

# Problems

**Decision Problems:** Is there a (feasible) solution?

Examples:

1) PATH: Given a graph  $G$  and two vertices  $u, v$  in  $G$  and an integer  $k$ , is there a path of length at most  $k$  between  $u$  and  $v$ ?

2) HAMILTONIAN PATH: Given a graph  $G$ , is there a simple path containing all vertices in  $G$ ?

**Optimization Problems:** Feasible solution with the best value.

Examples:

1) Given a graph  $G$  and two vertices  $u, v$  in  $G$  find a shortest path (with minimum number of edges) between  $u$  and  $v$ .

2) Given a graph  $G$ , find the longest simple path in  $G$ .

If the optimization problem is easy, its decision version is easy as well.

Usually, we can recast the optimization problem as a decision problem that is no harder.

# Polynomial-time Reductions

We want to solve a decision problem  $A$  in polynomial time.

Suppose we know how to solve a different decision problem  $B$  in polynomial time.

We can solve  $A$  using  $B$  if we have a polynomial-time procedure that transforms **any** instance  $x$  of  $A$  into some instance  $y$  of  $B$  such that the answer for  $x$  is “yes” if and only the answer for  $y$  is “yes”.

$B$  is as hard as  $A$ .

Suppose  $A$  is a “hard” problem, say, it has no polynomial time algorithm.

Then the above reduction can be used to show that  $B$  has no polynomial time algorithm as well.

# Polynomial time

An **abstract problem** is a binary relation on a set  $I$  of instances and a set  $S$  of solutions.

For example, PATH problem is a mapping from an instance  $\langle G, u, v, k \rangle$  to 0 (no) or 1 (yes).

An algorithm takes an **encoding** of the problem instances as input.

A **binary** encoding of a problem is a mapping from the set  $I$  of instances to the set of binary strings.

Then, we say that a problem is polynomial-time solvable if there exists an algorithm to solve an instance of length  $n$  (in binary encoding) in time  $O(n^k)$  for some constant  $k$ .

$P$  is the set of decision problems which are polynomial-time solvable.

We will assume that all problem instances are binary encoded in some “standard” format.

## Example: Hamiltonian Path

Given a graph  $G$ , does  $G$  have a Hamiltonian Path?

One possible algorithm is to check all possible permutations of the vertices.

A “reasonable encoding” of  $G$  is that of its adjacency matrix.

If  $n$  is the number of vertices of  $G$ , then the size of the encoding is  $O(n^2)$ , but the running time of the algorithm is  $\Omega(n!) = \Omega(2^n)$  which is not a polynomial in the size of the encoding.

# NP

A **certificate** is a “proof” that an instance of a decision problem has a “yes” answer.

A verification algorithm takes an instance of the problem and a **certificate** and uses it to verify whether the decision problem has a “yes” solution.

**NP** is the set of decision problems that have polynomial-time verifiers.

This means that for problems in NP certificates are polynomial in the size of the input and the verification algorithm takes polynomial time. Thus for every “yes” instance of an NP problem, there is a polynomial-size certificate.

Example: HAMILTONIAN PATH  $\in$  NP.

The certificate is a permutation of the vertices and can be verified in polynomial time.

“Guess and Verify” method.

# NP-Complete Problems

The “hardest” problems in NP.

A problem  $B$  is NP-complete if it satisfies two conditions:

1.  $B$  is in NP and
2.  $B$  is **NP-Hard**: every  $A$  in NP is polynomial time reducible to  $B$ .

## P vs. NP

**Theorem 1.** *If any NP-complete problem is polynomial-time solvable then  $P = NP$ . Equivalently, if any problem in NP is not polynomial time solvable, then no NP-complete problem is polynomial-time solvable.*

**Proof.** Suppose  $A \in P$  and  $A$  is NP-complete.

Let  $B$  be any NP problem.

We can reduce  $B$  to  $A$  in polynomial time and hence we can solve  $B$  in polynomial time.  $\square$

## A NP-complete Problem

A boolean formula is in Conjunctive Normal Form (CNF) if it is a collection of clauses joined by ANDs ( $\wedge$ ), where a clause is a collection of literals joined by ORs ( $\vee$ ).

(A literal is a boolean variable or its negation.)

The size of the formula is the sum of the literals in all the clauses and the number of clauses.

A boolean formula is satisfiable if there is a boolean assignment of values to its variables (either TRUE (1) or FALSE (0)) which will make the formula evaluate to TRUE (1).

SAT: Given a boolean formula  $F$  in CNF, is  $F$  satisfiable?

$SAT \in NP$ .

**Theorem 2. [Cook-Levin Theorem]** *SAT is NP-complete.*

## How to show that $A$ is NP-complete

1. Show  $A \in NP$ .
2. Select a known NP-complete problem  $B$ .
3. Do a **polynomial-time reduction**: Give a polynomial time algorithm which will transform an instance  $x'$  of  $B$  to an instance of  $x$  of  $A$  such that  $x'$  has a “yes” answer (for  $B$ ) if and only if  $x$  has a “yes” answer (for  $A$ ).

# 3-SAT

A boolean formula is in 3-SAT form if it is in CNF form and each clause has exactly three literals per clause.

**Theorem 3.** *3-SAT is NP-Complete.*

**Proof.**

$3\text{-SAT} \in NP.$

We will give a polynomial-time reduction from SAT to 3-SAT.

Consider an instance  $F$  of SAT. We convert it into an instance  $F'$  of 3-SAT such that  $F$  is satisfiable iff  $F'$  is satisfiable.

In each clause of  $F$  that has 1 or 2 literals, we replicate one of the literals till the total number is three.

If a clause has more than 3 literals, say,  $(a_1 \vee a_2 \vee \dots \vee a_l)$ , then we replace it with the following  $l - 2$  clauses:

$$(a_1 \vee a_2 \vee z_1) \wedge (z_1^c \vee a_3 \vee z_2) \wedge (z_2^c \vee a_4 \vee z_3) \wedge \dots \wedge (z_{l-3}^c \vee a_{l-1} \vee a_l).$$

The size of  $F'$  is only polynomially larger than  $F$ .

If a clause in  $F$  is satisfiable then all the corresponding clauses in  $F'$  can be satisfied and viceversa.

□