P and NP

CS 381

Reading: “Introduction to the theory of computation” by Michael Sipser, 2nd edition, Chapter 7.

Recap from last time:

Definition (Running time). Let M be a deterministic Turing machine that halts on all inputs. The running time (time complexity) of M is the function $t: N \to N$, where $t(n)$ is the maximum number of steps that M takes on an input of length n. We say that M runs in time $t(n)$ and M is a $t(n)$ time Turing machine.

Recall: a Turing machine M decides a language $L \in \{0,1\}^*$ if for any word $w \in \{0,1\}^*$, the machine M accepts if $w \in L$ and rejects if $w \notin L$.

Recap from last time:

Definition (The class DTIME). Let $f: N \rightarrow N$ be a function. A language L is in DTIME(t(n)) if there exists a Turing machine that runs in O(t(n)) steps and decides L.

Recap from last time:

Definition (The class DTIME). Let $f: N \rightarrow N$ be a function. A language L is in DTIME(t(n)) if there exists a Turing machine that runs in O(t(n)) steps and decides L.

Definition (The class P). P is the class of languages decidable in polynomial time on deterministic Turing machines, i.e. $P = \bigcup_{k \geq 0} DTIME(n^k)$.

Informally: P is the class of efficiently solvable problems.

Verifiers

Definition (Verifier). A verifier for a language L is an algorithm V that takes inputs of the form $< w, u >$ and has the property that $w \in L \Leftrightarrow$ there exists u such that V accepts $< w, u >$.

A string u with this property is called a certificate (witness) for w.

Thus we have:

$$L = \{w \mid V \text{ accepts } < w, u > \text{ for some string } u\}.$$

Polynomial time verifiers and the class NP

Definition (Polynomial time verifier). A polynomial time verifier for a language L is an algorithm V that takes inputs of the form $< w, u >$ and runs in time $p(|w|)$ for some polynomial $p: N \to N$, such that

$w \in L \Leftrightarrow$ there exists u such that V accepts $< w, u >$.

Polynomial time verifiers and the class NP

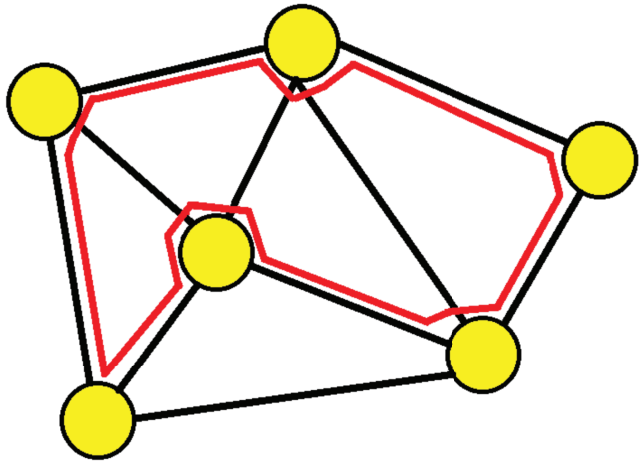
Definition (Polynomial time verifier). A polynomial time verifier for a language L is an algorithm V that takes inputs of the form $< w, u >$ and runs in time $p(|w|)$ for some polynomial $p: N \to N$, such that

$w \in L \Leftrightarrow$ there exists u such that V accepts $< w, u >$.

Definition (The class NP). NP is the class of languages $L \subseteq \{0,1\}^*$ that have polynomial time verifiers.

Informally: NP is the class of problems with efficiently verifiable solutions.

Examples of languages in NP

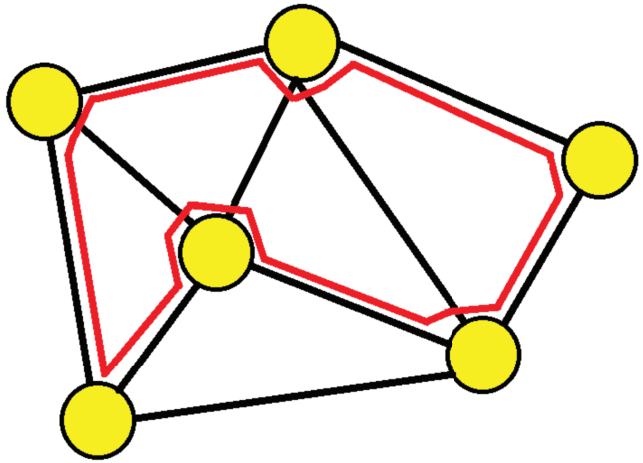


Example (Hamiltonian Path). A Hamiltonian path in a directed graph G is a directed path that goes through every vertex exactly once.

