exponential time TM.

The following are some of the important classes in complexity theory.

$$\mathbf{P} \stackrel{\text{def}}{=} \bigcup_{f(n)=\mathsf{poly}(n)} \textsc{Dtime}[f(n)] \qquad \mathbf{EXP} \stackrel{\text{def}}{=} \bigcup_{f(n)=\mathsf{poly}(n)} \textsc{Dtime}[2^{f(n)}]$$

$$\mathbf{NP} \stackrel{\text{def}}{=} \bigcup_{f(n)=\mathsf{poly}(n)} \textsc{Ntime}[f(n)] \qquad \mathbf{NEXP} \stackrel{\text{def}}{=} \bigcup_{f(n)=\mathsf{poly}(n)} \textsc{Ntime}[2^{f(n)}]$$

$$\mathbf{coNP} \stackrel{\text{def}}{=} \{L : \Sigma^* - L \in \mathbf{NP}\} \qquad \mathbf{coNEXP} \stackrel{\text{def}}{=} \{L : \Sigma^* - L \in \mathbf{NEXP}\}$$

Theorem 2.5 (Padding theorem) *If* $\mathbf{NP} = \mathbf{P}$*, then* $\mathbf{NEXP} = \mathbf{EXP}$.
Likewise, if $\mathbf{NP} = \mathbf{coNP}$*, then* $\mathbf{NEXP} = \mathbf{coNEXP}$.

Proof. We will only prove the first statement, i.e., "if $\mathbf{NP} = \mathbf{P}$, then $\mathbf{NEXP} = \mathbf{EXP}$."

Suppose $\mathbf{NP} = \mathbf{P}$. We will show that $\mathbf{NEXP} \subseteq \mathbf{EXP}$. Let $L \in \mathbf{NEXP}$. Let $\mathcal{M}$ be an NTM that decides L in time $2^{p(n)}$, where $p(n) = \mathsf{poly}(n)$. Consider the following language:

$$L' \stackrel{\text{def}}{=} \{w0\underbrace{11\cdots1}_{m} \ : \ w \in L \text{ and } m = 2^{p(|w|)}\}$$

We will first show that $L' \in \mathbf{NP}$. Consider Algorithm 1.

Since $\mathcal{M}$ is non-deterministic, Algorithm 1 is also non-deterministic. We can show that Algorithm 1 runs in polynomial time (in the length of the input u). Thus, $L' \in \mathbf{NP}$. By our assumption that $\mathbf{NP} = \mathbf{P}$, we obtain that $L' \in \mathbf{P}$. Let $\mathcal{M}'$ be a DTM that decides L' in polynomial time.

To show that $L \in \mathbf{EXP}$, consider the following algorithm that we denote by Algorithm 2.

Algorithm 1

Input: $u \in \Sigma^*$.

Task: Decide if $u \in L'$.

1: Check if u is of the form $w0\underbrace{11\cdots 1}_{m}$ for some m. and that $m = 2^{p(|w|)}$.
   If not, REJECT. Otherwise, continue.

2: Run $\mathcal{M}$ on w.

3: ACCEPT if and only if $\mathcal{M}$ accepts w.

Algorithm 2

Input: $w \in \Sigma^*$.

Task: Decide if $w \in L$.

1: Compute $m \stackrel{\mathsf{def}}{=} 2^{p(|w|)}$.

2: Run $\mathcal{M}'$ on input $w01^m$.

3: ACCEPT if and only if $\mathcal{M}$ accepts w.

Note that by the definition of L', Algorithm 2 decides the language L. It is deterministic because $\mathcal{M}'$ is deterministic. Moreover, it runs in exponential time in the length of the input word w. Therefore, $L \in \mathbf{EXP}$, as desired. This completes the proof that $\mathbf{NP} = \mathbf{P}$ implies $\mathbf{NEXP} = \mathbf{EXP}$. $\square$

2.2 Space complexity

Definition 2.6 Let $\mathcal{M}$ be a DTM/NTM, $w \in \Sigma^*$, $t \in \mathbb{N}$ and let $f : \mathbb{N} \to \mathbb{N}$ be a function.

- $\mathcal{M}$ decides w in t space (or, using t cells/space), if for every run of $\mathcal{M}$ on w:

  $$C_0 \ \vdash\ C_1\ \vdash\ \cdots\ \vdash\ C_N \qquad \text{where } C_N \text{ is an accepting/rejecting configuration,}$$

  the length $|C_i| \leqslant t$, for each $i = 0, \ldots, N$.

- $\mathcal{M}$ *uses* $O(f(n))$ *space*, if there is $c > 0$ such that for sufficiently long word w, $\mathcal{M}$ decides w using $c \cdot f(|w|)$ space.

- $\mathcal{M}$ *decides/accepts a language* L *in space* $O(f(n))$, if $L(\mathcal{M}) = L$ and $\mathcal{M}$ uses $O(f(n))$ space.

- $\textsc{Dspace}[f(n)] \stackrel{\mathsf{def}}{=} \{L : \text{there is a DTM } \mathcal{M} \text{ that decides } L \text{ using } O(f(n)) \text{ space}\}$.

- $\textsc{Nspace}[f(n)] \stackrel{\mathsf{def}}{=} \{L : \text{there is an NTM } \mathcal{M} \text{ that decides } L \text{ using } O(f(n)) \text{ space}\}$.

Again, note that the notion of $\mathcal{M}$ uses space $O(f(n))$ is the same for DTM and NTM. The only difference is that a DTM has only one run for each input word w, whereas NTM can have many runs for each input word w. In both cases, we can only say that $\mathcal{M}$ uses space $O(f(n))$, if for each input word w, for every run of $\mathcal{M}$ on w, the length of each configuration in the run is always $\leqslant cf(|w|)$.

We say that $\mathcal{M}$ uses *polynomial* and *exponential* space, if there is $f(n) = \mathsf{poly}(n)$ such that $\mathcal{M}$ runs in time $O(f(n))$ and $O(2^{f(n)})$, respectively. In this case we also say that $\mathcal{M}$ is a polynomial/exponential space TM. The following are some of the important classes in complexity

theory.

$$\mathbf{PSPACE} \stackrel{\mathsf{def}}{=} \bigcup_{f(n)=\mathsf{poly}(n)} \textsc{Dspace}[f(n)]$$
$$\mathbf{EXPSPACE} \stackrel{\mathsf{def}}{=} \bigcup_{f(n)=\mathsf{poly}(n)} \textsc{Dspace}[2^{f(n)}]$$

2.3 Logarithmic space complexity

Another interesting classes are $\mathbf{L}$ and $\mathbf{NL}$. We say that a language L is in $\mathbf{L}$, if there is a 2-tape DTM $\mathcal{M}$ that decides L and a constant $c > 0$ such that for every input word w:

- The first tape always contains only the input word w, i.e., $\mathcal{M}$ never changes the content of the first tape.
- $\mathcal{M}$ uses $c \cdot \log(|w|)$ space in its second tape.

Likewise, we say that a language L is in $\mathbf{NL}$, if there is a 2-tape NTM $\mathcal{M}$ that decides L such that the above two conditions are satisfied.

2.4 Some basic relations between complexity classes

Obviously, we have $\mathbf{L} \subseteq \mathbf{NL}$, $\mathbf{P} \subseteq \mathbf{NP}$, and $\mathbf{PSPACE} \subseteq \mathbf{NPSPACE}$.

Proposition 2.7

- $\mathbf{L} \subseteq \mathbf{P}$.
- $\mathbf{NP} \subseteq \mathbf{PSPACE}$.

Deterministic/non-deterministic time/space hierarchy theorem states that for every $k \geqslant 1$, the following holds.*

$$\textsc{Dtime}[n^k] \subsetneq \textsc{Dtime}[n^{k+1}] \qquad \textsc{Dspace}[n^k] \subsetneq \textsc{Dspace}[n^{k+1}]$$
$$\textsc{Ntime}[n^k] \subsetneq \textsc{Ntime}[n^{k+1}] \qquad \textsc{Nspace}[n^k] \subsetneq \textsc{Nspace}[n^{k+1}]$$

Some classic results in complexity theory are: (They will be proved later on.)

- $\mathbf{NL} \subseteq \mathbf{P}$.
- If $L \in \textsc{Nspace}[n^k]$, then $\Sigma^* - L \in \textsc{Nspace}[n^k]$.
- $\textsc{Nspace}[n^k] \subseteq \textsc{Dspace}[n^{2k}]$.

The third bullet is the reason why we only have the class $\mathbf{PSPACE}$.

Combining all these inclusions together, we obtain:

$$\mathbf{L} \ \subseteq \ \mathbf{NL} \ \subseteq \ \mathbf{P} \ \subseteq \ \mathbf{NP} \ \subseteq \ \mathbf{PSPACE}$$

From the deterministic/non-deterministic space hierarchy, it is also known that $\mathbf{L} \subsetneq \mathbf{PSPACE}$ and $\mathbf{NL} \subsetneq \mathbf{PSPACE}$. So, we know that at least one of the inclusions must be strict, but we don't know which one.

*Due to time constraint, we will not prove the hierarchy theorem in this class.

3 NP-complete languages

3.1 An alternative definition of the class NP

Recall that for a string w, the length of w is denoted by $|w|$. In the previous lesson, we define the class **NP** as follows.

Definition 3.8 A language L is in **NP** if there is $f(n) = \mathsf{poly}(n)$ and an NTM $\mathcal{M}$ such that $L(\mathcal{M}) = L$ and $\mathcal{M}$ runs in time $O(f(n))$.

Definition 3.9 below is an alternative definition of **NP**.

Definition 3.9 A language $L \subseteq \Sigma^*$ is in **NP** if there is a language $K \subseteq \Sigma^* \cdot \{\#\} \cdot \Sigma^*$, where $\# \notin \Sigma$, such that the following holds.

