Complexity Theory

So far in this course, almost all algorithms had **polynomial running time**, i.e., on inputs of size n, worst-case running time is $O(n^k)$ for some constant k.

Is this always the case?

Obviously **no** (just consider basic FF algorithm: running time depends on $|f^*|$).

Some problems are even "**insolvable**", regardless of how much time is provided, e.g., the famous *halting problem*:

> It is not possible to tell, in general, whether a given algorithm will halt for some given input.

Also, there are problems that are solvable, but not in time $O(n^k)$ for *any* constant k (they may require strictly exponential time, or even more).

Problems are divided into **complexity classes**. Informally:

$\mathcal{P}$ is the class of problems for which there are algorithms that **solve** the problem in time $O(n^k)$ for some constant k.

$\mathcal{NP}$ is the class of problems for which there are algorithms that **verify** solutions in time $O(n^k)$ for some constant k.

Intuitively, solving harder that merely verifying, so maybe problems in $\mathcal{NP}$-$\mathcal{P}$ "harder" than problems in $\mathcal{P}$.

. . . but. . .

is there something in $\mathcal{NP}$-$\mathcal{P}$????!?

One of the most famous question in TCS:

$$\mathcal{P} = \mathcal{NP} \text{ or } P \neq \mathcal{NP} \text{ ?}$$

Obviously, $\mathcal{P} \subseteq \mathcal{NP}$.

Study "$\mathcal{NP}$-complete" problems. They are in $\mathcal{NP}$, and, in a sense, they are "**hard**" among all problems in $\mathcal{NP}$ (or, as hard as any other); will be formalised later.

Will prove later: if one can show for **only one** $\mathcal{NP}$-complete problem that it is in fact in $\mathcal{P}$, then $P = \mathcal{NP}$. Seems to be difficult, so far nobody has succeeded ;-)

Sometimes slight modifications to some problem make a huge difference.

Euler tour vs Hamiltonian cycle. Euler tour: does given directed graph have a cycle that traverses each *edge* exactly once? Easy, $O(E)$. Hamiltonian cycle: does given directed graph have a simple cycle that contains each *vertex*? $\mathcal{NP}$-complete!

Back to this **verification** business.

Example **Hamiltonian cycle**. As mentioned, $\mathcal{NP}$-complete, so apparently hard, perhaps no polynomial-time algorithm that can compute a solution (for all instances).

But: *verifying* a solution/certificate is trivial! Certificate is sequence

$$(v_1, v_2 m \ldots, v_{|V|})$$

(some ordering of vertices), just have to check whether it's a proper cycle.

Techniques for proving $\mathcal{NP}$-completeness **differ** from "usual" techniques for designing or analysing algorithms. In the following, a few key concepts.

1) Decision problems vs optimisation problems

Many problems are *optimisation problems*: compute shortest paths, maximum matching, maximum clique, maximum independent set, etc.

Concept of $\mathcal{NP}$-completeness does not apply (directly) to those, but to *decision problems*, where answer is just "*yes*" or "*no*"

Does this given graph have a Hamiltonian cycle?

Given this graph, two vertices, and some k, does G contain a path of length at most k from u to v?

However, usually close relationship. In a sense, decision is "easier" than "optimisation" (or not harder).

Example: can solve SP decision problem by solving SP optimisation: just compare length of (computed) shortest path with k.

General idea: evidence that decision problem is hard implies evidence, that related optimisation problem is hard.

2) Encodings

When we want an algorithm to solve some problem, **problem instances** must be **encoded**.

For instance, encode natural numbers as strings

$$\{0, 1, 10, 11, 100, 101, \ldots\}$$

encoding $e : \mathbb{N} \to \{0,1\}^*$ with $e(i) = (i)_2$

Concrete problem: problem whose instance set is the set of binary strings (read: encodings of "real" instances)

We say an alg. solves some concrete decision problem in time $O(T(n))$ if, when given problem instance i of length $n = |i|$, it can produce solution in time $O(T(n))$.

A concrete problem is poly-time solvable, if there is an alg. to solve it in time $O(n^k)$ for some constant k. $\mathcal{P}$ is the set of all poly-time solvable concrete decision problems.