We consider the problem of testing whether a directed graph contains a Hamiltonian path connecting two specified nodes. That is, we define the following language:

$$HAMPATH = \{< G, s, t > \mid G \text{ is a directed graph with a Hamiltonian path from s to t}\}$$

Examples of languages in NP

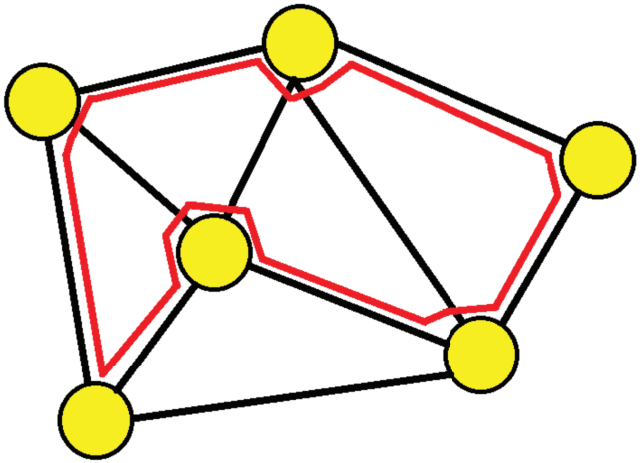


Example (Hamiltonian Path).

$HAMPATH = \{< G, s, t > \mid G$ is a directed graph with a Hamiltonian path from s to t$\}$.

To show that $HAMPATH \in NP$, we need to find a polynomial time verifier.

Examples of languages in NP



Example (Hamiltonian Path).

$HAMPATH = \{< G, s, t > \mid G$ is a directed graph with a Hamiltonian path from s to t$\}$.

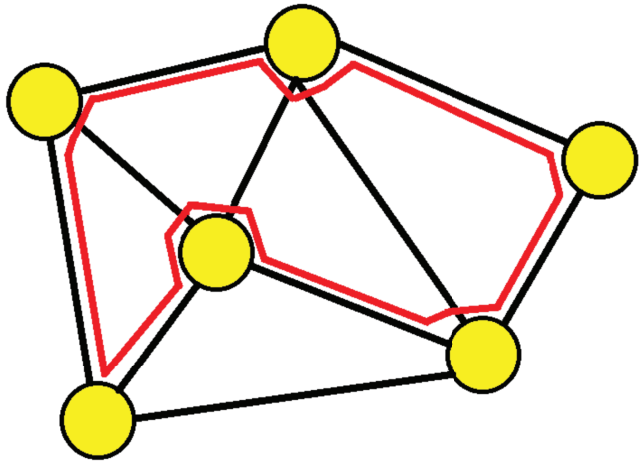
The verifier V will take inputs of the form $< w, u >$, where

- w = the input $< G, s, t >$
- u = candidate path $x_1, \ldots, x_m$

How V works:

- Check if the path $x_1, \ldots, x_m$ is Hamiltonian (i.e. $m = n$, no repeated vertices, each (x_i, x_{i+1}) forms an edge in the graph G)
- Output yes if the path is Hamiltonian and no otherwise.

Examples of languages in NP



Example (Hamiltonian Path).

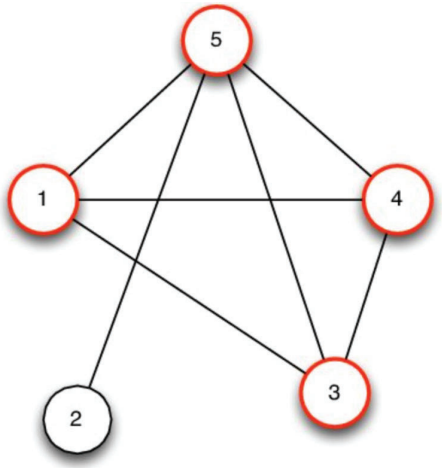
$HAMPATH = \{< G, s, t > \mid G$ is a directed graph with a Hamiltonian path from s to t$\}$.

Since HAMPATH has a polynomial time verifier, we get $HAMPATH \in NP$.

