

Computational Intractability

Rangaballav Pradhan
ITER, SOADU

Input Size and Complexity

- Time complexity of algorithms:

➤ Polynomial time (efficient) vs. Exponential time (inefficient)

$f(n)$	$n=10$	30	50
n	0.00001 sec	0.00003 sec	0.00005 sec
n^5	0.1 sec	24.3 sec	5.2 mins
2^n	0.001 sec	17.9 mins	35.7 yrs

- Easy Problems vs Hard Problems
- Sometimes the dividing line between “easy” and “hard” problems is a fine one. For example,
 - Find the shortest path in a graph from X to Y (easy)
 - Find the longest path (with no cycles) in a graph from X to Y (hard)

Polynomial time solvable?

yes	probably no
shortest path	longest path
min cut	max cut
2-satisfiability	3-satisfiability
planar 4-colorability	planar 3-colorability
bipartite vertex cover	vertex cover
matching	3d-matching
primality testing	factoring
linear programming	integer linear programming

Classify problems according to computational complexity

- Motivation: is it possible to efficiently solve “hard” problems?
 - Efficiently solve means polynomial time solutions.
- Some problems have been proved that no efficient algorithms for them. For example, Turing’s Halting Problem.
- However, for many problems we cannot prove there exists no efficient algorithms, and at the same time, we cannot find one either-NP Complete.

Tractable and Intractable Problems

- A problem is tractable if there exists a polynomial bound algorithm that solves it.
- Worst-case growth rate can be bounded by a polynomial. i.e.,
 $T(n) = \theta(n^k)$
- A problem in CS is **intractable** if a computer has difficulty solving it
- A problem is intractable if it is not tractable
- Any algorithm with a growth rate not bounded by a polynomial. i.e.,
 $C^n, n^{\log n}, n!$, etc.

Three General Categories of Problems

1. Problems for which polynomial-time algorithms have been found (e.g., Searching, Sorting, etc.)
2. Problems that have been proven to be intractable (e.g., The Halting Problem, Tower of Hanoi, etc.)
3. Problems that have not been proven to be intractable, but for which polynomial-time algorithms have never been found (e.g., Traveling salesperson, Sum of subsets, etc.)

Abstract Problems

- An abstract problem Q is a binary relation on a set I of problem instances and a set S of problem solutions.
- Example: Shortest-Path: Given an unweighted, undirected graph $G = (V, E)$ and two vertices $x, y \in V$, the Shortest Path problem is the problem of finding the shortest path from x to y in G . That is, the path from x to y that requires the least number of edges.
- An instance i of Shortest-Path is a triple $\langle G, x, y \rangle$ consisting of a graph and two vertices. A solution s is a path in G .
- The abstract problem for Shortest-Path is the set of pairs (i, s) , such that s is a solution for instance i . That is, s is a sequence of vertices v_1, \dots, v_k in the graph such $v_1 = x$, $v_k = y$, $(v_i, v_{i+1}) \in E$ for $i = 1, \dots, k-1$, and k is the smallest number for which this is true.
- There are two distinct types of abstract problems: optimization problem and decision problems.

Optimization Problems

- An optimization problem is one in which you want to find, not just a solution, but the best solution.
- In an optimization problem, one or more objectives are given as Objective functions along with a set of constraints.
- $\text{minimize } f(x)$
subject to $g(x) \leq k$
 $h(x) \leq l$
- In complexity theory, the focus is on how the worst-case computing time grows as the problem size grows.
- Problem size can be roughly thought of as the number of variables and constraints in an optimization problem with (obviously the cost of evaluating the objective and constraints for a given choice of the variable also counts).

Decision problems

- Problem where the output is a simple “yes” or “no”
- Theory of NP-completeness is developed by restricting problems to decision problems
- Optimization problems can be transformed into decision problems
- Optimization problems are at least as hard as the associated decision problem
- If polynomial-time algorithm for the optimization problem is found, we would have a polynomial time algorithm for the corresponding decision problem

Example Problems

- Problem: Shortest Path
- Instance: A graph $G = (V, E)$, two vertices $x, y \in V$, and a positive integer k .
- Solution: A path from x to y .
- Decision Problem: Is there a solution of length at most k ?
- Related Optimization Problem: A path of shortest length from x to y .
- Traveling Salesman: For a given positive number d , is there a tour having length $\leq d$?
- 0-1 Knapsack: For a given profit P , is it possible to load the knapsack such that total weight $\leq W$?

The class P

- The set of all decision problems that can be solved by polynomial-time algorithms
- Decision versions of searching, shortest path, spanning tree, etc. belong to P.
- Do problems such as traveling salesperson and 0-1 Knapsack (no polynomial-time algorithm has been found), etc., belong to P?
- No one knows
- To know a decision problem is not in P, it must be proven it is not possible to develop a polynomial-time algorithm to solve it

Deterministic and Non-deterministic Algorithms

- A deterministic algorithm is the one that produces the same output for a problem instance each time the algorithm is run.
- A non-deterministic algorithm is a two-stage procedure that takes as its input an instance I of a decision problem and does the following:
 - Stage 1: Non-deterministic (“Guessing”) Stage: An arbitrary string S is generated that can be thought of as a candidate solution to the given instance I (could be sometimes, not a correct solution too).
 - Stage 2: Deterministic (“Verification”) Stage: A deterministic algorithm takes both I and S as its input and outputs yes if S represents a solution to instance I .
- A non-deterministic algorithm for the Traveling Salesman problem could be the one that inputs a complete weighted graph of n vertices, then (stage 1): generates an arbitrary sequence of n vertices and (stage 2): verifies whether or not that each vertex, except the starting vertex, is repeated only once in the sequence, and outputs a yes/no answer accordingly.

The class NP

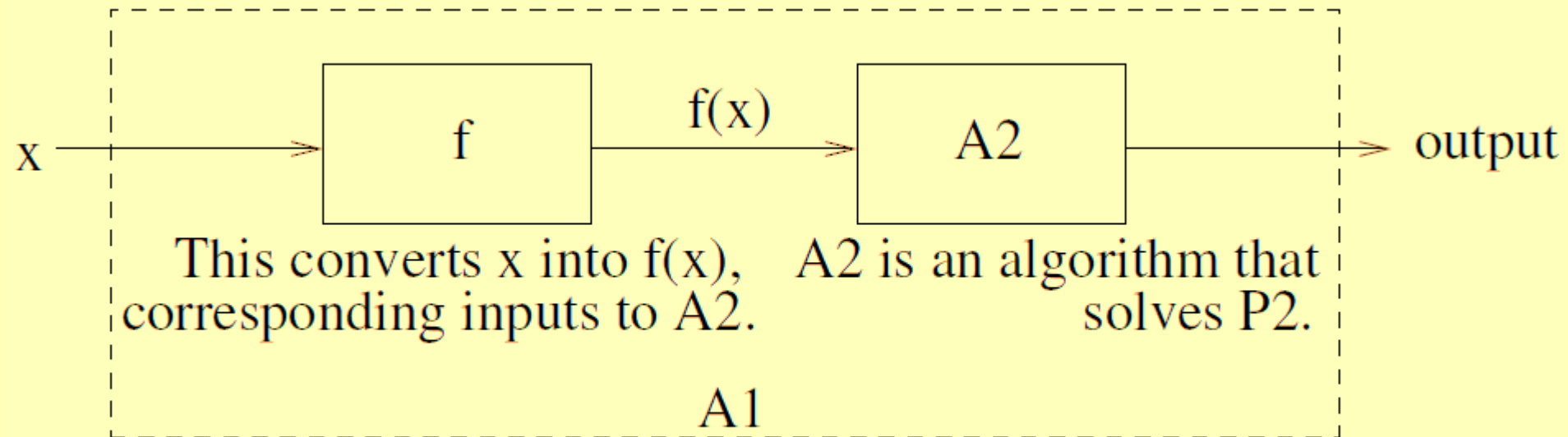
- Class NP (Non-deterministic Polynomial) is the class of decision problems that can be solved by non-deterministic polynomial algorithms.
- Note that all problems in Class P are in Class NP: We can replace the non-deterministic guessing of Stage 1 with the deterministic algorithm for the decision problem, and then in Stage 2, deterministically verify the correctness of the solution of the deterministic algorithm.
- In other words, $P \subseteq NP$. However, it is not clear whether $P = NP$ or $P \neq NP$.

Problem Reduction

- A problem P_1 is said to be reducible to another problem P_2 in polynomial time (denoted $P_1 \leq_P P_2$), if every instance of P_1 can be reduced to an instance of P_2 in polynomial time. So, if a polynomial time algorithm exists to solve an instance of problem P_2 , then we can say that there exists a polynomial time algorithm for P_1 .
- The key idea to demonstrate the hardness of a problem is that of a reduction, or translation, between two problems.
- Reductions are operations that convert one problem into another. In other words, a translation of instances from one type of problem to instances of another such that the answers are preserved is what is meant by a reduction.

Problem Reduction

- A problem P_1 is reducible to a problem P_2 if there is a function f that takes any input x to P_1 and transforms it to an input $f(x)$ of P_2 , such that the solution to P_2 on $f(x)$ is the solution to P_1 on x .



$A1$ is an algorithm that solves $P1$.

Problem Reduction

- Here's a simple example. Suppose we already have an algorithm for solving the following problem:

Problem 1: Squaring a Matrix

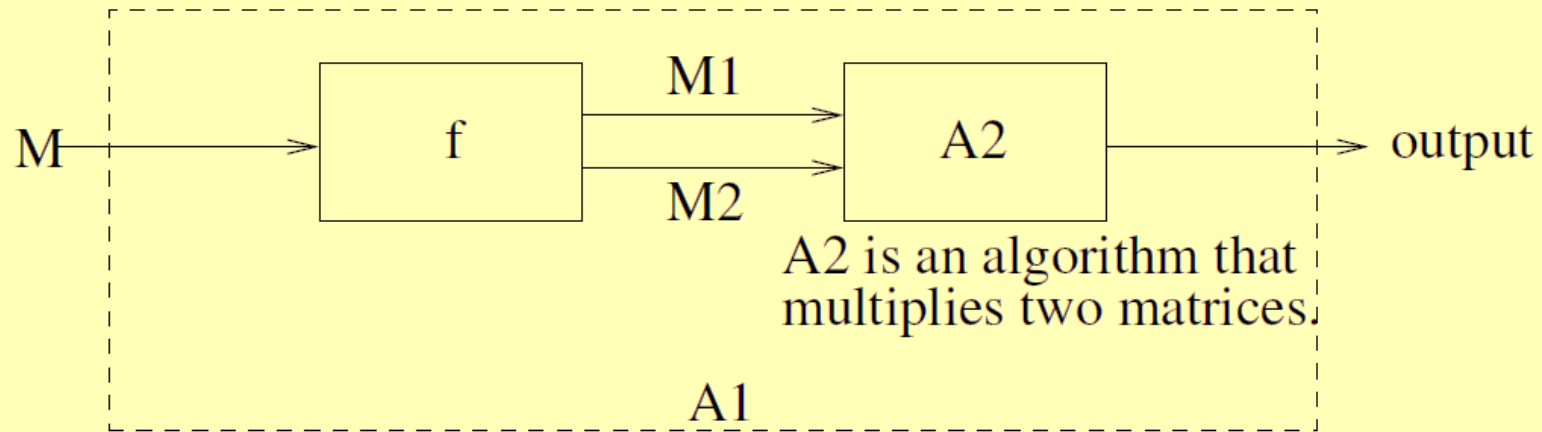
Input: A matrix, M .

Output: Result of squaring M .

Problem 2: Matrix Multiplication

Input: Two matrices, M_1 , M_2 .

Output: Result of multiplication of matrices M_1 and M_2 .



A_2 is an algorithm that multiplies two matrices.

A_1 is an algorithm that squares a matrix.

Polynomial Reduction

- Problem A is reducible, to problem B , if there a main program M to solve problem A that lacks only a procedure to solve problem B . The program M is called the reduction from A to B .
- A is polynomial-time reducible to B if M runs in polynomial time. A is linear-time reducible to B if M runs in linear time and makes at most a constant number of calls to the procedure for B .
- Assume that the reduction runs in time $P(n)$, exclusive of $R(n)$ recursive calls to B on an input size of $f(n)$. Then if one plugs in an algorithm for B that runs in time $B(n)$, one gets an algorithm for A that runs in time $A(n) = P(n) + R(n) \cdot B(f(n))$.

Polynomial Reduction

- Let A and B be decision problems. We say A is *many-to-one* reducible to B if there is reduction M from A to B of following special form:
 - M contains exactly one call to the procedure for B , and this call is the last step in M .
 - M returns what the procedure for B returns.
- Equivalently, A is *many-to-one* reducible to B if there exists a computable function f from instances of A to instances of B such that for all instances I of A it is the case that I has output 1 in problem A if and only if $f(I)$ has output 1 in problem B .
- Reductions can be used to both find efficient algorithms for problems, and to provide evidence that finding efficient algorithms for some problems will likely be difficult. We will mainly be concerned with the later use.

