NP-COMPLETENESS

- ***Polynomial-time algorithms*** : on inputs of size *n*, their worst-case running time is $O(n^k)$ *for some constant k.*
- ***Superpolynomial time algorithms***: it can be solved, but not in time $O(n^k)$ *for any constant k.*
- Generally, we think of problems that are solvable by polynomial-time algorithms as being ***tractable***, and problems that require superpolynomial time as being ***intractable***.
- There are also problems that cannot be solved by any computer, no matter how much time is provided.

- "***NP-complete***" problems, whose status is unknown. No polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove a superpolynomial-time lower bound for any of them.

- A particularly tantalizing aspect of the NP-complete problems is that several of them seem on the surface to be similar to problems that have polynomial-time algorithms.
- Shortest vs. longest paths
- Euler tour vs. Hamiltonian
- 2-CNF satisfiability vs. 3-CNF satisfiability

NP-completeness and the Classes

- The class **P** consists of those problems that are solvable in polynomial time.
- The class **NP** consists of those problem that are "verifiable" in polynomial time.
- Any problem in P is also in NP, that is $P \subseteq NP$.
- A problem is in the class NPC if it is in NP and is as "hard" as any problem in NP.

NP-completeness and the Classes – contd.

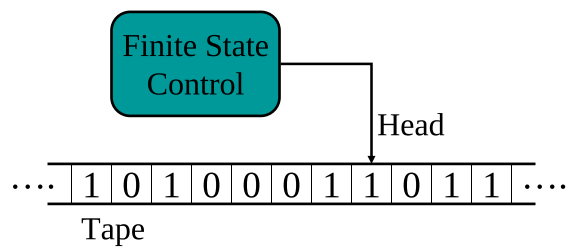
- In summary, the three classes of problems
  - P: problems solvable in poly time.
  - NP: problems verifiable in poly time.
  - NPC: problems in NP and as hard as any problem in NP.

Traditional definition of NP

- Turing machine model of computation
  - Simple model where data is on an infinite capacity tape
  - Only operations are reading char stored in current tape cell, writing a char to current tape cell, moving tape head left or right one square
- Deterministic versus nondeterministic computation
  - Deterministic: At any point in time, next move is determined
  - Nondeterministic: At any point in time, several next moves are possible
- NP: Class of problems that can be solved by a nondeterminstic turing machine in polynomial time

Turing Machines

A Turing machine has a finite-state-control (its program), a two way infinite tape (its memory) and a read-write head (its program counter)



NP-Completeness (Verifiable)

- Verifiable in poly time: given a certificate of a solution, could verify the certificate is correct in poly time.
- Examples (their definitions come later):
  - Hamiltonian-cycle, given a certificate of a sequence (v_1,v_2,…, v_n), easily verified in poly time.
  - 3-CNF, given a certificate of an assignment 0s, 1s, easily verified in poly time.
  - (so try each instance, and verify it, but 2^n instances)
- Why not defined as “solvable in exponential time?” or “Non Poly time”?

NP-Completeness (why NPC?)

- A problem $p \in$NP, and any other problem p' $\in$NP can be translated as p in poly time.
- So if p can be solved in poly time, then all problems in NP can be solved in poly time.
- All current known NP hard problems have been proved to be NPC.

Relation among P, NP, NPC

- $P \subseteq NP$ (Sure)
- $NPC \subseteq NP$ (sure)
- $P = NP$ (or $P \subset NP$, or $P \neq NP$) ???
- $NPC = NP$ (or $NPC \subset NP$, or $NPC \neq NP$) ???
- $P \neq NP$: one of the deepest, most perplexing open research problems in (theoretical) computer science since 1971.

Arguments about P, NP, NPC

- No poly algorithm found for any NPC problem (even so many NPC problems)
- No proof that a poly algorithm cannot exist for any of NPC problems, (even having tried so long so hard).
- Most theoretical computer scientists believe that NPC is intractable (i.e., hard, and $P \neq NP$).

