Analysis and Design of Algorithms

National University of Singapore

CS3230

Week 11 (Part-1)

NP-Completeness

Diptarka Chakraborty

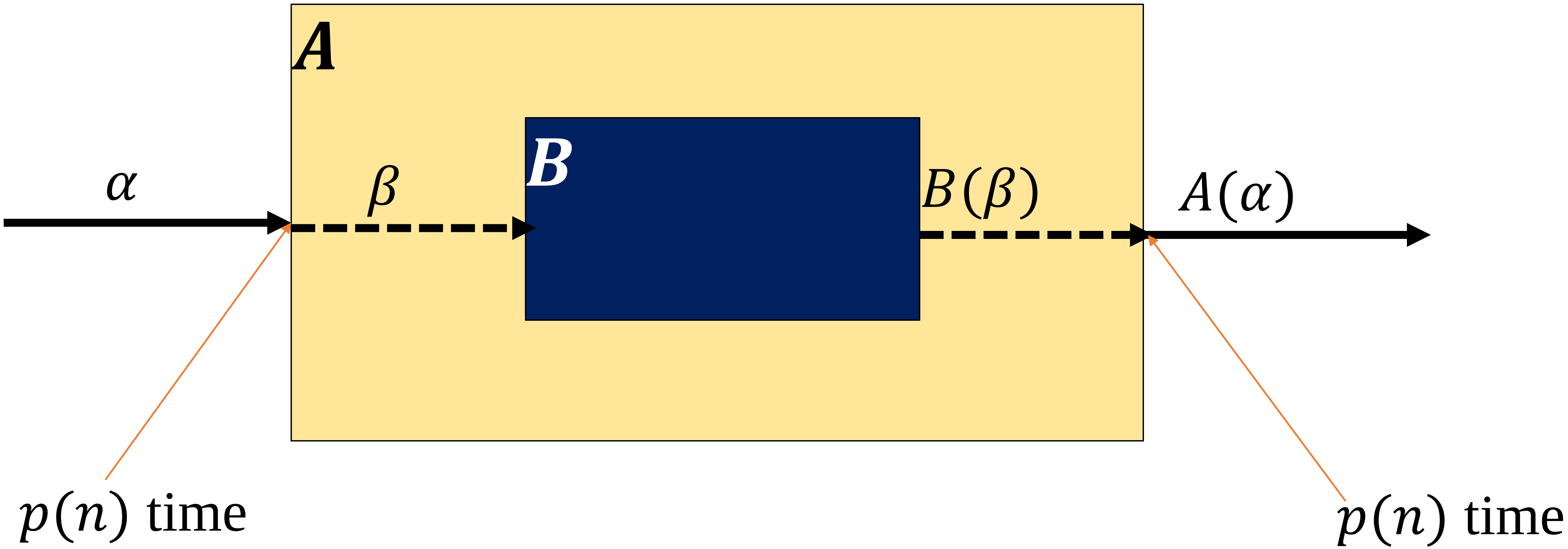
Ken Sung

Recap

- Reductions are a basic tool in algorithm design: using an algorithm for one problem to solve another.

- If you have a poly time reduction from A to B and you also have a poly time algorithm for B, then you get a poly time algorithm for A.

$\boldsymbol{p(n)}$-time Reduction



Poly-time Reduction

$$A \leq_P B$$

If B has a polynomial time algorithm, then so does A!

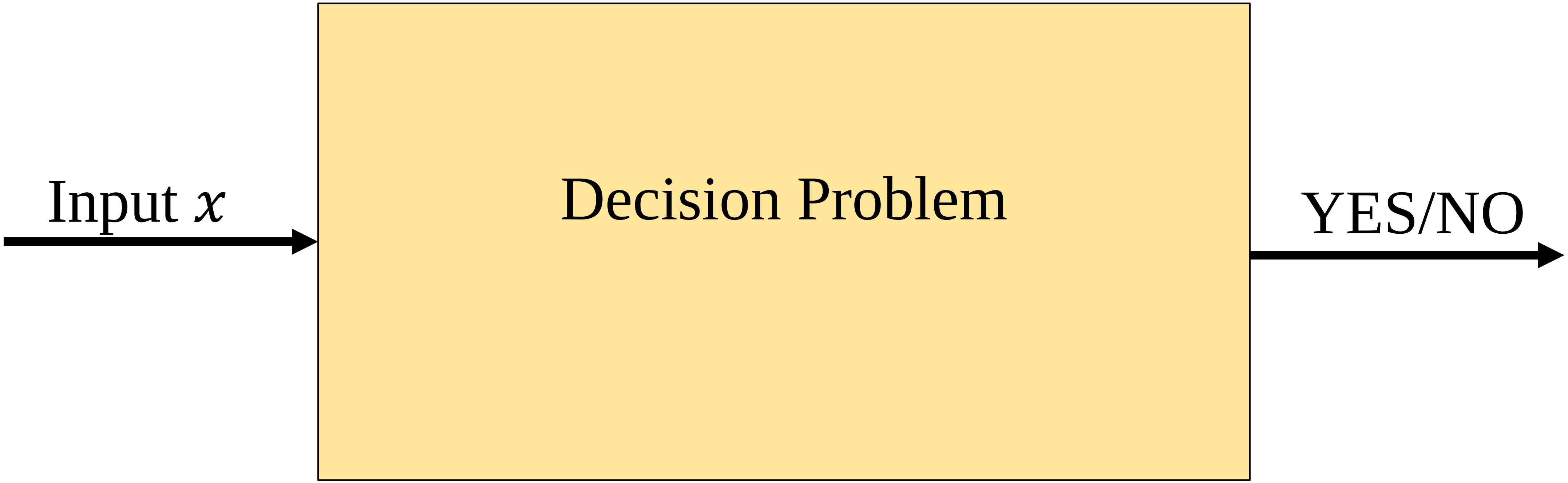
Poly-time Reduction

$$A \leq_P B$$

If A is "hard", then so is B!

Decision Problems

A **decision problem** is a function that maps an instance space I to the solution set $\{\mathrm{YES}, \mathrm{NO}\}$.



Reductions between Decision Problems

Given two decision problems A and B, a **polynomial time reduction** from A to B, denoted $A \leq_P B$, is a transformation from instances α of A to instances β of B such that:

1. α is a YES-instance for A **if and only if** β is a YES-instance for B.
2. The transformation takes polynomial time in the size of α.

