NP-completeness

In the last lecture we talked about various computation models, in particular Turing Machine model. We defined the complexity classes \( P \) and \( NP \). In this lecture, we focus on studying about the hardest problems in \( NP \), namely the \( NP \)-complete problems. To describe \( NP \)-completeness, we introduce the concept of polynomial reductions, and \( NP \)-hardness. We also show a few examples of \( NP \)-complete problems.

Our goal is to characterize the hardest problems in the complexity class \( NP \). Let us consider two different problems - Boolean satisfiability (\( SAT \)) and the reachability problems in \( NP \). One can verify that both the problems belong to \( NP \). But, we know that the reachability problem is much simpler than the satisfiability problem. Therefore they must belong to different levels or floors in this so-called building of \( NP \) problems. To classify certain problems as harder than others, we mathematically formalize the concept of “hardness”, using the \( NP \)-completeness notion.

2.1 Reductions

We compare relative hardness between two problems using the concept of reductions.

**Definition 2.1.** For two languages \( L_1, L_2 \subseteq \{0, 1\}^* \), we say that \( L_1 \) polytime Karp reduces to \( L_2 \) (denoted by \( L_1 \leq_P L_2 \)) if \( \exists \) a polynomial time computable function \( f : \{0, 1\}^* \rightarrow \{0, 1\}^* \), such that

\[ x \in L_1 \iff f(x) \in L_2 \]

Figure 2.1 shows the graphical interpretation of a reduction. As is evident from the figure, note that \( f \) may or may not be a surjective map. Also, don’t confuse polynomial reductions with Turing reductions which is a concept to be touched in later part of the course. If \( L_1 \) reduces to \( L_2 \), intuitively it means that \( L_1 \) is easier than \( L_2 \), converse may or may not be true. It also means that an algorithm for solving problem \( L_2 \) can be used to solve problem \( L_1 \). This is done by sending the problem instances of \( L_1 \) to instances of \( L_2 \), and then solving them using algorithm for \( L_2 \).
2.2 Definition of NP-completeness

First of all we define NP-hard problems.

Definition 2.2. A language $L \subseteq \{0,1\}^*$ is NP-hard if $\forall$ languages $L'$,

$$L' \in \text{NP}, \quad L' \leq_p L$$

that is, all languages in NP can be polynomial time reduced to a NP-hard language.

Definition 2.3. $L$ is NP-complete if $L \in \text{NP}$ and $L$ is NP-hard.

Thus, a NP-complete problem is one of the hardest problems in class NP, because all other problems in NP reduce to it.

Class NP is closed under polynomial reductions, that is,

**Theorem 2.4.** If $A \in \text{NP}$ and $B \leq_p A$, then $B \in \text{NP}$

**Proof.** $A \in \text{NP}$, therefore there exists NDTM $N_A$ such that $L(N_A) = A$, and it accepts it in polynomial time. Let $B \leq_p A$ via function $f$.

**Algorithm:**

- **Input:** String $w$

1. Calculate $f(w)$ in polynomial time.
2. Run $N_A$ on input $f(w)$.
3. If it accepts, accept, otherwise reject.

$$w \in B \iff f(w) \in A$$

Thus, $B$ has a non-deterministic polynomial time verifier algorithm.

---

Figure 2.1: Reduction $f : L_1 \rightarrow L_2$
Exercise 2.1. Note that all kinds of complexity classes are not closed under polynomial reductions. Give few concrete examples of such complexity classes.

Exercise 2.2. Prove that if $B$ is NP-hard and $B \leq_p A$, then $A$ is NP-hard. [Hint: Prove the transitivity of polynomial reduction]

Proof. First we prove the following lemma,

Lemma 2.5. If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Proof. Suppose $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ via polytime functions $f$ and $g$. Construct function $h$ such that $h(x) = g(f(x))$. Since $f$ and $g$ are polynomial time computable, so is their composition. (Polynomials are closed with respect to composition). Now,

$$x \in L_1 \iff f(x) \in L_2 \iff g(f(x)) \in L_3 \iff h(x) \in L_3.$$

Therefore, $L_1 \leq_p L_3$.

