Complexity Classes - P & NP

Wei Wang @ HKUST(GZ)

April 28, 2025

Introduction

The course so far: techniques for designing efficient algorithms, e.g., divide-and-conquer, dynamic-programming, greedy-algorithms.

What happens if you can't find an efficient algorithm? Is it your "fault" or the problem's?

Showing that a problem has an efficient algorithm is, relatively, easy. "All" that is needed is to demonstrate an algorithm.

Proving that no efficient algorithm exists for a partic-ular problem is difficult. How can we prove the non-existence of something?

We will now learn about **NP Complete Problems**, which provide us with a way to approach this question.

NP-Complete (NPC) Problems

A huge class of thousands of practical problems for which

- Unknown if the problems have "efficient" solutions
  - despite man-years efforts and failing
  - known as $P \neq NP$? with a US$1,000,000 award offered by the Clay Institute (`http://www.claymath.org/`).
- known:
  - if **any one** of the NP-Complete Problems has an efficient solution then **all** of the NP-Complete Problems have efficient solutions
  - there is a large body of tools that often permit us to prove when a new problem is NP-complete.

Outline

- notation and terminology needed to properly discuss NP-Complete problems
- tools required to prove that problems are NP-complete.
  - Note: Proving that a problem is NP-Complete does not prove that the problem is hard. It does indicate that the problem is **very likely** to be hard.

Contents of this Lecture

In this lecture we introduce the concepts that will permit us to discuss whether a problem is 'hard' or 'easy'.

- **Input size of problems.**: the number of bits required to encode the problem.
- **Optimization problems vs. decision problems.**
  - **Decision Problems**: have Yes/No answers.
  - **Optimization Problems** require answers that are optimal configurations.
  - Decision problems are **no harder** than the corresponding optimization problems:
    - Found a method to solve the optimization problem $\Longrightarrow$ induces a method to solve the decision problem.
    - But not the other way around

Outline

- **Polynomial time algorithms. The Class P.**
- **The Class NP.**
- **Problems in the two classes.**
- **the class co-NP.**

Encoding the Inputs of Problems

Need to be formal about the **input size** of a problem.

Example

How do we encode graphs?

A graph G may be represented by its adjacency matrix $A = [a_{ij}]_{i,j \in [n]}$. G can then be encoded as the binary string of length n^2,

$$a_{11} \ldots a_{1n} a_{21} \ldots a_{2n} \ldots a_{n1} \ldots a_{nn}$$

Remark: the inputs of any problem can be encoded as binary strings.

The Input Size of Problems

The input size of a problem may be defined in a number of ways.

Definition

(Standard Definition) The input size of a problem s is the minimum number of bits ($\{0, 1\}$) needed to encode the input of the problem.

Note:

- The exact input size s (i.e., the "minimum" part), is hard to compute in most cases.
- Nevertheless, for most problems, it is sufficient to choose some natural and (usually) simple, encoding and use the size s of this encoding.

Input Size Example: Composite

Problem: Given a positive integer n, are there integers $j, k > 1$ such that $n = j \cdot k$? (i.e., is n a **composite number**?)

Question

What is the input size of this problem?

Answer: Any integer $n > 0$ can be represented in the binary number system as $n = \sum_{i=0}^{k} a_i 2^i$, where $k = \lceil \log_2(n+1) \rceil - 1$ and so be **represented** by the string $a_0 a_1 \ldots a_k$ of length $\lceil \log_2(n+1) \rceil$, and hence a natural measure of input size (or just $\log_2 n$)

$$\begin{array}{ccc} Obj & \xrightarrow{?} & size(Obj) \\ \downarrow h & & \uparrow = \\ z & \xrightarrow{g} & val \end{array}$$

Input Size Example: Sorting

Sorting Problem: Sort n integers $a_1, \ldots, a_n$.

Question

What is the input size of this problem?

Solution: Using fixed length encoding writes a_i as binary string of length

$$m = \lceil \log_2 \max(|a_i| + 1) \rceil$$

This coding gives input size nm.

Warning

Running times of algorithms, unless otherwise specified, should be expressed in terms of **input size**.

For example, the naive **sieve** algorithm for determining whether n is composite compares n against the first $n-1$ numbers to see if any of them divides n. This makes $\Theta(n)$ comparisons so it might seem **linear** and very efficient.

But, note that the size of the problem is $size(n) = \log_2 n$ so the number of comparisons performed is actually

$$\Theta(n) = \Theta\left(2^{size(n)}\right)$$

which is **exponential** and not very good.