Confusion about Running Time

- We should always count the running time in terms of the number of bits in the input.

- Strictly speaking, we should always let $\boldsymbol{n}$ be the input length in terms of number of bits.

- In algorithm design we generally consider word-RAM model. So input is stored in an array of words, and each arithmetic or logical operation (+, -, *, /, OR, AND, NOT) involving a **constant number** of words takes **constant number of cycles (time).** We count only number of **instructions** as running time.

NP
A class of problems

and how it came into existence

How does any scientific theory/definition get developed ?

- Unexplained facts in a field of science
- Persistent search for the truth
- A collective effort for many years or decades

Similar is the history behind the development of the class **NP**.

Go back to 1960's

Efficient algorithm was found	**No Efficient algorit till date**
Shortest Path	Longest Path
Minimum spanning Tree	Travelling salesman Pro
Euler tour	Hamiltonian cycle
Min Cut	Balanced Cut
Independent Set on trees	Independent Set
Bipartite matching	3D matching
Linear Programming	Integer Programming
⋮	⋮

It was quite surprising and even frustrating to be unable to find efficient algorithm for so many problems when their similar looking versions had very efficient algorithms.

- **Longest path problem**

Decision version: Given a graph $\boldsymbol{G}$, does there exist a path of length at least $\boldsymbol{k}$.

Searching for a path of length at least $\boldsymbol{k}$ appears to be difficult.

But what about **verifying** whether a given path is of length at least $\boldsymbol{k}$?

It is quite easy ☺.

- **Vertex cover**

Decision version: Given a graph $\boldsymbol{G}$, does there exist a vertex cover of size $\leq \boldsymbol{k}$.

Searching for a subset of $\boldsymbol{k}$ vertices that is a vertex cover of $\boldsymbol{G}$ appears difficult.

But what about **verifying** whether a given subset of $\boldsymbol{k}$ vertices is a vertex cover ?

It is quite easy ☺.

No Efficient algorithm		
Longest Path	short certificate	**Search:** difficult **Verification:** easy
Travelling salesman Problem		
Balanced Cut		
Hamiltonian cycle		
Independent Set		
3D matching		
Integer Programming		
⋮		

NP class

Yes instance

No instance

X : any decision problem

I : any (input) instance of X

How to capture the fact that A is efficient

certifier for X :

algorithm A with output {yes,no}

- **Input** : (I, s)

Proposed solution

- **Behavior**: A can <u>verify</u> if proposed solution s is right or wrong.

NP class

Yes instance

No instance

X : any decision problem

I : any (input) instance of X

How to capture the fact that s is short

Efficient certifier for X :

A polynomial time algorithm A with output {yes,no}

- **Input** : (I, s)

Proposed solution

- **Behavior**: There is a polynomial function p such that I is yes-instance of X **if and only if** there exists a string s with $|s| \leq p(|I|)$ such that A outputs yes on input (I, s).

NP class

Definition (**NP**):

The set of all decision problems which have **efficient certifier**.

NP : “Non-deterministic polynomial time”

Example: HAM-CYCLE

<u>Recall</u>: In Ham-Cycle, given a graph G, problem is to decide whether there is a simple cycle that visits each vertex exactly once.

Certificate is the cycle itself. Verifier checks in polynomial time whether it is a cycle and visits each vertex once.

Hence, HAM-CYCLE is in NP.

NP class

Definition (**NP**):

The set of all decision problems which have **efficient certifier**.

NP : “Non-deterministic polynomial time”

Definition (**P**):

The set of all decision problems which have **efficient** (poly-time) algorithm.

Is there any Relation between **P** and **NP** ?

NP class

Yes instance

No instance

X : any decision problem in **P**

I : any (input) instance of X

Let Q be a polynomial time algorithm for solving X.

Efficient certifier for X :

A polynomial time algorithm A with output {yes,no}

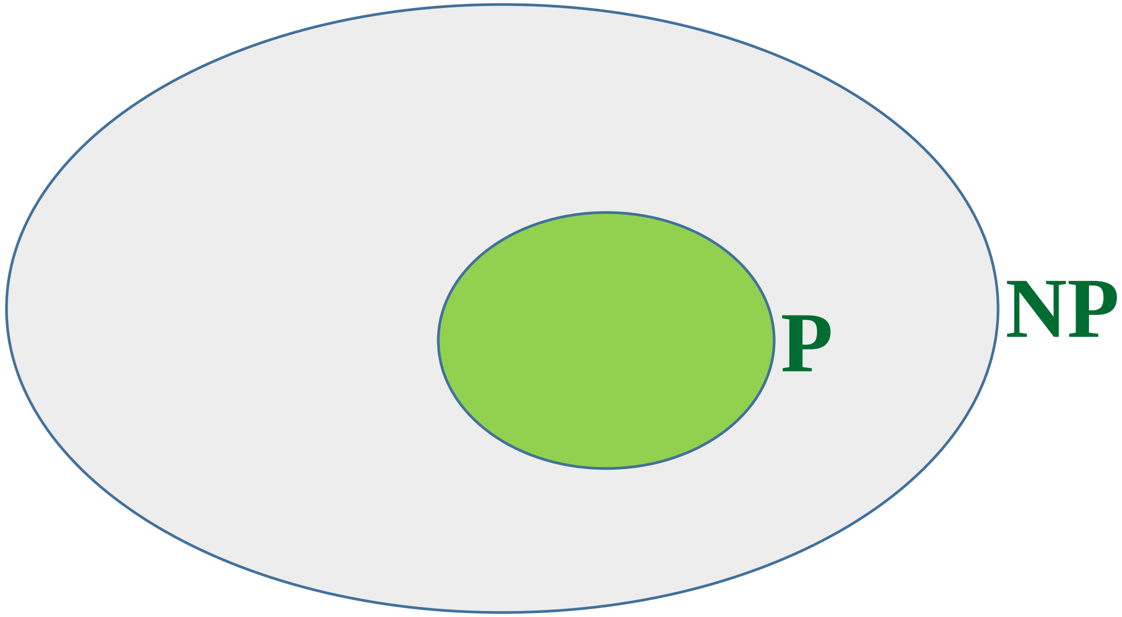
- **Input** : (I, s)

Proposed solution

- **Behavior**: On getting input (I, s), just ignore s, and execute the algorithm Q on input I. If the answer is yes, output yes; if the answer is no, output no.

