Chapter 8

Introduction to Computational Complexity

Introduction to Computational Complexity

- A decision problem is decidable if there is an algorithm that can answer it in principle
- We will try to identify the problems for which there are *practical* algorithms
  - Ones that can answer reasonable-size instances in a reasonable amount of time
- For example, the *Sudoku puzzle* is decidable, but the known algorithms aren't much of an improvement on the brute-force algorithm that takes exponential time on *general instances*
- **Our focus is the running time (not the problems that cannot be solved!)**

The Time Complexity of a Turing Machine, and the Set *P*

- The set *P* is the set of problems that can be decided by a TM in *polynomial time* (number of moves), as a function of the instance size.
- *NP* is defined similarly, except that we allow the use of a *nondeterministic* TM
- Most people assume that *NP* is a larger set, but no one has been able to demonstrate that $P \neq NP$
- We will discuss different classes of hardness of problems, and will learn how to compare algorithms.
- In this chapter: **Our focus is the running time (not the problems that cannot be solved!)**

The Time Complexity of a Turing Machine

- A TM deciding a language $L \subseteq \Sigma^*$ solves a decision problem: Given $x \in \Sigma^*$, is $x \in L$?
  - A measure of the size of the problem is the length of the input string x

The Time Complexity of a Turing Machine

- A TM deciding a language $L \subseteq \Sigma^*$ solves a decision problem: Given $x \in \Sigma^*$, is $x \in L$?
  - A measure of the size of the problem is the length of the input string x
- **Definition**: Suppose T is a TM with input alphabet Σ that eventually halts on every input string
  - The *time complexity* of T is the function

$$\tau_T : \mathbb{N} \to \mathbb{N},$$

input string length n

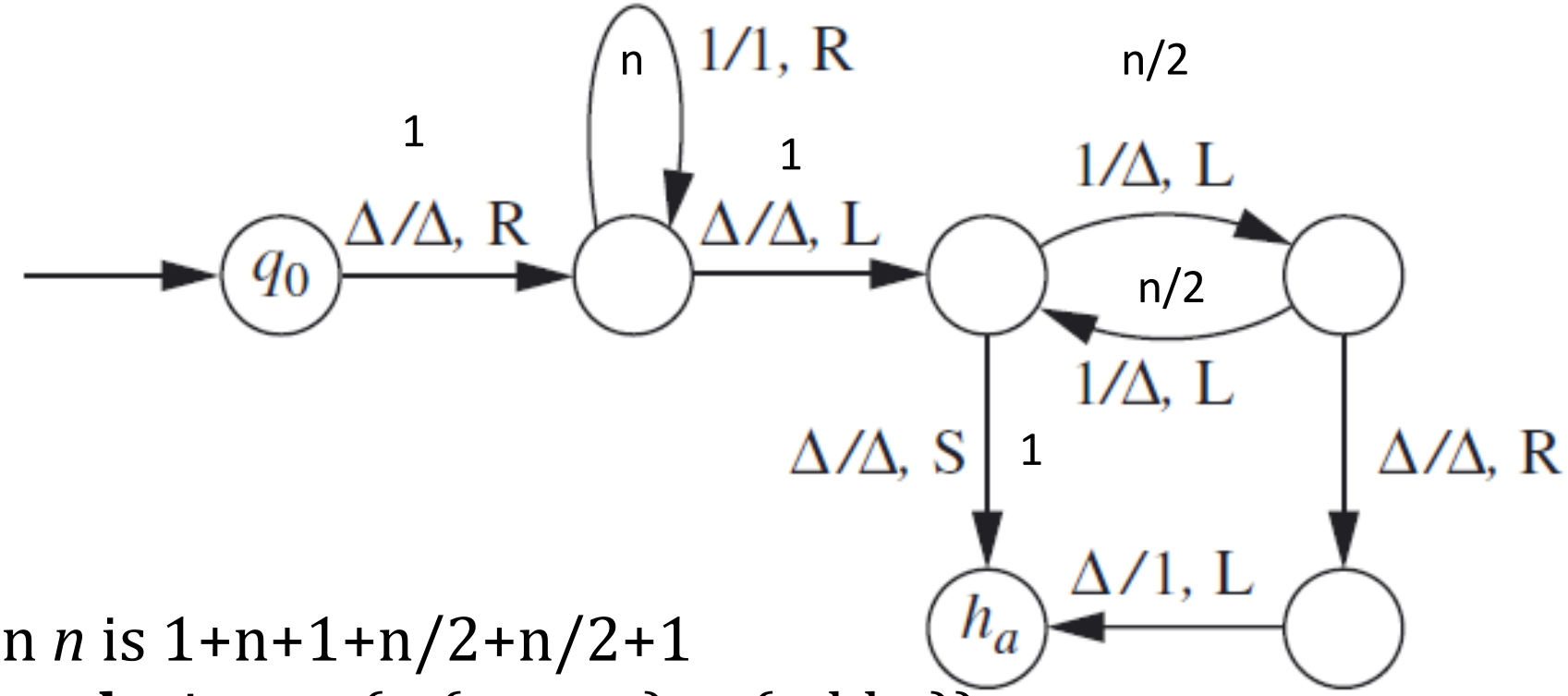
the number of moves T makes on that string of length n before halting, and letting $\tau_T(n)$ be the maximum of these numbers

TM with a certain time complexity halts on every input.

Example: TM for computing the remainder Mod 2

Moves the tape head to the end of the string, then makes a pass from right to left in which the 1's are counted and erased by pairs. The final output is a single 1 if the input was odd and nothing otherwise.

Input: 11…1 of length *n* where *n* is even.



$\tau_T(n)$ for even n is 1+n+1+n/2+n/2+1

$\tau_T(n)$ for **general** n is $\max(\tau_T(\text{even } n), \tau_T(\text{odd } n))$

The Time Complexity of a Turing Machine

- Definition: If f and g are partial functions from $\mathbb{N}$ to $\mathbb{R}^+$; that is, both functions have values that are nonnegative real numbers wherever they are defined

  We say that

  $$f \in O(g), \text{ or } f(n) \in O(g(n))$$

  (which we read "f is big-oh of g")

  if, for some positive numbers C and N,

  $$f(n) \leq C\, g(n) \text{ for every } n \geq N.$$

  – For example, every polynomial of degree k with positive leading coefficient is $O(n^k)$

Example of Big-*O* proof

The number of steps of some TM is

$$f(n) = n^2 + \frac{5n}{2} + 4$$

For every $n \geq 1$

$$n^2 + \frac{5n}{2} + 4 \leq n^2 + \frac{5n^2}{2} + 4n^2 = 7.5n^2$$

Thus,

$$f(n) \in O(n^2)$$

because we found such $C = 7.5$, and $N = 1$ for $g(n) = n^2$.

Example of Big-*O* proof

The number of steps of some TM is

$$f(n) = n^3 + 2n + 1$$

Does it belong to $O(n)$?
If it is true then there exist appropriate constants C and N, i.e.,

$$n^3 + 2n + 1 \leq C \cdot n$$

for every $n \geq N$. Let us take $n = C + N$

$$(C+N)^3 + 3(C+N) + 1 \leq C(C+N)$$

$$C^3 + 3C^2N + 3CN^2 + N^3 + 3C + 3N + 1 \leq C^2 + CN$$

$$C^3 + 3(N-1)C^2 + 3CN^2 + N^3 + 3C + 3N - CN + 1 \leq 0$$

Sum is positive

false

Fast way to obtain *O(...)* for some *f(n)* is to find its dominant term

Expression	Dominant term	$O(...)$
$5 + 0.001n^3 + 0.025n$	$0.001n^3$	$O(n^3)$
$500n + 100n^{1.5} + 50n \log n$	$100n^{1.5}$	$O(n^{1.5})$
$n^2 \log n + n(\log n)^2$	$n^2 \log n$	$O(n^2 \log n)$
$10^8 n + 0.000000001n^2$	$0.000000001n^2$	$O(n^2)$
$10^{10} n \log n + n(\log n)^2$	$n(\log n)^2$	$O(n(\log n)^2)$
$7 \cdot 2^n + 10^{100} n^{1000}$	$7 \cdot 2^n$	$O(2^n)$

Properties of $O(\ldots)$

- Is it true that $O.5n + 8n^2 + 100n^3 \in O(n^4)$

  True

- What is $O(f+g)$?

  $O(f+g) = max(O(f), O(g))$

- Is it true that $O(fg) = O(f)\ O(g)$?

  True

- Is it true that if $g \in O(f)$ and $h \in O(f)$ then $g \in O(h)$?

  False

  if $g \in O(f)$ and $f \in O(h)$ then $g \in O(h)$

Logical connectives: reminder

Boolean variables

AND OR

p	q	$p \wedge q$	$p \vee q$
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

The Time Complexity of a Turing Machine

- An instance of the *satisfiability problem* (SAT) is a Boolean expression
  - It involves Boolean variables $x_1, x_2, \ldots, x_n$ and the logical connectives $\wedge$, $\vee$, and $\neg$
  - It is in conjunctive normal form (the conjunction of several clauses, each of which is a disjunction).

  Example:

  Expression in CNF: $(x_1 \vee x_2 \vee \neg x_5) \wedge (x_1 \vee \neg x_2 \vee \neg x_6 \vee x_3 \vee x_4)$

  Assignment: x_1=1, x_2=0, x_3=1, x_4=0, x_5=1, x_6=1

  Result: $(1 \vee 0 \vee 0) \wedge (1 \vee 1 \vee 0 \vee 1 \vee 0) = 1 \wedge 1 = 1$ (i.e., the expression is satisfied)

The Time Complexity of a Turing Machine

- An instance of the *satisfiability problem* (SAT) is a Boolean expression
  - It involves Boolean variables $x_1, x_2, \ldots, x_n$ and the logical connectives $\wedge$, $\vee$, and $\neg$
  - It is in conjunctive normal form (the conjunction of several clauses, each of which is a disjunction).

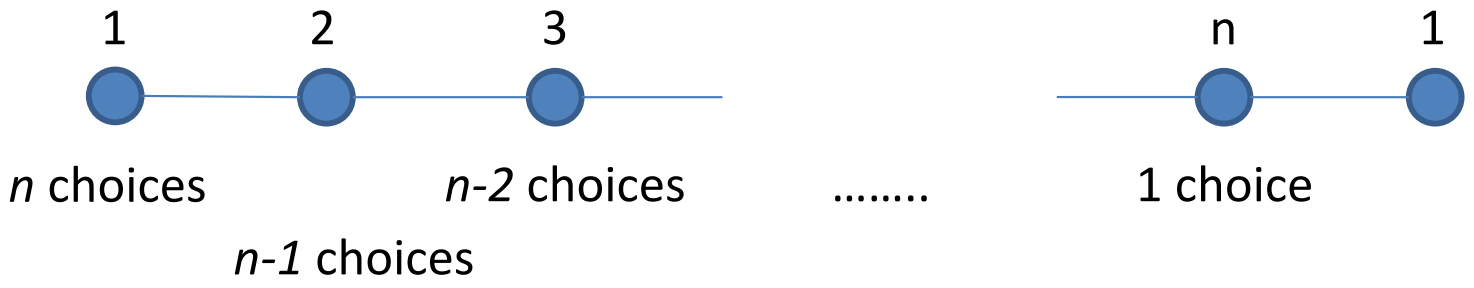
    Example: $(x_1 \vee x_2 \vee \neg x_5) \wedge (x_1 \vee \neg x_2 \vee \neg x_6 \vee x_3 \vee x_4)$
- SAT Problem: Is there an assignment of 1/0 to the variables that satisfies the expression (makes it true)?
  - Is this problem decidable? Yes.
  - Decision algorithm? Try every possible assignment of values to variables (total 2^n assignments)

- The *traveling salesman problem* (TSP) considers *n* cities that a salesman must visit, with a distance specified for every pair of cities.
- **TSP question**: what is the shortest possible route that visits each city exactly once and returns to the origin city?
  - It's easy to formulate this as an optimization problem
    - Determine the order that minimizes the total distance traveled

- How to turn TSP into decision problem?

  Introduce a variable *k* and ask whether there is an order in which the cities could all be visited by traveling no more than distance *k*

- The *traveling salesman problem* (TSP) considers *n* cities that a salesman must visit, with a distance specified for every pair of cities.
- **Decision TSP question**: is there a route that visits each city exactly once and returns to the origin city that is not longer than *k* ?
- Is this problem decidable? Yes. What is the algorithm?
- There's a brute-force solution to the TSP problem too
  - Consider all *n*! possible permutations of the cities



- With current hardware we can solve very large problems, if the problems require time $O(n)$
- We can still solve largish problems if they take time $O(n^2)$ or even $O(n^3)$
- Exponential time algorithms are another story
  - If the problem really requires time proportional to 2^n, then even doubling the speed of the machine doesn't help much.
  - *Try to write a simple $O(2^n)$ algorithm to satisfy a SAT formula. Check if you will be able to run it for 12-20 variables.*

- **However, showing that a brute-force approach takes a long time does not necessarily mean that the problem is complex**
  - The SAT and TSP problems are assumed to be hard, not because the brute-force approach takes exponential or factorial time, but because *we don't know a way of solving either problem that doesn't take at least exponential time*

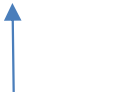
What problems are *tractable*?

- The most common answer is those that can be solved in *polynomial time* on a TM or a comparable computer

- One reason for this characterization is that it is relatively robust, as problems that can be solved in polynomial time on any computer can be solved in polynomial time on a TM as well, and vice-versa

- There are many theorems that look like

There is _____-time overhead to convert a code in _____ to TM.



polynomial
linear
logarithmic
…

Python
C++
electric circuit
…

The Set *P*

- **Definition**: P is the set of languages L such that for some TM T deciding L and some $k \in \mathbb{N}$,
$$\tau_T(n) \in O(n^k)$$

- Maybe the SAT and TSP problems seem to be good candidates for real-life problems that are not in P?

The Set *NP* and Polynomial Verifiability

- TSP seems like a hard problem but testing a potential answer is easy (and there are many potential answers)
- We can approach this problem nondeterministically
  - We guess an answer (a particular route) and then test it deterministically
  - This can be done in polynomial time

O(poly(*n*))

The Set *NP* and Polynomial Verifiability

- **Definition**: If T is an NTM with input alphabet Σ such that, for every $x \in \Sigma^*$, every possible sequence of moves of T on input x eventually halts, the time complexity is defined as follows

$$\tau_T : \mathbb{N} \rightarrow \mathbb{N}$$

Input length

Number of moves

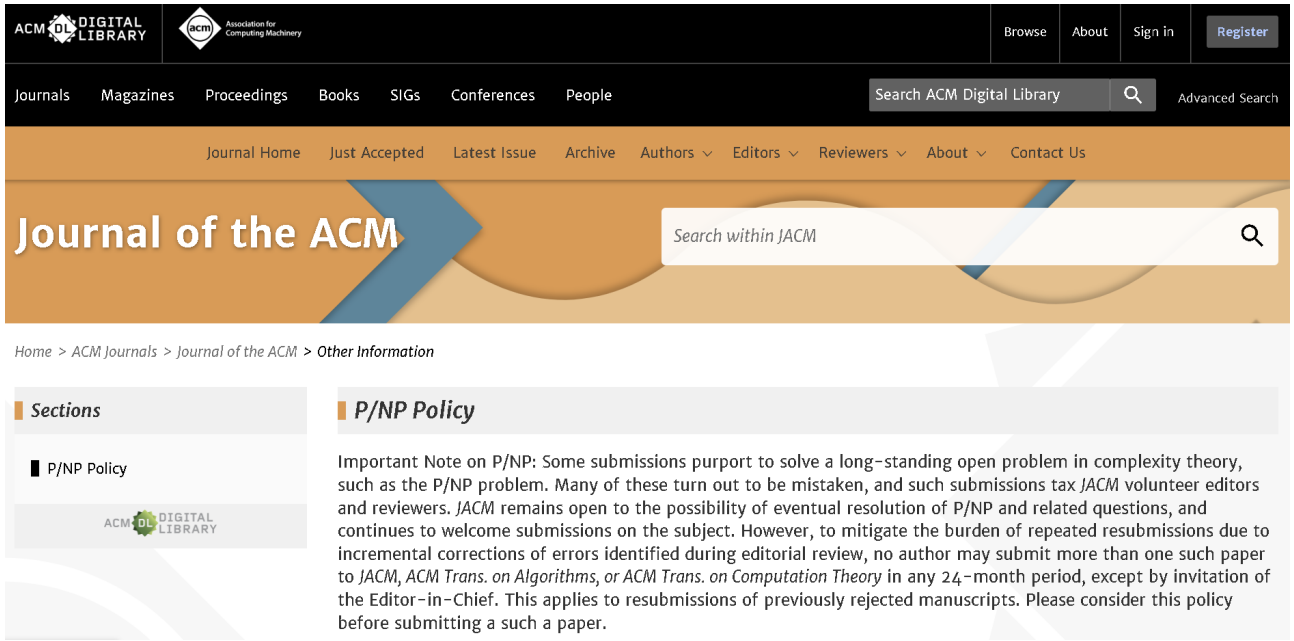
  - Let $\tau_T(n)$ be the maximum number of moves T can possibly make on any input string of length n before halting

(We are assuming implicitly that no input string can cause it to loop forever)

- **Definition**: *NP* is the set of languages L such that for some NTM T that cannot loop forever on any input, and some integer k, T accepts L and

  $\tau_T(n) = O(n^k)$

  - We say that a language in *NP* can be accepted in *nondeterministic polynomial time*
  - It is clear that $P \subseteq NP$
  - **But if you can prove or disprove that $NP \subseteq P$ then you'll get an A in this course, $1M from the Clay Institute, a full professor position in any school, Turing award, etc.**
  - The *SAT* problem is in *NP* (the "guess-and-test" strategy is typical of problems in *NP*, and we can formalize this by constructing an appropriate NTM)



Important Note on P/NP: Some submissions purport to solve a long-standing open problem in complexity theory, such as the P/NP problem. Many of these turn out to be mistaken, and such submissions tax JACM volunteer editors and reviewers. JACM remains open to the possibility of eventual resolution of P/NP and related questions and continues to welcome submissions on the subject. However, to mitigate the burden of repeated resubmissions due to incremental corrections of errors identified during editorial review, no author may submit more than one such paper to JACM, ACM Trans. on Algorithms, or ACM Trans. on Computation Theory in any 24-month period, except by invitation of the Editor-in-Chief. This applies to resubmissions of previously rejected manuscripts. Please consider this policy before submitting a such a paper.

