8. Computational Complexity

Pukar Karki
Assistant Professor

Introduction

- Some common functions, ordered by how fast they grow are highlighted below.

constant	$O(1)$
logarithmic	$O(\log n)$
linear	$O(n)$
n-log-n	$O(n \times \log n)$
quadratic	$O(n^2)$
cubic	$O(n^3)$
exponential	$O(k^n)$, e.g. $O(2^n)$
factorial	$O(n!)$
super-exponential	e.g. $O(n^n)$

- Computer Scientists divide these functions into two classes:

Introduction

- **Polynomial functions:** Any function that is $O(n^k)$, i.e. bounded from above by n^k for some constant k.

E.g. $O(1)$, $O(\log n)$, $O(n)$, $O(n \times \log n)$, $O(n^2)$, $O(n^3)$

- Here the word ‘polynomial’ is used to lump together functions that are bounded from above by polynomials. So, log n and n × log n, which are not polynomials in our original sense, are polynomials by our alternative definition, because they are bounded from above by, e.g., n and n^2 respectively.

Introduction

- **Super Polynomial functions:** The remaining functions.

  E.g. $O(2^n)$, $O(n!)$, $O(n^n)$

- A function of the form k^n is genuinely exponential.
- But now some functions which are worse than polynomial but not quite exponential, such as $O(n^{\log n})$ also fall in this category.
- And some functions which are worse than exponential, such as the super-exponentials, e.g. $O(n^n)$, also fall in this category.

Introduction

- The reasons for lumping functions together into these two broad classes can be elucidated by the following table.

	10	50	100	300	1000
$5n$	50	250	500	1500	5000
$n \times \log n$	33	282	665	2469	9966
n^2	100	2500	10000	90000	1 million (7 digits)
n^3	1000	125000	1 million (7 digits)	27 million (8 digits)	1 billion (10 digits)
2^n	1024	a 16-digit number	a 31-digit number	a 91-digit number	a 302-digit number
$n!$	3.6 million (7 digits)	a 65-digit number	a 161-digit number	a 623-digit number	unimaginably large
n^n	10 billion (11 digits)	an 85-digit number	a 201-digit number	a 744-digit number	unimaginably large

(The number of protons in the known universe has 79 digits.)
(The number of microseconds since the Big Bang has 24 digits.)

Polynomial Running Time and Super Polynomial Running Time

- On the basis of this classification of functions into polynomial and exponential, we can classify algorithms:
- **Polynomial-Time Algorithm:** an algorithm whose order-of-magnitude time performance is bounded from above by a polynomial function of n, where n is the size of its inputs.
- **Super Polynomial-Time Algorithm:** an algorithm whose order-of-magnitude time performance is not bounded from above by a polynomial function of n.

Polynomial Running Time and Super Polynomial Running Time

- The table below depicts the time taken by different algorithms with different growth rate(assume that one instruction can be executed every microsecond).

	10	20	50	100	300
n^2	$\frac{1}{10000}$ second	$\frac{1}{2500}$ second	$\frac{1}{400}$ second	$\frac{1}{100}$ second	$\frac{9}{100}$ second
n^5	$\frac{1}{10}$ second	3.2 seconds	5.2 minutes	2.8 hours	28.1 days
2^n	$\frac{1}{1000}$ second	1 second	35.7 years	400 trillion centuries	a 75-digit number of centuries
n^n	2.8 hours	3.3 trillion years	a 70-digit number of centuries	a 185-digit number of centuries	a 728-digit number of centuries

(The Big Bang was approximately 15 billion years ago.)

Tractable and Intractable Problems

- And, in a similar way, we can classify problems into two broad classes:
- **Tractable Problem:** a problem that is solvable by a polynomial-time algorithm. The upper bound is polynomial.
- **Intractable Problem:** a problem that cannot be solved by a polynomial-time algorithm. The lower bound is exponential.

Tractable and Intractable Problems

- Here are examples of tractable problems (ones with known polynomial-time algorithms):
  - Searching an unordered list
  - Searching an ordered list
  - Sorting a list
  - Multiplication of integers
  - Finding a minimum spanning tree in a graph.

Tractable and Intractable Problems

- Here are examples of intractable problems (ones that have been proven to have no polynomial-time algorithm).
  - Towers of Hanoi: We can prove that any algorithm that solves this problem must have a worst-case running time that is at least $2n - 1$.
  - List all permutations (all possible orderings) of n numbers.
  - Travelling Salesman Problem.

Class P Problem

- Class P problems refer to problems where an algorithm would take a polynomial amount of time to solve, or where Big-O is a polynomial (i.e. O(1), O(n), O(n^2), etc).
- These are problems that would be considered ‘easy’ to solve, and thus do not generally have immense run times.

Class NP Problem

- The class of problems for which an answer can be verified in polynomial time is NP, which stands for "nondeterministic polynomial time.
- The class NP consists of those problems that are verifiable in polynomial time i.e. it is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information.
- Hence, we aren’t asking for a way to find a solution, but only to verify that an alleged solution really is correct.
- Every problem in this class can be solved in exponential time using exhaustive search.
- **Eg: Travelling Salesman Problem**

Class NP-Complete Problem

- A problem is NP-complete if all other problems in NP reduce to it.
- This is a very strong requirement indeed.
- For a problem to be NP-complete, it must be useful in solving every search problem in the world!
- Informally, if any NP-complete problem can be solved in polynomial time, then every problem in NP has a polynomial-time algorithm.

