Lecture 3: Logic, Circuits and Computational Complexity

Computational Complexity

Angsheng Li

BeiHang University

Computational Theory
8, Oct. 2019

Outline

Basic notions

A **proposition** is a declarative sentence that is either true or false, bot not both.
A **propositional variable** (or statement variable) is a variable representing a proposition, written $p, q, r, s, t, x, y \cdots$.
The **truth value** of a proposition or propositional variable is **true**, written T, if the proposition is true, otherwise, then **false**, written F.
We can also use 0 and 1 to denote that a proposition is false and true, respectively.
The theory deals with propositions is called **propositional calculus, or propositional logic**.
Started by George Boole, The Laws of Thought, 1854, 英国 (1815-1864)

Chinese culture

- 阴
- 阳

科学 **vs** 文化

Basic Operations

Definition

1. Let p be a proposition. The **negation of** p, written $\neg p$ or $\bar{p}$, is the proposition **it is not the case that** p. Read **not** p
2. Let p, q be two propositions. The **conjunction of** p **and** q, denoted by $p \wedge q$, is the proposition p **and** q.
3. The **disjunction of** p **and** q, denoted by $p \vee q$, is the proposition p **or** q.

Truth Table, XOR

- **inclusive or**

$$p \vee q$$

p **or** *q*, **including both**

- **exclusive or**

$$\mathrm{XOR}(p, q)$$

$$p \oplus q$$

p **or** *q*, **but not both**

Implication

Definition

Let p and q be two propositions. An **implication** is the proposition $p \rightarrow p$, meaning that **if p, then q**.

Here, we call

1. p: **hypothesis, or antecedent, or premise**
2. q: **conclusion, or consequence**

$$p \rightarrow q$$

is equivalent to

$$\neg p \vee q$$

Bi-implication

Definition

$$p \leftrightarrow q$$

is equivalent to both

$$p \rightarrow q$$

and

$$q \rightarrow p$$

Logic and Bit Operations

A **bit** is a symbol with two possible values, 0 or 1, corresponding false and true, respectively.

A variable is called a **Boolean variable**, if it takes values 0 or 1.

Bit operations:

1. OR
2. AND
3. XOR
4. NE

Bit String

A **bit string** is a $0, 1$-string. The **length** of the string is the number of bits in the string.

Bit string operations:

- bitwise OR
- bitwise AND
- bitwise XOR

Logic Gates

- OR gate
- AND gate
- $\neg$ gate

Compound Proposition

Atomic proposition: 0, 1, $p, q, \cdots$

Proposition is generated from propositions using bit operations.

Tautology, 永真式 (Valid)

A proposition ϕ is called a **tautology**, or called **valid**, if any any assignment of the atomic Beoolean variables, ϕ takes the value true.

Logical Equivalent

Two compound propositions p and q are called **logically equivalent**, if $p \leftrightarrow q$ is a tautology, written

$$p \equiv q$$

De Morgan's Laws

印度 (1806-1871)

$$\neg(p \wedge q) \equiv \neg p \vee \neg q$$

$$\neg(p \vee q) \equiv \neg p \wedge \neg q$$

Identity Laws

$$p \wedge T \equiv p$$

$$p \vee F \equiv p$$

Domination Laws

$$p \vee T \equiv T$$

$$p \wedge F \equiv F$$

Idempotent Laws

$$p \vee p \equiv p$$

$$p \wedge p \equiv p$$

Double Negation Law

$$\neg(\neg p) \equiv p$$

Commutative Laws

$$p \vee q \equiv q \vee p$$

$$p \wedge q \equiv q \wedge p$$

Associative Laws

$$(p \vee q) \vee r \equiv p \vee (q \vee r)$$

$$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$$

Distributive Laws

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$

$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$$

Absorption Laws

$$p \vee (p \wedge q) \equiv p$$

$$p \wedge (p \vee q) \equiv p$$

Negation Laws

$$p \vee \neg p \equiv T$$

$$p \wedge \neg p \equiv F$$

Implication Laws

$$p \rightarrow q \equiv \neg p \vee q$$

$$p \rightarrow q \equiv \neg q \rightarrow \neg p$$

Satisfiability

A compound proposition ϕ is called **satisfiable**, if there is an assignment to the Boolean variables that make the proposition takes value true. Otherwise, it is called **unsatisfiable**.

It is a hard question to decide whether or not a proposition is satisfiable.

We use **SAT** to denote the problem to decide whether a given formula ϕ is satisfiable.