Strong Church-Turing Thesis

Every physically realizable computation model can be simulated by a TM with polynomial overhead (i.e., t steps on the model can be simulated in t^c steps on the TM, where c is a constant that depends upon the model).

What are the possible objections to accept it?

- Precision - TM's compute with discrete symbols, whereas physical quantities may be real numbers in R. TM computations may only be able to approximately simulate the real world.

Strong Church-Turing Thesis

Every physically realizable computation model can be simulated by a TM with polynomial overhead (i.e., t steps on the model can be simulated in t^c steps on the TM, where c is a constant that depends upon the model).

What are the possible objections to accept it?

- Randomness: The TM as defined is deterministic. If randomness exists in the world, one can conceive of computational models that use a source of random bits (i.e., ”coin tosses”).

Strong Church-Turing Thesis

Every physically realizable computation model can be simulated by a TM with polynomial overhead (i.e., t steps on the model can be simulated in t^c steps on the TM, where c is a constant that depends upon the model).

What are the possible objections to accept it?

- Quantum mechanics: A computational model might use some of the counterintuitive features of quantum mechanics. However, it is not yet clear whether a scalable quantum system is truly physically realizable. Also, quantum computers currently seem only able to efficiently solve only very few "well-structured" problems that are (presumed to be) not in P.

Strong Church-Turing Thesis

Every physically realizable computation model can be simulated by a TM with polynomial overhead (i.e., t steps on the model can be simulated in t^c steps on the TM, where c is a constant that depends upon the model).

What are the possible objections to accept it?

- Other exotic physics, such as string theory. Though an intriguing possibility, it hasn't yet had the same scrutiny as use of quantum mechanics.
- Energy considerations: how much energy is consumed during the computation?

The Set *NP* and Polynomial Verifiability

- Definition: If $L \subseteq \Sigma^*$, we say that a TM T is a *verifier* for L if:
  - T accepts a language $L_1 \subseteq \Sigma^*\$\Sigma^*$
  - T halts on every input, and
  - $L = \{x \in \Sigma^* \mid \text{for some } a \in \Sigma^*, x\$a \in L_1\}$ (we will call such a value a a *certificate* for x)

some special symbol

- A verifier T is a *polynomial-time verifier* if:
  - There is a polynomial p such that for every x and every a in Σ^*, the number of moves T makes on the input string $x\$a$ is no more than $p(|x|)$

The Set *NP* and Polynomial Verifiability

- Theorem: For every language $L \in \Sigma^*$

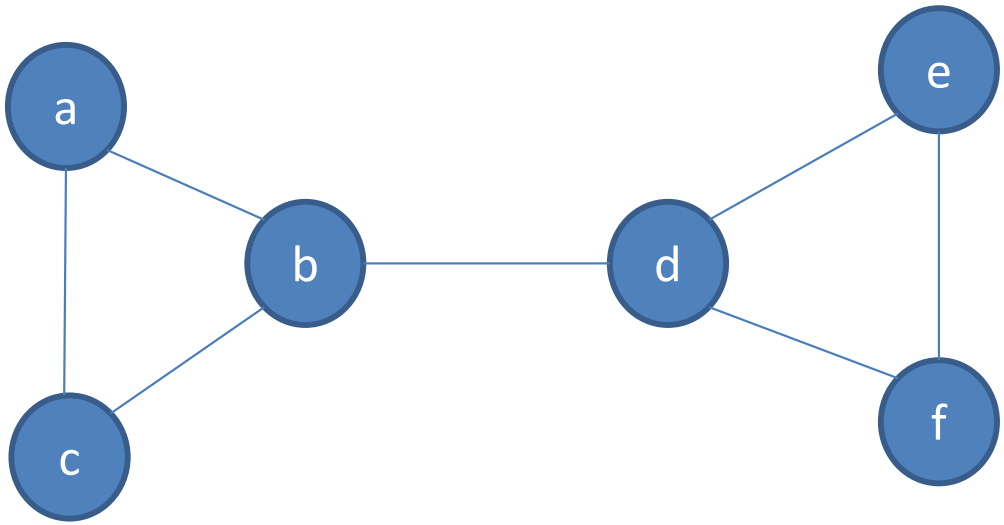
  $L \in NP$ **if and only if** L is polynomially verifiable

  i.e., there is a polynomial-time deterministic TM that is a verifier for L
- Proof: See book

For example:

- A verifier for SAT could take a specific truth assignment as a certificate;
- A verifier for TSP could take a permutation of the cities as a certificate

- A graph G=(V,E) is a pair, where V is a set of nodes, and E is a set of edges. Each edge connects a pair of nodes (*i*,*j*), where *i*, and *j* are in *V*. In **undirected** graphs the order of *i*, and *j* is not important. (However, it is important in **directed** graphs.)
- A **path** is a sequence of edges leading from one node to another. A **Hamiltonian path** is a path that visits every node exactly once.
- Example: there are paths for all pairs of nodes, but there is no Hamiltonian path from *a* to *c*.



- **PATH={<*G,a,b*> | *G* is a graph with path from *a* to *b*}**

This language is in *P.* You can check if there is a *a*-*b* path by running BFS from *a*.

In general, the complexity of running BFS is linear in *n+m*, the number of nodes+edges, i.e., it is *O(n+m)*. *However, the actual running time depends on the representation of G.*

- **HAMPATH={<*G,a,b*> | *G* is a graph with Hamiltonian path from *a* to *b*}**

This language is in *NP.* Trivial algorithm for finding HP has to check an exponential number of possibilities but there is a fast nondeterministic algorithm for it because we can guess the path (this will be a certificate for verifier).

- PRIME={all prime numbers in binary format}
- COMPOSITE={all composite numbers in binary format}

If *m* is an input for an algorithm that decides PRIME, its length is $\log_2 m$, i.e., *we want an algorithm that runs in polynomial time in a number of bits!* Not in time that is proportional to *m.*

It is easy to define a certificate for COMPOSITE (two factors).

Number theory gives a certificate for PRIME.

For many years it has been assumed that PRIME is in *NP.*

In 2002 it was shown that PRIME is in *P*, and so is the complementary problem COMPOSITE but *there is still no polynomial-time algorithm known for the factorization.*

Space Complexity

- The space used by a TM corresponds to the memory used by a computer
- When we compute the space used by a TM, we don't count the input space
- **Definition**: TM runs in space *S(n)* is for all inputs of length *n*. M uses at most *S(n)* cells in total on its work tapes.
- Example: SAT can be decided in …

Space Complexity

- The space used by a TM corresponds to the memory used by a computer
- When we compute the space used by a TM, we don't count the input space
- **Definition**: TM runs in space *S(n)* is for all inputs of length *n*. M uses at most *S(n)* cells in total on its work tapes.
- Example: SAT can be decided in linear space. We need a space for 1 assignment and some more space to keep track.
- Can space complexity exceed time complexity? No

- **Theorem**: If a TM runs in time *T(n)* then it runs in space at most *T(n)*

Space Complexity

- **Theorem**: If a TM runs in time $T(n)$ then it runs in space at most $T(n)$
- **Theorem**: Suppose a deterministic TM runs in $S(n)$ space with $|\Gamma|=g$ letters in the tape alphabet and $|Q|=q$. If this TM runs for longer than $qng^{S(n)}$ steps on an input of length n, then it stuck in an infinite loop.
- **Consequence**: If L is accepted by TM T running in space $S(n)$, where $S(n) > \log n$, then L is accepted by a TM T' that runs in space $O(S(n))$ but always halts.
- The nondeterministic space is defined similarly for NTMs.

Polynomial and Logarithmic Spaces

PSPACE = all languages that can be decided in polynomial space by deterministic TM.

NPSPACE = all languages that can be decided in polynomial space by nondeterministic TM.

After all the uncertainty about P, and NP it is surprising that

Theorem: PSPACE = NPSPACE (not hard to prove)

L = all problems solvable in O(log n) space.

NL = all problems solvable in O(log n) space by NTM.

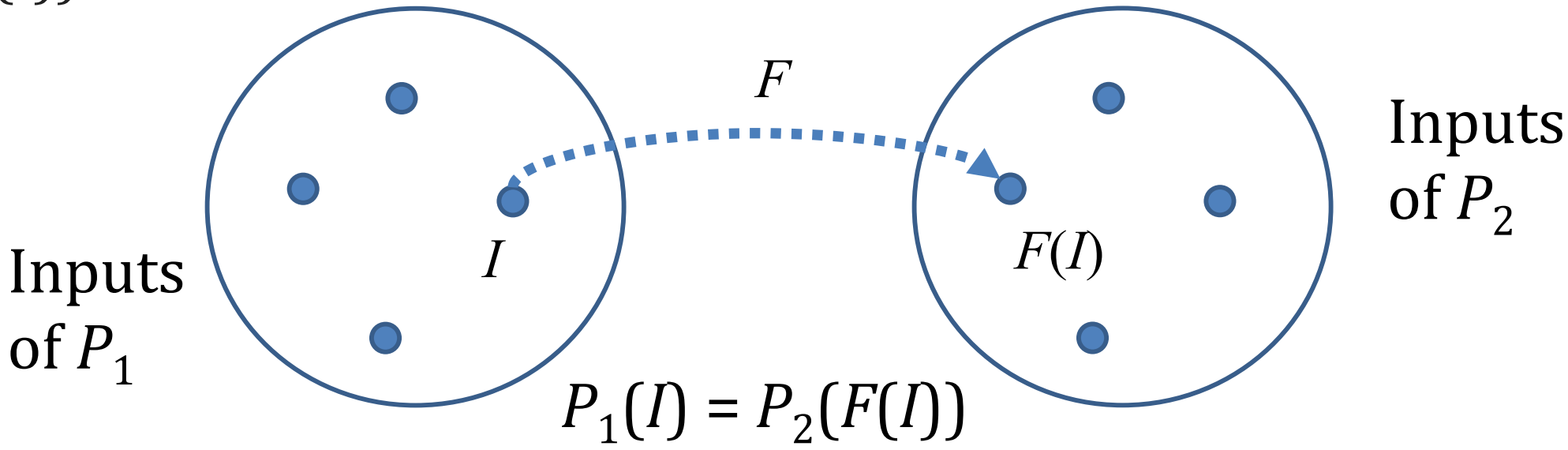
This is what we know about these six classes:

$L \subseteq NL \subseteq P \subseteq NP \subseteq$ PSPACE = NPSPACE and $L \neq$ PSPACE

Reduction

Definition: Suppose P_1 and P_2 are decision problems.

We say P_1 is reducible to P_2 $(P_1 \leq P_2)$ if there is an algorithm that finds, for an arbitrary instance I of P_1, an instance $F(I)$ of P_2 such that the **two answers are the same**, i.e., (the answer to P_1 for the instance I, and the answer to P_2 for the instance $F(I)$)

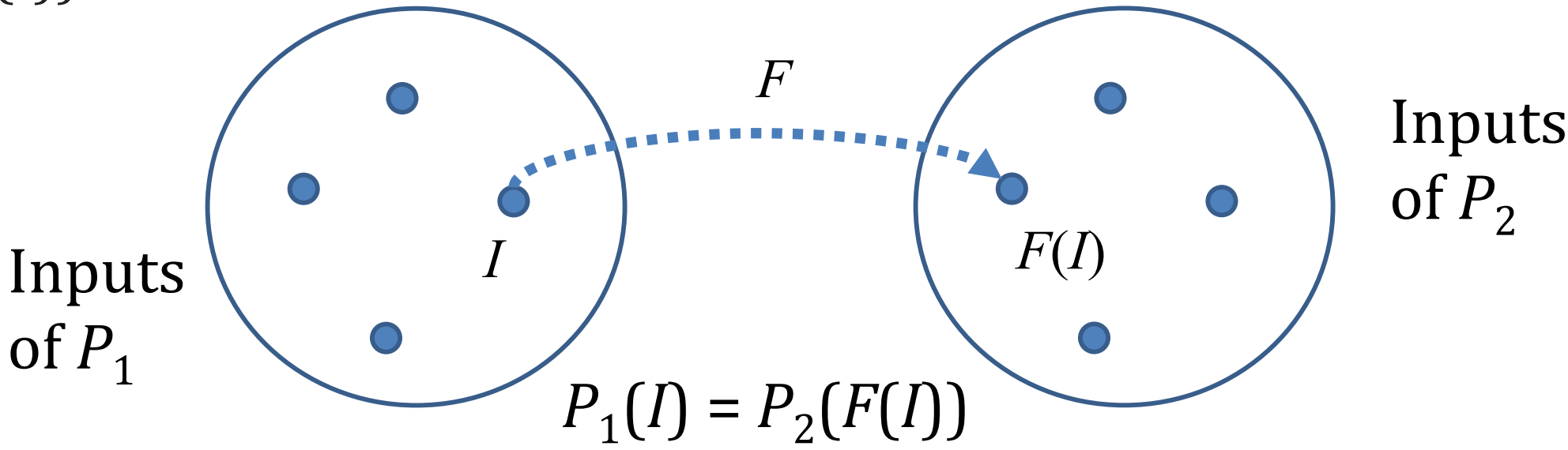


Idea 1: For example, I don't know how to solve P_1 but I can solve P_2 and know how to map the instances to preserve answers. Then I can solve P_1

Reduction

Definition: Suppose P_1 and P_2 are decision problems.

We say P_1 is reducible to P_2 $(P_1 \leq P_2)$ if there is an algorithm that finds, for an arbitrary instance I of P_1, an instance $F(I)$ of P_2 such that the **two answers are the same**, i.e., (the answer to P_1 for the instance I, and the answer to P_2 for the instance $F(I)$)



Idea 2: Say, P_1 is computationally difficult, and I don't know the difficulty of P_2. If I know how to map the instances, then I can state that P_2 is at least as difficult as P_1

Polynomial-Time Reductions and *NP*-Completeness

- Just as we can show that a problem is decidable by reducing it to another one that is also decidable, we can show that a language is in *P* by reducing it to another that is.
- To see whether $x \in L_1$, all we have to do is compute $f(x)$ and see whether it is in L_2; and f is computable.
  - In the case of decidability (i.e., we need to decide y/n, not to compute something), we only needed the reduction to be computable
  - *Here we need the reduction function to be computable in polynomial time*

- **Definition**: If L_1 and L_2 are languages over respective alphabets Σ_1 and Σ_2, a *polynomial-time reduction* from L_1 to L_2 is a function $f : \Sigma_1^* \rightarrow \Sigma_2^*$ satisfying two conditions

  1) for every $x \in \Sigma_1^*$, $\quad x \in \boldsymbol{L_1}$ **if and only if** $f(x) \in L_2$

  2) f can be computed in **polynomial** time, i.e., there is a TM with polynomial time complexity that computes f

- If there is a polynomial-time reduction from L_1 to L_2, we write $L_1 \leq_p L_2$ and say that L_1 is polynomial-time reducible to L_2.
- In this case, we said that deciding L_1 is no harder than deciding L_2, because we considered only two degrees of hardness, decidable and undecidable.

Polynomial -Time Reductions and *NP*-Completeness

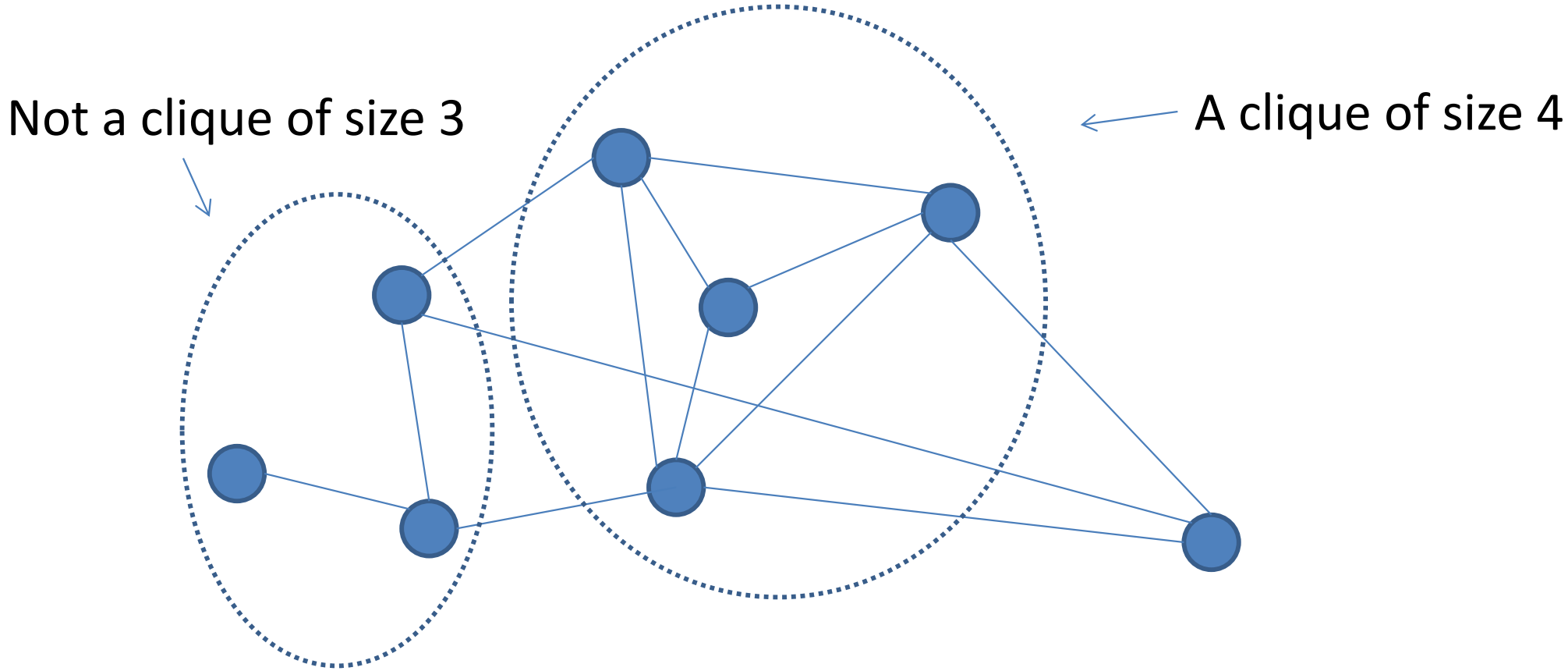
- Theorem:
  - Polynomial-time reducibility is transitive:
    - If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ then $L_1 \leq_p L_3$
  - If $L_1 \leq_p L_2$ and $L_2 \in P$, then $L_1 \in P$
- Proof sketch:
  - For the first statement, simply use the composition of the reduction functions
  - For the second statement, simply combine the TM that accepts L_2 and the one that computes the reduction f

Example: Reduction from 3-SAT to Clique (3-SAT $\leq_p$ Clique)

The Clique decision problem:

Input: Graph $G = (V, E)$ and an integer k

Output: "yes" if there is a clique of size k in G, "no" otherwise.