Consider any $L$ which is in NP. Then $L \leq_p B$ (since $B$ is NP-hard). And $B \leq_p A$, therefore by above lemma $L \leq_p A$. Hence $A$ is NP-hard.

Thus, if a polynomial time algorithm is found for any NP-complete problem, then all problems in NP would be polynomial time solvable that is, $P = NP$. The notion of NP-completeness can save researchers from wasting time in searching a polynomial time algorithm for a particular problem, if they can prove the problem to be NP-complete. Now we show few concrete examples of NP-complete problems.

2.3 SAT: Boolean Satisfiability problem

Once, we have one NP-complete problem, we may obtain others by polynomial time reduction from it. However, establishing the first NP-complete problem was difficult. Boolean satisfiabilty problem was historically the first problem shown to be NP-complete, by Stephen Cook and Leonid Levin independently in the early 1970's.

Definitions: A boolean variable is one which takes values either 0(false) or 1(true). The boolean operations AND($\land$), OR($\lor$) and NOT($\neg$) are defined as follows:

$$
\begin{align*}
0 \land 0 &= 0 & 0 \lor 0 &= 0 & \overline{0} &= 1 \\
0 \land 1 &= 0 & 0 \lor 1 &= 1 & \overline{1} &= 0 \\
1 \land 0 &= 0 & 1 \lor 0 &= 1 \\
1 \land 1 &= 1 & 1 \lor 1 &= 1
\end{align*}
$$

A Boolean formula is defined recursively as:

$$
\Phi = \begin{cases}
\Phi_1 \land \Phi_2 \\
\Phi_1 \lor \Phi_2 \\
\overline{\Phi}
\end{cases}
$$
where $\Phi_1, \Phi_2, \Phi'$ are boolean formula of size lesser than $\Phi$. For example, $\Phi : (\neg x \land y) \lor z$. A literal is a boolean variable or it's negation. ($x$ or $\overline{x}$). A truth assignment $\tau$ is an assignment of values 0 or 1 to every variable. $\tau : \{x_1, x_2, \ldots, x_n\} \rightarrow \{0, 1\}$. Evaluation of a boolean formula on a truth assignment is denoted as $\Phi(\tau)$. A boolean formula $\Phi$ is said to be satisfiable if there exists a truth assignment $\tau$, such that $\Phi(\tau) = 1$.

**Definition 2.6.** $\text{SAT} = \{ \langle \Phi \rangle \mid \Phi$ is a satisfiable boolean formula $\}$.

**Theorem 2.7 (Cook-Levin).** $\text{SAT}$ is NP-complete.

**Proof Sketch:** We need to show that $\text{SAT}$ is in NP, and all other problems in NP can be reduced to it.

**Lemma 2.8.** $\text{SAT} \in \text{NP}$

**Proof.**

**Certificate:** A truth assignment $\tau$ on variables of boolean formula $\Phi$.

**Verifier’s algorithm:**

Input: $\langle \Phi, \tau \rangle$

1. Calculate $\Phi(\tau)$ in polynomial time.
2. If $\Phi(\tau) = 1$, accept, else reject.

**Lemma 2.9.** $\forall A \in \text{NP}, A \leq_p \text{SAT}.$

FIGURE 2.2: A tableau is an $n^k \times n^k$ table of configurations
Proof.

Tableau construction:
Let $A \in \text{NP}$. Therefore there exists a polytime NDTM $N$ such that $A = L(N)$. Suppose running time of $N$ is less than or equal to $n^k$. Thus, $N$ work’s tape consists of at most $n^k$ cells. We construct a tableau for a computation path of $N$ on input $w$, as shown in figure 2.2. A tableau is a $n^k \times n^k$ table where each row corresponds to a configuration of $N$ for that particular computation path. A tableau is accepting if any of the rows corresponds to an accept configuration of $N$. Thus $w \in A$ if and only if there exists an accepting tableau.

