INT202
Complexity of Algorithms

NP-Completeness

Year 2020-2021

Complexity class NP (non-deterministic polynomial time)

The class NP consists of all decision problems for which there exists an *efficient certifier*.

An *efficient certifier* for a decision problem X is an algorithm B that takes two input strings s and t . The string s is the input to the decision problem.

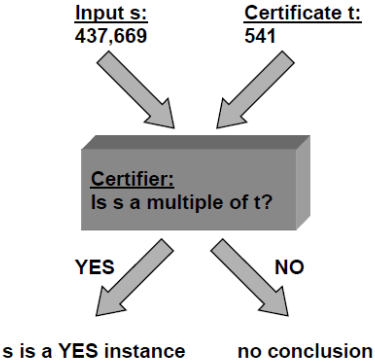
“Efficient” means that B is a polynomial-time algorithm, i.e. there is a polynomial function $q(\cdot)$ so that for every string s, we have $s \in L(X)$ if and only if there exists a string t such that $|t| \leq q(|s|)$ and $B(s, t)$ = “yes”.

We can think of the string t as being a “proof” that the answer to the decision problem is “yes” for the input string s.

Certifiers and Certificates

COMPOSITE: Given integer s, is s composite?
Observation. s is composite iff there exists an integer 1 < t < s such that s is a multiple of t.

YES instance: s = 437,669.
– with certificate t = 541 or 809 (a factor)

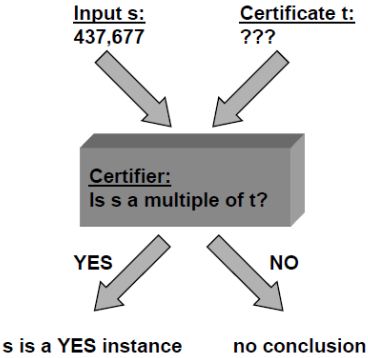


Certifiers and Certificates

COMPOSITE: Given integer s, is s composite?
Observation. s is composite iff there exists an integer 1 < t < s such that s is a multiple of t.

YES instance: s = 437,669.
– with certificate t = 541 or 809 (a factor)

NO instance: s = 437,677.
– no witness can fool verifier into saying YES



Input s:
437,677

Certificate t:
???

Certifier:
Is s a multiple of t?

YES

NO

s is a YES instance

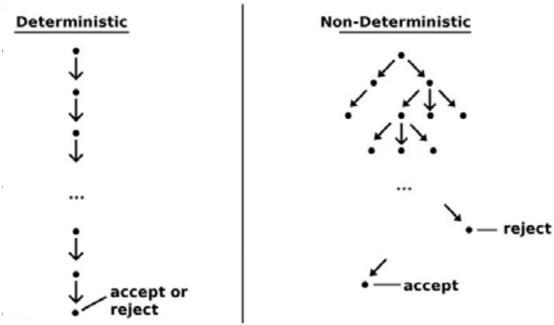
no conclusion

Another (equivalent) definition

An algorithm that *chooses* (by a good guess) some number of *non-deterministic bits* during its execution is called a *non-deterministic algorithm*.

We say that an algorithm *A non-deterministically accepts* a string *s* if there exists a *choice of non-deterministic bits* that ultimately leads to the answer "yes."

Possible exception?
Quantum computers:
no conventional gates.



Complexity Class NP

The class NP is the set of decision problems X (or languages $L(X)$) that can be *non-deterministically accepted* in polynomial time.

(A determinstic computer is what we know)
A nondeterministic computer is one that can “guess” the right answer or solution.
Thus NP can also be thought of as the class of problems whose solutions can be verified in polynomial time.

In the language of the previous definition, for an input string s, the non-deterministic algorithm can generate a (polynomial length) string t, then we use the algorithm B to check if $B(s,t)$ = “yes.” (And, importantly, if the answer is “yes”, then B will terminate in polynomial time with that output.)

Why NP?

The search for a string *t* that will cause an efficient certifier to accept the input *s* is often viewed as a *non-deterministic search* over a set of possible proofs.

For this reason, NP was named as an acronym for "non-deterministic polynomial time."

$\{0, 1\}$ Knapsack is in $\mathcal{NP}$

$\{0,1\}$ **Knapsack:** Let $I = \{i_1, i_2, \ldots\ i_n\}$ denote a collection of items where item i_j has integer weight $w_j > 0$ and gives integer benefit b_j. Also, we're given a maximum weight W and an integer k.

Suppose someone proposes a subset of items, it is easy to check:

- if those items have total weight at most W;
- if the total value is at least k.

