$\mathcal{P}$ and $\mathcal{NP}$

Definitions

Throughout, if we say “machine” we mean deterministic machine, unless we specifically say “non-deterministic.”

A languague L over an alphabet Σ is *decided* by a machine M if, given any input string $w \in \Sigma^*$, M halts and accepts w if $w \in L$, and rejects w if $w \notin L$.

A languague L over an alphabet Σ is *accepted* by a non-deterministic machine M if both of these conditions hold:

- For any $w \in L$, there is a computation of M with input w which halts in an accepting state.
- If $w \notin L$, there is no computation of M with input w which halts in an accepting state.

Without loss of generality, we assume that at any point in a computation of a non-deterministic machine there are at most two choices. We can make this assumption since multiway branching can be emulated by sequential two-way branching.

Guide Strings. A *guide string* is a string which tells a non-deterministic machine which choice to make at each step. Since we are assuming that branch points are two-way, a guide string is a binary string.

Polynomially Bounded Functions. A function $f(n)$ is *polynomially bounded in n* if there is a constant k such that $f(n)$ is eventually less than n^k, meaning that for some number N, $f(n) < n^k$ for all $n > K$. Let $\mathcal{P}(n)$ be the class of functions which are polynomially bounded in n.

A language L over an alphabet Σ is in the class $\mathcal{P}$–TIME, usually written as simply $\mathcal{P}$, if there is some machine M which decides L in polynomial time, meaning that for any $w \in \Sigma^*$, M decides whether an input string $w \in \Sigma^*$ is a member of L in time which is polynomially bounded in n, where $n = |w|$, the length of the string w. That is, there is some constant k such that, for any $w \in \Sigma^*$, M accepts w within $O\left(|w|^k\right)$ time if and only if $w \in L$.

L is the class $\mathcal{NP}$–TIME if there is a non-deterministic machine M which accepts L is polynomial time. That is, there is a constant k such that, for every $w \in L$, there is a computation of M with input w which reaches an accepting state in no more more than n^k steps, and furthermore, that for any input $w \notin L$, there is no computation of M which reaches an accepting state. Since every deteriministic machine is also non-deterministic, $\mathcal{P}$–TIME $\subseteq$ $\mathcal{NP}$–TIME.

Theorem 1 *Every $\mathcal{NP}$ language is decidable.*

Proof: Let L be an $\mathcal{NP}$ language, and k a constant and M a non-deterministic machine that accepts L in n^k time. Let $w \in \Sigma^*$, $n = |w|$, and $\mathcal{B}$ the set of all binary strings of length $|w|^k$. Note that $\mathcal{B}$ has order 2^{n^k}. Let M_2 be a deterministic machine which emulates M once for each guide string, until it either reaches an accepting state of M, in which case it accepts w, or has used all the guide strings without reaching an accepting state of M, in which case it rejects w. Thus, M_2 accepts w if and only if M accepts w, since, if M accepts w, there must be some sequence of choices M can make which leads to accepting state. $\square$

In summary, Theorem 1 states that any language accepted by a non-deterministic machine in polynomial time is decided by some deterministic machine in exponential time.

The way we constructed M_2 might seem inefficient. Can we do better? Can every $\mathcal{NP}$ language be decided in polynomial time? That is the same as saying $\mathcal{NP}$–TIME $= \mathcal{P}$–TIME. No one knows any proof that this statement is either true or false – it is commonly stated to be the most important open problem in the theory of computation, and certainly is one of the most important open problems in all mathematics.

Verifier Definition of $\mathcal{NP}$

There is an equivalent definition of $\mathcal{NP}$ which is frequently easier to work with, the *verifier* definition, given below.

Let $L \subseteq \Sigma^*$ be a language. Then L is $\mathcal{NP}$ if and only if there is an integer k and a machine V, called a *verifier* of L, Basically, V verifies that $w \in L$ using a *certificate* c. An input of V is the concatenation w, c where $w \in \Sigma^*$, and its output is Boolean.

For any $w \in \Sigma^*$, and $n = |w|$.

1. If $w \in L$, there is some certificate c such that V returns 1 (true) in $\mathcal{P}(n)$ time.
2. If $w \notin L$, then V always returns 0 (false) regardless of the choice of certificate.

It is important to note that you have to choose a *correct* certificate. That is, even if $w \in L$, V will return 0 given input w, c if c is not chosen correctly.

Boolean Satisfiability

We now consider one of the most important $\mathcal{NP}$ languages, Boolean satisfiability, abbreviated SAT, which in fact is $\mathcal{NP}$–complete, a property we define later.

let BOOL be the language of all Boolean expressions, over an appropriate alphabet. An *assignment* of an expression $E \in \text{BOOL}$ is a mapping of the set of all variables which appear in E to the Boolean alphabet $\{0, 1\}$, where 0 means false and 1 means true. The assignment is *satisfying* if replacing each variable by its assigned truth value causes E to become true. SAT is the set of Boolean expressions which are satisfiable, *i.e.* have satisfying assignments.

It is easy to describe a verifier for SAT. A *certificate* for any $E \in \text{SAT}$ is a satisfying assignment of E. For example, if E_1 is the expression $(!x + y) * (!y + z)$, the assignment $x = 0, y = 1, z = 1$ satisfies E_1, while the assignment $x = 1, y = 0, z = 1$ does not. On the other hand, the expression $E_2 = x * (y + z) * (!x+!y) * (!z)$ is a contradiction, *i.e.* has no satisfying assignment. Hence $E_1 \in \text{SAT}$, while $E_2 \notin \text{SAT}$. Our verifier is a simple program that evaluates E after using c to assign a truth value to each variable.