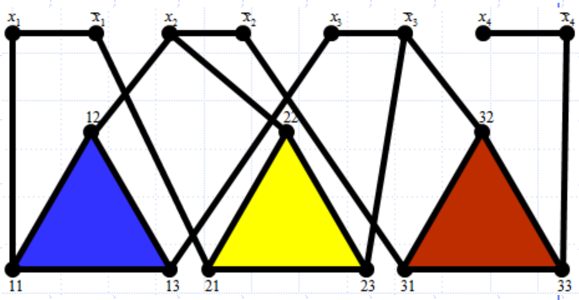
Lesson 15
NP-Complete Problems:
Handling Problems From the Field of All Possibilities

Wholeness of the Lesson

Decision problems that have no known polynomial time solution are considered *hard*, but hard problems can be further classified to determine their degree of hardness. A decision problem belongs to NP if there is a polynomial *p* and an algorithm *A* such that for any instance of the problem of size *n*, a correct solution to the problem can be *verified* using *A* in at most *p*(*n*) steps. In addition, the problem is said to be *NP-complete* if it belongs to NP and every NP problem can be polynomial-reduced to it.



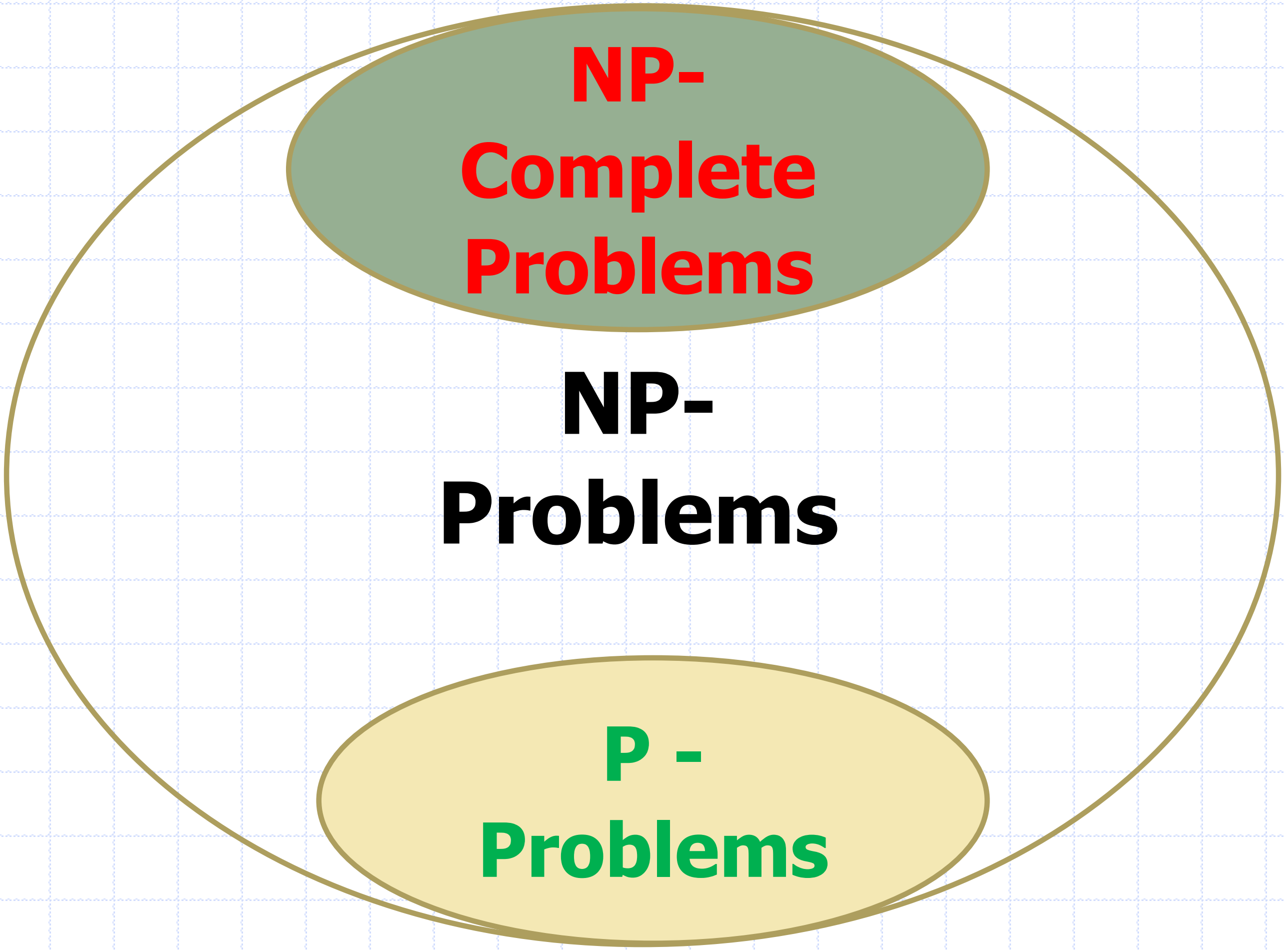
Science of Consciousness: The human intellect can grasp truths within a certain range but is not the only faculty of knowing. The transcendental level of awareness is a field beyond the grasp of the intellect ("beyond even the intellect is he" -- Gita, III.42). And the field of manifest existence, from gross to subtle, is too vast and complex to be grasped by the intellect either ("unfathomable is the course of action" – Gita IV.17).

