Polynomial Verification

The problems Composite, SAT and HAM have something in common.

In each case, there is a search space of possible solutions.

the factors of $x$; a truth assignment to the variables of $\phi$; a list of the vertices of $G$.

The number of possible solutions is exponential in the length of the input.

Given a potential solution, it is easy to check whether or not it is a solution.

Verifiers

A verifier $V$ for a language $L$ is an algorithm such that

$$L = \{ x \mid (x, c) \text{ is accepted by } V \text{ for some } c \}$$

If $V$ runs in time polynomial in the length of $x$, then we say that $L$ is polynomially verifiable.

Many natural examples arise, whenever we have to construct a solution to some design constraints or specifications.

Nondeterministic Complexity Classes

We have already defined TIME($f(n)$) and SPACE($f(n)$).

$\text{NTIME}(f(n))$ is defined as the class of those languages $L$ which are accepted by a nondeterministic Turing machine $M$, such that for every $x \in L$, there is an accepting computation of $M$ on $x$ of length at most $f(n)$.

$$\text{NP} = \bigcup_{k=1}^{\infty} \text{NTIME}(n^k)$$

For a language in $\text{NTIME}(f(n))$, the height of the tree is bounded by $f(n)$ when the input is of length $n$.

Nondeterminism

For a language in $\text{NTIME}(f(n))$, the height of the tree is bounded by $f(n)$ when the input is of length $n$.
\textbf{NP}

A language $L$ is polynomially verifiable if, and only if, it is in \textbf{NP}.

To prove this, suppose $L$ is a language, which has a verier $V$, which runs in time $p(n)$.

The following describes a \textit{nondeterministic algorithm} that accepts $L$

1. input $x$ of length $n$
2. nondeterministically guess $c$ of length $\leq p(n)$
3. run $V$ on $(x, c)$

\textbf{Generate and Test}

We can think of nondeterministic algorithms in the generate-and-test paradigm:

\begin{tikzpicture}
  \node [startstop] (start) {\textit{generate}};
  \node [process, right of=start] (p1) {\textit{verify}};
  \node [decision, right of=p1, below of=p1] (p2) {\textit{yes}};
  \node [decision, right of=p1, above of=p1] (p3) {\textit{no}};
  \draw (start) -- (p1);
  \draw (p1) -- (p2);
  \draw (p1) -- (p3);
\end{tikzpicture}

Where the \textit{generate} component is nondeterministic and the \textit{verify} component is deterministic.

\textbf{NP}

In the other direction, suppose $M$ is a nondeterministic machine that accepts a language $L$ in time $n^k$.

We define the \textit{deterministic algorithm} $V$ which on input $(x, c)$ simulates $M$ on input $x$.

At the $i$\textsuperscript{th} nondeterministic choice point, $V$ looks at the $i$\textsuperscript{th} character in $c$ to decide which branch to follow.

If $M$ accepts then $V$ accepts, otherwise it rejects.

$V$ is a polynomial verifier for $L$.

\textbf{Reductions}

Given two languages $L_1 \subseteq \Sigma_1^*$, and $L_2 \subseteq \Sigma_2^*$,

A \textit{reduction} of $L_1$ to $L_2$ is a \textit{computable} function

$$f : \Sigma_1^* \rightarrow \Sigma_2^*$$

such that for every string $x \in \Sigma_1^*$,

$$f(x) \in L_2 \text{ if, and only if, } x \in L_1$$
**Resource Bounded Reductions**

If \( f \) is computable by a polynomial time algorithm, we say that \( L_1 \) is *polynomial time reducible* to \( L_2 \).

\[
L_1 \leq_p L_2
\]

If \( f \) is also computable in \( \text{SPACE}(\log n) \), we write

\[
L_1 \leq_L L_2
\]

**Completeness**

The usefulness of reductions is that they allow us to establish the *relative* complexity of problems, even when we cannot prove absolute lower bounds.

Cook (1972) first showed that there are problems in \( \text{NP} \) that are maximally difficult.

A language \( L \) is said to be \textit{NP-hard} if for every language \( A \in \text{NP} \), \( A \leq_p L \).

A language \( L \) is \textit{NP-complete} if it is in \( \text{NP} \) and it is \textit{NP-hard}. 

**Reductions 2**

If \( L_1 \leq_p L_2 \) we understand that \( L_1 \) is no more difficult to solve than \( L_2 \), at least as far as polynomial time computation is concerned.

That is to say,

\[
\text{If } L_1 \leq_p L_2 \text{ and } L_2 \in \text{P}, \text{ then } L_1 \in \text{P}
\]

We can get an algorithm to decide \( L_1 \) by first computing \( f \), and then using the polynomial time algorithm for \( L_2 \).