If both conditions are true, then the subset of items is a certificate for the decision problem, i.e., it verifies that the answer to the 0/1 knapsack decision problem is "Yes"

P and NP

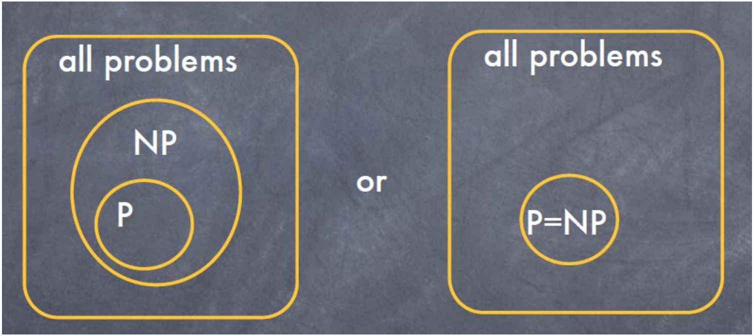
Note that $\mathrm{P} \subseteq \mathrm{NP}$. Basically, if there is a polynomial-time algorithm *A* to solve a problem, then we can “ignore” any proposed solution *t* and return the answer that the algorithm *A* gives (i.e. whatever *A* returns, either “yes” or “no”, we give that same answer).

The (million dollar) question that mathematicians and computer scientists *don’t know* the answer to is whether $\mathrm{P} = \mathrm{NP}$ or $\mathrm{P} \neq \mathrm{NP}$.

There is generally common *belief* that P and NP are different, i.e. there is some problem that is in NP but is not in P.

P and N P

Although poly-time verifiability seems like a weaker condition than poly time solvability, it is unknown whether P = NP.

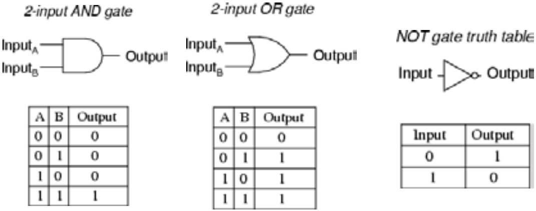


Boolean Circuit

In discrete Math, you learned about *propositional logic*, and the basic operations for combining truth values.

A Boolean Circuit is a directed graph where each vertex, called a *logic gate* corresponds to a simple Boolean function, one of AND, OR, or NOT.

Incoming edges: inputs for its Boolean function
Outgoing edges: outputs

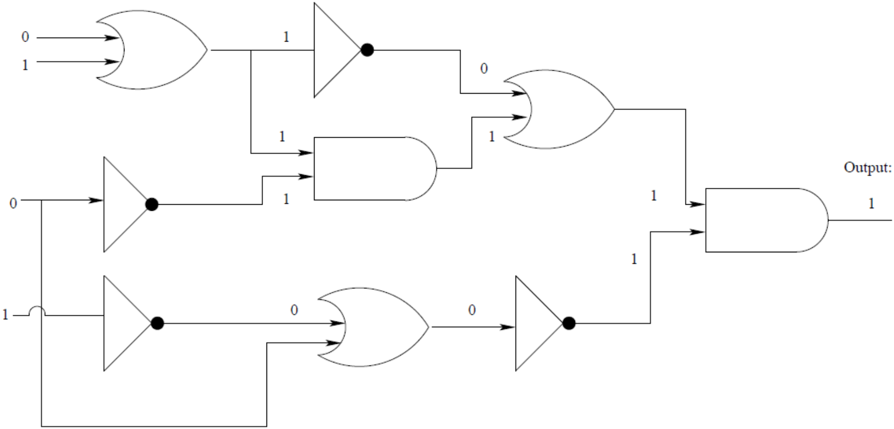


A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Input	Output
0	1
1	0

Boolean Circuit - Example



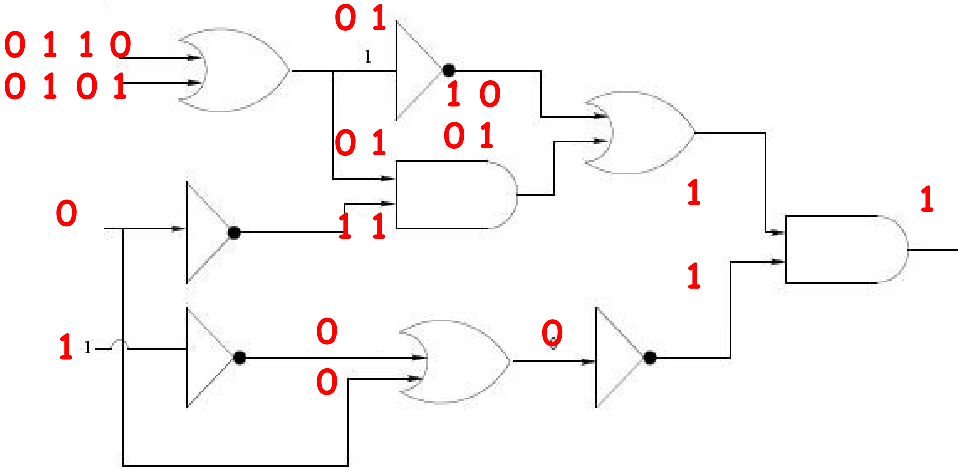
Circuit-SAT

Input: a Boolean Circuit with a single output vertex
Question: is there an assignment of values to the inputs so that the output value is 1?