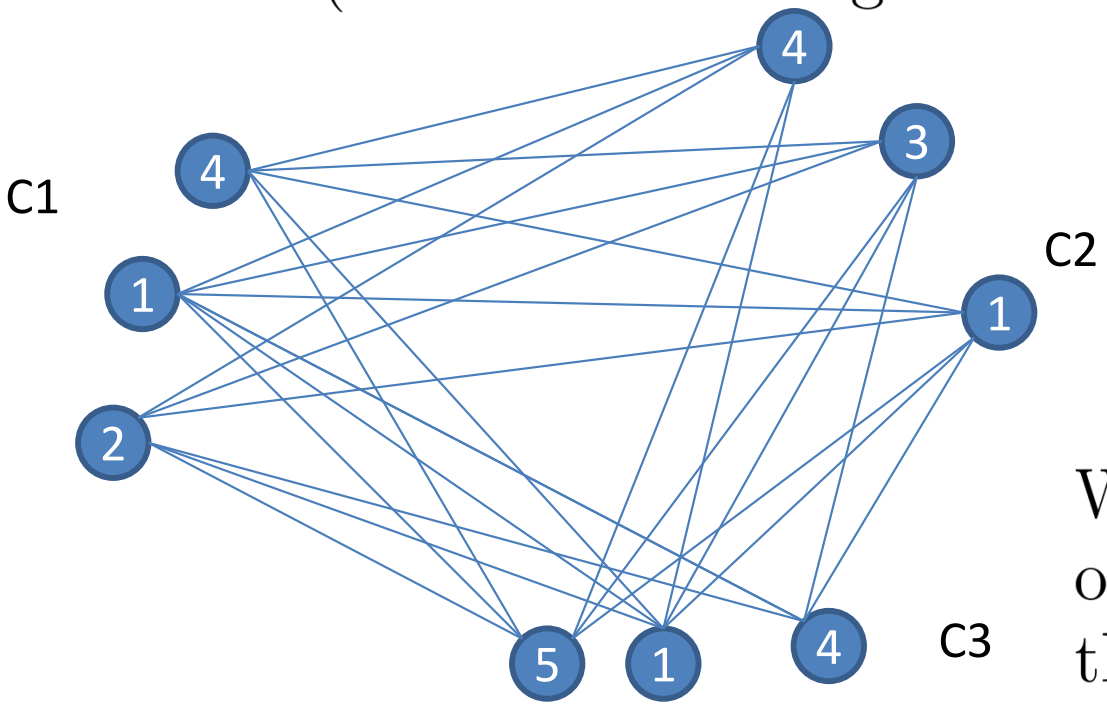
clique of size k = complete subgraph of size k

Reduction

Given a 3-SAT formula ϕ we want to create $G = (V, E)$.

$$\phi = \overset{\text{C1}}{(x_1 \vee x_4 \vee \neg x_2)} \wedge \overset{\text{C2}}{(x_4 \vee x_1 \vee x_3)} \wedge \overset{\text{C3}}{(x_1 \vee \neg x_4 \vee x_5)}$$

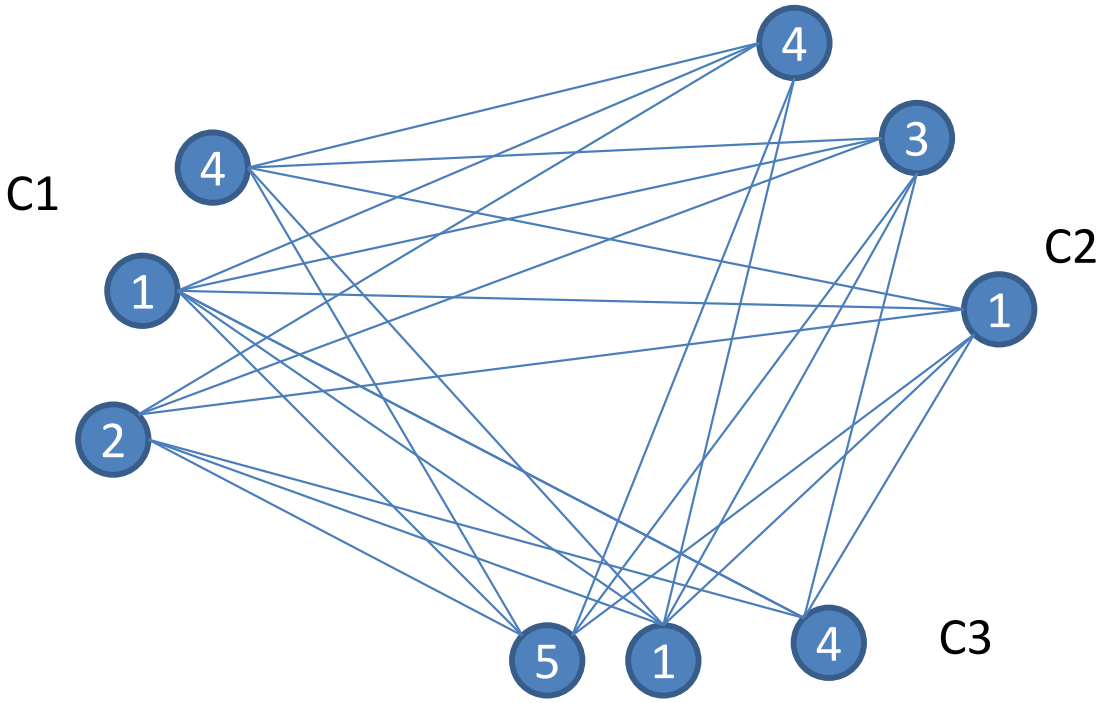
For every clause in ϕ, we create 3 vertices corresponding to the literals (variable or its negation) in that clause. Add an edge between vertices from different clauses if and only if those two literals are consistent (one is not the negation of the other).



We will show that G has a clique of size m iff ϕ is satisfiable, so the reduction maps ϕ to (G, m).

C1 C2 C3

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

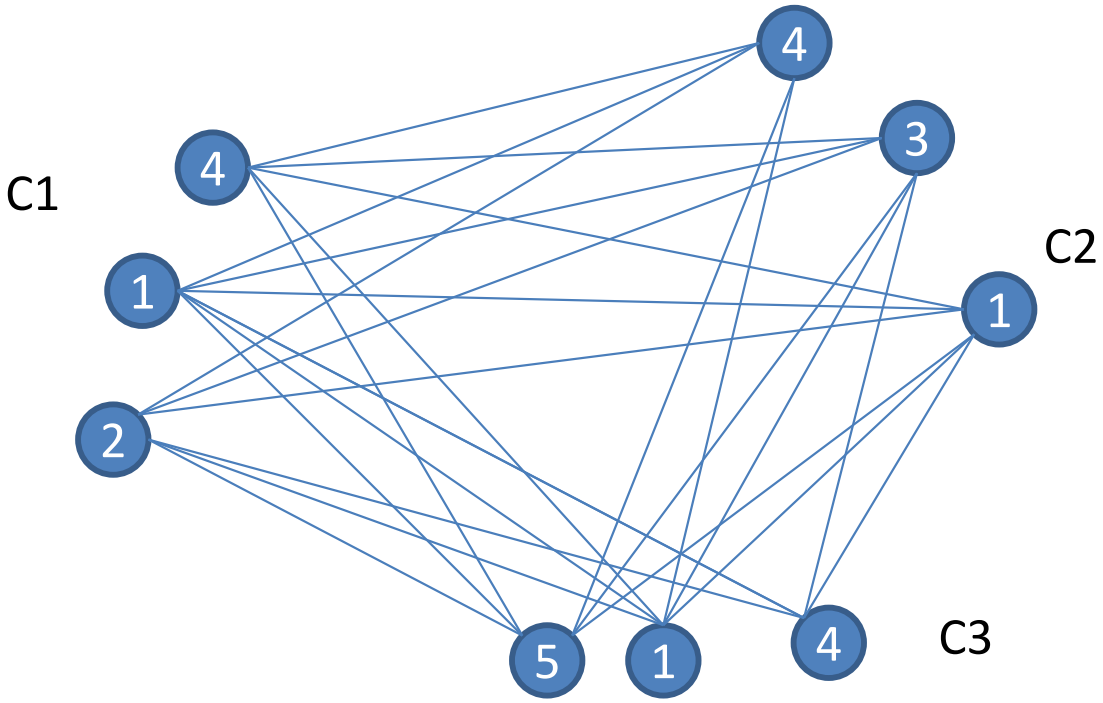


1. We show "yes" for 3-SAT $\Rightarrow$ "yes" for clique.

If the 3-SAT instance is a "yes" then there is a truth assignment (T/F values assigned to the variables) such that every clause has a true literal. By selecting a corresonding vertex to a true literal in each clause, then we see that we have a clique in G of size m (where m is the number of clauses).

C1 C2 C3

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



2. "yes" for clique $\Rightarrow$ "yes" for 3-SAT.

Assume there is a clique of size m, where m is the number of clauses. The truth assignment induced by the labels of the vertices satisfies ϕ because (1) every pair of vertices in the clique has an edge, i.e., no both T and F for the same variable, (2) every trio of vertices corresponding to a clause has no edges between those vertices, and we must have a vertex from every clause.

Example: Reduction from 3-SAT to Clique (3-SAT $\leq_p$ Clique)

The decision problem clique:

Input: Graph $G = (V, E)$ and an integer k

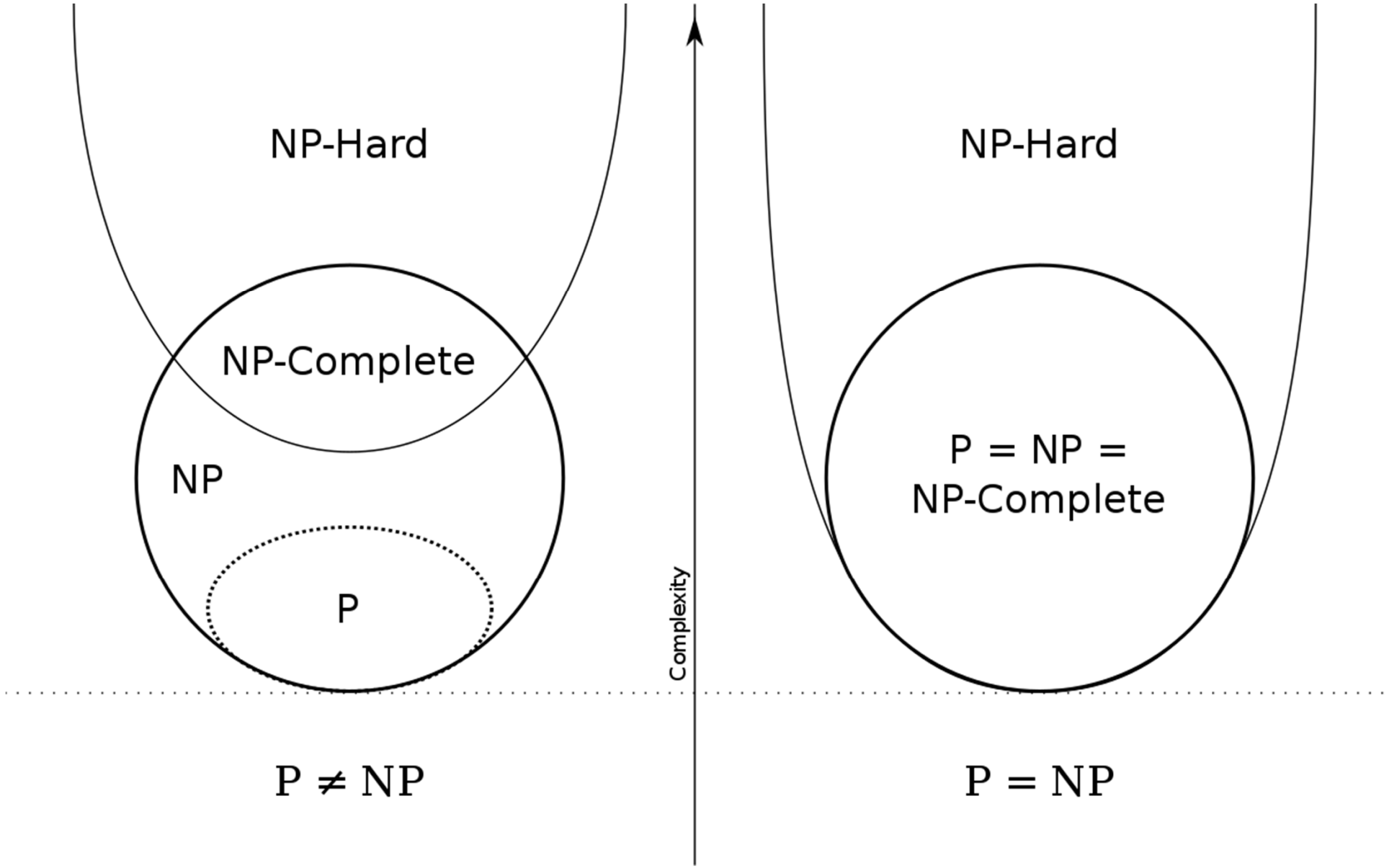
Output: "yes" if there is a clique of size k in G, "no" otherwise.

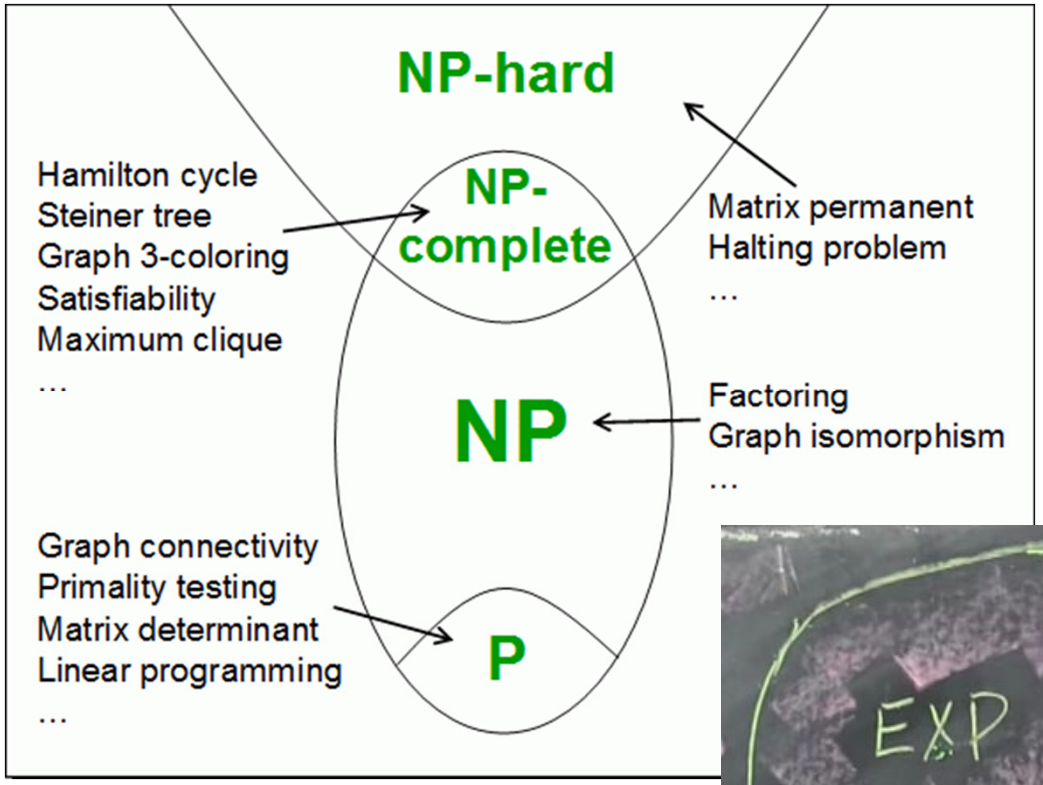
We have a poly-time reduction from 3-SAT to Clique because