Input Size of Problems

Definition

Two positive functions $f(n)$ and $g(n)$ are of the **same type** if

$$c_1 g(n^{a_1})^{b_1} \leq f(n) \leq c_2 g(n^{a_2})^{b_2}$$

for all large n, where a_1, b_1, c_1, a_2, b_2, c_2 are some positive constants.

For example, all polynomials are of the same type, but polynomials and exponentials are of different types.

Suppose s is the actual input size in bits needed to encode the problem. From this point of view, any quantity t, **satisfying**

$$s^{a_1} \leq t \leq s^{a_2}$$

for some positive constants a_1 and a_2 (independent of s), **may also be used as a measure of the input size of a problem**. This will simplify our discussions.

Input Size Example: Graphs Redux

Graph problems: For many graph problems, the input is a graph $G = (V, E)$.

Question

What is the input size of this problem?

A natural choice: There are n vertices and e edges. So we need to encode $n + e$ objects. With fixed length coding, the input size is

$$(n + e)\lceil \log_2(n + e + 1) \rceil$$

Since

$$[(n + e)\lceil \log_2(n + e + 1) \rceil]^{1/2} < n + e < (n + e)\lceil \log_2(n + e + 1) \rceil$$

we may use $n + e$ as the input size.

Input Size Example: Integer Multiplication

Integer multiplication problem: Compute $a \times b$.

Question

What is the input size of this problem?

Solution: The (minimum) input size is

$$s = \lceil \log_2(a+1) \rceil + \lceil \log_2(b+1) \rceil$$

A natural choice is to use

$$t = \log_2 \max(a, b)$$

as the input size since

$$\frac{s}{2} \leq t \leq s$$

Decision Problems

Definition

A **decision problem** is a question that has two possible answers, **yes** and **no**.

Note: If L is the problem and x is the input we will often write $x \in L$ to denote a yes answer and $y \notin L$ to denote a no answer.

Note: This notation comes from thinking of L as a **language** and asking whether x is in the language L (yes) or not (no).

Definition

An **optimization problem** requires an answer that is an optimal configuration.

Decision Problems

Remark: An optimization problem usually has a corresponding decision problem.

Examples that we will see:

- **MST** vs. **Decision Spanning Tree (DST)**
- **Knapsack** vs. **Decision Knapsack (DKnapsack)**
- **SubSet Sum** vs. **Decision Subset Sum (DSubset Sum)**

Decision Problem: MST

Optimization problem: Minimum Spanning Tree

Given a weighted graph G, find a minimum spanning tree (MST) of G.

Decision problem: Decision Spanning Tree (DST)

Given a weighted graph G and an integer k, does G have a spanning tree of weight at most k?

The inputs are of the form (G, k). So we will write $(G, k) \in DST$ or $(G, k) \notin DST$ to denote, respectively, yes and no answers.

Decision Problem: Knapsack

We have a knapsack of capacity W (a positive integer) and n objects with weights $w_1, ..., w_n$ and values $v_1, ..., v_n$, where v_i and w_i are positive integers.

Optimization problem: Knapsack

Find the largest value $\sum_{i \in T} v_i$ of any subset that fits in the knapsack, that is, $\sum_{i \in T} w_i \leq W$.

Decision problem: Decision Knapsack (DKnapsack)

Given k, is there a subset of the objects that fits in the knapsack and has total value at least k?

Decision Problem: Subset Sum

The input is a positive integer C and n objects whose sizes are positive integers $s_1, \ldots, s_n$.

Optimization problem: Subset Sum

Among subsets of the objects with sum at most C, what is the largest subset sum?

Decision problem: Decision Subset Sum (DSubset Sum)

Is there a subset of objects whose sizes add up to **exactly** C?

Optimization and Decision Problems

- For almost all optimization problems there exists a corresponding **simpler** decision problem.
- Given a subroutine for solving the optimization problem, solving the corresponding decision problem is usually trivial.

  Example: If we know how to solve **MST** we can solve **DST**.
  Solution:
  1. $y =$ solve the MST problem
  2. if $y \leq k$, answer Yes; else, answer No.
- Thus if we prove that a given decision problem is hard to solve efficiently, then it is obvious that the optimization problem must be (at least as) hard.