NP versus P

Is P = NP ?



Verifying a <u>proposed solution</u> versus **finding a <u>solution</u>**

NP Complete
A class of problems

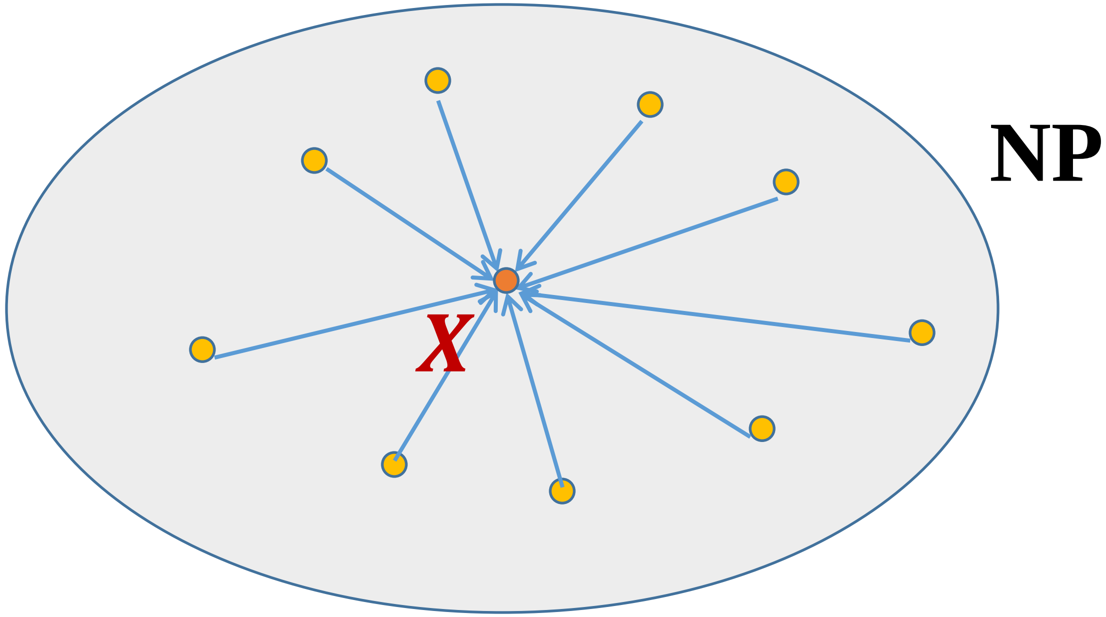
and how it came into existence

NP-complete

If $\boldsymbol{X}$ is not known to be in **NP,** then we say $\boldsymbol{X}$ is just **NP-hard**

- A problem $\boldsymbol{X}$ in **NP** class is **NP-complete** if for every $\boldsymbol{A} \in$ **NP**

$$\boldsymbol{A} \leq_{\boldsymbol{P}} \boldsymbol{X}$$



Does any NP-complete problem exist ?

It really needs

- courage to ask such a question and
- great insight to pursue its answer

Because:

- Every problem, known as well unknown, from class **NP** has be reducible to this problem.
- Such a problem would indeed be the hardest of all problems in **NP**.

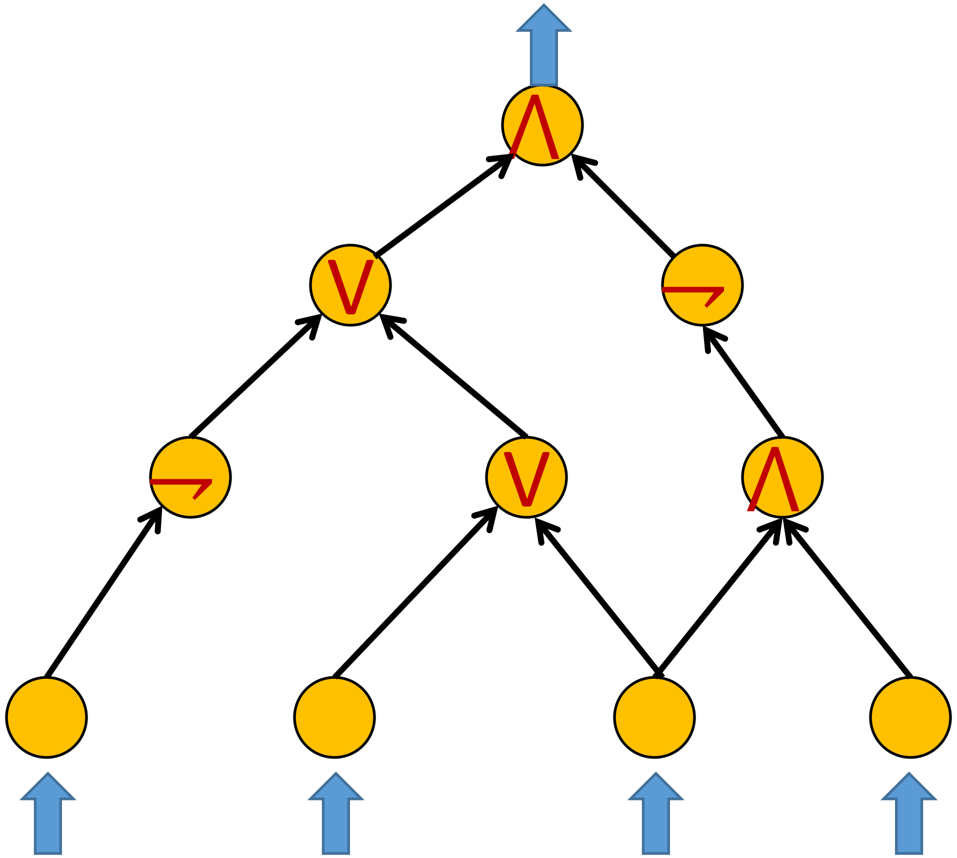
But only such great questions in science lead to great inventions.

Does any NP-complete problem exist ?

Circuit satisfiability problem:
[Cook and Levin , 1971]

A DAG with nodes corresponding to **AND**,**NOT**,**OR** gates and n binary inputs,

does there exist any binary input which gives output 1 ?



Why is in **NP**?

Certificate is an binary input that gives output 1

This slide is optional
(meant for the student whose aim is beyond just a good grade)

Question: How can every problem from NP be reduced to **circuit satisfiability** ?