Overview

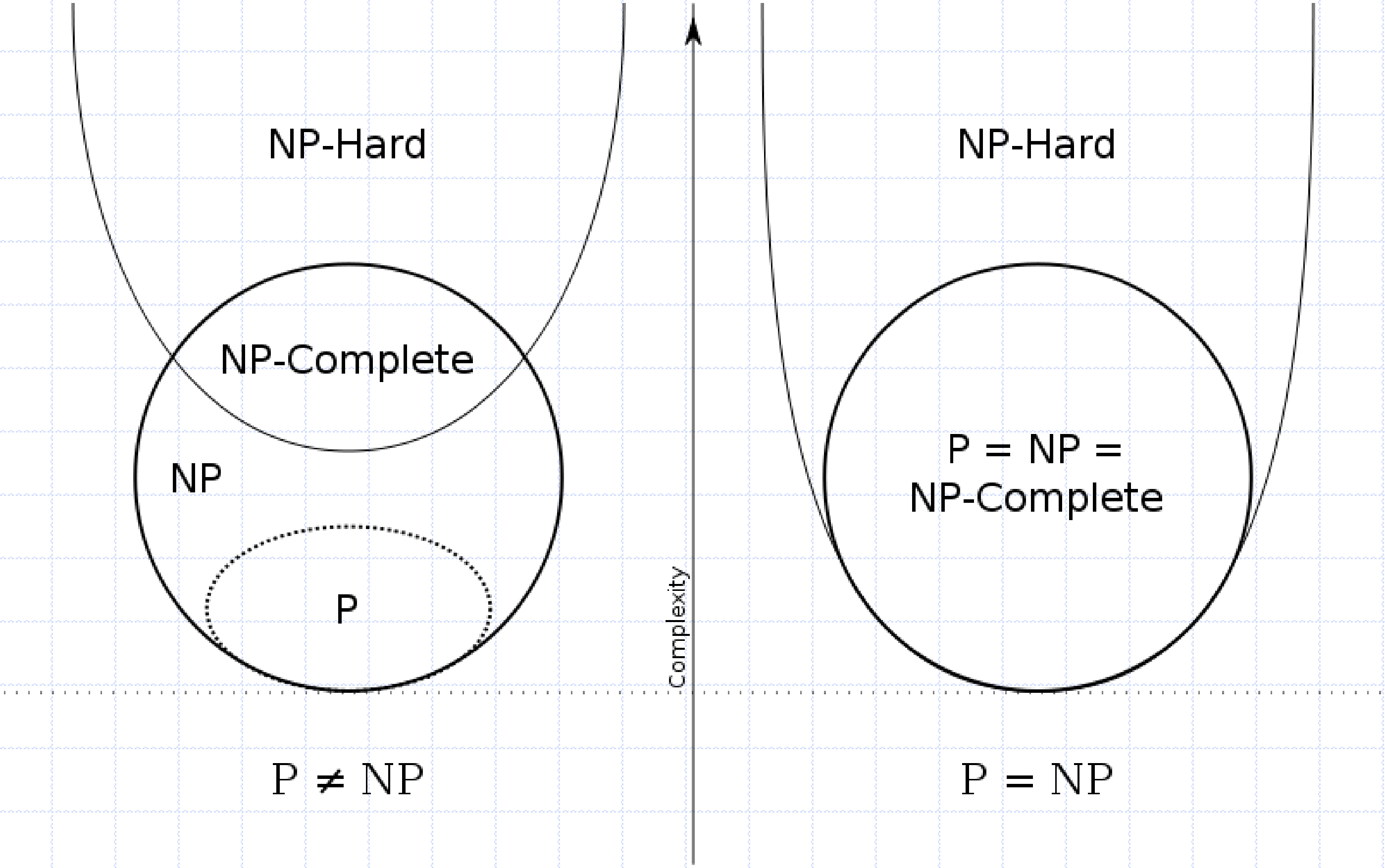
In this lesson:

- Review polynomial time decision problems and the class *P*
- Review the class *NP* of decision problems.
- Describe the class of “hard” NP problems – the *NP complete problems.*
- Demonstrating NP-completeness, with examples, and the *P=NP* problem
- Techniques for handling *NP*-completeness in practice, with examples.

P versus NP problem



P versus NP problem



Polynomial-Time Algorithms

- Are some problems solvable in polynomial time?
  - Of course: many algorithms we've studied provide polynomial-time solutions to some problems
- Are all problems solvable in polynomial time?
  - No: Turing's "Halting Problem" is not solvable by any computer, no matter how much time is given
- Most problems that do not yield polynomial-time algorithms are either optimization or decision problems.

How Do We Know When a Problem Does Not Belong to *P*?

- Hard to know for sure because even if there is no known polynomial time algorithm today, tomorrow someone may come up with one.
- Modern-day example: The **IsPrime** problem. Before 2002, all known deterministic algorithms to solve this problem ran in superpolynomial time.
- **AKS Primality Test** was the first polynomial-time solution. Its fastest known implementation runs in

  $$O(\mathit{length}(n)^6 * \log^k(\mathit{length}(n)))$$

  for some k. (AKS stands for Agrawal–Kayal–Saxena)
- *EXPTIME-Complete problems*, like nxn chess, *do require* superpolynomial time and therefore do not belong to *P*

The class *NP*

- To understand decision problems that may not belong to *P*, one approach is to see how hard it is to *check* whether a given solution is correct. Typically easier to check a solution than to obtain a solution in the first place.
- The class of decision problems with the property that a solution can be verified in polynomial time is denoted *NP*.