Reduction:
We want that on an input $w$, the reduction outputs a boolean formula $\Phi$. First, we define the variables of $\Phi$. Let $C = \{Q \cup \Gamma\}$, where $Q$ is the set of states, $\Gamma$ is the alphabet set along with blank symbol $\beta$. For all $i,j$ between 1 to $n^k$ and for each $s \in C$, we define a variable $x_{i,j,s}$, which equals 1 if cell $[i,j]$ of the tableau contains symbol $s$. Given $x \in A$, construct a boolean formula $\Phi_x$ as:

$$\Phi_x = \Phi_{cell} \land \Phi_{start} \land \Phi_{move} \land \Phi_{accept}$$

$\Phi_{cell}$: ensures that cell $[i,j]$ contains exactly a single symbol $s$ from $C$. This is captured by the following definition:

$$\Phi_{cell} = \bigwedge_{1 \leq i,j \leq n^k} \left( \bigvee_{s \in C} x_{i,j,s} \right) \land \left( \bigwedge_{s,t \in C} \overline{x_{i,j,s}} \lor \overline{x_{i,j,t}} \right)$$

$\Phi_{start}$: ensures that first row of tableau is a start configuration, through the following expression:

$$\Phi_{start} = x_{1,1,q_0} \land x_{1,2,w_1} \land \cdots \land x_{1,n+1,w_n} \land x_{1,n+2,\beta} \land \cdots \land x_{1,n^k,\beta}$$

$\Phi_{accept}$: ensures that tableau is accepting. This means that $q_{accept}$ state occurs in some cell of the tableau. Thus,

$$\Phi_{accept} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{accept}}$$

$\Phi_{move}$: ensures configuration $i+1$ follows legally from configuration $i$, as dictated by the transition function of $N$. A $2 \times 3$ window of table cells is legal if the it’s content does not violate the transition function. $\Phi_{move} = 1$ if all possible $2 \times 3$ windows are legal. For example window shown in fig 2.2 is legal if $\delta(q_j, w_{k+1}) = (q_i, w_p, R)$. The column size of window is 3 because the turing machine head can either move one step left or one step right on some input alphabet.

$$\Phi_{move} = \bigwedge_{1 \leq i < n^k, 1 \leq j < n^k} \left( \text{the (i,j)-window is legal} \right)$$

Replace $(i,j)$-window is legal with following formula where $a_1 \cdots a_6$ are the 6 cells of window.

$$\bigvee \left( x_{i,j-1,a_1} \land x_{i,j,a_2} \land x_{i,j+1,a_3} \land x_{i+1,j-1,a_4} \land x_{i+1,j,a_5} \land x_{i+1,j+1,a_6} \right)$$

2-5
Correctness Proof:

(⇒) If \( w \in A \), then there exists a path from start configuration to accept configuration. This implies that \( \Phi_{\text{start}}, \Phi_{\text{accept}}, \Phi_{\text{move}} \) and \( \Phi_{\text{cell}} \) are satisfiable and hence \( \Phi \) is satisfiable.

(⇐) If \( \Phi \) is satisfiable, then there exists an accepting tableau, which in turn implies that there exists a computation path from start configuration to accept configuration. Thus,

\[
x \in A \Leftrightarrow \Phi_x \text{ is satisfiable}
\]

Thus \( A \leq_p \text{SAT} \) if we can show that \( \Phi_x \) can be constructed from \( N \) in polynomial time. This is true because the size of boolean formula is \( O(n^{2k}) \), a polynomial in \( n \).

Thus Lemma 2.8 and 2.9 prove theorem 2.7.

Exercise 2.3. Read complete proof from [3] and an alternative proof from [1].

2.4 3SAT problem

A literal is defined as a variable \( x \) or its negation \( \neg x \). A clause is an OR of literals. A CNF formula is an AND of clauses. A 3-CNF formula is an AND of clauses where each clause has exactly 3 literals.

Definition 2.10. 3SAT = \{ \langle \Phi \rangle | \Phi \text{ is a satisfiable 3CNF formula} \}.

For example, \((x_1 \lor x_3 \lor x_4) \land (\overline{x_2} \lor x_3 \lor x_5) \land (\overline{x_1} \lor \overline{x_3} \lor x_4)\)

Lemma 2.11. 3SAT ∈ NP

Proof. Same proof as that of lemma 2.8.