- For every $w \in \Sigma^*$, $w \in L$ if and only if there is $v \in \Sigma^*$ such that $w\#v \in K$.
- There is $f(n) = \mathsf{poly}(n)$ such that for every $w\#v \in K$, $|v| \leqslant f(|w|)$.
- The language K is accepted by a polynomial time DTM.

For $w\#v \in K$, the string v is called the *certificate/witness* for w. We call the language K the *certificate/witness language* for L.

Indeed Def. 3.8 and 3.9 are equivalent. That is, for every language L, L is in **NP** in the sense of Def. 3.8 if and only if L is in **NP** in the sense of Def. 3.9.

3.2 NP-complete languages

Recall that a DTM $\mathcal{M}$ computes a function $F : \Sigma^* \to \Sigma^*$ in time $O(g(n))$, if there is a constant $c > 0$ such that on every word w, $\mathcal{M}$ computes $F(w)$ in time $\leqslant cg(|w|)$. If $g(n) = \mathsf{poly}(n)$, such function F is called *polynomial time computable* function. Moreover, if $\mathcal{M}$ uses only logarithmic space, it is called *logarithmic space computable* function. See Appendix A for more details.

The following definition is one of the most important definitions in computer science.

Definition 3.10 A language L_1 is *polynomial time reducible* to another language L_2, denoted by $L_1 \leqslant_p L_2$, if there is a polynomial time computable function F such that for every $w \in \Sigma^*$:

$$w \in L_1 \quad \text{if and only if} \quad F(w) \in L_2$$

Such function F is called polynomial time reduction, also known as *Karp reduction.*

If F is logarithmic space computable function, we say that L_1 is *log-space reducible* to L_2, denoted by $L_1 \leqslant_{\log} L_2$.

If L_1 and L_2 are in **NP** with certificate languages K_1 and K_2, respectively, we say that F is *parsimonious*, if for every $w \in \Sigma^*$, w has the same number of certificates in K_1 as $F(w)$ in K_2.

Definition 3.11 Let L be a language.

- L is **NP**-*hard*, if for every $L' \in \mathbf{NP}$, $L' \leqslant_p L$.
- L is **NP**-*complete*, if $L \in \mathbf{NP}$ and L is **NP**-hard.

Recall that a propositional formula (Boolean formula) with variables $x_1, \ldots, x_n$ is in Conjunctive Normal Form (CNF), if it is of the form: $\bigwedge_i (\ell_{i,1} \vee \cdots \vee \ell_{i,k_i})$ where each $\ell_{i,j}$ is a literal, i.e., a variable x_k or its negation $\neg x_k$. It is in 3-CNF, if it is of the form $\bigwedge_i \big(\ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3}\big)$. A formula φ is satisfiable, if there is an assignment of Boolean values true or false to each variable in φ that evaluates to true.

SAT	
Input:	A propositional formula φ in CNF.
Task:	Output true, if φ is satisfiable. Otherwise, output false.

SAT can be viewed as a language, i.e., $\mathsf{SAT} \stackrel{\text{def}}{=} \{\varphi : \varphi \text{ is satisfiable CNF formula}\}$.

Theorem 3.12 (Cook 1971, Levin 1973) SAT *is* **NP**-*complete.*

Proof. We have to show that $\mathsf{SAT} \in \mathbf{NP}$ and SAT is **NP**-hard. We first show that $\mathsf{SAT} \in \mathbf{NP}$. Consider the following non-deterministic algorithm that decides SAT. On input formula φ, do the following.

- Let $x_1, \ldots, x_n$ be the variables in φ.
- For each $i = 1, \ldots, n$ do:
  - Non-deterministically assign the value of x_i to either true or false.
- Check if the formula φ evaluates to true under the assignment.
- If the formula evaluates to true, then ACCEPT.

  If the formula evaluates to false, then REJECT.

It is not difficult to show that the algorithm above accepts a formula φ if and only if it is satisfiable. This completes the proof that $\mathsf{SAT} \in \mathbf{NP}$.

Now we show that SAT is **NP**-hard. That is, for every $L \in \mathbf{NP}$, $L \leqslant_p \mathsf{SAT}$.

Let $L \in \mathbf{NP}$. Let $\mathcal{M} = \langle \Sigma, \Gamma, Q, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}, \delta \rangle$ be the NTM that decides L in time $f(n) = \mathsf{poly}(n)$, where Σ is the input alphabet, Γ is the tape alphabet, Q is the set of states, q_0 is the initial state, q_{acc} is the accepting state, q_{rej} is the rejecting state and δ is the set of transitions. We denote by $\sqcup$ the blank symbol. We may assume that $\mathcal{M}$ has only 1 tape.

We will describe a deterministic algorithm $\mathcal{A}$ such that on every word w, it output a CNF formula φ such that the following holds.

$$w \in L \quad \text{if and only if} \quad \varphi \text{ is satisfiable.}$$

Intuitively, φ "describes" the accepting run of $\mathcal{M}$ on w such that it is satisfiable if and only if there is an accepting run of $\mathcal{M}$ on w. Let $n = |w|$. See Figure 1.

To describe the run, it uses the following boolean variables for every $q \in Q$, for every $\sigma \in \Gamma$, for every $1 \leqslant i, j \leqslant f(|w|)$:

$$X_{q,\sigma,i,j} \qquad \text{and} \qquad X_{\sigma,i,j}$$

Intuitively, $X_{q,\sigma,i,j}$ is true if and only if in step-j the head is in cell-i reading symbol σ and the TM is in state q; and $X_{\sigma,i,j}$ is true if and only if in step-j the content of cell-i is σ.

Essentially the formula φ states the following.

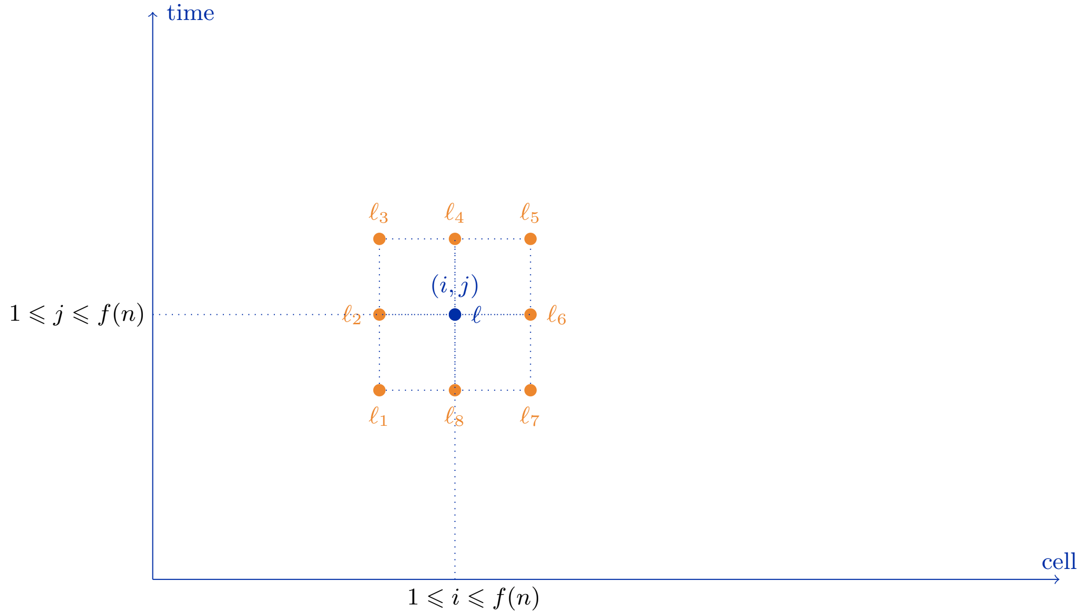


Figure 1: Each point (i,j) is labeled with a symbol $\ell \in (Q \times \Gamma) \cup \Gamma$. If $\ell = (q,\sigma) \in Q \times \Gamma$, it means in time-$j$ the NTM $\mathcal{M}$ is in state q and the head is in cell-i reading symbol σ. If $\ell = \sigma \in \Gamma$, it means in time-j the content of cell-i is σ. The labels ℓ and those in the neighboring points $\ell_1, \ldots, \ell_8$ must obey the transitions in of the NTM $\mathcal{M}$.

- In time-1 the labels of the points $(1,1), \ldots, (1, f(n))$ is the initial configuration. It can be expressed as the following formula.

$$X_{q_0,a_1} \wedge X_{a_2} \wedge \cdots \wedge X_{a_n} \wedge \bigwedge_{i=n+1}^{f(n)} X_{\sqcup} \tag{1}$$

- The accepting state must appear somewhere. It can be expressed as the following formula.

$$\bigvee_{1 \leqslant i,j \leqslant f(n)} \bigvee_{\sigma \in \Gamma} X_{q_{\mathsf{acc}},\sigma,i,j} \tag{2}$$

- For every $1 \leqslant i,j \leqslant f(n)$, the labels in $(i-1,j),(i,j),(i+1,j+1)$ and the labels in $(i-1,j+1),(i,j+1),(i+1,j+1)$ must obey the transitions in $\mathcal{M}$.