Using Reductions to Develop Algorithms

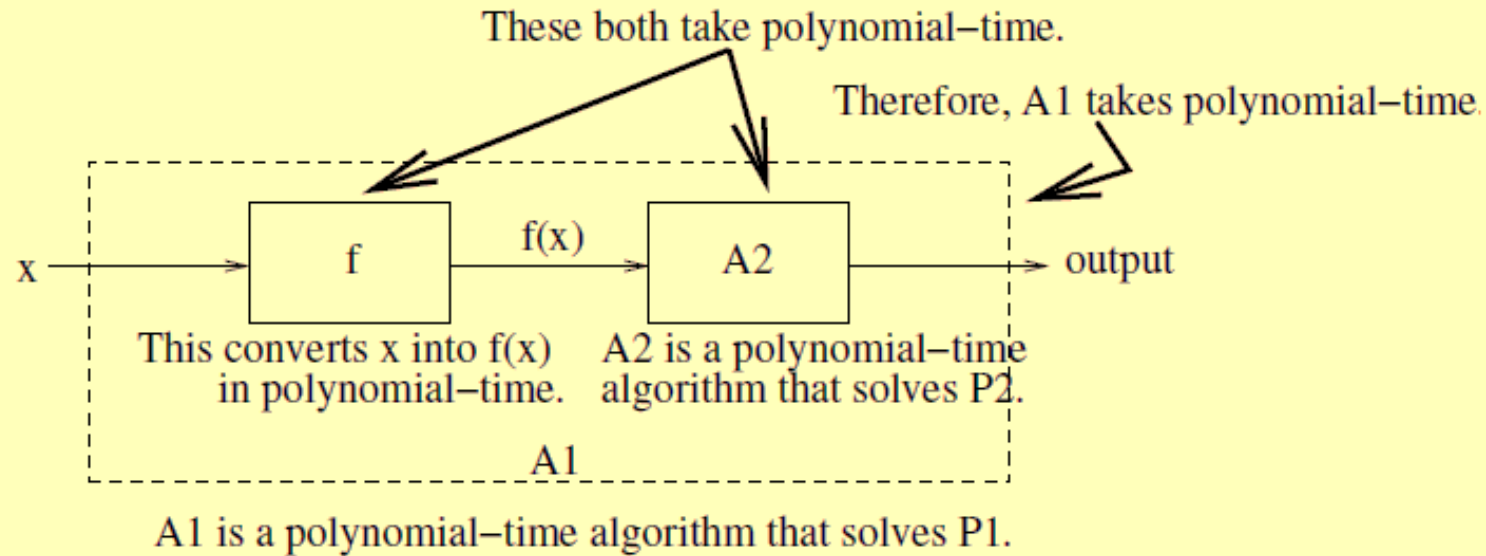
- Assume that A is some new problem that you would like to develop an algorithm for, and that B is some problem that you already know an algorithm for. Then showing $A \leq B$ will give you an algorithm for problem A .
- Example: Let A be the problem of determining whether n numbers are distinct. Let B be the sorting problem. One way to solve the problem A is to first sort the numbers (using a call to a black box sorting routine), and then in linear time to check whether any two consecutive numbers in the sorted order are equal. This give an algorithm for problem A with running time $O(n)$ plus the time for sorting. If one uses an $O(n \log n)$ time algorithm for sorting, like Mergesort, then one obtains an $O(n \log n)$ time algorithm for the element uniqueness problem.

Using Reductions to Examine hardness

- Suppose $P_1 \leq_P P_2$.
 - This rules out the possibility that P_1 is intractable while P_2 is tractable.
 - It also means if P_2 is tractable, then P_1 is tractable.
 - Equivalently (the contrapositive): if P_1 is proved to be intractable, then P_2 is also intractable.
 - Highly informally, it means that P_2 is ‘as hard as’ P_1 .

Using Reductions to Examine hardness

- **Theorem:** Suppose $P_1 \leq_P P_2$. If P_2 is tractable, then P_1 is tractable.
 - Proof: If P_2 tractable, then we will have a polynomial-time algorithm A_2 for solving P_2 . From this, we can construct a polynomial-time algorithm, A_1 , for solving P_1 , as follows:



- The conversion, f , takes polynomial-time and A_2 takes polynomial-time, so A_1 (which comprises f followed by A_2 in sequence) must take polynomial-time. Hence, P_1 would be tractable too.

Proving a problem to be tractable

- Suppose $P_1 \leq_P P_2$.
- To prove P_1 is tractable, we need to show:
 - P_2 is tractable.
 - The reduction f is polynomial.
- Example 1: Matrix multiplication and Matrix squaring.
 - We've already seen that Squaring a Matrix reduces to Matrix Multiplication. And, in fact, Squaring a Matrix \leq_P Matrix Multiplication.
 - If Matrix Multiplication is tractable.
 - What do you conclude about Squaring a Matrix and why?

Complement of a Decision Problem

- Consider a decision problem P . (It returns YES or NO.)
- The complement of P returns YES whenever P returns NO, and returns NO whenever P returns YES.
- Example: Input: An integer x and a finite-length list of integers, L .

Output: YES if x is a member of L and NO otherwise.

– Its complement is:

Input: An integer x and a finite-length list of integers,

Output: NO if x is a member of L and YES otherwise.

- Observation: Suppose P_2 is a tractable decision problem and that P_1 is its complement then P_1 is tractable.

Proving a problem to be intractable

- Suppose $P_1 \leq_P P_2$.
 - If P_2 is tractable, then P_1 is tractable.
- But we can also use the contrapositive of the above statement:
 - If P_1 is proved to be intractable, then P_2 is also intractable.
- Note: Reductions are a powerful tool for proving a problem to be non-computable. Suppose $P_1 \leq P_2$, If P_1 is non-computable, then P_2 is also non-computable.

Closest Pair Problem


- The closest pair problem asks to find the pair of numbers within a set that have the smallest difference between them. The corresponding decision problem can be formed as follows.
 - Input: A set S of n numbers, and threshold t .
 - Output: Is there a pair $s_i, s_j \in S$ such that $|s_i - s_j| \leq t$?
- The closest pair is a simple application of sorting, since the closest pair must be neighbors after sorting. This gives the following algorithm:

CloseEnoughPair(S, t)

Sort S .

Is $\min_{1 \leq i \leq n} |s_i - s_{i+1}| \leq t$?

Closest Pair Problem

- Given $S = \{70, 24, 61, 11, 35, 42, 53, 19\}$
- Sorted array $S = \{11, 19, 24, 35, 42, 53, 61, 70\}$


The diagram shows the sorted array $S = \{11, 19, 24, 35, 42, 53, 61, 70\}$ with blue brackets underneath indicating the distances between consecutive elements: 8 (between 11 and 19), 5 (between 19 and 24), 11 (between 24 and 35), 7 (between 35 and 42), 11 (between 42 and 53), 8 (between 53 and 61), and 9 (between 61 and 70).
- $\min_{1 \leq i \leq n} |s_i - s_{i+1}| = 5$. The closest pair of points are (19, 24).
- For the decision version of the problem, if $t = 8$, then output is *YES*.
- For $t = 7$, then output is *YES*.
- For $t = 6$, then output is *YES*.
- For $t = 5$, then output is *YES*.
- For $t = 4$, then output is *No*.

Closest Pair Problem

- The decision version captured what is interesting about the problem, meaning it is no easier than finding the actual closest pair.
- The complexity of this algorithm depends upon the complexity of sorting. Use an $O(n \log n)$ algorithm to sort, and it takes $O(n \log n + n)$ to find the closest pair.
- This reduction and the fact that there is an $\Omega(n \log n)$ lower bound on sorting does not prove that a close-enough pair must take $\Omega(n \log n)$ time in the worst case. Perhaps this is just a slow algorithm for a close-enough pair, and there is a faster one exist somewhere?
- On the other hand, if we knew that a close-enough pair required $\Omega(n \log n)$ time to solve in the worst case, this reduction would suffice to prove that sorting couldn't be solved any faster than $\Omega(n \log n)$ because that would imply a faster algorithm for a close-enough pair.

Least Common Multiple

- **Problem: Least Common Multiple (lcm)**

- Input: Two integers x and y .
- Output: Return the smallest integer m such that m is a multiple of both x and y .

- **Problem: Greatest Common Divisor (gcd)**

- Input: Two integers x and y .
- Output: Return the largest integer d such that d divides x and d divides y .

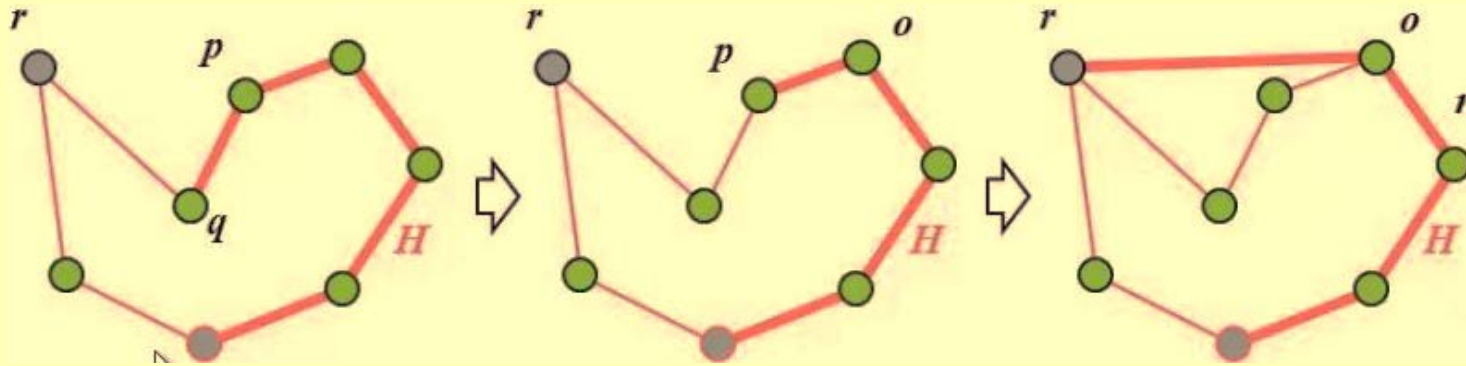
- For example, $\text{lcm}(24, 36) = 72$ and $\text{gcd}(24, 36) = 12$.
- Both problems can be solved easily after reducing x and y to their prime factorizations, but no efficient algorithm is known for factoring integers.

Least Common Multiple

- Euclid's algorithm gives an efficient way to solve greatest common divisor without factoring. It is a recursive algorithm that rests on two observations.
 - First, if $b|a$, then $\gcd(a, b) = b$. i.e., if b divides a , then $a = bk$ for some integer k , and thus $\gcd(bk, b) = b$.
 - Second, if $a = bt + r$ for integers t and r , then $\gcd(a, b) = \gcd(b, r)$.
- Since $x \cdot y = \text{lcm}(x, y) \cdot \gcd(x, y) \longrightarrow \text{lcm}(x, y) = (x \cdot y) / \gcd(x, y)$.
- This observation, coupled with Euclid's algorithm, provides an efficient way to compute least common multiple, namely
- LeastCommonMultiple (x, y)
 - Return $(x \cdot y / \gcd(x, y))$.

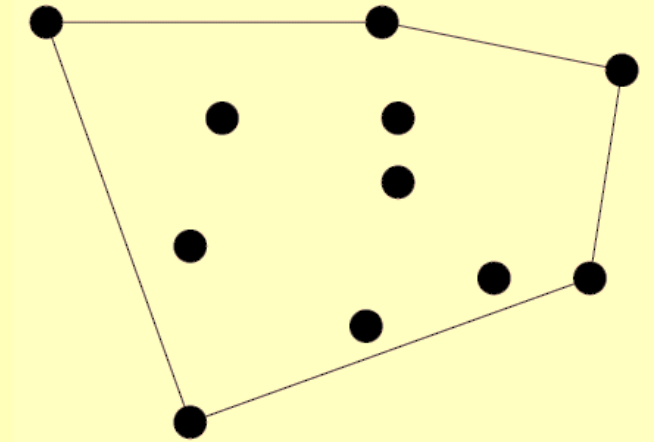
Convex Hull

- A polygon is convex if the straight line segment drawn between any two points inside the polygon P must lie completely within the polygon.



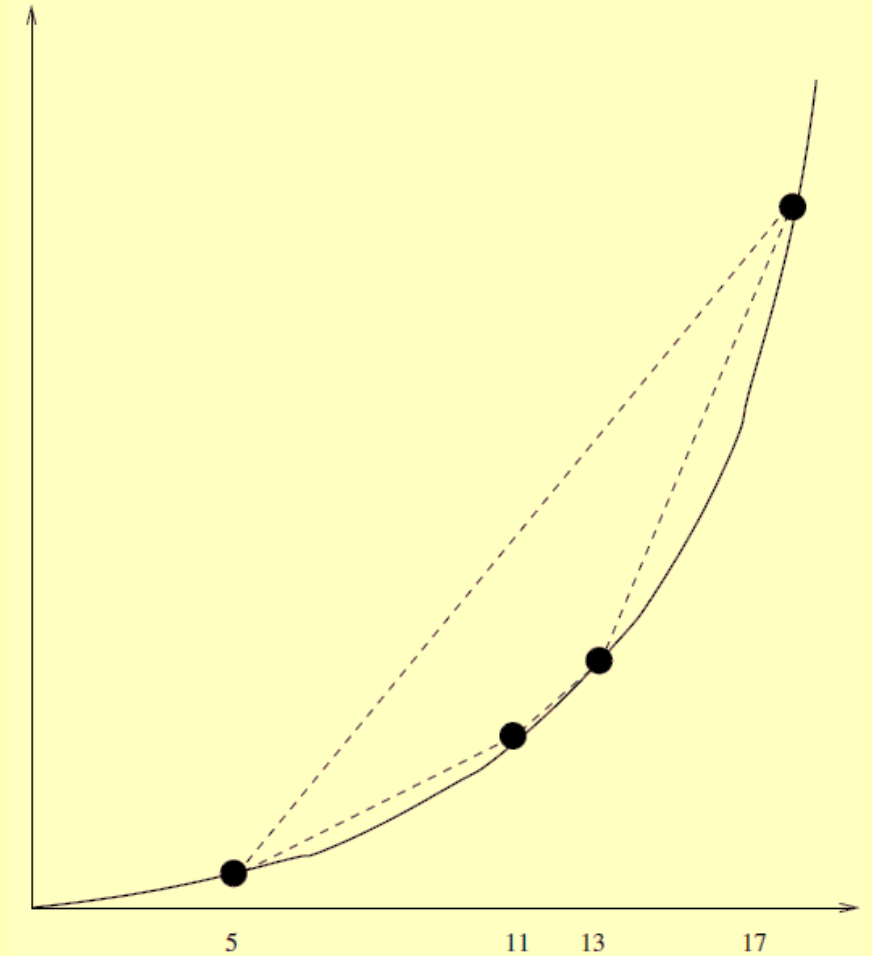
- **Problem: Convex Hull**

- Input: A set S of n points in the plane.
- Output: Find the smallest convex polygon containing all the points of S .
- The Sorting problem can be reduced to Convex Hull.



