Fall 2023 - Analysis and Design of Algorithms

Lectures 10 and 11: NP-Completeness 2

Ahmed Kosba

Department of Computer and Systems Engineering
Faculty of Engineering
Alexandria University

Recap

- Polynomial-time algorithms
- Relation between P, NP and co-NP

Today

- Definition of NP-complete problems
- Polynomial-time reducibility
- NP-complete problems

Note

- Remember that definitions of the complexity classes P and NP are concerned with decision problems.
- The definitions of complexity classes can appear in different forms:
  - Some definitions mention "decision problems" directly.
  - Other definitions use the formal language framework.

- Languages and problems are used in the same context in our discussion.
  - A decision problem Q is in NP.

  This can rephrased as:
  - The language of the decision problem Q is in NP.
    - The language of the problem Q is the set of inputs (encoding of problem instances) where the answer is "yes".

NP-Complete Class

NP-complete Problems

- Properties (informal):
  - Hardest problems in NP.
  - Furthermore, if any NP-complete problem is solved in polynomial-time, then P = NP, i.e., all problems in NP can be solved in polynomial time.
    - **Important: How does solving one problem have this huge implication?**
- Any problem in NP can be **reduced** to any NP-complete problems.
  - This is part of the definition of NP-complete problems, as we will see shortly.
- Before showing the formal definition, we will discuss a key concept: Reducibility.

Reducibility

- A problem Q can be reduced to another problem Q', if we can *easily rephrase* any instance of Q as an instance of Q'.

- We are interested in the cases when this *rephrasing* can be done efficiently, e.g., in polynomial time.

Reducibility Example 1

Q: Problem of solving linear equations

Q': Problem of solving quadratic equations

Any instance of Q can be expressed as:

Find x, such that $ax + b = 0$

This can be rephrased as an instance of Q'

Find x, such that $0*x^2 + ax + b = 0$

Note: The above problems are not decision problems, but the concept will apply normally.

Reducibility Example 2

Q: Given a set of n Boolean variables, is all of them True?

Q': Given a set of n integers, is their sum greater than or equal to n?

Any instance of Q can be expressed as:

- Given Boolean variables $(x_1, x_2, \ldots, x_n)$ is all of them True?

This can be rephrased as an instance of Q'

- Given the integer variables $(y_1, y_2, \ldots, y_n)$, where
  $y_i = 1$ if x_i is True, 0 otherwise
  Is their sum greater than or equal to n?

More illustration for example 2

- Reducing Q to Q' means that we can solve Q if we have an algorithm that solves Q'.
- Let's illustrate this using a code example.

Suppose that Q' can be solved by an algorithm A'.

```
A' (int[] y){
    int sum = 0;
    int n = y.length;
    for(int i = 0; i < n; i++){
        sum+=y[i];
    }
    return sum >= n;
}
```

How can we use A' to solve an instance of Q?

More illustration for example 2

- Let's write an algorithm A for solving Q using A'.

```
A (boolean[] x){
    int n = x.length;
    int[] y = new int[n];
    for(int i = 0; i < n; i++){
        y[i] = x[i]? 1: 0;
    }
    return A'(y);
}
```

```
A' (int[] y){
    int sum = 0;
    int n = y.length;
    for(int i = 0; i < n; i++){
        sum += y[i];
    }
    return sum >= n;
}
```

We can use A' to solve any instance of A, but we have to do a reduction first (the blue code).

Additional Examples

- Activity selection can be reduced to finding maximum independent set in a graph.
- Sorting can be reduced to the convex hull problem.

We will see more examples in the next lecture, but in the context of NP-complete problems.

Polynomial-time Reducibility

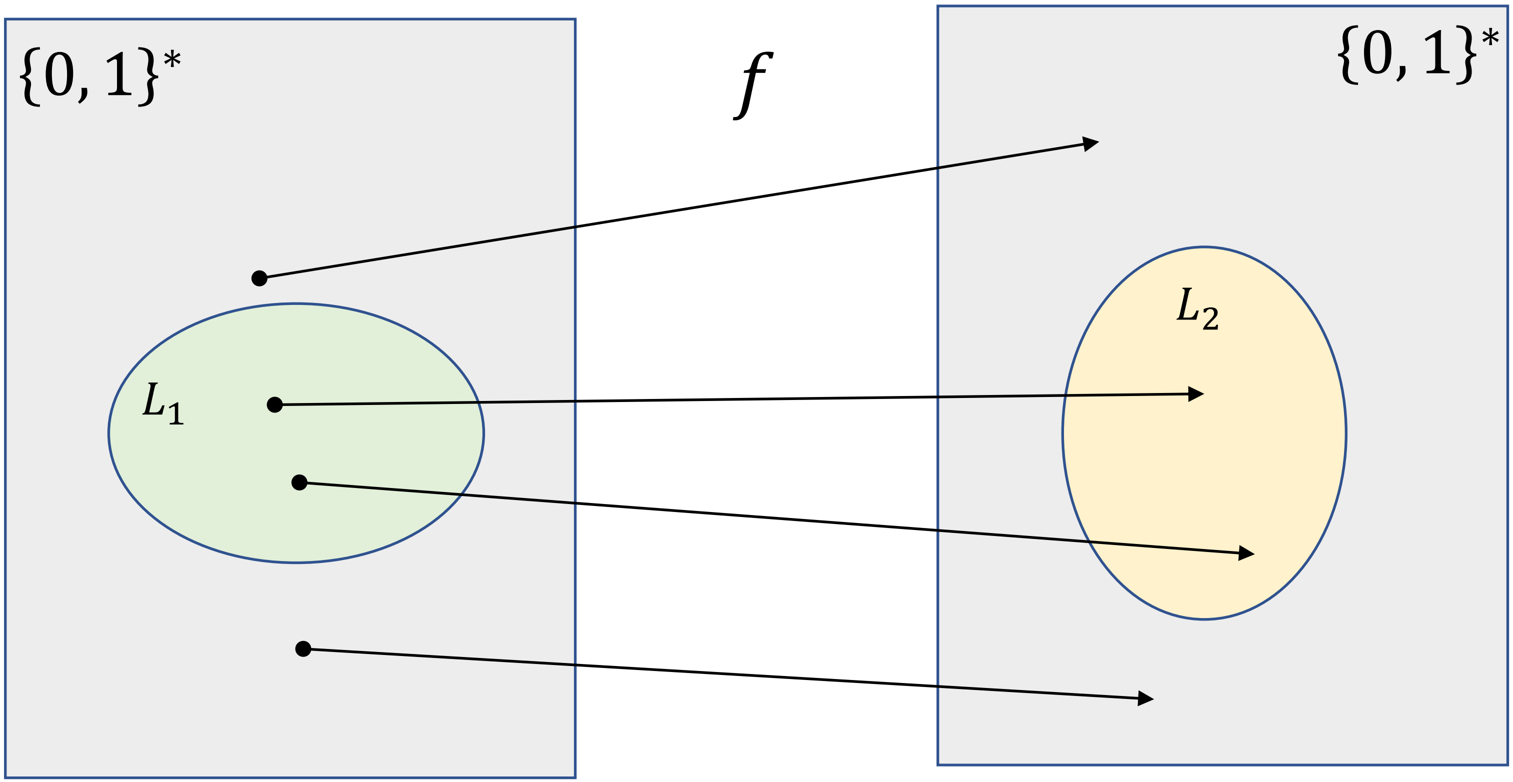
A language L_1 is polynomial-time reducible to a language L_2 if there exists a polynomial-time computable function $f: \{0,1\}^* \rightarrow \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$x \in L_1$ if and only if $f(x) \in L_2$