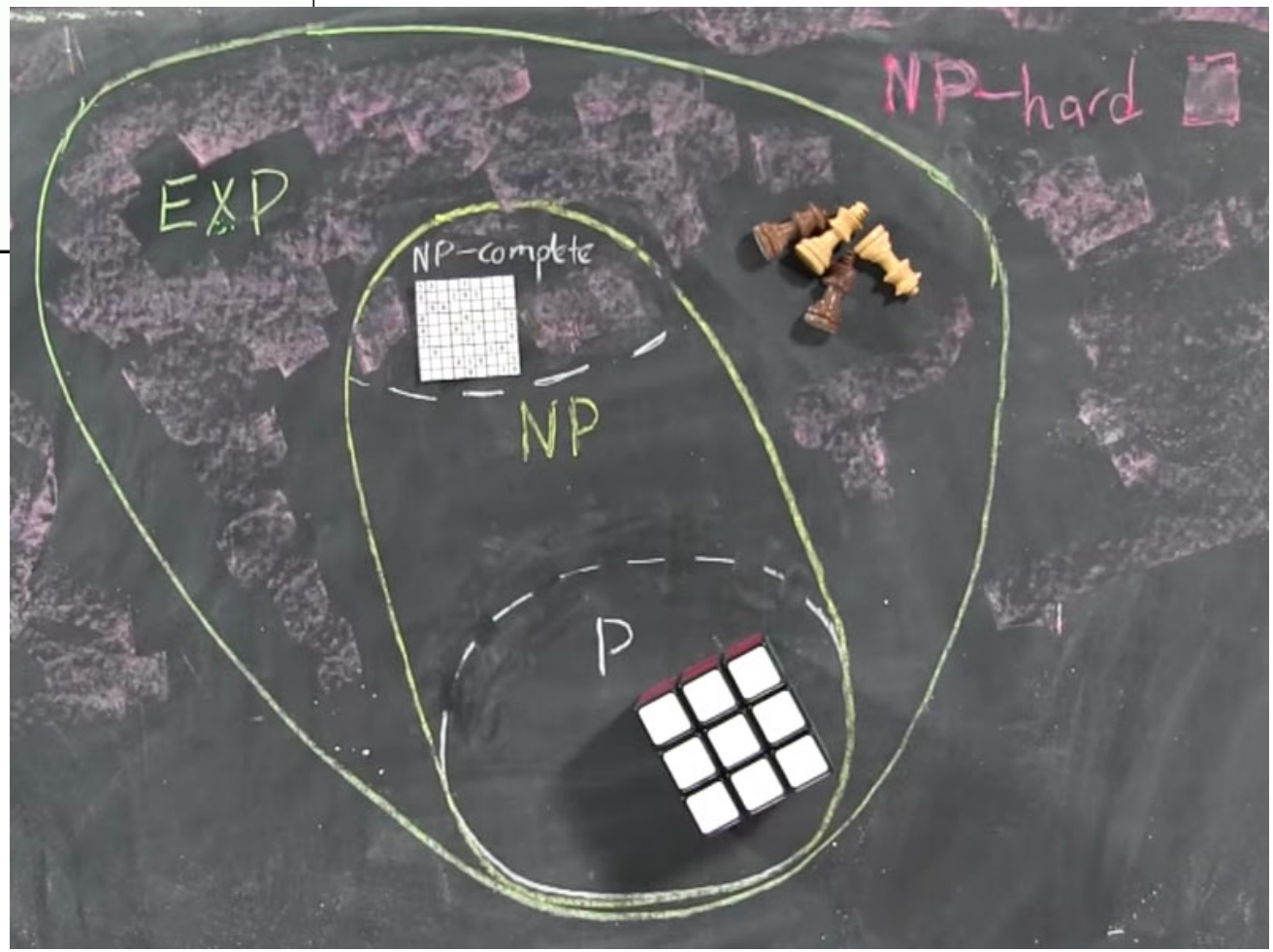
- "yes" for 3-SAT $\Rightarrow$ "yes" for Clique
- "yes" for 3-SAT $\Leftarrow$ "yes" for Clique
- creating G from ϕ takes polynomial time

Polynomial-Time Reductions and *NP*-Completeness

- **Definition**: A language L is *NP-hard*
  if $L_1 \leq_p L$ for every $L_1 \in NP$
- **Definition**: A language L is *NP-complete*
  if $L \in NP$ and L is *NP*-hard
- Theorem:
  - If L and L_1 are languages such that L is *NP*-hard and $L \leq_p L_1$, then L_1 is also *NP*-hard
  - If L is any *NP*-complete language, then $L \in P$ if and only if $P = NP$
- There are many *NP-complete* problems. The standard method to prove NP-completeness is to take a problem that is known to be NP-complete and reduce it to your problem.


NP-Hard
NP-Complete
NP
P
Complexity
NP-Hard
P = NP =
NP-Complete
P ≠ NP
P = NP


NP-hard
Hamilton cycle
Steiner tree
Graph 3-coloring
Satisfiability
Maximum clique
…
NP-complete
Matrix permanent
Halting problem
…
NP
Factoring
Graph isomorphism
…
Graph connectivity
Primality testing
Matrix determinant
Linear programming
…
P


NP-hard
EXP
NP-complete
NP
P

The Cook-Levin Theorem

- Theorem:
  - The language *Satisfiable* (or the corresponding decision problem *SAT*) is *NP*-complete
- Proof:
  - We know that *Satisfiable* is in *NP*, so we need to show that every language $L \in NP$ is reducible to *Sat*
  - We do this by using a TM *T* that accepts *L*; the reduction considers the details of *T* and takes a string *x* to a Boolean formula that is satisfiable if and only if *x* is accepted by *T*
  - The details are complex and can be found in the book

Some Other *NP*-Complete Problems

- Theorem:
  - The clique problem (Given a graph G and an integer k, does G have a complete subgraph with k vertices?) is *NP*-complete.
- Proof sketch:
  - By reduction from *SAT* or *3-SAT*

    + show that the language is in *NP*.

Reduction is a tool for demonstrating the hardness

- We want to demonstrate that a problem X is NP-complete (i.e., it is currently computationally difficult because we don't know if there is a polynomial algorithm for it)

- We find a problem Y that is NP-complete and check:

Can we solve Y using X (having in mind that hopefully X is at least as hard as Y) ?

i.e., we reduce X to Y. In other words, given a black box that solves Y, can we solve Y?

- To answer this question, we need to find a bijection
  f : Inputs(X) → Inputs(Y) such that X(i) = Y(f(i)), i.e. solutions are equal

- The bijection f should be of polynomial time complexity

Reduction is a tool for demonstrating the hardness

- If X can be reduced to Y it is denoted by $X \leq_p Y$

- It means that X is at least as hard as Y because if we can solve X, we can solve Y, i.e., we need to reduce to the problem we want to show is the harder problem.

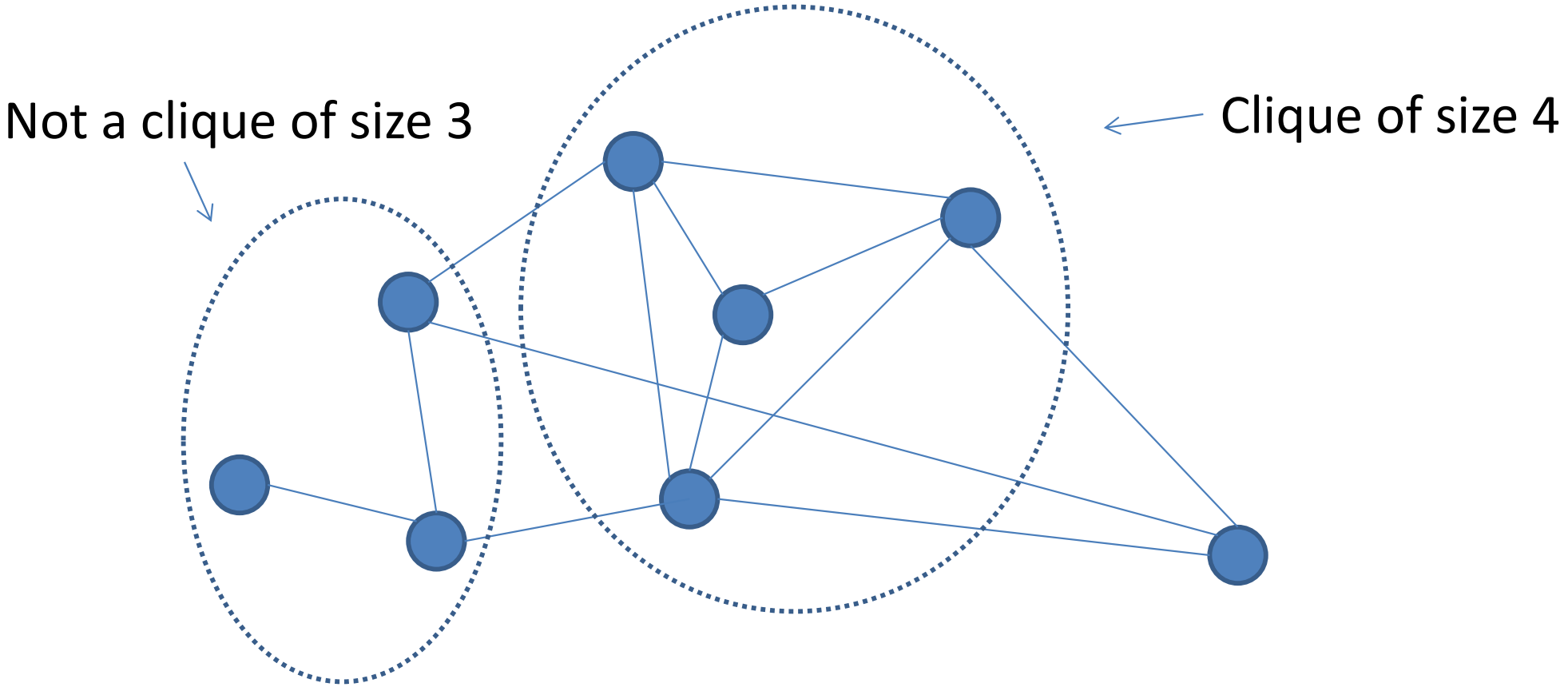
  y ← instance of X
  x ← *f*(y) // *f* is a bijection
  d ← solve Y with input x // decision problem d=0/1
  d is a solution of X with input y

- If we could find hard problem Y, we could prove that another problem X is hard by reducing Y to X.

Reduction is a tool for demonstrating the hardness

- We can prove that SAT is NP-complete
- We can construct a reduction from SAT to 3-SAT and show that 3-SAT is in NP, i.e., 3-SAT is also NP-complete
- We want to prove that Clique problem is NP-complete

The Clique decision problem:
Input: Graph $G = (V, E)$ and an integer k
Output: "yes" if there is a clique of size k in G, "no" otherwise.



clique of size k = complete subgraph of size k

Reduction is a tool for demonstrating the hardness

1) Clique is in NP. This is because for a given size of the set |S| we can enumerate all $\binom{n}{|S|}$ solutions, i.e., non-deterministic TM can guess the solution and we can verify it in polynomial time.

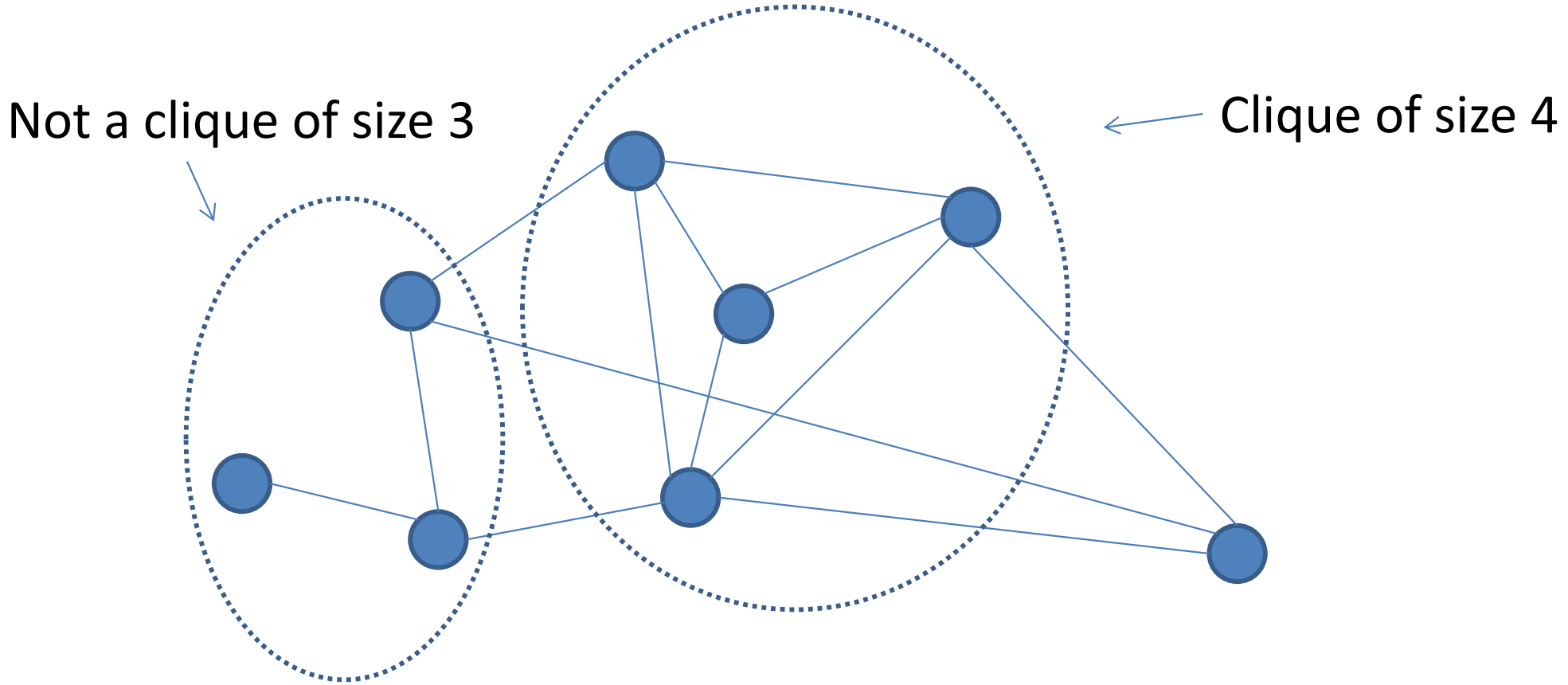
2) Prove that there is a polynomial reduction from 3-SAT to Clique

Example: Reduction from 3-SAT to Clique (3-SAT $\leq_p$ Clique)

The Clique decision problem:

Input: Graph $G = (V, E)$ and an integer k

Output: "yes" if there is a clique of size k in G, "no" otherwise.



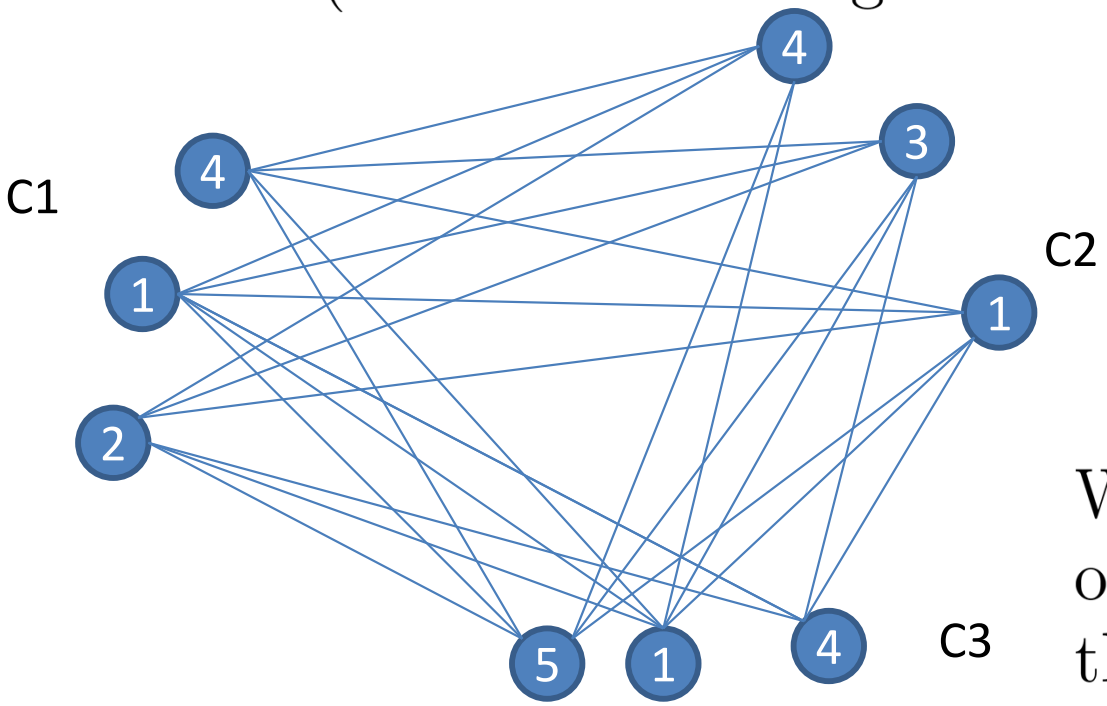
clique of size k = complete subgraph of size k

Reduction

Given a 3-SAT formula ϕ we want to create $G = (V, E)$.

$$\phi = \overset{\text{C1}}{(x_1 \vee x_4 \vee \neg x_2)} \wedge \overset{\text{C2}}{(x_4 \vee x_1 \vee x_3)} \wedge \overset{\text{C3}}{(x_1 \vee \neg x_4 \vee x_5)}$$

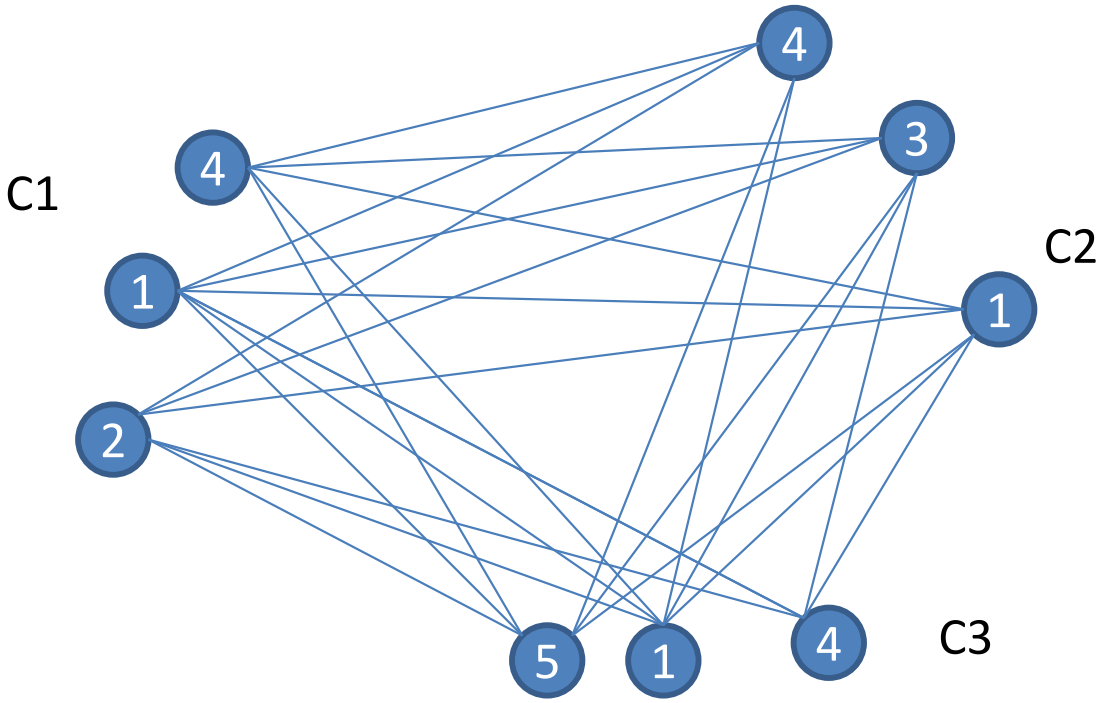
For every clause in ϕ, we create 3 vertices corresponding to the literals (variable or its negation) in that clause. Add an edge between vertices from different clauses if and only if those two literals are consistent (one is not the negation of the other).