Algorithm for HAMPATH: an option is exhaustive search:

- Run through each sequence of vertices $x_1, \ldots, x_n$ in the graph in lexicographic order and check if it represents a Hamiltonian path.
- Runtime is exponential.

Examples of languages in NP

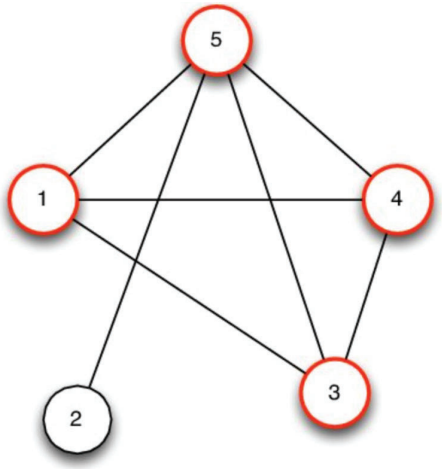


Example (k-Clique).

$CLIQUE = \{< G, k > \mid G$ is an undirected graph with a clique of size k$\}$.

To show that $\mathrm{CLIQUE} \in NP$, we need to find a polynomial time verifier.

Examples of languages in NP



Example (k-Clique).

$CLIQUE = \{< G, k > \mid G$ is an undirected graph with a clique of size k$\}$.

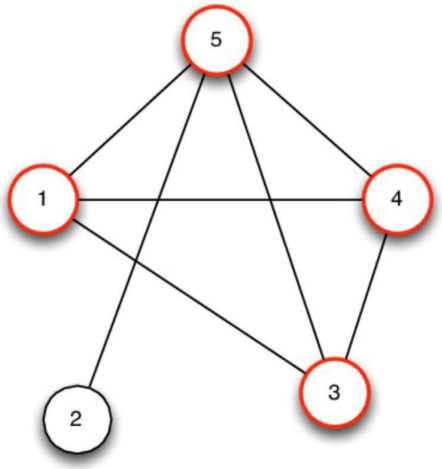
The verifier V will take inputs of the form $< w, u >$, where

- w = the input $< G, k >$
- u = set of vertices of G

How V works:

- Check if u consists of k different vertices $y_1, \ldots, y_k$ of G
- Check if $y_1, \ldots, y_k$ is a clique: if yes, output yes, else output no.

Examples of languages in NP



Example (k-Clique).

$CLIQUE = \{< G, k > \mid G$ is an undirected graph with a clique of size k$\}$.

Since CLIQUE has a polynomial time verifier, we get $CLIQUE \in NP$.

Algorithm for CLIQUE: an option is exhaustive search:

- Run through each possible subset of k vertices $x_1, \ldots, x_k$ in the graph in lexicographic order and check if it they form a clique.
- Runtime is exponential.

Examples of languages in NP

Example (PARTITION).

$PARTITION = \{< S > |$ S is a set of integers such that there is a subset A with the property: $\sum_{x \in A} x = \sum_{x \in A'} x\}$.

For instance, if S = $\{1,\ 5,\ 11,\ 5\}$, a solution is A = {11} and A' = {1,5,5}.

To show that PARTITION $\in NP$, we need to find a polynomial time verifier.

Examples of languages in NP

Example (PARTITION).

$PARTITION = \{< S > \mid$ S is a set of integers such that there is a subset A with the property: $\sum_{x \in A} x = \sum_{x \in A'} x\}$.

The verifier V will take inputs of the form $< w, u >$, where

- w = the input S
- u = set of integers

How V works:

- Check if u consists of a set of different numbers in S
- Check if $\sum_{x \in u} x = \sum_{x \in S \setminus \mathrm{u}} x$

Examples of languages in NP

Example (PARTITION).

$PARTITION = \{< S > \mid$ S is a set of integers such that there is a subset A with the property: $\sum_{x \in A} x = \sum_{x \in A'} x\}$.

Since PARTITION has a polynomial time verifier, we get PARTITION $\in NP$.

Algorithm for PARTITION: an option is exhaustive search:

- Run through each possible subset A of S and check if $\sum_{x \in A} x = \sum_{x \in A'} x$
- Runtime is exponential.

P, NP, and EXP

Definition (The class EXP). $EXP = \bigcup_{k\geq 0} DTIME\left(2^{n^k}\right)$.

Proposition. $\mathrm{P} \subseteq NP \subseteq EXP$.