Class NP-Complete Problem

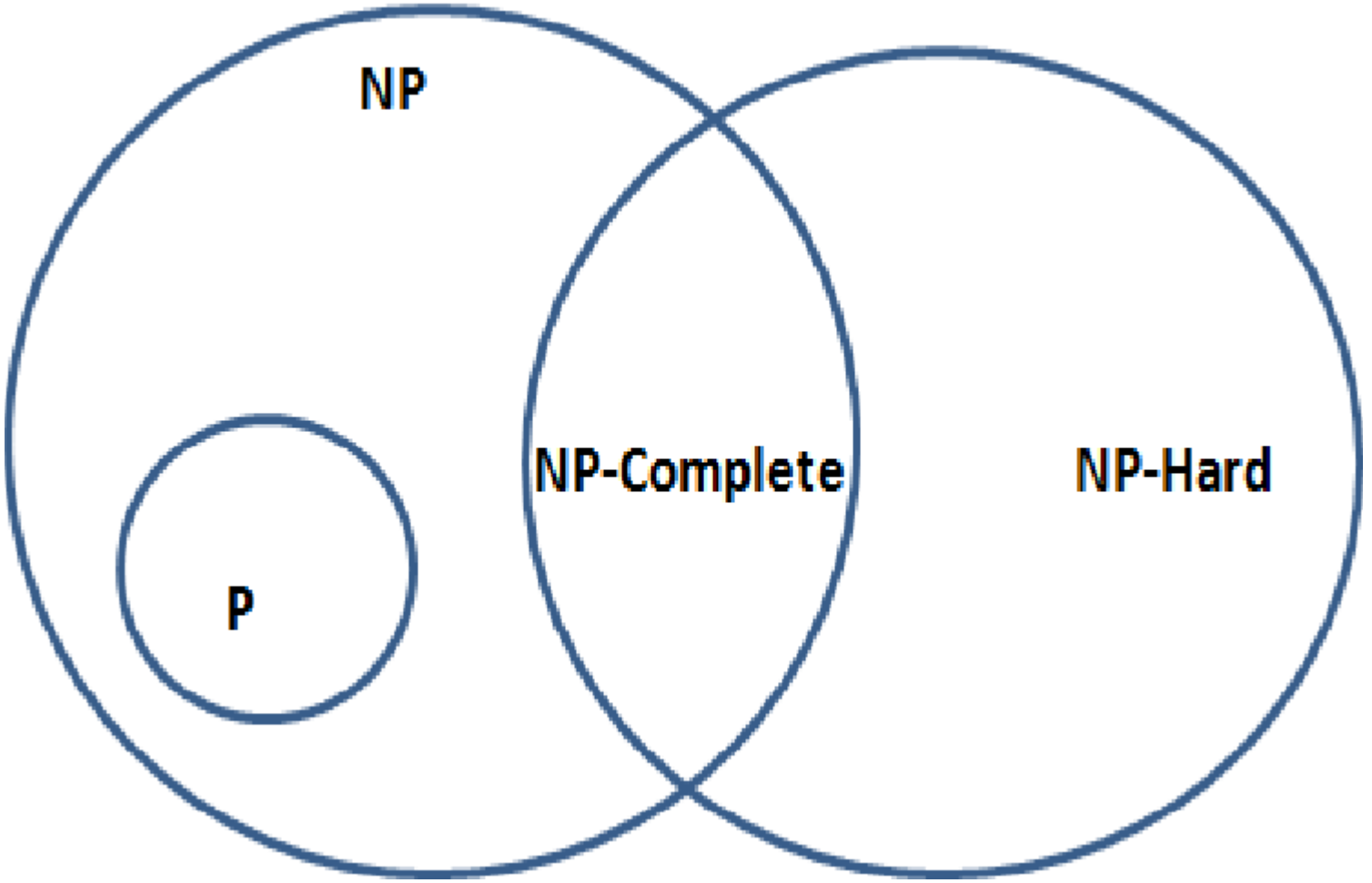
- No polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them.
- This so-called **P ≠ NP** question has been one of the deepest, most perplexing open research problems in theoretical computer science since it was first posed in 1971.

I'D TELL YOU ANOTHER NP-COMPLETE JOKE, BUT ONCE YOU'VE HEARD ONE,

YOU'VE HEARD THEM ALL.

Class NP-Hard Problem

- NP-hardness (non-deterministic polynomial-time hardness) is the defining property of a class of problems that are informally "at least as hard as the hardest problems in NP".
- A simple example of an NP-hard problem is the subset sum problem.


NP
P
NP-Complete
NP-Hard

Cook’s Theorem

- Cook–Levin theorem, also known as Cook's theorem, states that the Boolean satisfiability problem is NP-complete.
- That is, it is in NP, and any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the Boolean satisfiability problem.

Cook's Theorem

What is a Boolean satisfiability problem?

- Boolean satisfiability problem is a problem of determining whether a boolean expression that combines Boolean variables using Boolean operators is **satisfiable** or **unsatisfiable**.
- An expression is **satisfiable** if there is some assignment of truth values to the variables that makes the entire expression true.
- An expression is **unsatisfiable** if it is not possible to assign such values.

Cook’s Theorem

What is a Boolean satisfiability problem?

- A boolean expression A v B’ is **satisfiable** because when A = TRUE and B = FALSE then the expression evaluates to be TRUE.
- A boolean expression A ^ A’ is **unsatisfiable** because it always gives false result.

Cook's Theorem

- An important consequence of this theorem is that **if there exists a deterministic polynomial time algorithm for solving Boolean satisfiability, then every NP problem can be solved by a deterministic polynomial time algorithm(so the complexity class NP would be equal to the complexity class P)**.
- The question of whether such an algorithm for Boolean satisfiability exists is thus equivalent to the **P versus NP problem**, which is widely considered the most important unsolved problem in theoretical computer science.

Reducibility

- Our plan is to explore the space of computationally hard problems, eventually arriving at a mathematical characterization of a large class of them.
- Our basic technique in this exploration is to compare the relative difficulty of different problems; we'd like to formally express statements like, **"Problem X is at least as hard as problem Y."**
- We will formalize this through the notion of reduction: we will show that a particular problem X is at least as hard as some other problem Y by arguing that, if we had a "black box" capable of solving X, then we could also solve Y.

Reducibility

- To make this precise, we add the assumption that X can be solved in polynomial time directly to our model of computation.
- Suppose we had a black box that could solve instances of a problem X; if we write down the input for an instance of X, then in a single step, the black box will return the correct answer. We can now ask the following question:

Can arbitrary instances of problem Y be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to a black box that solves problem X?

Reducibility

- If the answer to this question is yes, then we write $Y \leq_P X$; we read this as “Y is polynomial-time reducible to X,” or “X is at least as hard as Y (with respect to polynomial time).”

Reducibility

- Intuitively, a problem Q can be reduced to another problem Q' if any instance of Q can be "easily rephrased" as an instance of Q', the solution to which provides a solution to the instance of Q.
- For example, the problem of solving linear equations in an indeterminate x reduces to the problem of solving quadratic equations.
- Given an instance **$ax + b = 0$**, we transform it to **$0x^2 + ax + b = 0$**, whose solution provides a solution to **$ax + b = 0$.**
- Thus, if a problem Q reduces to another problem Q', then Q is in a sense, "no harder to solve" than Q'.

Reducibility

An important consequence of our definition of $\leq_P$ is the following.

- Suppose $Y \leq_P X$ and there actually exists a polynomial-time algorithm to solve X. Then, problem Y becomes an algorithm that involves a polynomial number of steps, plus a polynomial number of calls to a subroutine that runs in polynomial time; in other words, it has become a polynomial-time algorithm.
- We have therefore proved the following fact.