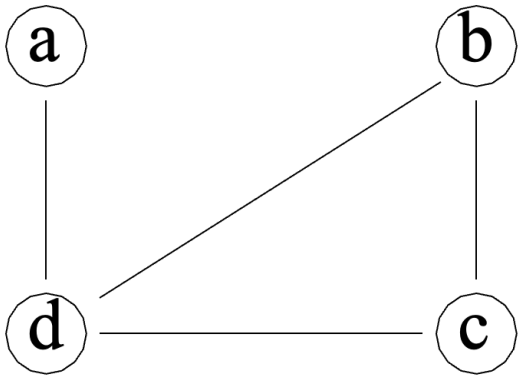
Note: The reason for introducing Decision problems is that it will be more convenient to compare the 'hardness' of decision problems than of optimization problems (since all decision problems share the same form of output, either **yes** or **no**.)

Decision Problems: Yes-Inputs and No-Inputs

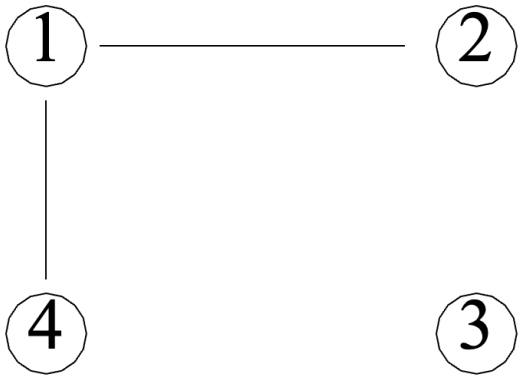
Definition

Yes-Input and No-Input: An instance of a decision problem is called a yes-input (resp. no-input) if the answer to the instance is yes (resp. no).

CYC Problem: Does an undirected graph G have a cycle?
Example of Yes-Inputs and No-Inputs:



Yes-input G



No-input G

Decision Problems: Yes-Inputs and No-Inputs

Decision Problem (TRIPLE): Does a triple (n, e, t) of nonnegative integers satisfy $e \neq n - t$?

Example of **Yes-Inputs**: $(9, 8, 2)$, $(20, 2, 17)$.

Example of **No-Inputs**: $(10, 8, 2)$, $(20, 2, 18)$.

Complementary Problems

Let L denote some decision problem. The **complementary problem** $\overline{L}$ is the decision problem such that the yes-answers of $\overline{L}$ are exactly the no-answers of L. Note that

$$\overline{\overline{L}} = L$$

Example:

- COMPOSITE: is given positive integer n composite?
- PRIMES: is given positive integer n a prime number?

$$\overline{\text{COMPOSITE}} = \text{PRIMES}$$
$$\overline{\text{PRIMES}} = \text{COMPOSITE}$$

Complexity Classes

The Theory of Complexity deals with

- the classification of certain **"decision problems"** into several classes:
  - the class of "easy" problems,
  - the class of "hard" problems,
  - the class of "hardest" problems;
- relations among the three classes;
- properties of problems in the three classes.

Question: How to classify decision problems?

Answer: Use **"polynomial-time algorithms."**

Polynomial-Time Algorithms

Definition

An algorithm is **polynomial-time** if its running time is $O(n^k)$, where k is a constant independent of n, and n is the **input size** of the problem that the algorithm solves.

Remark

When deciding whether an algorithm is polynomial time, it does not whether an algorithm is polynomial time.

This explains why we introduced the concept of two functions being of the **same type** earlier on. Using the definition of polynomial-time it is not necessary to fixate on the input size as being the exact minimum number of bits needed to encode the input!

Polynomial-Time Algorithms

Examples of Polynomial-Time Algorithms:

- The standard multiplication algorithm learned in school has time $O(m_1 m_2)$ where m_1 and m_2 are, respectively, the number of digits in the two integers.
- DFS has time $O(n + e)$.
- Kruskal's MST algorithm runs in time $O((e + n) \log n)$.

Non-polynomial-Time Algorithms

Definition

An algorithm is **non-polynomial-time** if the running time is not $O(n^k)$ for any fixed $k \geq 0$.

Example

Consider the brute force algorithm fo PRIME:

- it checks, in time $\Theta((\log N)^2)$, whether K divides N for each K with $2 \leq K \leq N - 1$.

The algorithm uses $\Theta(N(\log N)^2)$ time $\Rightarrow$ The algorithm is nonpolynomial! **Why?**

Non-polynomial-Time Algorithms

Definition

An algorithm is **non-polynomial-time** if the running time is not $O(n^k)$ for any fixed $k \geq 0$.

Example

Consider the brute force algorithm fo PRIME:

- it checks, in time $\Theta((\log N)^2)$, whether K divides N for each K with $2 \leq K \leq N-1$.