Would like to talk about **abstract problems** rather than concrete ones, **independent** of any particular encoding.

But. . . efficiency depends heavily on encoding.

Example: suppose integer k is input to an alg., running time is $\Theta(k)$. If k is given in **unary**, then then running time is $O(n)$ in length-n inputs, polynomial. If k is given **binary**, then input length is $n = \lfloor \log k \rfloor + 1$; exponential running time $\Theta(k) = \Theta(2^n)$.

In practice, rule out "expensive" encodings like unary.

A formal-language framework

An **alphabet** Σ is finite set of sysmbols.

A **language** L over Σ is any set of strings made up of symbols from Σ.

Empty string is ϵ, **empty language** is $\emptyset$.

Language of **all strings** over Σ is Σ^*
Ex: $\Sigma = \{0, 1\} \Rightarrow \Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$, all binary strings.

Every language L over Σ is subset of Σ^*.

Union $L_1 \cup L_2$ and **intersection** $L1 \cap L_2$ just like with ordinary sets, **complement** $\bar{L} = \Sigma^* - L$.

Concatenation of L_1 and L_2 is $\{xy : \ x \in L_1, y \in L_2\}$.

Closure of L is $\{\epsilon\} \cup L \cup L^2 \cup L^3 \cup \cdots$ with L^k being L concatenated to itself k times.

Set of instances for any decision problem Q is Σ^*, where $\Sigma = \{0, 1\}$.

Q is entirely characterised by those instances that produce 1 (*yes*), so we view Q as language L over $\Sigma = \{0, 1\}$ with

$$L = \{x \in \Sigma^* : \ Q(x) = 1\}$$

Ex: ($<$something$>$ meaning "standard" enc. of something)

$$\begin{aligned} \text{HAMILTON} &= \{< G > : \ G \text{ contains Hamiltonian cycle}\} \\ \text{SAT} &= \{< F > : \ \text{formula } F \text{ is satisfyable}\} \end{aligned}$$

Encoding: Using formal languages is ”fine”

We say function $f : \{0,1\}^* \to \{0,1\}^*$ is **polynomially computable** if there is polynomial-time TM M that, given $x \in \{0,1\}^*$, computes $f(x)$.

For set I of problem instances, two encodings e_1 and e_2 are **polynomially related** if there are two poly-time computable functions f_{12} and f_{21} such that $\forall i \in I$,

$$f_{12}(e_1(i)) = e_2(i) \text{ and } f_{21}(e_2(i)) = e_1(i)$$

If two encodings of some abtract problem are polynomially related, then whether problem is in $\mathcal{P}$ is independent of which encoding we use.

Note: the length can increase only by a polynomial factor!

It can be shown: the ”standard inputs” (graph as an adjacency matrix,...) are polynomially related.

Lemma. Let Q be an abstract decision problem on an instance set I, let e_1, e_2, be polynomially related encodings on I. Then, $e_1(Q) \in \mathcal{P}$ iff $e_2(Q) \in \mathcal{P}$.

Proof. We show one direction ($e_1(Q) \in \mathcal{P} \Rightarrow e_2(Q) \in \mathcal{P}$), other is symmetric.

Suppose $e_1(Q) \in \mathcal{P}$, i.e., $e_1(Q)$ can be solved in time $O(n^k)$ for some constant k, and that for any instance i, $e_1(i)$ can be computed from $e_2(i)$ in time $O(n^c)$ for some constant c, $n = |e_2(i)|$.

To solve $e_2(Q)$, on input $e_2(i)$, first compute $e_1(i)$, and run alg. for $e_1(Q)$ on $e_1(i)$.

Time: conversion takes $O(n^c)$, therefore $|e_1(i)| = O(n^c)$ (can't write more). Solving $e_1(i)$ takes $O(|e_1(i)|^k) = O(n^{ck})$, polynomial. Thus $e_2(Q) \in \mathcal{P}$.

3) Machine Model

Our goal is to say that Problem A can not be solved in polynomial time.

But on which machine? A parallel machine with 1000 processors? A modern computer? Can we solve more in polynomial time if the computers get faster?

In complecity theory people use a very simple machine model, the so-called Turing machine.

One can show that everything that can be solved by a modern computer in polynomial time can also be solved on a TM in polynomial time.

