Chapter 5
Computational Complexity Theory

Computational complexity theory

- Even when a problem is decidable and thus computationally solvable in principle, it may not be solvable in practice if the solution requires an inordinate amount of resources (time or space)
- And the computational complexity theory tries to investigate the time, memory, or other resources required for solving computational problems.
- But, we limit our discussion to issues of time – complexity.

Computational complexity theory

Goal: A general theory of the *resources* needed to solve *computational problems*.

What types of resources?

time

energy

space

What types of computational problems?

composing a poem

optimization

sorting a database

decision problem

Decision problems

A decision problem is a computational problem with a yes or no answer.

Example: Is the number *n* prime?

Why focus on decision problems?

Decision problems are simple: This makes it easy to develop a rigorous mathematical theory.

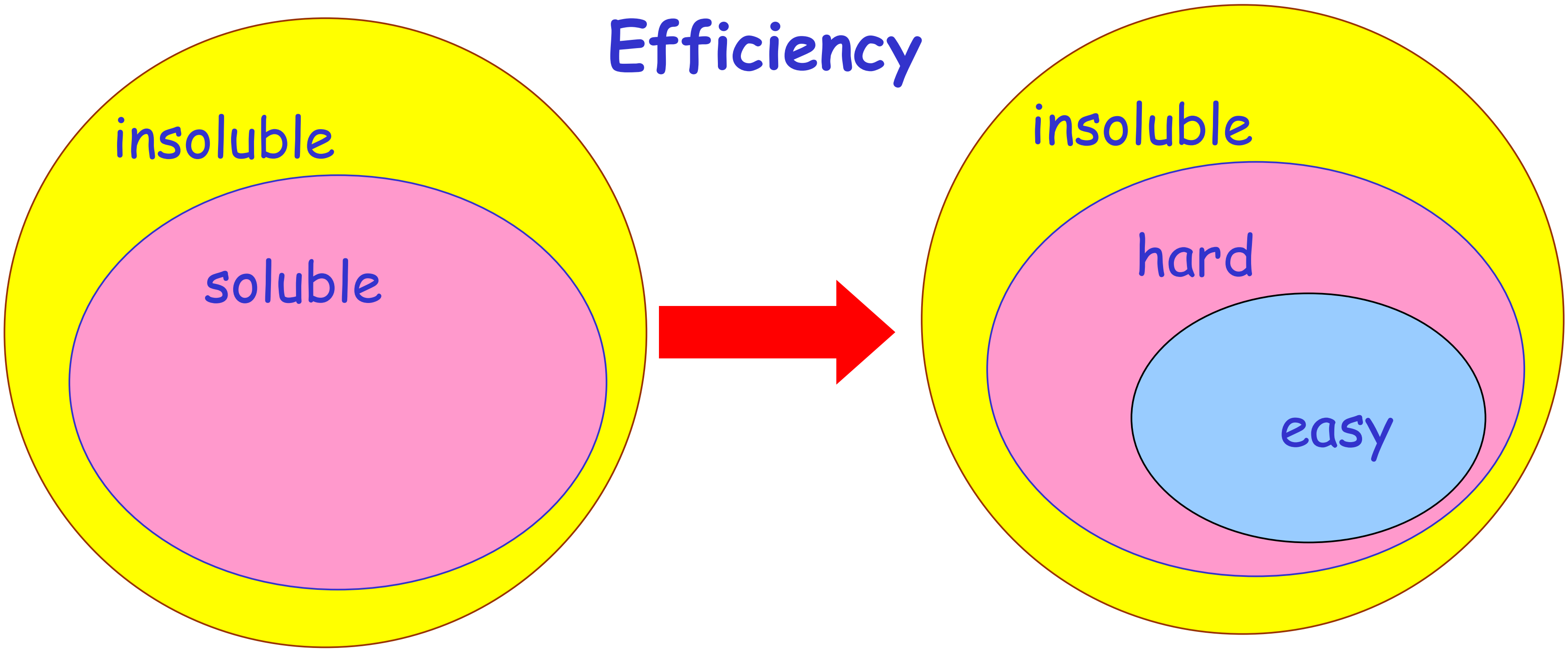
Recasting other problems as decision problems

Multiplication problem: What is the product of *m* and *n*?

Multiplication decision problem: Is the *k*th bit of the product of *m* and *n* a one?

Time required to solve one of these problems is the same (to within a small overhead) as the time required to solve the other.

Efficiency



Nomenclature: easy = "tractable" ="efficiently computable"

hard = "intractable" = "not efficiently computable"

Definition: A problem is *easy* if there is a Turing machine to solve the problem that runs in time *polynomial* in the size of the *problem input*. Otherwise the problem is *hard*.

A problem is *poly-time solvable* if $T(n) = O(n^k)$ for some constant k.

This definition is usually applied to both decision problems and more general problems.

Time Complexity

Time Complexity: The number of steps during a computation

Space Complexity: Space used during a computation

Motivation

Our main goal in this course is to analyze problems and categorize them according to their complexity.

Measuring Time Complexity

- Model of computation: TM
- *timereq*(M) is a function of n:
  - If M is ***deterministic***

    timereq(M) = $f(n)$ = the maximum number of steps that M executes on any input of length n.
  - If M is ***nondeterministic***

    timereq(M) = $f(n)$ = the number of steps on the longest path that M executes on any input of length n.

Big-O-Notation(Asymptotic analysis)

- Cosidering the highest order term of expression for the running time of algorithm,disregarding both the coefficient of that term and any lower order terms.
- Because the highest order term dominates the order terms on large inputs.
- $f(n) = 6n^3 + 2n^2 + 20n + 45$
- Highest order is $6n^3$.By disgarding the coffecient
- Big O notation f(n)=O(n^3).

Asymptotic upper bound - $\mathcal{O}$

$f(n) \in \mathcal{O}(g(n))$ iff there exists a positive integer k and a positive constant c such that:

$$\forall n \geq k, \; (f(n) \leq c\, g(n)).$$

In other words, ignoring some number of small cases (all those of size less than k), and ignoring some constant factor c, $f(n)$ is bounded from above by $g(n)$.

In this case, we'll say that f is "big-oh" of g or g asymptotically dominates f or g grows at least as fast as f does

Complexity Classes

1. P (Polynomial Time):
   P represents the class of decision problems that can be solved by a deterministic Turing machine in polynomial time. In other words, these problems have efficient algorithms with a runtime that is polynomial in the size of the input.

2. NP (Nondeterministic Polynomial Time):

NP represents the class of decision problems for which a potential solution can be verified in polynomial time by a deterministic Turing machine. However, finding the solution itself may require more than polynomial time. The famous example is the Boolean satisfiability problem (SAT), which asks whether there exists an assignment of truth values to variables that satisfies a given logical formula.

3. NP-Complete (Nondeterministic Polynomial-Time Complete):

NP-Complete is a subset of NP problems that are believed to be among the hardest problems in NP. A problem is NP-Complete if every problem in NP can be reduced to it in polynomial time. The classic example is the traveling salesman problem (TSP), which involves finding the shortest possible route that visits each city exactly once and returns to the starting city.

4. NP-Hard (Nondeterministic Polynomial-Time Hard):

NP-Hard represents the class of problems that are at least as hard as the hardest problems in NP. Unlike NP-Complete problems, they may or may not be in NP. An example of an NP-Hard problem is the halting problem, which asks whether a given program halts on a specific input.

5. EXP (Exponential Time):

EXP represents the class of decision problems that can be solved by a deterministic Turing machine in exponential time. These problems require an exponential amount of time to solve as the input size increases. An example is the subset sum problem, which asks whether there is a subset of numbers that sum up to a given target value.

6. PSPACE (Polynomial Space):

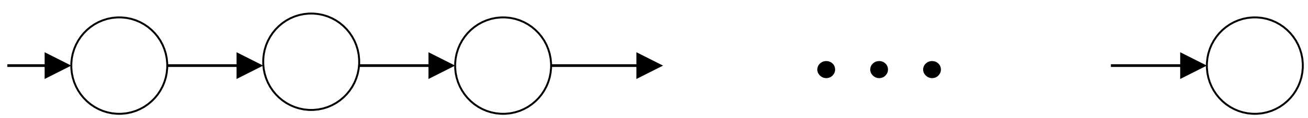
PSPACE represents the class of problems that can be solved by a deterministic Turing machine using a polynomial amount of memory space. Examples include solving chess or solving the game of Go, which require storing the game state and exploring possible moves within a limited amount of memory.

