



Lecture 15: NP Completeness

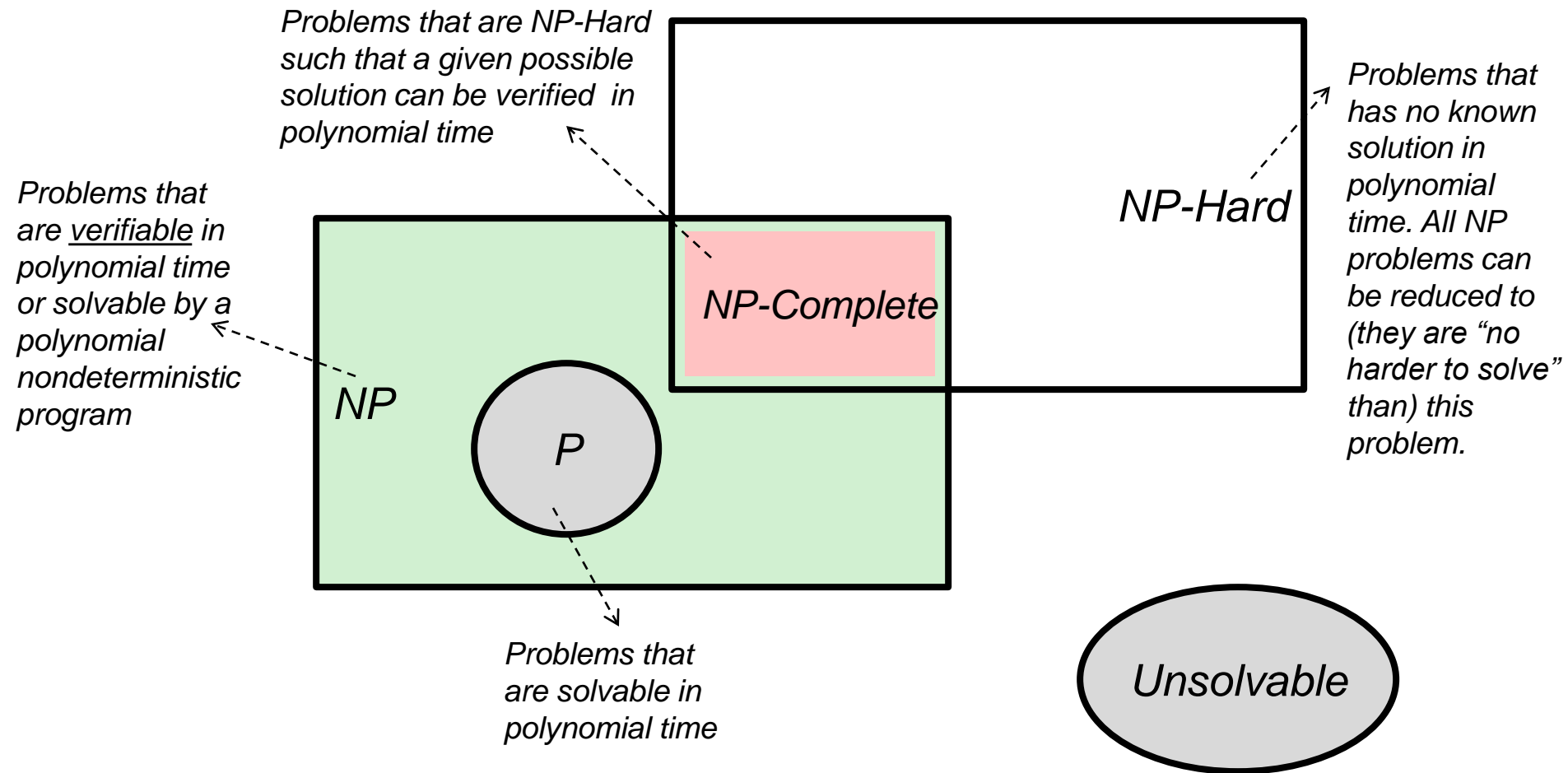
Dr. Khalil Ahmad Yousef

*Reading Assignment:
Read Chapter 34 of the book*

Course Learning Outcomes

- **Describe the characteristics of the major complexity classes (P class, NP class)**