Importance of NP-completeness Importance of “Is P=NP” Question

- Practitioners view
  - There exist a large number of interesting and seemingly different problems which have been proven to be NP-complete
  - The P=NP question represents the question of whether or not all of these interesting and different problems belong to P
  - As the set of NP-complete problems grows, the question becomes more and more interesting

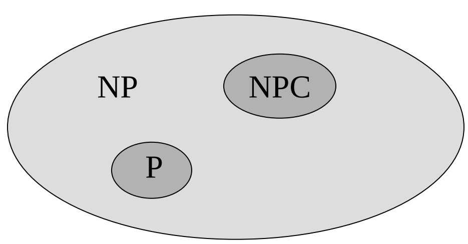
List of Problem Types from Garey & Johnson, 1979

- Graph Theory
- Network Design
- Sets and Partitions
- Storage and Retrieval
- Sequencing and Scheduling
- Mathematical Programming
- Algebra and Number Theory
- Games and Puzzles
- Logic
- Automata and Languages
- Program Optimization
- Miscellaneous

Importance of NP-completeness
Importance of "Is P=NP" Question

- Theoretician's view
  - NP is exactly the set of problems that can be "verified" in polynomial time
  - Thus "Is P=NP?" can be rephrased as follows:
    - Is it true that any problem that can be "verified" in polynomial time can also be "solved" in polynomial time?
- Hardness Implications
  - It seems unlikely that all problems that can be verified in polynomial time also can be solved in polynomial time
  - If so, then P≠NP
  - Thus, proving a problem to be NP-complete is a hardness result as such a problem will not be in P if P≠NP.

View of Theoretical Computer Scientists on P, NP, NPC



$P \subset NP$, $NPC \subset NP$, $P \cap NPC = \varnothing$

Why discussion on NPC

- If a problem is proved to be NPC, a good evidence for its intractability (hardness).
- Not waste time on trying to find efficient algorithm for it
- Instead, focus on design approximate algorithm or a solution for a special case of the problem
- Some problems looks very easy on the surface, but in fact, is hard (NPC).

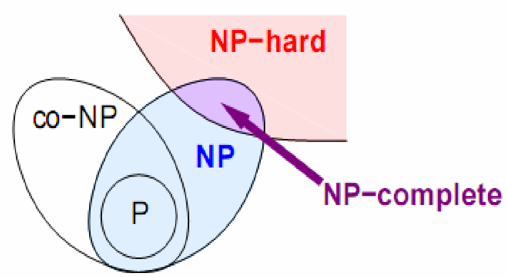
Overview of showing problems to be NP-complete

- The techniques we use to show that a particular problem is NP-complete differ from the techniques used throughout of this book to design and analyze algorithms. We rely on three key concepts in showing a problem to be NP-complete:
- Decision problems vs. optimization
- Reductions
- A first NP-complete problem

Complexity Class co-NP

- $P \neq NP$?
- Complexity class co-NP
  $co-NP = \{L \mid \overline{L} \in NP\}$
- $NP = co-NP$?
- Obviously $P \subset NP \cap co-NP$
- $P = NP \cap co-NP$?

Complexity Classes at-a-Glance



More of what we *think* the world looks like.

Polynomial time

- All problems in **P** are generally regarded as tractable, but for philosophical, not for mathematical, reasons. We can offer three supporting arguments:
  - 1. There are very few practical problems that require time on the order of such a high-degree polynomial-time $\Theta(n^{100})$.
  - 2. For many reasonable models of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another.
  - 3. The class of polynomial-time solvable problems has nice closure properties, since polynomials are closed under addition, multiplication, and composition.

Abstract Problems

- We define an ***abstract problem*** *Q* to be a binary relation on a set *I* of problem ***instances*** and a set *S* of problem ***solutions***.
- The theory of NP-completeness restricts attention to ***decision problems:*** those having a yes/no solution.
- Many abstract problems are not decision problems, but rather ***optimization problems***, in which some value must be minimized or maximized.