Convex Hull

- We must translate each number to a point by mapping x to (x, x^2) .
- It means each integer is mapped to a point on the parabola $y = x^2$.
- Since this parabola is convex, every point must be on the convex hull.
- Furthermore, since neighboring points on the convex hull have neighboring x values, the convex hull returns the points sorted by the x -coordinate—i.e., the original numbers.



Convex Hull

- **Sort(S)**

For each $i \in S$, create point (i, i^2) .

Call subroutine **Convex-Hull** on this point set.

From the leftmost point in the hull,

read off the points from left to right.

- Creating and reading off the points takes $O(n)$ time.
- The Sorting has lower bound of $\Omega(n \lg n)$. If we could compute convex hull in better than $n \lg n$, this reduction implies that we could sort faster than $\Omega(n \lg n)$, which violates our lower bound. Thus, convex hull must take $\Omega(n \lg n)$ as well!
- Observation: Any $O(n \lg n)$ convex hull algorithm also gives us a complicated but correct $O(n \lg n)$ sorting algorithm as well.

Elementary Hardness Reductions

- Till now, we have studied reductions between pairs of problems for which efficient algorithms exist.
- However, we are mainly concerned with using reductions to prove hardness of the problems.
- If $P_1 \leq_P P_2$:
 - A fast algorithm for P_2 implies a fast algorithm for P_1
 - On the other hand, if we can prove that P_1 is hard, it would imply that P_2 is also hard.

Problem: The Traveling Salesman Problem (TSP)

Input: A weighted graph G .

Output: Which tour $\{v_1, v_2, \dots, v_n\}$ minimizes $\sum_{i=1}^{n-1} d[v_i, v_{i+1}] + d[v_n, v_1]$?

Hamiltonian Cycle and TSP

- The Hamiltonian cycle problem seeks a tour that visits each vertex of a given graph exactly once.
- **Problem: Hamiltonian Cycle**
- Input: An unweighted graph G .
- Output: Does there exist a simple tour that visits each vertex of G without repetition?
- Hamiltonian cycle has some obvious similarity to the traveling salesman problem. Both problems seek a tour that visits each vertex exactly once.
- There are also differences between the two problems. TSP works on weighted graphs, while Hamiltonian cycle works on unweighted graphs.

Hamiltonian Cycle and TSP

- We can reduce the *Hamiltonian Cycle problem* to *TSP* as follows.

```
HamiltonianCycle( $G = (V, E)$ )
```

```
    Construct a complete weighted graph  $G' = (V', E')$  where  $V' = V$ .
```

```
     $n = |V|$ 
```

```
    for  $i = 1$  to  $n$  do
```

```
        for  $j = 1$  to  $n$  do
```

```
            if  $(i, j) \in E$  then  $w(i, j) = 1$  else  $w(i, j) = 2$ 
```

```
    Return the answer to Traveling-Salesman-Decision-Problem( $G', n$ ).
```

- The translation from unweighted to weighted graph is easily performed in $O(n^2)$ time.
- This translation is designed to ensure that the answers of the two problems will be identical. If the graph G has a Hamiltonian cycle $\{v_1, \dots, v_n\}$, then this exact same tour will correspond to n edges in E' , each with weight 1. This gives a TSP tour in G' of weight exactly n .

Hamiltonian Cycle and TSP

- We can reduce the Hamiltonian Cycle problem to TSP as follows.

```
HamiltonianCycle( $G = (V, E)$ )
```

```
    Construct a complete weighted graph  $G' = (V', E')$  where  $V' = V$ .
```

```
     $n = |V|$ 
```

```
    for  $i = 1$  to  $n$  do
```

```
        for  $j = 1$  to  $n$  do
```

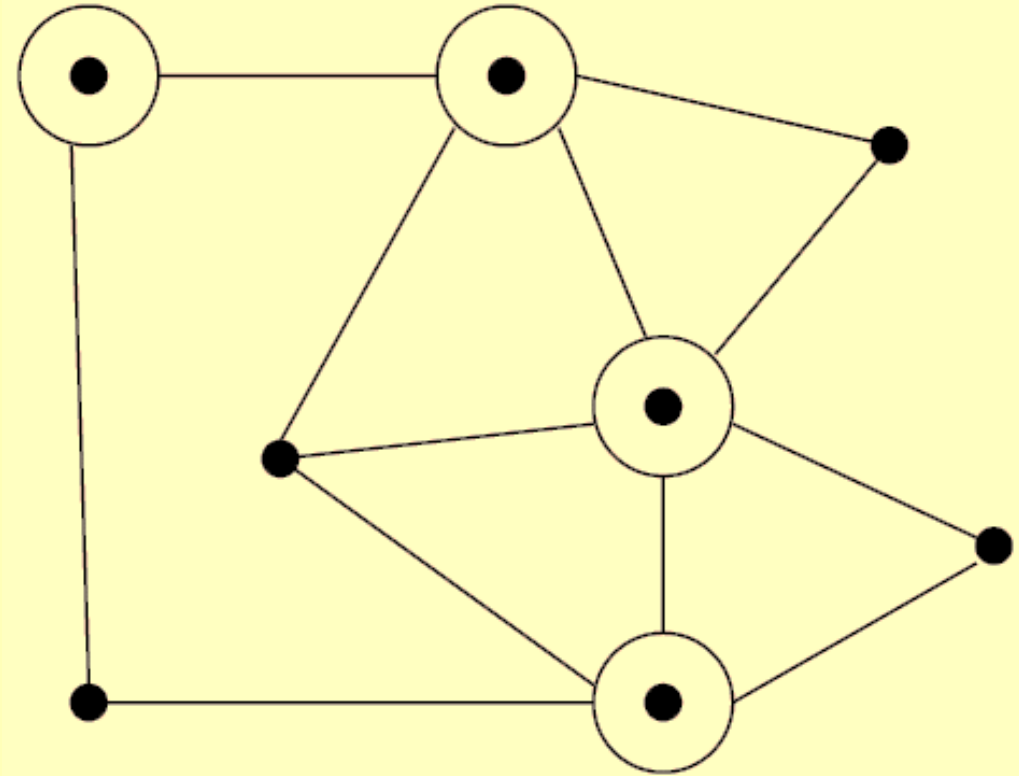
```
            if  $(i, j) \in E$  then  $w(i, j) = 1$  else  $w(i, j) = 2$ 
```

```
    Return the answer to Traveling-Salesman-Decision-Problem( $G', n$ ).
```

- If G does not have a Hamiltonian cycle, then there can be no such TSP tour in G' because the only way to get a tour of cost n in G would be to use only edges of weight 1, which implies a Hamiltonian cycle in G .
- The Hamiltonian cycle problem is known to be hard which implies that TSP is hard, at least as hard as the Hamiltonian cycle. This can be concluded from this reduction.

Independent Set and Vertex Cover

- The vertex cover problem finds a small set of vertices that contacts each edge in a graph.
- **Problem: Vertex Cover**
- Input: A graph $G = (V, E)$ and integer $k \leq |V|$.
- Output: Is there a subset S of at most k vertices such that every $e \in E$ contains at least one vertex in S ?
- The trivial approach to find a vertex cover of a graph is to consider the cover that consists of all the vertices. But the goal is to find the smallest possible set of vertices.



Independent Set and Vertex Cover

- A set S of vertices of a graph G is independent if there are no edges (x, y) where both $x \in S$ and $y \in S$. This means, there are no edges between any two vertices in independent set.
- The maximal independent set decision problem is formally defined:
- **Problem: Independent Set**
- Input: A graph G and integer $k \leq |V|$.
- Output: Does there exist an independent set of k vertices in G ?
- Both vertex cover and independent set are problems that revolve around finding specific subsets of vertices: the first with representatives of every edge, the second with no edges.

Independent Set and Vertex Cover

- If S is the vertex cover of G , the remaining vertices $S - V$ must form an independent set, because if an edge had both vertices in $S - V$, then S could not have been a vertex cover. This gives us a reduction between the two problems:

VertexCover(G, k)

$$G' = G$$

$$k' = |V| - k$$

Return the answer to IndependentSet(G', k')

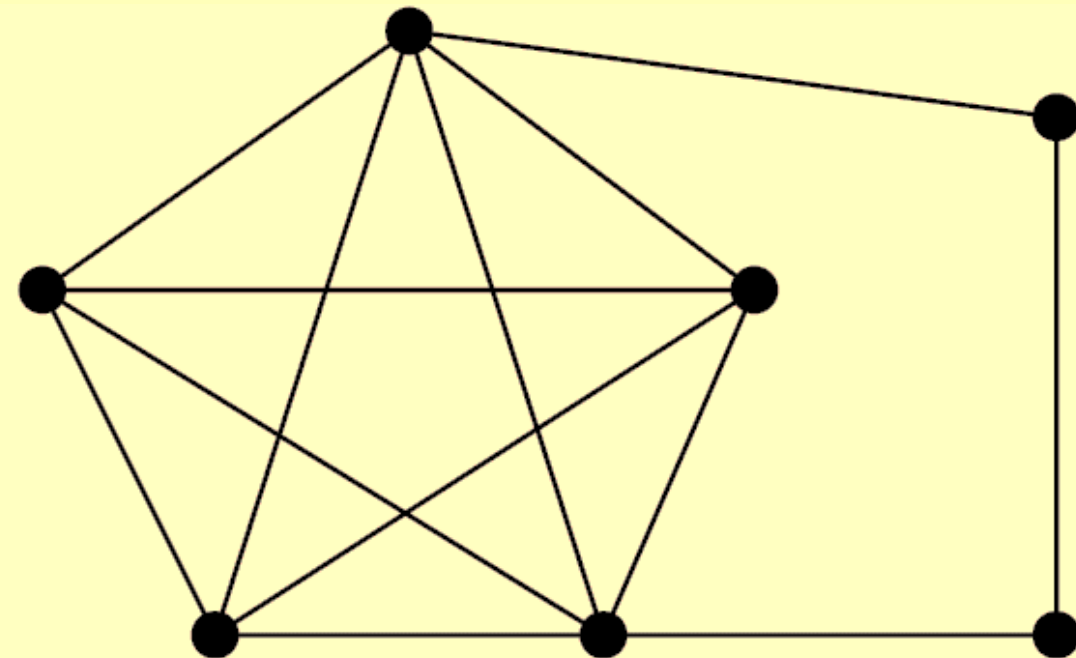
- This simple reduction shows that the two problems are identical. We transform the input, not the solution.
- This reduction shows that the hardness of vertex cover implies that independent set must also be hard. It is easy to reverse the roles of the two problems in this particular reduction, thus proving that both problems are equally hard.

The clique and Independent Set

- A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E . In other words, a clique is a complete subgraph of G .
- The size of a clique is the number of vertices it contains. The clique problem is the optimization problem of finding a clique of maximum size in a graph.

- **Problem: Maximum Clique**

- Input: A graph $G = (V, E)$ and integer $k \leq |V|$.
- Output: Does the graph contain a clique of k vertices; i.e. , is there a subset $S \subset V$, where $|S| \leq k$, such that every pair of vertices in S defines an edge of G ?



The clique and Independent Set

- In the independent set problem, we looked for a subset S with no edges between two vertices of S . This contrasts with clique, where we insist that there always be an edge between two vertices. A reduction between these problems follows by reversing the roles of edges and non-edges—an operation known as complementing the graph:

IndependentSet(G, k)

Construct a graph $G' = (V', E')$ where $V' = V$, and

For all (i, j) not in E , add (i, j) to E'

Return the answer to Clique(G', k)

- These last two reductions provide a chain linking of three different problems. The hardness of clique is implied by the hardness of independent set, which is implied by the hardness of vertex cover.
- By constructing reductions in a chain, we link together pairs of problems in implications of hardness.

Vertex cover and Set cover

SET-COVER. Given a set U of n elements, a collection $S = \{S_1, S_2, \dots, S_m\}$ of subsets of U , and an integer k , does there exist a collection of at most k of these subsets whose union is equal to all of U ?

$$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$$

$$S_a = \{ 3, 7 \}$$

$$S_b = \{ 2, 4 \}$$

$$S_c = \{ 3, 4, 5, 6 \}$$

$$S_d = \{ 5 \}$$

$$S_e = \{ 1 \}$$

$$S_f = \{ 1, 2, 6, 7 \}$$

$$k = 2$$

a set cover instance

Vertex cover and Set cover

SET-COVER. Given a set U of n elements, a collection $S = \{S_1, S_2, \dots, S_m\}$ of subsets of U , and an integer k , does there exist a collection of at most k of these subsets whose union is equal to all of U ?