  For example, if $(q,\sigma) \to (p,\alpha,\text{left})$ and $(q,\sigma) \to (r,\beta,\text{right})$ are transitions in $\mathcal{M}$, then the formula states the following.

$$\bigwedge_{1 \leqslant i,j \leqslant f(n)} \bigwedge_{\sigma_1,\sigma_2,\sigma_3 \in \Gamma} X_{\sigma_1,i-1,j} \wedge X_{q,\sigma_2,i,j} \wedge X_{\sigma_3,i+1,j} \to \begin{pmatrix} (X_{p,\sigma_1,i-1,j+1} \wedge X_{\alpha,i,j+1} \wedge X_{\sigma_3,i+1,j+1}) \\ \vee \\ (X_{\sigma_1,i-1,j+1} \wedge X_{\beta,i,j+1} \wedge X_{r,\sigma_3,i+1,j+1}) \end{pmatrix} \tag{3}$$

- For every time j, there is exactly one i such that the label of (i,j) is of the form $(q,\sigma) \in$

$Q \times \Gamma$. It can be expressed as the following formula.

$$\bigwedge_{p,q \in Q \text{ and } \sigma,\sigma' \in \Gamma} \bigwedge_{1 \leqslant j \leqslant f(n)} \bigwedge_{1 \leqslant i < i' \leqslant f(n)} X_{q,\sigma,i,j} \to \neg X_{p,\sigma',i',j} \tag{4}$$

$$\wedge \quad \bigwedge_{1 \leqslant j \leqslant f(n)} \bigvee_{q \in \Sigma} \bigvee_{\sigma \in \Gamma} \bigvee_{1 \leqslant i \leqslant f(n)} X_{q,\sigma,i,j} \tag{5}$$

The formula (4) states that there is at most one head and the formula (5) states that there is at least one head.

Formally, the algorithm $\mathcal{A}$ works as follows. On input w, it outputs the formula φ which is the conjunction of the formulas (1)– (5). It is not difficult to show that $w \in L$ if and only if φ is satisfiable. □

Remark 3.13 We note that in the proof of Theorem 3.12, the formula φ produced in the reduction from L to SAT satisfies the following.

The number of accepting run of $\mathcal{M}$ on w = The number of satisfying assignment of φ

Thus, the reduction in Theorem 3.12 is parsimonious.

Remark 3.14 There are two ways to show that a language L is **NP**-hard.

- The first is by definition, i.e., we show that for every language $K \in \mathbf{NP}$, there is a polynomial time reduction from K to L.
- The second is by choosing an appropriate **NP**-hard language, say SAT, and show that there is a polynomial time reduction from SAT to L.

3-SAT	
Input:	A propositional formula φ in 3-CNF.
Task:	Output true, if φ is satisfiable. Otherwise, output false.

Note that we can also view 3-SAT as the language 3-SAT $\stackrel{\text{def}}{=} \{\varphi : \varphi \text{ is satisfiable 3-CNF formula}\}$.

Theorem 3.15 3-SAT *is* **NP***-complete.*

Proof. That is 3-SAT is in **NP** follows immediately from Theorem 3.12. To show that it is **NP**-hard, we reduce it from SAT. On input a CNF formula φ, if it has a clause of length greater than 3:

$$\ell_1 \vee \cdots \vee \ell_k \qquad \text{where } k \geqslant 4$$

split it into two clauses, where z is a new variable:

$$(\ell_1 \vee \cdots \vee \ell_{\lfloor k/2 \rfloor} \vee z) \ \wedge \ (\ell_{\lfloor k/2 \rfloor+1} \vee \cdots \vee \ell_k \vee \neg z)$$

Repeat it on each clause of length $\geqslant 4$ until we get 3-CNF. □

3.3 More NP-complete problems

We need a few terminologies. Let $G = (V, E)$ be a (undirected) graph.

- G is 3-colorable, if we can color the vertices in G with 3 colors (every vertex must be colored with one color) such that no two adjacent vertices have the same color.
- A set $C \subseteq V$ is a clique in G, if every pair of vertices in C are adjacent.
- A set $W \subseteq V$ is a vertex cover, if every edge in E is adjacent to at least one vertex in W.
- A set $I \subseteq V$ is independent, if every pair of vertices in I are non-adjacent.
- A set $D \subseteq V$ is dominating, if every vertex in V is adjacent to at least one vertex in D.

All the following problems are **NP**-complete.

3-COL	
Input:	A (undirected) graph $G = (V, E)$.
Task:	Output true, if G is 3-colorable. Otherwise, output false.

CLIQUE	
Input:	A (undirected) graph $G = (V, E)$ and an integer $k \geqslant 0$ in binary form.
Task:	Output true, if G has a clique of size $\geqslant k$. Otherwise, output false.

IND-SET	
Input:	A (undirected) graph $G = (V, E)$ and an integer $k \geqslant 0$ in binary form.
Task:	Output true, if G has an independent set of size $\geqslant k$. Otherwise, output false.

VERT-COVER	
Input:	A (undirected) graph $G = (V, E)$ and an integer $k \geqslant 0$ in binary form.
Task:	Output true, if G has a vertex cover of size $\leqslant k$. Otherwise, output false.

DOM-SET	
Input:	A (undirected) graph $G = (V, E)$ and an integer $k \geqslant 0$ in binary form.
Task:	Output true, if G has an dominating set of size $\leqslant k$. Otherwise, output false.

3.4 coNP-complete languages

Analogous to **NP**-complete, we can also define **coNP**-complete problems.

Definition 3.16 Let K be a language.

- K is **coNP**-*hard*, if for every $L \in$ **coNP**, $L \leqslant_p K$.
- K is **coNP**-*complete*, if $K \in$ **coNP** and K is **coNP**-hard.

Theorem 3.17 *For every language K over the alphabet Σ, K is* **NP***-complete if and only if its complement $\overline{K}$ is* **coNP***-complete, where $\overline{K} \stackrel{def}{=} \Sigma^* - K$.*

Proof. Let K be a language. By definition, K is **NP** if and only if $\overline{K}$ is in **coNP**.

To show the hardness part, we first note that $L \leqslant_p K$ if and only if $\overline{L} \leqslant_p \overline{K}$. Thus, if K is **NP**-hard, by definition, $L \leqslant_p K$, for every language $L \in$ **NP**, which means $\overline{L} \leqslant_p \overline{K}$, for every language $L \in$ **NP**. Therefore, $L \leqslant_p \overline{K}$, for every language $L \in$ **coNP**, i.e., $\overline{K}$ is **coNP**-hard. The direction that $\overline{K}$ is **coNP**-hard implies K is **NP**-hard is similar. □

Corollary 3.18 $\overline{\mathsf{SAT}} \stackrel{def}{=} \{\varphi : \varphi$ *is not satisfiable*$\}$ *is* **coNP***-complete.*

4 The class NL and PSPACE

4.1 NL-complete languages

Let $F : \Sigma^* \to \Sigma^*$ be a function. We say that F is computable in logarithmic space, if there is a 3-tape DTM $\mathcal{M}$ such that on input word w, it works as follows.

- Tape 1 contains the input word w and its content never changes.
- There is a constant c such that $\mathcal{M}$ uses only $c \log |w|$ space in tape 2.
- The head in tape 3 can only "write" and move right, i.e., once it writes a symbol to a cell, the content of that cell will not change.

Tape 1 is called the *input tape*, tape 2 the *work tape* and tape 3 the *output tape.*

Definition 4.19 A language L is *log-space reducible* to another language K, denoted by $L \leqslant_{\log} K$, if there is a function $F : \Sigma^* \to \Sigma^*$ computable in logarithmic space such that for every $w \in \Sigma^*$, $w \in L$ if and only if $F(w) \in K$.

Remark 4.20 The relation $\leqslant_{\log}$ is transitive in the sense that if $L_1 \leqslant_{\log} L_2$ and $L_2 \leqslant_{\log} L_3$, then $L_1 \leqslant_{\log} L_3$.

Definition 4.21 Let K be a language.

- K is **NL**-*hard*, if for every language $L \in \mathbf{NL}$, $L \leqslant_{\log} K$.
- K is **NL**-*complete*, if $K \in \mathbf{NL}$ and K is **NL**-hard.

Define the following language PATH.

$$\mathsf{PATH} \stackrel{\mathsf{def}}{=} \{(G, s, t) : G \text{ is } \textit{directed} \text{ graph and there is a path in } G \text{ from vertex } s \text{ to vertex } t\}$$

Theorem 4.22 PATH *is* **NL**-*complete.*

Theorem 4.23 (Savitch 1970) NL $\subseteq$ DSPACE$[\log^2 n]$.

Proof. To prove Theorem 4.23, it suffices to show that $\mathsf{PATH} \in$ DSPACE$[\log^2 n]$.

Consider procedure $\textsc{Check}_G$ that decides whether there is a path from vertex u to vertex v with length at most 2^k. When running $\textsc{Check}_G(u, x, k-1)$ and $\textsc{Check}_G(x, v, k-1)$, Procedure $\textsc{Check}_G$ can use the same space. So it uses only $O(k \log n)$ space. Since deciding whether there is a path from u to v can be done using $\textsc{Check}_G$ with $k = \lceil \log n \rceil$, we can decide PATH using $O(\log^2 n)$ space in total.

□

Theorem 4.24 (Immerman 1988 and Szelepcsényi 1987) NL = coNL.

Proof. Consider the complement of PATH:

$$\overline{\mathsf{PATH}} \stackrel{\mathsf{def}}{=} \{(G, s, t) : G \text{ is } \textit{directed} \text{ graph and there is } \textit{no} \text{ path in } G \text{ from vertex } s \text{ to vertex } t\}$$

Since $\overline{\mathsf{PATH}}$ is **coNL**-complete, to prove Theorem 4.24, it suffices to show that $\overline{\mathsf{PATH}} \in \mathbf{NL}$.

We sketch an **NL** algorithm for $\overline{\mathsf{PATH}}$. On input graph G and two vertices s and t, do the following.

