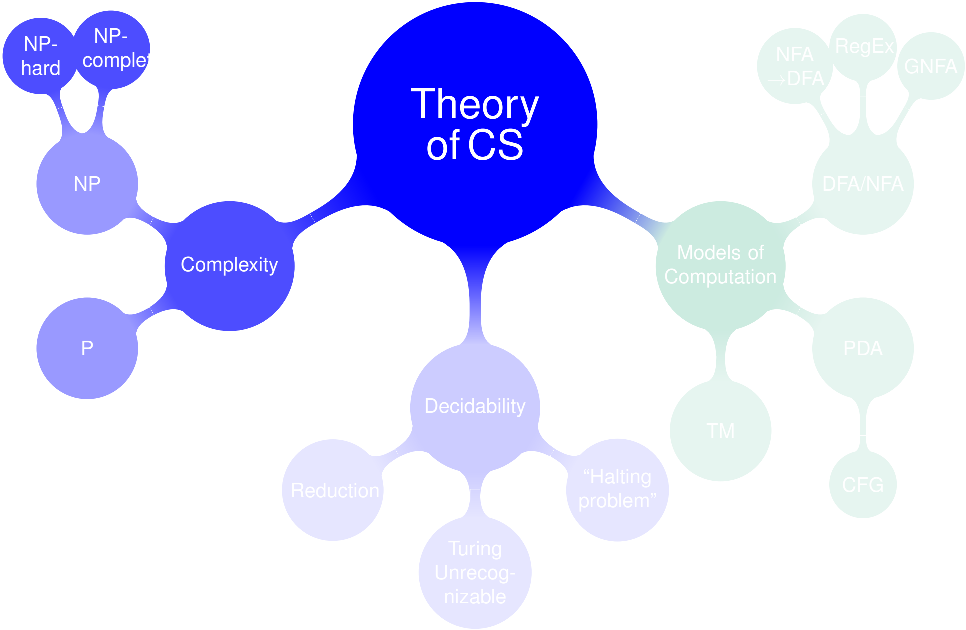
NP-Completeness

Dr Kamal Bentahar

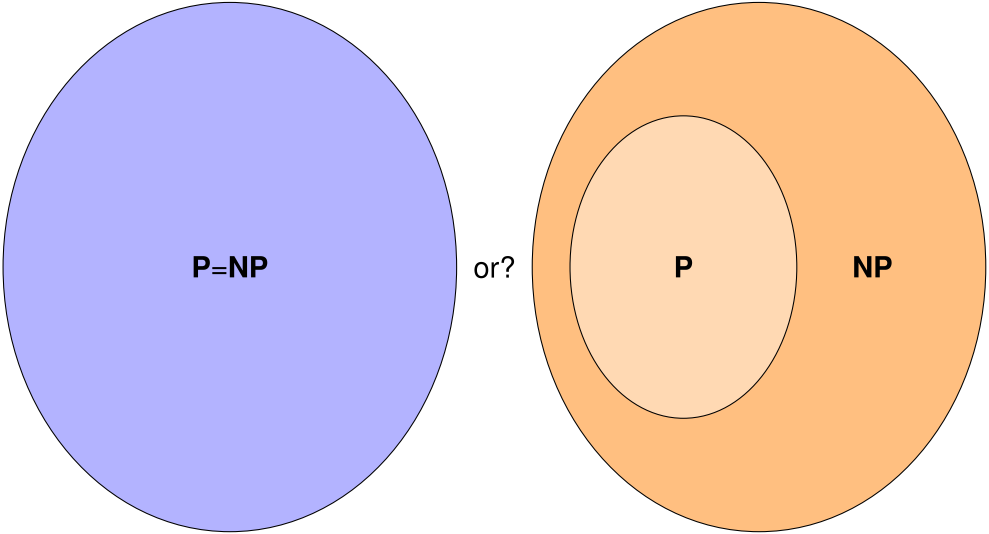
School of Computing, Electronics and Mathematics
Coventry University