Machine Model

A TM (Turing machine) consists of a tape of infinite length (in one direction) and a pointer. The tape consists of cells and every cell can store one symbol.

The pointer points to one of the memory cells. In the beginning the input is in cell $1, \ldots..$ and the pointer position is cell 1.

The TM has a finite *control*. In every step the control is in one of a finite amount of states.

In every step the TM does the following.

- It reads the symbol at the actual pointer position
- It writes a new symbol into the position (which one depends on the state)
- It can move the pointer one step to the left or to the right.

Formally, a *Turing Machine* $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is defined as follows.

- Q is the (finite) set of states.
- Γ is the set of tape symbols.
- Σ is the set of input symbols ($\Gamma \subset \Sigma$).
- $B = \{N, R, L\}$.
- $\delta : (Q \times \Gamma) \rightarrow (Q \times \Gamma \times \{L, R, N\}$ is the transition function.
- $q_0 \in Q$ is the initial state.
- $F \subset Q$ is an end state.

State of the TM:

$\alpha_1 q \alpha_2$ with $q \in Q$ and $\alpha_1, \alpha_2 \in \Gamma^*$. here $\alpha = \alpha_1 \alpha_2$ is the contents of the tape.

If now $\alpha_2 = \alpha \alpha_2'$ the TM looks up $(q, \alpha) = (q', \beta, M)$.

- If $M = N$ the next state will be $\alpha_1 q' \beta \alpha_2'$.
- If $M = R$ the next state will be $\alpha_1 \beta q' \alpha_2'$.
- If $M = R$ and $\alpha_1 = \alpha_1' \gamma$ the next state will be $\alpha_1 q' \gamma \beta \alpha_2'$.

We write $\alpha_1 q \alpha_2 \Rightarrow \alpha_1' q' \alpha_2'$ if a one step transition from $\alpha_1 q \alpha_2$ to $\alpha_1' q' \alpha_2'$ exists.

We write $\alpha_1 q \alpha_2 \overset{\Rightarrow}{*} \alpha_1' q' \alpha_2'$ if a (possibly long) transition from $\alpha_1 q \alpha_2$ to $\alpha_1' q' \alpha_2'$ exists.

Examples for TM: see homework.

The language L **accepted** by TM M is the set of words from Σ^* so that M reaches a state in F. To make our life easier we will say in the following that M outputs `1`.

In all other cases M **rejects** in input. It can stop in a state not in F and output `0`, or M can go into an endless loop.

We say TM M **accepts** string $x \in \{0,1\}^*$ if, given input x, M's output is $M(x) = 1$.

The language **accepted** by M is

$$L = \{x \in \Sigma^* : \ M(x) = 1\}.$$

M **rejects** x if $M(x) = 0$.

Important: even if L is accepted by some TM M, M will not necessarily reject $x \notin L$.

A language L is **decided** by TM M, if L is accepted by M and every string not in L is rejected.

L is **accepted/decided in polynomial time** by TM M, if it is accepted/decided by M in time $O(n^k)$ for some const. k.

Now: alternative definition

$$P = \{L \subseteq \{0,1\}^* : \ \exists \text{ TM } M \text{ that } \textbf{decides } L \text{ in poly-time}\}$$

Moreover:

Theorem. $P = \{L : L$ is **accepted** by a TM in poly-time$\}$.

Proof. Clearly decided$\Rightarrow$accepted, so we need only show accepted$\Rightarrow$decided.

Let L be accepted by M in time $O(n^k)$, thus there is constant c s.t. M accepts L in at most $T = cn^k$ steps.

Construct M': for any input string x, M' simulates M for T steps. If M has accepted, then M' accepts x by outputting 1, otherwise M' rejects by outputting 0 *q.e.d.*

Note: proof is nonconstructive, we may not know runtime bound for given $L \in \mathcal{P}$ (but it exists!).

Poly-time verification

Look at TMs that **verify membership** in languages.

Ex: for instance $< G >$ of HAMILTON, we are also given ordering c of vertices on cycle. Can easily check whether c *is* a proper cycle on all vertices of G, so c is **certificate** that G indeed belongs to HAMILTON.