Complexity classes: are set of languages/ functions that can be decided/ computed within a given resource.

Now let's introduce our first complexity classes

Consider a <u>deterministic</u> Turing Machine which <u>decides</u> a language

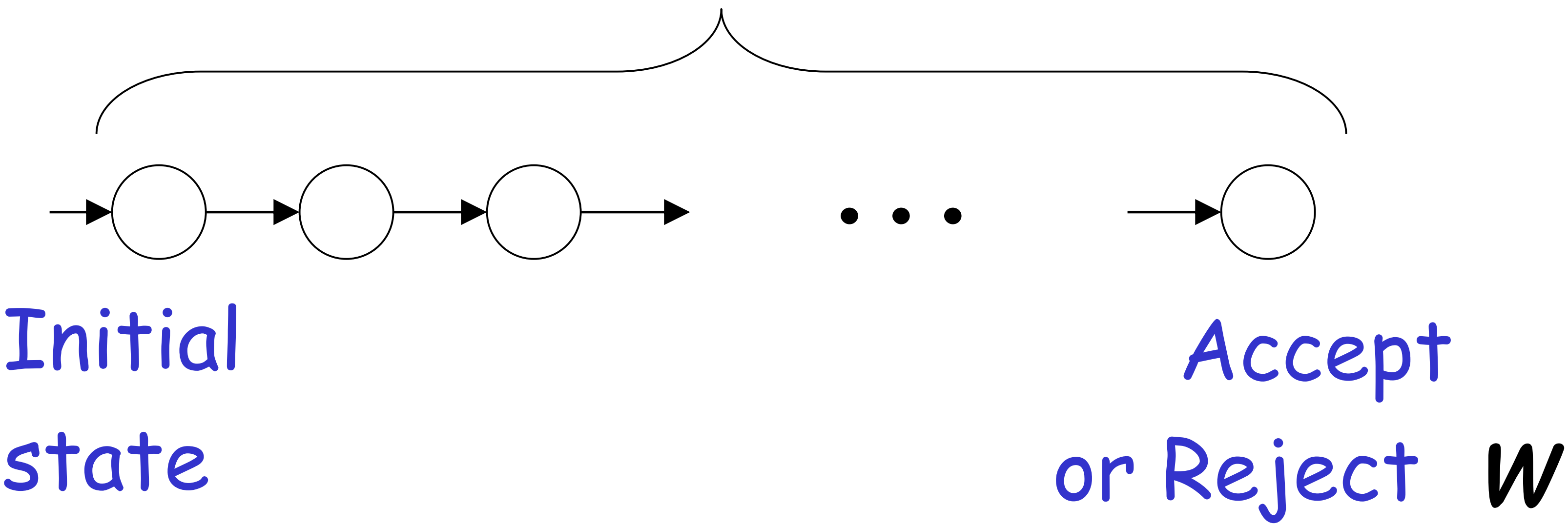
For any string w the computation of M terminates in a finite amount of transitions



Initial state

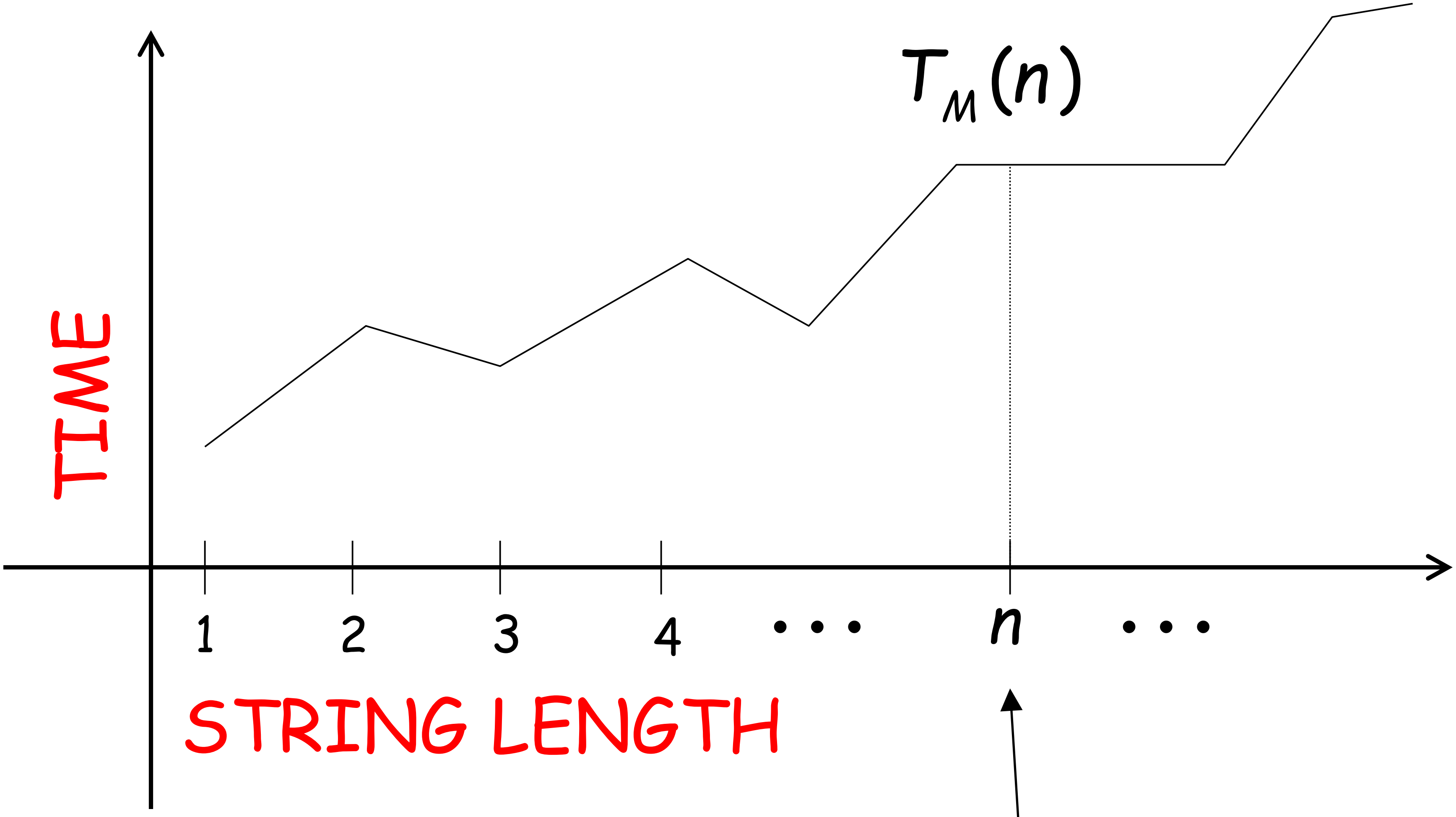
Accept or Reject w

Decision Time = #transitions



Consider now all strings of length n

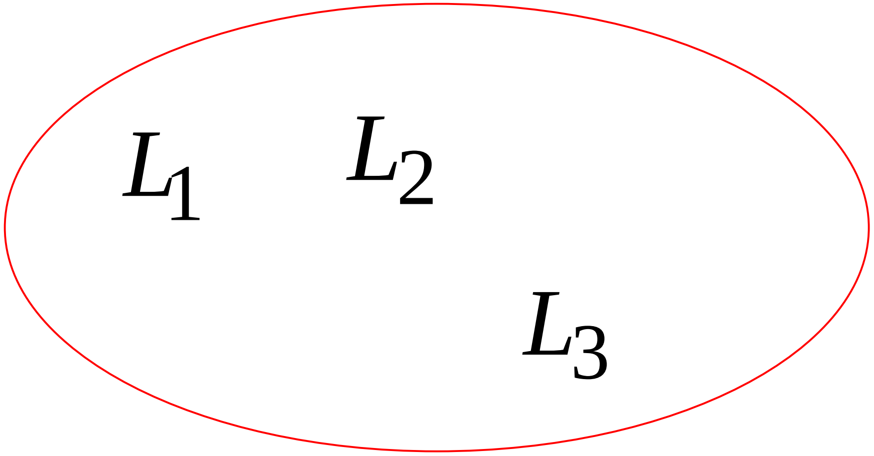
$T_M(n)$ = maximum time required to decide any string of length n