Encodings

- If a computer program is to solve an abstract problem, problem instances must be represented in a way that the program understands.
- An ***encoding*** of a set *S* of abstract objects is a mapping *e* from *S* to the set of binary strings.
- We call a problem whose instance set is the set of binary strings a ***concrete problem***.

- We can now formally define the ***complexity class P*** as the set of concrete decision problems that are solvable in polynomial time.
- We can use encodings to map abstract problems to concrete problems.
- The encoding of an abstract problem is quite important to our understanding of polynomial time. We cannot really talk about solving an abstract problem without first specifying an encoding.

- We say that a function $f : \{0, 1\}^* \to \{0, 1\}^*$ is ***polynomial-time computable*** *if there* exists a polynomial-time algorithm A that, given any input $x \in \{0, 1\}^*$, produces as output $f(x)$.
- For some set I of problem instances, we say that two encodings e_1 and e_2 are ***polynomially related*** if there exist two polynomial-time computable functions f_{12} and f_{21} such that for any $i \in I$, *we have* $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i)$.

- ***Lemma 34.1***

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

- In order to be able to converse in an encoding-independent fashion, we shall assume that the encoding of an integer is polynomially related to its binary representation, and that the encoding of a finite set is polynomially related to its encoding as a list of its elements, enclosed in braces and separated by commas.

A formal-language framework

- An ***alphabet*** Σ is a finite set of symbols. A ***language*** L over Σ is any set of strings made up of symbols from Σ.
- We denote the ***empty string*** by ε, and the ***empty language*** by Φ. The language of all strings over Σ is denoted Σ^*. Every language L over Σ is a subset of Σ^*.

- There are a variety of operations on languages. Set-theoretic operations, such as ***union*** and ***intersection***, follow directly from the set-theoretic definitions. We define the ***complement*** of L *by* $\overline{L} = \Sigma^* - L$.
- The ***concatenation*** of two languages L_1 and L_2 is the language

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\} .$$

- The ***closure*** or ***Kleene star*** of a language L is the language

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \quad . . ,$$

where L^k is the language obtained by concatenating L to itself k times.

- From the point of view of language theory, the set of instances for any decision problem Q is simply the set Σ^*, where $\Sigma = \{0, 1\}$. Since Q is entirely characterized by those problem instances that produce a 1 (yes) answer, we can view Q as a language L over $\Sigma = \{0, 1\}$, where

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

- We say that an algorithm A ***accepts*** a string $x \in \{0, 1\}^*$ if, given input x, the algorithm outputs $A(x) = 1$. The language ***accepted*** by an algorithm A is the set $L = \{x \in \{0, 1\}^* : A(x) = 1\}$, that is, the set of strings that the algorithm accepts. An algorithm A ***rejects*** a string x if $A(x) = 0$.

- A language L is ***decided*** by an algorithm A if every binary string is either accepted or rejected by the algorithm.
- A language L is ***accepted in polynomial time*** by an algorithm A if for any length-n string $x \in L$, the algorithm accepts x in time $O(n^k)$ for some constant k.
- A language L is ***decided in polynomial time*** by an algorithm A if for any length-n string $x \in \{0, 1\}^*$, the algorithm decides x in time $O(n^k)$ for some constant k.

Polynomial-time verification

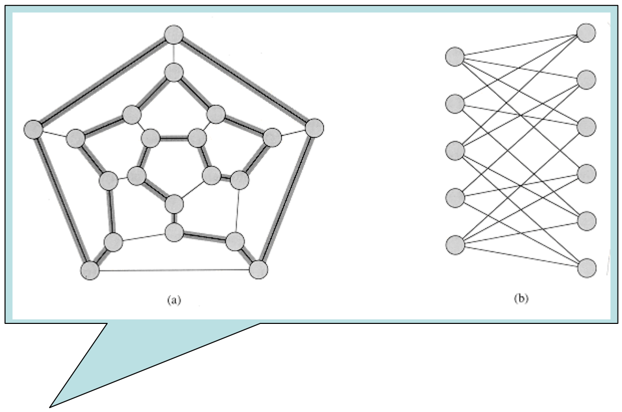
- We now look at algorithms that "verify" membership in languages.