The algorithm uses $\Theta(N(\log N)^2)$ time $\Rightarrow$ The algorithm is nonpolynomial! **Why?**
The input size is $n = \log_2 N$, and so $\Theta(N(\log N)^2) = \Theta(2^n n^2)$.

Is Knapsack Polynomial?

Consider the knapsack problem with capacity W and n objects with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ (all parameters are positive integers).

- The optimization problem is to find the largest value $\sum_{i \in T} v_i$ of any subset that fits in the knapsack, that is, $\sum_{i \in T} w_i \leq W$.
- The decision problem is, given k, to find if there is a subset of the objects that fits in the knapsack and has total value at least k?

Question

In class, we saw a $\Theta(nW)$ dynamic programming algorithm for solving the optimization version of Knapsack. Is this a polynomial algorithm?

Is Knapsack Polynomial?

Consider the knapsack problem with capacity W and n objects with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ (all parameters are positive integers).

- The optimization problem is to find the largest value $\sum_{i \in T} v_i$ of any subset that fits in the knapsack, that is, $\sum_{i \in T} w_i \leq W$.
- The decision problem is, given k, to find if there is a subset of the objects that fits in the knapsack and has total value at least k?

Question

In class, we saw a $\Theta(nW)$ dynamic programming algorithm for solving the optimization version of Knapsack. Is this a polynomial algorithm?

No! The size of the input is

$$\text{size}(I) = \log_2 W + \sum_i \log_2 w_i + \sum_i \log_2 v_i$$

nW is not polynomial in size(I).

Is Knapsack Polynomial?

Further comments:

- Depending upon the values of the w_i and v_i, nW could even be **exponential** in size(I).
- It is unknown as to whether there exists a polynomial time algorithm for Knapsack.
  - It is an **NP-Complete problem**, requiring us to prove either P = NP or P $\neq$ NP.

Polynomial- vs. Nonpolynomial-Time

- Non-polynomial-time algorithms are **impractical**.

  e.g., if an algorithm has 2^n complexity. When $n = 100$, and one can execute 10^{12} operations per second: It takes $2^{100}/10^{12} \approx 10^{18.1}$ seconds $\approx 4 \cdot 10^{10}$ years.

- For the sake of our discussion of complexity classes Polynomial-time algorithms are **"practical"**.

  Note: in reality

  - an $O(n^{20})$ algorithm is not really practical.
  - even an $O(n^2)$ algorithm can be impractical for big data.

Polynomial-Time Solvable Problems

Definition

A problem is **solvable in polynomial time** (or, the problem is **in polynomial time**) if there exists an algorithm which solves the problem in polynomial time.

Examples: The integer multiplication problem, and the cycle detection problem for undirected graphs.

Remark: Polynomial-time solvable problems are also called **tractable** problems.

The Class P

Definition

The class **P** consists of all **decision problems** that are solvable in polynomial time. That is, there exists an algorithm that will decide in polynomial time if any given input is a yes-input or a no-input.

Question

How to prove that a decision problem is in P?

You need to find a polynomial-time algorithm for this problem.

Question

How to prove that a decision problem is not in P?

You need to prove there is no polynomial-time algorithm for this problem (**much harder**).

The Class P: An Example

Example

Is a given connected graph G a tree?

This problem is in P.

Proof.

We need to show that this problem is solvable in polynomial time. We run DFS on G for cycle detection. If a back edge is seen, then output NO, and stop. Otherwise, output YES.

Recall that the input size is $n + e$, and DFS has running time $O(n + e)$. So this algorithm is linear, and the problem is in P. □

The Class P: Another Example

Question

DST: Given weighted graph G and parameter $k > 0$ does G have a spanning tree with weight $\leq k$?

This problem is in P.

The Class P: Another Example

Question

DST: Given weighted graph G and parameter $k > 0$ does G have a spanning tree with weight $\leq k$?

This problem is in P.

Proof.

Run Kruskal's algorithm and find a **minimal spanning tree**, T, of G. Calculate $w(T)$ the weight of T. If $k \leq w(T)$, answer Yes; otherwise, answer No.

Recall that Kruskal's algorithm runs in $O((e+n)\log n)$ time so this is polynomial in the size of the input. □

Certificates and Verifying Certificates

What about class NP? Before introducing NP, we **must** first introduce the concept of **Certificates**.

Observation: A decision problem is usually formulated as:

Is there an object satisfying some conditions?

A **Certificate** is a specific object satisfying the conditions (exists **only** for **yes-inputs** by definition).