Lecture 9



Theory of CS
Complexity
NP
NP-hard
NP-complet
P
Decidability
Reduction
Turing Unrecog-nizable
"Halting problem"
Models of Computation
DFA/NFA
NFA →DFA
RegEx
GNFA
PDA
CFG
TM

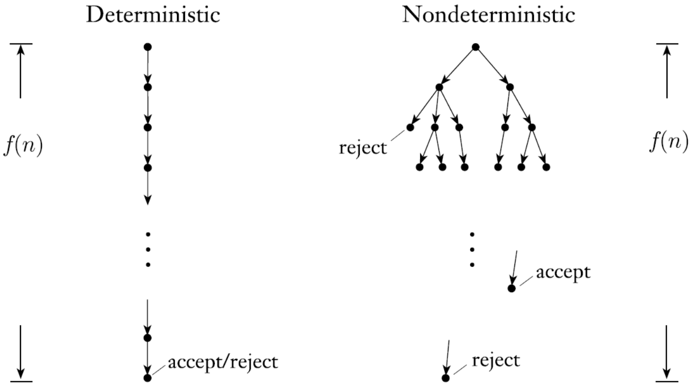
Last time…



Time complexity

The **time complexity** of a decider (TM that always halts) is the <u>maximum</u> number of steps that it makes on **any** input of length *n*.

For nondeterministic TMs consider **all the branches** of its computation.



The class **P**

Nondeterministic Polynomial time complexity class

$$TIME(t(n)) = \{\text{Languages decided by an } O(t(n)) \text{ time deterministic TM}\}.$$

The class **P**

Class of languages that are decidable in polytime on a deterministic TM.

$$\mathbf{P} = TIME(1) \cup TIME(n) \cup TIME(n^2) \cup TIME(n^3) \cup \cdots .$$

The class **NP**

Nondeterministic Polynomial time complexity class

$$NTIME(t(n)) = \{\text{Languages decided by an } O(t(n)) \text{ time non-deterministic TM}\}.$$

The class **NP**

Class of languages that are decidable in polytime on a non-deterministic TM.

$$\mathbf{NP} = NTIME(1) \cup NTIME(n) \cup NTIME(n^2) \cup NTIME(n^3) \cup \cdots .$$

NP is the class of languages that have polynomial time verifiers.

The satisfiability problem

- Boolean variables (*true*, *false* or 0, 1)
- Logic operations ($\wedge, \vee, \neg$)
- Boolean formula, e.g.

$$x$$
$$\bar{x}$$
$$x \wedge y$$
$$x \vee \bar{y}$$
$$x \wedge \bar{x}$$
$$\bar{x} \wedge (x \vee y)$$
$$(y \vee \bar{z}) \wedge (x \vee y)$$

- "**Satisfiable**" if formula can be *true* for some variables assignment.

The satisfiability problem (SAT)

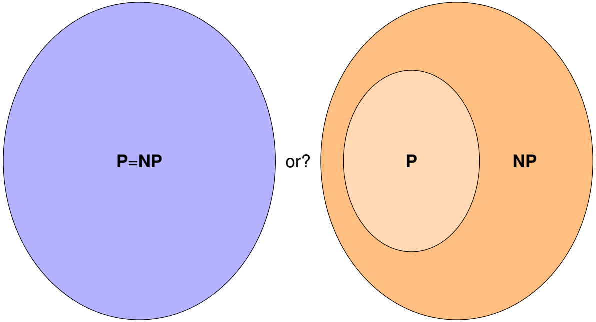
$$SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$$

Link between *SAT* and the "P vs NP" question

Theorem (Cook 1971)

$$\mathit{SAT} \in \mathbf{P} \iff \mathbf{P} = \mathbf{NP}$$

→ if we can decide *SAT* efficiently then we can also efficiently decide any **NP** problem.