We will show that G has a clique of size m iff ϕ is satisfiable, so the reduction maps ϕ to (G, m).

C1 C2 C3

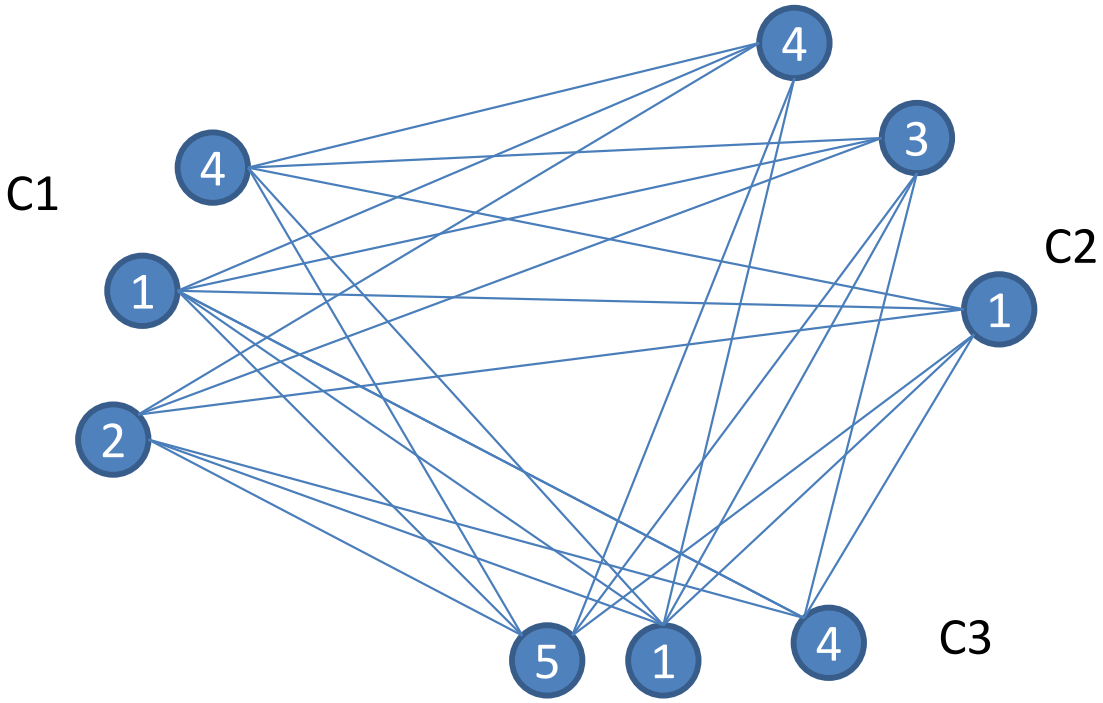
$$\phi = (x_1 \vee x_4 \vee \neg x_2) \wedge (x_4 \vee x_1 \vee x_3) \wedge (x_1 \vee \neg x_4 \vee x_5)$$



1. We show "yes" for 3-SAT $\Rightarrow$ "yes" for clique. If the 3-SAT instance is a "yes" then there is a truth assignment (T/F values assigned to the variables) such that every clause has a true literal. By selecting a corresonding vertex to a true literal in each clause, then we see that we have a clique in G of size m (where m is the number of clauses).

C1 C2 C3

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

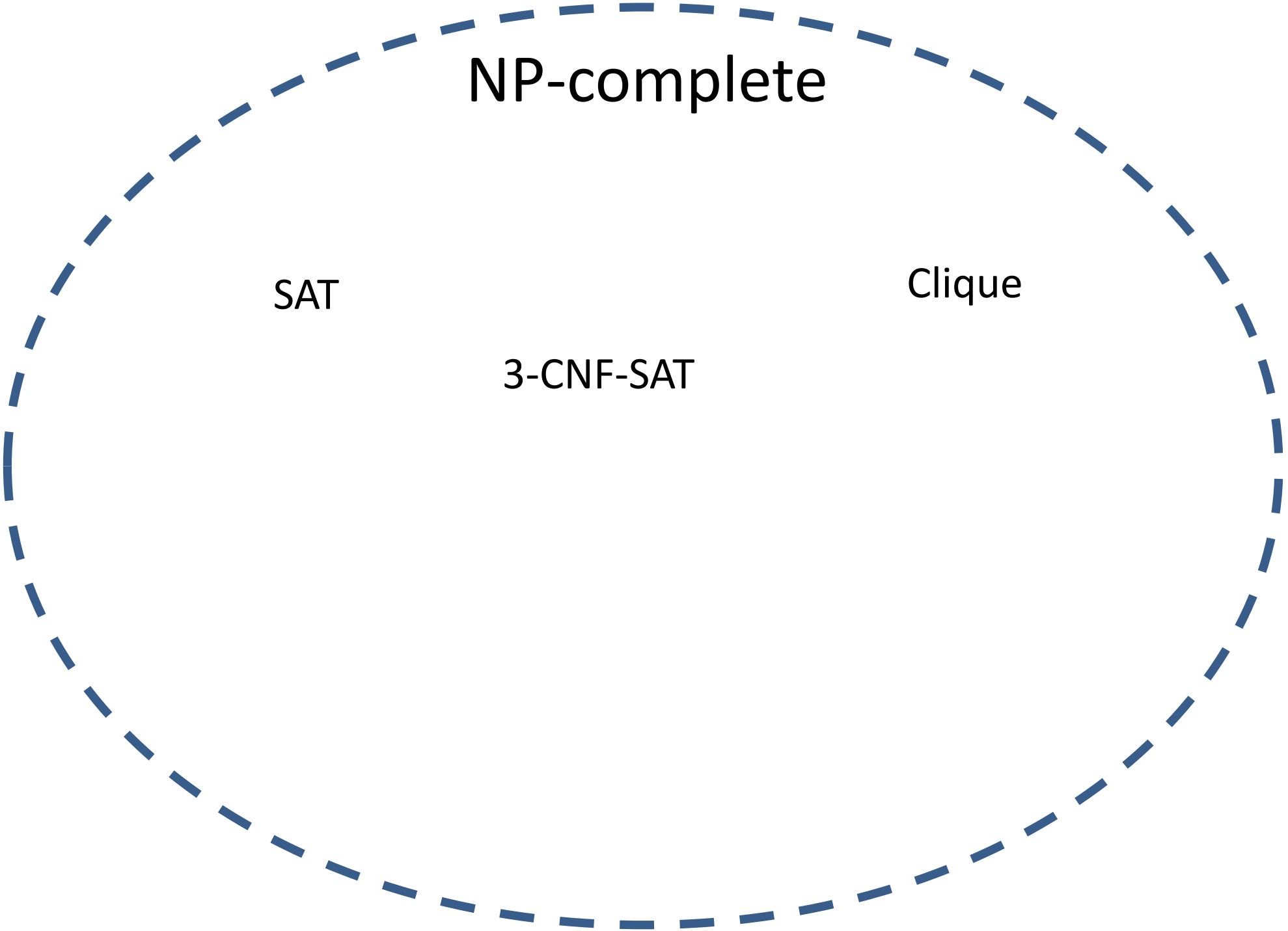


2. "yes" for clique $\Rightarrow$ "yes" for 3-SAT. Assume there is a clique of size m, where m is the number of clauses. The truth assignment induced by the labels of the vertices satisfies ϕ because (1) every pair of vertices in the clique has an edge, i.e., no both T and F for the same variable, (2) every trio of vertices corresponding to a clause has no edges between those vertices, and we must have a vertex from every clause.

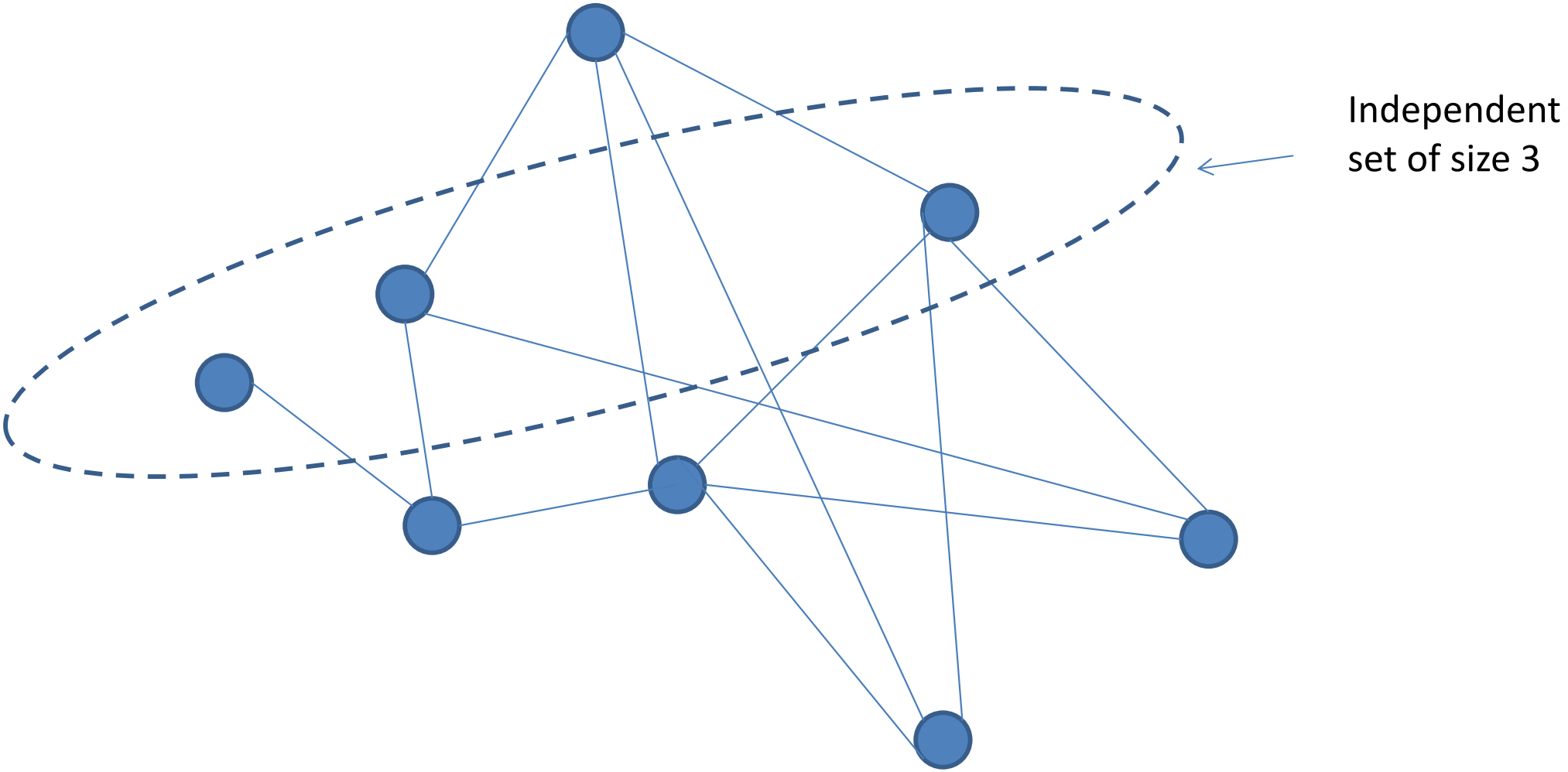
Example: Reduction from 3-SAT to Clique (3-SAT $\leq_p$ Clique)
The decision problem clique:
Input: Graph $G = (V, E)$ and an integer k
Output: "yes" if there is a clique of size k in G, "no" otherwise.

We have a poly-time reduction from 3-SAT to Clique because

- "yes" for 3-SAT $\Rightarrow$ "yes" for Clique
- "yes" for 3-SAT $\Leftarrow$ "yes" for Clique
- creating G from ϕ takes polynomial time


NP-complete
SAT
Clique
3-CNF-SAT

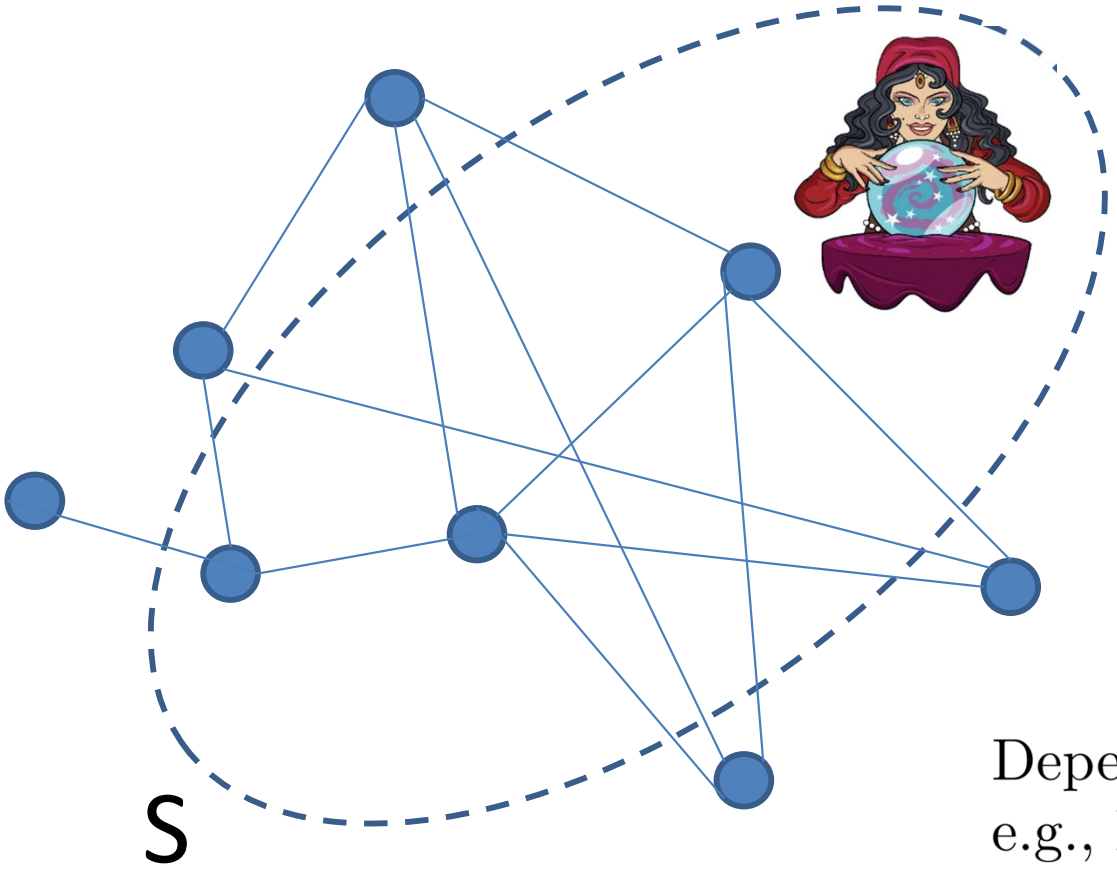
- Independent set (IS) decision problem: Given graph *G* and integer *k*, does *G* contain a subgraph *H* of *k* nodes such that *H* has no edges.



- Prove that IS is NP-complete

Part I of the NP-completeness proof: Independent set is in NP.

Given a certificate (i..e, a subset of nodes) can we verify if the corresponding subgraph contains edges in polynomial time?



How many edges we need to check?

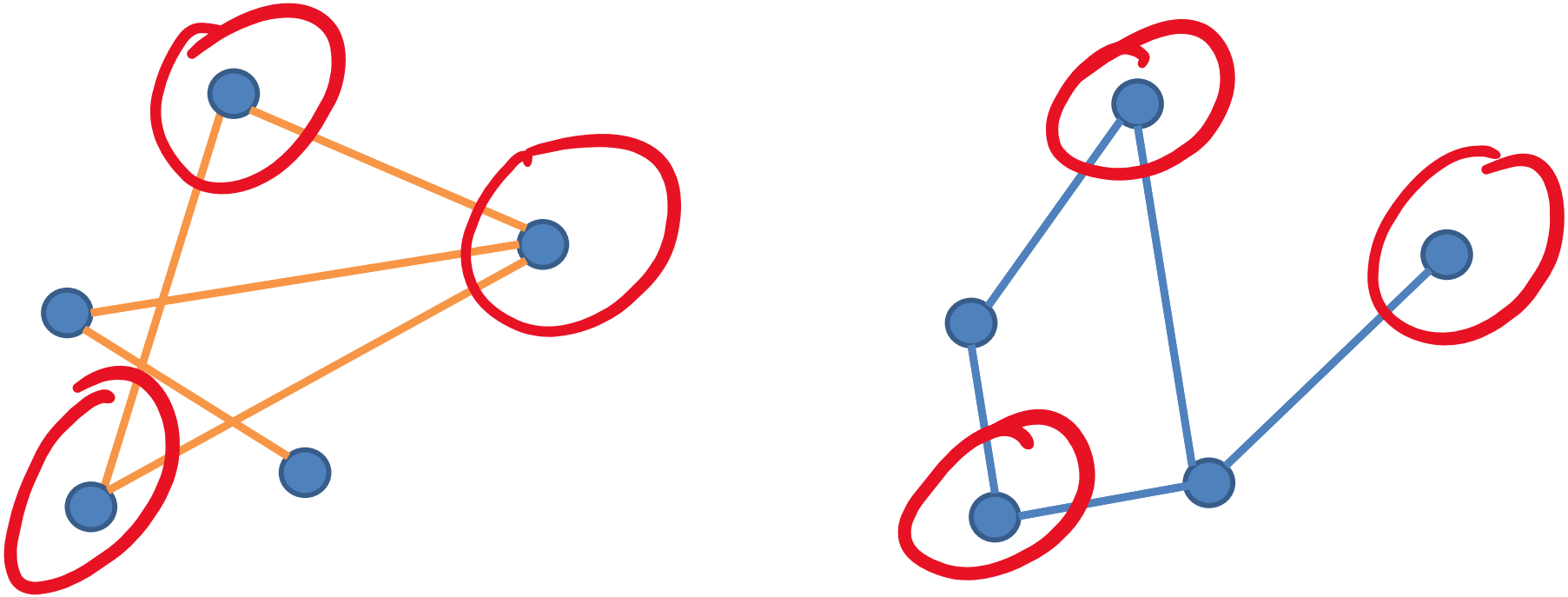
At most $\frac{|S|(|S|-1)}{2} \in O(n^2)$

Time to check one edge?