Example: Given the universe $U = \{1, 2, 3, 4, 5, 6, 7\}$ and the following sets, which is the minimum size of a set cover?

$$U = \{1, 2, 3, 4, 5, 6, 7\}$$

$$S_a = \{1, 4, 6\}$$

$$S_b = \{1, 6, 7\}$$

$$S_c = \{1, 2, 3, 6\}$$

$$S_d = \{1, 3, 5, 7\}$$

$$S_e = \{2, 6, 7\}$$

$$S_f = \{3, 4, 5\}$$

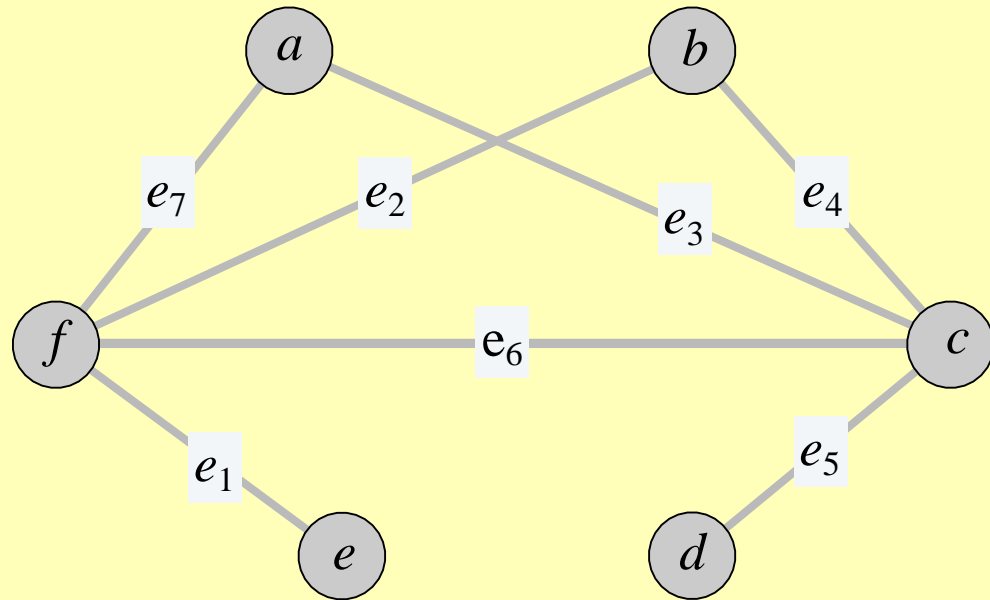
Vertex cover and Set cover

Claim. VERTEX-COVER \leq_p SET-COVER.

Pf. Given a VERTEX-COVER instance $G = (V, E)$ and k , we construct a SET-COVER instance (U, S, k) that has a set cover of size k iff G has a vertex cover of size k .

Construction.

- Universe $U = E$.
- Include one subset for each node $v \in V$: $S_v = \{e \in E : e \text{ incident to } v\}$.



vertex cover instance
($k = 2$)

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$	
$S_a = \{ 3, 7 \}$	$S_b = \{ 2, 4 \}$
$S_c = \{ 3, 4, 5, 6 \}$	$S_d = \{ 5 \}$
$S_e = \{ 1 \}$	$S_f = \{ 1, 2, 6, 7 \}$

set cover instance
($k = 2$)

Satisfiability

- To analyze the hardness of all problems using reductions, we must start with a single problem that is absolutely, certifiably, undeniably hard. The mother of all NP-complete problems is a logic problem named **SATISFIABILITY**:
- **Problem: Satisfiability**
- Input: A set X of Boolean variables and a set C of clauses over X .
- Output: Does there exist a satisfying truth assignment for C over X — i.e. , a way to set the variables x_1, x_2, \dots, x_n true or false so that each clause contains at least one true literal/term?
- Each variable can take the value 0 or 1 (equivalently, “false” or “true”).
- A clause is simply a disjunction of distinct terms $t_1 \vee t_2 \vee \dots \vee t_n$ where each $t_i \in \{x_1, x_2, \dots, x_n, \overline{x_1}, \overline{x_2}, \dots, \overline{x_n}\}$. We say the clause has length l if it contains l terms.

Satisfiability

- A truth assignment for C is an assignment of the value 0 or 1 to each x_i ; in other words, it is a function $v : X \rightarrow \{0, 1\}$.
- The assignment v implicitly gives \bar{x}_i the opposite truth value from x_i .
- An assignment satisfies a clause C if it causes C to evaluate to 1; this is equivalent to requiring that at least one of the terms in C should receive the value 1.
- An assignment satisfies a collection of clauses C_1, C_2, \dots, C_k if it causes all of the C_i to evaluate to 1; in other words, if it causes the conjunction $C_1 \wedge C_2 \wedge \dots \wedge C_k$ to evaluate to 1.
- In this case, we will say that v is a satisfying assignment with respect to C_1, C_2, \dots, C_k ; and that the set of clauses C_1, C_2, \dots, C_k is satisfiable.

Satisfiability

- Example: Suppose we have the three clauses $(x_1 \vee \overline{x_2})$, $(\overline{x_1} \vee \overline{x_3})$, $(x_2 \vee \overline{x_3})$
- Then the truth assignment v that sets all variables to 1 ($x_1=1, x_2=1, x_3=1$) is not a satisfying assignment, because it does not satisfy the second clause.
- But the truth assignment v' that sets all variables to 0 ($x_1=0, x_2=0, x_3=0$) is a satisfying assignment.
- So, the Satisfiability Problem, also referred to as SAT can be stated as:
- Given a set of clauses C_1, C_2, \dots, C_k over a set of variables $X = \{x_1, x_2, \dots, x_n\}$, does there exist a satisfying truth assignment?

3-Satisfiability

- There is a special case of SAT that will turn out to be equivalently difficult and is somewhat easier to think about; this is the case in which all clauses contain exactly three terms (corresponding to distinct variables). We call this problem 3-Satisfiability or 3-SAT:
- Given a set of clauses C_1, C_2, \dots, C_k , each of length 3, over a set of variables $X = \{x_1, x_2, \dots, x_n\}$, does there exist a satisfying truth assignment?
- **Problem: 3-Satisfiability (3-SAT)**
- Input: A collection of clauses C where each clause contains exactly 3 terms/literals, over a set of Boolean variables X .
- Output: Is there a truth assignment to X such that each clause is satisfied?

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

3-Satisfiability

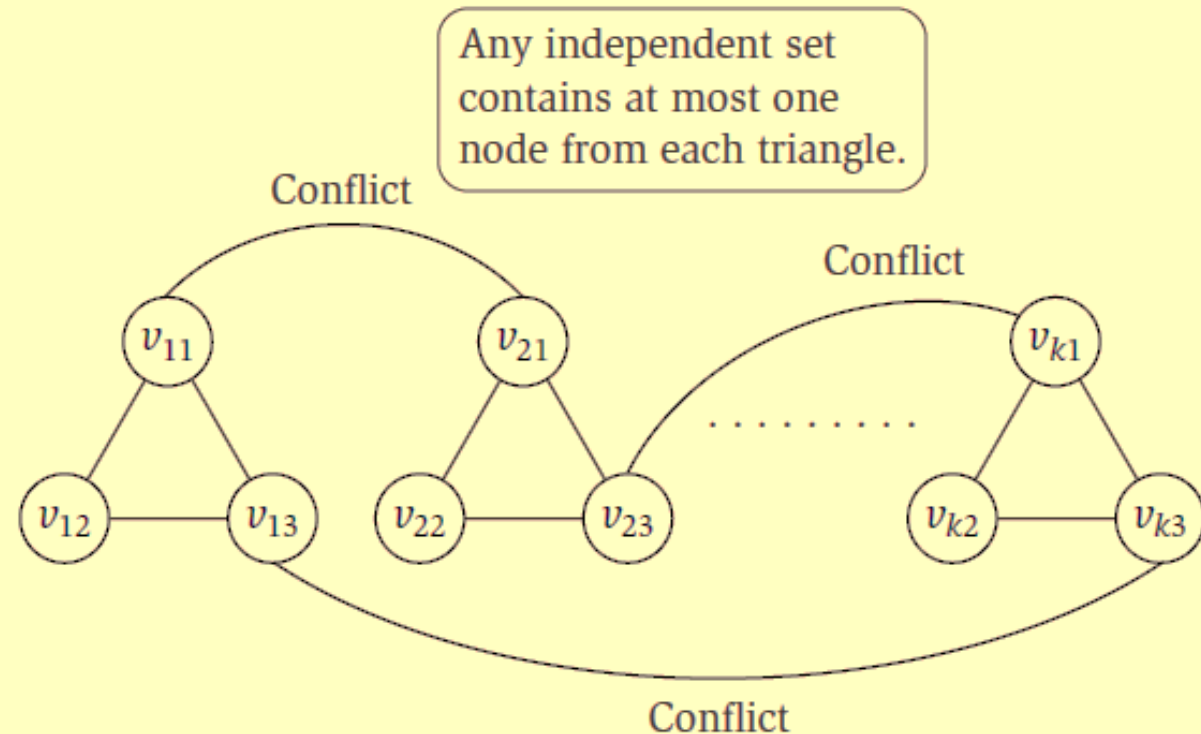
- Since 3-SAT is a restricted case of satisfiability, the hardness of 3-SAT implies that satisfiability is hard. The converse isn't true, since the hardness of general satisfiability might depend upon having long clauses.
- Satisfiability and 3-Satisfiability are really fundamental combinatorial search problems; they contain the basic ingredients of a hard computational problem in very “bare-bones” fashion.
- We have to make n independent decisions (the assignments for each x_i) so as to satisfy a set of constraints. There are several ways to satisfy each constraint in isolation, but we have to arrange our decisions so that all constraints are satisfied simultaneously.

Reducing 3-SAT to Independent Set

- 3-SAT \leq_P Independent Set.
- 3-SAT is about setting Boolean variables in the presence of constraints, while Independent Set is about selecting vertices in a graph. To solve an instance of 3-SAT using a black box for Independent Set, we need a way to encode all these Boolean constraints in the nodes and edges of a graph, so that satisfiability corresponds to the existence of a large independent set.
- There are two conceptually distinct ways of thinking about an instance of 3-SAT.
 - We have to make an independent 0/1 decision for each of the n variables, and we succeed if we manage to achieve one of three ways of satisfying each clause.
 - We have to choose one term from each clause, and then find a truth assignment that causes all these terms to evaluate to 1, thereby satisfying all clauses.

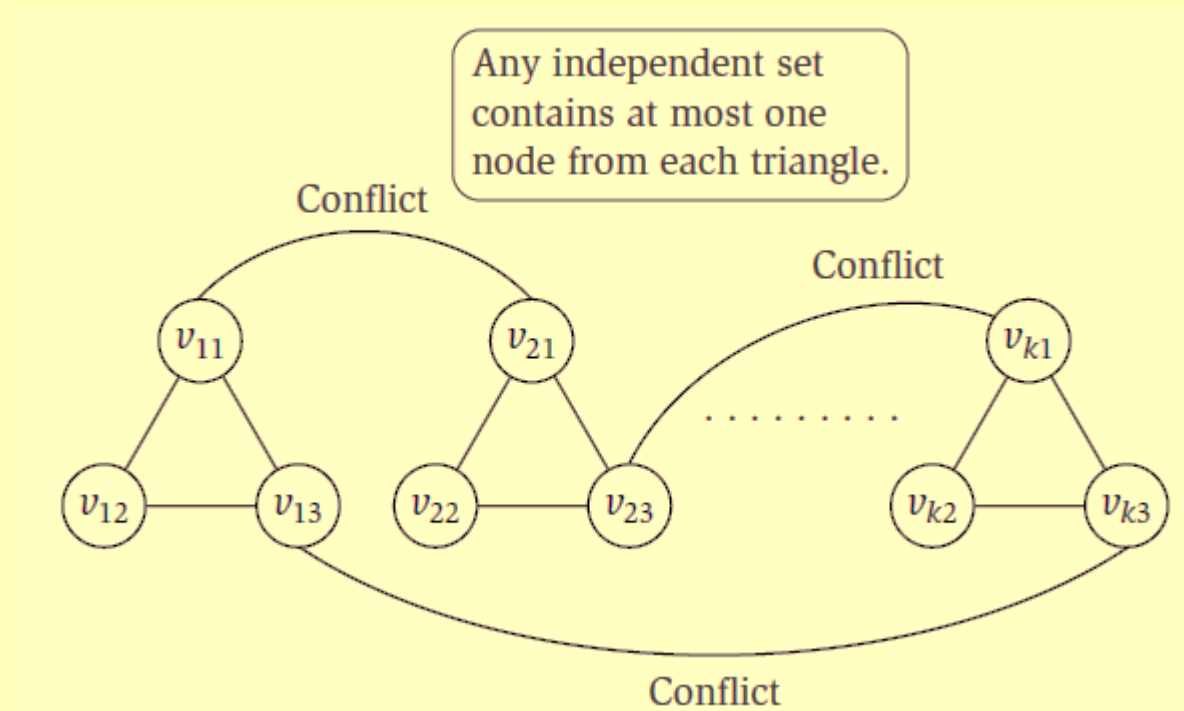
Reducing 3-SAT to Independent Set

- We succeed if we can select a term from each clause in such a way that no two selected terms “conflict”.
- we say that two terms conflict if one is equal to a variable x_i and the other is equal to its negation \bar{x}_i . If we avoid conflicting terms, we can find a truth assignment that makes the selected terms from each clause evaluate to 1.
- **Reduction.** First, construct a graph $G = (V, E)$ consisting of $3k$ nodes grouped into k triangles. That is, for each clause C_i , we construct three vertices v_{i1}, v_{i2}, v_{i3} joined to one another by edges.