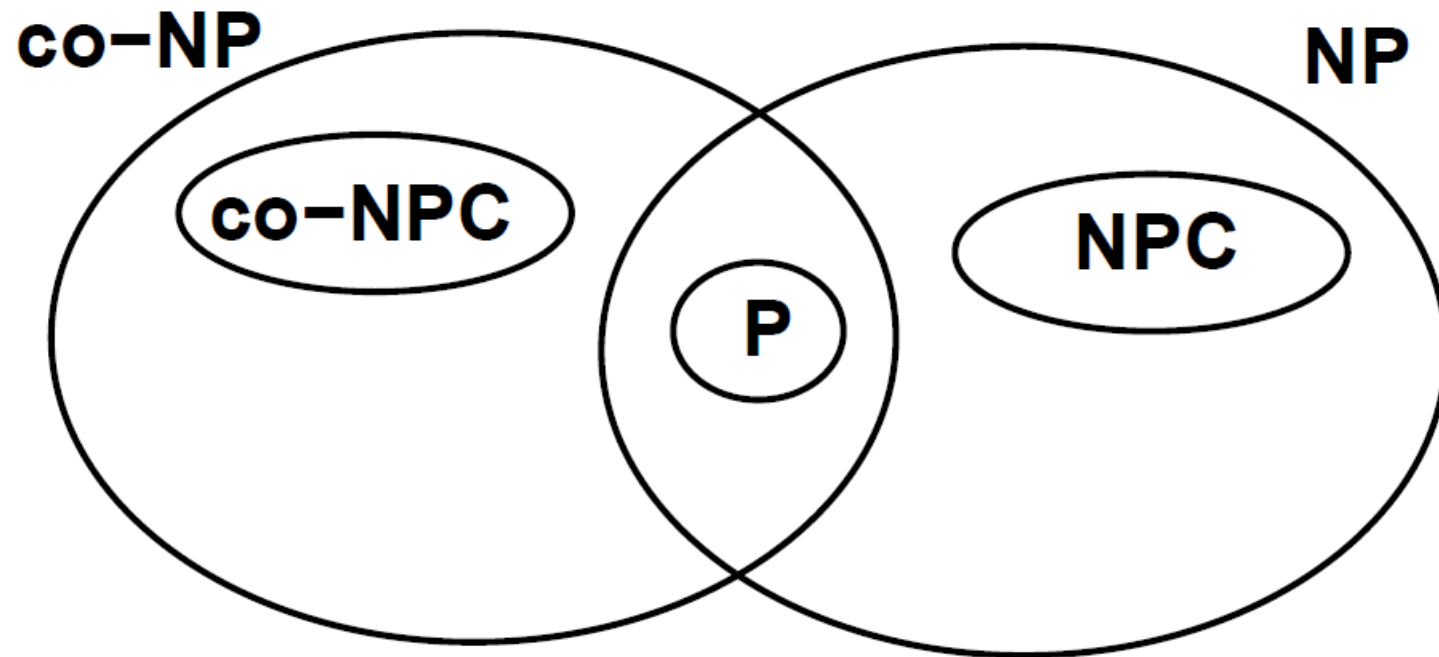
Big Picture



Terminology (informal)

- A problem belongs to the class P (polynomial) if it can be “solved” in polynomial time.
- A problem belongs to the class NP (nondeterministic polynomial) if a given solution can be “verified” in polynomial time.
- A problem is NP-hard if all NP problems can be reduced to (they are “no harder to solve” than) this problem. \bar{L}
- A problem is NP-complete if it is NP-hard and belongs to the class NP.
- The complexity class of co-NP is defined as the set of languages L such that $\bar{L} \in NP$
- It is not certain that NP is closed under complementation, that is it is unknown whether $NP = co-NP$.
 - What is known is that $P \subseteq (NP \cup co-NP)$, since P is closed under complementation.