Answer:

Consider any problem $\boldsymbol{X} \in$ **NP**.

What we know is that it has an efficient certifier, say $\boldsymbol{Q}$.

Any algorithm which outputs yes/no can be represented as a DAG

- Where internal nodes are gates.
- Leaves are binary inputs
- Output is 1/0.

So Cook & Levin essentially <u>transform</u> $\boldsymbol{Q}$ into the corresponding DAG. And thus <u>simulates</u> $\boldsymbol{Q}$ on the proposed solution.

[This is just a sketch. Interested students should study it sometime in future.]

Satisfiability (CNF-SAT)

- **Literal**: A Boolean variable or its negation. $\boldsymbol{x_i}, \boldsymbol{\overline{x_i}}$

- **Clause**: A disjunction (OR) of literals. $\boldsymbol{C_j = x_1 \lor \overline{x_2} \lor x_3}$

- **Conjunctive Normal Form (CNF)**: a formula Φ that is a conjunction (AND) of clauses $\boldsymbol{\Phi = C_1 \land C_2 \land C_3 \land C_4}$

- **CNF-SAT**: Given a CNF formula Φ, does it have a satisfying truth assignment?

3-SAT

SAT where **each clause contains exactly 3 literals** corresponding to different variables.

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

Satisfying assignment: $x_1 = \text{True}, x_2 = \text{True}, x_3 = \text{False}, x_4 = \text{True}$

Unsatisfying assignment: $x_1 = \text{True}, x_2 = \text{False}, x_3 = \text{False}, x_4 = \text{False}$

Circuit Satisfiability $\leq_P$ CNF-SAT $\leq_P$ 3-SAT

So **3-SAT** is **NP**-complete

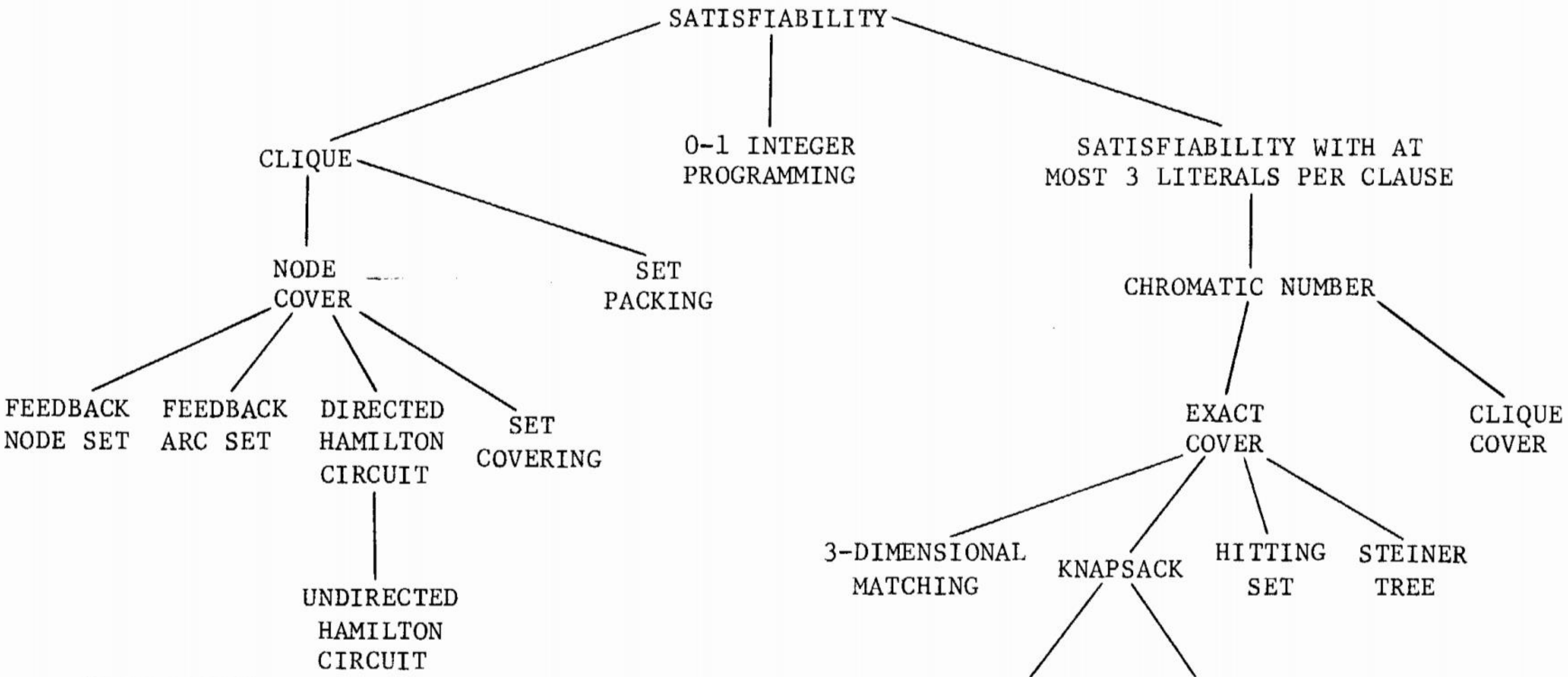
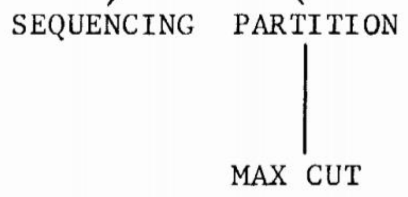
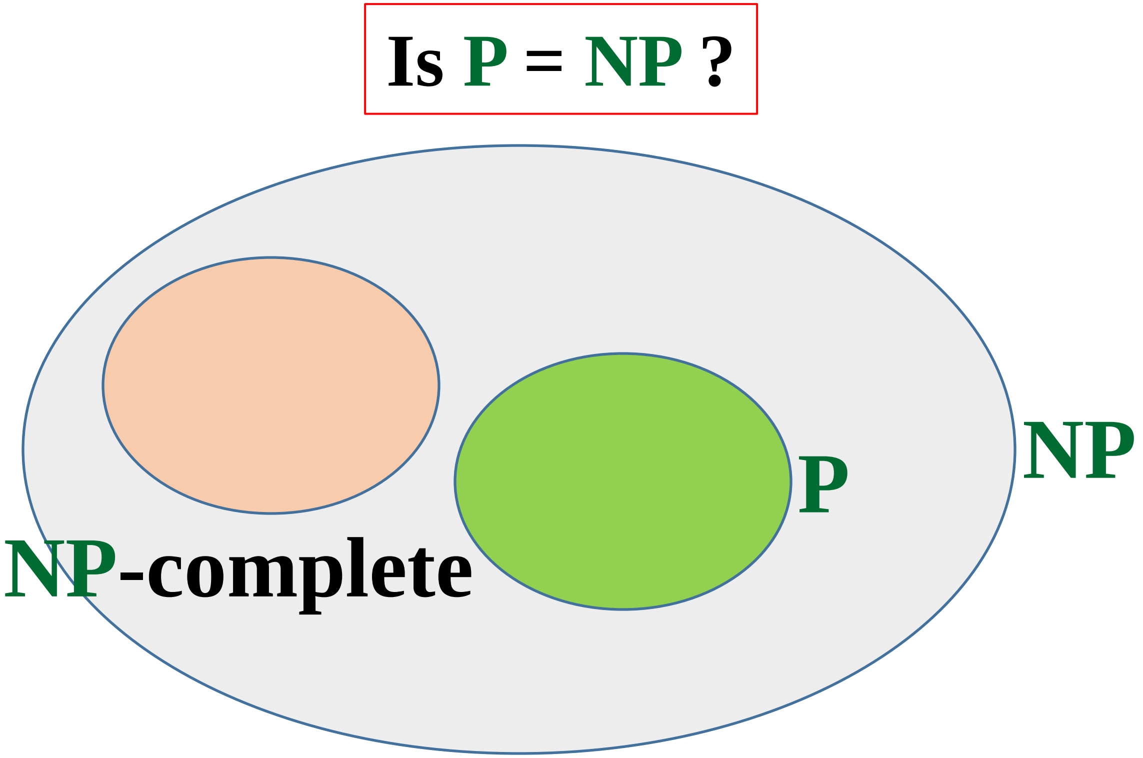