Optimization/Decision Problems

- Optimization Problems
  - An optimization problem is one which asks, “What is the optimal solution to problem X?”
  - Examples:
    - 0-1 Knapsack
    - Fractional Knapsack
    - Minimum Spanning Tree
- Decision Problems
  - A decision problem is one with yes/no answer
  - Examples:
    - Does a graph G have a MST of weight $\leq$ W?

Optimization/Decision Problems

- An optimization problem tries to find an optimal solution
- A decision problem tries to answer a yes/no question
- Many problems will have decision and optimization versions
  - Eg: Traveling salesman problem
    - optimization: find hamiltonian cycle of minimum weight
    - decision: is there a hamiltonian cycle of weight ≤ k
- Some problems are decidable, but *intractable*: as they grow large, we are unable to solve them in reasonable time
  - *Is there a polynomial-time algorithm that solves the problem?*

NP-Complete Vrs. NP-Hard

- **NP-complete** problems are a set of problems that any other **NP**-problem can be reduced to in polynomial time, but retain the ability to have their solution verified in polynomial time. Alternatively,

  A problem (in NP) is NP-complete if any problem in NP is reducible to it

  Examples: Traveling Salesman Problem & Hamiltonian Cycle problem

- NP-hard problems are those at least as hard as **NP**-complete problems, meaning all **NP**-problems can be reduced to them, but not all **NP-hard** problems are in **NP**, meaning not all of them have solutions verifiable in polynomial time.

Yet Another Definition of NP

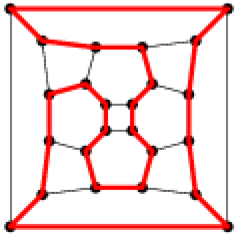
NP: the class of decision problems that are solvable in polynomial time on a *nondeterministic* machine (or with a nondeterministic algorithm)

- (A *determinstic* computer is what we know)
- A *nondeterministic* computer is one that can "guess" the right answer or solution
  - Think of a nondeterministic computer as a parallel machine that can freely spawn ***an infinite number*** of processes (to some extent similar to **Quantum** Computing).
- Note that *NP* stands for "Nondeterministic Polynomial-time"

NP Examples

- The HamiltonianCycle and VertexCover problems have already been shown to belong to *NP.*

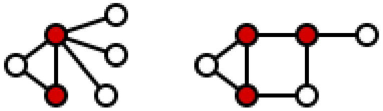
HamiltonianCycle: Given G=(V,E), does G have a Hamiltonian Cycle? Solution data: a subset of E



Hamiltonian path is a **path** in an undirected or directed graph that visits each vertex exactly once. A **Hamiltonian cycle** (or **Hamiltonian circuit**) is a **Hamiltonian path** that is a **cycle**

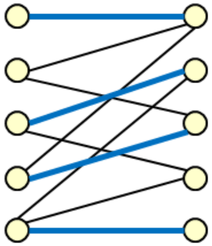
set of vertices that include atleast one end pt. of all the edges in a graph

VertexCover: Given G=(V,E) and k, does G have a vertex cover of size at least k? Solution data: subset of V



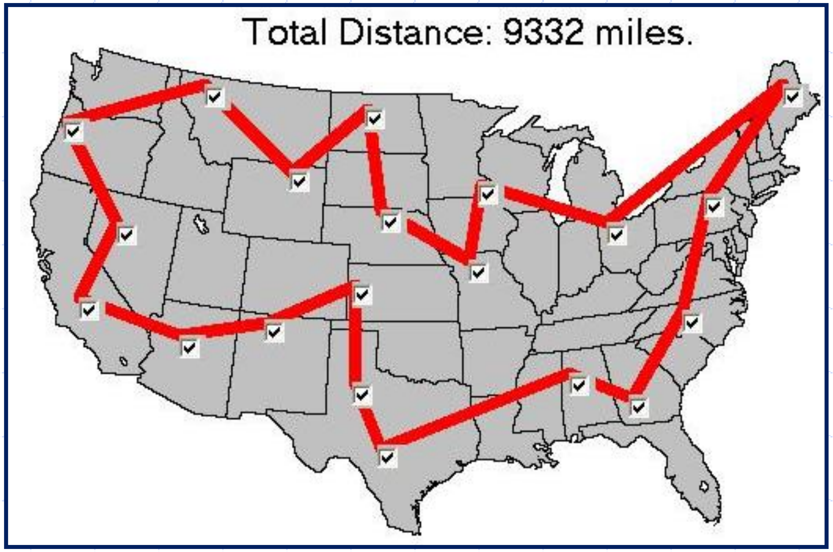
A matching in a graph is a set of edges no two of which share an endpoint.