P, NP, and Co-NP



Tractability

- Some problems are *intractable* (hard): as they grow large, we are unable to solve them in *reasonable time*.
- What constitutes *reasonable time*?
 - Standard working definition: *polynomial time* (tractable or easy)
 - On an input of size n the worst-case running time is $O(n^k)$ for some constant k
 - Where n is a suitable measure of the size of the input.
 - Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
 - Not in polynomial time: $O(2^n)$, $O(n^n)$, $O(n!)$

Tractability

- Problems that are solvable in polynomial time are said to be **tractable** or **easy**
- Problems that require super-polynomial time are said to be **intractable** or **hard**

Polynomial and Exponential Time Algorithms

A **polynomial time algorithm** is defined to be one whose time complexity is $O(p(n))$ for some polynomial function p , where n is input length. An **exponential time algorithm** cannot be so bounded. The distinction between these two types of algorithms can be illustrated by the following growth rate table (given a unit of measure in microseconds (10^{-6}) and that s stands for second, m for minute, d for day, y for year, and c for century):

Algorithm Complexity	Size of n					
	10	20	30	40	50	60
n	.00001 s	.00002 s	.00003 s	.00004 s	.00005 s	.00006 s
n^2	.0001 s	.0004 s	.0009 s	.0016 s	.0025 s	.0036 s
n^3	.001 s	.008 s	.027 s	.064 s	.125 s	.216 s
n^5	.1 s	3.2 s	24.3 s	1.7 m	5.2 m	13 m
2^n	.001 s	1.0 s	17.9 m	12.7 d	35.7 y	366 c
3^n	.059 s	58 m	6.5 y	3855 c	2×10^8 c	1.3×10^{13} c

Polynomial and Exponential Time Algorithms

Even with improvements in computer technology, it is difficult to improve matters for exponential time algorithms. The following table shows the size of the largest problem that is solvable in one hour as technology improves.

Time	Present Computer	100 times faster	1000 times faster
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_4$	$3.98N_4$
2^n	N_5	$6.64 + N_5$	$9.97 + N_5$
3^n	N_6	$4.19 + N_6$	$6.29 + N_6$

These tables indicate some of the reasons that polynomial time algorithms are regarded as tractable; whereas, exponential time algorithms are intractable. We shall refer to a problem as being intractable if it is so hard that no polynomial time algorithm can solve it.

Tractability: More detailed View

- Some problems are *undecidable*: no computer can solve them
 - E.g., Turing's "Halting Problem"
 - A problem that is not solvable by any computer, no matter how much time is given or provided.
 - We don't care about such problems here;
 - Take a theory class
- Other problems are *decidable*, but *intractable*: as they grow large, we are unable to solve them in reasonable or *Polynomial* time.

Tractability - P

- Some problems are provably decidable in polynomial time on an ordinary computer (Deterministic Machine DTM)
 - We say such problems belong to the set **P**
 - Technically, a computer with unlimited memory
 - *How do we typically prove a problem $\in P$?*

Tractability - NP

- Some problems are provably decidable in polynomial time on a nondeterministic computer or machine (NDTM)
 - We say such problems belong to the set NP
 - Can think of a nondeterministic computer as a parallel machine that can freely spawn an infinite number of processes
 - *How do we typically prove a problem \in NP?*
 - *We shall come to answer this question later on.*

Tractability - **P** And **NP**

- **P** = set of problems that can be solved in polynomial time
- **NP** = set of problems for which a solution can be verified in polynomial time by a NDTM
- **P** \subseteq **NP**
 - The big question: Does **P** = **NP**?

NP-Complete Problems

- The *NP-Complete* problems are an interesting class of problems whose status is unknown
 - No polynomial-time algorithm has been discovered for an NP-Complete problem
 - No suprapolynomial lower bound has been proved for any NP-Complete problem, either

NP-Complete Problems

- We will see that NP-Complete problems are the “hardest” problems in NP:
 - If any *one* NP-Complete problem can be solved in polynomial time...
 - ...then *every* NP-Complete problem can be solved in polynomial time...
 - ...and in fact *every* problem in **NP** can be solved in polynomial time (which would show **P = NP**)
 - Thus: solve hamiltonian-cycle in $O(n^{100})$ time, you’ve proved that **P = NP**. **Retire rich & famous.**

NP-Complete Problems

- *Intuitively and informally, what does it mean for a problem to be NP-Complete?*
 1. *The problem \in NP*
 - (Will focus on this now)
 2. *All other problems in NP can be **easily** reduced to my problem*
 - Reduction Issue: (Will come to this later)