Depends on the data structure,
e.g., linear in the number of edges in G

Part II of the NP-completeness proof: Reduction from Clique to IS

- Reverse the graph: remove all existing edges, add non-edges
- A clique in *G* will correspond to the independent set in the reversed graph
- This process takes polynomial number of steps. The maximum possible number of edges is n(n-1)/2, where n is the number of nodes



A clique of size 3 is converted into independent set of size 3.

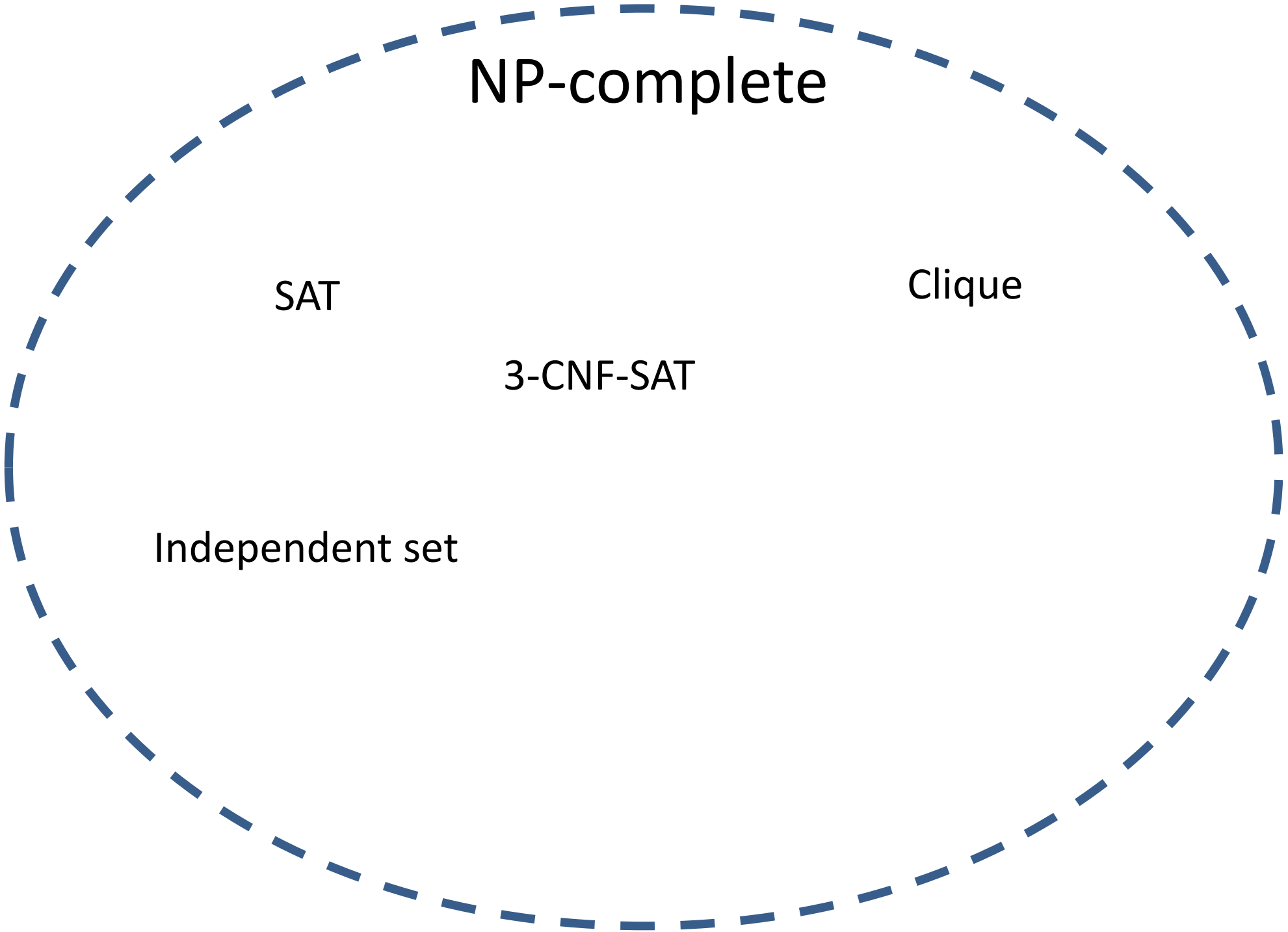
Applications of the maximum independent set

1. **Biology**: Identify clusters of proteins or genes that are not functionally related. This helps in understanding the structure of complex biological systems.

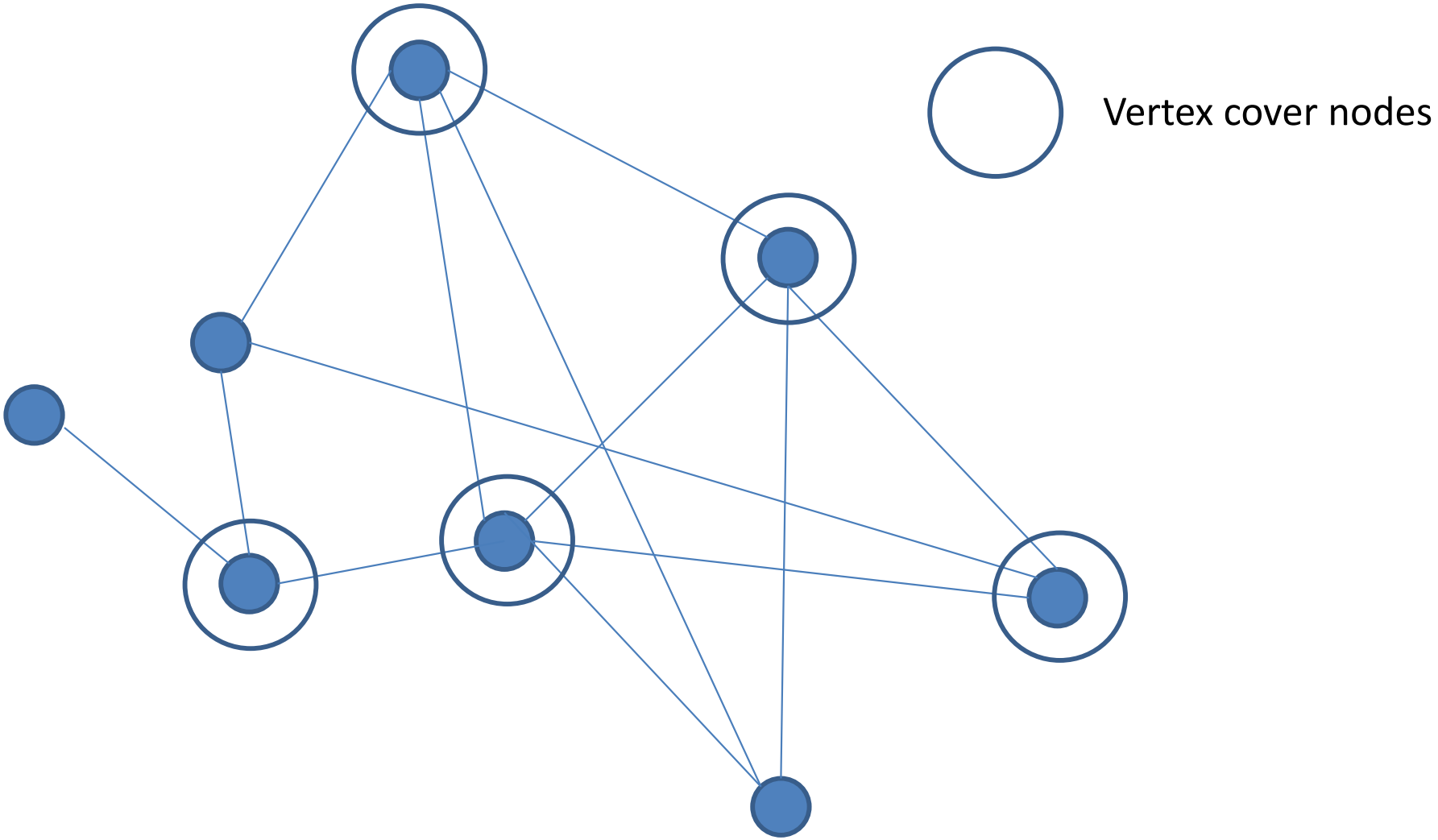
2. **Operations Research**: Optimize the placement of wireless communication devices or sensors in a network.

3. **Image Processing**: Segment images into different regions. The vertices in the graph represent pixels, and edges represent the similarity between pixels. Finding the MIS helps to identify regions of the image that are not related to each other.

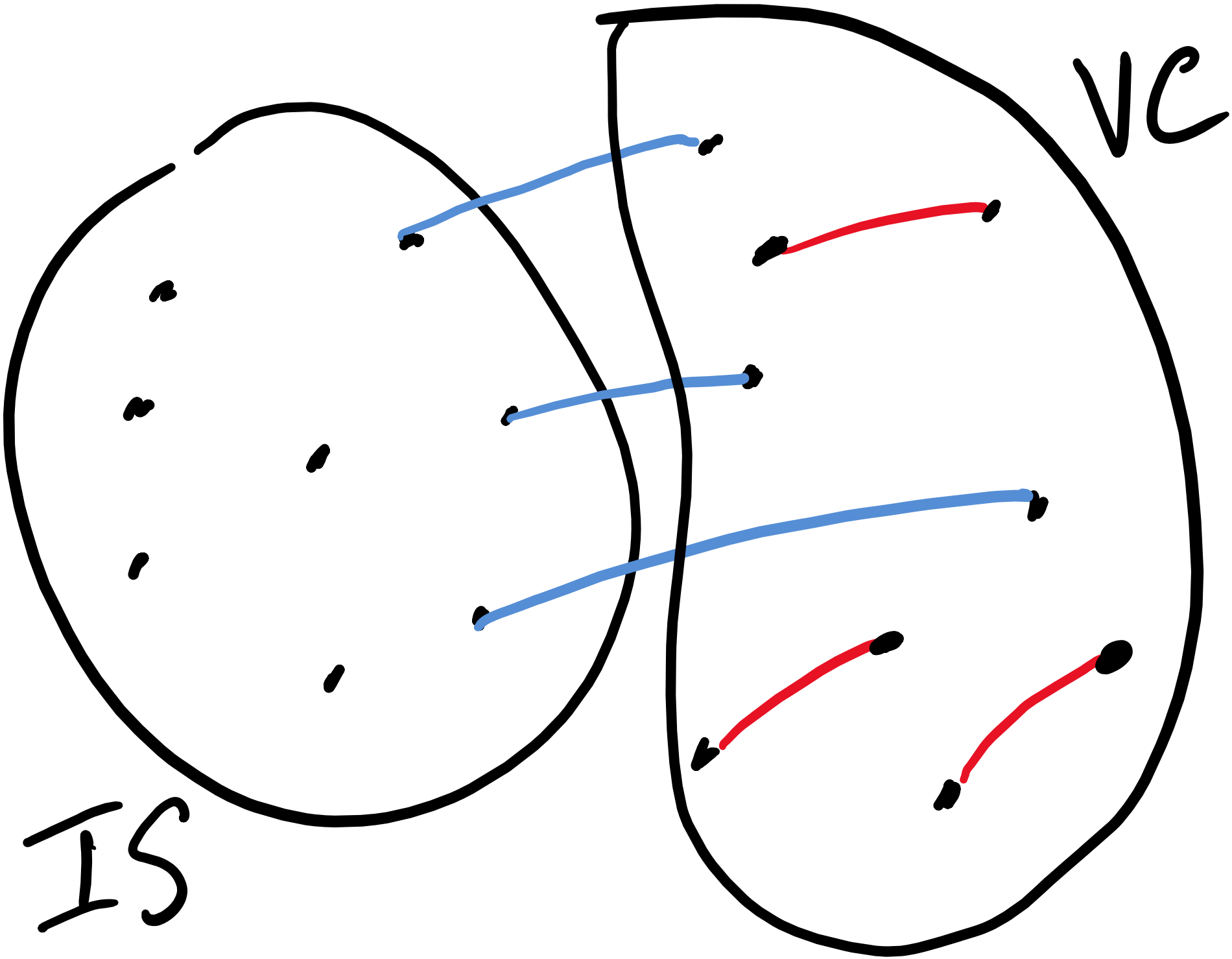
4. **Social Sciences**: Used in social network analysis to identify groups of people who are not connected to each other in a social network. This helps to understand the structure of social networks and the dynamics of social interactions.


NP-complete
SAT
Clique
3-CNF-SAT
Independent set

- A vertex cover of a graph is a set *S* of nodes such that every edge has at least one endpoint in *S*.



- Decision problem: Given a graph *G* and integer *k*, does *G* contain a vertex cover of size at most *k*. Is it NP-complete?


VC
IS

Theorem. *If $G = (V, E)$ is a graph, then S is an independent set iff $V - S$ is a vertex cover.*

Proof of $\Rightarrow$

If S is an independent set, and let $ij \in E$ is an edge then only one of i and j can be in S. Hence, at least one of i and j is in $V \setminus S$, i.e., it is a vertex cover.

Proof of $\Leftarrow$

Suppose $V - S$ is a vertex cover, and let i and j be in S. There cannot be an edge between i and j (otherwise that edge wouldn't be covered in $V - S$), so S is an independent set.

Theorem. *Independent Set* $\leq_P$ *Vertex Cover.*

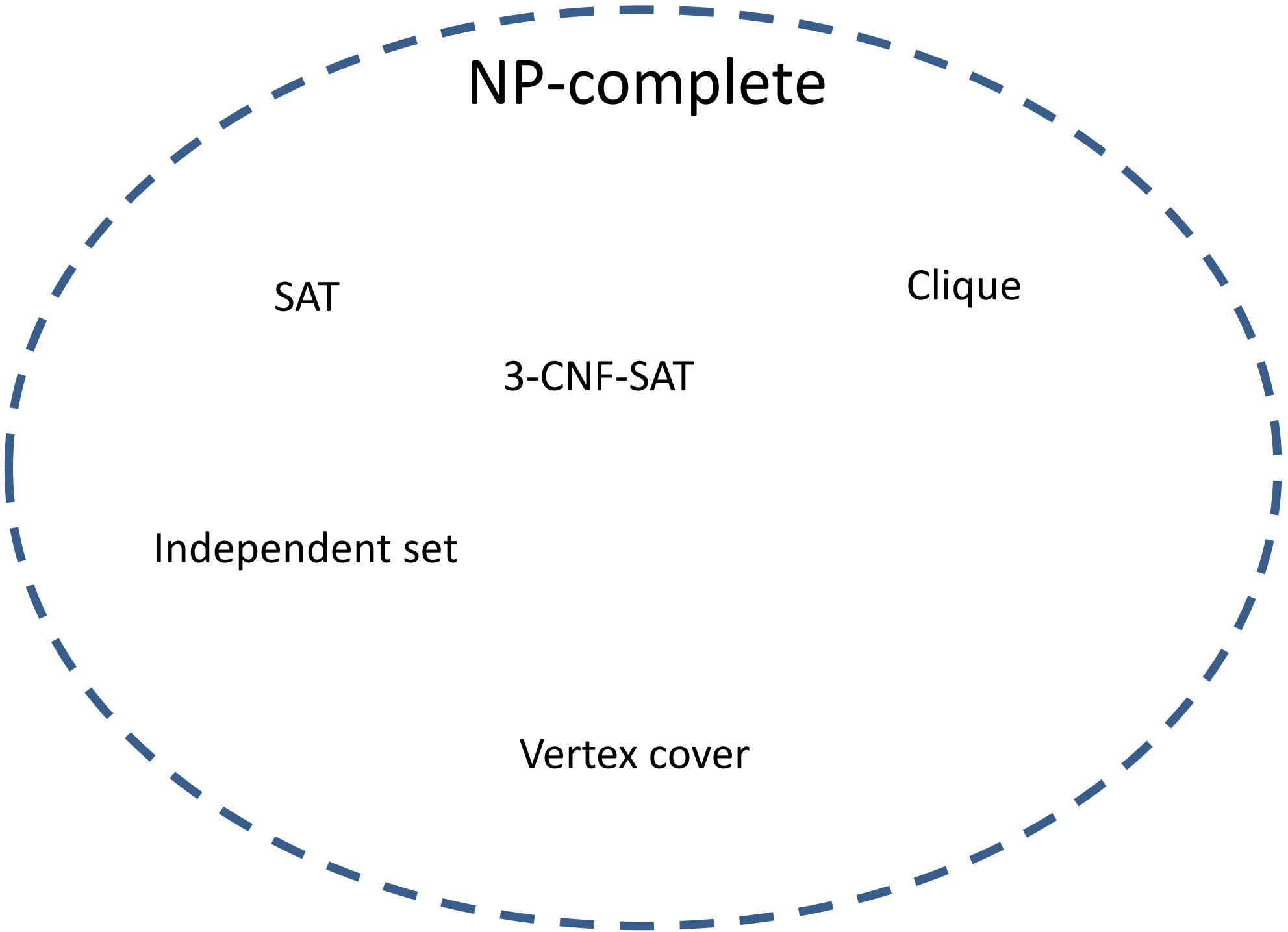
Proof. Given an instance of IS, namely, G and integer k, we convert it into the instance of vertex cover, i.e.,

- we ask the vertex cover black box solver if there is a vertex cover $(V - S)$ of size $\leq |V| - k$.