As we will see later, HAMILTON is $\mathcal{NP}$-complete, and thus most likely *not* in $\mathcal{P}$, but verification is easy.

Define **verification TM**: two argument TM M , one argument is ordinary input string x, other is binary string y called **certificate**. M verifies input string x if $\exists$ certificate y s.t. $M(x, y) = 1$. The **language verified** by M is

$$L = \{x \in \{0, 1\}^* : \exists y \in \{0, 1\}^* \text{ s.t. } M(x, y) = 1\}$$

Intuition: M verifies L if $\forall x \in L$ there is y that M can use to prove $x \in L$. For any $x \notin L$, there must be no certificate proving that $x \in L$.
Ex: If $G \notin$ HAMILTON there must be no permutation of vertices that can fool verifier into believing G is hamiltonian.

Def: a language L belongs to $\mathcal{NP}$ if and only if $\exists$ two-input poly-time TM M and constant c s.t.

$$L = \{x \in \{0,1\}^* : \exists \text{ certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } M(x,y) = 1\}$$

Historically, $\mathcal{NP}$="*non-deterministic poly-time*", all the problems that are accepted by a poly-time non-deterministic Turing machine (NTM), which

1. non-deterministically "guesses" a solution (certificate) if there is one, and
2. deterministically verifies it

in poly-time. See the similarity?

Facts:

$\mathcal{NP} \neq \emptyset$ (HAMILTON$\in \mathcal{NP}$)

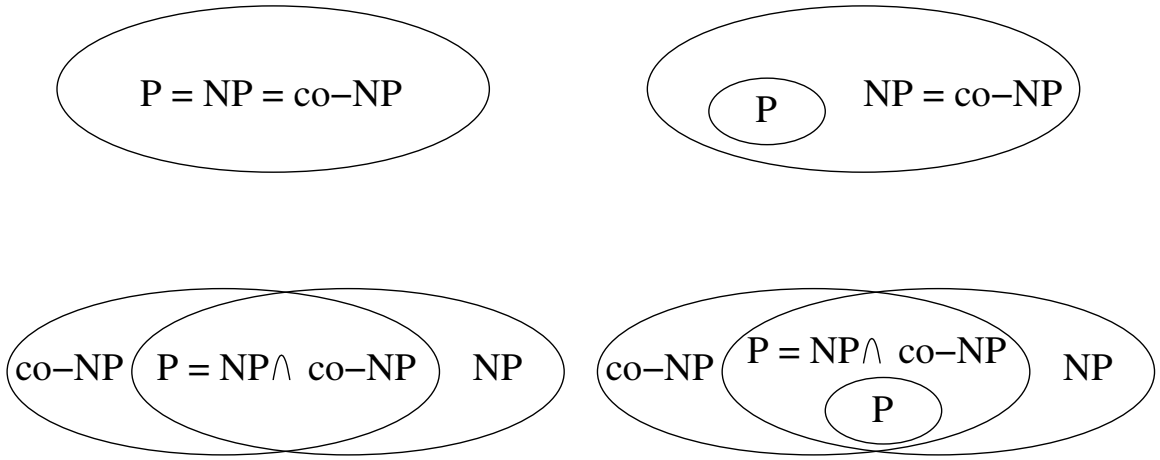
$\mathcal{P} \subseteq \mathcal{NP}$

Open questions:

$\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \neq \mathcal{NP}$?

$\mathcal{NP}$ closed under complement, i.e., $L \in \mathcal{NP} \Rightarrow \bar{L} \in \mathcal{NP}$?
(with co-$\mathcal{NP} = \{L : \bar{L} \in \mathcal{NP}\}$, equiv. to $\mathcal{NP} =$ co-$\mathcal{NP}$)

Since $\mathcal{P}$ **is** closed under complement, $\mathcal{P} \subseteq \mathcal{NP} \cap$ co-$\mathcal{NP}$. Thus four possibilities:



TL: $\mathcal{NP} =$ co-$\mathcal{NP}$ and $P = \mathcal{NP}$, most unlikely of the four
TR: $\mathcal{NP} =$ co-$\mathcal{NP}$ and $\mathcal{P} \neq \mathcal{NP}$
BL: $\mathcal{NP} \neq$ co-$\mathcal{NP}$ and $P = \mathcal{NP} \cap$ co-$\mathcal{NP}$
BR: $\mathcal{NP} \neq$ co-$\mathcal{NP}$ and $P \neq \mathcal{NP} \cap$ co-$\mathcal{NP}$, most likely

$\mathcal{NP}$-completeness

Remember: class of $\mathcal{NP}$-complete problems; property: if **one** of them is in $\mathcal{P}$, then all of $\mathcal{NP}$.