An NP-Complete Problem: Hamiltonian Cycles

- An example of an NP-Complete problem:
 - A hamiltonian cycle of a weighted undirected graph:
 - It is a simple cycle that contains every vertex exactly once except the first vertex which will be visited twice at the start and at the end of the circle.
 - The hamiltonian-cycle problem: given a graph G, **does it have a hamiltonian cycle?**
 - **Decision problem:** YES/NO answers.
 - Travel sales man problem.
 - *Describe a naive algorithm for solving the hamiltonian-cycle problem. Running time?*

Decision Problems

- A **decision problem** is a problem of YES or NO answer
- A **decision problem** is “polynomial” if it admits a polynomial solution algorithm.
- **Decision problem Specification**
 - Each decision problem can be specified by giving a set of finite strings (sequence of characters which we will refer to as a **Language**) over a finite alphabet Σ .
 - Problem Instances (Assigning values to the parameters of the problem)
 - YES instances
 - NO instances
- **The reason for restricting our attention to **decision problems** is that **decision problems** map easily into formal-language theory, which forms a mathematically precise theory of computation.**

Decision Problems

- Many problems are optimization problems, in that some value must be minimized or maximized. These problems can be recast as decision problems by imposing a bound on the value to be optimized.
 - i.e. **Optimization problems** have a corresponding **decision problems** formed by adding a threshold to each problem specification without altering the essential computation aspects of the problem.
- **Note that:**
 - If we can solve an optimization problem quickly, we can certainly solve a decision problem variant quickly too.
 - Also, if we can show evidence that the decision problem is hard, we provide evidence that the optimization problem is also hard.

Decision Problems

- **Example:** The shortest path problem, which is considered as an optimization problem, can be converted into a decision problem as follows:
 - (The YES/NO Question) Is there any shortest path shorter than K ?
 - K is a certain threshold
- **Example:** Travel Sales-Man Problem (n cities to visit)
 - Minimize the total tour time by restricting that the every city is visited exactly once except the first city, which will be visited twice.
 - **Optimization Problem:** Find the best tour or shortest tour.
 - **Decision problem:** Is there a tour of length K or less?

Decision Problems

- Key Issues:

1. We need some encoding (Instance + Question)
 - For example: to see that T.S.P. is a decision problem we need to encode the graph $G(V,E)$ into some finite string in some finite alphabet (Language)
 - Two Important notions:
 - ◆ **Problem**: general question to be answered without giving specific parameters
 - ◆ **Problem Instance**: Specifying the parameters of the problem like the weights, vertices, edges, alphabet, etc...
 - **Encoding Scheme**: given an instance I of a problem, then the encoding scheme maps I to an string(s) or language L describing the instance.
2. We need some sort of a certificate or polynomially non-deterministic (NDTM) algorithm to prove decidability (YES/NO answers) of the problem (Some sort of a GUESS function or algorithm)

Recall the Hamiltonian-Cycle Problem

- The hamiltonian-cycle problem on an undirected graph:
 1. Encoding
 - Instance: Given a weighted undirected graph $G(V,E)$
 - Question: Does it have a Hamiltonian cycle?
 2. The Certificate will be the sequence of vertices $(V_1, V_2, V_3, \dots, V_{|V|})$ such that it easy to check in polynomial time that:
 - $(V_i, V_{i+1}) \in E$ for $i=1, 2, 3, \dots, |V|-1$ and
 - $(V_{|V|}, V_1) \in E$ as well

Decision Problems

Example: Traveling Salesperson Problem (TSP)

INSTANCE: A finite set $C = \{c_1, c_2, \dots, c_m\}$ of cities and a distance $d(c_i, c_j) \in \mathbb{Z}^+$ for each city pair $c_i, c_j \in C$, and a bound $B \in \mathbb{Z}^+$.

QUESTION: Is there a tour of all cities in C having a total length of no more than B , that is, is there an ordering $\langle c_{\pi(1)}, c_{\pi(2)}, \dots, c_{\pi(m)} \rangle$ of C such that $\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(m)}, c_{\pi(1)}) \leq B$?

Decision Problems: Other Examples

- Boolean Satisfiability

- Given a Boolean formula f of n variables, Is there a way to set the variables such that to make the formula true?

- Variations:

- ◆ SAT (Disjunction of literals)

- The Boolean formula has one or more clauses of n -variables. All variables or **literals** in each clause are **ORed** together.
- It is said to satisfiable *iff* at least one of the literals is true in each clause.