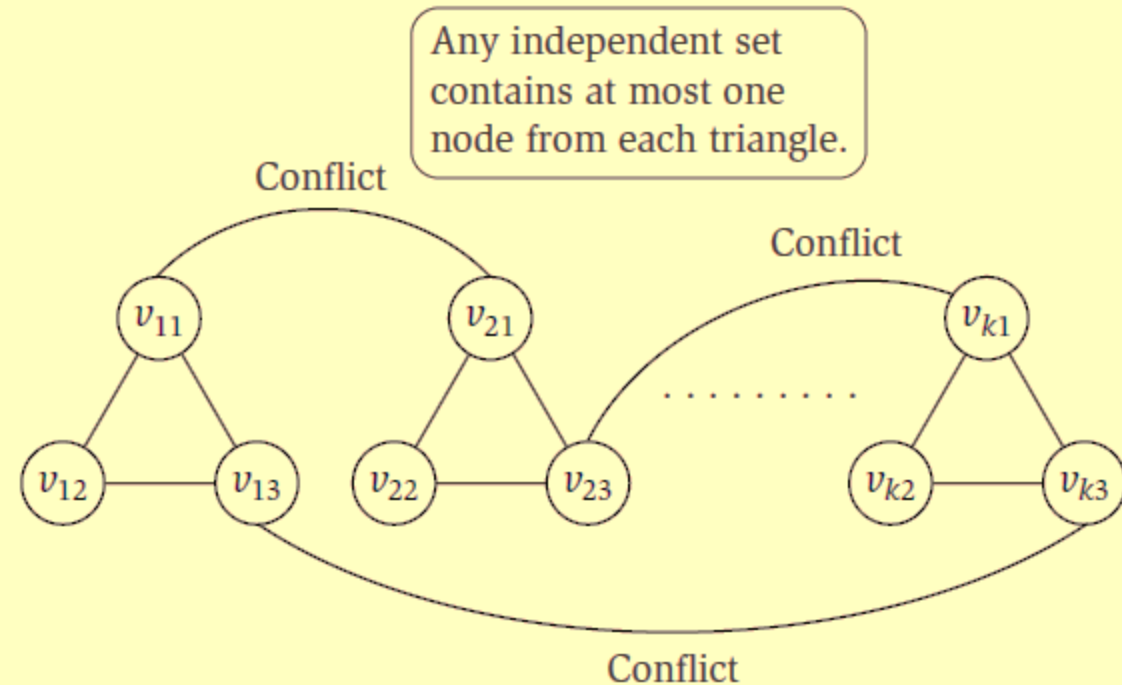
Reducing 3-SAT to Independent Set

- For each pair of vertices whose labels correspond to terms that conflict, we add an edge between them.
- **Claim:** The original 3-SAT instance is satisfiable if and only if the graph G we have constructed has an independent set of size at least k .



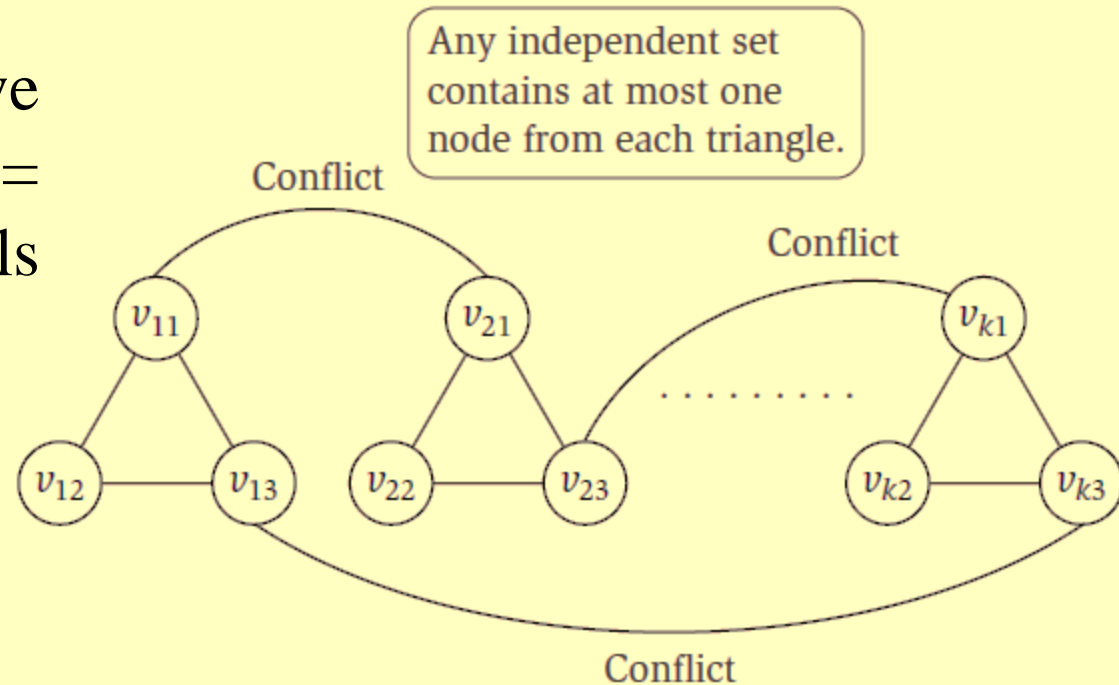
Reducing 3-SAT to Independent Set

- If the 3-SAT instance is satisfiable, then each triangle in our graph contains at least one node whose label evaluates to 1.
- Let S be a set consisting of one such node from each triangle. We claim S is independent; because if there were an edge between two nodes $u, v \in S$, then the labels of u and v would have to conflict; but this is not possible, since they both evaluate to 1.
- Conversely, suppose our graph G has an independent set S of size at least k . Then, first of all, the size of S is exactly k , and it must consist of one node from each triangle.



Reducing 3-SAT to Independent Set

- Now, we claim that there is a truth assignment v for the variables in the 3-SAT instance with the property that the labels of all nodes in S evaluate to 1.
- For each variable x_i , if neither x_i nor \bar{x}_i appears as a label of a node in S , then we arbitrarily set $v(x_i) = 1$.
- If x_i appears as a label of a node in S , we set $v(x_i) = 1$, and otherwise we set $v(x_i) = 0$. By constructing v in this way, all labels of nodes in S will evaluate to 1.



Transitivity of Reductions

- \leq_P is a transitive relation.
- If $X \leq_P Y$ and $Y \leq_P Z$, then $X \leq_P Z$.
- Given a black box for Z , we show how to solve an instance of X .
- $X \leq_P Y$ implies that the instance of X can be reduced in polynomial time to get solved by an algorithm A_Y for Y . That means, the input x to X can be mapped to the input of Y as $f(x)$.
- Again, $Y \leq_P Z$ implies that the instance of Y can be reduced in polynomial time to get solved by an algorithm A_Z for Z . That means, the input $f(x)$ to Y can be mapped to the input of Z as $g(f(x))$.

Transitivity of Reductions

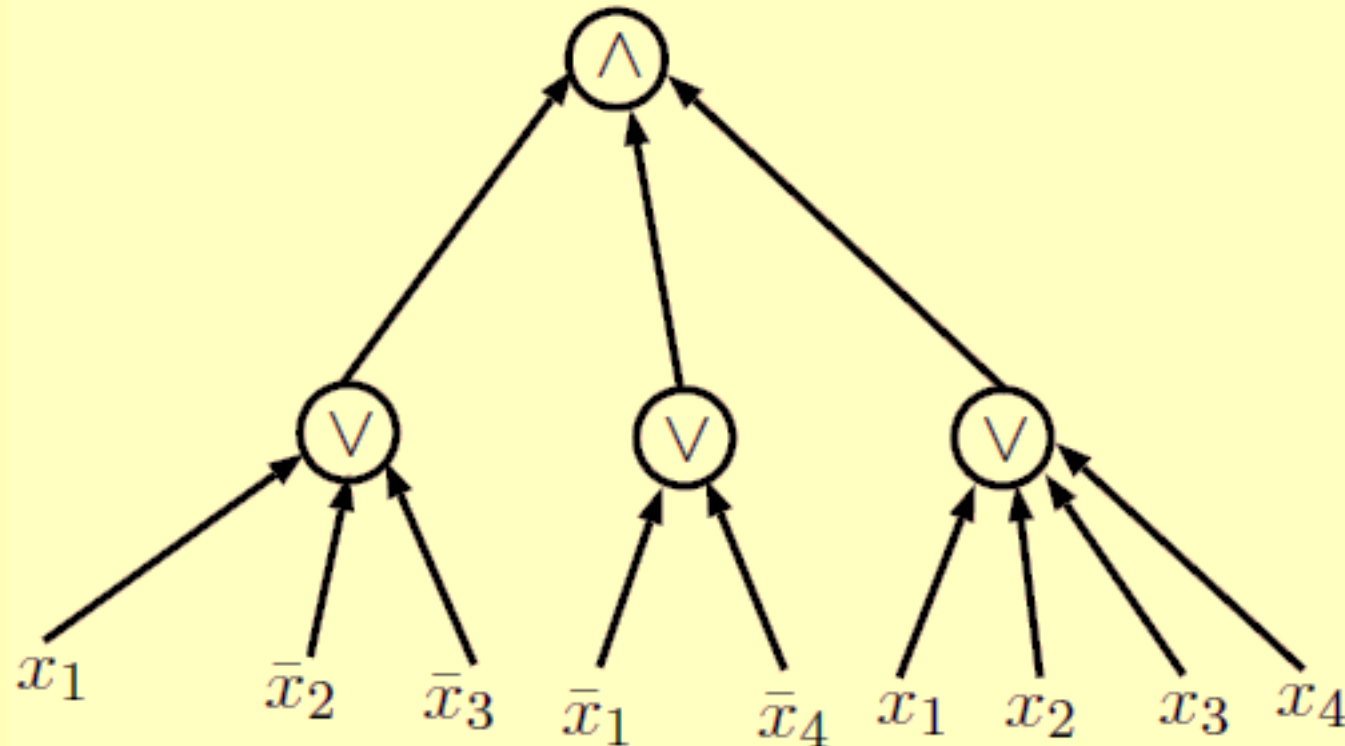
- If $X \leq_P Y$ and $Y \leq_P Z$, then $X \leq_P Z$.
- Now, we can just compose the two algorithms A_Y and A_Z . We run the algorithm for X using a black box A_Y for Y ; but each time the A_Y is called, we simulate it in a polynomial number of steps using A_Z that solves instances of Z .
- So, the reduction from X to Z = the reduction from X to Y + the reduction from Y to Z .
- Ex. $3\text{-SAT} \leq_P \text{INDEPENDENT-SET} \leq_P \text{VERTEX-COVER}$

Formula-SAT or CNF-SAT

- The view of SAT that we have discussed till now is basically a Boolean Formula over a set of variables that evaluates to 1.
- A Boolean formula ϕ over the variables $\{x_1, x_2, \dots, x_n\}$ consists of the variables and logical conjunction (\wedge), disjunction (\vee) and negation (\neg). If A and B are Boolean formulae, then so are $A \wedge B$ and $A \vee B$.
- A Boolean formula ϕ is satisfiable if $\exists z \in \{0, 1\}^n$ such that $\phi(z) = 1$.
- A Boolean formula over variables $\{x_1, x_2, \dots, x_n\}$ is in Conjunctive Normal Form (CNF), if it is AND of OR's of variables or their negations.
- The SAT we discussed is known as the CNF-SAT.
- If k is the length of each clause in the formula, then it is called k -CNF-SAT.

Formula-SAT or CNF-SAT

- For example, $(x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_3 \vee x_4)$ is a CNF formula.
- The figure shows a CNF formula $(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_4) \wedge (x_1 \vee x_2 \vee x_3 \vee x_4)$, which is satisfied by $x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 0$, but is not satisfied by $x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 1$.



Circuit-SAT

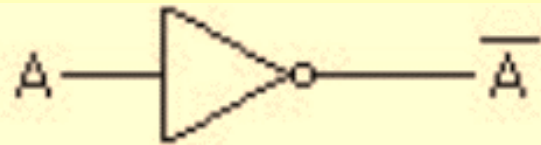
- The circuit satisfiability problem (CIRCUIT-SAT) is the circuit analogue of SAT.
- Given a Boolean circuit C , is there an assignment to the variables that causes the circuit to output 1?
- **Boolean circuits:** For any natural number n , an n input, single output Boolean circuit is a directed acyclic graph, which has n Boolean inputs (also called sources) denoted by x_1, x_2, \dots, x_n , one output (also called sink) which denoted by $C(x)$, where $x = (x_1, x_2, \dots, x_n) \in \{0, 1\}^n$.
- All the non-source vertices are called gates. These gates compute the Boolean functions AND, OR and NOT, denoted by \wedge , \vee and \neg respectively.

Circuit-SAT

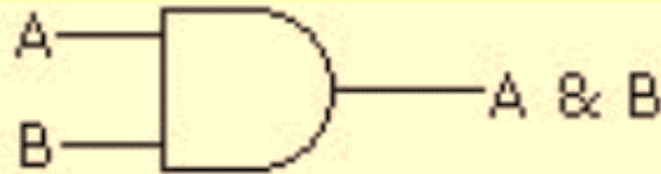
- The gates \wedge and \vee has fanin 2, and the \neg gate has fanin 1.
- Wires of the circuits connect the gates and inputs by carrying the Boolean value 0 or 1.
- The size of a circuit C , denoted by $|C|$, is the number of gates in it.
- A Boolean circuit C over x_1, x_2, \dots, x_n accepts $\alpha \in \{0, 1\}^n$ if and only if $C(\alpha) = 1$.
- **Circuit-SAT:** The Circuit-SAT problem is stated as follows. Given a circuit C , does there exist an input $\alpha \in \{0, 1\}^n$ such that $C(\alpha) = 1$?

Circuit-SAT

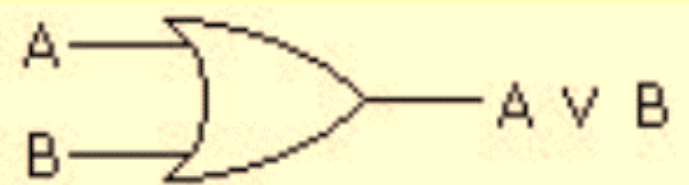
- **Circuit-SAT:** The Circuit-SAT problem is stated as follows. Given a circuit C , does there exist an input $\alpha \in \{0, 1\}^n$ such that $C(\alpha) = 1$?