Reducibility

Intuition: problem Q can be reduced to Q' if any instance of Q can be "easily rephrased" as instance of Q'

We say that language L_1 is **poly-time reducible** to language L_2, written $L_1 \leq_p L_2$, if there exists a poly-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ s.t. $\forall x \in \{0,1\}^*$,

$$x \in L_1 \text{ if and only if } f(x) \in L_2.$$

f is called **reduction function**.

$\mathcal{NP}$-completeness II

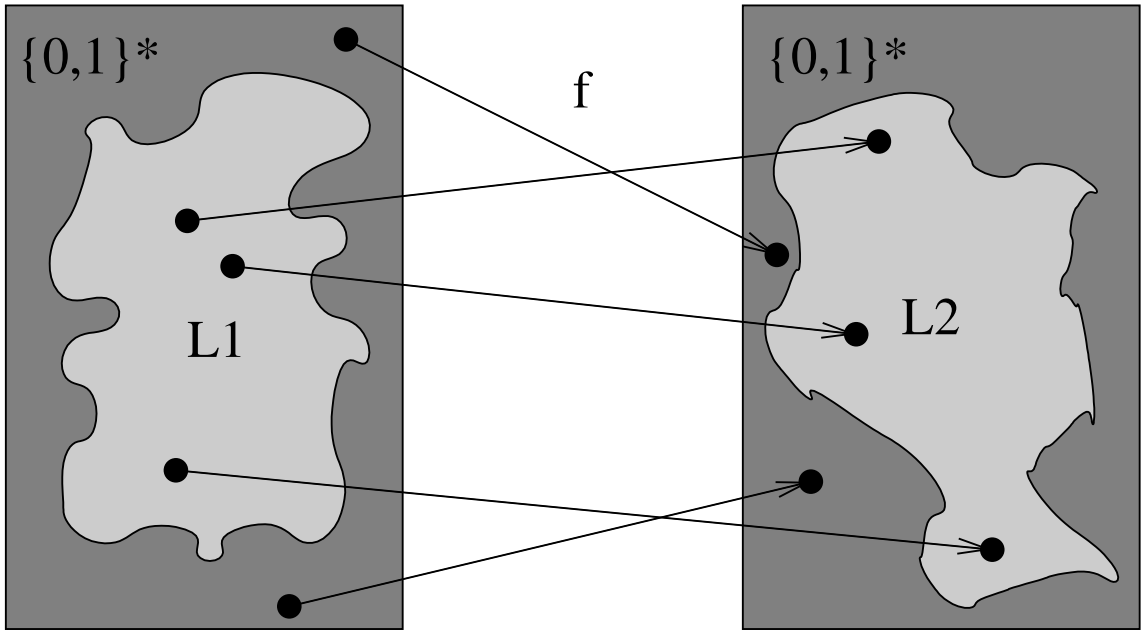
A language $L \subseteq \{0,1\}^*$ is called $\mathcal{NP}$**-complete** if

1. $L \in \mathcal{NP}$, and
2. $L' \leq_p L$ for every $L' \in \mathcal{NP}$.

If L satisfies only (2) it is called $\mathcal{NP}$**-hard**.

Idea of poly-time reduction from L_1 to L_2:

- $L_1, L_2 \subseteq \{0,1\}^*$
- f provides poly-time mapping s.t.
  - if $x \in L_1$ then $f(x) \in L_2$
  - if $x \notin L_1$ then $f(x) \notin L_2$
- Thus f maps any instance x of decision problem represented by L_1 to instance $f(x)$ of problem represented by L_2
- Answer to whether $f(x) \in L_2$ directly provides answer to whether $x \in L_1$


{0,1}*
f
{0,1}*
L1
L2