Max time to accept a string of length n

Time Complexity Class: $TIME(T(n))$

All Languages decidable by a deterministic Turing Machine in time $O(T(n))$



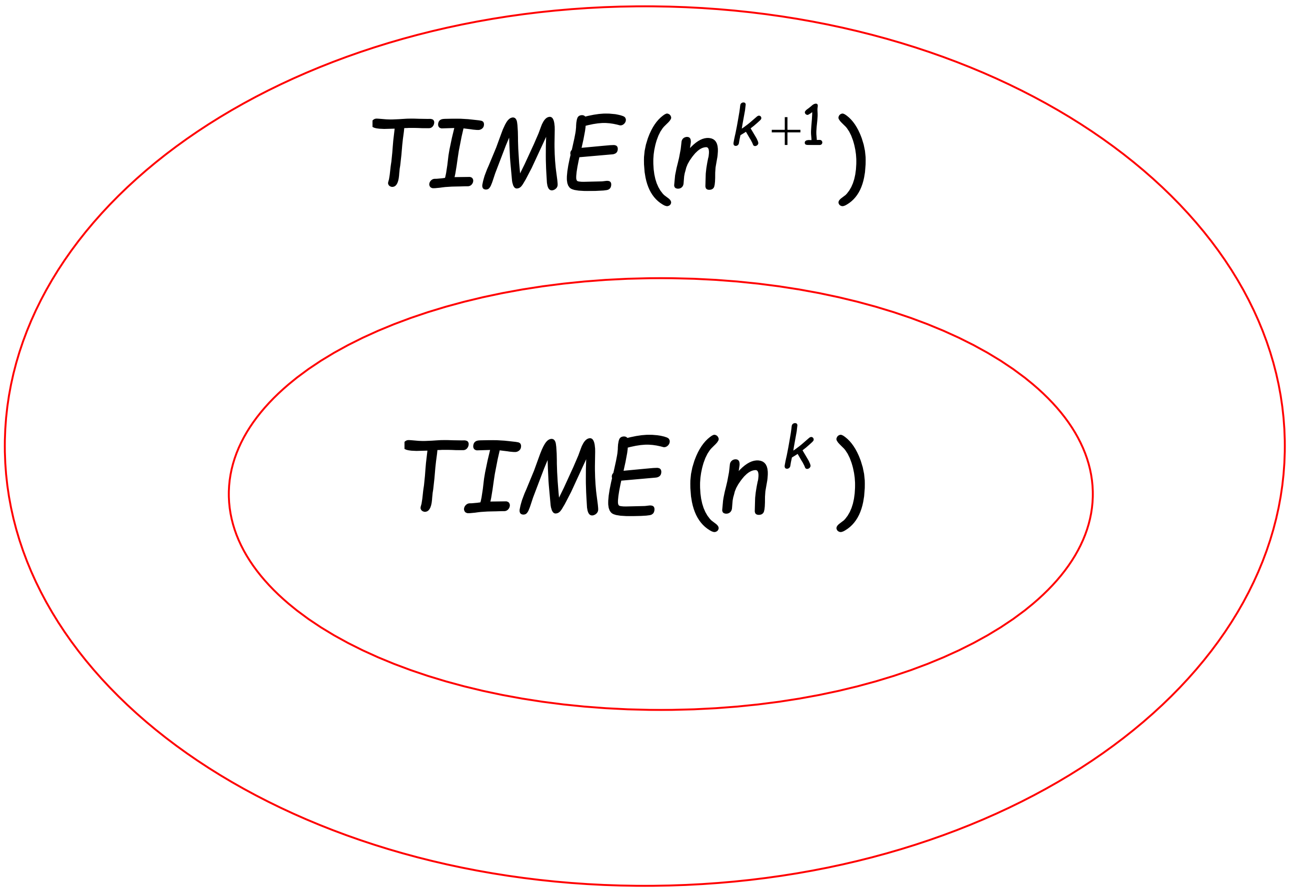
Polynomial time algorithms: $TIME(n^k)$

constant $k > 0$

Represents tractable algorithms:

for small k we can decide the result fast

It can be shown: $TIME(n^{k+1}) \subset TIME(n^{k})$



The Time Complexity Class P

- Class P is a set of languages that are decidable in polynomial time on a deterministic single tape Turing machine.

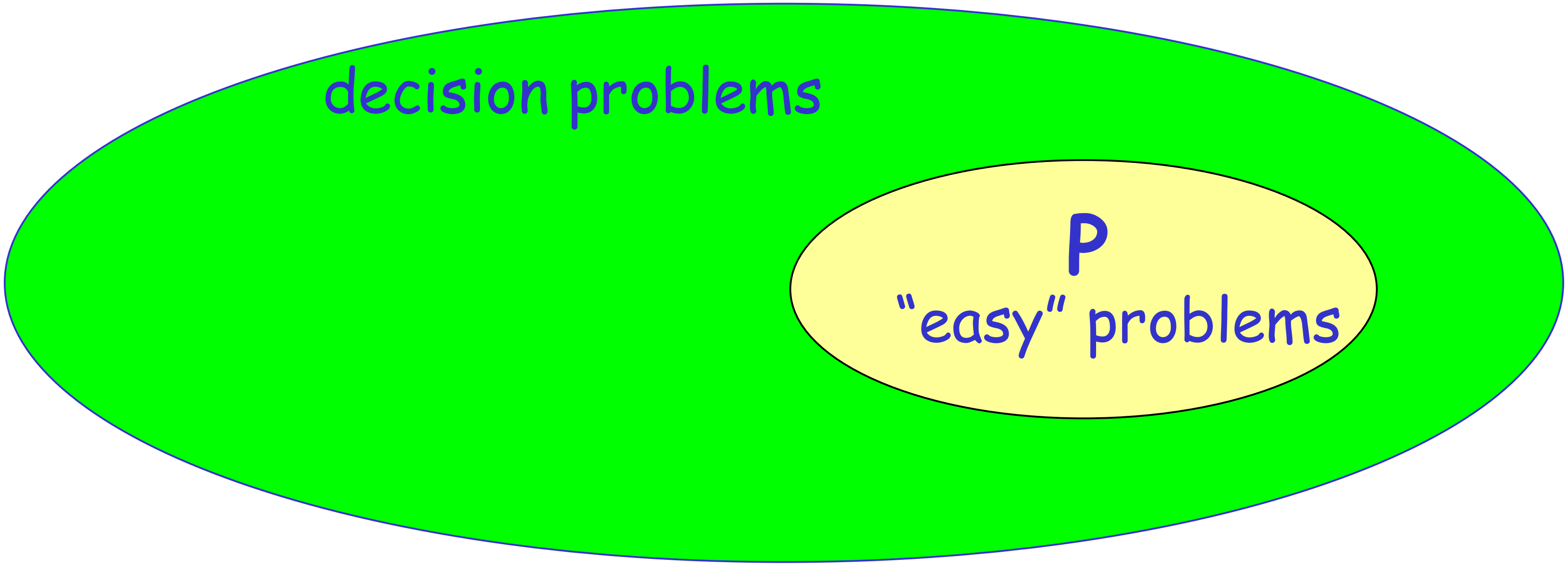
$$P = \bigcup_{k>0} TIME(n^k)$$

Represents:

- polynomial time algorithms
- "tractable" problems

Our first computational complexity class: "P"

Definition: The set of all decision problems soluble in *polynomial time* on a Turing machine is denoted **P**.



Terminology: "Multiplication is in **P**" means "The multiplication decision problem is in class **P**".

"Factoring is thought not to be in **P**" means "The factoring decision problem is thought not to be in class **P**".

Many important problems aren't known to be in P

There are many problems in which a polynomial time algorithm not known to exist to solve them.

Example: Factoring.