A	NOT A
0	1
1	0



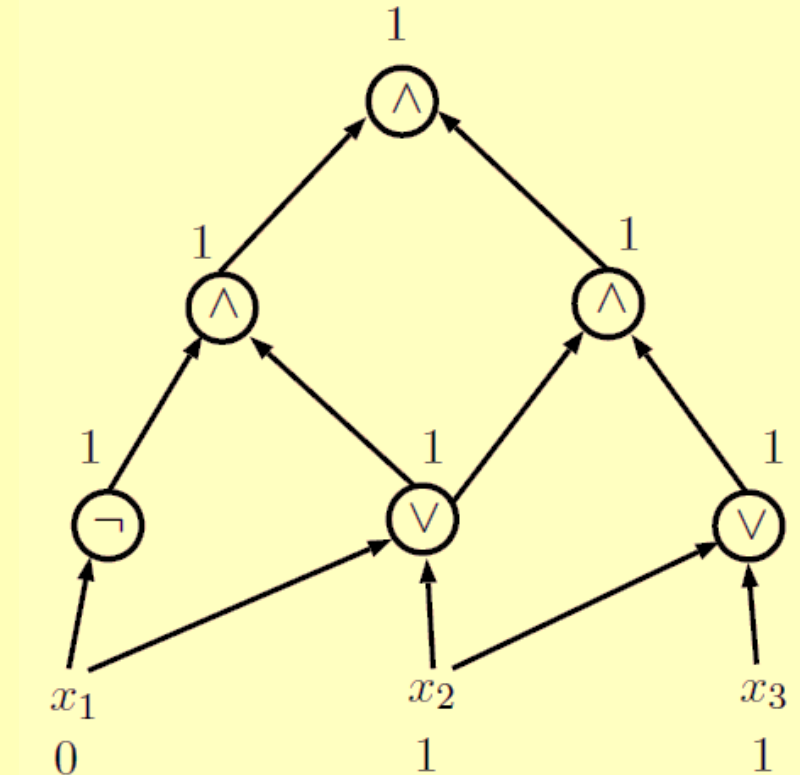
A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1



A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Circuit-SAT

- **Circuit-SAT** The Circuit-SAT problem is stated as follows. Given a circuit C , does there exist an input $\alpha \in \{0, 1\}^n$ such that $C(\alpha) = 1$?
- For example, the Boolean circuit in the following figure accepts $x_1 = 0$, $x_2 = 1$, $x_3 = 1$.
- **SAT:** This is a special case of circuit SAT, where the circuit represents a CNF formula, which has:
 - an unbounded fanin \wedge at top
 - followed by unbounded fanin \vee
 - followed by literals.



Circuit-SAT \leftrightarrow CNF-SAT

- **3-CNF-SAT \leq_p Circuit-SAT:** Given 3-CNF formula ϕ with n variables and k clauses, create a Circuit C .
- Inputs to C are the n boolean variables x_1, x_2, \dots, x_n
- Use NOT gate to generate literal \bar{x}_i for each variable x_i
- For each clause $(t_1 \vee t_2 \vee t_3)$ use two OR gates to mimic formula
- Combine the outputs for the clauses using AND gates to obtain the final output
- Example: $\phi = (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (x_1 \vee x_3 \vee x_4) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee x_4)$

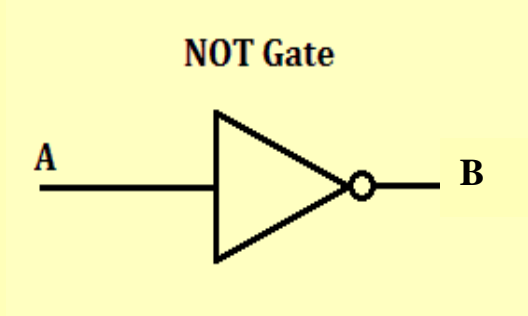
Circuit-SAT \leftrightarrow CNF-SAT

- **Circuit-SAT \leq_p CNF-SAT:** The Circuit-SAT problem is stated as follows. Given a circuit C , does there exist an input $\alpha \in \{0, 1\}^n$ such that $C(\alpha) = 1$?
- For the given circuit C , identify the inputs x_1, x_2, \dots, x_n and the gates.
- Assign a variable x_j as the output of each gate.
- The inputs to the SAT constitute the set X of Boolean variables. i.e., $X = \{x_1, x_2, \dots, x_n\} \cup \{x_j\}$.
- Convert each gate into a “*if and only if*” formula. (ϕ_k is the formula for gate k)
- Formulate the Boolean formula ϕ by combining the final output with the formulae for each gate as follows:

$$\phi = x_f \wedge \phi_1 \wedge \phi_2 \dots \wedge \phi_m$$

Circuit-SAT \leftrightarrow CNF-SAT

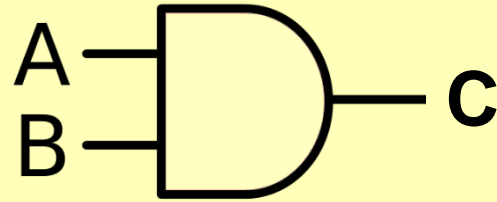
- **Circuit-SAT \leq_p CNF-SAT:** The Circuit-SAT problem is stated as follows. Given a circuit C , does there exist an input $\alpha \in \{0, 1\}^n$ such that $C(\alpha) = 1$?
- Convert each gate into a “*if and only if*” formula.
- The NOT gates can be reduced as follows:



- $\varphi = (B \Leftrightarrow \bar{A})$
 $= (\bar{B} \vee \bar{A}) \wedge (B \vee A)$

Circuit-SAT \leftrightarrow CNF-SAT

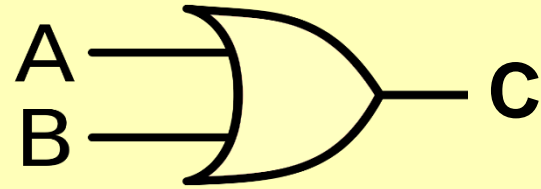
- The AND gates can be reduced as follows:



- $\varphi = (C \Leftrightarrow (A \wedge B))$
 $= (\bar{C} \vee (A \wedge B)) \wedge (C \vee \overline{(A \wedge B)})$
 $= (\bar{C} \vee A) \wedge (\bar{C} \vee B) \wedge (C \vee \bar{A} \vee \bar{B})$

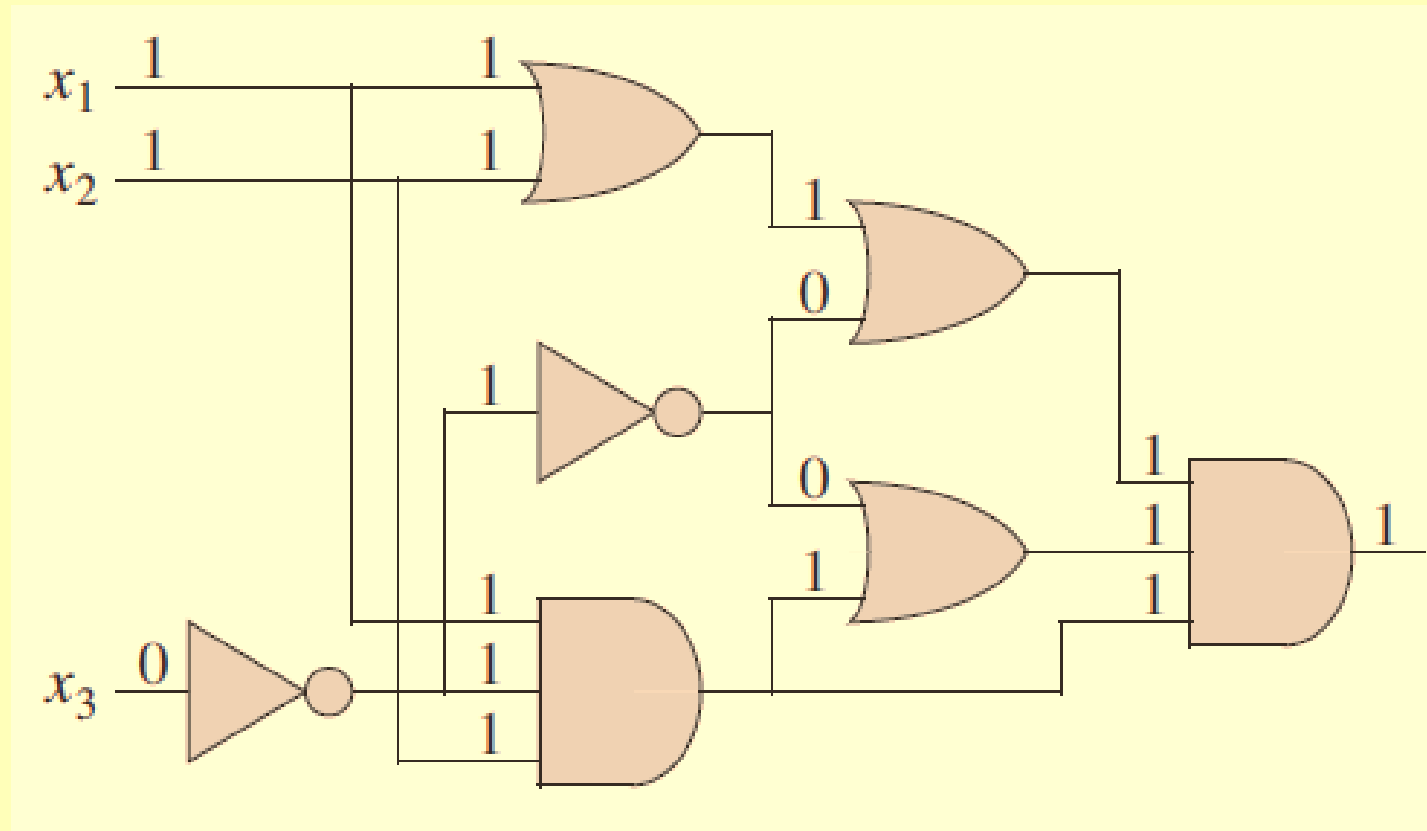
Circuit-SAT \leftrightarrow CNF-SAT

- The OR gates can be reduced as follows:



- $$\begin{aligned}\varphi &= (C \Leftrightarrow (A \vee B)) \\ &= (\bar{C} \vee (A \vee B)) \wedge (C \vee \overline{(A \vee B)}) \\ &= (\bar{C} \vee A \vee B) \wedge (C \vee (\bar{A} \wedge \bar{B})) \\ &= (\bar{C} \vee A \vee B) \wedge (C \vee \bar{A}) \wedge (C \vee \bar{B})\end{aligned}$$

Circuit-SAT \leftrightarrow CNF-SAT



Reduction from SAT to 3-SAT

- The reduction takes an arbitrary SAT instance φ as input, and transforms it to a 3-SAT instance φ' , such that satisfiability is preserved, i.e., φ' is satisfiable if and only if φ is satisfiable.
- Recall that a SAT instance is an AND of some clauses, and each clause is OR of some terms/literals. A 3-SAT instance is a special type of SAT instance in which each clause has exactly 3 literals.
- The reduction replaces each clause in φ with a set of clauses, each having exactly three literals.
- Assume that φ involves n variables x_1, x_2, \dots, x_n . The new formula φ' will have some new variables in addition to the x_i s.

Reduction from SAT to 3-SAT

- We now describe how we replace each clause in φ' . Let C is an arbitrary clause in φ .
 - case 1: C has one term
 - case 2: C has two terms
 - case 3: C has three terms (Keep it unchanged)
 - case 4: C has more than three terms

Reduction from SAT to 3-SAT

- **case 1:** C has one term
 - Let C consist of a single literal l . l is either x_i or \bar{x}_i for some i .
 - Let z_1 and z_2 be two new variables. We replace C by the following clause Z :
 - $Z = (l \vee z_1 \vee z_2) \wedge (l \vee \bar{z}_1 \vee z_2) \wedge (l \vee z_1 \vee \bar{z}_2) \wedge (l \vee \bar{z}_1 \vee \bar{z}_2)$

l	z_1	z_2	$(l \vee z_1 \vee z_2)$	$(l \vee \bar{z}_1 \vee z_2)$	$(l \vee z_1 \vee \bar{z}_2)$	$(l \vee \bar{z}_1 \vee \bar{z}_2)$	Z
T	T	T	T	T	T	T	T
T	T	F	T	T	T	T	T
T	F	T	T	T	T	T	T
T	F	F	T	T	T	T	T
F	T	T	T	T	T	F	F
F	T	F	T	F	T	T	F
F	F	T	T	T	F	T	F
F	F	F	F	T	T	T	F

Reduction from SAT to 3-SAT

- **case 2:** C has two terms
 - Let $C = l_1 \vee l_2$, where each of l_1 and l_2 is either x_i or \bar{x}_i for some i .
 - Let z_1 be a new variable. We replace C by the following clause Z :
 - $Z = (l_1 \vee l_2 \vee z_1) \wedge (l_1 \vee l_2 \vee \bar{z}_1)$

l_1	l_2	z_1	$C=l_1 \vee l_2$	$(l_1 \vee l_2 \vee z_1)$	$(l_1 \vee l_2 \vee \bar{z}_1)$	Z
T	T	T	T	T	T	T
T	T	F	T	T	T	T
T	F	T	T	T	T	T
T	F	F	T	T	T	T
F	T	T	T	T	T	T
F	T	F	T	T	T	T
F	F	T	F	T	F	F
F	F	F	F	F	T	F

Reduction from SAT to 3-SAT

- **case 4:** C has k terms ($k > 3$)
 - Let $C = l_1 \vee l_2 \dots \vee l_k$, where each l_i is either x_i or \bar{x}_i for some i .
 - Let z_1, z_2, \dots, z_{k-3} be $(k-3)$ new variables. We replace C by the following clause Z :
 - $Z = (l_1 \vee l_2 \vee z_1) \wedge (l_3 \vee \bar{z}_1 \vee z_2) \wedge (l_4 \vee \bar{z}_2 \vee z_3) \wedge \dots \wedge (l_{k-2} \vee \bar{z}_{k-4} \vee z_{k-3}) \wedge (l_{k-1} \vee l_k \vee \bar{z}_{k-3})$
 - Unlike cases 1 and 2, the clauses C and Z are not logically equivalent. However, the following two statements can be verified to be true.
 - i. When C is satisfiable Z is also satisfiable (Given an assignment to x_1, \dots, x_n in which C is TRUE, there is a way of setting the new variables z_1, z_2, \dots, z_{k-3} such that Z is TRUE.)
 - ii. When C is not satisfiable Z is not satisfiable (Given an assignment to x_1, \dots, x_n in which C is FALSE, there is no way of setting the new variables z_1, z_2, \dots, z_{k-3} such that Z is TRUE.)