FIGURE 1 - Complete Problems

Dick Karp (1972)
1985 Turing Award

NP versus P

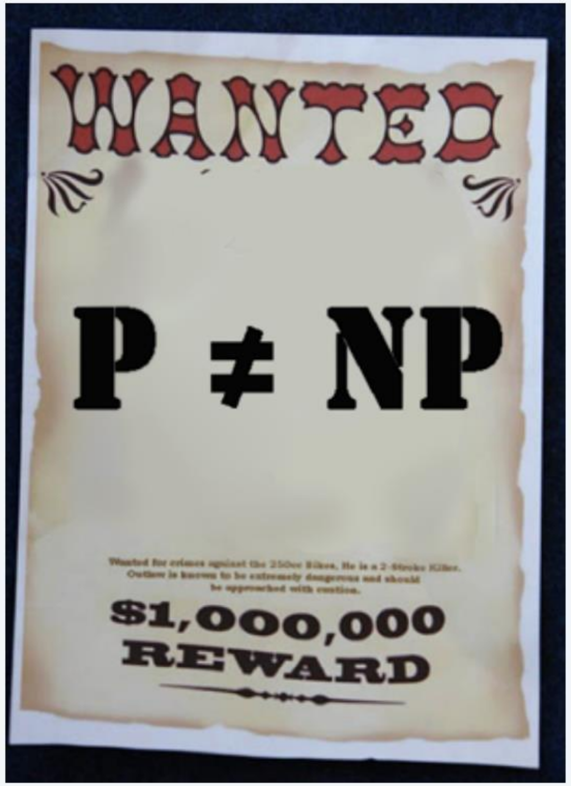


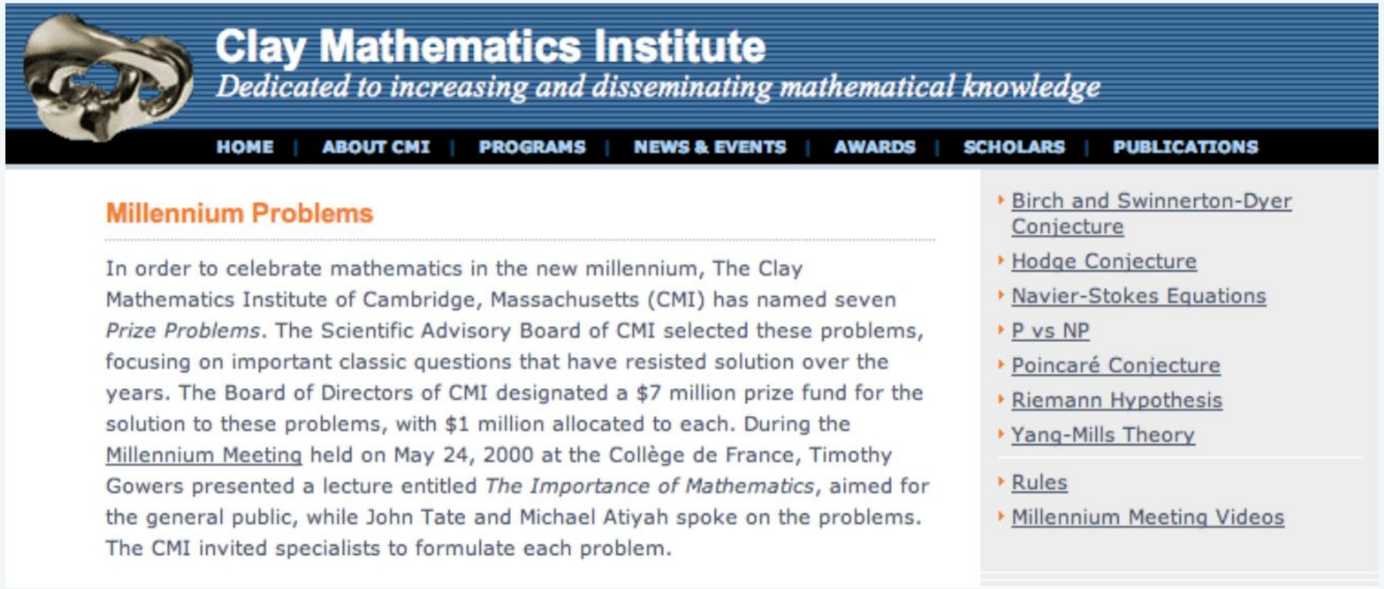
If any **NP**-complete problem is solved in polynomial time