Notation:

$$L_1 \leq_p L_2$$

L_1 is polynomial-time reducible to L_2



Typical decision problem:
Given x, is $x \in L_1$?

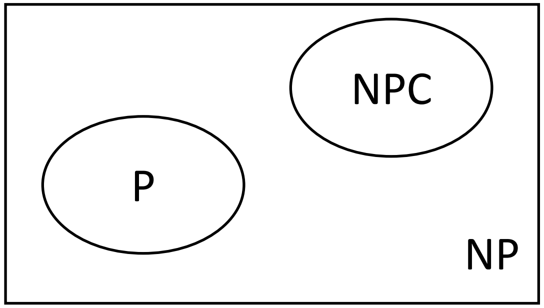
Another way to express this question is to check if $f(x) \in L_2$

NP-Complete Class

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

1- $L \in NP$

2- $L' \leq_p L$ for every $L' \in NP$



Assuming P is not equal to NP

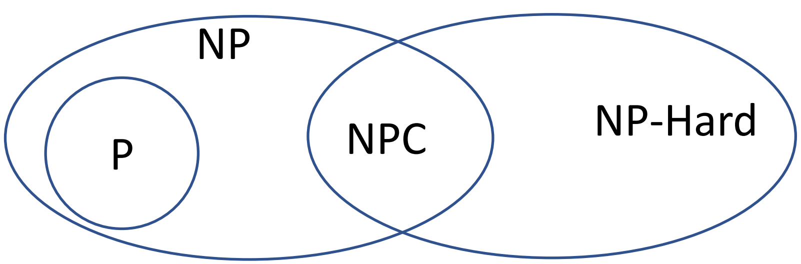
NP-Complete vs NP-Hard

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

1- $L \in NP$

2- $L' \leq_p L$ for every $L' \in NP$

If a language satisfies condition 2 then it's called an **NP-hard** language.



Assuming P is not equal to NP

NP-Complete Class

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

1- $L \in NP$

2- $L' \leq_p L$ for every $L' \in NP$

How to show that a problem is NP-complete?

- Proving condition 1 can be similar to what we did in the Hamiltonian cycle problem.
- However, how to prove condition 2? The number of problems in NP is infinite!

NP-Complete Class

- One way to prove that a problem Q is NP-complete:
  - Step 1:
    - Prove that the problem is in NP.
  - Step 2:
    - Start from a known NP-complete problem Q'.
    - Show that Q' is reducible in polynomial-time to Q.

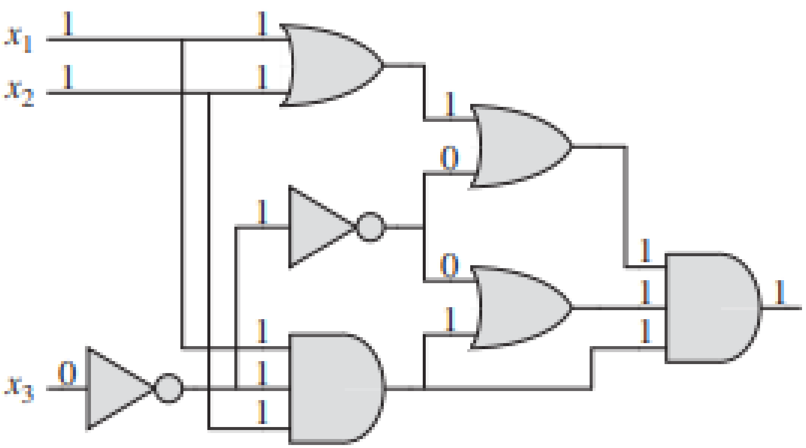
- Goal of this lecture
  - Discuss a first NP-complete problem.
  - Discuss multiple known NP-complete problems and show how to prove their NP-completeness through the above methodology.

First NP-Complete Problem

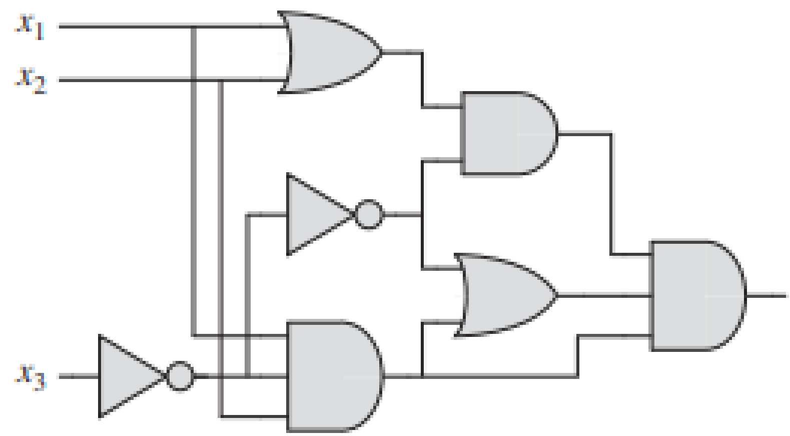
- The problem we will start from is the circuit satisfiability problem.
  - Given a Boolean combinational circuit that is composed of ANR/OR/NOT gates, is the circuit **satisfiable**?
  - A single-output circuit is **satisfiable**, if there is at least one set of input values that can make the output of the circuit to be 1.
    - In this case, the input assignment is called a satisfying assignment.