Conjunctive Normal Form

We call a compound proposition a proposition formula or a formula for simplicity. We use ϕ, *psi*, $\cdots$ to denote a formula.

A **conjunctive normal form (CNF)** is a formula of the following form:

$$\phi := C_1 \wedge C_2 \wedge \cdots \wedge C_m$$

where each C_i has the form:

$$l_1 \vee l_2 \vee l_k$$

each l_j is either x or $\neg x$ for some Boolean variable x. We call such an l a **literal**（文字）.
Each C_i is called a **cause**（从句）。

CNF theorem

Theorem

Every formula has an equivalent conjunctive normal form (CNF).

Disjunctive Normal Form (DNF)

$$\psi := D_1 \vee D_2 \vee \cdots \vee D_m$$

$$D_i := l_1 \wedge l_2 \wedge \cdots \wedge l_k$$

where each l_j is a literal.

DNF Theorem

Theorem

Every formula has an equivalent disjunctive normal form (DNF).

Predicates and Quantifiers

A **predicate** is a proposition with variable, x say, written $P(x)$.

The **quantifier** specifies the range of values of x making $P(x)$ true.

Universal quantification

$$\forall x P(x)$$

Existential quantification

$$\exists x P(x)$$

Random quantification

for randomly chosen x $P(x)$, $RxP(x)$

where R stands for **random**

Outline

1. Boolean functions
2. Expressions of Boolean functions
3. Logic gates
4. Circuit complexity
5. The classes NC and AC
6. Final examination - questions and answers

General view

- Circuits are basic devices for Computer, Electronics, and Algorithms

Backgrounds

- Boole, 1854
  Basic rules of logic
- Shannon, 1938 - Master thesis
  Circuits based on logic
- This forms the Boolean algebra.
  Why Boolean?

Elements

- The universe $\{0, 1\}$
- Complement, $\bar{x}$, $\bar{0} = 1$, $\bar{1} = 0$.
- The Boolean sum ($+ =$, OR)
- The Boolean product ($\cdot =$ AND)
- 0: False
- 1: True

Note: It is not the field $\mathbb{F}(2)$.

Boolean Expressions and Functions

Definition

A function

$$f: \{0,1\}^n \to \{0,1\}$$

is called an n-ary Boolean function.

Definition

The Boolean expressions in the variables $x_1, x_2, \cdots, x_n$ are:

1. $0, 1, x_1, \cdots, x_n$ are Boolean expressions.
2. If E_1 and E_2 are Boolean expressions, then so are

$$\bar{E}_1, \quad E_1 \cdot E_2, \quad E_1 + E_2.$$

Equivalence of Boolean Expressions and Functions

Definition

For Boolean expressions E_1 and E_2, or Boolean functions f_1 and f_2, we say that they are equivalent, if they take the same value for all assignments of the Boolean variables.

Identities of Boolean Algebra

- Law of the double complement

$$\bar{\bar{x}} = x$$

- Idempotent laws

$$x + x = x, \quad x \cdot x = x$$

- Identity laws

$$x + 0 = x, \quad x \cdot 1 = x$$

- Domination law

$$x + 1 = 1, \quad x \cdot 0 = 0$$

- Commutative laws

$$x + y = y + x, \quad x \cdot y = y \cdot x$$

Identities of Boolean Algebra - continued

- Associative laws
- Distributive laws
- De Morgan's laws

$$\overline{xy} = \bar{x} + \bar{y}, \quad \overline{x + y} = \bar{x} \cdot \bar{y}$$

- Absorption laws

$$x + xy = x, \quad x(x + y) = x$$

- Unit law $x + \bar{x} = 1$
- Zero property: $x \cdot \bar{x} = 0$

Duality

Given a Boolean expression E, the dual expression of E is obtained from E by

- interchange $+$ and $\cdot$
- interchange 0 and 1

Boolean Algebra

Definition

A Boolean algebra is a set B with two binary operations $\vee$ and $\wedge$, elements 0 and 1 and a unary operation $\overline{x}$ satisfying, for all $x, y, z \in B$,

1. Identity laws: $x \vee 0 = x, \quad x \wedge 1 = x$
2. Complement laws: $x \vee \bar{x} = 1, \quad x \wedge \bar{x} = 0$
3. Associative laws, commutative laws and distributive laws

Understanding of Boolean Algebra by Logic and Lattice

1. Logic
   - $B = \{0, 1\}$
   - OR - $+$
   - AND – $\cdot$
   - Negation - $^-$
   - F - 0
   - T– 1
2. Lattice
   - $x \vee y$: The least upper bound
   - $x \wedge y$: the greatest lower bound

Sum of Productions

Definition

A literal is a Boolean variable or its complement. A minterm is the product of some literals.