Procedure CHECK$_G$

Input: (u, v, k) where u and v are two vertices in G, and k is an integer $\geqslant 0$.
Task: Return true, if there is a path in G of length $\leqslant 2^k$ from u to v. Otherwise, return false.

```
1: if k = 0 then
2:   return true iff (u = v or (u, v) is an edge in G).
3: for all vertex x in G do
4:   b := CHECK_G(u, x, k − 1).
5:   if b = true then
6:     b := CHECK_G(x, v, k − 1).
7:     if b = true then
8:       return true.
9: return false.
```

- Assume that we can count the number of vertices reachable from the vertex s in logarithmic space, which we denote by $\wp$.
- For every vertex u in G, guess whether u is reachable from s.
  - If the guess is "reachable," guess the path from s to u (in logarithmic space).
  - Decrease the value $\wp$ by 1.
- ACCEPT if and only if the vertex t is not one of the "reachable" vertices and the value $\wp$ is zero.

We will now show how to count the number of vertices reachable from s in logarithmic space. The idea is to count $\wp_k$, the number of vertices from s in distance k for each $k \in \{1, \ldots, n\}$ where n is the number of vertices in G. It works by recursion on k.

- If $k = 1$, count the number of outgoing edges from the vertex s.
- If $k \geqslant 2$:

  Let $\wp_{k-1}$ be the number of vertices reachable from s in distance $\leqslant k - 1$.

  Initialize $\wp_k$ with 0.

  For each vertex v in G, guess whether it is reachable from s in distance $\leqslant k$.

  If the guess is "reachable," then guess the path from s to v of length $\leqslant k$ and increase $\wp_k$ by 1.

  If the guess is "not reachable," we need to verify that indeed there is no path from s to v of length $\leqslant k$ as follows.

  - Let z be a dummy variable to store $\wp_{k-1}$.
  - For every vertex w in G, guess whether w is reachable from s of length $\leqslant k - 1$.
    * If the guess is "reachable," guess the path from s to u (in logarithmic space).
    * Check that indeed there is no edge (u, w) in G.
    * Decrease the value z by 1.

  After the iteration, we also insist that the value z must be zero. Otherwise, the algorithm REJECTS immediately.

Note that $\wp_n$ is the number of vertices reachable from vertex s. Note also that to compute $\wp_k$, it suffices to remember only $\wp_{k-1}$. Thus, the algorithm uses only logarithmic space. □

4.2 PSPACE-complete languages

Definition 4.25 Let K be a language.

- K is **PSPACE**-*hard*, if for every language $L \in$ **PSPACE**, $L \leqslant_p K$.
- K is **PSPACE**-*complete*, if $K \in$ **PSPACE** and K is **PSPACE**-hard.

Quantified Boolean formulas (QBF) are formulas of the form:

$$Q_1 x_1 \ Q_2 x_2 \ \cdots \ Q_n x_n \ \varphi(x_1, \ldots, x_n)$$

where each $Q_i \in \{\forall, \exists\}$ and $\varphi(x_1, \ldots, x_n)$ is a Boolean formula with variables $x_1, \ldots, x_n$.

The intuitive meaning of each Q_i is as follows.

- $\forall x \ \psi$ means that for all $x \in \{\mathsf{true}, \mathsf{false}\}$, ψ is true.
- $\exists x \ \psi$ means that there is $x \in \{\mathsf{true}, \mathsf{false}\}$ such that ψ is true.

We define the problem TQBF:

TQBF	
Input:	A QBF φ.
Task:	Return true, if φ is true. Otherwise, return false.

As usual, it can be viewed as a language TQBF $\stackrel{\mathsf{def}}{=} \{\psi : \psi \text{ is a true QBF}\}$. Note also that the usual Boolean formula can be viewed as a QBF, where each Q_i is $\exists$. Thus, TQBF is a more general problem than SAT.

Theorem 4.26 (Stockmeyer and Meyer 1973) TQBF *is* **PSPACE**-*complete.*

Theorems 4.27 and 4.28 below are the polynomial space analog of Theorem 4.23 and 4.24, respectively. In fact, they can be easily generalized to the so called *time* and *space constructible functions*. See Appendix B.

Theorem 4.27 (Savitch 1970) $\text{NSPACE}[n^k] \subseteq \text{DSPACE}[n^{2k}]$.

Theorem 4.28 (Immerman 1988 and Szelepcsényi 1987) $\text{NSPACE}[n^k] = \text{coNSPACE}[n^k]$.

Note that Theorem 4.27 implies **PSPACE** = **NPSPACE** = **coNPSPACE**.

5 P-complete languages

Boolean circuits. Let $n \in \mathbb{N}$, where $n \geqslant 1$. An n-input *Boolean circuit* C is a directed acyclic graph with n *source* vertices (i.e., vertices with no incoming edges) and 1 *sink* vertex (i.e., vertex with no outgoing edge).

The source vertices are labelled with $x_1, \ldots, x_n$. The non-source vertices, called *gates*, are labelled with one of $\wedge, \vee, \neg$. The vertices labelled with $\wedge$ and $\vee$ have two incoming edges, whereas the vertices labelled with $\neg$ have one incoming edge. The *size* of C, denoted by $|C|$, is the number of vertices in C.

On input $w = x_1 \cdots x_n$, where each $x_i \in \{0, 1\}$, we write $C(w)$ to denote the output of C on w, where $\wedge, \vee, \neg$ are interpreted as "and," "or" and "negation," respectively and 0 and 1 as false and true, respectively.

(Boolean) straight line programs. It is sometimes more convenient to view a boolean circuit as a straight line program. The following is an example of straight line program, where the input is $w = x_1 \cdots x_n$.

1: $p_1 := x_1 \wedge x_3$.
2: $p_2 := \neg x_4$.
3: $p_3 := p_1 \vee p_2$.
$\vdots$
ℓ: $p_\ell := p_i \wedge p_j$.

Intuitively, straight line programs are programs without **if** branch and **while** loop, hence, the name "straight line" programs. It is assumed that such program always outputs the value in the variable in the last line. In our example above, it outputs the value of variable p_ℓ.

Define the following problem.

CIRCUIT-EVAL	
Input:	An n input boolean circuit C and $w \in \{0, 1\}^n$.
Task:	Output $C(w)$.

It can also be defined as the language CIRCUIT-EVAL $\stackrel{\text{def}}{=} \{(C, w) : C(w) = 1\}$.

For our proof of Theorem 5.29 below, it is also convenient to assume that vertices labelled with $\wedge$ and $\vee$ can have more than 2 incoming edges.

Theorem 5.29 CIRCUIT-EVAL *is* **P***-complete via log-space reductions.*

Proof. Follows the reduction for the **NP**-completeness of SAT. □

6 Alternating Turing machines

6.1 Definition

A 1-tape *alternating Turing machine* (ATM) is a system $\mathcal{M} = \langle \Sigma, \Gamma, Q, U, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}, \delta \rangle$, where each component is as follows.

- $\Sigma = \{0, 1\}$ and $\Gamma = \{0, 1, \sqcup\}$ are the input and tape alphabets, respectively.
- Q is a finite set of states.
- $U \subseteq Q$ is a finite subset of Q.
- $q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}$ are the initial state, accepting state and rejecting state, respectively.
- $\delta \subseteq (Q - \{q_{\mathsf{acc}}, q_{\mathsf{rej}}\}) \times \Gamma \times Q \times \Gamma \times \{\mathtt{Left}, \mathtt{Right}\}$.

Note that ATM is very much like NTM, except that it has one extra component U. The states in U are called *universal* states, and the states in $Q - U$ are called *existential* states.

The notions of *initial/halting/accepting/rejecting* configuration are defined similarly as in NTM/DTM. A configuration C is called *existential/universal* configuration, if the the state in C is an existential/universal state. The notion of "one step computation" $C \vdash C'$ for ATM is also similar to the one for DTM/NTM. When $C \vdash C'$, we say that C' is one of the next configuration of C (w.r.t. $\mathcal{M}$).

On input word w, *the run of* $\mathcal{M}$ *on* w is a *tree* T where each node in the tree is labelled with a configuration of $\mathcal{M}$ according to the following rules.

- The root node of T is labelled with the initial configuration of $\mathcal{M}$ on w.
- Every other node x in T is labelled as follows.

  If x is labelled with a configuration C and $C_1, \ldots, C_n$ are all the next configurations of C, then x has n children $y_1, \ldots, y_n$ labelled with $C_1, \ldots, C_n$, respectively.

Note that if x is labelled with C that does not have next configuration, then it is a leaf node, i.e., it does not have any children.

Let T be the run of $\mathcal{M}$ on w and let x be a node in T. We say that x *leads to acceptance*, if the following holds.

- x is a leaf node labelled with an accepting configuration.
- If x is labelled with an existential configuration, then one of its children leads to acceptance.
- If x is labelled with a universal configuration, then all of its children lead to acceptance.

We say that T is *accepting run*, if its root node leads to acceptance. The ATM $\mathcal{M}$ accepts w, if the run of $\mathcal{M}$ on w is accepting run. As before, $L(\mathcal{M}) \stackrel{\mathsf{def}}{=} \{w : \mathcal{M} \text{ accepts } w\}$.

Note that NTM is simply ATM where all the states are existential, and DTM is simply NTM where every configuration (except the accepting/rejecting configuration) has exactly one next configuration. The generalization of ATM to multiple tapes is straightforward.

6.2 Time and space complexity for ATM

Let $\mathcal{M}$ be a ATM, $w \in \Sigma^*$, $t \in \mathbb{N}$ and let $f : \mathbb{N} \to \mathbb{N}$ be a function.

- $\mathcal{M}$ *decides* w *in time* t *(or, in* t *steps)*, if the run of $\mathcal{M}$ on w has depth at most t.
- $\mathcal{M}$ *decides* w *in space* t *(or, uses* t *cells/space)*, if in the run of $\mathcal{M}$ on w, every node is labelled with configuration of length t.