Circuit Examples

Example from CLRS



(a)



(b)

Satisfiable circuit

Unsatisfiable circuit

Q: Given an input assignment, what is the cost of computing the output?

The Circuit Satisfiability Problem

- Using formal language definitions:
  - $\text{CIRCUIT-SAT} = \{\langle C \rangle : C \text{ is a satisfiable Boolean combinational circuit}\}$

- How should we define the size of the problem instance?
  - The input size here is the size of the encoding that represents the circuit in memory.
  - A combinational circuit can be encoded as a DAG.
  - The size will be polynomial in the number of wires (edges) and the number of gates (vertices).

The Circuit Satisfiability Problem

- Naïve solution for this problem?
  - Check all possible value assignments, and see if any assignment leads to output 1.
  - If the number of inputs is k, then the number of possible assignments to check is: 2^k
  - Is this a polynomial-time algorithm?
    - If the circuit size is polynomial in k, then this the above approach will not lead to a polynomial time algorithm.

- Is this problem in NP?
  - Yes, a satisfying assignment for a circuit can be verified in polynomial time (w.r.t. the size of the circuit).
    - Note that the length of the certificate here is the number of input wires, which is polynomial in the size of the input circuit.
    - The verification effort is linear in the size of the circuit here.

The Circuit Satisfiability Problem

Theorem: Circuit-SAT is NP-complete.

- To show that Circuit-SAT is NP-complete, we need to show
  - Circuit-SAT is in NP (Done).
  - All problems / languages in NP are polynomial-time reducible to Circuit-SAT.
- Proving the second statement is involved. We will provide a high-level intuition.

Intuition

- Each language in NP has a polynomial-time verification algorithm that can verify membership of an input x in the language using a certificate y, where $|y| = O(|x|^c)$.
- Any verification algorithm of this kind can be viewed as a Boolean circuit that receives $|x|$ bits for the input instance and $O(|x|^c)$ bits for the certificate, and outputs 0 or 1.
  - As the algorithm is polynomial-time, the corresponding circuit can be shown to have polynomial size.
- The question of whether an input x belongs to an NP language can be reduced to whether there is a satisfying assignment to the circuit corresponding to the verification algorithm, where the input bits are hardcoded according to the input x.

Example: Reducing an NP problem to Circuit-SAT

Consider the decision problem of **composite numbers**:
Given a positive integer x, is x a composite number?

- This is in NP, verifying that a number x is composite can be done given a certificate that consists of two integers p and q, such that $x = p*q$ and both $p, q > 1$.

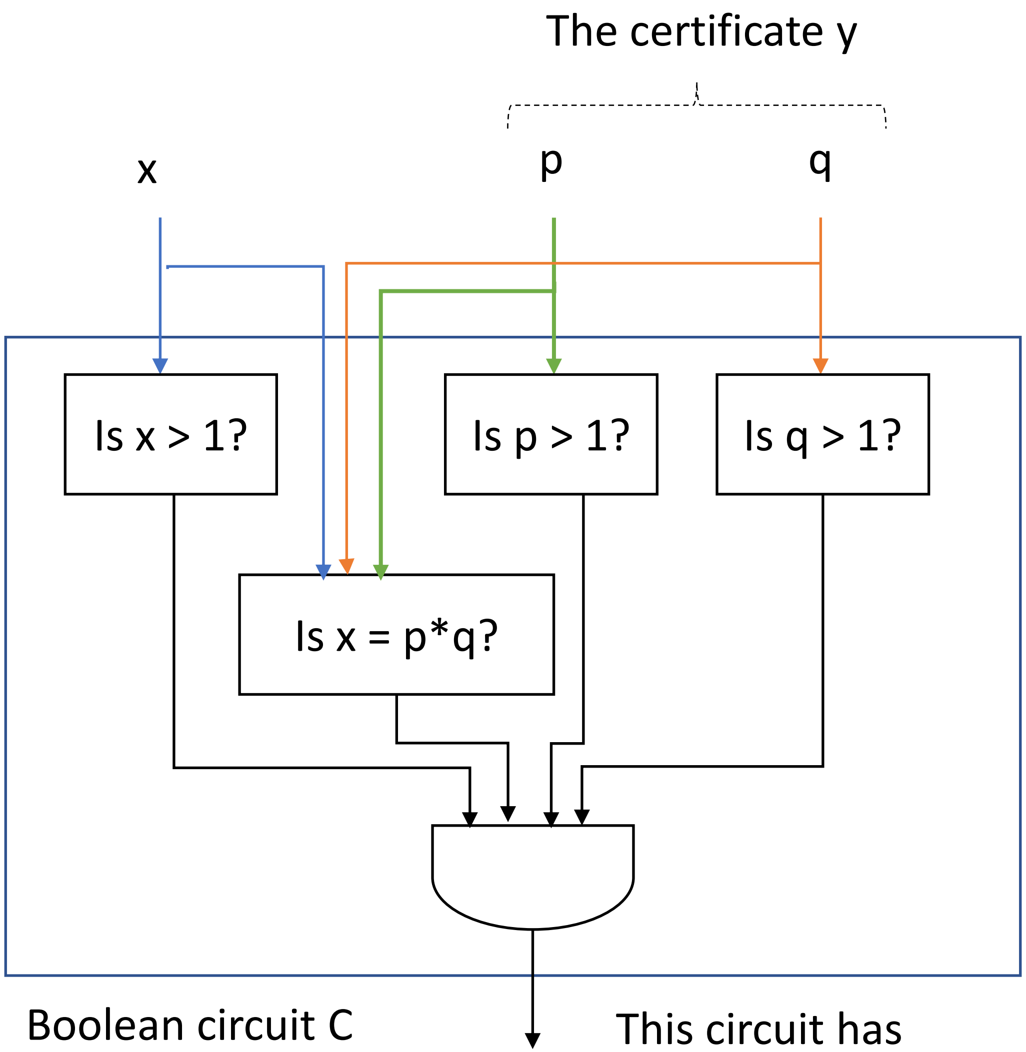
- How could we reduce solving the above problem to solving CIRCUIT-SAT?