Verifying a certificate: Check that the given object (certificate) satisfies the conditions (that is, verifying that the input is a yes-input).

COMPOSITE: Is given positive integer n composite?

Certificate: an integer a dividing n such that $1 < a < n$.

Verifying a certificate: Given a certificate a, check whether a divides n. This can be done in time $O((\log_2 n)^2)$ (recall that input size is $\log_2 n$ so this is polynomial in input size).

Certificates and Verifying Certificates Example

DSubsetSum: Input is a positive integer C and n positive integers $s_1, ..., s_n$. Is there a subset of these integers that add up to **exactly** C?

Certificate: a subset T of subscripts (the corresponding integers should add up to C).

Verifying a certificate: Given a subset T of subscripts, check whether $\sum_{i \in T} s_i = C$

Input-size is $m = (\log_2 C + \sum_{i=1}^{n} \log_2 s_i)$

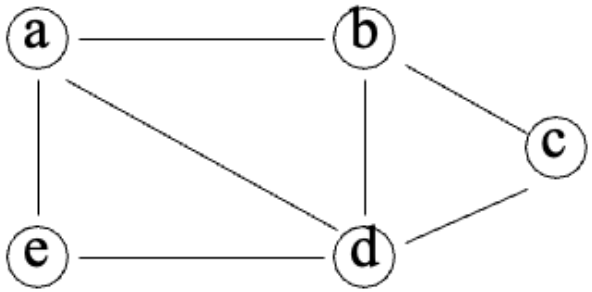
and verification can be done in time

$$O\left(\log_2 C + \sum_{i \in T} \log_2 s_i\right) = O(m)$$

so this is polynomial time.

Hamiltonian Cycle: Input is a graph $G = (V, E)$. A cycle of graph G is called **Hamiltonian** if it contains every vertex exactly once.

Example:



Find a Hamiltonian cycle for this graph

Optimization problem: HamCyc

Find a Hamiltonian cycle for this graph or say that one doesn't exist.

Decision problem: DHamCyc

Does G have a Hamiltonian cycle?

DHamCyc: Verifying a Certificate

Certificate: an ordering of the n vertices in G (corresponding to their order along the Hamiltonian Cycle), i.e., $v_{i_1}, v_{i_2}, \ldots, v_{i_n}$.

Verification: Given a certificate the verification algorithm checks whether it is a Hamiltonian cycle of G by simply checking whether all of the edges

$$(v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \ldots, (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1})$$

appear in the graph. This can be done in $O(n)$ time so this is polynomial.

Important Note: There are many possible types of certificates for DHamCyc. This is only one such. Another type of **certificates** might be a set of n edges for which it has to be confirmed that they are all in G and that they form a Hamiltonian Cycle.

The Class NP

Definition

The class **NP** consists of all decision problems such that, for each yes-input, there exists a **certificate** that can be **verified** in polynomial time.

Example: DSubsetSum $\in$ **NP**. (As shown earlier, there is a polynomial-time algorithm to verify a certificate.)

Example: DHamCyc $\in$ **NP**. (As shown earlier, there is a polynomial time algorithm to verify a certificate.)

Remark: NP stands for **"nondeterministic polynomial time"**. The class NP was originally studied in the context of nondeterminism, here we use an equivalent notion of verification.

P = NP?

One of the most important problems in computer sci- ence is whether $P = NP$ or $P \neq NP$? Put another way, is every problem that can be **verified** in polynomial time also **decidable** in polynomial time? Or, does there exist some decision problem L which has certificates that can be verified in polynomial time but for which no algorithm can ever be constructed that decides L in polynomial time?

At first glance, it seems "obvious" that $P \neq NP$; after all, **deciding** a problem is much more restrictive than **verifying** a certificate.

However, so far, we are still no closer to solving it and do not know the answer. The search for a solution, though, has provided us with deep insights into what distinguishes an "easy" problem from a "hard" one.

We will now introduce **Satisfiability (SAT)**, which, we will see later, is one of the most important NP problems.

Definition

A **Boolean formula** is a logical formula which consists of

- **boolean variables** (0=false, 1=true),
- **logical operations**
  - $\overline{x}$, **NOT**,
  - $x \vee y$, **OR**,
  - $x \wedge y$, **AND**.

These are defined by:

x	y	$\overline{x}$	$x \vee y$	$x \wedge y$
0	0	1	0	0
0	1		1	0
1	0	0	1	0
1	1		1	1

Satisfiability II

A given Boolean formula is **satisfiable** if there is a way to assign truth values (0 or 1) to the variables such that the final result is 1.