By previous theorem, S is an IS iff $V - S$ is a vertex cover, i.e., if the black box VC solver says

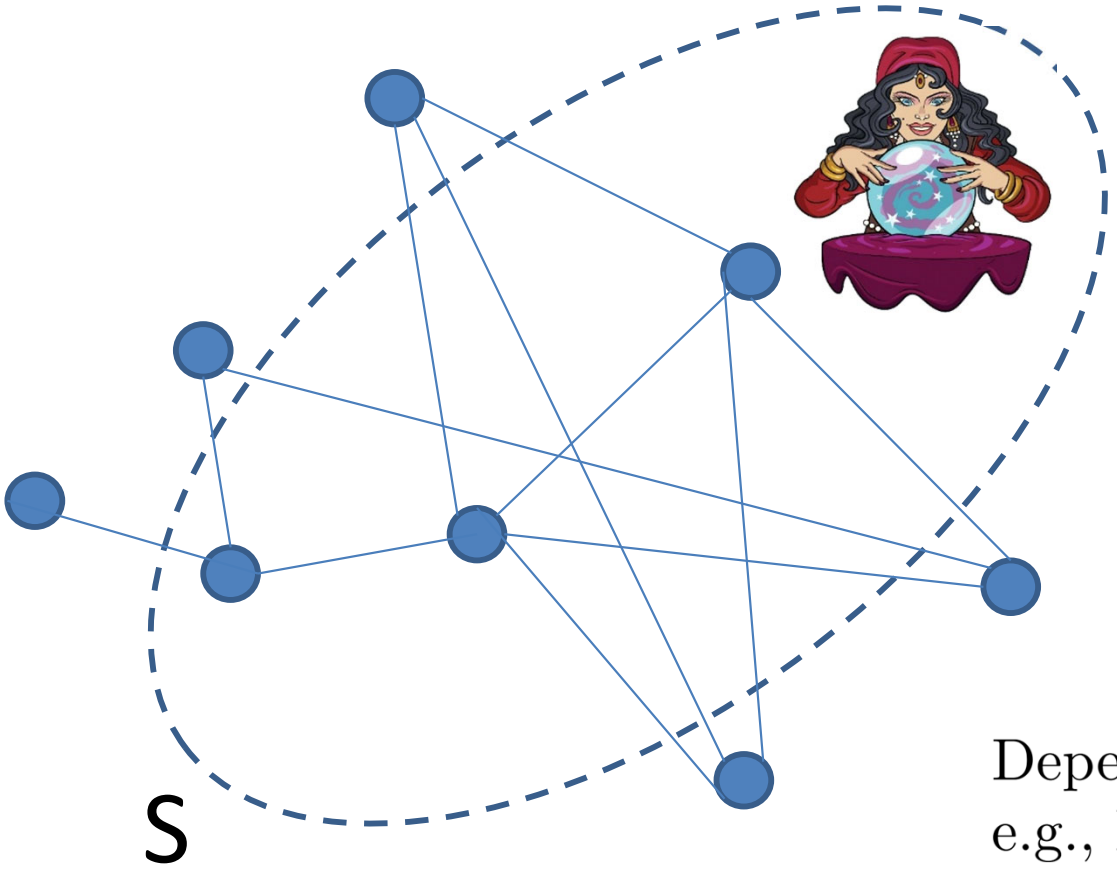
- yes: then S must be an independent set of size $\geq k$.
- no: then, there is no vertex cover $V - S$ of size $\leq |V| - k$, hence there is no IS of size $\geq k$.

□


NP-complete
SAT
Clique
3-CNF-SAT
Independent set
Vertex cover

Part I of the NP-completeness proof: Vertex cover is in NP.

Given a certificate (i..e, a subset of nodes) can we verify if the corresponding subgraph touches all edges in polynomial time?



How many edges we need to check?

At most $\frac{|V|(|V|-1)}{2} \in O(n^2)$

Time to check one edge?

Depends on the data structure,
e.g., linear in the number of edges in G

Applications of the minimum vertex cover

1. **Quantum Computing**: The MVC arises in the study of entanglement. The MVC corresponds to the minimum set of qubits that need to be measured to fully characterize the entanglement of a quantum state.

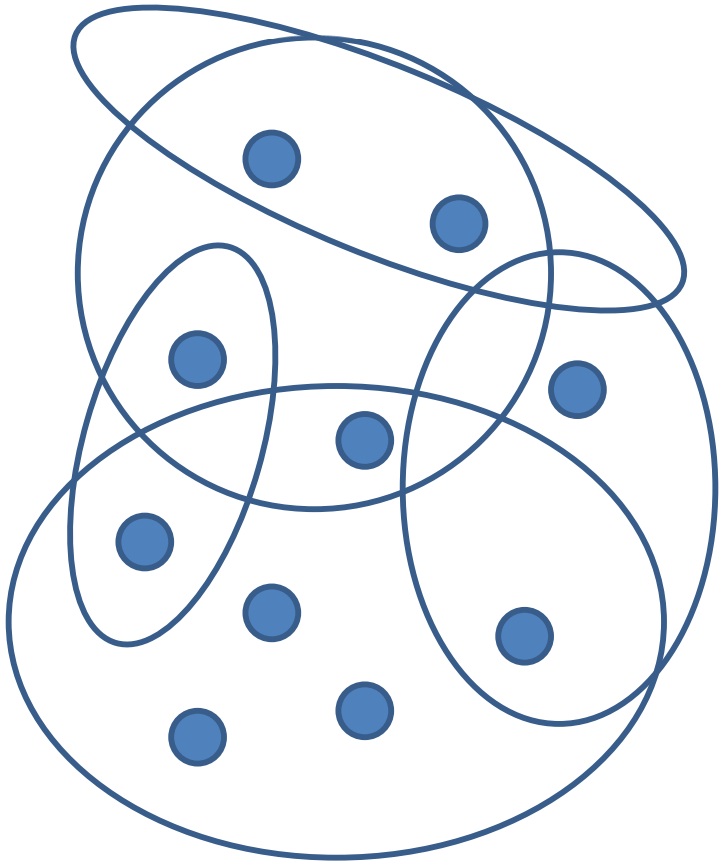
2. **Operations Research**: Determine the placement of expensive surveillance cameras or sensors in a network of streets.

3. **Biology**: The MVC problem can be used to identify the minimum set of genetic markers required to predict an individual's risk of developing a particular disease.

4. **Chemistry**: In chemical graph theory, the MVC problem arises in the study of molecular graphs. The MVC corresponds to the minimum set of atoms that need to be included to represent the entire molecule.

Set Cover (SC) Problem

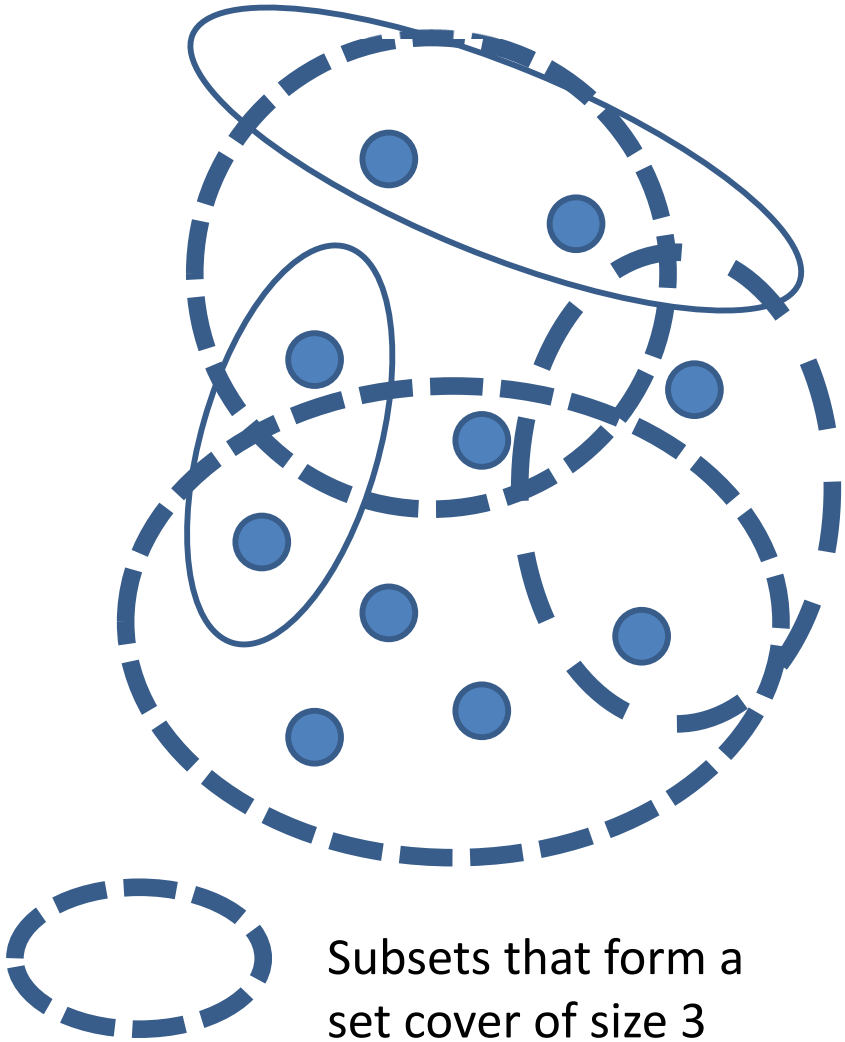
Given a set U of elements and a collection $S_1, ..., S_m$ of subsets of U, is there a collection of at most k of these sets whose union equals U?



Set Cover (SC) Problem

Given a set U of elements and a collection $S_1, \ldots, S_m$ of subsets of U, is there a collection of at most k of these sets whose union equals U?

- Is this problem NP-complete?
- We will show that SC is in NP and
- Vertex Cover is reducible to Set Cover

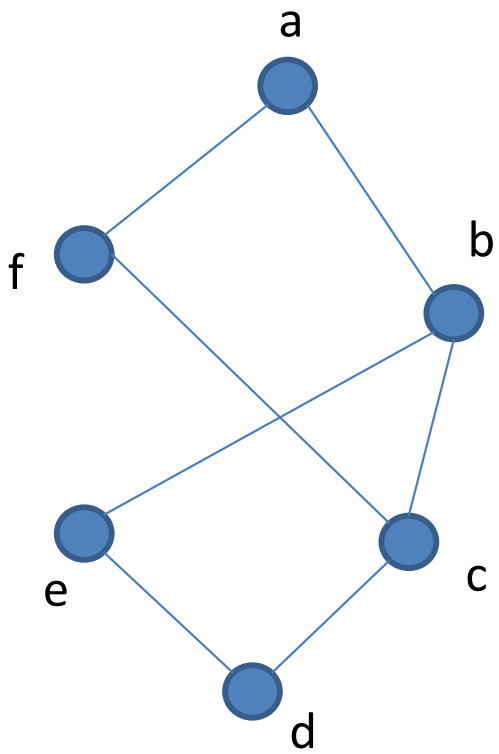


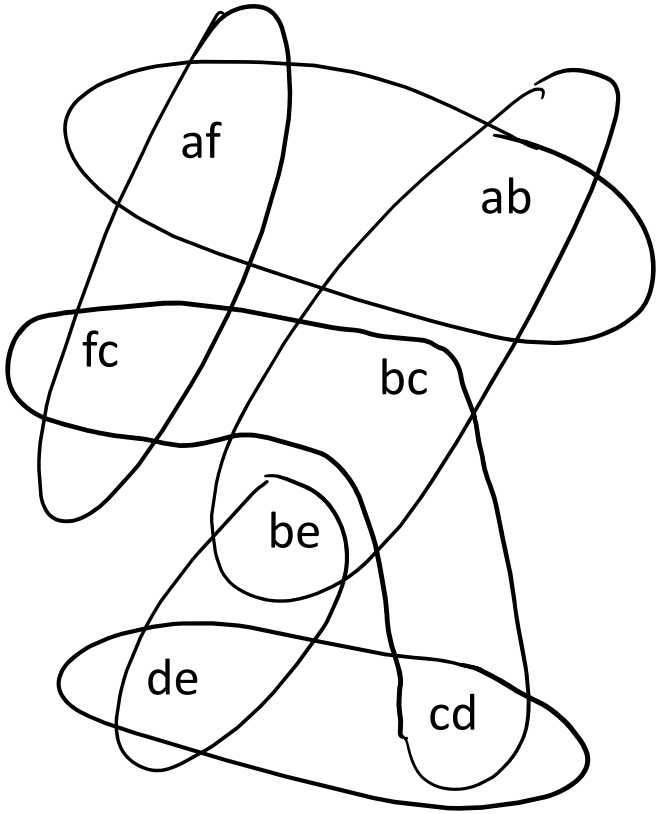
Subsets that form a set cover of size 3

Theorem 1. *Vertex Cover* $\leq_P$ *Set Cover*

Proof. Let $G = (V, E)$ and k be an instance of VC. We create an instance of SC

- $U = E$
- Create S_u for each $u \in V$, where S_u contains the edges adjacent to u.



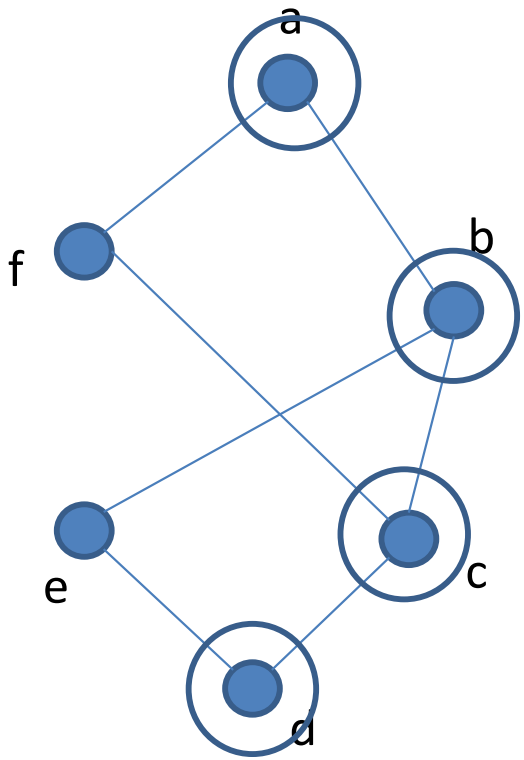


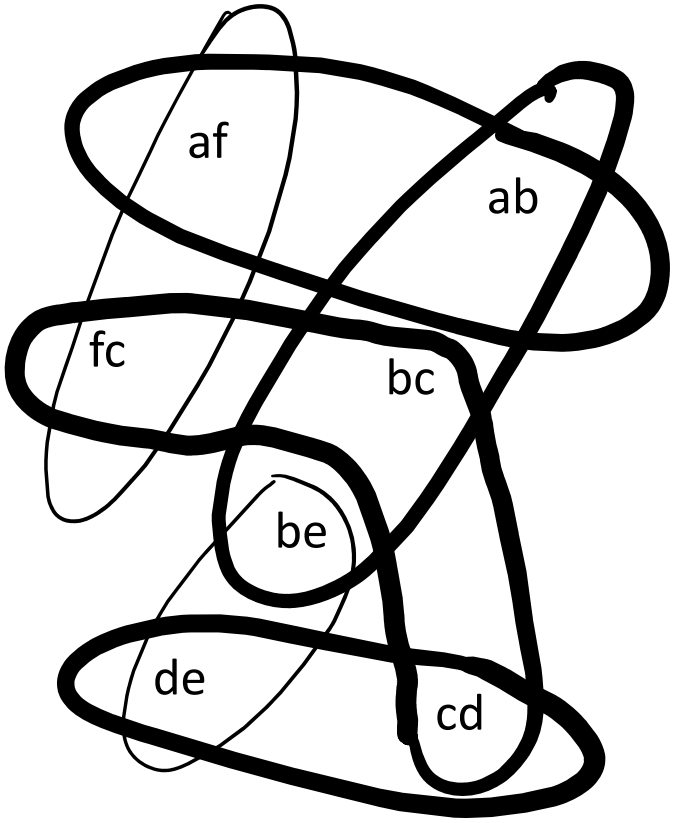
U can be covered by $\leq k$ sets iff G has a vertex cover of size $\leq k$.

Theorem 1. *Vertex Cover* $\leq_P$ *Set Cover*

Proof. Let $G = (V, E)$ and k be an instance of VC. We create an instance of SC

- $U = E$
- Create S_u for each $u \in V$, where S_u contains the edges adjacent to u.





U can be covered by $\leq k$ sets iff G has a vertex cover of size $\leq k$.

Theorem 1. *Vertex Cover* $\leq_P$ *Set Cover*

Proof. Let $G = (V, E)$ and k be an instance of VC. We create an instance of SC

- $U = E$
- Create S_u for each $u \in V$, where S_u contains the edges adjacent to u.

U can be covered by $\leq k$ sets iff G has a vertex cover of size $\leq k$.

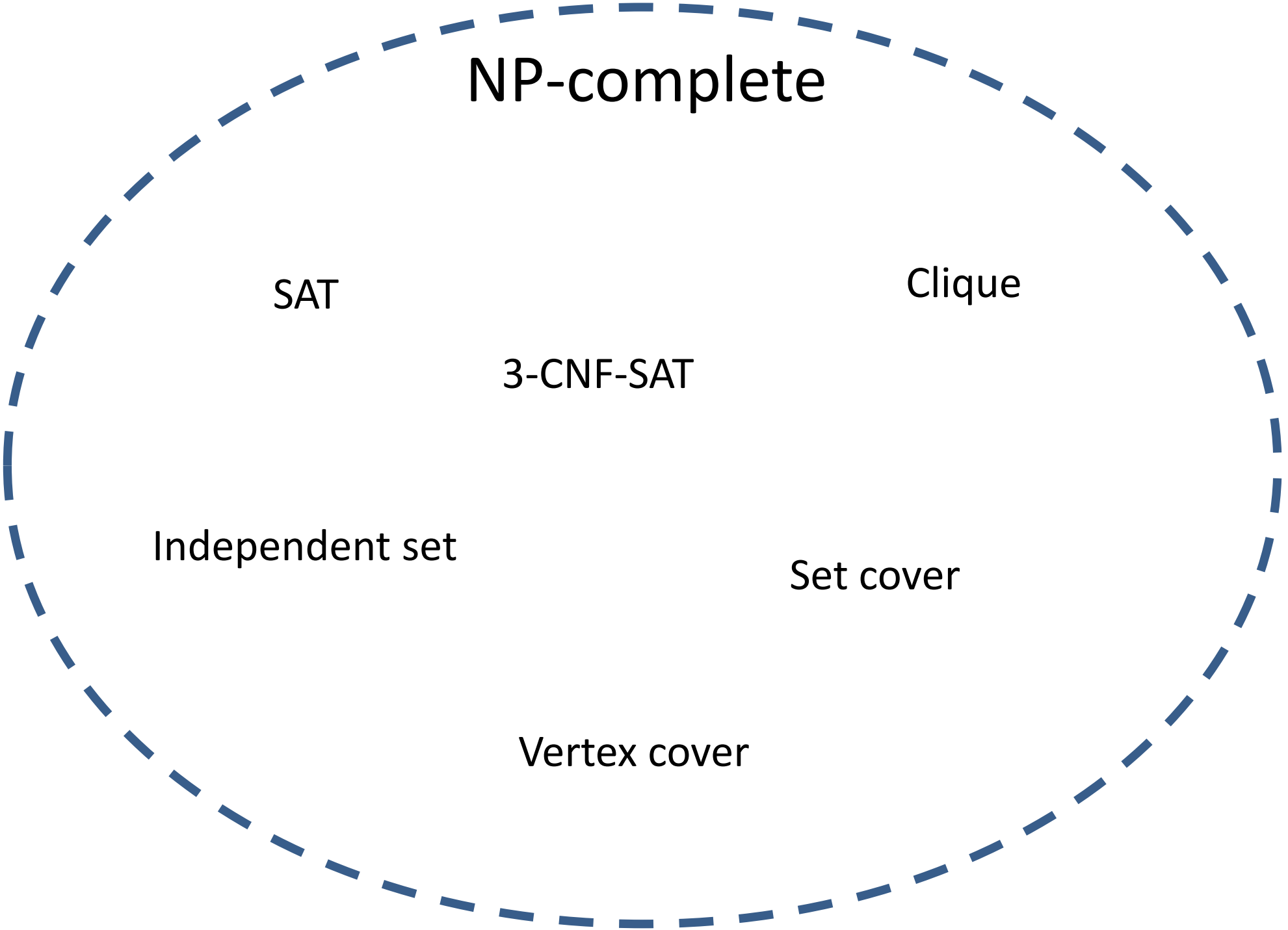
Because if k sets $S_{u_1}, ..., S_{u_k}$ cover U then every edge is adjacent to at least one of the vertices $u_1, ..., u_k$, yealding a vertex cover of size k.