(***Recall*** $P = \bigcup_{k\geq 0} DTIME\left(n^k\right)$ and NP is the class of languages $L \subseteq \{0,1\}^*$ that have polynomial time verifiers).

P, NP, and EXP

Definition (The class EXP). $EXP = \bigcup_{k \geq 0} DTIME\left(2^{n^k}\right)$.

Proposition. $\mathrm{P} \subseteq NP \subseteq EXP$.

Proof: $\mathrm{P} \subseteq NP$.

Let $L \subseteq \{0,1\}^*$. Suppose $L \in P$. Then there is a polynomial time algorithm A that decides L; that is, given input $w \in \{0,1\}^*$, A returns yes if and only if $w \in L$.

We can design a polynomial time verifier from A by letting the certificate be the empty string; that is, the verifier V takes inputs $< w, u >$ where u is the empty string, and runs A on w, returning its answer.

P, NP, and EXP

Definition (The class EXP). $EXP = \bigcup_{k\geq 0} DTIME\left(2^{n^k}\right)$.

Proposition. $\mathrm{P} \subseteq NP \subseteq EXP$.

Proof: $NP \subseteq EXP$.

Let $L \subseteq \{0,1\}^*$. Suppose $L \in NP$. By definition, there is a polynomial time verifier V that takes inputs of the form $< w, u >$ and runs in time $p(|w|)$ for some polynomial p: $N \to N$, so that: $w \in L \Leftrightarrow \exists u$ s.t. V accepts $< w, u >$.

Can we obtain from this an exponential time algorithm for L, i.e. running in $O\left(2^{n^k}\right)$ on inputs of length n?

P, NP, and EXP

Definition (The class EXP). $EXP = \bigcup_{k\geq 0} DTIME\left(2^{n^k}\right)$.

Proposition. $\mathrm{P} \subseteq NP \subseteq EXP$.

Proof: $NP \subseteq EXP$.

Let $L \subseteq \{0,1\}^*$. Suppose $L \in NP$. By definition, there is a polynomial time verifier V that takes inputs of the form $< w, u >$ and runs in time $p(|w|)$ for some polynomial p: $N \to N$, so that: $w \in L \Leftrightarrow \exists u$ s.t. V accepts $< w, u >$.

We can use V to obtain an exponential time algorithm for L as follows:

- Generate in lexicographic order strings u of length at most p(|w|) and check for each of them if V accepts $< w, u >$.

P, NP, and EXP

Definition (The class EXP). $EXP = \bigcup_{k \geq 0} DTIME\left(2^{n^k}\right)$.

Proposition. $\mathrm{P} \subseteq NP \subseteq EXP$.

Proof: $NP \subseteq EXP$.

E.g., suppose L = PARTITION $= \{< S > \mid$ S is a set of integers that can be divided in two subsets A and $A' = S \backslash A$ with equal sum: $\sum_{x \in A} x = \sum_{x \in A'} x\}$.

Recall $PARTITION \in NP$. On input S, a poly-time verifier V takes inputs of the form $< S, A >$ and checks that A is a valid subset of S and $\sum_{x \in A} x = \sum_{x \in A'} x$.

P, NP, and EXP

Definition (The class EXP). $EXP = \bigcup_{k \geq 0} DTIME\left(2^{n^k}\right)$.

Proposition. $\mathrm{P} \subseteq NP \subseteq EXP$.

Proof: $NP \subseteq EXP$.

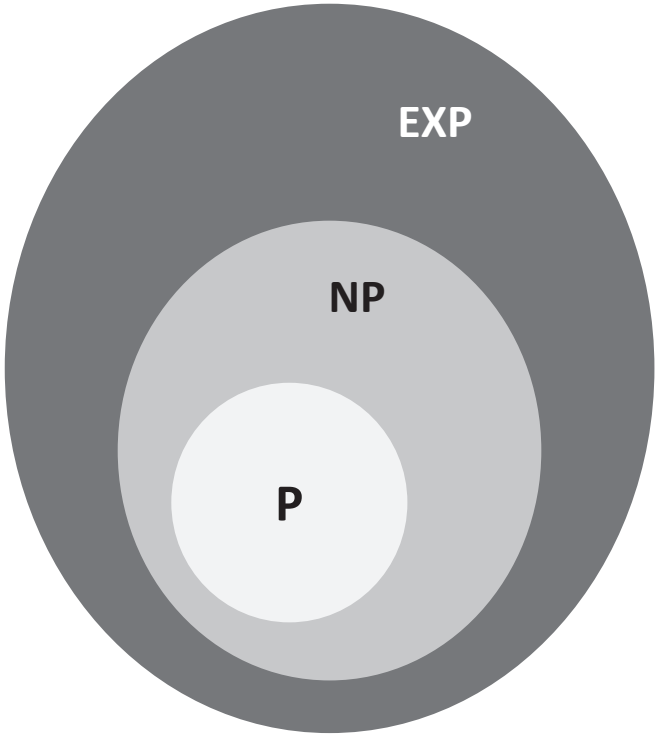
E.g. suppose L = PARTITION.