➔ P = NP

Millennium Prize

$1 million dollars for resolution of **P=NP** or **P≠NP**





Some Quotes

"I conjecture that there is no good algorithm for the traveling salesman problem. My reasons are the same as for any mathematical conjecture: (i) It is a legitimate mathematical possibility and (ii) I do not know." — **Jack Edmonds (1966)**

"If I had to bet now, I would bet that P is not equal to NP. I estimate the half-life of this problem at 25–50 more years, but I wouldn't bet on it being solved before 2100. " — **Robert Tarjan (2002)**

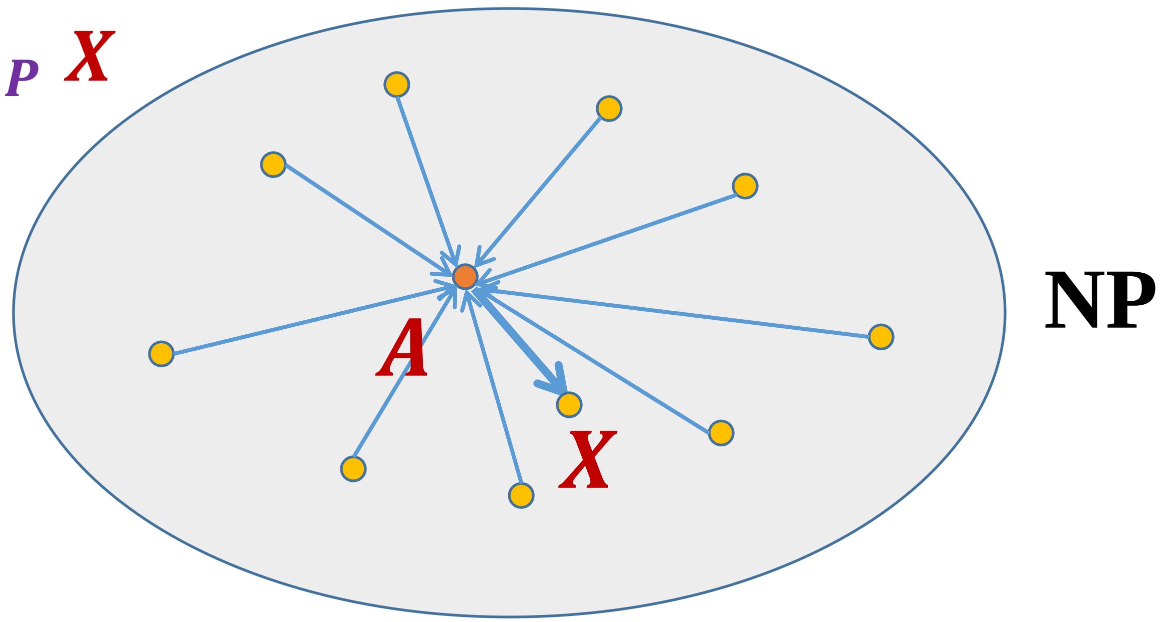
Some Quotes

“ I think that in this respect I am on the loony fringe of the mathematical community: I think (not too strongly!) that P=NP and this will be proved within twenty years. Some years ago, Charles Read and I worked on it quite bit, and we even had a celebratory dinner in a good restaurant before we found an absolutely fatal mistake. ” — **Béla Bollobás (2002)**

How to show a problem to be NP-complete ?

Let X be a problem which we wish to show to be **NP**-complete

1. Show that $X \in$ **NP**
2. Pick a problem A which is already known to be **NP**-complete
3. Show that $A \leq_P X$



3-SAT

SAT where **each clause contains exactly 3 literals** corresponding to different variables.

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

Satisfying assignment: $x_1 = \mathrm{True}, x_2 = \mathrm{True}, x_3 = \mathrm{False}, x_4 = \mathrm{True}$

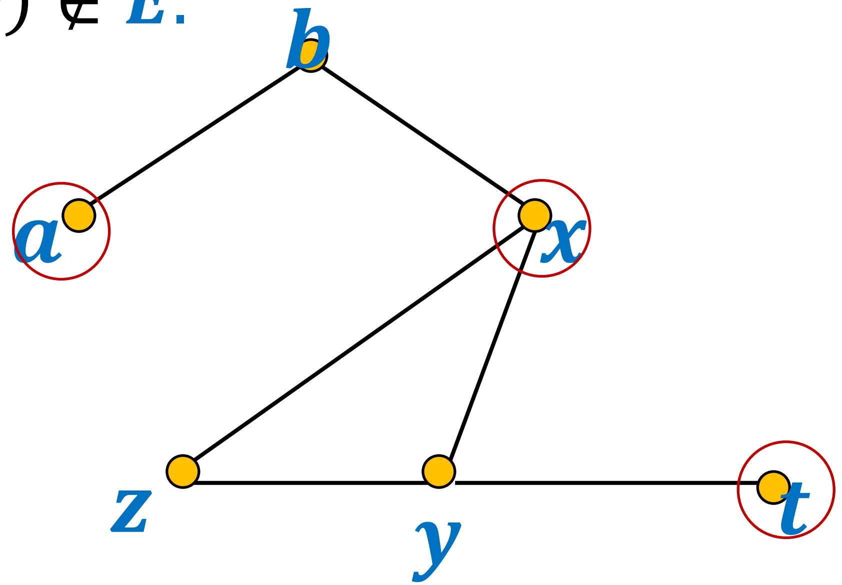
Unsatisfying assignment: $x_1 = \mathrm{True}, x_2 = \mathrm{False}, x_3 = \mathrm{False}, x_4 = \mathrm{False}$

INDEPENDENT-SET

Definition: Given an undirected graph $G = (V, E)$,

a subset $X \subseteq V$ is said to be an **independent** set if

For each $u, v \in X$, $(u, v) \notin E$.



Optimization version: compute Independent set of <u>Largest</u> size.

Decision version: Does there exist an independent set of size >k ?

3-SAT $\leq_P$ INDEPENDENT-SET

Simple Exercise:
To show in **NP**

Given an instance Φ of 3-SAT, goal is to construct an instance (G, k) of INDEPENDENT-SET so that G has an independent set of size k **if and only if** Φ is satisfiable.