Example

Verification algorithm for language COMPOSITE:

Assume y = (p, q)

```
A(x, y){
    Assert x > 1
    (p, q) = y
    Assert p > 1 and q > 1
    if p*q == x
        return 1
    else
       return 0
}
```



Boolean circuit C realizing algorithm A

This circuit has polynomial size w.r.t. to the size of input x. (why?)

The question of whether x is a composite number can be transformed into whether the circuit C with hardcoded input x is satisfiable.

Remark

- Note that a single example is not sufficient to prove that a polynomial-time reduction always exists.
  - As emphasized in previous lectures, using examples to prove claims is wrong.
  - We use examples for illustration in this lecture.
- The previous simple example is just to provide an intuition.
  - What if the verification algorithm had loops and other instructions?
  - We need a more **universal** way to represent algorithms as circuits.
- You can find a proof sketch in CLRS which makes use of how computers execute instructions in general (Figure 34.9)
- For the CIRCUIT-SAT problem, we need to show that
  - The verification algorithm for **any** NP language can have a corresponding circuit of polynomial size.
  - Generating the corresponding circuit must be done in polynomial time.
  - The generated circuit will be satisfiable if and only if the input x is in the NP language.

NP-Completeness Proofs

- After discussing our first NP-complete problem, we can use the following to prove the NP-completeness of other problems.
- In a nutshell, to prove that a problem Q is NP-complete:
  - Step 1: Prove that the problem Q is in NP.
  - Step 2: Show that a known NP-complete problem, e.g., Circuit-SAT, is reducible in polynomial-time to Q.

NP-Completeness Proofs

- More formally, to prove that a language L is NP-complete [CLRS]
  - Prove that $L \in NP$
  - Select a known NP-complete language L'
  - Find a function *f* that maps **every instance** *x* of L' to an instance *f*(*x*) of L.
  - Prove that the function *f* satisfies the following:
    - For all $x \in \{0, 1\}^*$, $x \in L'$ **if and only if** $f(x) \in L$.
    - The algorithm that computes *f* runs in **polynomial time.**

Other NP-complete problems

- Satisfiability of a Boolean formula
- Satisfiability of 3-CNF
- CLIQUE
- Vertex cover
- Hamiltonian Cycle
- Traveling Salesman Problem

Satisfiability of a Boolean formula

Given a Boolean formula ϕ that has

- n Boolean variables $x_1, x_2, \ldots, x_n$
- m Boolean connectives: $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\rightarrow$ (implication), $\leftrightarrow$ (if and only if)
- Parenthesis

Is ϕ satisfiable?

Example of a satisfiable formula: $(x_1 \vee x_2) \wedge (x_3 \vee \neg x_1)$

The corresponding language

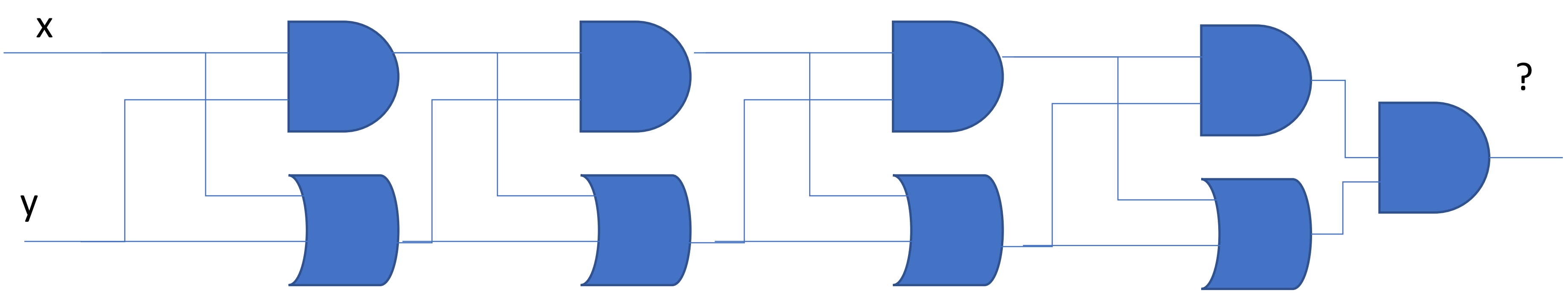
$\mathrm{SAT} = \{\langle\phi\rangle: \phi \text{ is a satisfiable Boolean formula}\}$

SAT is NP-complete

- To prove that the previous language is NP-complete:
  - Show that SAT is in NP.
    - Verifying a satisfying assignment can be done in polynomial time.
  - Show that Circuit-SAT $\leq_p \text{SAT}$.
    - Can we map any circuit satisfiability problem to the problem of whether a Boolean formula is satisfiable?
    - Yes, but this has to be done carefully.