Theorem

Every Boolean function can be expressed by a sum of productions.

This is similar to the case of Boolean expressions: Every Boolean expression can be expressed as a disjunctive normal form. (DNF)

Product of Sums

Theorem

Every Boolean function can be expressed as the product of sums of literals.

This is similar to: Every expression can be expressed as a conjunctive normal form (CNF).

Functional Completeness

- $\{+, ^-\}$
- NAND, written $|$

$$x|y = 1 \iff \text{it is not the case that } x = y = 1.$$

- NOR, written $\downarrow$

$$x \downarrow y = 1 \iff x = y.$$

- $\{\cdot, ^-\}$.

Basic Gates

1. Inverter
2. OR gate
3. AND gate

The Complexity

Let C be a Boolean circuit. The complexity of circuit C is the number of gates contained in C.
It is a hard problem to minimise the number of gates.

Boolean Circuits as Model of Computation

A Boolean circuit is a diagram showing how to derive an output from a binary input string by applying a sequence of basic Boolean operations OR ($\vee$), AND ($\wedge$) and NOT ($\neg$) on the input bits.

Definition

For every $n \in \mathbb{N}$, and n-input, single-output Boolean circuit is a directed acyclic graph C (DAG) with n sources (input vertices) and one sink (output vertex). All nonsource vertex are called gates and are labelled with one of $\vee$, $\wedge$ and $\neg$. The vertices labelled $\vee$ and $\wedge$ have fin-in 2, and the vertices labelled with $\neg$ have fin-in 1. The size of C, denoted by $|C|$, is the number of vertices in C.

Computation of a Circuit

If C is a Boolean circuit, and $x \in \{0,1\}^n$, then the output of C on input x, denoted by $C(x)$, is the value of the sink or output vertex of C.
For every vertex v of C, the value of v is defined as follows:

1. If v is the ith input vertex, then the value of v is
$$\mathrm{val}(v) = x_i.$$
2. Otherwise, $\mathrm{val}(v)$ is defined recursively by applying v's logical operations on the values of the vertices pointing to v.
3. The output $C(x)$ is the value of the output vertex of C.

Circuit Families

Definition

Let $T : \mathbb{N} \to \mathbb{N}$ be a function. A $T(n)$-size circuit family, is a sequence $\{C_n\}_{n\in\mathbb{N}}$ of Boolean circuits, where C_n has n inputs and single output, and have size $|C_n| \leq T(n)$, for each n.
We say that a language L is in $\mathrm{SIZE}(T(n))$, if there is a $T(n)$-size circuit family $\{C_n\}$ such that for every n and every $x \in \{0,1\}^n$,

$$x \in L \iff C_n(x) = 1.$$

Key: Nonuniform computation

P/poly

Definition

P/poly is the class of languages that are decided by polynomial-sized circuit families. That is,

$$P/\mathrm{poly} = \cup_{c \geq 1} \mathrm{SIZE}(n^c).$$

- For some fixed c, for every n, $L \cap \{0,1\}^n$ is decided by a circuit of size n^c.
- Nonuniform model of computation

$P \subset P/\mathrm{poly}$

Theorem

$P \subset P/\mathrm{poly}$

- Every polynomial time Turing machine can be transformed to a circuit family of polynomial size.
- P/poly contains even undecidable languages.

P-uniform Circuit Family

Definition

A circuit family $\{C_n\}$ is P-uniform if there is a polynomial time algorithm that on input 1^n outputs C_n.

Theorem

A language L is decidable by a P-uniform circuit family if and only if $L \in P$.

Logspace Uniform Circuit Family

Definition

A circuit family $\{C_n\}$ is logspace uniform if there is a log space algorithm that on input 1^n outputs C_n.

Theorem

A language is computable by a log space uniform circuit if and only if it is in P.

Parallel Computation

We say that a computational problem has an efficient parallel algorithm if it can be solved for inputs of size n using a parallel computer with $n^{O(1)}$ processors in time $\log^{O(1)} n$.

- Parallel computation is closely related to circuits
- The depth of a circuit is the length of the longest directed path from an input vertex to the output vertex. The depth is the parallel time of the circuit.

The Class NC

Definition

For every d, a language L is in NC^d if L can be decided by a family of circuits $\{C_n\}$, where C_n has $\mathrm{poly}(n)$ size and depth $O(\log^d n)$.