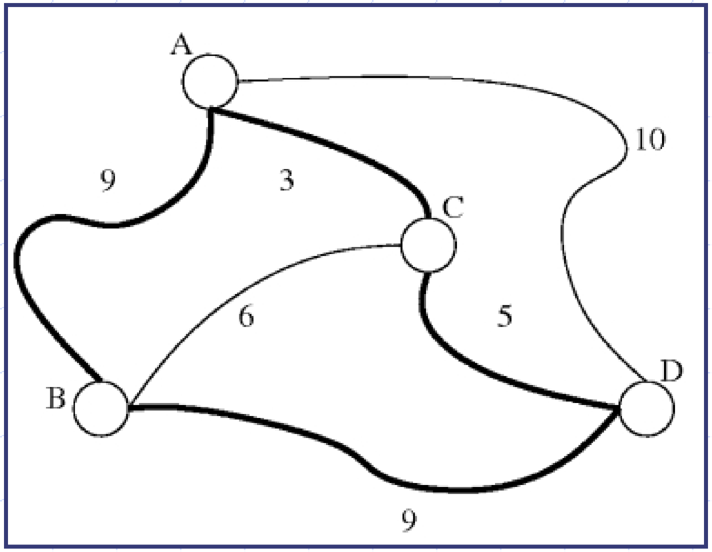
Konig's Theorem. In a bipartite graph, the number of edges in a maximum matching is the same as the number of vertices in a minimum vertex cover.



Traveling Salesman Problem

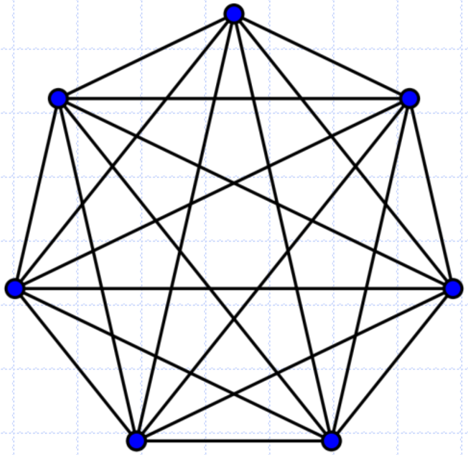
- *Traveling Salesman Problem* (TSP): Given a complete graph G with cost function c: E → N and a positive integer k, is there a Hamiltonian cycle C in G so that the sum of the costs of the edges in C is at most k? Solution data: a subset of E.





Complete Graph, Induced Graph and Clique

A complete graph is a graph where every pair of distinct vertices are connected by a unique edge. Example: K_7 - a complete graph with 7 vertices



A subgraph, *H*, of a graph, *G*, is said to be **induced** (or **full**) if, for every pair of vertices *x* and *y* of H, *xy* is an edge of *H* if and only if *xy* is an edge of G. Check the other slide for NP

A **clique** is subset of vertices of an undirected graph, such that its induced subgraph is complete.

TSP Belongs to NP

- Given an instance I of TSP with input data a complete graph G with n vertices, a cost function c: E → N and a positive integer k, and given a certificate E_0, a subset of E, the algorithm B checks that E_0 is a Hamiltonian cycle for G and then computes the sum of c(e) over the edges e in E_0 to verify that it is at most k.

Example: Not in ***NP***

- *PowerSet problem.* Given a set X of size n, a kind of optimization problem concerning the power set of X is to generate all subsets of X (“find the largest possible collection of subsets of X without duplicates”). Whatever method is used, just writing out the output requires at least 2^n steps. A corresponding decision problem is: Given a set X and a collection P, is P = P(X)? Any algorithm that solves this problem must check every element of P to see if it is a subset of X, so the algorithm requires at least 2^n steps in the worst case. So this problem does not belong to P.

  Moreover, verifying correctness requires checking that each set that is output is a subset of X, and this has to be done 2^n times, so it doesn’t belong to NP either.

[A Power Set is a set of *all the subsets of a set*. E.g. for set {a,b,c}, the Power set is { {}, {a}, {b}, {c}, {a,b}, {a,c}, {b,c}, {a,b,c} }]

- *Finite Halting Problem and N x N Chess.* Finite Halting Problem: Given a program P and positive integers n, k, does P, when running with input n, produce an output after excecuting k or fewer steps? Neither of these problems belong to P, but it is not known whether they belong to NP.

Is ***P*** = ***NP***?

- The answer is not known
- Many thousands of research papers have been written in an attempt to make progress in solving this problem.
- If it were true, then thousands of problems that were believed to have infeasible solutions would suddenly have feasible solutions
- Clay Mathematics Institute (CMI, Rhode Island, Providence, USA) in 2000 has announced $1M award if one can prove that P = NP or P! = NP.

[In fact, there are 6 other problems, each for $1M. These are called Millennium Prize Problems]

A Remarkable Fact About ***NP***

- Many of the problems that are known to be in NP can also be shown to be *NP-complete.* Intuitively, this means they are the hardest among the NP problems.
- It can be shown that if someone ever figures out a polynomial-time algorithm to solve an NP-complete problem, then *all problems in NP will also have polynomial time solutions.*
- One consequence: If anyone finds a polynomial time solution to an NP-complete problem, then P = NP.
- These points are elaborated in the next slides

Reducibility

- *Intuitively*: Q is *polynomial reducible* to R if, in polynomial time, you can transform a solvable problem of type Q into a solvable one of type R so that a solution to one yields a solution to the other.

- *Formally*: A problem Q is *polynomial reducible* to a problem R if there is a polynomial p(y) and an algorithm C so that when C runs on input data X of size O(n) for an instance I_Q of Q, C outputs input data Y of size O(p(n)) for an instance I_R of R in O(p(n)) steps so that

  I_Q has a solution iff I_R has a solution

  In this case, we say that C, p(y) *witness* that Q is polynomial reducible to R.