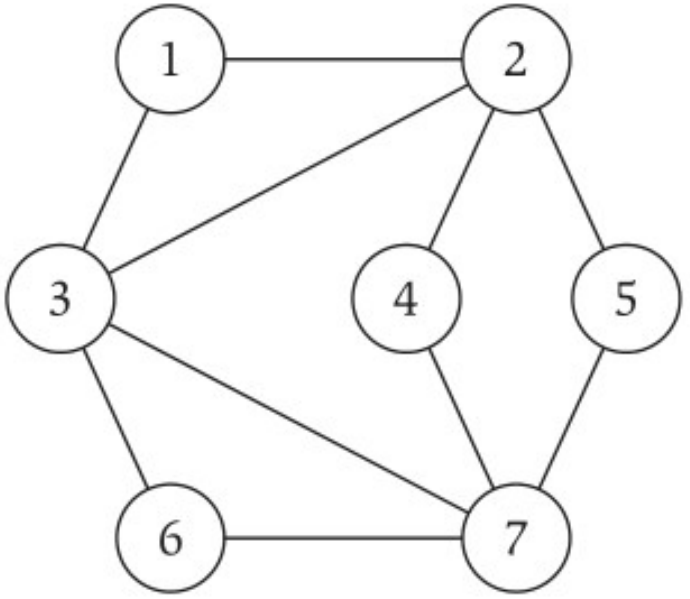
Suppose $Y \leq_P X$. If X can be solved in polynomial time, then Y can be solved in polynomial time.

- Also

Suppose $Y \leq_P X$. If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.

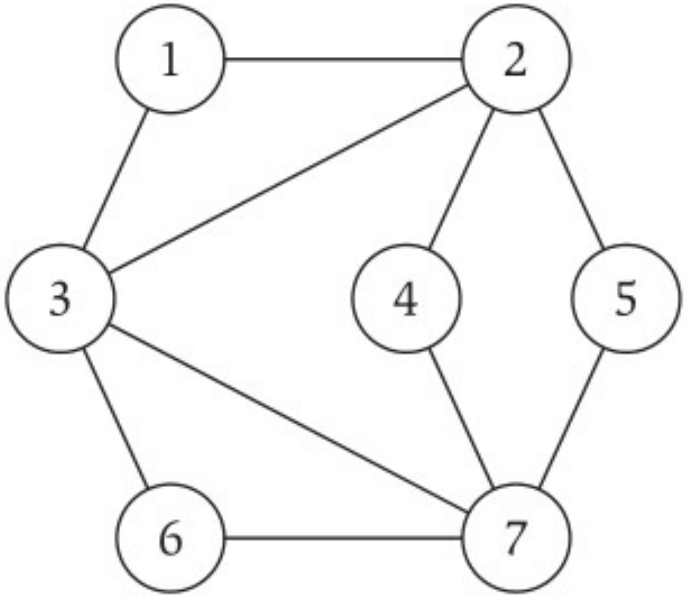
A First Reduction: Independent Set and Vertex Cover

- In a graph G = (V, E), we say a set of nodes S ⊆ V is independent if no two nodes in S are joined by an edge.
- It is easy to find small independent sets in a graph; the hard part is to find a large independent set, since you need to build up a large collection of nodes without ever including two neighbors.
- For example, the set of nodes {3, 4, 5} is an independent set of size 3 in the graph in Figure.
- And the set of nodes {1, 4, 5, 6} is a larger independent set.



A First Reduction: Independent Set and Vertex Cover

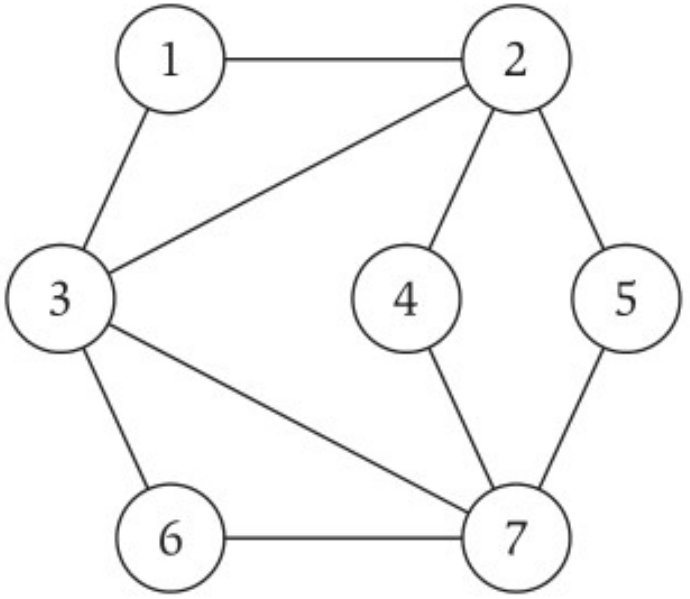
- We phrase Independent Set as follows.



Given a graph G and a number k, does G contain an independent set of size at least k?

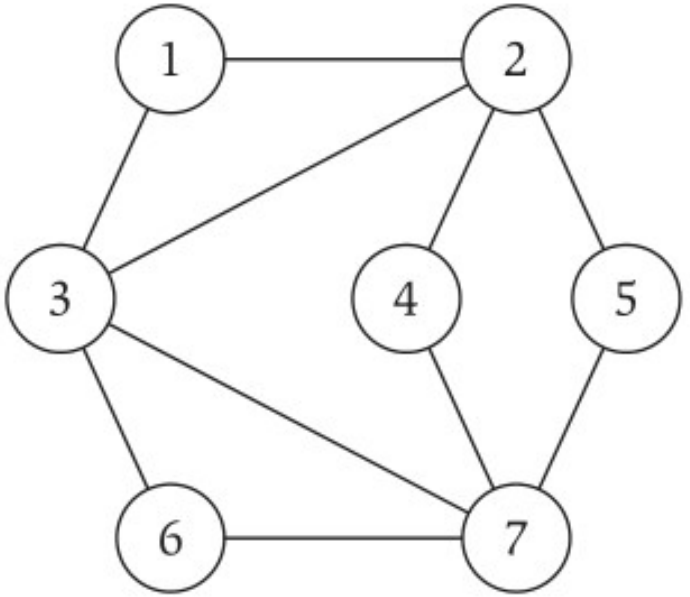
A First Reduction: Independent Set and Vertex Cover

- Given a graph G = (V, E), we say that a set of nodes $S \subseteq V$ is a vertex cover if every edge $e \in E$ has at least one end in S.
- In a vertex cover, the vertices do the "covering," and the edges are the objects being "covered."
- Now, it is easy to find large vertex covers in a graph (for example, the full vertex set is one); the hard part is to find small ones.
- In the graph in Figure, the set of nodes {1, 2, 6, 7} is a vertex cover of size 4, while the set {2, 3, 7} is a vertex cover of size 3.



A First Reduction: Independent Set and Vertex Cover

- We formulate the Vertex Cover problem as follows



Given a graph G and a number k, does G contain a vertex cover of size at most k?

A First Reduction: Independent Set and Vertex Cover

- We now show that they are equivalently hard, by establishing that

  Independent Set $\leq_P$ Vertex Cover and Vertex Cover $\leq_P$ Independent Set

- This will be a direct consequence of the following fact

- Let G = (V, E) be a graph. Then S is an independent set if and only if its complement V − S is a vertex cover.