Exponential time algorithms: $TIME(2^{n^k})$

Represent intractable algorithms:

Some problem instances
may take centuries to solve

Non-Determinism

Language class: $NTIME(n)$

$NTIME(n)$

L_1 L_2 L_3

A Non-Deterministic Turing Machine accepts each string of length n in time $O(n)$

In a similar way we define the class

$$NTIME(T(n))$$

for any time function: $T(n)$

Examples: $NTIME(n^2), NTIME(n^3), \ldots$

Non-Deterministic Polynomial time algorithms:

$$L \in NTIME(n^k)$$

The class NP

$$NP = \bigcup_{k>0} NTIME(n^k)$$

Non-Deterministic Polynomial time

NP Class

- NP is the class of languages that have polynomial time verifiers.
- There are problems *hard* to determine the solution but easy to *verify* the solution once it is found
- The term NP comes from nondeterministic polynomial time by using nondeterministic Turing machines.
- Problems in NP are sometimes called NP-problems.

Non-determinstic algorithm:

The algorithm in which every operation is uniquely defined is called deterministic algorithm.

The algorithm in which every operation may not have unique result, rather there can be specified set of possibilities for every operation, such an algorithm is called non deterministic algorithm

.non deterministic means no particular rule is followed to make the guess.

The non deterministic algorithm is a two stage algorithm:

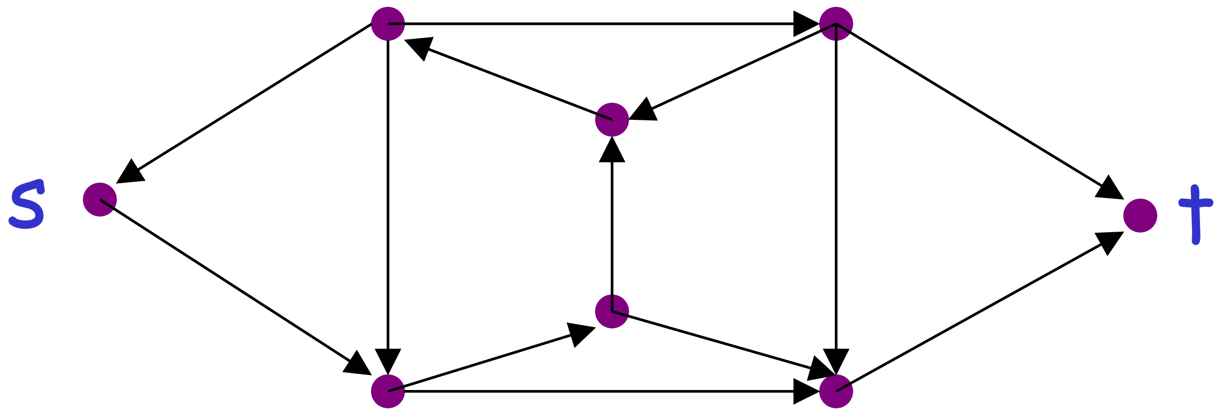
1)Non deterministic (Guessing stage)- generate an arbitrary string that can be thought of as a candidate solution.

2)Determinstic (" verification") Stage:In this stage,it takes as input, the candidate solution and the instance to the problem and returns yes if the candidate solution represents actual solution.

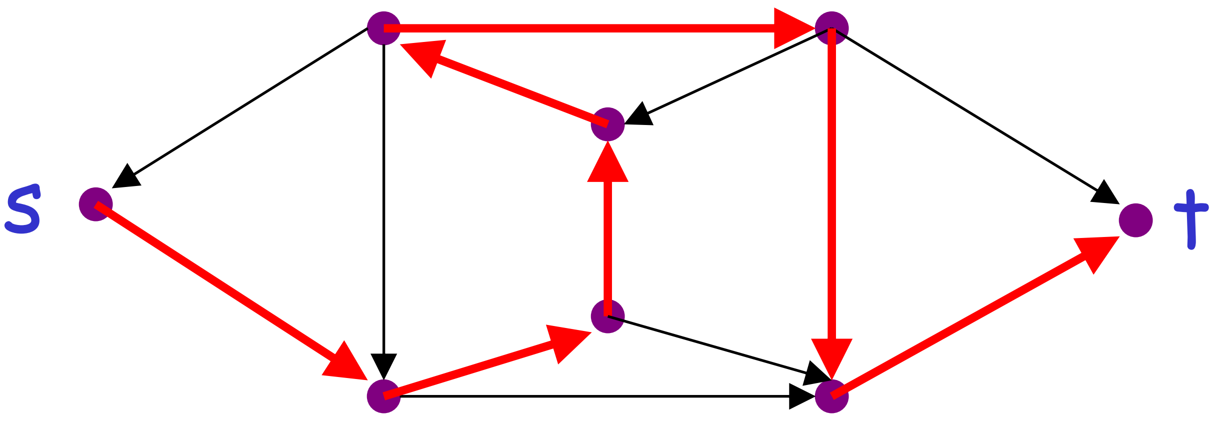
- In this algorithm, we use three functions used:
- Choose()-arbitarily chooses one of the element from given input set.
- Fail()-indicates unsuceessful completion
- Verify()-indicates successful completion.

Other Examples

Example: the Hamiltonian Path Problem



Question: is there a Hamiltonian path from s to t?



YES!

- **Hamiltonian Cycle**:This is a problem in which graph *G* is accepted as input and it is asked to find a simple cycle in *G* that visits each vertex of *G* exactly once and returns to its starting vertex,Such a cycle is called an Hamiltonian Cycle.
- **Theorem**:Hamiltonian Cycle is in NP.
- **Proof**:Let *A* be some non-determinstic algorithm to which graph *G* is given as input.

- The vertices of graph are numbered from 1 to N. We have to call the algorithm recursively in order to get the sequence S.
- This sequence will have all the vertices without getting repeated.
- The vertex from which the sequence starts must be ended at the end.
- This check on the sequence S must be made in polynomial time n.
- We need only add a check to verify that the potential sequence is Hamiltonian.

- Now if there is a Hamiltonian Cycle in the graph then algorithm will output "yes".
- Similarly if we get output of algorithm as "yes" then we could guess the cycle in *G* with every vertex appearing exactly once and the first visited vertex getting visited at the last.
- That means *A* non-deterministically accepts the language HAMILTONIAN CYCLE. It is therefore proved that HAMILTONIAN CYCLE is in NP.

Example: The Satisfiability Problem

Boolean expressions in
Conjunctive Normal Form:

$$t_1 \wedge t_2 \wedge t_3 \wedge \cdots \wedge t_k \quad \text{clauses}$$

$$t_i = x_1 \vee \bar{x}_2 \vee x_3 \vee \cdots \vee \bar{x}_p$$

Variables

satisfying assignment: a true assignment causing the output to be 1.

Question: is the expression satisfiable?

Example: The satisfiability problem

$$L = \{w : \text{expression } w \text{ is satisfiable}\}$$

Non-Deterministic algorithm:

- Guess an assignment of the variables
- Check if this is a satisfying assignment

Example: $(\overline{x}_1 \vee x_2) \wedge (x_1 \vee x_3)$

Satisfiable: $x_1 = 0,\ \ x_2 = 1,\ \ x_3 = 1$

$$(\overline{x}_1 \vee x_2) \wedge (x_1 \vee x_3) = 1$$

Example: $(x_1 \vee x_2) \wedge \overline{x}_1 \wedge \overline{x}_2$

Not satisfiable

$$L = \{w : \text{expression } w \text{ is satisfiable}\}$$

Time for n variables:

- Guess an assignment of the variables $O(n)$

- Check if this is a satisfying assignment $O(n)$

Total time: $O(n)$

$$L = \{w : \text{expression } w \text{ is satisfiable}\}$$

$$L \in NP$$

The satisfiability problem is an NP - Problem

The P vs. NP Question

P: Languages for which membership can be decided quickly

- Solvable by a DTM in poly-time

NP: Languages for which membership can be verified quickly (i.e. can be tested in poly-time)