- $\mathcal{M}$ *runs in time/space* $O(f(n))$, if there is $c > 0$ such that for sufficiently long word w, $\mathcal{M}$ decides w in time/space $c \cdot f(|w|)$.
- $\mathcal{M}$ *decides a language* L *in time/space* $O(f(n))$, if $\mathcal{M}$ runs in time/space $O(f(n))$ and $L(\mathcal{M}) = L$.
- $\textsc{Atime}[f(n)] \stackrel{\mathsf{def}}{=} \{L : \text{there is ATM } \mathcal{M} \text{ that decides } L \text{ in time } O(f(n))\}$.
- $\textsc{Aspace}[f(n)] \stackrel{\mathsf{def}}{=} \{L : \text{there is ATM } \mathcal{M} \text{ that decides } L \text{ in space } O(f(n))\}$.

Analogous to the DTM/NTM, we can define the classes of languages accepted by ATM run in algorithmic/polynomial/exponential time/space.

$$\begin{aligned}
\mathbf{AL} &\stackrel{\mathsf{def}}{=} \{L : \text{there is ATM } \mathcal{M} \text{ that decides } L \text{ in space } O(\log n)\} \\
\mathbf{AP} &\stackrel{\mathsf{def}}{=} \bigcup_{f(n)=\mathsf{poly}(n)} \textsc{Atime}[f(n)] \\
\mathbf{APSPACE} &\stackrel{\mathsf{def}}{=} \bigcup_{f(n)=\mathsf{poly}(n)} \textsc{Aspace}[f(n)] \\
\mathbf{AEXP} &\stackrel{\mathsf{def}}{=} \bigcup_{f(n)=\mathsf{poly}(n)} \textsc{Atime}[2^{f(n)}]
\end{aligned}$$

The following lemma links time/space complexity classes for ATM with those for DTM.

Lemma 6.30 *Let* $T : \mathbb{N} \to \mathbb{N}$ *and* $S : \mathbb{N} \to \mathbb{N}$ *such that* $T(n) \geqslant n$ *and* $S(n) \geqslant \log n$*, for every* n*.*

(a) $\textsc{Atime}[T(n)] \subseteq \textsc{Dspace}[T(n)]$.

(b) $\textsc{Dspace}[S(n)] \subseteq \textsc{Atime}[S(n)^2]$.

(c) $\textsc{Aspace}[S(n)] \subseteq \textsc{Dtime}[2^{O(S(n))}]$.

(d) $\textsc{Dtime}[T(n)] \subseteq \textsc{Aspace}[\log T(n)]$.

Proof. (a) and (c) is by straightforward simulation of ATM with DTM. (b) is similar to the proof of Savitch's theorem. (d) is similar to the proof of Theorem 5.29 below, i.e., by viewing the computation of DTM as a boolean circuit. □

Theorem 6.31 (Chandra, Kozen, Stockmeyer 1981)

- $\mathbf{AL} = \mathbf{P}$.
- $\mathbf{AP} = \mathbf{PSPACE}$.
- $\mathbf{APSPACE} = \mathbf{EXP}$.
- $\mathbf{AEXP} = \mathbf{EXPSPACE}$.
- $\cdots$.

7 The polynomial hierarchy

For every integer $i \geqslant 1$, the class $\mathbf{\Sigma}_i^p$ is defined as follows. A language $L \subseteq \{0,1\}^*$ is in $\mathbf{\Sigma}_i^p$, if there is a polynomial $q(n)$ and a polynomial time DTM $\mathcal{M}$ such that for every $w \in \{0,1\}^*$, $w \in L$ if and only if the following holds.

$$\exists y_1 \in \{0,1\}^{q(|w|)} \ \forall y_2 \in \{0,1\}^{q(|w|)} \ \cdots \ Q y_i \in \{0,1\}^{q(|w|)} \ \mathcal{M} \text{ accepts } (w, y_1, \ldots, y_i) \qquad (6)$$

where $Q = \exists$, if i is odd and $Q = \forall$, if i is even.

The class $\mathbf{\Pi}_i^p$ is defined as above, but the sequence of quantifiers in (6) starts with $\forall$. Alternatively, it can also be defined as $\mathbf{\Pi}_i^p \stackrel{\text{def}}{=} \{\overline{L} : L \in \mathbf{\Sigma}_i^p\}$. Note that $\mathbf{NP} = \mathbf{\Sigma}_1^p$ and $\mathbf{coNP} = \mathbf{\Pi}_1^p$.

Remark 7.32 The class $\mathbf{\Sigma}_i^p$ can also be defined as follows. A language L is in $\mathbf{\Sigma}_i^p$, if there is a polynomial time ATM $\mathcal{M}$ that decides L such that for every input word $w \in \{0,1\}^*$, the run of $\mathcal{M}$ on w can be divided into i layers. Each layer consists of nodes of the same depth in the run. (Recall that the run of an ATM is a tree.) In the first layer all nodes are labeled with existential configurations, in the second layer with universal configurations, and so on. It is not difficult to show that this definition is equivalent to the one above.

The *polynomial time hierarchy* (or, in short, *polynomial hierarchy*) is defined as the following class.

$$\mathbf{PH} \stackrel{\text{def}}{=} \bigcup_{i=1}^{\infty} \mathbf{\Sigma}_i^p$$

Note that $\mathbf{PH} \subseteq \mathbf{PSPACE}$.

It is conjectured that $\mathbf{\Sigma}_1^p \subsetneq \mathbf{\Sigma}_2^p \subsetneq \mathbf{\Sigma}_3^p \subsetneq \cdots$. In this case, we say that *the polynomial hierarchy does not collapse.* We say that *the polynomial hierarchy collapses*, if there is i such that $\mathbf{PH} = \mathbf{\Sigma}_i^p$, in which case we also say that *the polynomial hierarchy collapses to level i.*

We define the notion of hardness and completeness for each $\mathbf{\Sigma}_i^p$ as follows. For $i \geqslant 1$, a language K is $\mathbf{\Sigma}_i^p$-*hard*, if for every $L \in \mathbf{\Sigma}_i^p$, $L \leqslant_p K$. It is $\mathbf{\Sigma}_i^p$-*complete*, if it is in $\mathbf{\Sigma}_i^p$ and it is $\mathbf{\Sigma}_i^p$-hard. The same notion can be defined analogously for $\mathbf{PH}$ and each $\mathbf{\Pi}_i^p$.

Define the language Σ_i-SAT as consisting of true QBF of the form:

$$\exists \bar{x}_1 \ \forall \bar{x}_2 \ \cdots \ Q \bar{x}_i \ \varphi(\bar{x}_1, \ldots, \bar{x}_i)$$

where $\varphi(\bar{x}_1, \ldots, \bar{x}_i)$ is quantifier-free Boolean formula and $Q = \exists$, if i is odd, and $Q = \forall$, if i is even. Here $\bar{x}_1, \ldots, \bar{x}_i$ are all vectors of boolean variables. In other words, Σ_i-SAT is a subset of TQBF where the number of quantifier alternation is limited to $(i-1)$. The language Π_i-SAT is defined analogously with the starting quantifiers being $\forall$.

Theorem 7.33

- *For every $i \geqslant 1$, Σ_i-SAT is $\mathbf{\Sigma}_i^p$-complete and Π_i-SAT is $\mathbf{\Pi}_i^p$-complete.*
- *If $\mathbf{\Sigma}_i^p = \mathbf{\Pi}_i^p$ for some $i \geqslant 1$, then the polynomial hierarchy collapses.*
- *If there is language that is $\mathbf{PH}$-complete, then the polynomial hierarchy collapses.*

8 The complexity of counting

8.1 The class FP

We denote by **FP** the class of functions $f : \{0,1\}^* \to \mathbb{N}$ computable by polynomial time DTM. Here the convention is that a natural number is always represented in binary form. So, when we say that a DTM $\mathcal{M}$ computes a function $f : \{0,1\}^* \to \mathbb{N}$, on input word w, the output of $\mathcal{M}$ on w is $f(w)$ in the binary representation.

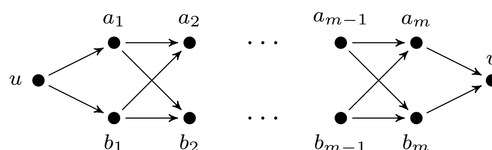
Let ♯CYCLE be the following problem.

♯CYCLE	
Input:	A directed graph G.
Task:	Output the number of cycles in G.

As before, ♯CYCLE can also be viewed as a function. Note also that the number of cycles in a graph with n vertices is at most exponential in n, thus, its binary representation only requires polynomially many bits.

Theorem 8.34 *If ♯CYCLE is in* **FP**, *then* $\mathbf{P} = \mathbf{NP}$.

Proof. Let G be a (directed) graph with n vertices. We construct a graph G' obtained by replacing every edge (u, v) in G with the following gadget:



Note that every simple cycle in G of length ℓ becomes $(2^m)^\ell$ cycles in G'. Now, let $m \stackrel{\text{def}}{=} n \log n$.

It is not difficult to show that G has a hamiltonian cycle (i.e., a simple cycle of length n) if and only if G' has more than $n^{(n^2)}$ cycles. So, if ♯CYCLE $\in$ **FP**, then checking hamiltonian cycle can be done is in **P**. □

Note that checking whether a graph has a cycle itself can be done in polynomial time. However, as Theorem 8.34 above states, it is unlikely that counting the number of cycles can be done in polynomial time.

8.2 The class ♯P

Definition 8.35 A function $f : \{0,1\}^* \to \mathbb{N}$ is in ♯**P**, if there is a polynomial $q(n)$ and a polynomial time DTM $\mathcal{M}$ such that for every word $w \in \{0,1\}^*$, the following holds.

$$f(w) \quad = \quad |\{y : \mathcal{M} \text{ accepts } (w,y) \text{ and } y \in \{0,1\}^{q(|w|)}\}|$$

Alternatively, we can say that f is in ♯**P**, if there is a polynomial time NTM $\mathcal{M}$ such that for every word $w \in \{0,1\}^*$, $f(w) =$ the number of accepting runs of $\mathcal{M}$ on w.