A First Reduction: Independent Set and Vertex Cover

Let G = (V, E) be a graph. Then S is an independent set if and only if its complement V − S is a vertex cover.

Proof:

- First, suppose that S is an independent set. Consider an arbitrary edge e = (u, v). Since S is independent, it cannot be the case that both u and v are in S; so one of them must be in V − S. It follows that every edge has at least one end in V − S, and so V − S is a vertex cover.

- Conversely, suppose that V − S is a vertex cover. Consider any two nodes u and v in S. If they were joined by edge e, then neither end of e would lie in V − S, contradicting our assumption that V − S is a vertex cover. It follows that no two nodes in S are joined by an edge, and so S is an independent set.

A First Reduction: Independent Set and Vertex Cover

Independent Set $\leq_P$ Vertex Cover.

Proof: If we have a black box to solve Vertex Cover, then we can decide whether G has an independent set of size at least k by asking the black box whether G has a vertex cover of size at most n − k.

Vertex Cover $\leq_P$ Independent Set.

Proof: If we have a black box to solve Independent Set, then we can decide whether G has a vertex cover of size at most k by asking the black box whether G has an independent set of size at least n − k.

Conclusion: Although, we don't know how to solve either Independent Set or Vertex Cover efficiently, above tell us how we could solve either given an efficient solution to the other, and hence these two facts establish the relative levels of difficulty of these problems.

The Satisfiability Problem

- Suppose we are given a set X of n Boolean variables $x_1, \ldots, x_n$; each can take the value 0 or 1 (equivalently, “false” or “true”).
- By a term over X, we mean one of the variables x_i or its negation $\sim x_i$.
- Finally, a clause is simply a disjunction of distinct terms

$$t_1 \vee t_2 \vee \cdots \vee t_\ell$$

$$\text{each } t_i \in \{x_1, x_2, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\}$$

- We say the clause has length ℓ if it contains ℓ terms.

The Satisfiability Problem

- A truth assignment for X is an assignment of the value 0 or 1 to each x_i. In other words, it is a function $\nu : X \rightarrow \{0, 1\}$.
- An assignment satisfies a clause C if it causes C to evaluate to 1 under the rules of Boolean logic. In simple words, 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 $C_1, \ldots, C_k$ if it causes all of the C_i to evaluate to 1. In simple words, if it causes the conjunction

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k$$

- to evaluate to 1. In this case, we will say that ν is a satisfying assignment with respect to $C_1, \ldots, C_k$ and that the set of clauses $C_1, \ldots, C_k$ is satisfiable.

The Satisfiability Problem

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 ν that sets all variables to 1 is not a satisfying assignment, because it does not satisfy the second of these clauses
- The truth assignment ν′ that sets all variables to 0 is a satisfying assignment.

The Satisfiability Problem

- We can now state the Satisfiability Problem, also referred to as SAT:

Given a set of clauses $C_1, \ldots, C_k$ over a set of variables $X = \{x_1, \ldots, x_n\}$, does there exist a satisfying truth assignment?

The Satisfiability Problem

- 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 $C_1, \ldots, C_k$, each of length 3, over a set of variables $X = \{x_1, \ldots, x_n\}$, does there exist a satisfying truth assignment?

Reducing 3-SAT to Independent Set

- Now, we will show that 3-SAT $\leq_P$ Independent Set.
- The difficulty in proving a thing like this is clear; 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.

3-SAT $\leq_P$ Independent Set

Proof:

- We have a black box for Independent Set and want to solve an instance of 3-SAT consisting of variables $X = \{x_1, \ldots, x_n\}$ and clauses $C_1, \ldots, C_k$.
- The key to thinking about the reduction is to realize that there are two conceptually distinct ways of thinking about an instance of 3-SAT.

3-SAT $\leq_P$ Independent Set

Proof:

- We have a black box for Independent Set and want to solve an instance of 3-SAT consisting of variables $X = \{x_1, \ldots, x_n\}$ and clauses $C_1, \ldots, C_k$.
- The key to thinking about the reduction is to realize that there are two conceptually distinct ways of thinking about an instance of 3-SAT.

First way,

- You have to make an independent 0/1 decision for each of the n variables, and you succeed if you manage to achieve one of three ways of satisfying each clause.

3-SAT $\leq_P$ Independent Set

Proof:

Second Way,

- You 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.
- So you succeed if you 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 $\sim x_i$.
- If we avoid conflicting terms, we can find a truth assignment that makes the selected terms from each clause evaluate to 1.

3-SAT $\leq_P$ Independent Set

Proof:

Second Way,

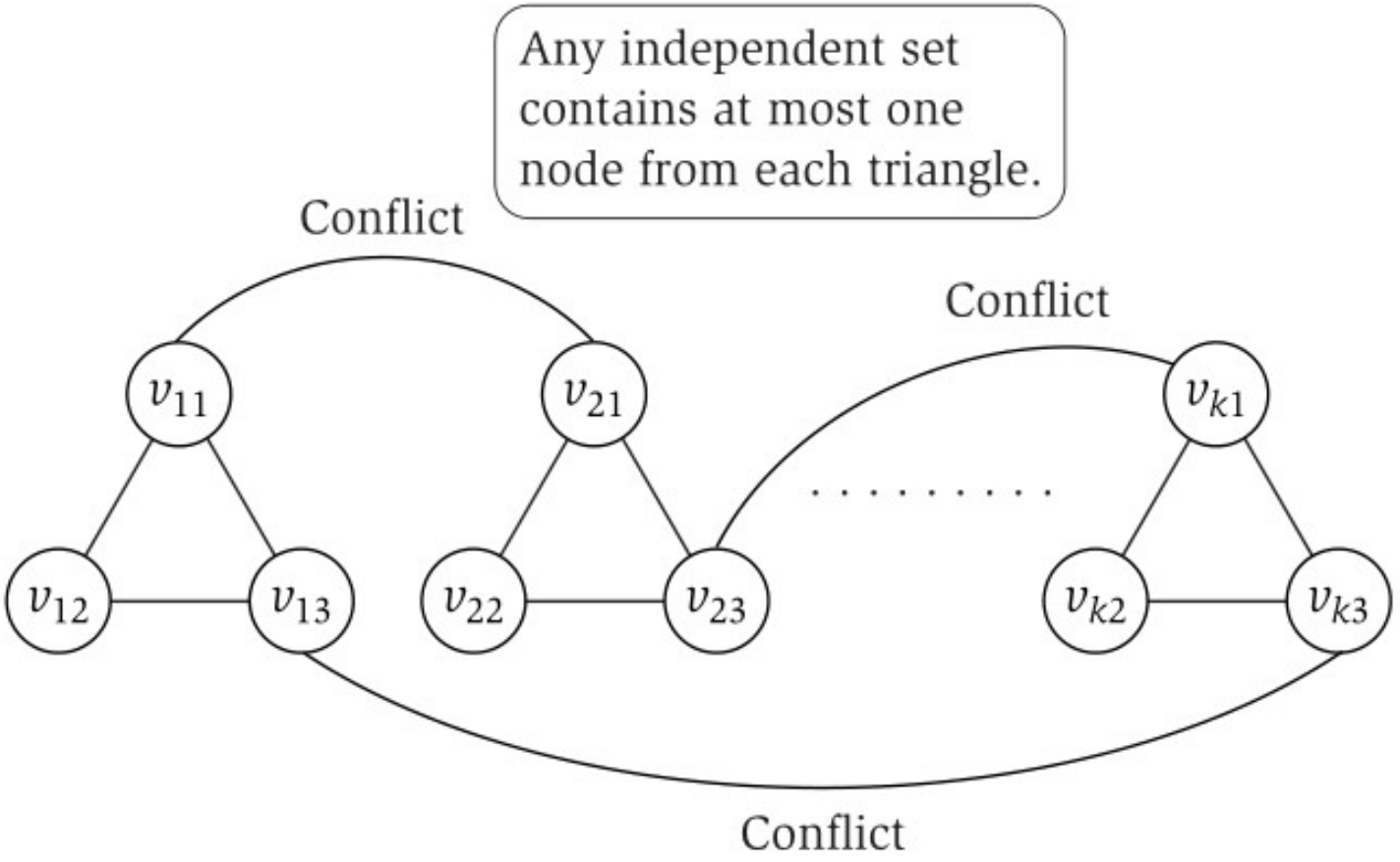
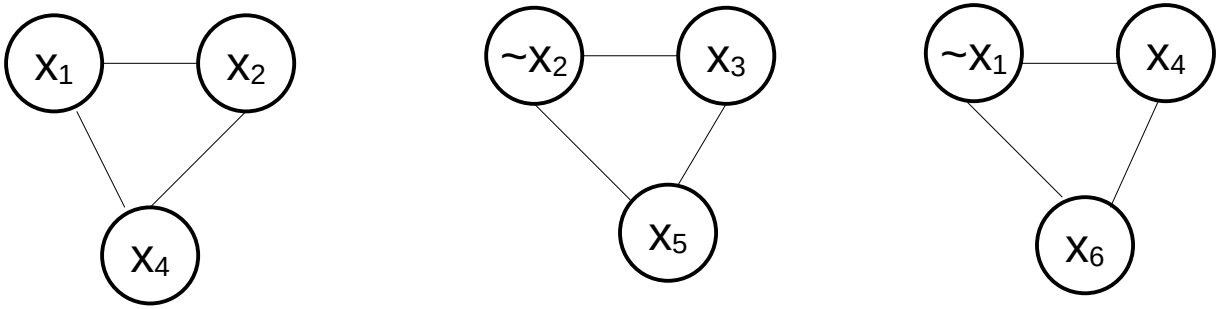


Figure The reduction from 3-SAT to Independent Set.

3-SAT $\leq_P$ Independent Set

Proof:

- Our reduction will be based on this second view of the 3-SAT instance; here is how we encode it using independent sets in a graph.
- First, construct a graph G = (V, E) consisting of 3k nodes grouped into k triangles as shown in Figure

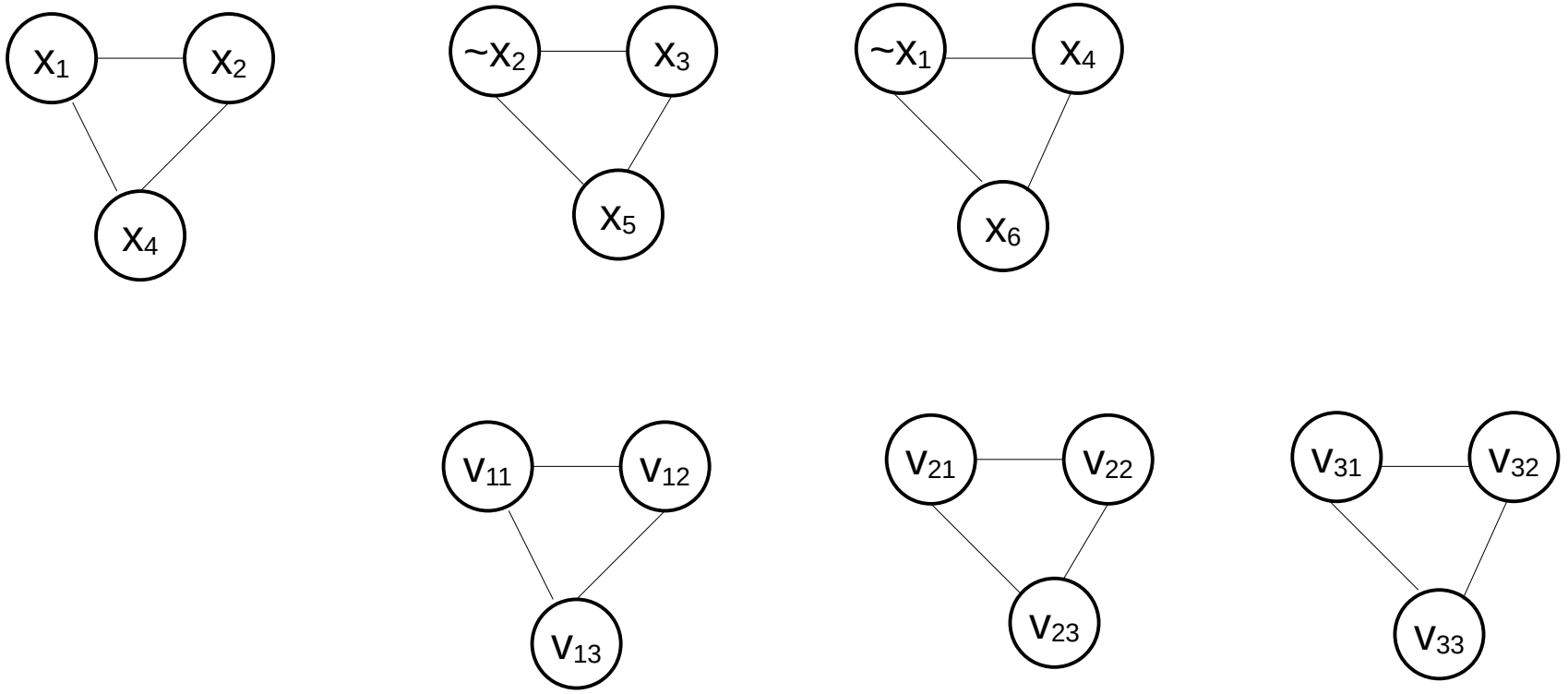


$(x_1 \vee x_2 \vee x_4) \wedge (\sim x_2 \vee x_3 \vee x_5) \wedge (\sim x_1 \vee x_4 \vee x_6)$.