- Solvable by a NDTM in poly-time

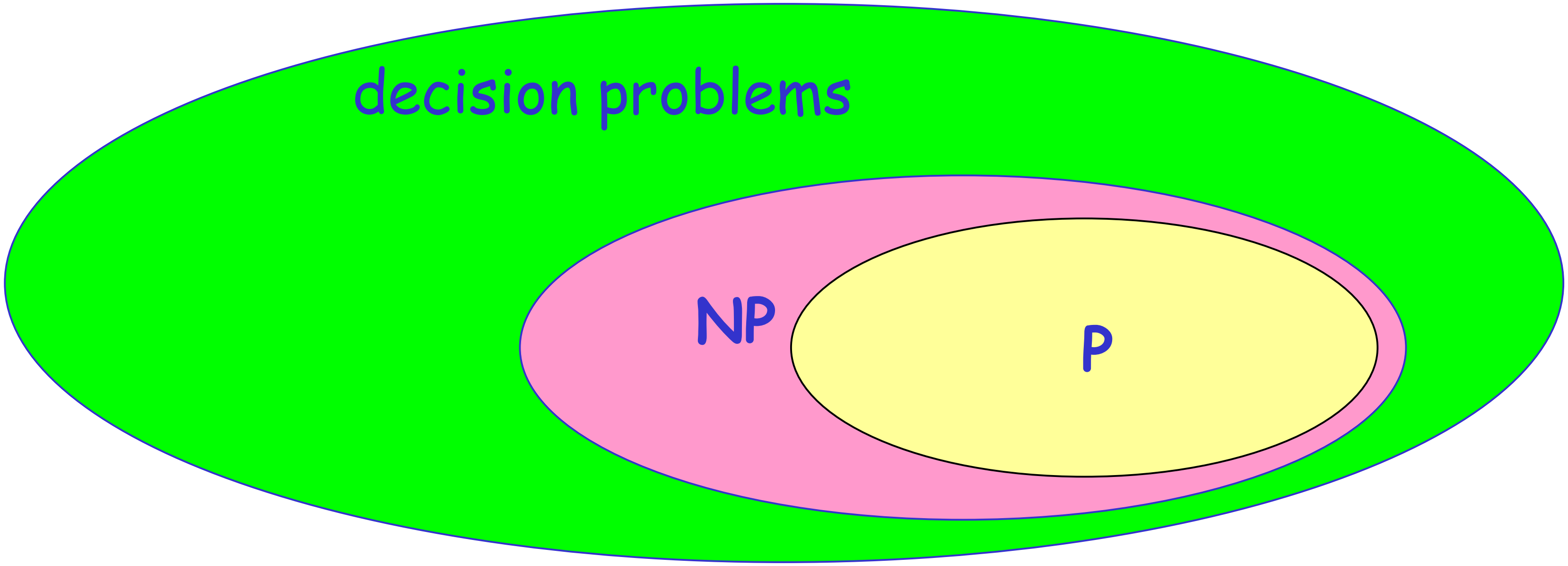
Set **L** is in **P** if membership in **L** can be decided in poly-time.

Set **L** is in **NP** if each x in **L** has a short "proof of membership" that can be verified in poly-time.

Fact: $P \subseteq NP$

Question: Does $NP \subseteq P$?

The relationship of P to NP



Observation:

$$P \subseteq NP$$

Deterministic Polynomial

Non-Deterministic Polynomial

NP includes all problems in **P** and some problems possibly outside **P**

Open Problem: $P = NP$?

WE DO NOT KNOW THE ANSWER

This is the most famous *unsolved problem* in computer science and mathematics.

Open Problem: $P = NP$?

Example: Does the Satisfiability problem have a polynomial time deterministic algorithm?

If these classes were equal, then any polynomally verifiable problem would be polynomially decidable.

WE DO NOT KNOW THE ANSWER

Why Care?

NP Contains Lots of Problems We Don't Know to be in P.

Some of them are:

Hamiltonian cycle

Clique

Classroom Scheduling

Packing objects into bins

Scheduling jobs on machines

Finding cheap tours visiting a subset of cities

Allocating variables to registers

Finding good packet routings in networks

Decryption

…

To reason about the P=NP problem?

That is how can we prove that **NP** $\subseteq$ **P**?

I would have to show that every set in **NP** has a polynomial time algorithm…

How do I do that?

It may take a long time!

Also, what if I forgot one of the sets in **NP**?

We can describe just one problem **L** in **NP**, such that if this problem L is in **P**, then **NP** $\subseteq$ **P**.

It is a problem that can capture all other problems in **NP**.

Polynomial Time Reductions

- Consider L' is polynomial time reducible to K if there is a polynomial time bounded TM that for each input x produces an output y that is in L if x is in L'

$\leq_p$ is transitive(Polynomialy reducible):

if $\mathbf{L_1 \leq_p L_2}$ and $\mathbf{L_2 \leq_p L_3}$, then $\mathbf{L_1 \leq_p L_3}$

if $\mathbf{L_2}$ Ɛ **NP** and $\mathbf{L_1 \leq_p L_2}$, then $\mathbf{L_1}$ Ɛ **NP**.

Example of a polynomial-time reduction:

We will reduce the

3CNF-satisfiability problem

to the

CLIQUE problem

3CNF definition:

- A *literal* in a boolean formula is an occurrence of a variable or its negation.
- CNF (Conjunctive Norman Form) is a boolean formula expressed as AND of clauses, each of which is the OR of one or more literals.
- 3CNF is a CNF in which each clause has exactly 3 distinct literals (a literal and its negation are distinct)

3CNF-SAT Problem: whether a given 3CNF is satisfiable?

3CNF formula:

literal

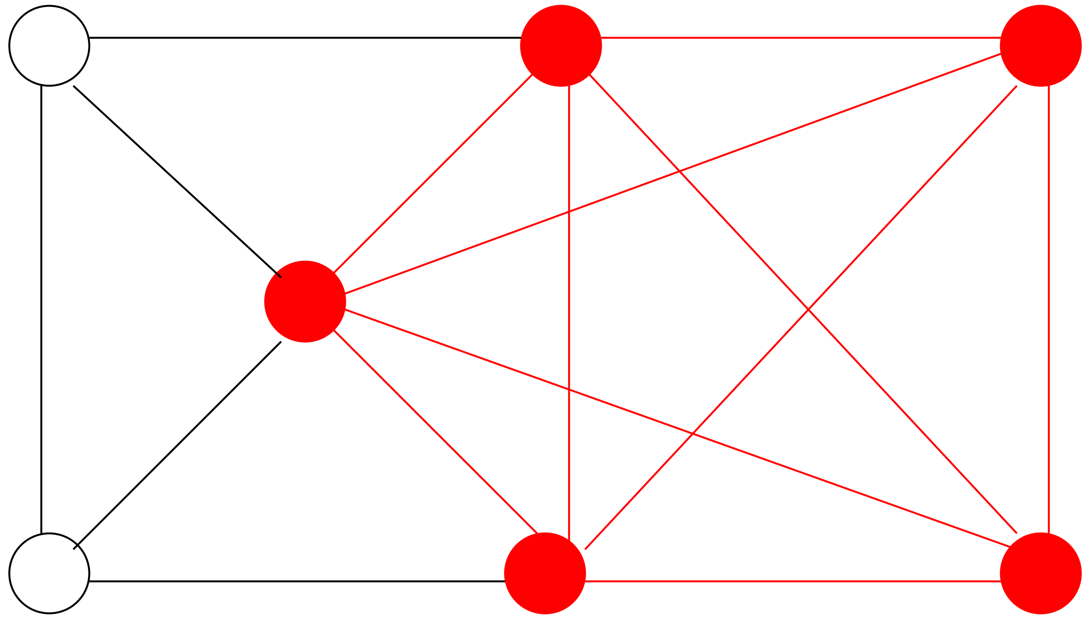
$$\underbrace{(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6)}_{\text{clause}} \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6)$$

Each clause has three literals

Language:

3CNF-SAT $= \{ w : w$ is a satisfiable 3CNF formula$\}$

A 5-clique in graph G



Language:

$$\text{CLIQUE} = \{ \langle G, k \rangle : \text{ graph } G \text{ contains a } k\text{-clique} \}$$

Theorem: 3CNF-SAT is polynomial time reducible to CLIQUE

- Proof:It takes a Boolean expression as an input and converts it into an equivalent graph .
- Assume that the 3-CNF expression has *k* clauses.
- We then construct a graph such that if the 3-CNF expression is satisfiable, then there will be a clique on the graph of size *k*.