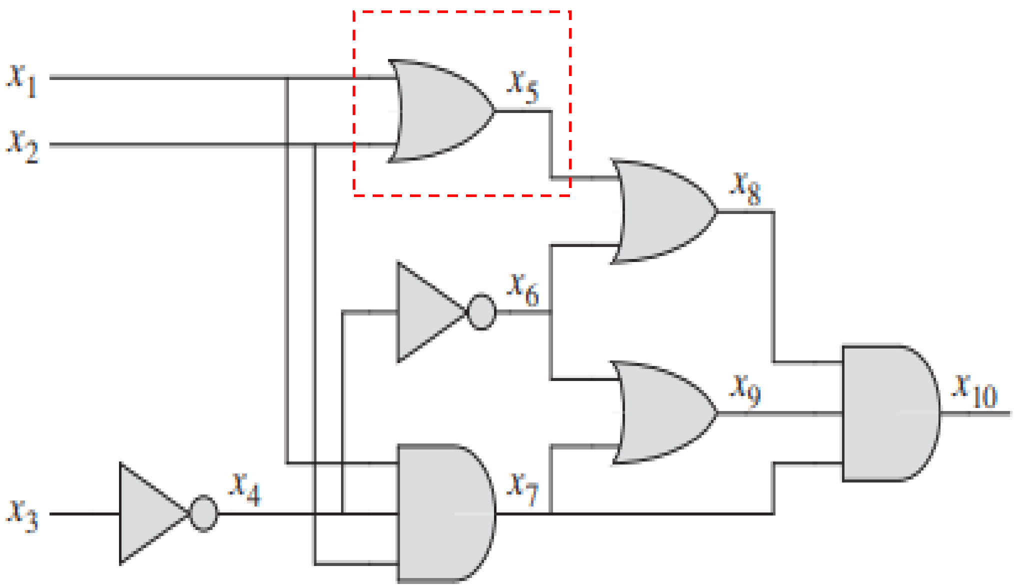
Reducing Circuit-SAT

- The naïve way:
  - Starting from the input variables, find the Boolean formula that corresponds to each wire till the output wire is reached.
  - Problem: The resulting formula could be exponential in the size of the circuit.
    - For example, in the circuit below, what is the size of the output formula?



Reducing Circuit-SAT to SAT (A better way)

- To avoid the exponential growth of formula length, introduce intermediate variables.
- Example from CLRS



$$
\begin{aligned}
\phi = x_{10} & \wedge (x_4 \leftrightarrow \neg x_3) \\
& \wedge \boxed{(x_5 \leftrightarrow (x_1 \vee x_2))} \\
& \wedge (x_6 \leftrightarrow \neg x_4) \\
& \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
& \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\
& \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\
& \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))
\end{aligned}
$$

The corresponding formula will be only satisfiable **if and only if** the circuit is satisfiable. See the textbook for more formal arguments.

3-CNF

- Conjunctive normal form (CNF):
  - A Boolean formula is in CNF form if it is the AND of clauses where each is OR of literals.
  - A literal is a variable or its negation

  Example: $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT),

  $(x_1 \vee x_2 \vee x_4) \wedge (x_2 \vee x_3) \wedge (\neg x_4 \vee x_5 \vee x_6)$

  - A CNF formula is 3-CNF is when each clause **has exactly three distinct literals**

    $(x_1 \vee x_2 \vee x_4) \wedge (x_2 \vee x_3 \vee x_7) \wedge (\neg x_4 \vee x_5 \vee x_6)$

3-CNF-SAT is NP-complete

- Although a 3-CNF formula has some structure, it is still hard to find if it is satisfiable in the general case.
- In fact, 3-CNF-SAT is NP-complete.
- To prove the above
  - Show that 3-CNF-SAT is in NP.
    - This is similar to the SAT case.
  - Show that SAT $\leq_p$ 3−CNF−SAT.

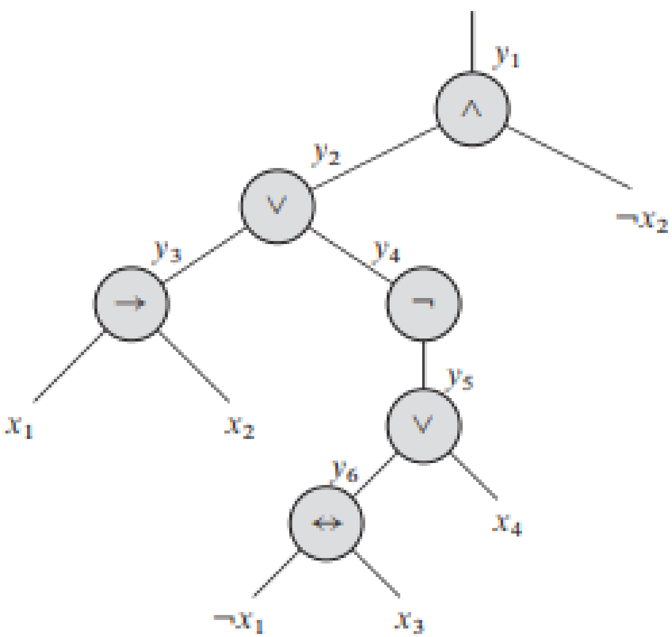
Reducing SAT to 3–CNF–SAT

Example from CLRS

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

Find the parse tree of ϕ, introduce intermediate variables and write a formula ϕ' that captures the parse tree.

$\rightarrow$ (implication), $\leftrightarrow$ (if and only if)



$$\begin{aligned}
\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\
&\wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\
&\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\
&\wedge (y_4 \leftrightarrow \neg y_5) \\
&\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\
&\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)) .
\end{aligned}$$

Reducing SAT to 3–CNF–SAT

- We will need to convert ϕ' to a 3-CNF formula.
- Note that each clause has at most three literals.
  - We need each clause to have exactly three distinct literals connected by OR operations.
- For any clause that uses other operations, e.g., $y_2 \leftrightarrow y_3 \vee y_4$

  To convert this to a CNF formula, **find the corresponding truth table**, and then use basic Boolean algebra to express this as a CNF formula (product of sums).

  Note that since each clause has 3 literals at most, processing each clause can be done in constant time.

Reducing SAT to 3–CNF–SAT

- What remains is converting the CNF formula to a 3-CNF.
- For clauses that have fewer than three literals, a transformation is straightforward.
  - If a clause has only two literals, e.g., $x_1 \vee x_2$, we can introduce a variable p, and replace $x_1 \vee x_2$ by:

    $$(x_1 \vee x_2 \vee p) \wedge (x_1 \vee x_2 \vee \neg p)$$

  - The same can be applied for clauses that have one literal only. If a clause has only x_1, this can be converted to

    $$(x_1 \vee p \vee q) \wedge (x_1 \vee p \vee \neg q) \wedge (x_1 \vee \neg p \vee q) \wedge (x_1 \vee \neg p \vee \neg q)$$

Exercise: Use Boolean algebra to verify the correctness of this.

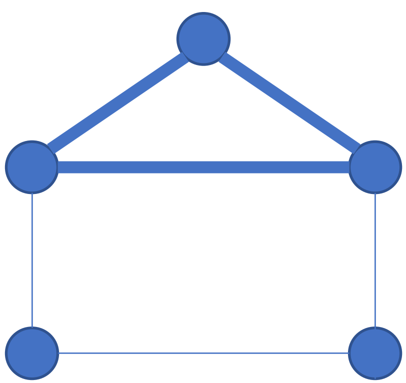
Reducing SAT to 3–CNF–SAT

- **Any** Boolean formula ϕ can be converted to a 3-CNF formula ϕ'.
  - This can be done in polynomial time.
  - The 3-CNF formula ϕ' will only be larger than ϕ by a constant factor.
- ϕ will only be satisfiable **if and only if** ϕ' is satisfiable.