3-SAT $\leq_P$ Independent Set

Proof:

- That is, for i = 1, 2, . . . , k, we construct three vertices v_{i1}, v_{i2}, v_{i3} joined to one another by edges. We give each of these vertices a label; v_{ij} is labeled with the j^{th} term from the clause C_i of the 3-SAT instance.

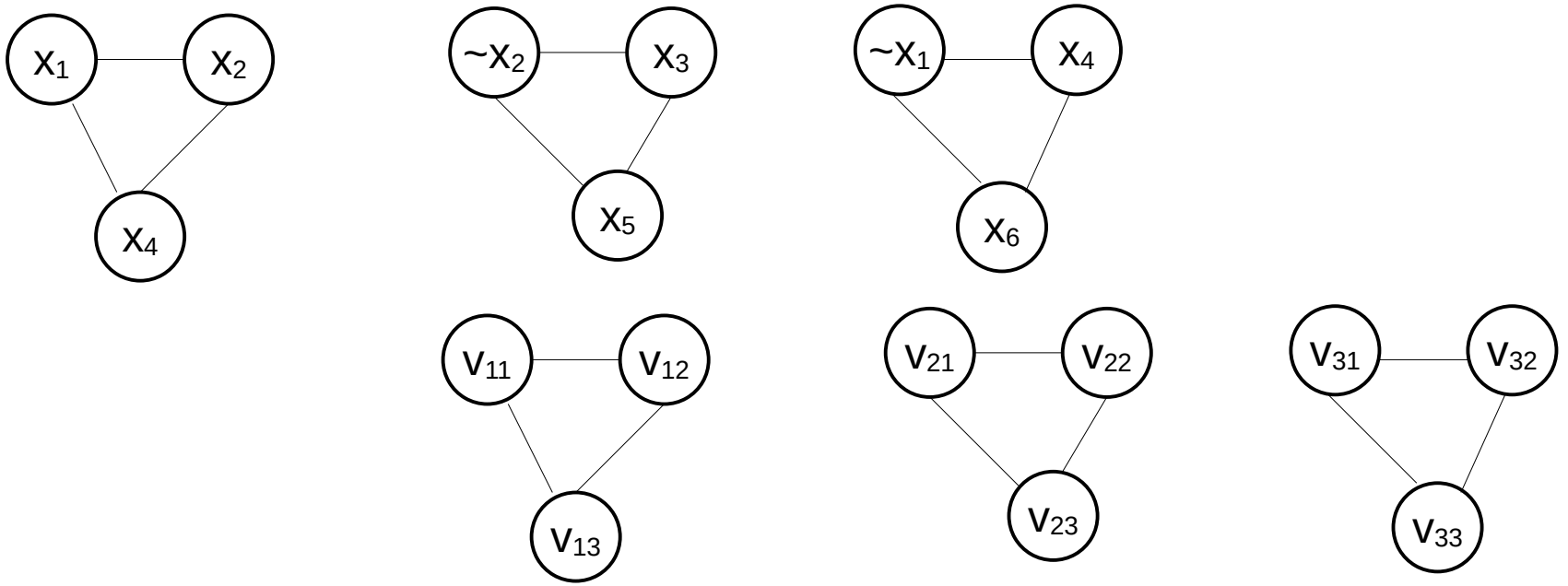


$(x_1 \vee x_2 \vee x_4) \wedge (\sim x_2 \vee x_3 \vee x_5) \wedge (\sim x_1 \vee x_4 \vee x_6)$.

3-SAT $\leq_P$ Independent Set

Proof:

- Before proceeding, consider what the independent sets of size k look like in this graph: Since two vertices cannot be selected from the same triangle, they consist of all ways of choosing one vertex from each of the triangles.
- This is implementing our goal of choosing a term in each clause that will evaluate to 1; but we have so far not prevented ourselves from choosing two terms that conflict.

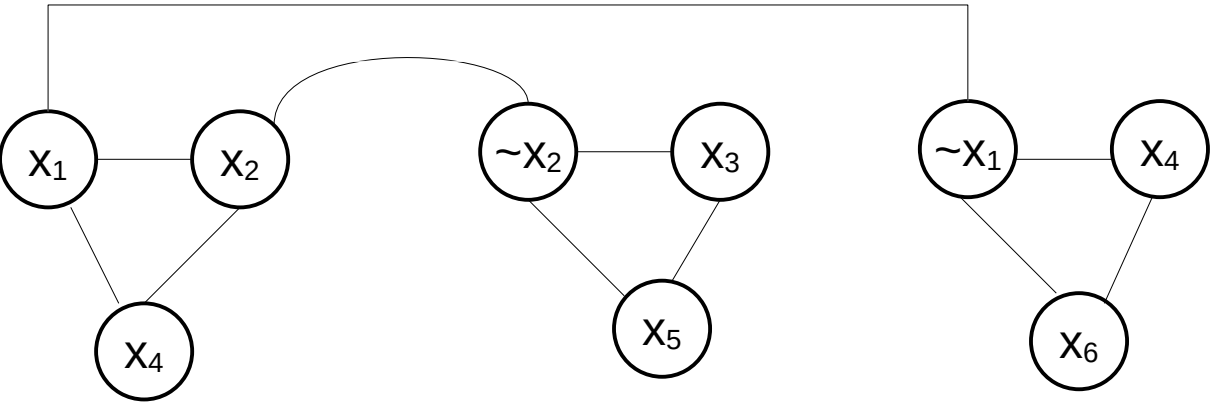


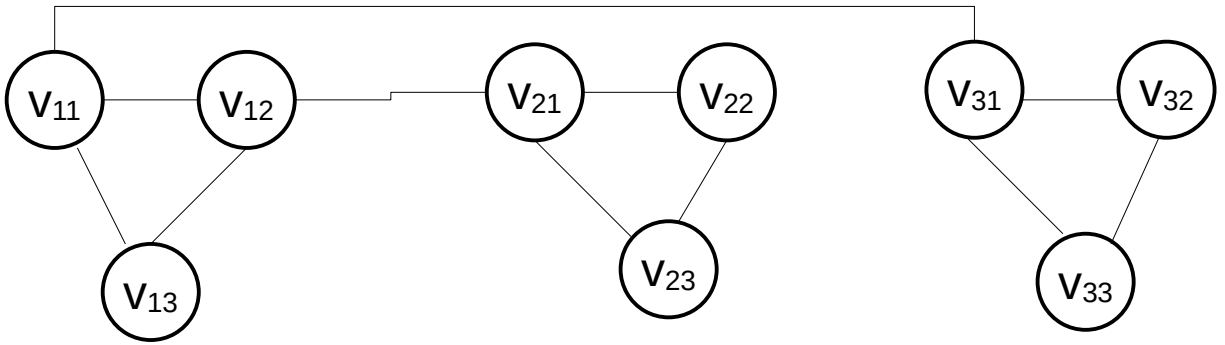
$(x_1 \vee x_2 \vee x_4) \wedge (\sim x_2 \vee x_3 \vee x_5) \wedge (\sim x_1 \vee x_4 \vee x_6)$.