For a function $f : \{0,1\}^* \to \mathbb{N}$, the language associated with the function f, denoted by O_f, is defined as $O_f \stackrel{\text{def}}{=} \{(w,i) : \text{the } i^{\text{th}} \text{ bit of } f(w) \text{ is } 1\}$. When we say that *a TM $\mathcal{M}$ has oracle access to a function f*, we mean that it has oracle access to the language O_f.

We define $\mathbf{FP}^f$ as the class of functions $g : \{0,1\}^* \to \mathbb{N}$ computable by a polynomial time DTM with oracle access to f.

Definition 8.36 Let $f : \{0,1\}^* \to \mathbb{N}$ be a function.

- f is $\sharp\mathbf{P}$-*hard*, if $\sharp\mathbf{P} \subseteq \mathbf{FP}^f$, i.e., every function in $\sharp\mathbf{P}$ is computable by a polynomial time DTM with oracle access to f.
- f is $\sharp\mathbf{P}$-*complete*, if $f \in \sharp\mathbf{P}$ and f is $\sharp\mathbf{P}$-hard.

Let $\sharp$SAT be the following problem.

$\sharp$SAT	
Input:	A boolean formula φ.
Task:	Output the number of satisfying assignments for φ.

As before, the output numbers are to be written in binary form. We can also view $\sharp$SAT as a function $\sharp\mathsf{SAT} : \{0,1\}^* \to \mathbb{N}$, where $\sharp\mathsf{SAT}(\varphi)$ = the number of satisfying assignment for φ.

Theorem 8.37 *$\sharp$SAT is $\sharp\mathbf{P}$-complete.*

Proof. Cook-Levin reduction (to prove the **NP**-hardness of SAT) is parsimonious. □

There are usually two ways to prove a certain function is $\sharp\mathbf{P}$-hard, as stated in Remark 8.38 and 8.39 below.

Remark 8.38 Let f_1 and f_2 be functions from $\{0,1\}^*$ to $\mathbb{N}$. Suppose L_1 and L_2 be languages in **NP** such that f_1 and f_2 are the functions for the number of certificates for L_1 and L_2, respectively. That is, for every word $w \in \{0,1\}^*$,

$$f_i(w) \quad = \quad \text{the number of certificates of } w \text{ in } L_i, \qquad \text{for } i = 1, 2.$$

If f_1 is $\sharp\mathbf{P}$-hard and there is a parsimonious (polynomial time) reduction from L_1 to L_2, then f_2 is $\sharp\mathbf{P}$-hard.

Remark 8.39 Let f and g be two functions from $\{0,1\}^*$ to $\mathbb{N}$. If f is $\sharp\mathbf{P}$-hard and $f \in \mathbf{FP}^g$, then g is $\sharp\mathbf{P}$-hard.

Since there is a parsimonious reduction from SAT to 3-SAT, by Theorem 8.37 and Remark 8.38, we have the following corollary.

Corollary 8.40 *$\sharp$3-SAT is $\sharp\mathbf{P}$-complete.*

Corollary 8.40 can also be proved by showing $\sharp\mathsf{SAT} \in \mathbf{FP}^{\sharp\text{3-SAT}}$.

8.3 The complexity of computing the permanent

The definition of permanent

For an integer $n \geqslant 1$, let $[n] = \{1, \ldots, n\}$. The *permanent* of an $n \times n$ matrix A over integers is defined as:

$$\mathrm{per}(A) \quad \stackrel{\mathsf{def}}{=} \quad \sum_{\sigma} \prod_{i=1}^{n} A_{i,\sigma(i)}$$

where σ ranges over all permutation on $[n]$. Here $A_{i,j}$ denotes the entry in row i and column j in matrix A.

Consider the following problem.

PERM	
Input:	A square matrix A over integers.
Task:	Output the permanent of A.

We denote it by 0|1-PERM, when the entries in the input matrix A are restricted to 0 or 1.

Theorem 8.41 (Valiant 1979) 0|1-PERM *is* $\sharp\mathbf{P}$*-complete.*

To show that 0|1-PERM is in $\sharp\mathbf{P}$, consider the following algorithm.

Input: A 0-1 matrix A.
1: Guess a permutation σ on $[n]$, i.e., for each $i \in [n]$, guess a value $v_i \in [n]$.
2: If the guessed σ is not a permutation, REJECT.
3: Compute the value $\prod_{i=1}^{n} A_{i,\sigma(i)}$.
4: ACCEPT if and only if the value is 1.

It is obvious that on input A, the number of accepting runs is the same as $\text{per}(A)$.

A combinatorial view of permanent

Let $G = (V, E, w)$ be a complete directed graph, i.e., $E = V \times V$, and each edge (u, v) has a weight $w(u, v) \in \mathbb{Z}$. We write a (simple) cycle as a sequence $p = (u_1, \ldots, u_\ell)$, and its weight is defined as:

$$w(p) \stackrel{\text{def}}{=} w(u_1, u_2) \cdot w(u_2, u_3) \cdot \ldots \cdot w(u_{\ell-1}, u_\ell) \cdot w(u_\ell, u_1)$$

A loop (u, u) is considered a cycle.

A *cycle cover* of G is a set $R = \{p_1, \ldots, p_k\}$ of pairwise disjoint cycles such that for every vertex $u \in V$, there is a cycle $p_j \in R$ such that u appears in p_j. The weight R is defined as:

$$w(R) \stackrel{\text{def}}{=} \prod_{p_j \in R} w(C_j)$$

Note that a cycle or a cycle cover can also be viewed as a set of edges.

Assuming that the vertices in G are $\{1, \ldots, n\}$, let A be the adjacency matrix of G, i.e., A is an $(n \times n)$ matrix over $\mathbb{Z}$ such that $A_{i,j} = w(i, j)$.

A permutation $\sigma = (d_{1,1}, \ldots, d_{1,k_1}), \ldots, (d_{l,1}, \cdots, d_{l,k_l})$ on $[n]$ can be viewed as a cycle cover whose weight is exactly the value $\prod_{i \in [n]} A_{i,\sigma(i)}$. Thus, we have the equation:

$$\text{per}(A) = \sum_{R \text{ is a cycle cover of } G} w(R)$$

8.4 Reduction from 3-SAT to cycle cover

In this section we will show how to encode 3-SAT as the cycle cover problem.

An overview of the main idea

Let Ψ be a formula in 3-CNF. Let $x_1, \ldots, x_n$ be the variables and $C_1, \ldots, C_m$ be the clauses. We will construct a complete directed graph $G = (V, E, w)$, where the weight of each edge can be arbitrary integer and every boolean assignment $\phi : \{x_1, \ldots, x_n\} \to \{0, 1\}$ is associated with a set F_ϕ of cycle covers of G such that the following holds.

- For two different assignments ϕ_1, ϕ_2, the sets F_{ϕ_1} and F_{ϕ_2} are disjoint.
- If ϕ is a satisfying assignment for Ψ, the total weight of cycle covers in F_ϕ is 4^{3m}, i.e.,

$$\sum_{R \in F_\phi} w(R) \quad = \quad 4^{3m}$$

- If ϕ is not a satisfying assignment for Ψ, the total weight of cycle covers in F_ϕ is 0, i.e.,

$$\sum_{R \in F_\phi} w(R) \quad = \quad 0$$

- The total weight of cycle covers not in any F_ϕ is 0, i.e.,

$$\sum_{R \notin F_\phi \text{ for any } \phi} w(R) \quad = \quad 0$$

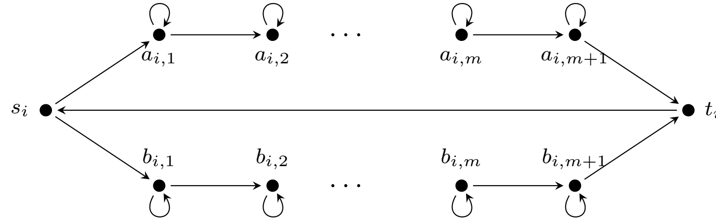
If A is the adjacency matrix of G, it is clear that:

$$\text{per}(A) \quad = \quad 4^{3m} \times (\text{the number of satisfying assignment for } \Psi)$$

The construction of the graph G

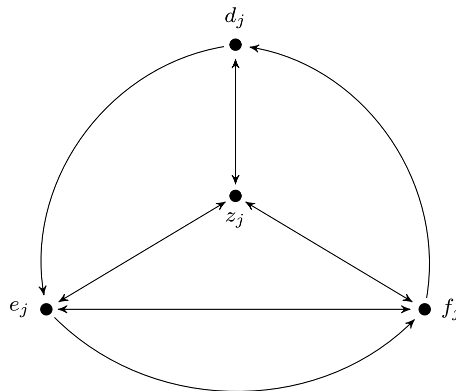
In the following we will draw an edge with a label indicating its weight. If the label is missing, it means the weight is 1. When an edge is not drawn, it means the weight is 0.

Variable gadget. For each variable x_i, we have the following "variable gadget":



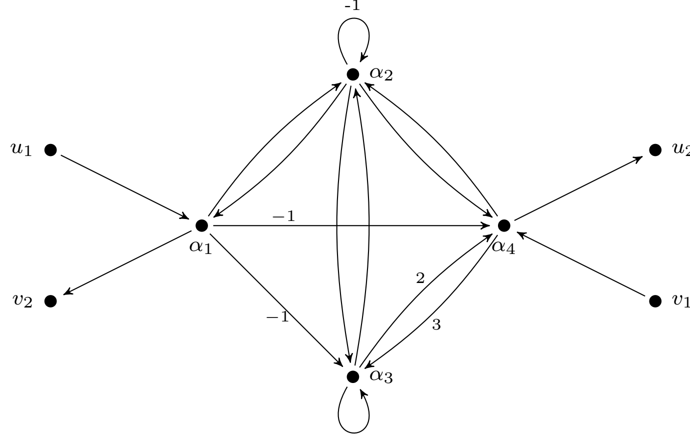
The upper edges, i.e., $(a_{i,1}, a_{i,2}), \ldots, (a_{i,m}, a_{i,m+1})$, are called the *external "true" edges* of x_i, and the lower edges, i.e., $(b_{i,1}, b_{i,2}), \ldots, (b_{i,m}, b_{i,m+1})$, the *external "false" edges* of x_i.