- We write $Q \xrightarrow{\text{poly}} R$

↳ Q problem can be reduced
to R " in poly time

VertexCover $\xrightarrow{\text{poly}}$ HamiltonianCycle

It can be shown that VertexCover is polynomial reducible to HamiltonianCycle. What does this mean?

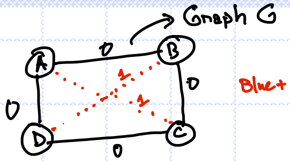
- *Formally:* There is a polynomial p(y) and an algorithm C that does the following: Using an instance G, k of the VertexCover problem (where G has n vertices) as input to C, C outputs, in O(p(n)) time, a graph H with O(p(n)) vertices so that:

  G has a vertex cover of size at most k iff

  H has a Hamiltonian cycle

- *Intuitively*: the VertxCover problem can be turned into a Hamiltonian Cycle problem in polynomial time, so if you can solve the Hamiltonian Cycle problem, you can solve the VertexCover problem without doing much more work.

- *Note*: The details for this algorithm C are tricky – not given here, but we make use of this result later.



Blue+ Red ⇒ graph H

① The HCycle in G is the HCycle in H

② We have to proof in both directions

HamiltonianCycle $\xrightarrow{\text{poly}}$ TSP

We show HamiltonianCycle is reducible to TSP

- Given a graph G = (V,E) on n vertices (input for HamiltonianCycle) – notice G is a subgraph of K_n. Obtain an instance H, c, k of TSP as follows: Let H be the complete graph on n vertices (i.e. H is K_n), obtained by adding the missing edges to G. Let
  $c(e) = 0$ if $e \in E$, else $c(e) = 1$. Let $k = 0$.
- Need to show: G has a Hamiltonian cycle if and only if H, c, k has a Hamiltonian cycle with edge cost $\leq k$
- If G has Hamiltonian cycle C, C is Hamiltonian in H also. Since each edge e of C is in G, $c(e) = 0$. So cost sum $\leq k$. Converse: A solution C for H,c,k implies all edges of C have weight 0; therefore, every edge of C also is an edge in G. Therefore C is an HC in G.

NP-Complete Problems (Review)

A problem Q is ***NP**-complete* if Q belongs to *NP*, and for *every* problem R in ***NP***, R is polynomial reducible to Q.

Exptime (EXP) and Exptime-Complete Problems

- A problem Q is ***EXPTIME****-complete* if Q belongs to ***EXP***, and for *every* problem R in ***EXP***, R is polynomial time reducible to Q.
- Notice that although we don't know if NP is equal to P or not, we do know that EXPTIME-complete problems are not in P; it has been proven that these problems cannot be solved in polynomial time. These are harder than NP-Complete problems.

The First NP-Complete Problem

- Cook-Levin discovered the first NP-complete problem, known as the Satisfiability Problem (called SAT). The proof is complicated. We are skipping the proof (***use logic table to show the idea)***.

- SAT is the following problem: Given an expression using boolean variables p,q,r,… joined together using connectives AND, OR, NOT, is there a way to assign values "true" or "false" to the variables so that the expression evaluates to true?

  Example: Consider

  p AND NOT (q OR (NOT r)) →

  p AND NOT (q or (Not T))

  → T AND F (F) → T AND T

  → T

  Assigning T to p and to r and assigning F to q causes the expression to evaluate to T (i.e. "true").

  False of false is not False which is True

The *Satisfiability* (SAT) Problem

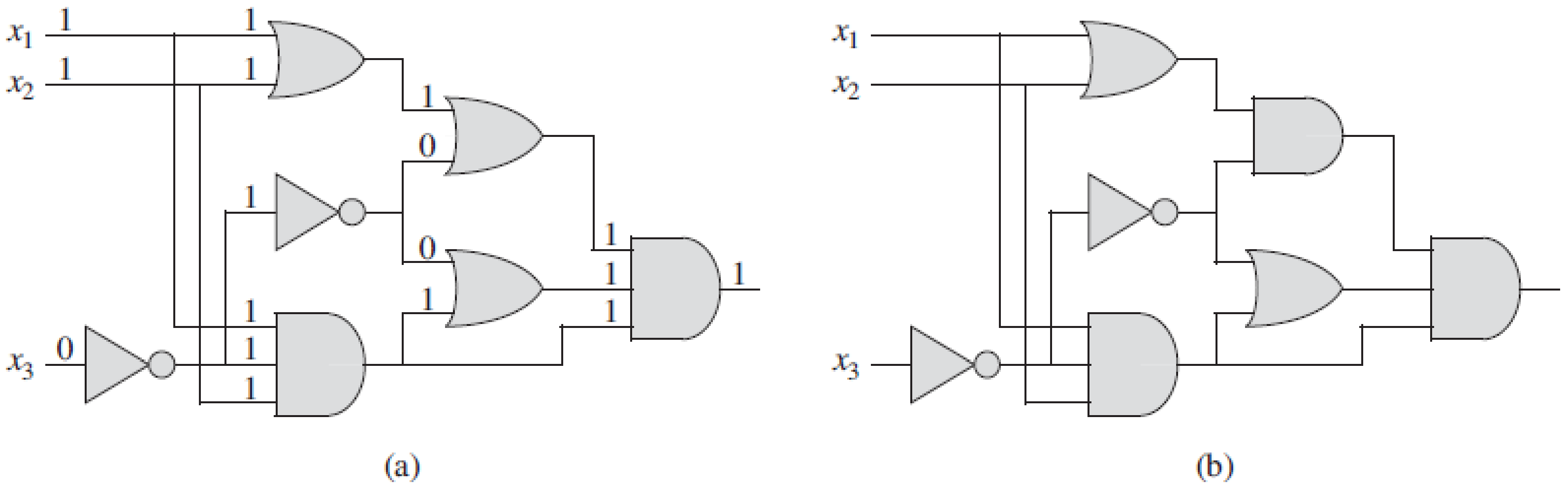


Figure 34.8 Two instances of the circuit-satisfiability problem. **(a)** The assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$ to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. **(b)** No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.