The class NC is:

$$\mathrm{NC} = \cup_{i \geq 1} \mathrm{NC}^i.$$

We define uniform NC to be the circuits that is generated by a log space algorithm.

The Class AC

Definition

The class AC^i is defined similarly to NC^i except that gates are allowed to have unbounded fan-in.

The class AC is:

$$\mathrm{AC} = \cup_{i \geq 0} \mathrm{AC}^i.$$

NC Theorem

Theorem

A language has efficient parallel algorithms if and only if it is in NC.

Parallel Computation Problem

IS

$$P = NC?$$

The parallel computation complexity is measured by the depth of the circuits - DAG.
Therefore, the depth is essential to circuits.
Question Are there any other measures of circuits (DAG) that can be used to classify computational complexity of algorithms?

Efficiency and Running Time

Every nontrivial computational task requires at least reading the entire input, we count the number of basic steps as a **function of the input length**.

Definition

(Computing a function and running time) Let $f : \{0,1\}^* \to \{0,1\}^*$ and let $T : \mathbb{N} \to \mathbb{N}$ be some functions, and let M be a Turing machine. We say that M **computes** f if for every $x \in \{0,1\}^*$, whenever M is initialized to the start configuration on input x, then it halts with $f(x)$ written on its output tape. We say that M **computes** f in $T(n)$**-time** if its computation on every input x requires at most $T(|x|)$ steps.

Notation Σ^*: the set of all strings with elements in Σ.

Time-constructible functions

A function $T : \mathbb{N} \to \mathbb{N}$ is **time constructible**, if $T(n) \geq n$ and there is a Turing machine that computes the function $x \mapsto \mathrm{bin}(T(n))$ in time $T(n)$, where $n = |x|$ and $\mathrm{bin}(T(n))$ is the binary expression of $T(n)$.

Functions and Languages

A **complexity class** is a set of functions that can be computed within given resource bounds.
We will pay special attention to **Boolean functions**, that is, the functions with output only one bit, 0 or 1.
These functions define exactly **decision problems** or **languages**.

Definition

A **Language** is a set $L \subset \{0,1\}^*$ (or Σ^* for an alphabet Σ). A Turing machine **decides** language L is it computes the characteristic function of L, a function $f_L : \{0,1\}^* \to \{0,1\}$ such that $f_L(x) = 1$ if and only if $x \in L$.

DTIME and the Class P - **Efficient Solvable**

Definition

(The class DTIME) Let $T : \mathbb{N} \to \mathbb{N}$ be a function. A language L is in **DTIME**$(T(n))$ if and only if there is a Turing machine that runs in time $c \cdot T(n)$ for some constant $c > 0$ and decides L.

- D means **deterministic Turing machine**

Definition

(The Class P)

$$\mathrm{P} = \cup_{c \geq 1} \mathrm{DTIME}(n^c).$$

P is used to capture the concept of **efficient computation** (differs from effective).

Understanding P

- The class is intuitively assumed to be the class of efficient problems In real world applications, $O(n)$, $O(n^2)$ or $O(n^3)$ are practical
- Usually, we first find one with $O(n^c)$ for some constant, (which maybe large, 20 say), and then later on, new algorithms with $O(n^5)$ or even $O(n^3)$ are found.
- Worst-case exact computation is too strict, and in current application even $O(n)$ is too large

The primality test, for instance, first $\tilde{O}(n^{12})$, then $\tilde{O}(n^6)$.
New notation: $\tilde{O}(n^c)$, meaning that

$$O(n^c \cdot (\log n)^d)$$

for some constant $d > 0$.

The Class NP - **Efficient Verifiable Solutions**

Definition

(The Class NP) A language $L \subseteq \{0,1\}^*$ is in **NP**, if there is a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial time Turing machine M, referred to as **the verifier for L** such that for every $x \in \{0,1\}^*$,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)}[M(x,u) = 1]. \tag{1}$$

If $x \in L$ and $u \in \{0,1\}^{p(|x|)}$ satisfying $M(x,u) = 1$, then we call u a **certificate for x in L**.

Example

Given a CNF (conjunctive norm form) formula

$$\phi : (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (x_2 \vee x_3 \vee \neg x_4)$$

Solving: To find an assignment for x_1, x_2, x_3, x_4 such that ϕ is satisfied.

Verifying: Given $x_1 = a_1, x_2 = a_2, x_3 = a_3, x_4 = a_4$, test whether or not a_1, a_2, a_3, a_4 satisfy ϕ.