Example: $f(x, y, z) = (x \wedge (y \vee \overline{z})) \vee (\overline{y} \wedge z \wedge \overline{x})$.
The assignment $x = 1$, $y = 1$, $z = 0$ makes $f(x, y, z)$ true, and hence it is satisfiable.
In fact, more than one satisfiable assignment.

x	y	z	$(x \wedge (y \vee \overline{z}))$	$(\overline{y} \wedge z \wedge \overline{x})$	$f(x, y, z)$
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	1	0	1

Satisfiability III

Example:
$f(x, y) = (x \vee y) \wedge (\overline{x} \vee y) \wedge (x \vee \overline{y}) \wedge (\overline{x} \vee \overline{y})$.

x	y	$x \vee y$	$\overline{x} \vee y$	$x \vee \overline{y}$	$\overline{x} \vee \overline{y}$	$f(x, y)$
0	0	0	1	1	1	0
0	1	1	1	0	1	0
1	0	1	0	1	1	0
1	1	1	1	1	0	0

Therefore, there is no assignment that makes $f(x, y)$ true, and hence it is NOT satisfiable.

SAT $\in$ NP

Definition

SAT Problem: Determine whether an input Boolean formula is satisfiable.

Claim

SAT $\in$ NP.

Proof.

The evaluation of a formula of length n (counting variables, operations, and parentheses) requires at most n evaluations, each taking constant time. Hence, to check a certificate takes time $O(n)$. □

Definition

For a fixed k, consider Boolean formulas in k-**conjunctive normal form** (k-CNF):

$$f_1 \wedge f_2 \wedge \cdots \wedge f_n$$

where each f_i is of the form

$$f_i = y_{i,1} \vee y_{i,2} \vee \cdots \vee y_{i,k}$$

where each $y_{i,j}$ is a variable or the negation of a variable.

An example of a 3-CNF formula is

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

k-SAT $\in$ NP

Definition

k-SAT problem: Determine whether an input Boolean k-CNF formula is satisfiable.

Non-trivial results:

Claim

3-SAT $\in$ NP.

Claim

2-SAT $\in$ P

2-SAT $\in$ P (Optional)

We provide a linear-time algorithm to decide whether a 2-SAT (with m clauses involving n variables) is satisfiable or not.

1. (Transformation) Notice that every clause in a 2-SAT is of the form $x_i \vee x_j$, which is equivalent to two implications: $\neg x_i \implies x_j$ and $\neg x_j \implies x_i$
2. (Graph construction) Construct the implication graph G with $2n$ vertices ($\{ x_i \}$ and $\{ \neg x_i \}$, and $\leq 2m$ edges.
3. (Lemma) A 2-CNF formula is unsatisfiable iff $\exists x_i$ such that x_i and $\neg x_i$ belong to the same SCC of G.
   - $\Rightarrow$: Notice the transitivity of implication; contradiction is $x_i \implies \neq x_i$.
   - $\Leftarrow$: Require topological sort on the condensed graph (modulo SCC). (details omitted)
4. (Algorithm)
   1. Run Tarjan's algorithm to compute SCCs of G.
   2. For every variable x, if $SCC(x) = SCC(\neg x)$, then return **UNSAT**; else return **SAT**;
   3. (optional) Produce an assignment if **SAT**.

Some Decision Problems in NP

Some where we have given proofs:
Decision subset sum problem (DSubsetSum).
Decision Hamiltonian cycle (DHamCyc).
Satisfiability (SAT).
Decision vertex cover problem (DVC).

Some others (without proofs given; try to find proofs):
Decision minimum spanning tree problem (DMST).
Decision 0-1 knapsack problem (DKnapsack).
Decision bin packing problem (DBinPacking)

The Class co-NP

Note that if $L \in \mathbf{NP}$, there is no guarantee that $\overline{L} \in \mathbf{NP}$.

- Having certificates for yes-inputs, does not mean that we have certificates for the no-inputs.

The class of decision problems L such that $\overline{L} \in \mathbf{NP}$ is called **co-NP**.

Example: COMPOSITE $\in$ NP and so
PRIMES $= \overline{\text{COMPOSITE}} \in$ co-NP.

Remark: in contrast, we do have that $L \in \mathbf{P}$, if and only if $\overline{L} \in \mathbf{P}$.

This is because a polynomial time algorithm for L is also a polynomial time algorithm for $\overline{L}$ (the NO-answers for L become Yes-answers for $\overline{L}$ and vice-versa).