History of *SAT*

- **Stephen Cook (1971)**
  Any problem in **NP** is transformable to *SAT* in polynomial time.
  Efficient solution to *SAT* $\implies$ Efficient solution to every problem in **NP**.
- **Richard Karp (1972)**
  List of 21 problems all transformable into each other in polynomial time.
- **Garey and Johnson (1979)**
  Book *"Computers and Intractability: A Guide to the theory of NP-Completeness"* lists 320 problems, all transformable into each other in polynomial time.

- These "**NP-complete**" problems are the "hardest in **NP**."
- If any **NP-complete** problem is not in **P** then all of them are not in **P**.
  ($\implies$ **P** $\neq$ **NP**).

Reductions

Idea: Transform a given problem A to another A', such that an algorithm for A' could be used as a **subroutine** to solve A.

Example

Let $S = \{x_1, \ldots, x_n\}$ be a set of integers.

A: Partition Problem (PP)

Can S be partitioned into two subsets with the same sum?

A': Subset-Sum Problem (SSP)

Can a subset of S sum to a given target t?

Given a set S for **PP**, we can transform it into an **SSP** instance as follows:

- Calculate $t = (x_1 + \cdots + x_n)/2$.
- The **SSP** instance is $\langle S, t \rangle$.

Solving **PP** has been **reduced** to solving **SSP**.

Computable functions

We need the reduction to be "efficient."

Polytime computable functions

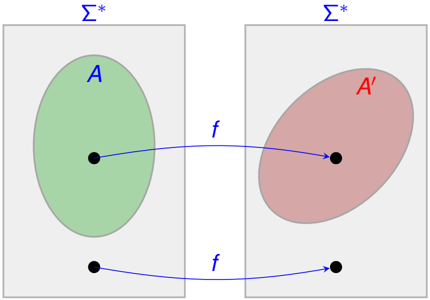
A function $f\colon \Sigma^* \to \Sigma^*$ is a polytime **computable function** if some polytime TM exists that, on input w, halts with just $f(w)$ on its tape.

The function f "efficiently transforms" the encodings of the two problems.

Polytime reducibility

A language A is polytime **reducible** to a language A' if a polytime computable function $f\colon \Sigma^* \to \Sigma^*$ exists such that

$$w \in A \iff f(w) \in A' \quad \text{for all } w \in \Sigma^*$$



Implications

We write $A \ \leq_P \ A'$ and read it: "A **is (polytime) reducible to** A'."

This means that if A' is known to have a polytime solution then we can construct a polytime solution to A too. So

$$(A \leq_P A' \ \text{ and } \ A' \in \mathbf{P}) \implies A \in \mathbf{P}$$

In other words, if A can be reduced to an "easy" problem A' then A is also "easy."

NP-Completeness and NP-Hardness

NP-Hardness

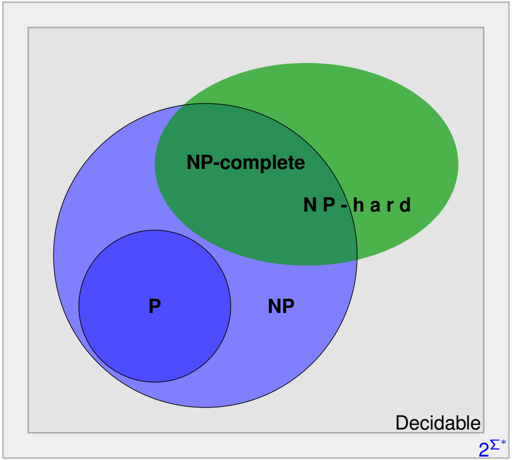
A language is **NP-hard** if every problem in **NP** is polytime reducible to it.

NP-Completeness

A language is **NP-complete** if it satisfies two conditions:

1. it is in **NP**,
2. it is **NP-hard**.

The word “**complete**” is used to mean that a solution to any such problem can be applied to all others in the class.