Hamiltonian Cycles

- Formally, a ***hamiltonian cycle*** of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V. A graph that contains a hamiltonian cycle is said to be ***hamiltonian;*** otherwise, it is **nonhamiltonian.**
- We can define the ***hamiltonian-cycle problem,*** "Does a graph G have a hamiltonian cycle?" as a formal language:

  HAM-CYCLE = { <*G*> : *G* is a hamiltonian graph} .

Hamiltonian Cycles – contd.



Verification algorithms

- We define a ***verification algorithm*** as being a two-argument algorithm A**,** where one argument is an ordinary input string x and the other is a binary string y called a ***certificate***. A two-argument algorithm A ***verifies*** an input string x if there exists a certificate y such that $A(x, y) = 1$.
- The ***language verified*** by a verification algorithm A is

  $L = \{x \in \{0, 1\}^* :$ there exists $y \in \{0, 1\}^*$ such that $A(x, y) = 1\}$.

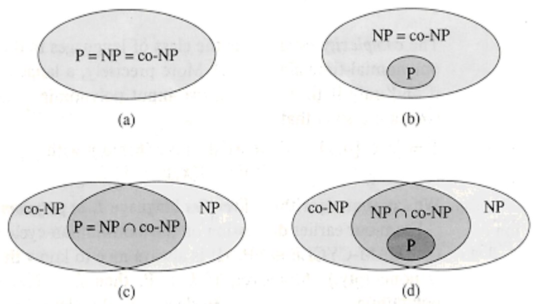
- Intuitively, an algorithm *A* verifies a language *L* if for any string $x \in L$, there is a certificate *y* that *A* can use to prove that $x \in L$. Moreover, for any string $x \notin L$ there must be no certificate proving that $x \in L$.

The complexity class NP

- The ***complexity class NP*** is the class of languages that can be verified by a polynomial-time algorithm. More precisely, a language *L* belongs to *NP* if and only if there exists a two-input polynomial-time algorithm *A* and constant *c* such that

  $L = \{x \in \{0,1\}^* : \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x,y) = 1\}$.
- We say that algorithm *A* ***verifies*** language *L* ***in polynomial time***.

- We can define the ***complexity class co-NP*** as the set of languages L such that $\overline{L} \in NP$.
- Many other fundamental questions beyond the P≠NP question remain unresolved.



NP-Completeness and Reducibility

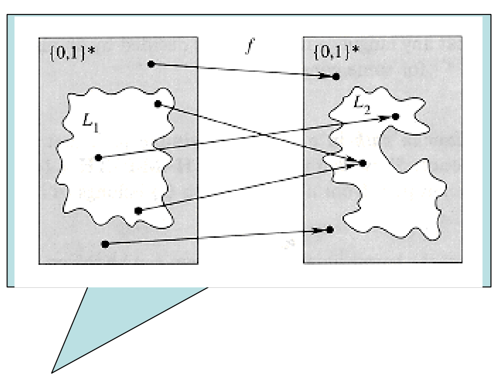
- The NP-complete languages are, in a sense, the "hardest" languages in NP. In this section, we shall show how to compare the relative "hardness" of languages using a precise notion called "polynomial-time reducibility."