Other NP-Complete Problems

- Once a single NP-complete problem was discovered, others could be found by using the Cook-Levin result
- *Example*: VertexCover can be shown to be NP-complete. The main idea is to prove that SAT is polynomial reducible to VertexCover. Then: Given any NP problem Q, to show Q is polynomial reducible to VertexCover, observe:
  - Q is polynomial reducible to Sat
  - Sat is polynomial reducible to VertexCover
  - Therefore, Q is polynomial reducible to VertexCover

  (the last step requires verification: Exercise: if PROB1 is poly reducible to PROB2 and PROB2 is poly reducible to PROB3, then PROB1 is poly reducible to PROB3.)

More on Reducibility (From Corman)

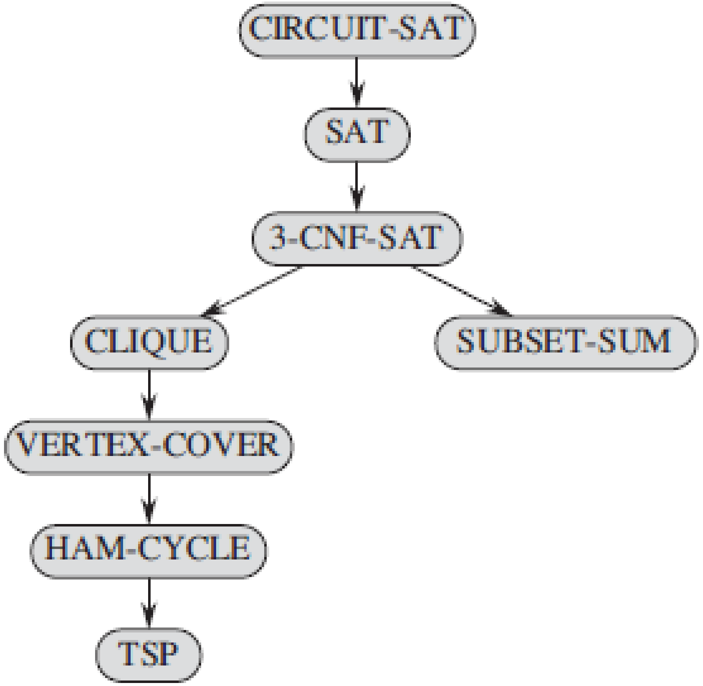


Figure 34.13 The structure of NP-completeness proofs in Sections 34.4 and 34.5. All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT.

HamiltonianCycle is NP-Complete

This is an outline of a proof that HamiltonianCycle is NP-Complete under the assumption that VertexCover is NP-complete:

- Recall that VertexCover is polynomial reducible to HamiltonianCycle.
- Therefore, since VertexCover is NP-complete, given any NP problem Q, Q is polynomial reducible to VertexCover
- Therefore, any such Q is polynomial reducible to HamiltonianCycle as well.

CSAT is *NP*-complete

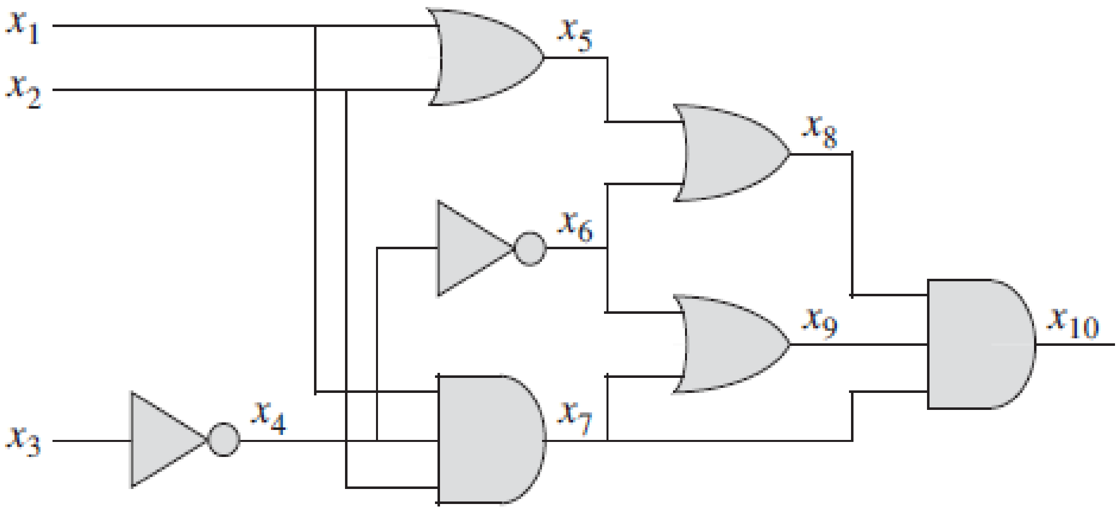


Figure 34.10 Reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit.