Recall $PARTITION \in NP$. On input S, a poly-time verifier V takes inputs of the form $< S, A >$ and checks that A is a valid subset of S and $\sum_{x \in A} x = \sum_{x \in A,} x$.

Exponential time algorithm obtained from this verifier: generate in lexicographic order all subsets A of S; for each of them, check if V accepts A.

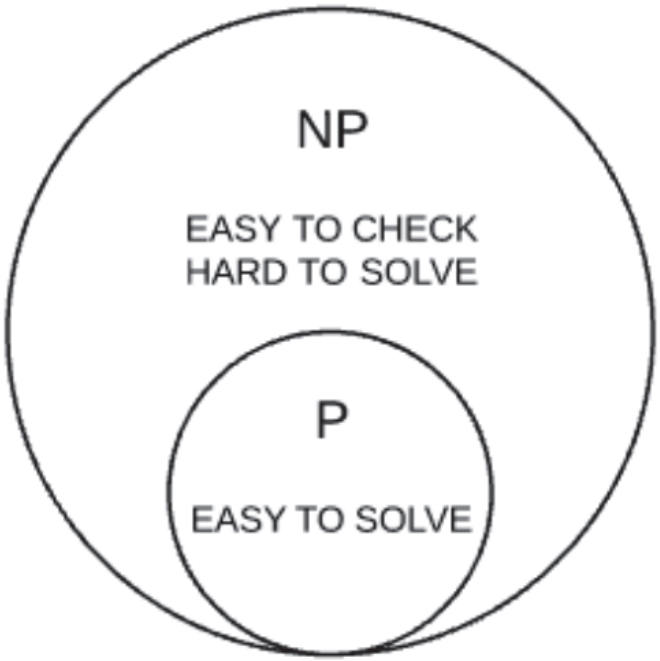
P, NP, and EXP

Proposition. $\mathrm{P} \subseteq NP \subseteq EXP$.



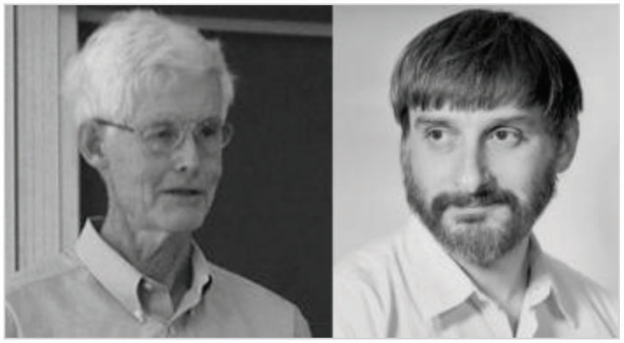
P vs NP

Million dollar question: is P = NP?



Widely believed that $P \neq NP$.

P vs NP Problem



Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971.

Image credit: on the left, Stephen Cook by Jiří Janíček (cropped). CC BY-SA 3.0

Rules:

Rules for the Millennium Prizes

Related Documents:

Official Problem Description

Minesweeper

Related Links:

Lecture by Vijaya Ramachandran

NP completeness

Certain problems in NP have the property that their individual complexity is related to that of the entire class.

- That is, if a polynomial time algorithm exists for any of these problems, then all problems in NP are solvable in polynomial time.
- These problems are called **NP-complete**.

Thus:

- If any problem in NP requires super-polynomial time, then $P \neq NP$.
- If any NP-complete problem is solvable in polynomial time, then $P = NP$.

NP completeness

Practical implications of $P \neq NP$: don't waste time trying to design a polynomial time algorithm for an NP-complete problem. If you can show the problem at hand is NP-complete, that is strong evidence that there is no polynomial time algorithm for it.

Satisfiability

A basic problem that will turn out to be NP-complete is called **Satisfiability**.