Reducibility

- We say that a language L_1 is ***polynomial-time reducible*** to a language L_2, written $L_1 \leq_P L_2$, if there exists a polynomial-time computable function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that for all $x \in \{0, 1\}^*$,

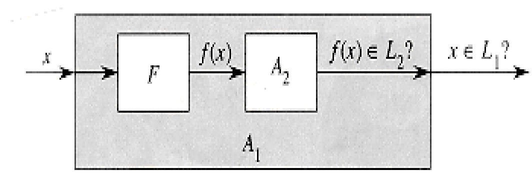
  $x \in L_1$ if and only if $f(x) \in L_2$.
- We call the function f the reduction function, and a polynomial-time algorithm F that computes f *is* called a ***reduction algorithm***.

Reducibility – contd.



- ***Lemma***

  If $L_1, L_2 \subseteq \{0, 1\}^*$ are languages such that $L_1 \leq_P L_2$, then $L_2 \in$ P implies $L_1 \in$ P.



NP-Completeness

- We can now define the set of NP-complete languages, which are the hardest problems in NP.

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

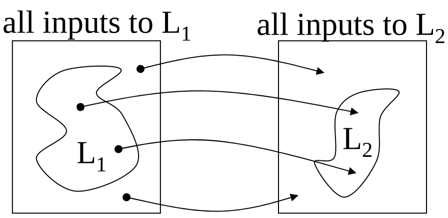
  1. $L \in NP$, and
  2. $L' \leq_p L$ for every $L' \in NP$.
- If a language L satisfies property 2, but not necessarily property 1, we say that L is ***NP-hard***. We also define NPC to be the class of NP-complete languages.

NP-Completeness – contd.

Definition: A decision problem L is NP-Complete (NPC) if:

1. L $\in$ NP, and

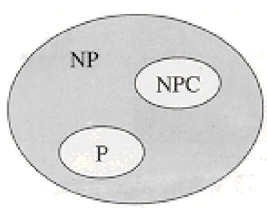
2. for every L' $\in$ NP, L' $\leq_p$ L (i.e., every L' in NP can be transformed to L, meaning L is at least as hard as every problem in NP).



$L_1 \leq_p L_2$

$\forall$ inputs x, x $\in L_1$ iff f(x) $\in L_2$

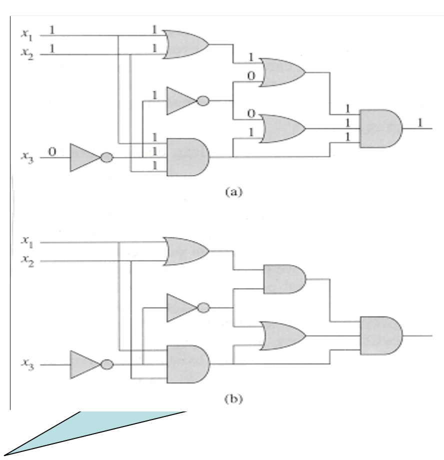
- ***Theorem***
- If any NP-complete problem is polynomial-time solvable, then P = NP. If any problem in NP is not polynomial-time solvable, then all NP-complete problems are not polynomial-time solvable.



Circuit Satisfiability

- A ***truth assignment*** for a boolean combinational circuit is a set of boolean input values. We say that a one-output boolean combinational circuit is ***satisfiable*** if it has a ***satisfying assignment***: a truth assignment that causes the output of the circuit to be 1.

Circuit Satisfiability – contd.



- The ***circuit-satisfiability problem*** is, "Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?"
- One can devise a graphlike encoding that maps any given circuit *C* into a binary string <*C*> whose length is not much larger than the size of the circuit itself. As a formal language, we can therefore define

  CIRCUIT-SAT = {<*C*>:*C* is a satisfiable boolean combinational circuit} .

- ***Lemma*** The circuit-satisfiability problem belongs to the class NP.

- ***Lemma*** The circuit-satisfiability problem is NP-hard.

- ***Theorem*** The circuit-satisfiability problem is NP-complete.

NP-Completeness Proofs

- ***Lemma***

  If L is a language such that $L' \leq_p L$ for some $L' \in$NPC, then L is NP-hard. Moreover, if $L \in$NP, then $L \in$NPC.