- ◆ 3-SAT (Conjunction of clauses)

- We have K -clauses and each clause has exactly 3 variables or literals, but the **clauses** themselves are **ANDed** together
- It is said to satisfiable *iff* all clauses are satisfiable i.e. at least one of the 3 literals in each clause is true.

Boolean Satisfiability

Satisfiability (*SAT*):

INSTANCE: A set $U = \{u_1, u_2, \dots, u_n\}$ and a collection of clauses $C = \{c_1, c_2, \dots, c_m\}$ over U .

QUESTION: Is there a satisfying truth assignment for C ?

Example 1: $U = \{u_1, u_2\}$ and $C = \{\{u_1, \bar{u}_2\}, \{\bar{u}_1, u_2\}\}$.

Example 2: $U = \{u_1, u_2\}$ and $C = \{\{u_1, u_2\}, \{u_1, \bar{u}_2\}, \{\bar{u}_1\}\}$.

Cook demonstrated in 1971 that satisfiability is an *NP*-complete problem. This proof required that he show that for all languages $L \in NP$, $L \leq L_{SAT}$. Once, *SAT* was established as an *NP*-complete problem, it became much easier to devise an *NP*-complete proof for other languages.

P and NP

- As mentioned, **P** is set of problems that can be solved in polynomial time
- **NP** (*nondeterministic polynomial time*) is the set of problems that can be solved in polynomial time by a *nondeterministic* computer
 - *What is that?*

Nondeterminism

- Think of a **non-deterministic computer (NDTM)** as a computer that magically “guesses” a solution, then has to verify that it is correct
 - If a solution exists, computer always guesses it
 - One way to imagine it: a parallel computer that can freely spawn an infinite number of processes
 - Have one processor work on each possible solution
 - All processors attempt to verify that their solution works
 - If a processor finds it has a working solution
 - So: **NP** = problems *verifiable* in polynomial time

Showing that a Decision Problem $\Pi \in \text{NP}$

- A language $L \in \text{NP}$ *iff* there is a polynomial scaling algorithm for language membership using a polynomial nondeterministic program such as a GUESS function if available (*Polynomially checkable certificate: Refer to the definition in the book.*)
 - i.e. $\Pi \in \text{NP}$ *iff* Π is solvable by a polynomial nondeterministic program
 - We will focus on YES Instances
 - We will use GUESS() Function
- Example
 - Show that $\text{SAT} \in \text{NP}$
 - INSATNCE: Given a set of literals $X = \{x_1, x_2, \dots, x_n\}$ in a clause C (Input Formula)
 - QUESTION: Is there a satisfying truth assignment for C

Show that $\text{SAT} \in \text{NP}$

- NDTM

- **GUESS(x)**

- Sets the Boolean variable x to either 0 or 1, so that the whole formula says “YES” to the input if possible.

- Using GUESS we get an easy program to check satisfiability

- The program to check satisfiability (Polynomial time)