Recall:

Boolean variables take values True or False (i.e. 1 or 0). Boolean operations AND, OR, NOT.

Boolean formula: expression involving Boolean variables and operations. E.g.

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

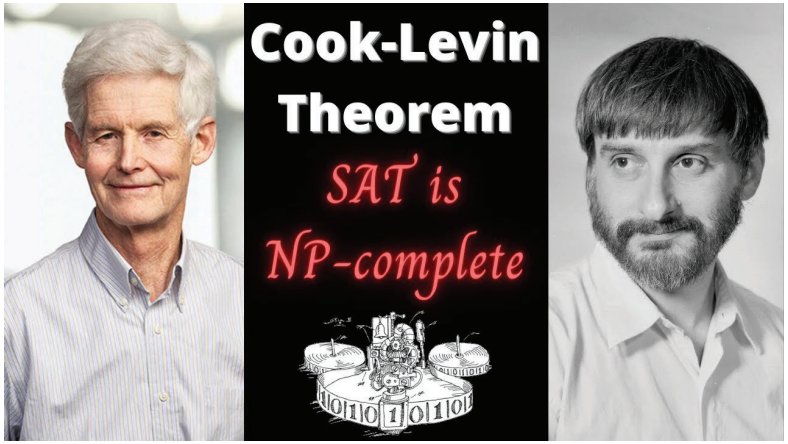
A Boolean formula is satisfiable if some assignment of the variables makes the formula evaluate to 1.

Satisfiability

The Satisfiability problem is to test whether a Boolean formula is satisfiable.

$SAT = \{< \phi > | \phi$ is a satisfiable Boolean formula $\}$.

Satisfiability



The Satisfiability problem is to test whether a Boolean formula is satisfiable.

$$SAT = \{< \phi > | \phi \text{ is a satisfiable Boolean formula } \}.$$

Cook-Levin Theorem: $SAT \in P$ if and only if $P = NP$.

[independently discovered by Cook 1971 (US); Levin 1973 (USSR) – but lectured on it earlier]

To prove the theorem, we will introduce a few notions.

Polynomial time reducibility

Definition. A function $f: \{0,1\}^* \to \{0,1\}^*$ is a **polynomial time computable function** if there exists a Turing machine M that runs in polynomial time and for each input $w \in \{0,1\}^*$, it halts with just $f(w)$ on the tape.

Polynomial time reducibility

Definition. A function $f: \{0,1\}^* \to \{0,1\}^*$ is a **polynomial time computable function** if there exists a Turing machine M that runs in polynomial time and for each input $w \in \{0,1\}^*$, it halts with just $f(w)$ on the tape.

Definition. A language A is **polynomial time mapping reducible** (or polynomial time reducible or polynomial time many-one reducible) to language B, denoted $A \leq_p B$ if there exists a polynomial time computable function $f: \{0,1\}^* \to \{0,1\}^*$ such that for each $w \in \{0,1\}^*$, we have

$$w \in A \Leftrightarrow f(w) \in B.$$

The function f is called the **polynomial time reduction** of A to B.

Polynomial time reducibility

Definition. A function $f: \{0,1\}^* \to \{0,1\}^*$ is a **polynomial time computable function** if there exists a Turing machine M that runs in polynomial time and for each input $w \in \{0,1\}^*$, it halts with just $f(w)$ on the tape.