3-SAT ≤P Independent Set

Proof:

- We encode conflicts by adding some more edges to the graph: For each pair of vertices whose labels correspond to terms that conflict, we add an edge between them.
- Have we now destroyed all the independent sets of size k, or does one still exist? It's not clear; it depends on whether we can still select one node from each triangle so that no conflicting pairs of vertices are chosen.



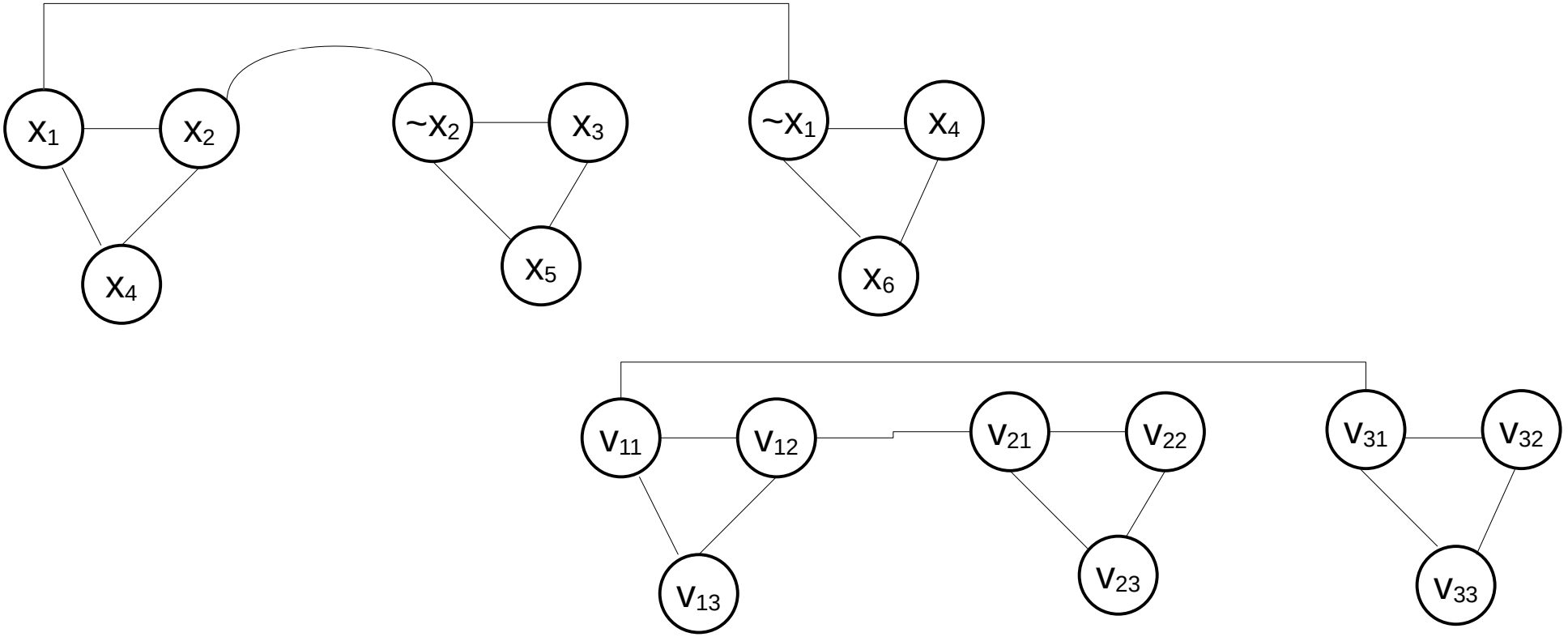


$(x_1 \vee x_2 \vee x_4) \wedge (\sim x_2 \vee x_3 \vee x_5) \wedge (\sim x_1 \vee x_4 \vee x_6)$.

3-SAT $\leq_P$ Independent Set

Proof:

- Let's claim, precisely, that 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.

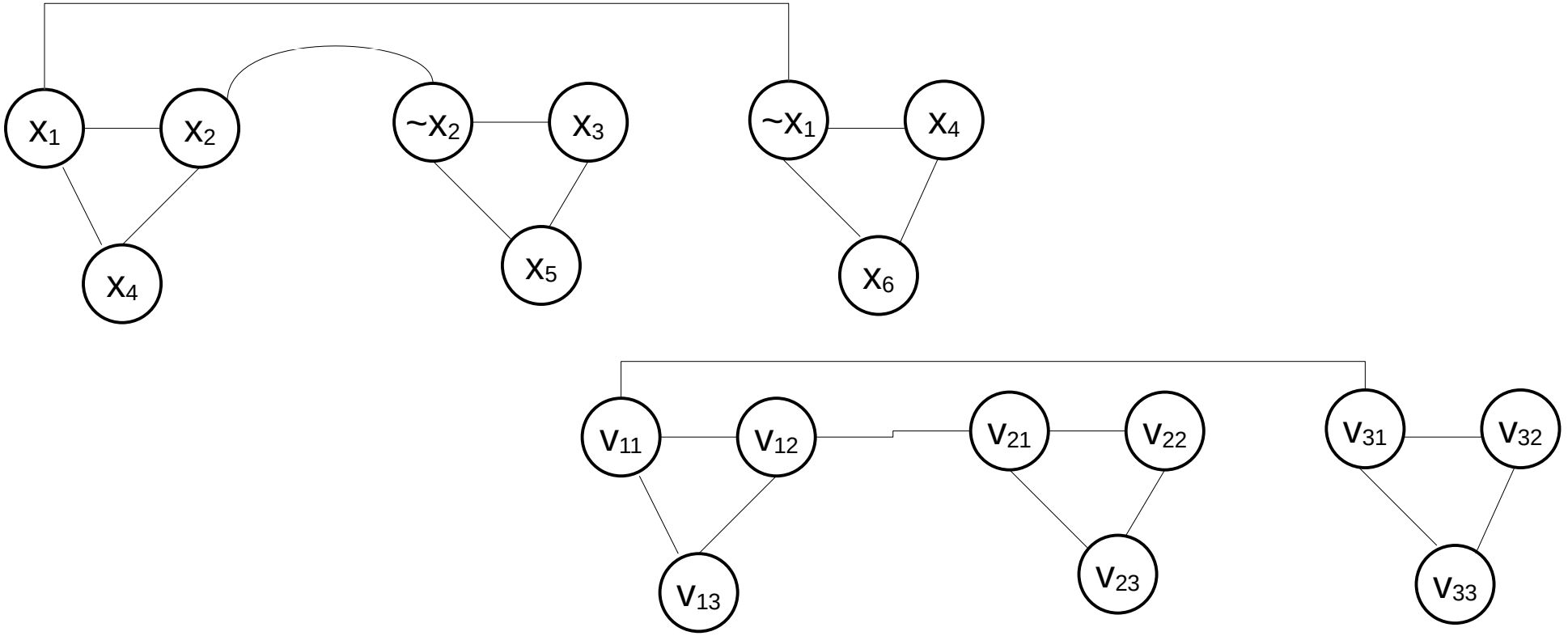


$(x_1 \vee x_2 \vee x_4) \wedge (\sim x_2 \vee x_3 \vee x_5) \wedge (\sim x_1 \vee x_4 \vee x_6)$.