Examples of **NP-complete** problems

The Cook-Levin Theorem

SAT is **NP-complete.**

- **Constraint Satisfaction**: SAT, 3SAT
- **Numerical Problems**: Subset Sum, Max Cut
- **Sequencing**: Hamilton Circuit, Sequencing
- **Partitioning**: 3D-Matching, Exact Cover
- **Covering**: Set Cover, Vertex Cover, Feedback Set, Clique Cover, Chromatic Number, Hitting Set
- **Packing**: Set Packing

How do we show a problem is in **NP**?

How do we show a problem is in **NP**?

1. Define a **certificate** and the **checking** procedure for it.
2. Define the **size of the input instance** in terms of natural parameters.
3. Analyze the **running time** of the checking procedure.
4. Verify that this time is **polynomial** in the input size.

Example (SSP is in NP – Proof using a verifier)

On input $\langle\langle S, t\rangle, c\rangle$ where c is a subset of S:

- Test whether c is a collection of numbers that sum to t
- Test whether S contains all the numbers in c
- If both pass, accept; otherwise, reject

Example (SSP is in NP – Proof using nondeterminism)

On input $\langle S, t\rangle$:

- Non-deterministically select a subset c of the numbers in S
- Test whether c is a collection of numbers that sum to t
- If test passes, accept; otherwise, reject

How do we show a problem A is **NP-complete**?

1/2 Prove that A is in **NP**.

2/2 Reduce a known **NP-complete** problem C to A, i.e. $C \leq_p A$:

- Define a polytime reduction.
  (How an instance of C is mapped to an instance of A in polynomial time.)
- (1/2) Prove that the reduction maps yes-instances of C to yes-instances of A.
- (2/2) Prove that the reduction maps yes-instances of A to yes-instances of C.

For **NP-hardness** we only need step **2/2**.

Example

The DOUBLE-SAT problem

$$\text{DOUBLE-SAT} = \{\langle\phi\rangle \mid \phi \text{ has at least two satisfying assignments}\}$$

1/2 Show that DOUBLE-SAT $\in$ **NP**:

On a Boolean input formula $\phi(x_1, \ldots, x_n)$, check the certificate is **two different variable assignments**, and verify that both satisfy ϕ.

2/2 Show that SAT $\leq_P$ DOUBLE-SAT:

On input $\phi(x_1, \ldots, x_n)$:

- Introduce a new Boolean variable y.
- Output formula:

$$\psi(x_1, \ldots, x_n, y) = \phi(x_1, \ldots, x_n) \wedge (y \vee \bar{y}).$$

- (1/2) If $\langle\phi(x_1, \ldots, x_n)\rangle \in$ SAT then ϕ has at least one satisfying assignment, and therefore $\psi(x_1, \ldots, x_n, y)$ has at least two satisfying assignments as we can satisfy the new clause $(y \vee \bar{y})$ by assigning either *true* or *false* to y, so $\langle\psi(x_1, \ldots, x_n, y)\rangle \in$ DOUBLE-SAT.
- (2/2) If $\langle\psi(x_1, \ldots, x_n, y)\rangle \in$ DOUBLE-SAT, then both $\phi(x_1, \ldots, x_n)$ and $(y \vee \bar{y})$ have to be satisfiable, so in particular $\langle\phi(x_1, \ldots, x_n)\rangle \in$ SAT.

Therefore, SAT $\leq_P$ DOUBLE-SAT, and hence DOUBLE-SAT is **NP-complete.**

Optimization problems

A decision problem has a *true* or *false* answer, whereas an optimization problem involves maximizing or minimizing a function of several parameters.

Optimization Problems

Maximize or minimize a function of the input variables.

- **NP** and **NP-complete** only apply to **decision problems**.
- Optimization version of a **NP-complete** problem is at least as hard.
- It is **NP-hard** (**NP-hard** problems do not need to be decision problems).

Useful strategies for tackling **NP-hard** problems

- Is it a tractable **special case** which can be solved quickly?
- Is a probabilistic approach or an approximation acceptable?
  Try **(meta-)heuristics** (fast, but not always correct).
- Try exponential or sub-exponential time algorithms that are better than exhaustive search.
  For example, Dynamic Programming if possible.