- For each Boolean variable x

- **GUESS(x)**

- Check if the guesses value satisfy the input formula

- **Return YES** if so
 - **Return NO**, otherwise

Reduction

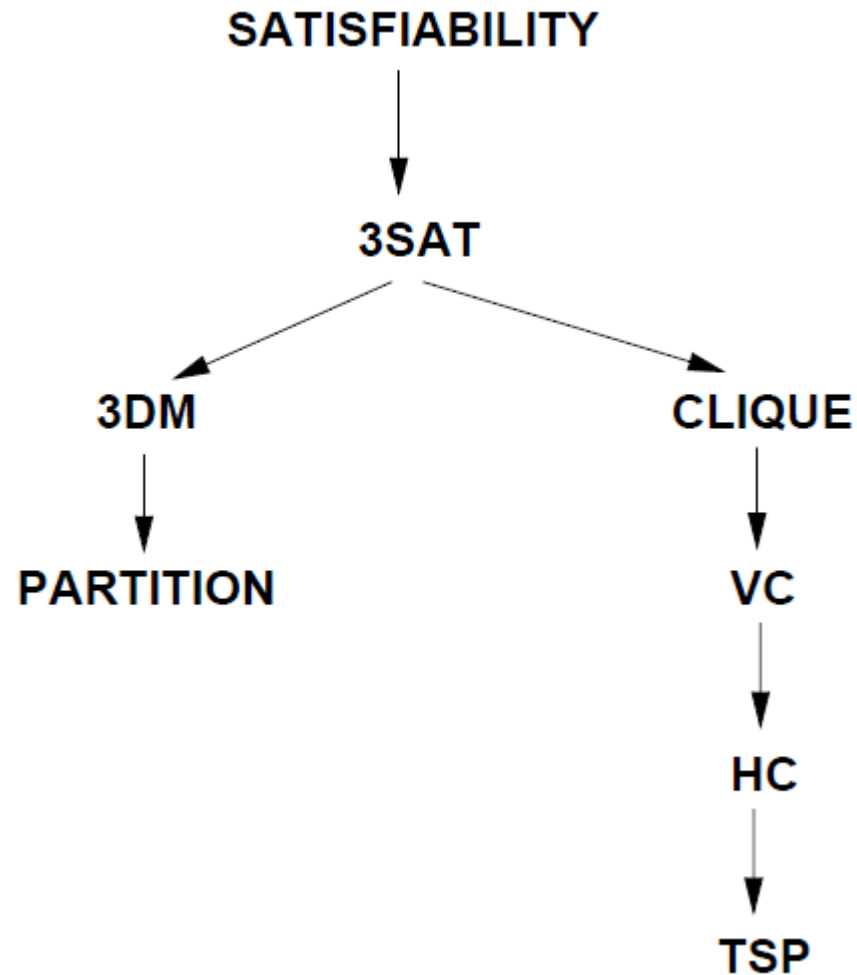
- The crux of NP-Completeness is *reducibility*
 - Informally, a problem P can be reduced to another problem Q if *any* instance of P can be “easily rephrased” as an instance of Q, the solution to which provides a solution to the instance of P
 - *What do you suppose “easily” means?*
 - This rephrasing is called *transformation*
- Intuitively: If P reduces in polynomial time to Q, P is “no harder to solve” than Q

Using Reductions

- If P is *polynomial-time reducible* to Q , we denote this $P \leq_p Q$
- Definition of NP-Complete:
 - If **Q is NP-Complete**, then (1) **$Q \in \text{NP}$** , and (2) **all problems $R \in \text{NP}$ are reducible to Q**
 - Formally: $R \leq_p Q \ \forall R \in \text{NP}$
- If $P \leq_p Q$ and P is NP-Complete, Q is also NP-Complete
 - This is the *key idea* you should take away today

Reductions

- The figure below displays a path of reductions from a known NP-complete problem to a target problem:



NP-Hard and NP-Complete

- If P is *polynomial-time reducible* to Q, we denote this $P \leq_p Q$
- Definition of NP-Hard and NP-Complete:
 - If all problems $R \in \mathbf{NP}$ are reducible to P, then P is *NP-Hard*
 - We say P is *NP-Complete* if P is NP-Hard and $P \in \mathbf{NP}$
- If $P \leq_p Q$ and P is NP-Complete, Q is also NP-Complete

Why Prove NP-Completeness?

- Though nobody has proven that $\mathbf{P} \neq \mathbf{NP}$, if you prove a problem NP-Complete, most people accept that it is probably intractable
- Therefore it can be important to prove that a problem is NP-Complete
 - Don't need to come up with an efficient algorithm
 - Can instead work on *approximation algorithms*

Steps in an NP-completeness Proof

To prove that a language L is NP -complete:

1. Prove that $L \in NP$.
2. Select a known NP -complete language L' .
3. Construct a function f that maps every instance of L' to an instance of L .
4. Prove that f is a polynomial transformation.
 - (a) Prove that f satisfies $x \in L'$ iff $f(x) \in L$.
 - (b) Prove that the algorithm computing f runs in polynomial time.

Review Questions

- *What do we mean when we say a problem is in **P**?*
- *What do we mean when we say a problem is in **NP**?*
- *What is the relation between **P** and **NP**?*

Review Questions

- *What, intuitively, does it mean if we can **reduce** problem P to problem Q ?*
- *How do we reduce P to Q ?*
- *What does it mean if Q is **NP-Hard**?*
- *What does it mean if Q is **NP-Complete**?*

The End