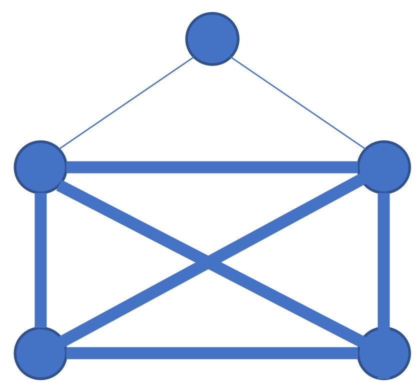
See the textbook for formal arguments.

The Clique Problem

- A **clique** in an undirected graph G is a complete subgraph of G.
- The size of a clique is defined by the number of vertices it has.



This graph has
a clique of size 3



This graph has a clique of size 4

The Clique Problem

- **Optimization problem:** Given an input graph G, find a clique of maximum size.
- **Decision problem:** Given an input graph G, is there a clique of size k in the graph? (known also as k-CLIQUE)
  - Note that the question is for a general value of k.
  - There are cases where this question can be answered in polynomial time. For example,
    - Is there a clique of size 3 in an input graph G = (V, E)?
    - Is there a clique of size |V| in an input graph G = (V, E)? (Check if the graph is complete)
  - However, this does not hold for all k. When k is |V|/2, the problem is challenging.

The Clique Problem

- Using the formal language framework
  - $\mathrm{CLIQUE} = \{\langle \mathrm{G, k} \rangle : \mathrm{G\ is\ a\ graph\ that\ has\ a\ clique\ of\ size\ k}\}$
- CLIQUE is NP-complete.
  - Showing that CLIQUE is in NP is straightforward.
  - For the second property we can show that
    3-CNF-SAT $\leq_p$ CLIQUE

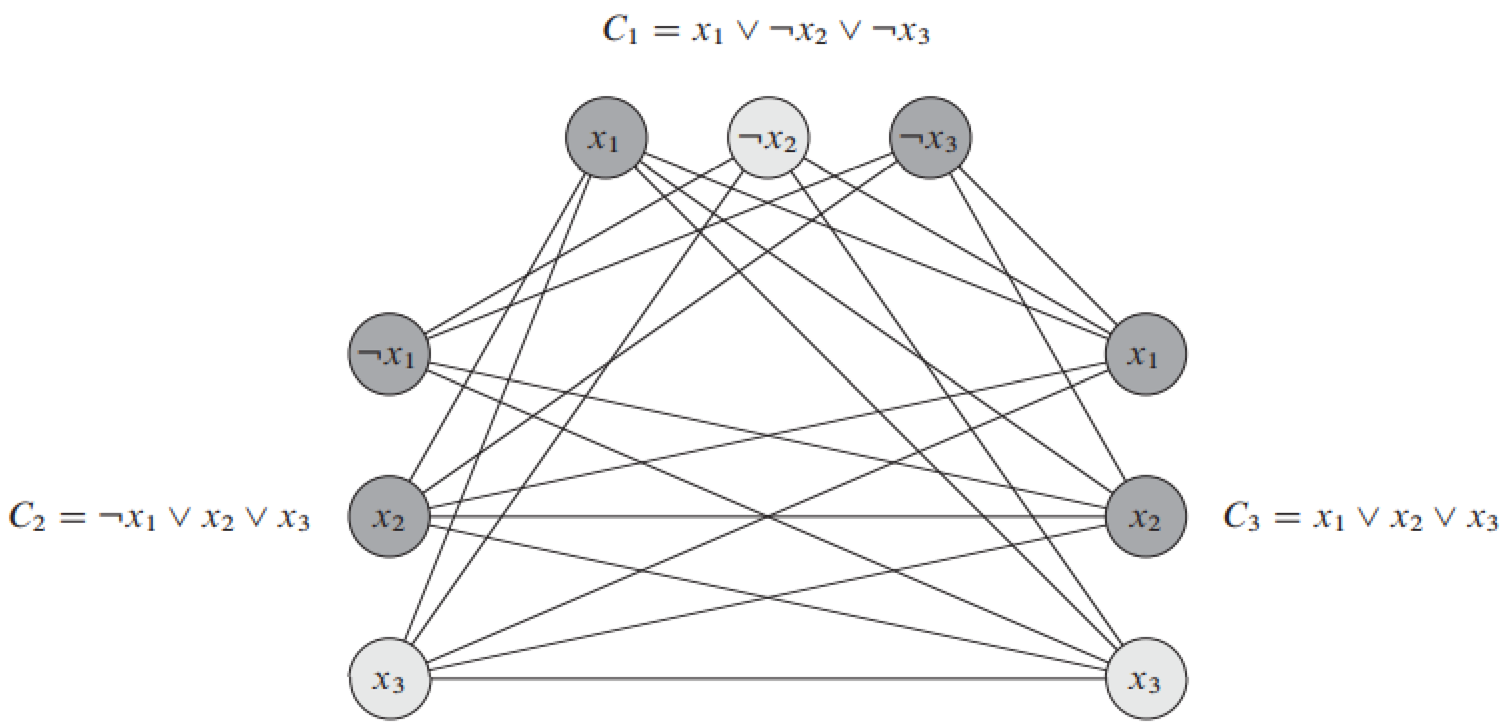
Reducing 3-CNF-SAT to CLIQUE

- How can we solve the satisfiability of a 3-CNF formula by solving a CLIQUE problem?
  - A transformation can be made such that any input 3-CNF can be converted to a graph.
    - **Vertices:** For each clause, add a triple of vertices, such that each vertex represents one of the literals.
      - For example, for $(x_1 \vee x_2 \vee \neg x_3)$, three vertices will be added, one for x_1, one for x_2 and one for $\neg x_3$.
      - For a formula that has k clauses, the number of vertices will be 3k.
    - **Edges:** Add an edge between two vertices only if:
      - They are in different triples.
      - The corresponding literals are consistent.