Step 1: Add vertices to the graph

- For each clause, add a vertices in the graph for each literal or negated literal (since each clause has exactly 3 literals there will be 3 vertices per clause in the graph giving a total of 3*k* vertices in the final graph).

Step 2: Add edges between vertices

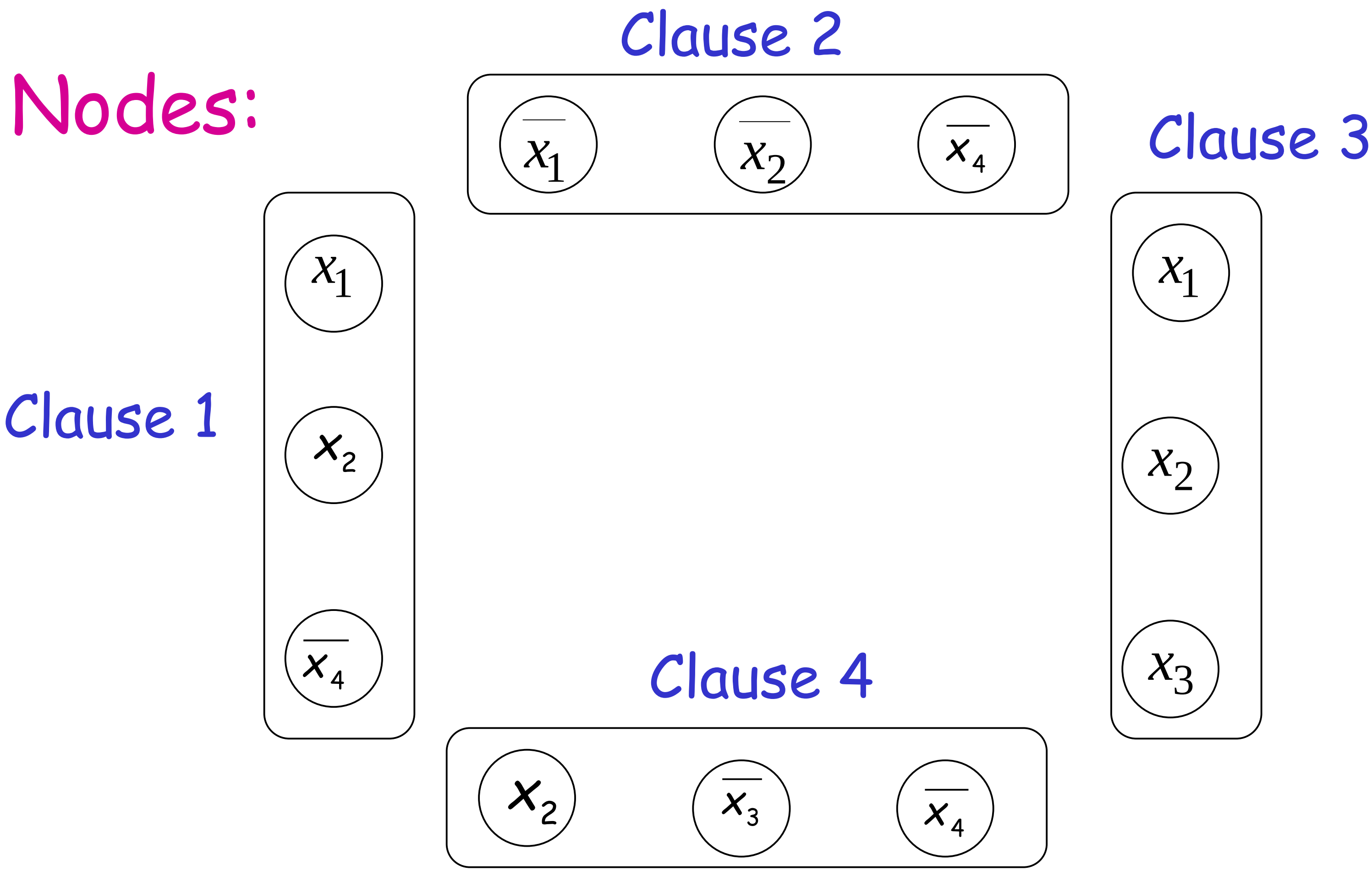
- Add edges between vertices under the following conditions
- The two vertices come from different clauses
- The vertices are *not* negations of each other

Transform formula to graph.

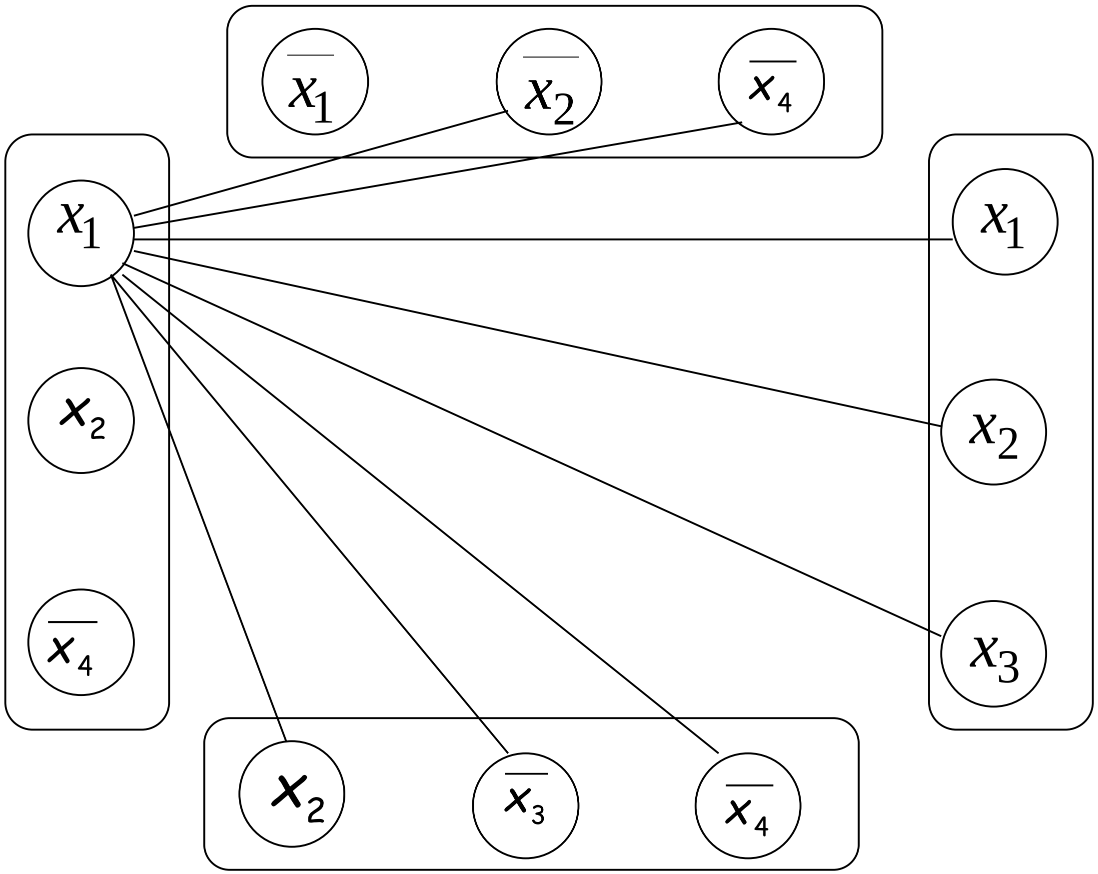
Example:

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

Create Nodes:

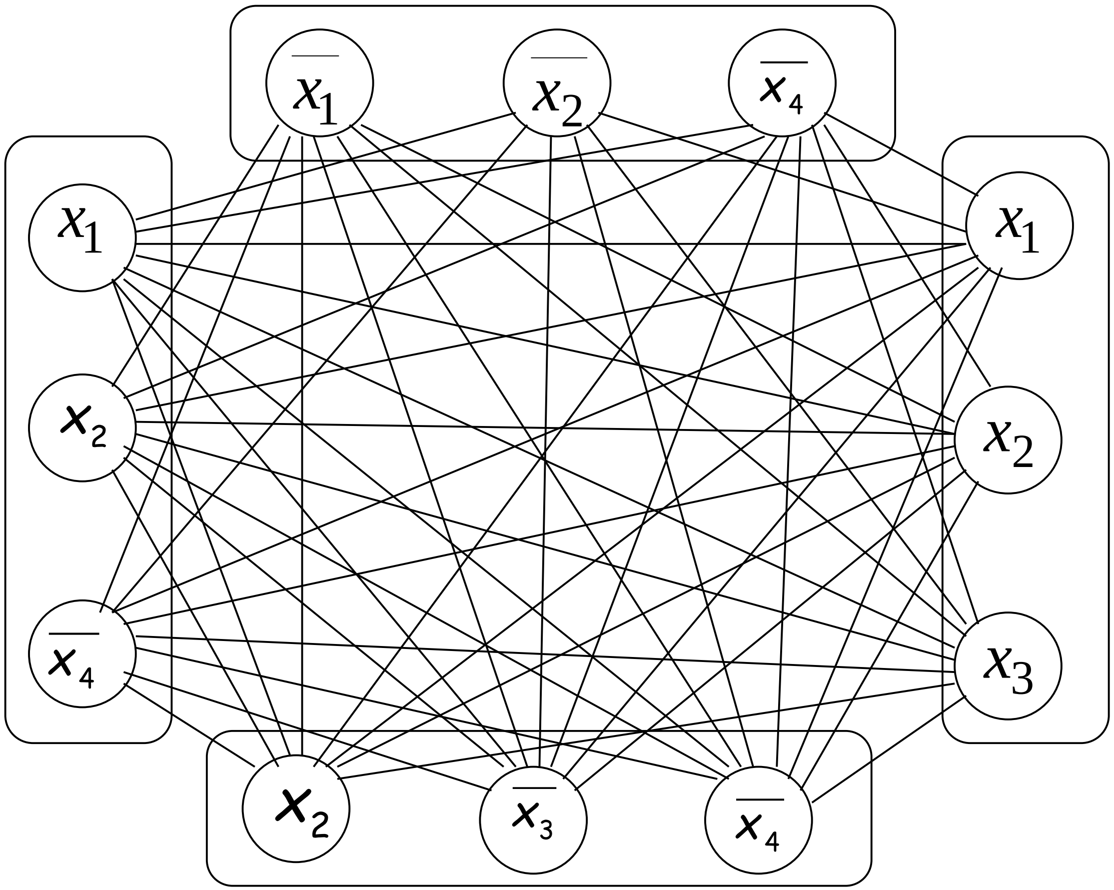


$$(x_1 \vee x_2 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4})$$



Add link from a literal x to a literal in every other clause, except the complement $\overline{x}$

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



Resulting Graph

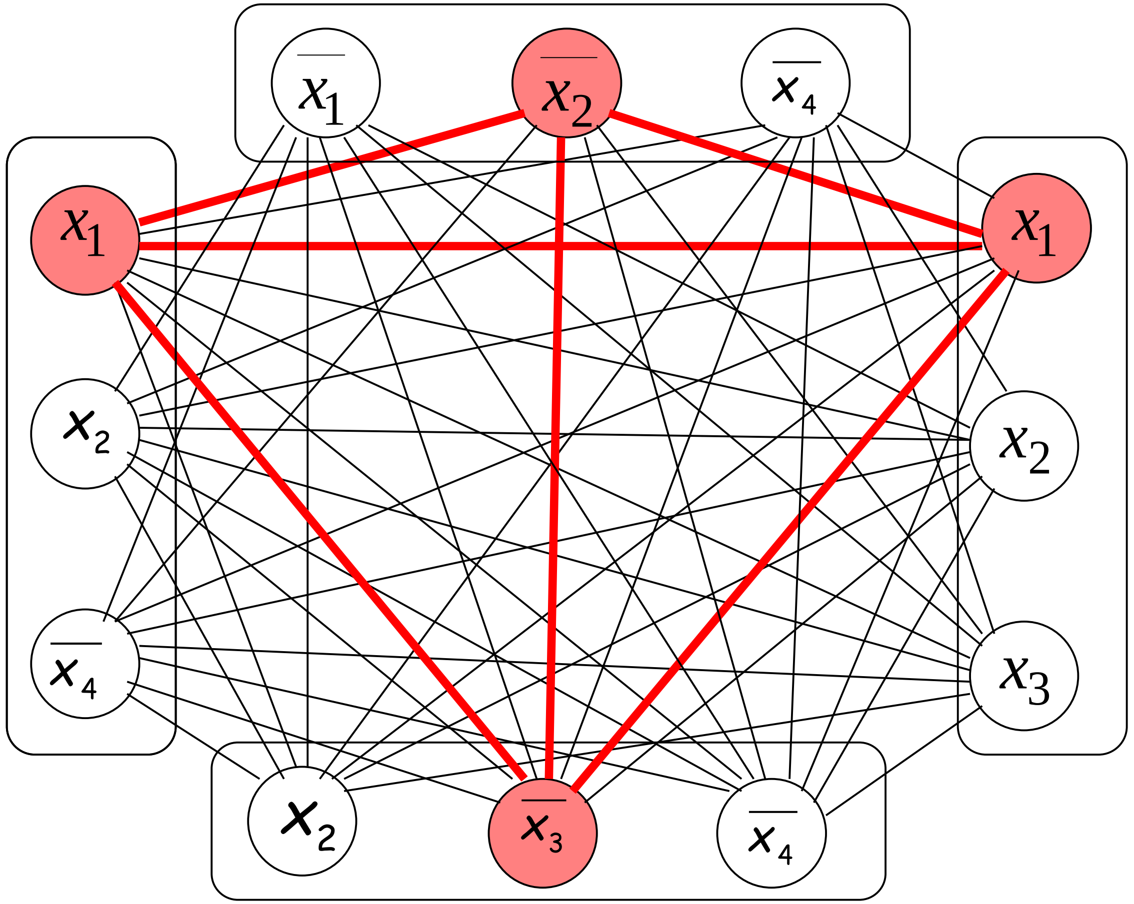
$$(x_1 \vee x_2 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4}) = 1$$

$x_1 = 1$

$x_2 = 0$

$x_3 = 0$

$x_4 = 1$



The formula is satisfied if and only if the Graph has a 4-clique

Now let's consider the how long it takes,

beginning with the string *w* representing *3CNF* to construct the string <*G*, *K*> representing a k-Clique graph.

The vertices of the graph can be constructed in a single scan of *w*.

For a particular literal in a particular clause of the formula, a new edge is obtained for each literal in another clause that is not the negation of the first one.

The activities:

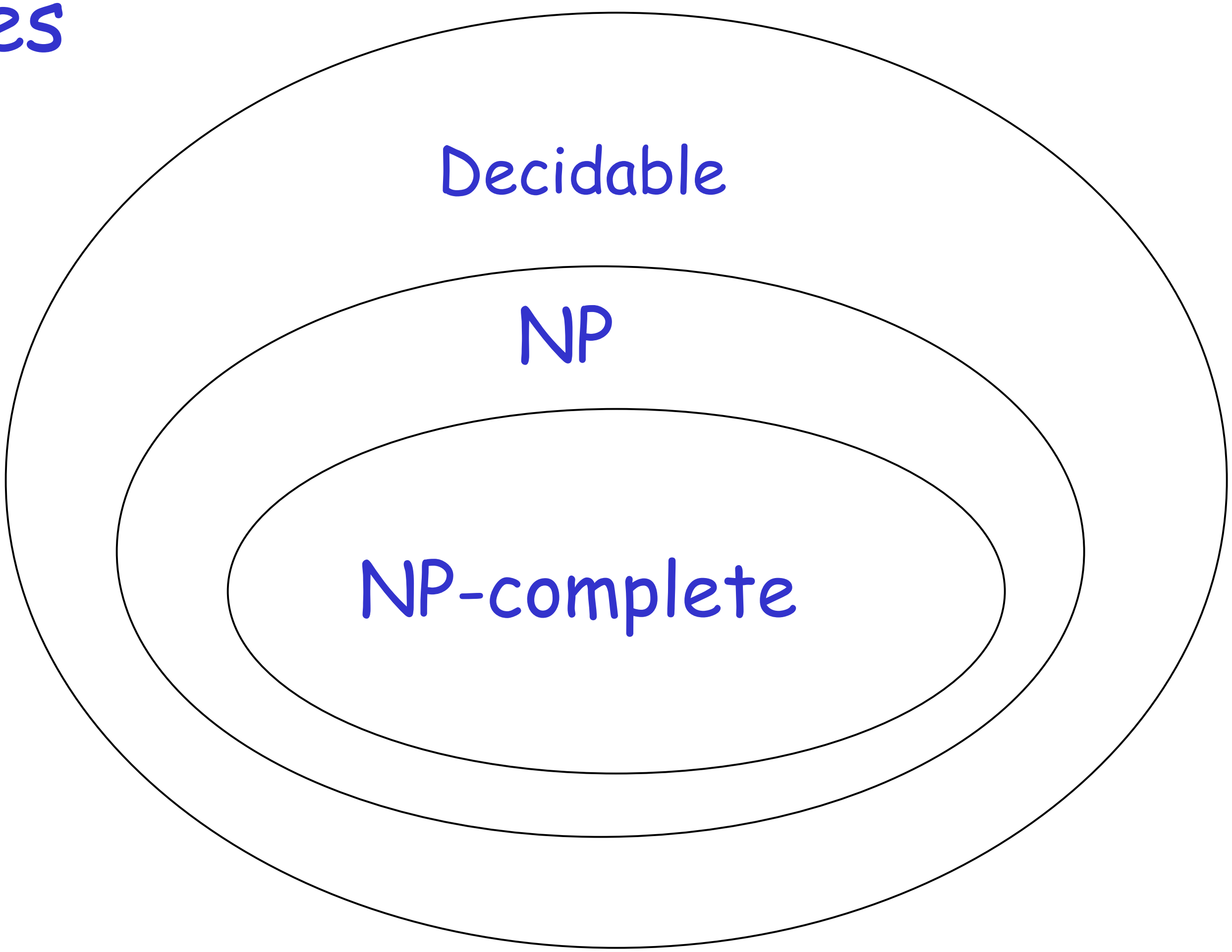
- Finding another clause,
- Identifying another literal within that clause, and
- Comparing that literal to the original one can be done within polynomial time.

It follows that the overall time is polynomial.

End of Proof

NP-complete Languages

We define the class of NP-complete languages



A language L is NP-complete if:

L is np and np-hard

- L is in NP, and

- Every language in NP is reduced to L in polynomial time (NP-HARD)

Theorem:

Suppose **Y** is an NP-complete problem. Then **Y** is solvable in poly-time ifand only if **P = NP**.

Proof:

$\Leftarrow$ If P = NP then Y can be solved in poly-time since Y is in NP.

$\Rightarrow$ Suppose Y can be solved in poly-time.

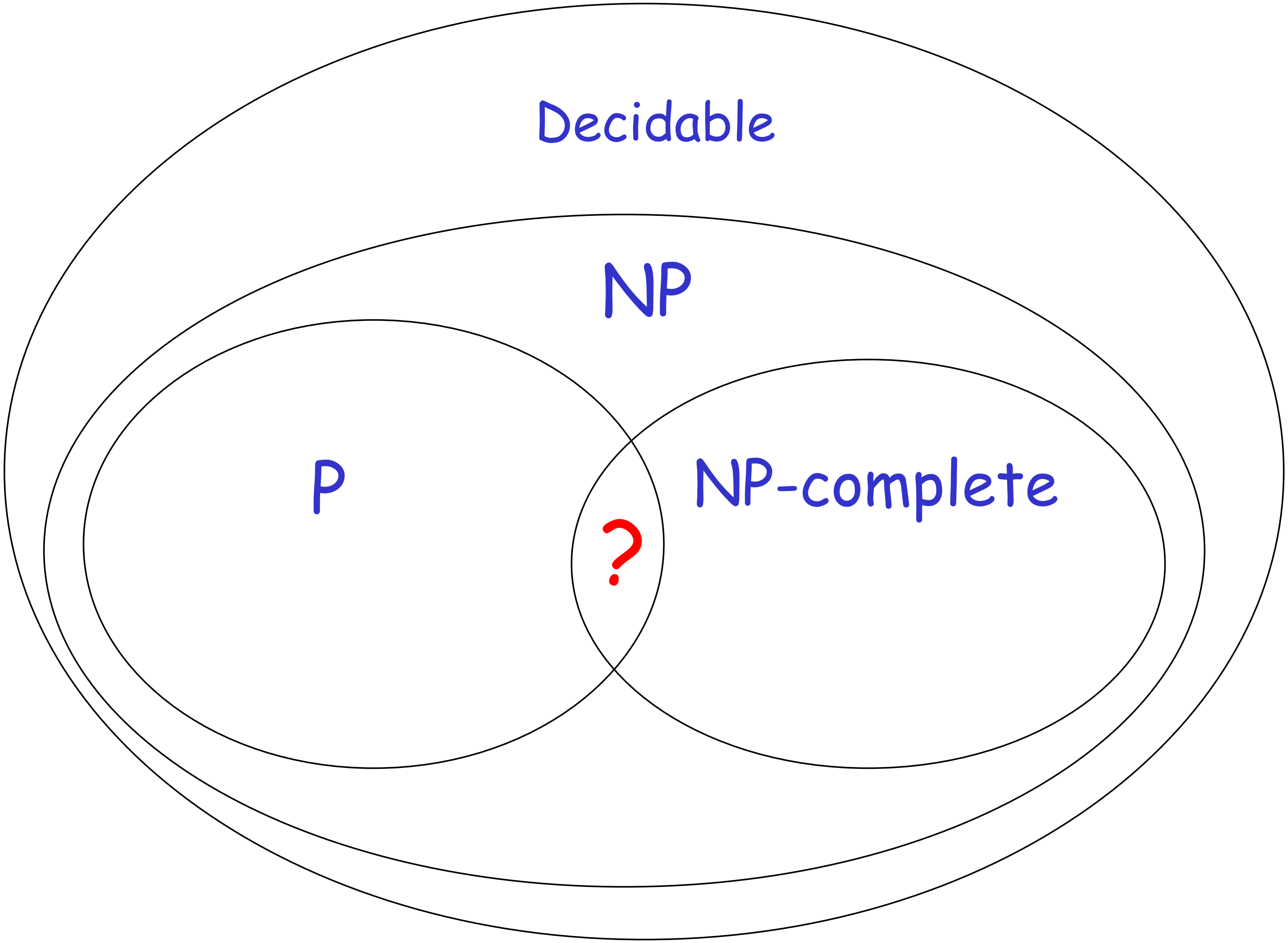
Let X be any problem in NP. Since $X \leq_p Y$, we can solve X in poly-time. This implies $NP \subseteq P$.

We already know $P \subseteq NP$. Thus P = NP.

Observation:

If a NP-complete language
is proven to be in P then:

$$P = NP$$



Fundamental question. Do there exist "natural" NP-complete problems?

An NP-complete Language

Cook-Levin Theorem:

Language SAT (satisfiability problem)
is NP-complete

Proof:

Part1: SAT is in NP

(we have shown this in previous class)

Part2: reduce all NP languages
to the SAT problem
in polynomial time(NP-HARD)

Proof idea

- The hard part of the proof is showing that any language in NP is polynomial time reducible to SAT.
- To do so, we construct a polynomial time reduction for each language *A* in NP to SAT.
- The reduction for *A* takes a string *w* and produces a Boolean formula φ that simulates the NP machine for *A* on input *w*

Take an arbitrary language $A \in NP$

We will give a polynomial reduction of A to SAT

Let M be the NonDeterministic Turing Machine that decides A in polyn. time

For any string w we will construct in polynomial time a Boolean expression $\varphi(M,w)$

such that: $w \in A \Leftrightarrow \varphi(M,w)$ is satisfiable

- If the machine accepts, φ has a satisfying assignment that corresponds to the accepting computation.
- If the machine doesn't accept, no assignment satisfies φ.
- Therefore, *w* is in *A* if and only if φ is satisfiable.