Reduction from SAT to 3-SAT

- **case 4a:** When C is satisfiable
- Let $C = l_1 \vee l_2 \dots \vee l_k$, where each l_i is either x_i or \bar{x}_i for some i .
- C is satisfiable if at least one of the literals (say l_j) is TRUE.
- $$Z = (l_1 \vee l_2 \vee z_1) \wedge (l_3 \vee \bar{z}_1 \vee z_2) \wedge (l_4 \vee \bar{z}_2 \vee z_3) \wedge \dots \wedge (l_{k-2} \vee \bar{z}_{k-4} \vee z_{k-3}) \wedge (l_{k-1} \vee l_k \vee \bar{z}_{k-3})$$
- If $l_j = l_1$ or l_2 , then set all z_i s to FALSE. Every clause except the first consists of a variable \bar{z}_i which will evaluate to TRUE. So, Z will be TRUE.

Reduction from SAT to 3-SAT

- **case 4a:** When C is satisfiable
- Let $C = l_1 \vee l_2 \dots \vee l_k$, where each l_i is either x_i or \bar{x}_i for some i .
- C is satisfiable if at least one of the literals (say l_j) is TRUE.
- $$Z = (l_1 \vee l_2 \vee z_1) \wedge (l_3 \vee \bar{z}_1 \vee z_2) \wedge (l_4 \vee \bar{z}_2 \vee z_3) \wedge \dots \wedge (l_{k-2} \vee \bar{z}_{k-4} \vee z_{k-3}) \wedge (l_{k-1} \vee l_k \vee \bar{z}_{k-3})$$
- If $l_j = l_{k-1}$ or l_k , then set all z_i s to TRUE. Every clause except the last consists of a variable z_i which will evaluate to TRUE. So, Z will be TRUE.

Reduction from SAT to 3-SAT

- **case 4a:** When C is satisfiable
- Let $C = l_1 \vee l_2 \dots \vee l_k$, where each l_i is either x_i or \bar{x}_i for some i .
- C is satisfiable if at least one of the literals (say l_j) is TRUE.
- $Z = (l_1 \vee l_2 \vee z_1) \wedge (l_3 \vee \bar{z}_1 \vee z_2) \wedge (l_4 \vee \bar{z}_2 \vee z_3) \wedge \dots \wedge (l_{k-2} \vee \bar{z}_{k-4} \vee z_{k-3}) \wedge (l_{k-1} \vee l_k \vee \bar{z}_{k-3})$
- If $l_j \notin \{l_1, l_2, l_{k-1}, l_k\}$, then set z_1, z_2, \dots, z_{j-2} to TRUE and $z_{j-1}, z_j, \dots, z_{k-3}$ to FALSE.
- Let the clause containing l_j is C_j . Every clause to the left of C_j consists of a variable z_i which will evaluate to TRUE.
- Every clause to the right of C_j consists of a variable \bar{z}_i which will evaluate to TRUE.
- So, Z will be TRUE.

Reduction from SAT to 3-SAT

- **case 4b:** When C is not satisfiable
- Let $C = l_1 \vee l_2 \dots \vee l_k$, where each l_i is either x_i or \bar{x}_i for some i .
- C is not satisfiable if none of the literals l_1, l_2, \dots, l_k is TRUE.
- $Z = (l_1 \vee l_2 \vee z_1) \wedge (l_3 \vee \bar{z}_1 \vee z_2) \wedge (l_4 \vee \bar{z}_2 \vee z_3) \wedge \dots \wedge (l_{k-2} \vee \bar{z}_{k-4} \vee z_{k-3}) \wedge (l_{k-1} \vee l_k \vee \bar{z}_{k-3})$
- For Z to be satisfiable, all its clauses must be TRUE.
- The first clause is TRUE only if z_1 is TRUE.
- If z_1 is TRUE, the second clause is TRUE only if z_2 is TRUE.
- If z_2 is TRUE, the third clause is TRUE only if z_3 is TRUE...
- ... and so on upto z_{k-3} is TRUE.
- But, if z_{k-3} is TRUE, then the last clause will be FALSE.
- So, no way we can make Z satisfiable.

Reduction from SAT to 3-SAT

$$\begin{aligned}\varphi = & \left(\neg x_1 \vee \neg x_4 \right) \wedge \left(x_1 \vee \neg x_2 \vee \neg x_3 \right) \\ & \wedge \left(\neg x_2 \vee \neg x_3 \vee x_4 \vee x_1 \right) \wedge \left(x_1 \right) .\end{aligned}$$

Equivalent form:

$$\begin{aligned}\psi = & \left(\neg x_1 \vee \neg x_4 \vee z \right) \wedge \left(\neg x_1 \vee \neg x_4 \vee \neg z \right) \\ & \wedge \left(x_1 \vee \neg x_2 \vee \neg x_3 \right) \\ & \wedge \left(\neg x_2 \vee \neg x_3 \vee y_1 \right) \wedge \left(x_4 \vee x_1 \vee \neg y_1 \right) \\ & \wedge \left(x_1 \vee u \vee v \right) \wedge \left(x_1 \vee u \vee \neg v \right) \\ & \wedge \left(x_1 \vee \neg u \vee v \right) \wedge \left(x_1 \vee \neg u \vee \neg v \right) .\end{aligned}$$

Complexity Classes

- P
- NP
- Co-NP
- NPH
- NPC

Complexity Classes

- In Complexity Theory we often concentrate on decision problems.
- Reminder: a decision problem is one whose return values are either YES or NO (or true or false, or 0 or 1).
- If you can't find a polynomial-time algorithm for a decision problem, then you're not likely to find one for the non-decision problems (the optimization problem) to which it is related.
- Suppose you've a decision problem and an algorithm A that solves that decision problem.
 - We'll say that algorithm A accepts input x if it returns YES.
 - We'll say that algorithm A rejects input x if it returns NO.

Complexity Class P

- The complexity **class P** is the set of all decision problems that can be solved with worst-case polynomial time-complexity.
- In other words, a problem is in the class P if it is a decision problem and there exists an algorithm that solves any instance of size n in $O(n^k)$ time, for some integer k .
- So P is just the set of tractable decision problems: the decision problems for which we have polynomial-time algorithms.
- P is a set of problems, not algorithms. P is called a complexity class, because it is a class of problems that have the same/similar computational complexity.
- The definition of P states that a problem belongs to the class P if there is some deterministic algorithm that solves it in polynomial time.

Complexity Class NP

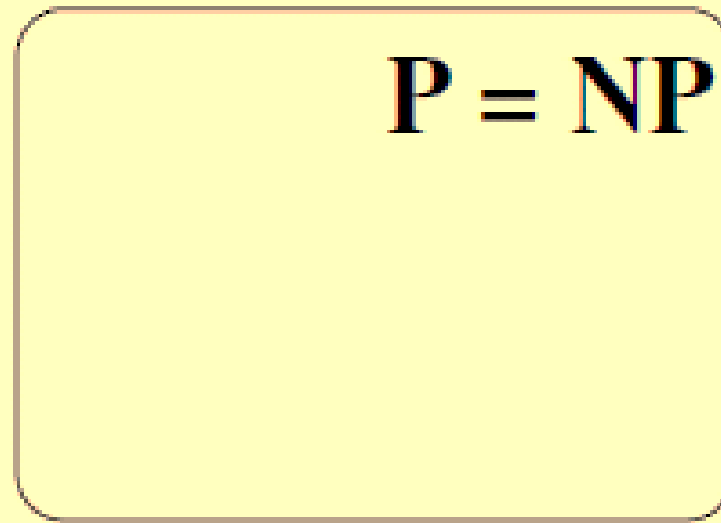
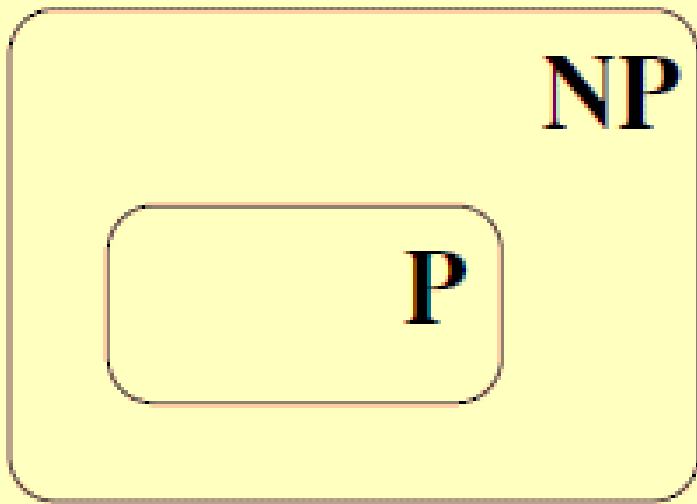
- The second class of decision problems that we look at is called NP, which stands for non-deterministic polynomial.
- The definition of NP involves the idea of a non-deterministic algorithm.
- Definition: The complexity class NP is the set of all decision problems that can be non-deterministically accepted in worst-case polynomial time.
- No conventional computer can implement a non-deterministic algorithm.
- No one has yet shown how even an unconventional computer (e.g. a quantum computer) could simulate a non-deterministic polynomial-time algorithm in polynomial time.
- So non-deterministic algorithms appear to be useless (we can't code them up and run them anywhere).

Complexity Class NP

- **Every problem in P is also in NP:** if a problem can be solved with a deterministic algorithm in polynomial time, it can be solved with a nondeterministic algorithm in polynomial time also. Therefore $P \subseteq NP$.
- The most important open question in computer science is whether $P = NP$ i.e., whether there are problems in NP that are not in P. No one has proved or disproved this statement, though many have tried.
- There are many important, practical problems known to be in NP, not yet known to be in P, and if someone were to prove that $P = NP$, this would imply that these problems did indeed have efficient solutions.
- If someone proved that $P \neq NP$, this would mean that some problems (Not specific ones) did not have an efficient solution, which leads to the concept of NP-completeness.

The $P = NP?$ Question

- We know that $P \subseteq NP$.
- The definition of NP allows for the inclusion of problems that may not be in P.
- But it may turn out that there are no such problems and that $P = NP$.



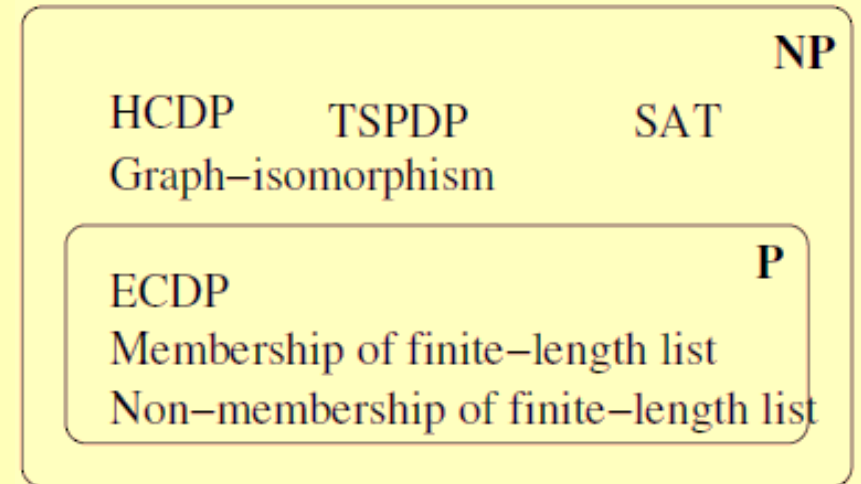
- We don't know which of these is the case. We know $P \subseteq NP$. But we don't know whether $P \subset NP$ (left-hand diagram) or $P = NP$ (right-hand diagram).

The $P = NP?$ Question

- The problems in the picture that are in NP but not in P are ones that we're not sure about:

- there is no known polynomial-time algorithm;
- but no proof of intractability.

(All we've managed to do is to show is that they're in NP.)



- So does $P = NP$? Or is $P \neq NP$ (i.e. $P \subset NP$?).
- If you can resolve this issue, you can win a million dollars.
<https://www.claymath.org/millennium-problems/millennium-prize-problems>

Class Co-NP

- Co-NP stands for the complement of NP Class. It means if the answer to a problem in Co-NP is NO, then there is proof that can be checked in polynomial time.
- If a problem X is in NP, then its complement X' is in Co-NP.
- For an NP and Co-NP problem, there is no need to verify all the answers at once in polynomial time, there is a need to verify only one particular answer “yes” or “no” in polynomial time for a problem to be in NP or Co-NP.
- Example of Co-NP problem: Integer Factorization.
- $P \subseteq \text{Co-NP}$.

Class NP-Hard

- We say that a decision problem P_i is NP-Hard if every problem in NP is polynomial-time reducible to P_i .
- i.e., P_i is NP-Hard if, for every $P_j \in \text{NP}$, $P_j \leq_P P_i$.
- Note that this doesn't require P_i to be in NP.
- Highly informally, it means that P_i is 'as hard as' all the problems in NP.
 - If P_i can be solved in polynomial-time, then so can all problems in NP.
 - Equivalently, if any problem in NP is ever proved intractable, then P_i must also be intractable.