Reducing 3-CNF-SAT to CLIQUE

Example from CLRS

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

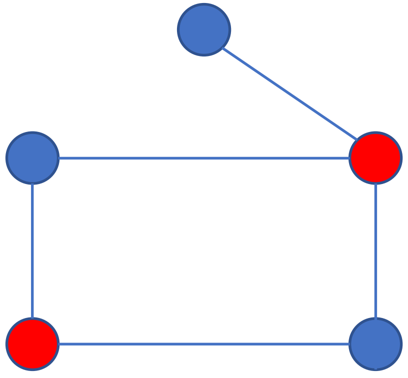


Reducing 3-CNF-SAT to CLIQUE

- It can be shown that any 3-CNF formula with k clauses is satisfiable **if and only if** the corresponding graph has a clique of size k.

The Vertex Cover Problem

- A vertex cover in an undirected graph G=(V, E) is a subset of vertices V' such that for each (u, v) in E, then u is in V', or v is in V'.
- The size of a vertex cover is the number of vertices in V'.



A vertex cover of size 2 (red)

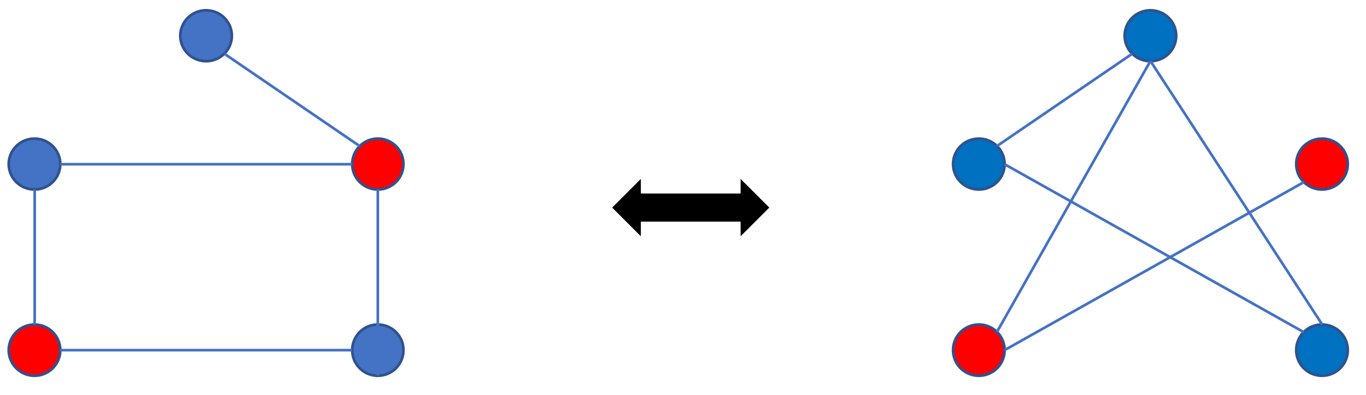
The Vertex Cover Problem

- Optimization problem: Given an input graph G, find a vertex cover of minimum size.
- Decision problem: Given an input graph G, does it have a vertex cover of size k?
  - Note that the question is for a general value of k.
  - There are cases where this question can be answered in polynomial time, but this does not hold for all k.
- $\text{VERTEX-COVER} = \{\langle \text{G, k} \rangle\text{: G is a graph that has a vertex cover of size k}\}$
- VERTEX-COVER is NP-complete.
  - Showing that VERTEX-COVER is in NP is straightforward.
  - For the second property we can show that
    $\text{CLIQUE} \leq_p \text{VERTEX}-\text{COVER}$

Relation between CLIQUE and VERTEX-COVER problems

In the previous vertex cover example, what can we observe about the blue vertices?

- No blue vertex is connected to any other blue vertex.
- If we compute the **complement of the graph**, all the blue vertices will be connected to each other.



A vertex cover of size 2 (red)

A clique of size 3 (blue) emerged.

Reducing CLIQUE to VERTEX-COVER

Goal: prove that $\text{CLIQUE} \leq_p \text{VERTEX}-\text{COVER}$,

We need to show that CLIQUE problem instances can be converted to VERTEX-COVER problem instances.

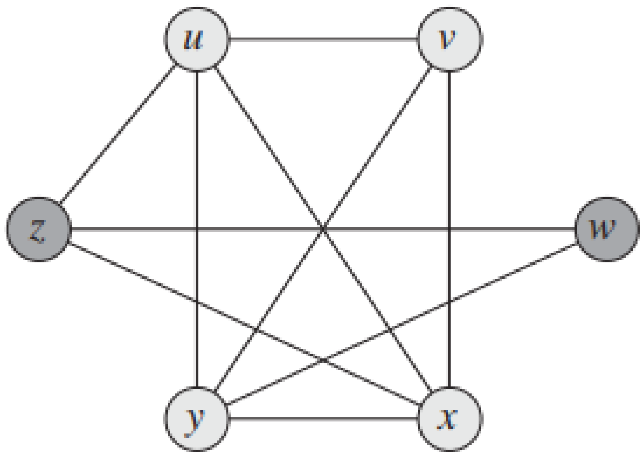
Thankfully, the previous transformation can be used in the opposite direction.

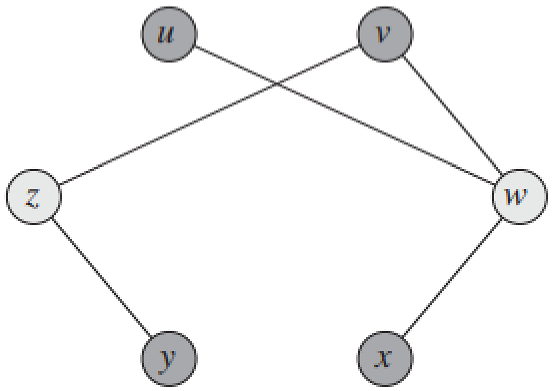
It can be formally shown that a clique of size k will exist in a graph **if and only if** there is a vertex cover of size |V| - k in the complement graph. (The proof is in the textbook)

Note that computing the complement graph can be done in polynomial time.