Exercise 2.4. Prove that SAT \( \leq_p \) 3SAT

Proof. Lemma 2.9 produces a formula that is almost in conjunctive normal form. \( \Phi_{\text{cell}}, \Phi_{\text{start}}, \Phi_{\text{accept}} \) all are in cnf. \( \Phi_{\text{move}} \) is a big AND of subformulas, each of which is an OR of ANDs that describe all possible windows. With distributive laws, we can convert an OR of ANDs with an equivalent AND of ORs, with only a constant blow up in the size of \( \Phi_{\text{move}} \).

Now we can convert a formula in cnf to one with three literals per clause. If the clause contains \( r \) literals,

\[
(l_1 \lor l_2 \lor \cdots \lor l_r)
\]

we can replace it with \( l - 2 \) clauses

\[
(l_1 \lor l_2 \lor z_1) \land (\overline{z_1} \lor l_3 \lor z_2) \land (\overline{z_2} \lor l_4 \lor z_3) \land \cdots \land (\overline{z_{l-3}} \lor l_{r-1} \lor l_r)
\]

It is easy to verify that the new formula is satisfiable if and only if the original formula is satisfiable.
Corollary 2.12. \textbf{3-SAT} is \textsf{NP}-complete.

\textit{Proof.} By exercise 2.2, 2.4 and 2.11.

Fact 2.13. \textbf{1-SAT} is solvable in linear time (just check whether a variable and its complement are present). \textbf{2-SAT} is also solvable in linear time due to Tarjan et al \cite{2}. \textbf{3-SAT} and higher \textsf{SAT} problems are \textsf{NP}-complete.

2.5 IndSet problem

Let $G = (V, E)$ be a graph. A subset $S \subseteq V$ is an independent set if for every pair of vertices $u, v \in S, (u, v) \notin E$

Definition 2.14. IndSet = \{\langle G, k \rangle | G \text{ has an independent set of size } \geq k \} 

Lemma 2.15. IndSet $\in$ \textsf{NP}

\textit{Proof.}

\textbf{Certificate:} A subset $S \subseteq G$ of vertices.

\textbf{Verifier’s algorithm:}

\textbf{Input:} $\langle G, S \rangle$

1. Check if $S \subseteq G$. If not, REJECT.
2. Check if size of $S$ is equal to $k$. If not REJECT.
3. Check whether for any two vertices $u, v$ in $S, (u, v) \in E$. If yes, REJECT else ACCEPT
Lemma 2.16. $3\text{SAT} \leq_p \text{IndSet}$

Proof. Consider a 3CNF formula with $k$ clauses. For example, $\Phi = (x_1 \lor x_3 \lor \overline{x_3}) \land (\overline{x_1} \lor \overline{x_2} \lor x_3)$ Construct a graph $G = (V, E)$ where $V = \{v_{ij} | 1 \leq i \leq k \text{ and } 1 \leq j \leq 3\}$ i.e 3 vertices(literals) for each of the $k$ clauses. The graph will contain a triangle for each clause as shown in the figure 2.3. Add edges between literals of same clause. Also, add edges between variables and their compliments, that is, $(v_i, v_j) \in E$ if and only if $l_i = \overline{l}_j$. The graph $G$ can be constructed in $O(n^2)$ time where, $n$ is the number of variables in $\Phi$.

Lemma 2.17. $\Phi$ is satisfiable if and only if graph $G$ has an independent size of size $k$, where $k$ is the number of clauses in $\Phi$.

Proof.

- If $\Phi$ is satisfiable, then each clause must be having a satisfying literal. The corresponding set of vertices for each satisfying literal forms an independent set of size $k$.

- If $G$ has an independent set of size $k$, then note that no two vertices can be from the same triangle. Also, no two vertices (literals) $v_i, v_j$ such that $v_i = \overline{v}_j$, can be part of this independent set. Thus, we have a satisfying literal from each clause, and hence $\Phi$ is satisfiable.

Thus the construction $G$ along with lemma 2.17 completes the proof.

Corollary 2.18. $\text{IndSet}$ is $\text{NP}$-complete.

Proof. Since we have shown that $3\text{-SAT}$ problem can be reduced to IndSet in Lemma 2.16 Thus by Exercise 2.2 IndSet is $\text{NP}$-complete.

References