Nondeterministic Turing Machine

NP: nondeterministic polynomial time

A **nondeterministic Turing machine** M has two **transition functions** δ_0 and δ_1 such that

$$\delta_0(q, s) = (q_0, s_0, X_0)$$

$$\delta_1(q, s) = (q_1, s_1, X_1)$$

When the the state is q reading s, there are two choices either $\delta_0(q, s)$ or $\delta_1(q, s)$ to proceed to the next step of the computation.

For every input x, $M(x) = 1$ if and only if there exists a sequence of choices $\sigma_1, \sigma_2, \cdots, \sigma_l$ make M halts with an accepting state q_{accept}. Otherwise, then $M(x) = 0$.

We say that M runs in $T(n)$, if for any choices of the transition functions, M halts with $T(|x|)$ steps.

NTIME

Definition

For every function $T : \mathbb{N} \to \mathbb{N}$ and $L \subseteq \{0,1\}^*$, we say that $L \in \mathrm{NTIME}(T(n))$, if there is a constant $c > 0$ and a $c \cdot T(n)$-time nondeterministic Turing machine M such that for every $x \in \{0,1\}^*$,

$$x \in L \iff M(x) = 1.$$

NP Characteristic Theorem

Theorem

$$\mathrm{NP} = \cup_{c \in \mathbb{N}} \mathrm{NTIME}(n^c). \tag{2}$$

Proof.

Idea: **nondeterministic choices = certificate**

□

Polynomial Time Reducibility

Definition

Let A and B be two languages. We say that A is **polynomial time reducible to** B, if there is a polynomial time computation function f such that for every input x,

$$x \in A \iff f(x) \in B,$$

written $A \leq_{\mathrm{p}} B$.

Definition

Let L be a language. We say that L **is** NP **hard**, if every NP language X is polynomial time reducible to L. We say that L is NP **complete**, if L is both in NP and NP hard.

Basic properties of polynomial time reducibility

Theorem

1. *(Transitivity) If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.*
2. *If L is NP hard and $L \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.*

Noe that: Polynomials are close under compositions.

Cook Theorem

Theorem

SAT *is* NP *complete.*

Note: 3SAT is NP complete.

Logspace Computation

Definition

A Turing machine M with:

1. An input tape
2. A work tape
3. An output tape

The computation on every input x uses $O(\log n)$ bits in the work tape, and generates the output on the output tape. During the computation, the machine may read the bits of the input. However, on the input tape, it is not allowed to write or change any bits of the input.

- Remark: **Space complexity can be $O(\log n)$** due to the use of input tape.
- **Time complexity is at least n, reading the entire input**.
- Logspace computations are used as **reductions**, intuitively, this is the class of highly parallel computations

$\mathrm{Logspace} \subseteq \mathrm{P}$

We use $\mathrm{Logspace}$ to denote the set of languages that can be decided by a logspace Turing machine. Of course, logspace Turing machine may compute a function.

Theorem

$$\mathrm{Logspace} \subseteq \mathrm{P}.$$

Space-bounded computation

Definition

Let $s : \mathbb{N} \to \mathbb{N}$ and $L \subseteq \{0,1\}^*$. We say that $L \in \mathrm{SPACE}(s(n))$ if there is a constant $c > 0$ and a Turing machine M that decides L in space $c \cdot s(n)$, that is, for every input x, M on input x using at most $c \cdot s(n)$ for $n = |x|$ bits in the work tape, excluding the input tape.

Similarly, we say that $L \in \mathrm{NSPACE}(s(n))$ if there is a constant $c > 0$ and a nondeterministic Turing machine M that decides L in space $c \cdot s(n)$, that is, for every input x, M on input x using at most $c \cdot s(n)$ for $n = |x|$ bits in the work tape, excluding the input tape, regardless of its nondeterministic choices.

Configuration Graph

A **configuration** of a Turing machine M is a tuple consists of the **state**, the **position of the head**, the **symbol** that is scanned by the head, and the **sequence of nonblank cells** of the work tape.
A **configuration graph** of a Turing machine on input x is the graph of the configurations of the computations on input x following the directions instructed by the instructions of the Turing machine.