3-SAT $\leq_P$ Independent Set

Proof:

- First, 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; for 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.

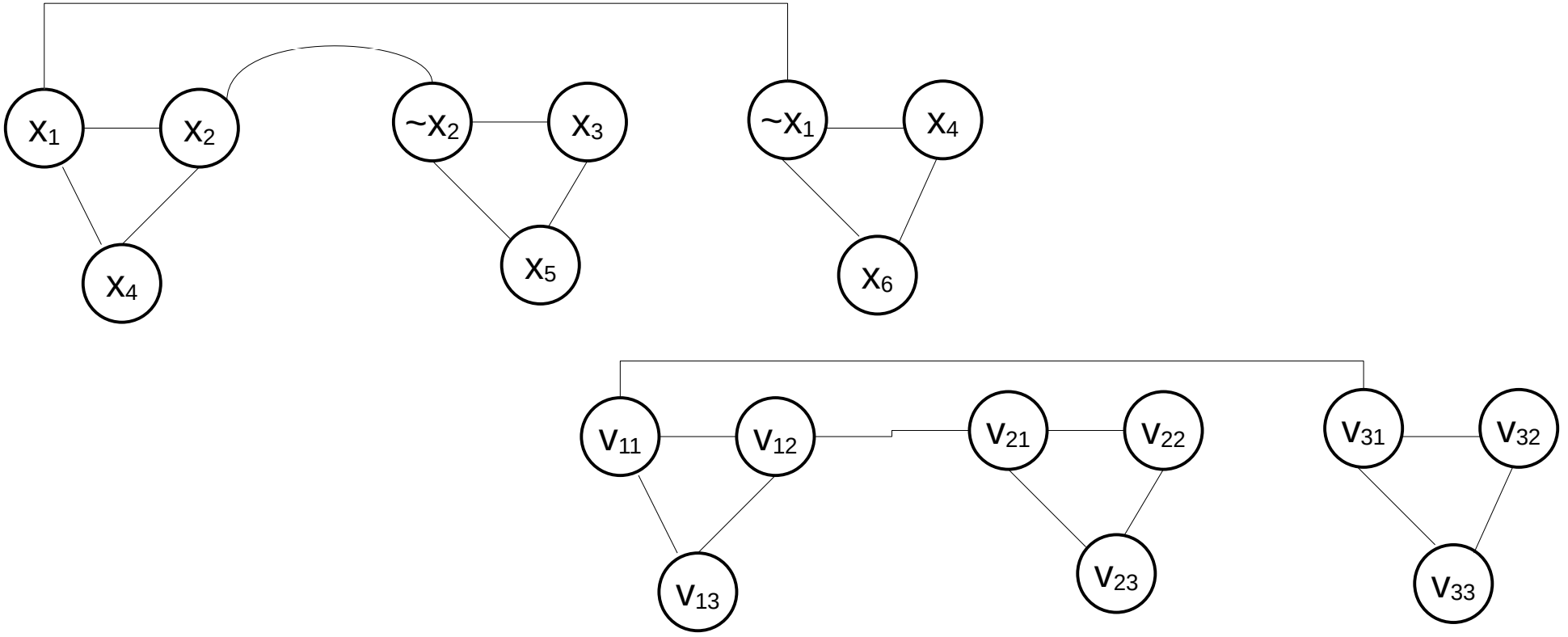


$(x_1 \vee x_2 \vee x_4) \wedge (\sim x_2 \vee x_3 \vee x_5) \wedge (\sim x_1 \vee x_4 \vee x_6)$.

3-SAT $\leq_P$ Independent Set

Proof:

- 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. Now, we claim that there is a truth assignment ν for the variables in the 3-SAT instance with the property that the labels of all nodes in S evaluate to 1. Here is how we could construct such an assignment ν.
- If x_i appears as a label of a node in S, we set $\nu(x_i) = 1$, and otherwise we set $\nu(x_i) = 0$. By constructing ν in this way, all labels of nodes in S will evaluate to 1.



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

3-SAT $\leq_P$ Independent Set

Proof:

- Since G has an independent set of size at least k if and only if the original 3-SAT instance is satisfiable, the reduction is complete.

Transitivity of Reductions

If $Z \leq_P Y$, and $Y \leq_P X$, then $Z \leq_P X$.

Proof:

- Given a black box for X, we show how to solve an instance of Z essentially, we just compose the two algorithms implied by $Z \leq_P Y$ and $Y \leq_P X$.
- We run the algorithm for Z using a black box for Y; but each time the black box for Y is called, we simulate it in a polynomial number of steps using the algorithm that solves instances of Y using a black box for X.

Transitivity of Reductions

- Transitivity can be quite useful.
- For example, since we have proved

  3-SAT $\leq_P$ Independent Set $\leq_P$ Vertex Cover

  we can conclude that 3-SAT $\leq_P$ Vertex Cover

General Strategy for Proving New Problems NP-Complete

- Given a new problem X, here is the basic strategy for proving it is NP-complete.

  1. Prove that $X \in NP$.
  2. Choose a problem Y that is known to be NP-complete.
  3. Prove that $Y \leq_P X$.

General Strategy for Proving New Problems NP-Complete

- We can refine the strategy above to the following outline of an NP-completeness proof.

  1. Prove that $X \in NP$.

  2. Choose a problem Y that is known to be NP-complete.

  3. Consider an arbitrary instance s_Y of problem Y, and show how to construct, in polynomial time, an instance s_X of problem X that satisfies the following properties:

  (a) If s_Y is a “yes” instance of Y, then s_X is a “yes” instance of X.

  (b) If s_X is a “yes” instance of X, then s_Y is a “yes” instance of Y.

  In other words, this establishes that s_Y and s_X have the same answer.

Review Questions

1) Explain the statement.

“Boolean satisfiability problem is NP-complete.”

2) Explain tractable and intractable problems.

3) Explain the complexity classes, P, NP, NP Complete and NP-Hard.

4) Show that Independent Set $\leq_P$ Vertex Cover.