SAT is *NP*-complete

→ 'implies

$$\phi = x_{10} \wedge (x_4 \leftrightarrow \neg x_3)$$

x_4 implies not of x_3

$$\wedge\ (x_5 \leftrightarrow (x_1 \vee x_2))$$

→ new line, not dependent on the prev one

$$\begin{aligned}
&\wedge\ (x_6 \leftrightarrow \neg x_4) \\
&\wedge\ (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
&\wedge\ (x_8 \leftrightarrow (x_5 \vee x_6)) \\
&\wedge\ (x_9 \leftrightarrow (x_6 \vee x_7)) \\
&\wedge\ (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))\,.
\end{aligned}$$

Given a circuit C, it is straightforward to produce such a formula ϕ in polynomial time.

3-CNF is NP Complete

- The following Boolean Formula is in 3-CNF (Conjunction of Disjunctions) form:

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

- Any Boolean Expression can be converted to CNF form (next page)

3-CNF is NP Complete

- Thus the following formula can be converted to 3-CNF in Polynomial time

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2 \,. \tag{34.3}$$

- Thus, SAT can be reduced to 3-CNF in Polynomial time. Hence 3-CNF is NP-Complete as SAT is NP-Complete.

Solving One NP-Complete Problem Solves Them All

- Suppose Q is an NP-complete problem and someone finds a polynomial-time algorithm A that solves it in O(p(n)) time.
- Let R be any problem in NP. R is polynomial reducible to Q, with witness polynomial q(y).
- Polynomial-time algorithm to solve R: Given an instance I_R of R, create an instance I_Q of Q (in O(q(n)) time) that has a solution iff I_R does. Solve I_Q in O(p(q(n))) time. Output solution to I_R . The algorithm runs in O(q(n) + p(q(n))).

Main Point

The hardest NP problems are *NP-complete.* These require the highest degree of creativity to solve. However, if a polynomial-time algorithm is found for any one of them, then all NP problems will automatically be solved in polynomial time. This phenomenon illustrates the fact that the field of pure consciousness, the source of creativity, is itself a field of *infinite correlation* – "an impulse anywhere is an impulse everywhere".

What To Do with NP-Complete Problems?

There are many thousands of NP-complete problems, and a large percentage of these would be very useful if they could be implemented in a feasible way. But the known algorithms are too slow. What can be done?

Handling NP-Hard Problems

- Find a more efficient algorithm somehow
- The IsPrime breakthrough of 2002
- The technique of dynamic programming – examples: SubsetSum (Also Knapsack as shown before)
- Sometimes exponential running time is good enough
- Approximation algorithms
- Probabilistic algorithms
- Use hard problems as an advantage - cryptosystems

Options: Find a Better Algorithm

In the case of NP problems not known to be NP-complete, this is at least a reasonable goal.

Example: a polynomial time solution to the IsPrime problem was discovered in 2002; all known (deterministic) algorithms before 2002 were exponential

Example: Subset Sum Problem using Dynamic Programming (DP is Covered Before)

- The Subset Sum optimization problem says: We have set S = $\{s_0, s_1, \ldots, s_{n-1}\}$ of n positive integers and a non-negative integer k. Find a subset T of S so that the sum of the s_r in T is k.

$$\sum_{s_r \in T} s_r = k$$

Note. A solution to the optimization problem gives a solution to the decision problem

Main Observation

$S = \{1, 2, 3, 4\}$

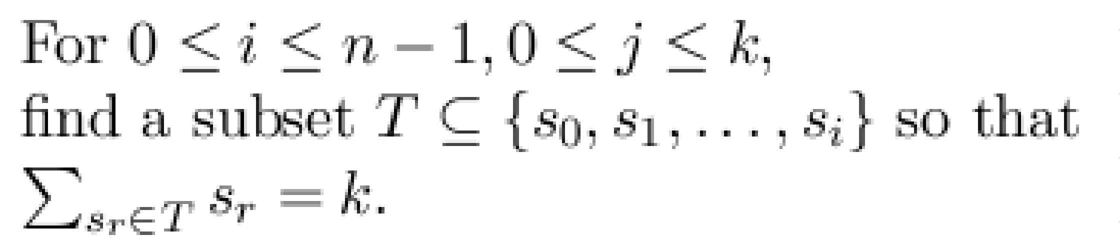
$k = 6$, $s_{n-1} = 4$

$T = \{1, 2, 3\}$

- Given an instance of SubsetSum: $S = \{s_0, s_1, \ldots, s_{n-1}\}$ and k. If T is a solution, then s_{n-1} is either in T or not.
- If s_{n-1} not in T, then T is a solution for $\{s_0, s_1, \ldots, s_{n-2}\}$, k.
- If s_{n-1} is in T, then $T - \{s_{n-1}\}$ is a solution for $\{s_0, s_1, \ldots, s_{n-2}\}$, $k - s_{n-1}$.

Applying Dynamic Programing to Solve SubsetSum

- There are only (k+1) * n problems to solve, namely:



For $0 \leq i \leq n-1, 0 \leq j \leq k$,
find a subset $T \subseteq \{s_0, s_1, \ldots, s_i\}$ so that
$\sum_{s_r \in T} s_r = k$.