If $u_1, ..., u_k$ is a vertex cover then $S_{u_1}, ..., S_{u_k}$ cover U.

□

Don't forget 1: Why it is polynomial time reduction?

Don't forget 2: Why the problem is in NP? The certificate is a list of *k* subsets. We can check the coverage in polynomial time.


NP-complete
SAT
Clique
3-CNF-SAT
Independent set
Set cover
Vertex cover

Applications of the minimum set cover

1. **Facility Location**: The Set Cover problem arises in the selection of locations for facilities, such as hospitals or schools, to serve a given population. The Set Cover problem can be used to identify the minimum set of facilities that can serve the population.

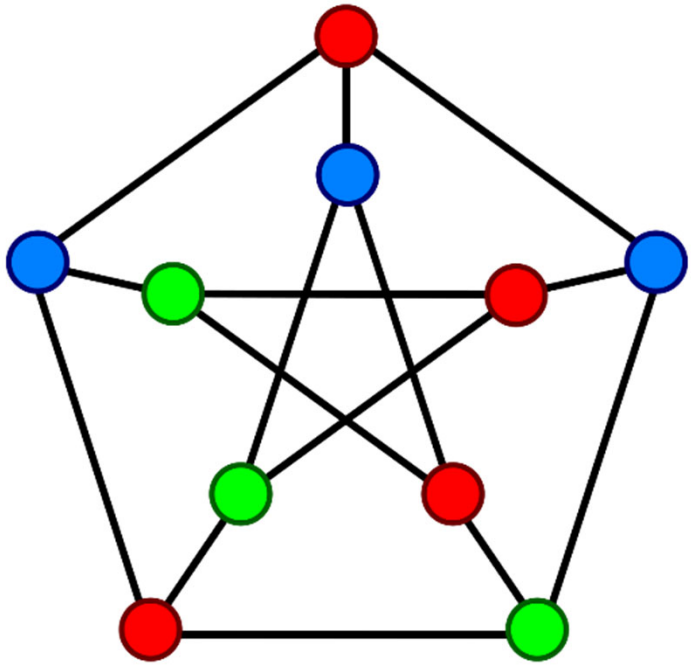
2. **Broadcasting**: The Set Cover problem arises in the selection of a minimum number of expensive broadcast channels to reach a given audience.

3. **Image Processing**: The Set Cover problem arises in the selection of a minimum set of patches to represent an image with high accuracy.

4. **Healthcare Resource Allocation**: The Set Cover problem arises in the selection of a minimum set of healthcare services or interventions to provide to a population in order to improve health outcomes.

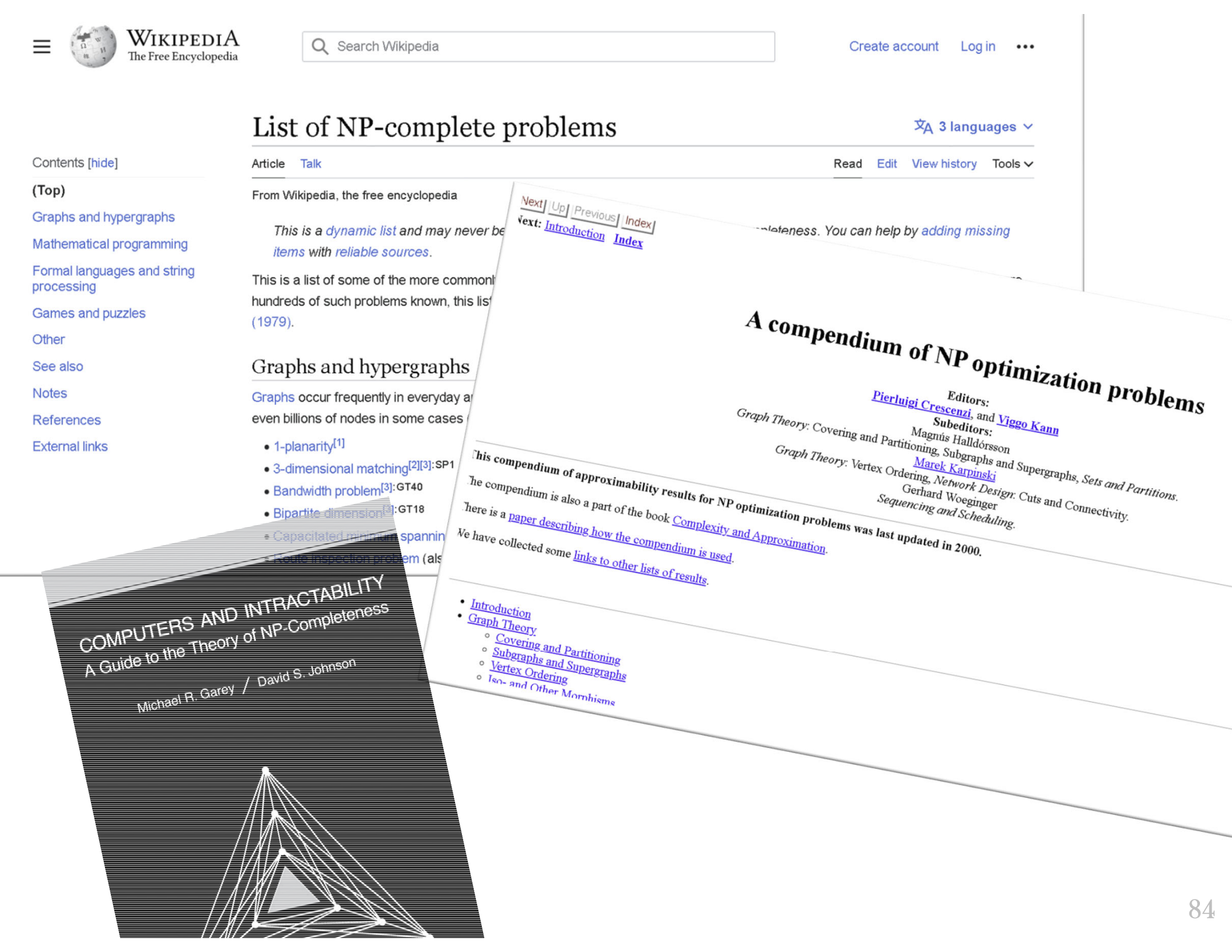
Some Other *NP*-Complete Problems

- A *k-coloring* of *G* is an assignment to each vertex of one of the *k* colors so that no two adjacent vertices are colored the same
- The *k-colorability problem*: Given *G* and *k*, is there a *k*-coloring of *G*?



Some Other *NP*-Complete Problems

- We now have five problems that are *NP*-complete
- There are hundreds of others that are also known to be *NP*-complete
- Many real-life decision problems require some kind of solution
  - If a polynomial-time algorithm does not present itself, it is worth checking whether the problem is *NP*-complete
  - If so, finding such an algorithm will be as hard as proving that $P = NP$


Wikipedia
The Free Encyclopedia
Search Wikipedia
Create account
Log in
List of NP-complete problems
3 languages
Contents [hide]
(Top)
Graphs and hypergraphs
Mathematical programming
Formal languages and string processing
Games and puzzles
Other
See also
Notes
References
External links
Article
Talk
Read
Edit
View history
Tools
From Wikipedia, the free encyclopedia
This is a dynamic list and may never be
items with reliable sources.
This is a list of some of the more commonl
hundreds of such problems known, this lis
(1979).
Graphs and hypergraphs
Graphs occur frequently in everyday a
even billions of nodes in some cases
1-planarity[1]
3-dimensional matching[2][3]:SP1
Bandwidth problem[3]:GT40
Bipartite dimension[3]:GT18
Capacitated minimum spannin
Route inspection problem (als
Next Up Previous Index
Next: Introduction Index
leteness. You can help by adding missing
A compendium of NP optimization problems
Editors:
Pierluigi Crescenzi, and Viggo Kann
Subeditors:
Magnús Halldórsson
Graph Theory: Covering and Partitioning, Subgraphs and Supergraphs, Sets and Partitions.
Marek Karpinski
Graph Theory: Vertex Ordering, Network Design: Cuts and Connectivity.
Gerhard Woeginger
Sequencing and Scheduling.
This compendium of approximability results for NP optimization problems was last updated in 2000.
The compendium is also a part of the book Complexity and Approximation.
There is a paper describing how the compendium is used.
We have collected some links to other lists of results.
Introduction
Graph Theory
Covering and Partitioning
Subgraphs and Supergraphs
Vertex Ordering
Iso- and Other Morphisms
COMPUTERS AND INTRACTABILITY
A Guide to the Theory of NP-Completeness
Michael R. Garey / David S. Johnson

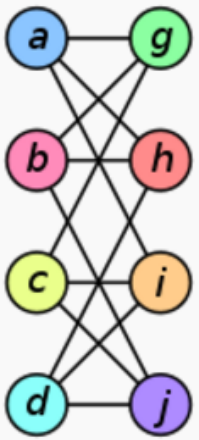
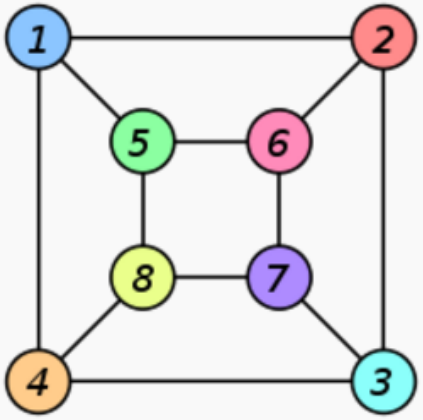
We still do not know the status of some problems

Example: Graph Isomorphism

An Isomorphism of graphs *G* and *H* is a bijection between the vertex sets of *G* and *H*

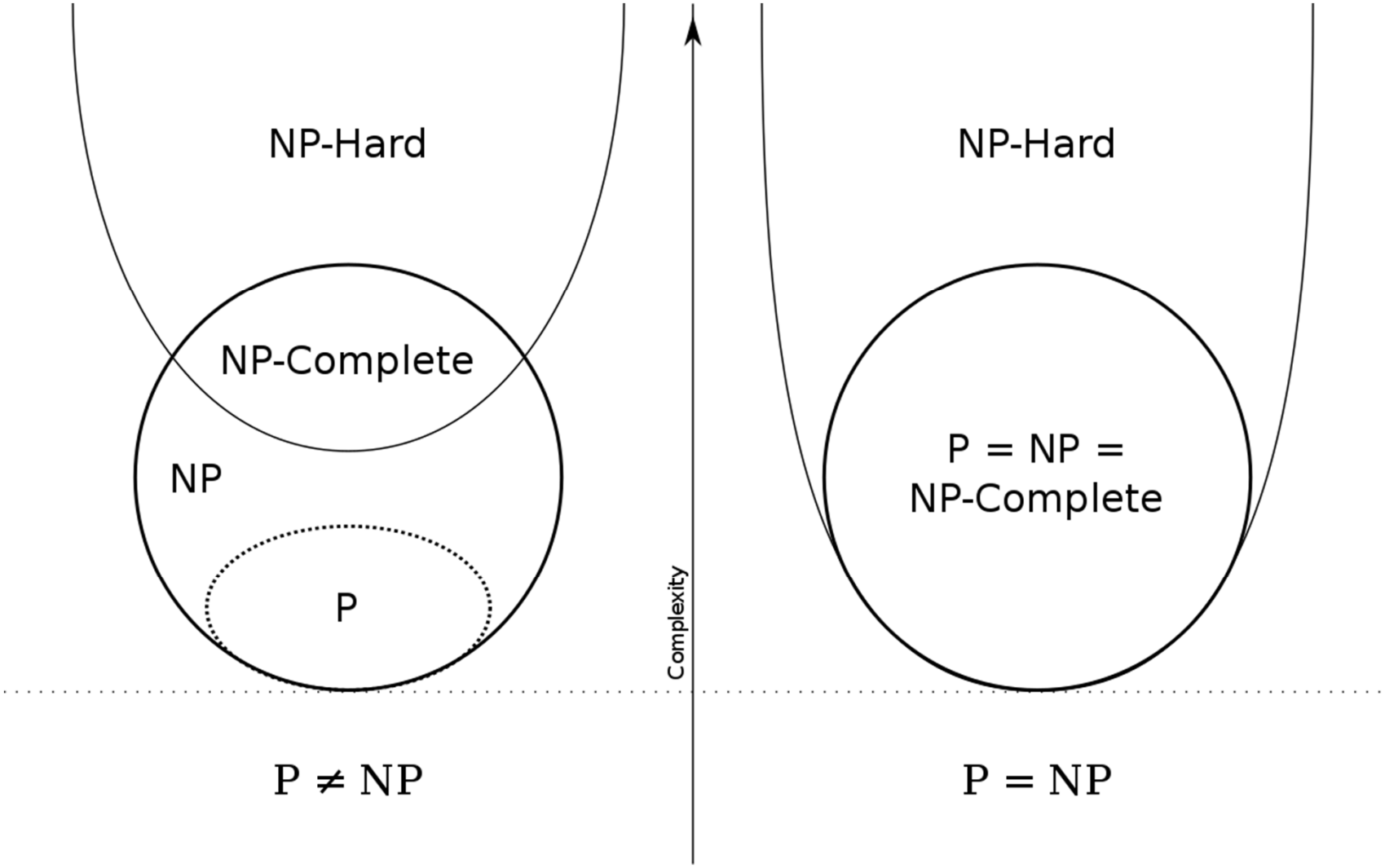
$f(V(G)) \rightarrow V(H)$

such that any two vertices *u* and *v* of *G* are adjacent in *G* if and only if *f(u)* and *f(v)* are adjacent in *H*.

Graph G	Graph H	An isomorphism between G and H
		$f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(g) = 5$ $f(h) = 2$ $f(i) = 4$ $f(j) = 7$

We still don't know the status of this problem!
Clearly, it is in NP but we don't know whether this problem is NP-complete or in P.

NP-Hard: These are at least as hard as any problem in NP. If we can solve these problems in polynomial time, we can solve any NP problem that can possibly exist. Note that these problems are not necessarily in NP, i.e., we may/may-not verify the solution in polynomial time.



NP-Complete: These are the problems which are both NP and NP-Hard. That means, if we can solve these problems, we can solve any other NP problem and the solutions to these problems can be verified in polynomial time.

Well … the decision problem is NP-complete, the corresponding optimization problem is NP-hard. What now?

- Can we hope for a fast algorithm that guarantees a "pretty good" solution?
- In many cases, the answer is "yes".

- Approximation algorithms (for example, greedy algorithms, probabilistic algorithms, …)
- Heuristics (do not produce provably good solutions on all instances but have been proven to be good in practice on many instances)

Approximation Algorithms

- A minimization problem is an optimization problem, where we look for a valid (or feasible) solution that minimizes a certain target function.

- For example: in the minimum vertex cover problem, we are looking of a minimum size subset of nodes that is a vertex cover; in the minimum coloring problem, we are looking for the minimum number of colors, and the coloring itself.

- Let Opt(I) denote the value of the target function for the optimal solution.

- Algorithm Alg for a minimization problem Min achieves an approximation factor $\alpha \geq 1$ if for all inputs I, we have $Alg(I)/Opt(I) \leq \alpha$. We will refer to Alg as an α-approximation algorithm for Min

Approximation Algorithm for the Minimum Vertex Cover

The MVC problem has many applications. Example: what is the fewest number of cameras we need to install in a bank in order to cover all its corridors. So, it is important to have a fast algorithm that produces good solutions.

Algorithm 1: Pick an arbitrary vertex with at least one uncovered edge incident to it, put it into the cover, and repeat.
Is it good or bad?

It is bad. An example of bad behavior: a star graph.

Algorithm 2: Pick a vertex that covers most uncovered edges.
Is it good or bad?
It is bad. An example is complicated but it produces an approximation with a factor of log(n)

Approximation Algorithm for the Minimum Vertex Cover

Algorithm 3: Pick an arbitrary edge. Add both of its endpoints to the vertex cover. Then, throw out all edges covered and repeat.

Is it good or bad?

It is not as bad as previous examples.
It gives an approximation factor 2.

Proof: The algorithm finds a matching (a set of edges no two of which share an endpoint) that is "maximal" (meaning that you can't add any more edges to it and keep it a matching).
This means if we take both endpoints of those edges, we must have a vertex cover. In particular, if the algorithm picked k edges, the vertex cover found has size 2k. But, any vertex cover must have size at least k since it needs to have at least one endpoint of each of these edges. So the algorithm is a 2-approximation.