Clause gadget. For each clause C_j, we have the following "clause gadget":



The "outer" edges $(d_j, e_j), (e_j, f_j), (f_j, d_j)$ are intended to represent the literals in C_j. If ℓ_1, ℓ_2, ℓ_3 are the literals in C_j, then their associated edges are $(d_j, e_j), (e_j, f_j), (f_j, d_j)$, respectively. To avoid clutter, we will call those edges ℓ_1-edge, ℓ_2-edge and ℓ_3-edge, respectively.

The XOR operator. We also have the "XOR operator" between two edges (u_1, u_2) and (v_1, v_2):



Definition 8.42 Let H be a graph, and let (u_1, u_2) and (v_1, v_2) are two non-adjacent edges in H.

- For a cycle cover R of H, we say that R *respects the property* $(u_1, u_2) \oplus (v_1, v_2)$, if R contains exactly one of (u_1, u_2) or (v_1, v_2).
- Let H' denotes the graph obtained from H by replacing the edges $(u_1, u_2), (v_1, v_2)$ with the edges in the XOR operator above.

  A cycle cover R' of H' is *an associated cycle cover* of R, if it satisfies the following condiiton.

  - If R contains (u_1, u_2), then R' contains a path from u_1 to u_2.
  - If R contains (v_1, v_2), then R' contains a path from v_1 to v_2.
  - $R \setminus \{(u_1, u_2), (v_1, v_2)\} \subseteq R'$.

Lemma 8.43 *Let H, H', R and $(u_1, u_2), (v_1, v_2)$ be as in Definition 8.42. Then, the following holds.*

$$\sum_{R' \text{ is associated with } R} w(R') = \begin{cases} 4w(R), & \text{if } R \text{ respects } (u_1, u_2) \oplus (v_1, v_2) \\ 0, & \text{otherwise} \end{cases}$$

Constructing the graph G. The graph G is defined as the disjoint union of all the variable and clause gadgets and the following additional edges to connect them: For every clause C_j, for every literal ℓ in C_j, if $\ell = x_i$, we "connect" the ℓ-edge in the clause gadget of C_j with the edge $(a_{i,j}, a_{i,j+1})$ via the XOR operator; and if $\ell = \neg x_i$, we "connect" it with the edge $(b_{i,j}, b_{i,j+1})$.

For an assignment $\phi : \{x_1, \ldots, x_n\} \to \{0, 1\}$, we say that *a cycle cover R is associated with* ϕ, if the following holds for every variable x_i.

- If $\phi(x_i) = 1$, the cycle $(s_i, a_{i,1}, \ldots, a_{i,m+1}, t_i)$ is in R.
- If $\phi(x_i) = 0$, the cycle $(s_i, b_{i,1}, \ldots, b_{i,m+1}, t_i)$ is in R.

Lemma 8.44 *For every assignment $\phi : \{x_1, \ldots, x_n\} \to \{0, 1\}$, the following holds.*

$$\sum_{R \text{ is associated with } \phi} w(R) = \begin{cases} 4^{3m}, & \text{if } \phi \text{ is satisfying assignment for } \Psi \\ 0, & \text{if } \phi \text{ is not} \end{cases}$$

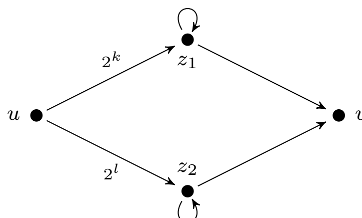
Combining Lemmas 8.43 and 8.44, it is immediate that the following holds.

$$\text{per}(A) \quad = \quad 4^{3m} \times (\text{the number of satisfying assignments for } \Psi)$$

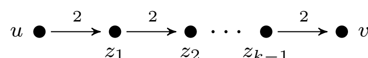
Here A is the adjacency matrix of G.

8.5 Reduction from matrices over $\mathbb{Z}$ to matrices over $\{0, 1\}$

Reduction to matrices over integers of the form -2^k, 0 or 2^k. For each edge (u, v) with weight $2^k + 2^l$, we can replace it with 2 "parallel" edges with weights 2^k and 2^l, respectively.



Reduction to matrices over integers of the form -1, 0 or 1. For each edge (u, v) with weight 2^k, we can replace it with k "series" edges, each with weights 2.



Each weight 2 edge can be further reduced to weight 1 edge as above.

Reduction to matrices over $\{0, 1\}$ (but on modular arithmetic). The permanent of an $n \times n$ matrix A over $\{-1, 0, 1\}$ can only in between $-n!$ and $n!$. Let $m = n^2$. Since $2^m + 1 > 2n!$, it is sufficient to compute per(A) in $\mathbb{Z}_{2^m+1}$. Since $-1 \equiv 2^m \pmod{2^m + 1}$, we can replace each -1 with 2^m, which can then be reduced to 1 as above.

8.6 ♯P-hardness of PERM – Putting all the pieces together

Putting together all the pieces from Sections 8.4 and 8.5, we design a polynomial time algorithm to compute ♯3-SAT (with oracle access to language O_{per}, i.e., the language associated with permanent). On input 3-CNF formula Ψ, do the following.

- Let n and m be the number of variables and clauses in Ψ.
- Construct a matrix A over $\{-1, 0, 1\}$ such that per(A) is 4^{3m} times the number of satisfying assignments for Ψ.
- Let m be an integer for which we can compute per(A) modulo $2^m + 1$.
- Let A' be the matrix obtained by replacing every -1 in A with 2^m.
- Compute per(A') by querying the oracle on each bit.
- Let Z be the remainder of per(A') divided by $2^m + 1$.
- Divide Z by 4^{3m} and output it.

9 Diagonalization on the class NP

In this section we will show two classical results on the class **NP** proved by the "diagonalization" method.

9.1 Ladner's theorem: NP-intermediate language

Theorem 9.45 (Ladner 1975) *If* $\mathbf{P} \neq \mathbf{NP}$*, then there is* $L \in \mathbf{NP}$ *such that* $L \notin \mathbf{P}$ *and* L *is not* $\mathbf{NP}$*-complete.*

For a function $f : \mathbb{N} \to \mathbb{N}$, we say that it is *polynomial time computable* (in unary representation), if there is a polynomial time algorithm that on input 1^n, outputs $1^{f(n)}$.

For a function $f : \mathbb{N} \to \mathbb{N}$, define SAT_f as follows.

$$\mathsf{SAT}_f \stackrel{\mathsf{def}}{=} \{\varphi 0 \underbrace{1 \cdots 1}_{n^{f(n)}} : \varphi \in \mathsf{SAT} \text{ and } |\varphi| = n\}$$

We first prove the following lemma.

Lemma 9.46 *Suppose* $\mathbf{NP} \neq \mathbf{P}$*. If* $h : \mathbb{N} \to \mathbb{N}$ *is polynomial time computable (in unary representation), non-decreasing and unbounded, i.e.,* $\lim_{n\to\infty} h(n) = \infty$*, then* SAT_h *is not* $\mathbf{NP}$*-hard.*

Proof. Suppose to the contrary that SAT_h is **NP**-hard. Let F be a polynomial time reduction from SAT to SAT_h that runs in time cn^k. Let N be an integer such that for every $n \geqslant N$, the following holds.

- $h(n) \geqslant 2k$. (This is possible because h is non-decreasing and unbounded.)
- $cn^{1/2} < n$.

Claim 1 *For every* $\varphi \in \mathsf{SAT}$ *with length at least* N*, the output of* F *on* φ*, denoted by* $F(\varphi) = \psi 0 1^{|\psi|^{h(|\psi|)}}$*, satisfies the following: If* $|\psi| > N$*, then* $|\psi| < |\varphi|$*.*

Proof.(of claim) Since F runs in cn^k time, it follows that:

$$|\psi|^{h(|\psi|)} \quad < \quad |\psi| + 1 + |\psi|^{h(|\psi|)} \quad \leqslant \quad c|\varphi|^k$$

Thus,

$$|\psi| \quad < \quad c|\varphi|^{k/h(|\psi|)} \quad \leqslant \quad c|\varphi|^{1/2} \quad < \quad |\varphi|$$

The second and third inequalities come from the fact that $|\psi|, |\varphi| \geqslant N$. □

We now present a polynomial time algorithm for SAT, which contradicts the assumption that $\mathbf{NP} \neq \mathbf{P}$. On input φ, do the following.

- If $|\varphi| \leqslant N$, check by brute force if it is satisfiable. Otherwise, continue.
- Run F on φ, and let the output be $\psi 0 1^m$, for some m.
- Check if $m = |\psi|^{h(|\psi|)}$ by doing the following.
  1. Let $\ell = h(1^{|\psi|})$. (Recall that h is polynomial time computable.)
  2. Convert $|\psi|$ in its binary form and compute $|\psi|^\ell$ (in binary form).
  3. Then, compare it with m.

- If $m \neq |\psi|^{h(|\psi|)}$, then REJECT immediately.
- Suppose $m = |\psi|^{h(|\psi|)}$.
  If $|\psi| \leqslant N$, check if ψ is satisfiable by brute force.
  If $|\psi| > N$, recursively call the algorithm on ψ. (Note that here $|\psi| < |\varphi|$.)

Each step in the algorithm takes polynomial time and the number of recursive call in this algorithm is at most $|\varphi|$. So, overall the algorithm runs in polynomial time. □

Next, consider the following lemma.