Space Complexity Classes

$$
\begin{aligned}
\mathrm{PSPACE} &= \cup_{c>0}\mathrm{SPACE}(n^c) \\
\mathrm{NPSPACE} &= \cup_{c>0}\mathrm{NSPACE}(n^c) \\
\mathrm{L} &= \mathrm{SPACE}(\log n) \\
\mathrm{NL} &= \mathrm{NSPACE}(\log n)
\end{aligned}
$$

Logspace Reduction

Definition

Let $f : \{0,1\}^* \to \{0,1\}^*$ be a function. We say that f **is logspace computable**, if there is a constant $c > 0$ such that for every x, the length of $f(x)$ is at most n^c, where $n = |x|$, and there is a logspace Turing machine that computes every bit of $f(x)$.

Let $A, B \subseteq \{0,1\}^*$. We say that A **is logspace reducible to** B, if there is a logspace computable function f such that for every x,

$$x \in A \iff f(x) \in B.$$

We use $A \leq_L B$ to denote that A is logspace reducible to B.

Logspace reducibility is transitive

Lemma

If $A \leq_L B$ and $B \leq_L C$, then $A \leq_L C$.

Proof.

Let $A \leq_L B$ via f and $B \leq_L C$ via g. Let $h(x) = g(f(x))$. We will show that for every i, the i-th bit of $h(x)$ is logspace computed. The difficulty is that there is no space to restore $f(x) = y$. To compute the i-th bit of $h(x)$, we may need to compute the j-th bit of y for many j's. However, whenever we need to compute the j-th bit of y, we restart the computation of $f(x)$, finding the j-th bit of $y = f(x)$. This shows that $g(f(x))$ is logspace computable.

Idea: Image that there is a **virtual input tape** restoring $f(x)$, which tells us the the bits of y are needed. When we want the j-th bit of $f(x)$, we run the logspace computation for f.

NL-completeness

Let $G = (V, E)$ be a directed graph, and s and t be two vertices. The **reachability** is to decide whether or not there is a directed path starting from s to reach t. Let PATH be the set of triples $(G; s, t)$ such that there is a path from s to t in G.
Clearly, PATH is in NL. That is, there is a nondeterministic Turing machine that decides PATH in $\log n$ space, denoted $\mathrm{PATH} \in \mathrm{NL}$.
In fact, we have

Theorem

PATH is NL-complete, under logspace reduction, of course.

Let M be a nondeterministic Turing machine in log space. For every x, Let G_x be the configuration graph of M on x.

$$M(x) = 1 \iff \exists s, t - \mathrm{path} \text{ in } G_x$$

s is the initial configuration, t is the acceptance configuration.

P-completeness

Theorem

CIRCUIT VALUE *is P-complete.*

CIRCUIT consists of AND, OR, NOT gates, and Boolean variables $x_1, x_2, \cdots, x_n$.

Proof.

Clearly, CIRCUIT VALUE is in P. We will show that a Turing machine in n^c time can be represented by a **computation table**. Every execution of the computation at (i, j) depends only on the state, the positions $(i-1, j-1), (i-1, j), (i-1, j+1)$, which can be represented by a Boolean formula $F_{i,j}$, where $F_{i,j}$ can be decomposed into several AND, OR and NOT gates. The entire computation table can be represented as a CIRCUIT VALUE instance. This representation can be realised by a logspace Turing machine.

Therefore, every language L in P is logspace reducible to CIRCUIT VALUE.

Cook's Theorem

Theorem

SAT *is NP-complete.*

Proof.

Every poly nondeterministic Turing machine can be logspace reducible to an SAT instance. We introduce Boolean variables to denote the choices of nondeterministic executions of the machine. Once the nondeterministic choices are made, the reduction is constructed by the same way as the P-completeness of CIRCUIT VALUE. □

PSPACE Completeness

A **quantified satisfiability, QSAT** instance is a formula of the following form:

$$\Phi : \exists x_1 \forall y_1 \cdots \exists x_n \forall y_n \phi(x_1, y_1, \cdots, x_n, y_n) \tag{3}$$

The QSAT problem is to decide, for a given instance Φ as above, whether or not Φ is satisfiable.

- QSAT is in PSPACE.
- QSAT is PSPACE complete.
- This is a game as GO, so SO is PSPACE complete.

The Complexity Classes

$$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \tag{4}$$

Theorem
(Hartmanis, 1965)

$$\mathrm{L} \subsetneq \mathrm{PSPACE}.$$

Research Directions of Computational Complexity

1. New concepts related to computation
2. New algorithmic ideas
3. Concrete algorithmic problems such that graph isomorphism problem, factoring, SAT etc.

Assignments

1. Show that every Boolean expression is equivalent to one in conjunctive normal form, and to one in disjunctive norm form.
2. Show that a Boolean expression is satisfiable if and only if its negation is a tautology (or in another word, valid).

Thank You!