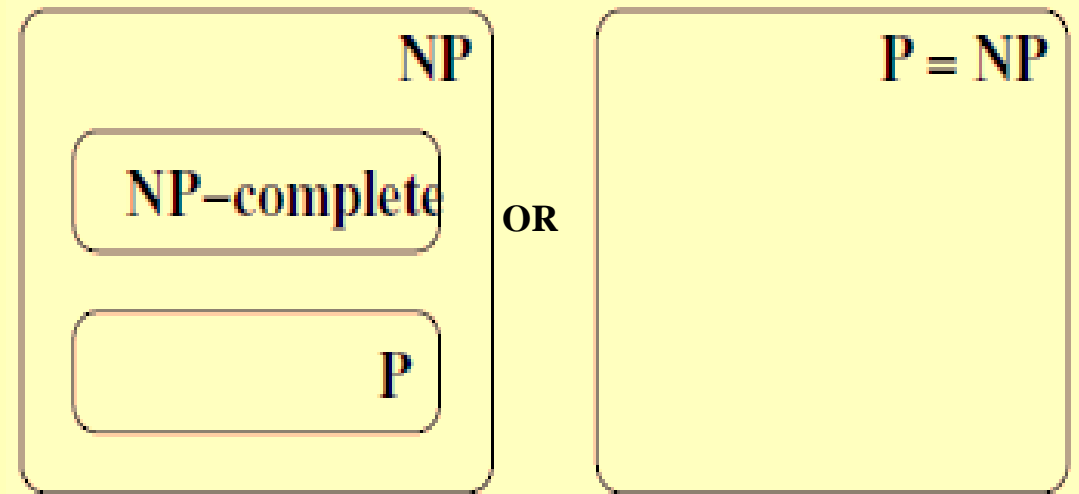
Class NP-Complete

- We say that a decision problem P_i is NP-Complete if:
 - it is NP-hard and
 - it is also in the class NP itself.
- i.e., P_i is NP-complete if P_i is NP-hard and $P_i \in \text{NP}$
- Highly informally, it means that P_i is one of the hardest problems in NP.
- So the NP-complete problems form a set of problems that may or may not be intractable but, whether intractable or not, are all, in some sense, of equivalent complexity.
- If anyone ever shows that an NP-complete problem is tractable, then
 - every NP-complete problem is also tractable
 - indeed, every problem in NP is tractable and so **P = NP**.

Class NP-Complete

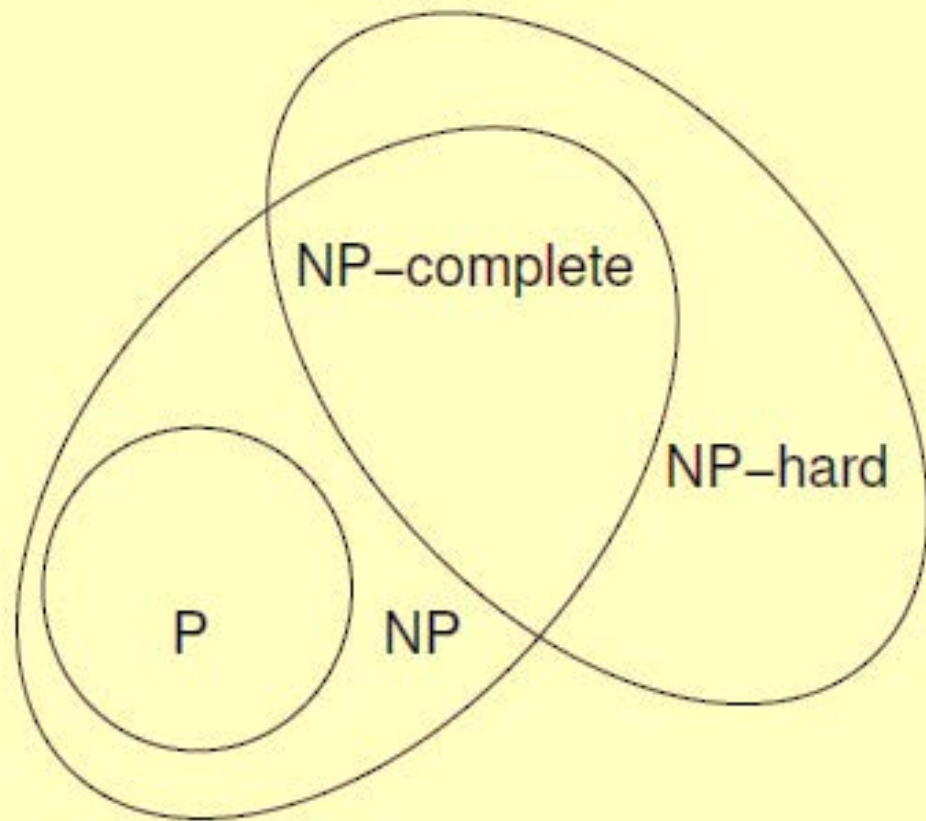
- If anyone ever shows that an NP-complete problem is intractable, then
 - every NP-complete problem is also intractableand, of course, $P \neq NP$.

- So there are two possibilities:

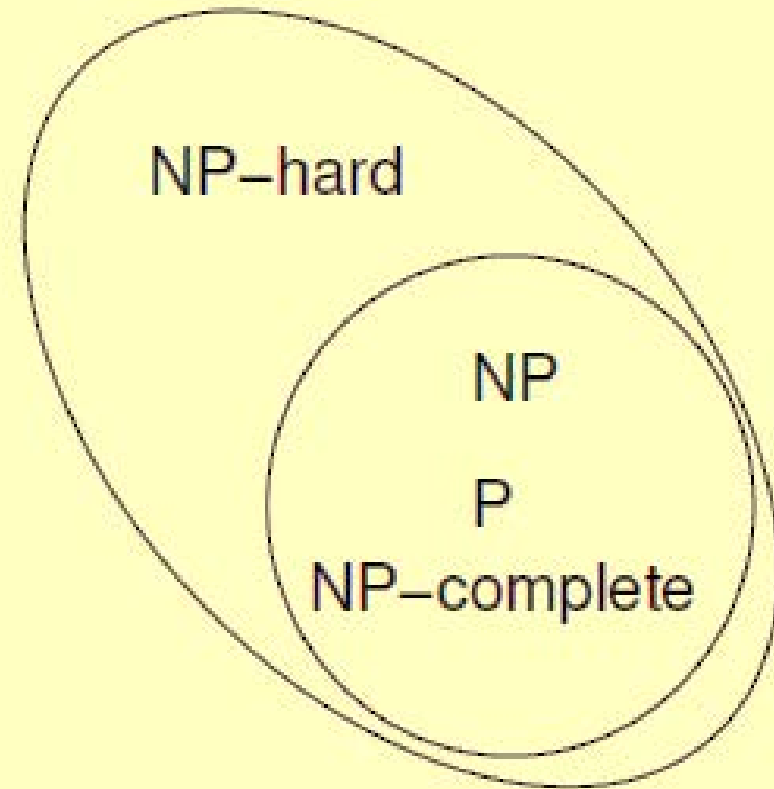


- We don't know which of these is the case.
- But this gives Computer Scientists a clear line of attack. It makes sense to put efforts on the NP-complete problems: they all stand or fall together.
- So these sound like very significant problems in our theory. But how would you show that a decision problem is NP-complete?

P, NP, NPC, NPH



if $P \neq NP$



if $P = NP$.

Efficient Certification

- The solutions to the NP problems operate in two stages: “Guess” then “verify”.
- There is a contrast between finding a solution and checking a proposed solution. For example, for Independent Set or 3-SAT, we may not know a polynomial-time algorithm to find solutions; but checking a proposed solution to these problems can be easily done in polynomial time.
- A decision problem can be viewed as.
- Problem X is a set of strings for which it can produce an answer *Yes* or *No*
- Instance s is one string.
- Algorithm A solves problem X : $A(s) = \begin{cases} yes & \text{if } s \in X \\ no & \text{if } s \notin X \end{cases}$

Efficient Certification

- A “checking algorithm” for a problem X has a different structure from an algorithm that actually seeks to solve the problem.
- In order to “check” a solution, we need the input string s , as well as a separate “certificate” string t that contains the evidence that s is a “yes” instance of X .
- Thus we say that C is an *efficient certifier* for a problem X if the following properties hold.
 - B is a polynomial-time algorithm that takes two input arguments s and t .
 - There is a polynomial function p so that for every string s , we have $s \in X$ if and only if there exists a string t such that $|t| \leq p(|s|)$ and $C(s, t) = \text{yes}$.
- Class NP = set of decision problems for which there exists a poly-time certifier.

Certifiers and certificates: satisfiability

SAT. Given a CNF formula Φ , does it have a satisfying truth assignment?

3-SAT. SAT where each clause contains exactly 3 literals.

Certificate. An assignment of truth values to the Boolean variables.

Certifier. Check that each clause in Φ has at least one true literal.

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

certificate t $x_1 = \text{true}, x_2 = \text{true}, x_3 = \text{false}, x_4 = \text{false}$

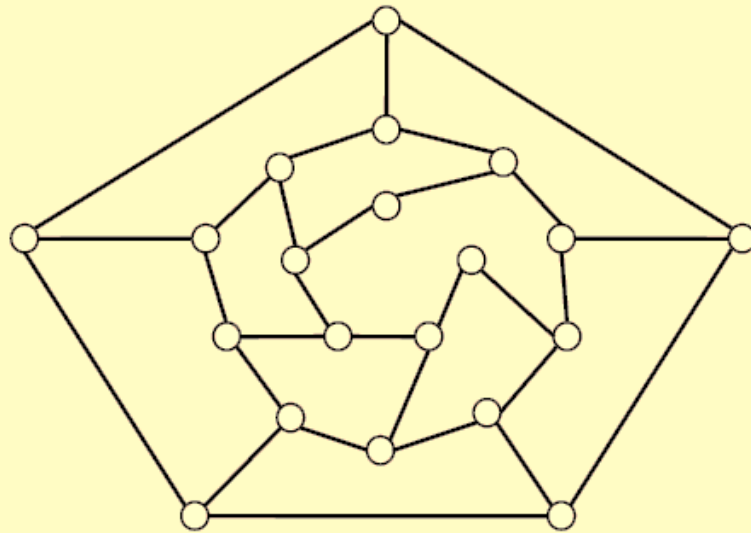
Conclusions. SAT \in **NP**, 3-SAT \in **NP**.

Certifiers and certificates: Hamilton path

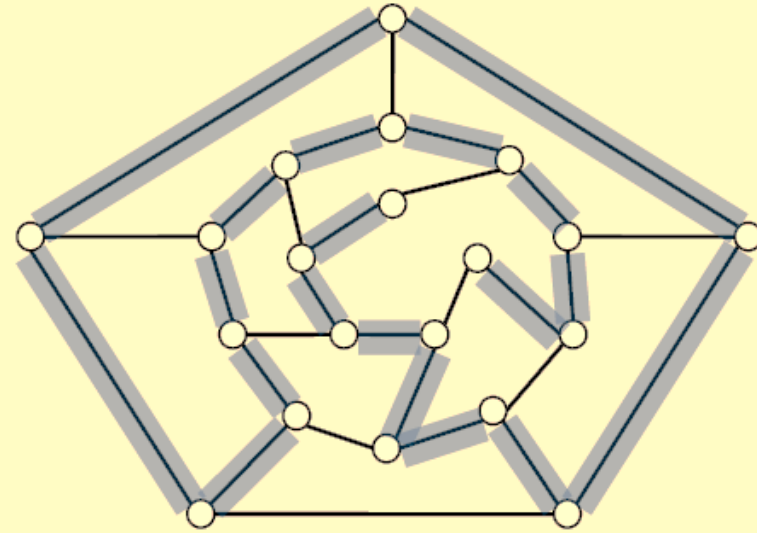
HAMILTON-PATH. Given an undirected graph $G = (V, E)$, does there exist a simple path P that visits every node?

Certificate. A permutation π of the n nodes.

Certifier. Check that π contains each node in V exactly once, and that G contains an edge between each pair of adjacent nodes.



instance s



certificate t

Conclusion. HAMILTON-PATH \in NP.

Proof for NP-Completeness

- To show a problem P_i is NP-complete (Method 1, from the definition)
 - First, confirm that P_i is a decision problem.
 - Then show P_i is in NP.
 - Then show that P_i is NP-hard by showing that every problem P_j in NP is polynomial-time reducible to P_i

Proving that a problem is in NP

- Let's show that the Hamiltonian Cycle Decision Problem is in NP.
- We'll come up with a non-deterministic algorithm
 - It will use a non-deterministic module to guess, in polynomial-time, a possible cycle.
 - Then, it will take polynomial-time to check whether this possible cycle is, in fact, a cycle covering all vertices of the graph except the start vertex exactly once.
- The first step is a black box, it does not describe any concrete steps for guessing the solution, just finds one.
- Both parts of the algorithm (the choosing and the checking) take polynomial time.
- This shows that the Hamiltonian Cycle Decision Problem is in NP.

Proof for NP-Completeness

Observation. Once we establish first “natural” **NP**-complete problem, others fall like dominoes.

To prove that $Y \in \mathbf{NP}$ -complete:

- Step 1. Show that $Y \in \mathbf{NP}$.
- Step 2. Choose an **NP**-complete problem X .
- Step 3. Prove that $X \leq_P Y$.

Proposition. If $X \in \mathbf{NP}$ -complete, $Y \in \mathbf{NP}$, and $X \leq_P Y$, then $Y \in \mathbf{NP}$ -complete.

Pf. Consider any problem $W \in \mathbf{NP}$. Then, both $W \leq_P X$ and $X \leq_P Y$.

- By transitivity, $W \leq_P Y$.
- Hence $Y \in \mathbf{NP}$ -complete. ▀

Example: NP-Complete Problem

- SAT is NP-complete.
- How was this proved?
- First, SAT is a decision problem.
- Second, SAT is in NP.
- Then it was shown SAT is NP-hard by showing that every problem in NP is polynomial-time reducible to SAT
 - This wasn't done one by one.
 - It was done by a general argument (By Cook–Levin theorem)

Example: NP-Complete Problem

Basic genres of NP-complete problems and paradigmatic examples.

- Packing/covering problems: SET-COVER, VERTEX-COVER, INDEPENDENT-SET.
- Constraint satisfaction problems: CIRCUIT-SAT, SAT, 3-SAT.
- Sequencing problems: HAMILTON-CYCLE, TSP.
- Partitioning problems: 3D-MATCHING, 3-COLOR.
- Numerical problems: SUBSET-SUM, KNAPSACK.

Practice. Most **NP** problems are known to be either in **P** or **NP**-complete.

NP-intermediate? FACTOR, DISCRETE-LOG, GRAPH-ISOMORPHISM,

Theorem. [Ladner 1975] Unless **P** = **NP**, there exist problems in **NP** that are neither in **P** nor **NP**-complete.