Lemma 9.47 *Suppose* $\mathbf{NP} \neq \mathbf{P}$. *If* $h : \mathbb{N} \to \mathbb{N}$ *is polynomial time computable (in unary representation) and bounded, i.e., there is a constant* c *such that* $h(n) \leqslant c$ *for every* n*, then* $\mathsf{SAT}_h \notin \mathbf{P}$.

Proof. Suppose $\mathsf{SAT}_h \in \mathbf{P}$. We will show that $\mathsf{SAT} \in \mathbf{P}$, which contradicts the assumption that $\mathbf{NP} \neq \mathbf{P}$. Consider the following algorithm. On input φ, do the following.

- Check if $\varphi 01^i \in \mathsf{SAT}_h$, for some $0 \leqslant i \leqslant |\varphi|^c$.
- ACCEPT iff there is i where $\varphi 01^i \in \mathsf{SAT}_h$.

□

Combined with Lemmas 9.46 and 9.47, the following lemma implies Ladner's theorem, i.e., SAT_h is the desired intermediate **NP** language.

Lemma 9.48 *Suppose* $\mathbf{NP} \neq \mathbf{P}$. *There is a non-decreasing function* $h : \mathbb{N} \to \mathbb{N}$ *such that:*

- *h is polynomial time computable (in unary representation).*
- $\mathsf{SAT}_h \in \mathbf{NP}$.
- $\mathsf{SAT}_h \in \mathbf{P}$ *if and only if* h *is bounded.*

The function h for Lemma 9.48 is defined as follow. For every $n \geqslant 1$, the value $h(n)$ is determined by **Algorithm 1** below. Here $\mathcal{M}_i$ is the DTM whose encoding is the binary representation of i.

Algorithm 1

Input: 1^n, where $n \geqslant 1$.
Task: Compute $1^{h(n)}$.

```
1: for i = 1, ..., log log(n) − 1 do
2:     Let M_i be the i^th (1-tape) DTM.
3:     for all x ∈ {0,1}* where |x| ⩽ log n do
4:         Compute SAT_h(x) (i.e., recursively check if x ∈ SAT_h).
5:         Simulate M_i on x in i|x|^i steps (using the UTM in Theorem 1.3).
6:     if the results in lines 4 and 5 agree on all x ∈ {0,1}* where |x| ⩽ log n then
7:         return i (in unary).
8: return log log n (in unary).
```

9.2 Limit of diagonalization

A TM $\mathcal{M}$ with oracle access to a language K, denoted by $\mathcal{M}^K$, is a TM with a special tape called *oracle tape* and three special states $q_{\mathsf{query}}, q_{\mathsf{yes}}, q_{\mathsf{no}}$. Each time it is in q_{query}, it moves to q_{yes}, if $w \in K$ and to q_{no}, if $w \notin K$, where w is the string found in the oracle tape. In other words, when it is in q_{query}, the machine can "query" the membership of the language K. Regardless of the choice of K, such query counts only as one step. We denote by $L(\mathcal{M}^K)$ the language accepted by $\mathcal{M}^K$.

For a language K, we define the classes $\mathbf{P}$ and $\mathbf{NP}$ relativized to K as follows.

$$\begin{aligned} \mathbf{P}^K &\stackrel{\mathsf{def}}{=} \{L : \text{there is a polynomial time DTM } \mathcal{M}^K \text{ such that } L(\mathcal{M}^K) = L\} \\ \mathbf{NP}^K &\stackrel{\mathsf{def}}{=} \{L : \text{there is a polynomial time NTM } \mathcal{M}^K \text{ such that } L(\mathcal{M}^K) = L\} \end{aligned}$$

Theorem 9.49 (Baker, Gill, Solovay 1975) *There is language A and B such that $\mathbf{P}^A = \mathbf{NP}^A$ and $\mathbf{P}^B \neq \mathbf{NP}^B$.*

Proof. For a **PSPACE**-complete language A, we can show that $\mathbf{P}^A = \mathbf{NP}^A$. (We will show in Lesson 4 that **PSPACE**-complete languages exist.)

To show the existence of B, we need the following notation. For a language $C \subseteq \{0,1\}^*$, define $\mathsf{unary}(C) \stackrel{\mathsf{def}}{=} \{1^n : \text{there is } w \in C \text{ with length } n\}$. Obviously, for every $C \subseteq \{0,1\}^*$, $\mathsf{unary}(C) \in \mathbf{NP}^C$.

The language B will be defined as $B \stackrel{\mathsf{def}}{=} \bigcup_{i \in \mathbb{N}} B_i$ where each B_i is a finite set defined inductively as follows. Each B_i is associated with an integer k_i such that $B_i = B \cap \{0,1\}^{\leqslant k_i}$. Here $\{0,1\}^{\leqslant k_i} \stackrel{\mathsf{def}}{=} \{w \in \{0,1\}^* : |w| \leqslant k_i\}$.

The base case is $B_0 = \emptyset$ and $k_0 = 0$. For the induction step, B_{i+1} is defined as follows, where we assume an enumeration of all oracle DTM $\mathcal{M}_0, \mathcal{M}_1, \ldots$.

- Let $n = k_i + 1$.
- Simulate oracle TM $\mathcal{M}_{i+1}$ on 1^n within $2^n/10$ steps.

  During the simulation $\mathcal{M}_{i+1}$ may query the oracle. For the query strings with length $\leqslant k_i$, the oracle answers are according to B_i. For the query strings with length $> k_i$, the oracle answers are "no."
- Let k_{i+1} be as follows.

$$k_{i+1} \stackrel{\mathsf{def}}{=} \begin{cases} n, & \text{if all the query strings has length } \leqslant k_i \\ m, & m \text{ is the maximal length of the query string with length } \geqslant n \end{cases}$$

- If $\mathcal{M}_{i+1}$ accepts 1^n within $2^n/10$ steps, we set $B_{i+1} \stackrel{\mathsf{def}}{=} B_i$.
- If $\mathcal{M}_{i+1}$ does not accept 1^n within $2^n/10$ steps, we set $B_{i+1} \stackrel{\mathsf{def}}{=} B_i \cup \{w\}$, where $w \in \{0,1\}^n$ and w is not one of the query strings.

From the definition of B, we can show that $\mathsf{unary}(B) \notin \mathbf{P}^B$. □

Appendix

A The notion of computable functions

Polynomial time computable functions. Let $F : \Sigma^* \to \Sigma^*$ be a function from Σ^* to Σ^*. Let $\mathcal{M}$ be a 2-tape DTM.

- $\mathcal{M}$ *computes the function* F, if $\mathcal{M}$ accepts every word $w \in \Sigma^*$ and when it halts, the content of its second tape is $F(w)$.
- $\mathcal{M}$ *computes* F *in time* $O(g(n))$, if there is a constant $c > 0$ such that on every word w, $\mathcal{M}$ decides w in time $c \cdot g(|w|)$.
- $\mathcal{M}$ *computes* F *in polynomial time*, if $\mathcal{M}$ computes F in time $O(g(n))$ for some $g(n) = \mathsf{poly}(n)$.
- F *is computable in polynomial time*, if there is a DTM $\mathcal{M}$ that computes F in polynomial time.

Logarithmic space computable function. A function $F : \Sigma^* \to \Sigma^*$ is computable in logarithmic space, if there is a 3-tape DTM $\mathcal{M}$ and a constant c such that on every $w \in \Sigma^*$ the following holds.

- $\mathcal{M}$ accepts w.
- $\mathcal{M}$ never change the content of tape-1, i.e., tape-1 always contains the input word w.

  In other words, tape-1 is "read-only" tape.
- $\mathcal{M}$ only uses at most $c \log |w|$ cells in tape-2.
- Tape-3 is "write-only" tape, i.e., the head in tape-3 can only write and move right.
- When $\mathcal{M}$ halts, the content of tape-3 is $F(w)$.

B Time and space constructible functions

Definition B.50 Let $T : \mathbb{N} \to \mathbb{N}$ be a function.

- We say that T is *time constructible*, if for every n, $T(n) \geqslant n$ and there is a DTM that on input 1^n computes $1^{T(n)}$ in time $O(T(n))$.
- We say that T is *space constructible*, if there is a DTM that on input 1^n computes $1^{T(n)}$ in space $O(T(n))$.

Intuitively, when we say that $\mathcal{M}$ runs in time/space $O(T(n))$, where T is time/space constructible function, we can assume that on input word w, $\mathcal{M}$ first "computes" the amount of time/space needed to decide w, before going on to process w.

Theorems 4.27 and 4.28 can be easily generalized to space constructible functions as follows.

Theorem B.51 *Let* $f : \mathbb{N} \to \mathbb{N}$ *be space constructible function such that* $f(n) \geqslant \log n$, *for every* n.

- **(Savitch 1970)** $\textsc{Nspace}[f(n)] \subseteq \textsc{Dspace}[f(n)^2]$.
- **(Immerman 1988 and Szelepcsényi 1987)** $\textsc{Nspace}[f(n)] = \text{co}\textsc{Nspace}[f(n)]$.

C Hardness via log space reduction

In our definition of hardness for **NP**, **coNP** and **PSPACE**, we require that the reduction is polynomial time reduction. It is also common to define hardness by insisting the reduction is log-space reduction. That is, we can define K as **NP**-hard by insisting $L \leqslant_{\log} K$, for every $L \in \mathbf{NP}$, rather than $L \leqslant_p K$. Similarly, for **coNP** and **PSPACE**.

Most **NP**-, **coNP**- and **PSPACE**-complete problems are known to remain complete even under log-space reduction, including SAT, 3-SAT and TQBF.

- SAT and 3-SAT are **NP**-complete under log-space reduction.
- TQBF is **PSPACE**-complete under log-space reduction.