Build a solution for bigger values of i and j using stored solutions for smaller values of i and j.

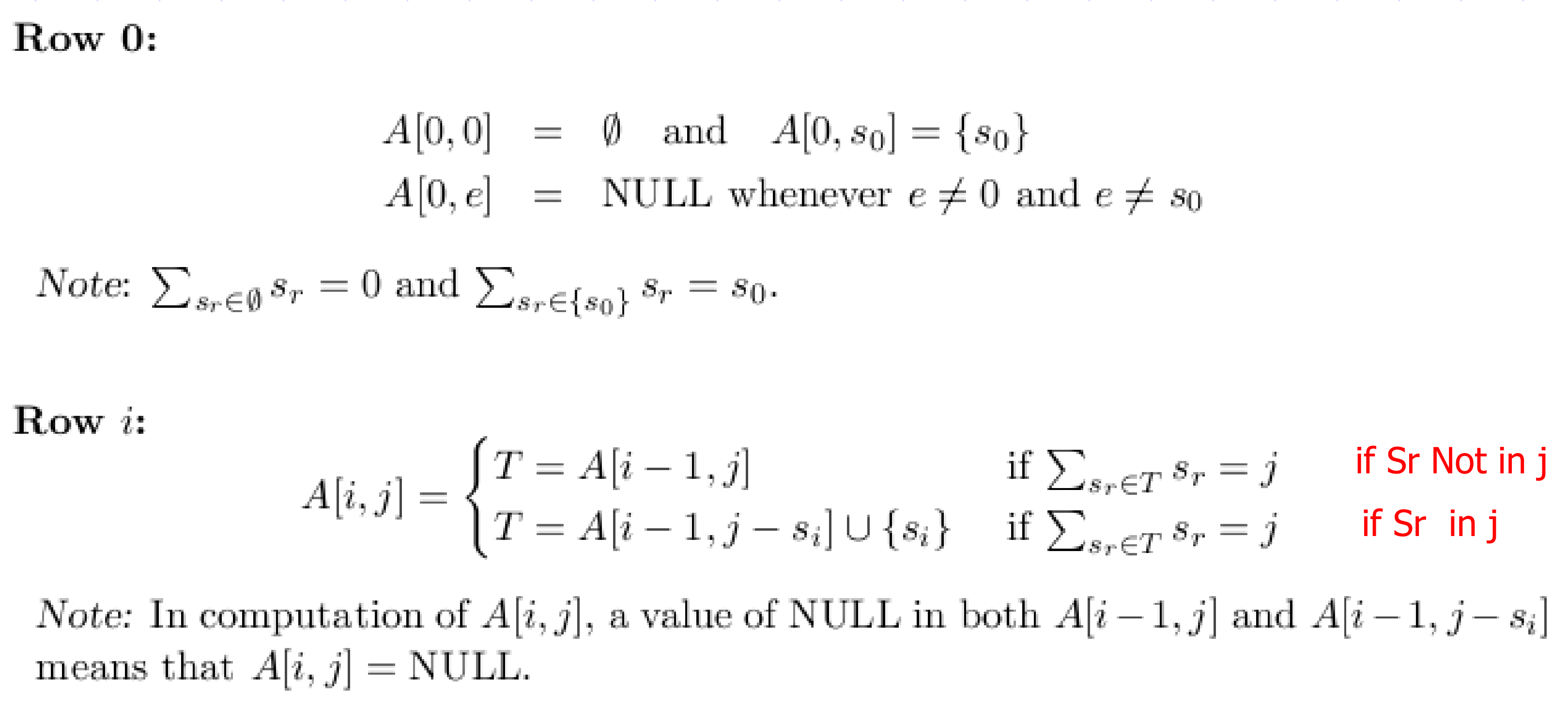
The Goal

Obtain a 2-dimensional array (a matrix) A so that

$$A[i,j] = \begin{cases} T & \text{where } T \subseteq \{s_0, s_1, \ldots, s_i\}, \sum_{s_r \in T} s_r = j \\ \text{NULL} & \text{if such a } T \text{ does not exist} \end{cases}$$

- If S contains values > k, we ignore them since they don’t contribute to the solution (computations for which j is too big are skipped – see the implementation in code)
- Fill row i = 0 first, then fill later rows based on values of earlier rows.
- Each cell requires O(1) time, so computation of A[n-1, k] requires O(kn) time.

Details



Row 0:

$$\begin{aligned} A[0,0] &= \emptyset \quad \text{and} \quad A[0,s_0] = \{s_0\} \\ A[0,e] &= \text{NULL whenever } e \neq 0 \text{ and } e \neq s_0 \end{aligned}$$

Note: $\sum_{s_r \in \emptyset} s_r = 0$ and $\sum_{s_r \in \{s_0\}} s_r = s_0$.

Row i:

$$A[i,j] = \begin{cases} T = A[i-1,j] & \text{if } \sum_{s_r \in T} s_r = j \\ T = A[i-1,j-s_i] \cup \{s_i\} & \text{if } \sum_{s_r \in T} s_r = j \end{cases}$$

if Sr Not in j

if Sr in j

Note: In computation of $A[i,j]$, a value of NULL in both $A[i-1,j]$ and $A[i-1,j-s_i]$ means that $A[i,j] = \text{NULL}$.