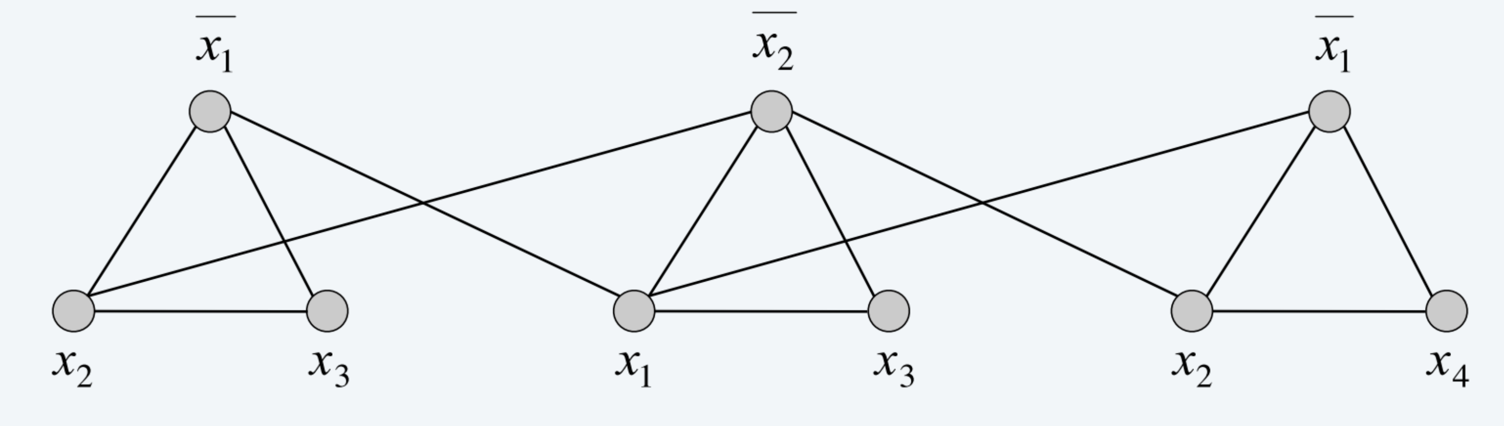
To think: why it suffices to show for exactly k, not >k

3-SAT $\leq_P$ INDEPENDENT-SET

Given an instance Φ of 3-SAT, goal is to construct an instance (G, k) of INDEPENDENT-SET so that G has an independent set of size k **if and only if** Φ is satisfiable.

Reduction

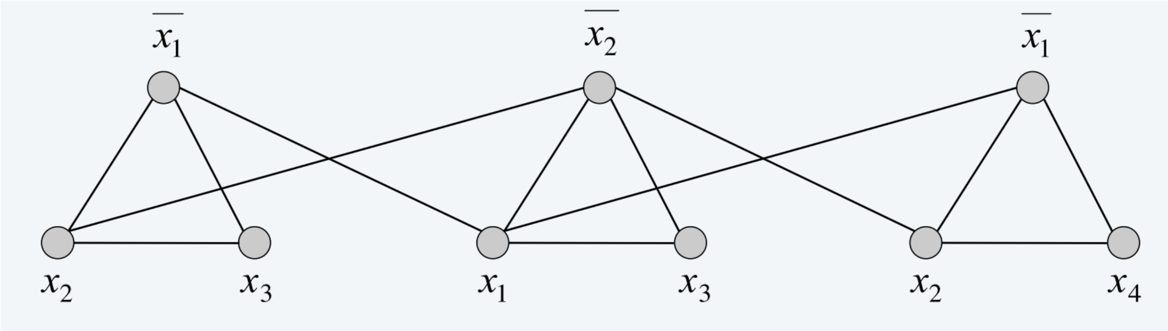
- G contains 3 vertices for each clause, one for each literal
- Connect 3 literals in clause in a triangle
- Connect literal to each of its negations
- Set k = number of clauses



$$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

3-SAT $\leq_P$ Independent-Set

Reduction
- G contains 3 vertices for each clause, one for each literal
- Connect 3 literals in clause in a triangle
- Connect literal to each of its negations
- Set k = number of clauses



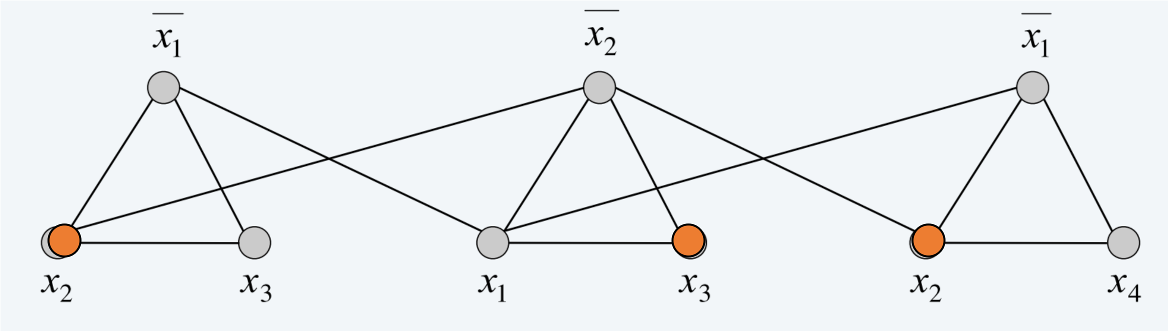
$$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

Reduction clearly runs in linear time.

3-SAT $\leq_P$ INDEPENDENT-SET

Reduction

- G contains 3 vertices for each clause, one for each literal
- Connect 3 literals in clause in a triangle
- Connect literal to each of its negations
- Set k = number of clauses



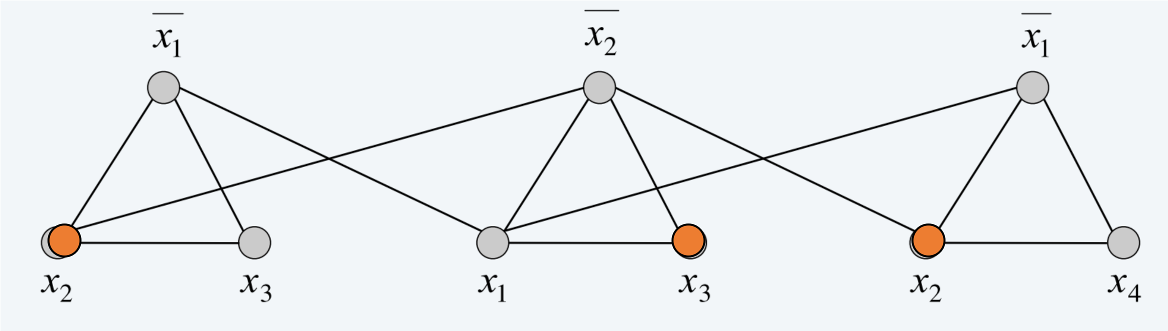
$$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

Suppose Φ is a YES-instance. Take any satisfying assignment for Φ and select a true literal from each clause. Corresponding k vertices form an independent set in G.