- Lemma gives us a method for proving that a language L is NP-complete:
  - 1. Prove $L \in \mathrm{NP}$.
  - 2. Select a known NP-complete language L'.
  - 3. Describe an algorithm that computes a function f mapping every instance of $x \in \{0, 1\}^*$ of L' to an instance $f(x)$ of L.
  - 4. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0, 1\}^*$.
  - 5. Prove that the algorithm computing f runs in polynomial time.

Formula Satisfiability

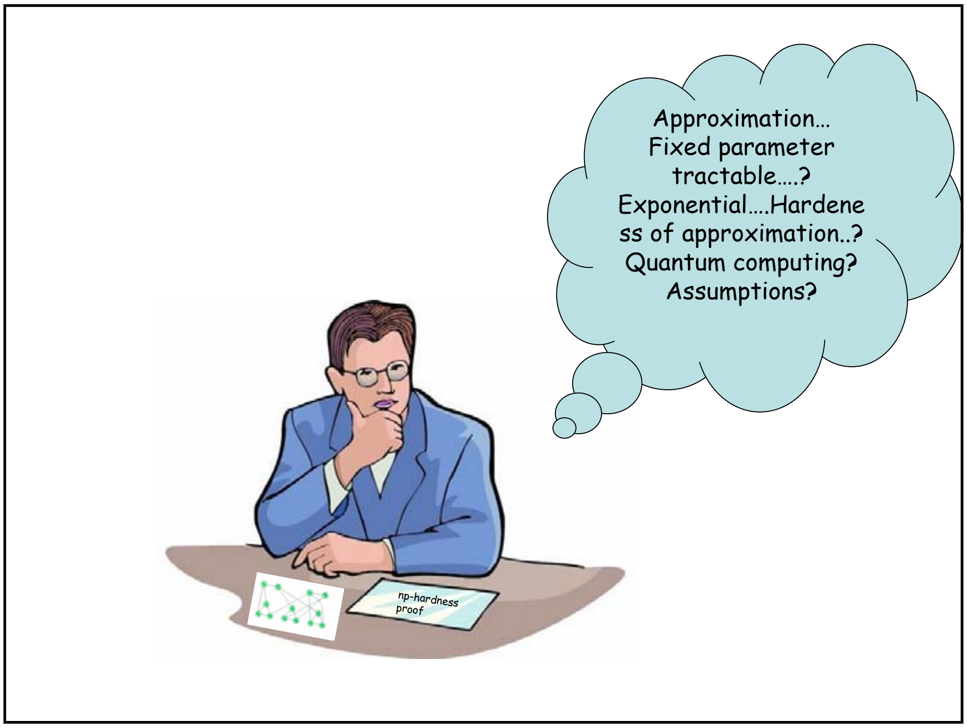
- We formulate the ***(formula) satisfiability*** problem in terms of the language SAT as follows. An instance of SAT is a boolean formula φ composed of
  - 1. boolean variables: $x_1, x_2, \ldots$;
  - 2. boolean connectives: any boolean function with one or two inputs and one output, such as $\wedge$(AND), $\vee$(OR), $\neg$(NOT), $\rightarrow$(implication), $\leftrightarrow$(if and only if); and
  - 3. parentheses.

- As in boolean combinational circuits, a ***truth assignment*** for a boolean formula is a set of values for the variables of φ.
- A ***satisfying assignment*** is a truth assignment that causes it to evaluate to 1.
- A formula with a satisfying assignment is a ***satisfiable*** formula.
- The satisfiability problem asks whether a given boolean formula is satisfiable; in formal-language terms,

  SAT = $\{\langle\varphi\rangle : \varphi$ is a satisfiable boolean formula$\}$.

- ***Theorem***

  Satisfiability of boolean formulas is NP-complete.


Approximation…
Fixed parameter
tractable….?
Exponential….Hardene
ss of approximation..?
Quantum computing?
Assumptions?
np-hardness
proof