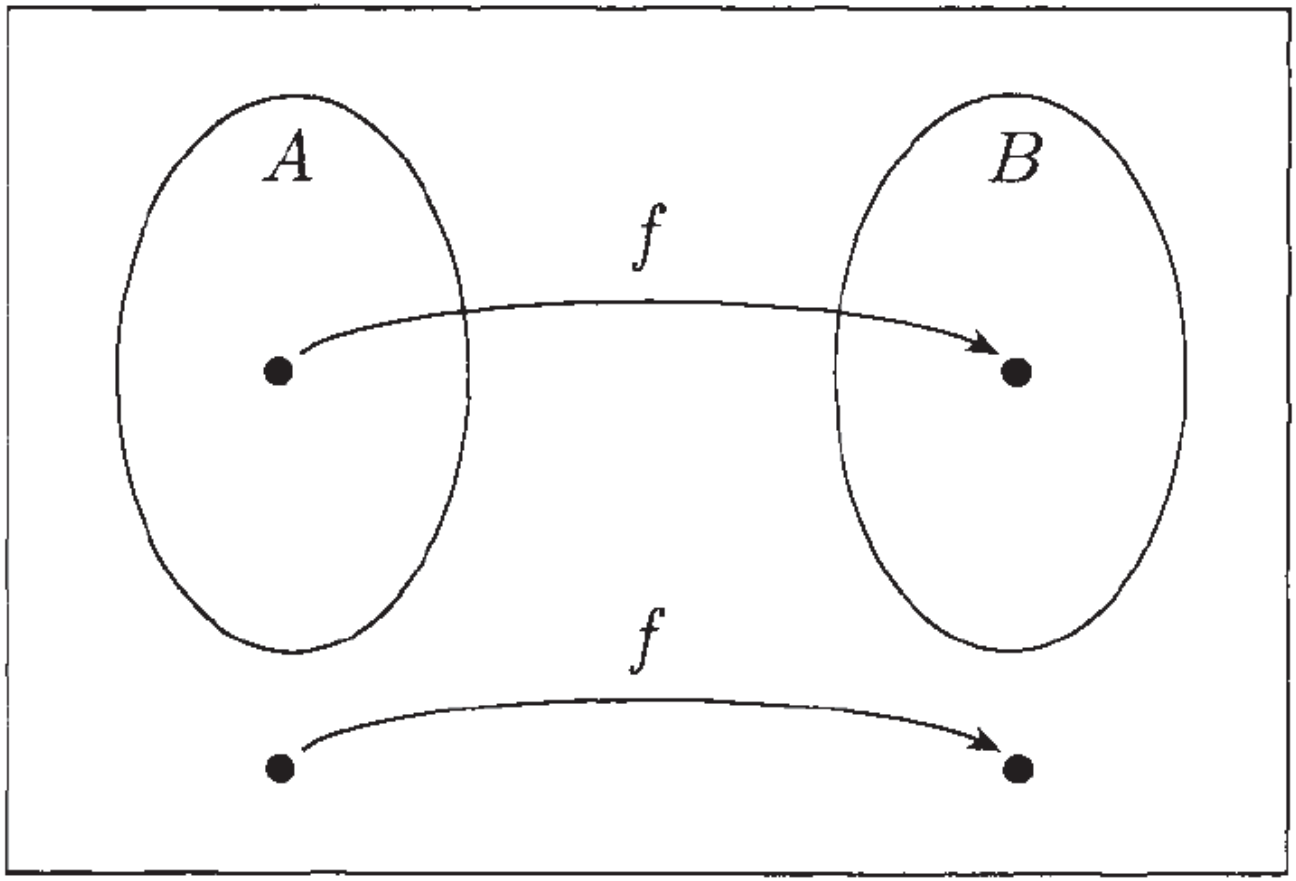
Definition. A language A is **polynomial time mapping reducible** (or polynomial time reducible or polynomial time many-one reducible) to language B, denoted $A \leq_p B$ if there exists a polynomial time computable function $f: \{0,1\}^* \to \{0,1\}^*$ such that for each $w \in \{0,1\}^*$, we have

$$w \in A \Leftrightarrow f(w) \in B.$$

The function f is called the **polynomial time reduction** of A to B.

Note: *More sophisticated types of reductions exist; this type will suffice for understanding the main ideas.*

Polynomial time reducibility



Polynomial time function *f* reducing A to B.

Polynomial time reducibility

Suppose $A \leq_p B$ and $B \in P$. What can we infer about A?

(***Recall:*** Language A is polynomial time mapping reducible to language B,

denoted $\mathrm{A} \leq_{\mathrm{p}} \mathrm{B}$ if there exists a polynomial time computable function

$f: \{0,1\}^* \to \{0,1\}^*$ such that for each $w \in \{0,1\}^*$, we have $\mathrm{w} \in \mathrm{A} \Leftrightarrow f(\mathrm{w}) \in \mathrm{B}$)

Polynomial time reducibility

Proposition. If $A \leq_p B$ and $B \in P$, then $A \in P$.

Polynomial time reducibility

Proposition. If $A \leq_p B$ and $B \in P$, then $A \in P$.

Proof. Let M be the poly-time algorithm deciding B and f the poly-time reduction from A to B.

We can get a poly-time algorithm N for deciding A as follows:

N = On input $w \in \{0,1\}^*$:

- Compute f(w)
- Run M on input f(w) and return M's output.