3-SAT $\leq_P$ INDEPENDENT-SET

Reduction

- G contains 3 vertices for each clause, one for each literal
- Connect 3 literals in clause in a triangle
- Connect literal to each of its negations
- Set k = number of clauses



$$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

Suppose (G, k) is a YES-instance. Let S be the independent set of size k. Each of the k triangles must contain exactly one vertex in S. Set these literals to true, so all clauses satisfied.

Worst Case Analysis

- Proof shows that **some** instances of INDEPENDENT-SET are as hard to solve as the 3-SAT problem. This does **not** mean that all instances of the INDEPENDENT-SET problem are hard!

- So, if there is no poly time algorithm that solves 3-SAT on *all* instances, there is no poly time algorithm that solves INDEPENDENT-SET on *all* instances.

Status of SAT

- Fastest algorithm known for 3-SAT runs in time $\approx 1.308^n$. It is believed that there is no $2^{o(n)}$-time algorithm for 3-SAT (**Exponential Time Hypothesis**).

- Often very convenient to reduce from 3-SAT to other problems, showing that those will also be hard if 3-SAT is hard.

Question 1

Suppose 3-SAT $\leq_P$ A for some decision problem A. Assume the exponential time hypothesis that there is no $2^{o(n)}$-time algorithm for 3-SAT. Then, there is no $2^{o(n)}$-time algorithm for A.

- True
- False

Question 1: Solution

False.

If the reduction runs in time n^c, then a $2^{o\left(n^{1/c}\right)}$-time algorithm for A implies a $2^{o(n)}$-time algorithm for 3-SAT. So, by the assumption, there are no $2^{o\left(n^{1/c}\right)}$-time algorithms for A. The lower bound for A depends on the running time of the reduction.

Extent and Impact

- Garey and Johnson's book, "Computers and Intractability", includes over 300 **NP-complete** problems and is the #1 cited reference in computer science!

- NP-completeness is used in more than 6,000 publications per year (more than "compiler", "OS", "database").

- Main intellectual export of computer science.

More…

- There are problems that are provably harder than **NP-complete** problems, problems that require polynomial space, problems that require large circuits, problems that are unsolvable even with unlimited time!

- Enter the world of complexity theory…

Attend **Computational Complexity** course!!!

Fine-grained Hardness : An emerging field

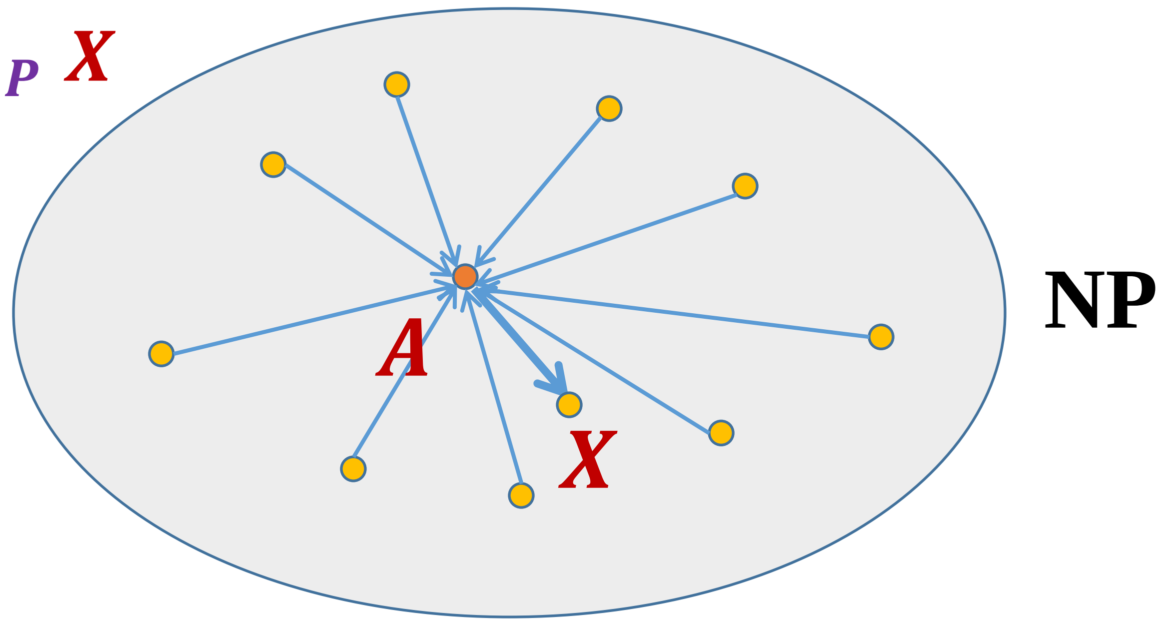
- Prove computational hardness using conjectures like…
- There is no $2^{o(n)}$-time algorithm for 3-SAT (**Exponential Time Hypothesis**), or even stronger (**Strong Exponential Time Hypothesis**).
- We can prove more fine-grained hardness like
- LCS can not be solved in time $O(n^{2-\varepsilon})$ for any $\varepsilon > 0$
- And many more…

Attend **Advanced Algorithms** course in the next semester!!!

How to show a problem to be NP-complete ?

Let X be a problem which we wish to show to be **NP**-complete

1. Show that $X \in$ **NP**
2. Pick a problem A which is already known to be **NP**-complete
3. Show that $A \leq_P X$



Acknowledgement

- The slides are modified from
  - The slides from Prof. Kevin Wayne
  - The slides from Prof. Surender Baswana
  - The slides from Prof. Erik D. Demaine and Prof. Charles E. Leiserson
  - The slides from Prof. Arnab Bhattacharya and Prof. Wing-Kin Sung