Reducing CLIQUE to VERTEX-COVER

Example from CLRS





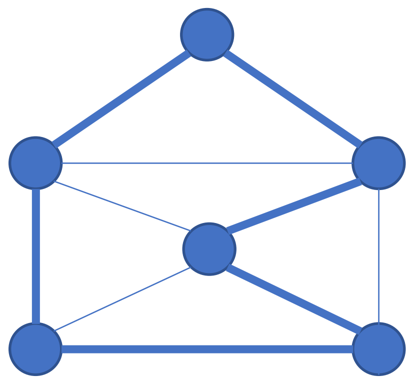
Is there a clique of size 4 in this graph?

Is there a vertex cover of size 2 in the complement graph?

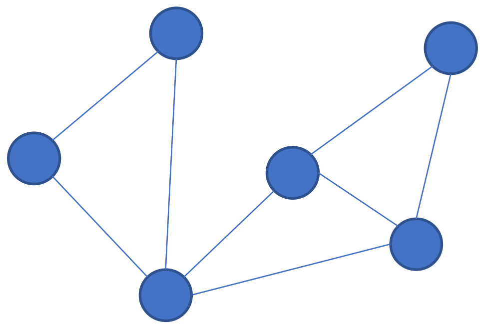
The Hamiltonian-Cycle Problem

From previous lecture:

- **Determine whether a graph has a Hamiltonian cycle:** A Hamiltonian cycle is a **simple cycle** that visits **each vertex** of the graph **exactly once**.



This graph has a Hamiltonian cycle



This graph does not have a Hamiltonian cycle.

The Hamiltonian-Cycle Problem

- Can be shown to be NP-complete by reducing the vertex cover problem to the Hamiltonian cycle problem (Details of the reduction are not covered in this lecture).

Traveling Salesman Problem (TSP)

- A salesman would like to visit n cities. The starting city should be the same as the last city, i.e., a cycle. Each city in the tour can be visited only once.
- Any two cities are connected, i.e., it's a complete undirected graph. The cost of traveling between any two cities is non-negative.
- Optimization Problem: Find the tour that minimizes the total cost.
- Decision Problem: Is there a tour with cost <= k?

Exercise

- Prove that the TSP problem is NP-complete.
  - Answered in class.

The subset-sum problem

Consider the following decision problem:

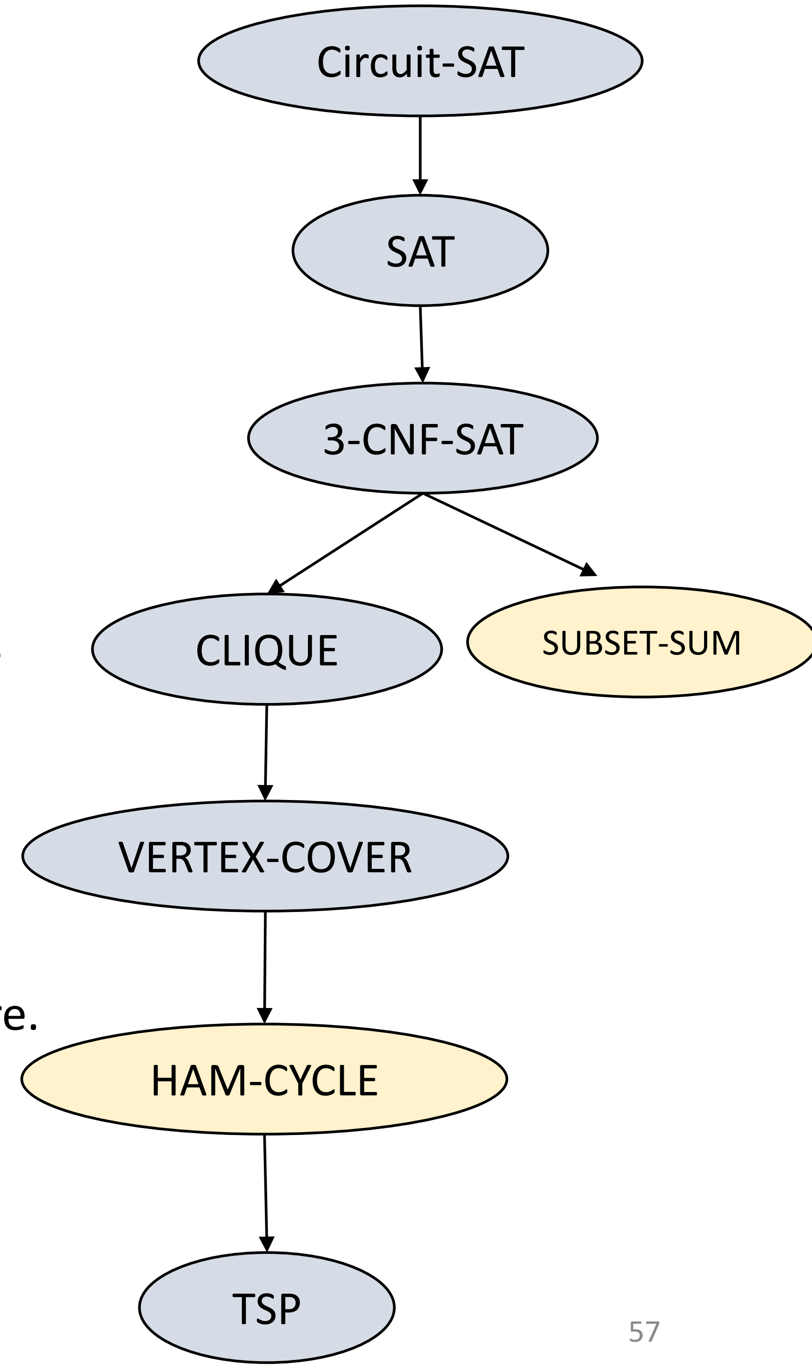
Given a finite set S of positive integers and an integer target t > 0, is there a subset S' ⊆ S such that the sum of elements of S' is equal to t?

This is also an NP-complete problem. A reduction from 3-CNF-SAT can be shown.

The proof is not covered in this lecture.

Summary

- How to prove a problem to be NP-complete.
- NP-complete problem examples
  - The proofs of the yellow problems were not covered in the slides of this lecture.



- Note: There are many other NP-complete problems that we did not cover in this lecture. Examples include the decision variants of:
  - 0-1/unbounded knapsack
  - Graph coloring (starting from 3 colors)
  - …