A non-deterministic algorithm chooses an assignment of input bits and checks deterministically whether this input generates an output of 1.

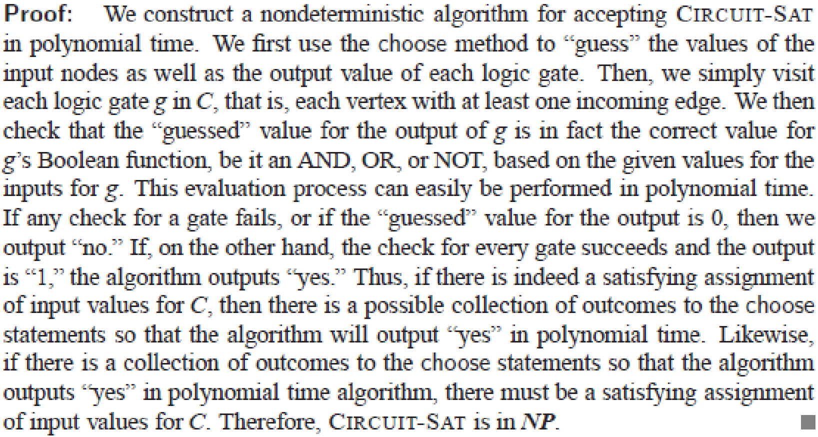
SAT means satisfiability

Circuit-SAT



Circuit-SAT is in NP

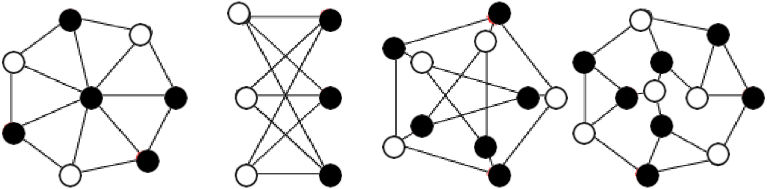
Circuit-SAT: Given a Boolean Circuit with a single output node, is there an assignment of values to the inputs so that the output value is 1?



Proof: We construct a nondeterministic algorithm for accepting CIRCUIT-SAT in polynomial time. We first use the choose method to "guess" the values of the input nodes as well as the output value of each logic gate. Then, we simply visit each logic gate g in C, that is, each vertex with at least one incoming edge. We then check that the "guessed" value for the output of g is in fact the correct value for g's Boolean function, be it an AND, OR, or NOT, based on the given values for the inputs for g. This evaluation process can easily be performed in polynomial time. If any check for a gate fails, or if the "guessed" value for the output is 0, then we output "no." If, on the other hand, the check for every gate succeeds and the output is "1," the algorithm outputs "yes." Thus, if there is indeed a satisfying assignment of input values for C, then there is a possible collection of outcomes to the choose statements so that the algorithm will output "yes" in polynomial time. Likewise, if there is a collection of outcomes to the choose statements so that the algorithm outputs "yes" in polynomial time algorithm, there must be a satisfying assignment of input values for C. Therefore, CIRCUIT-SAT is in NP. ■

Vertex Cover (VC)

Given a graph G = (V,E)
A vertex cover is a subset C $\subseteq$ V such that for every edge (v,w) in E, v $\in$ C or w $\in$ C

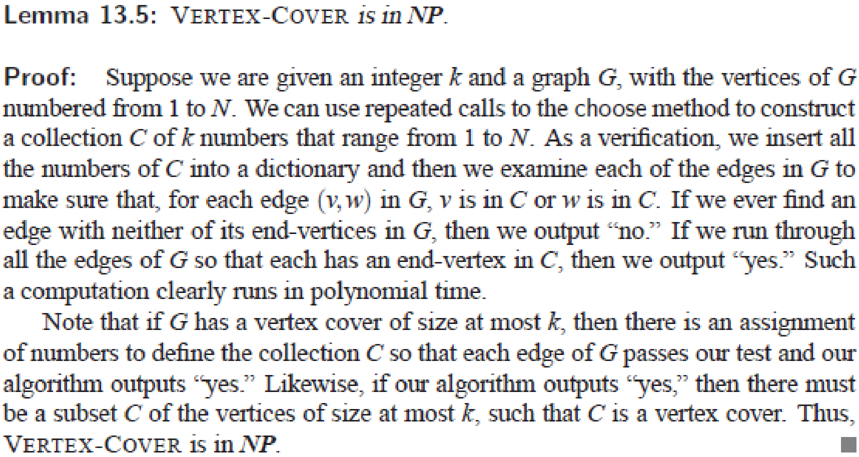


some graphs and their vertex cover (shaded vertices)

The optimisation problem is to find as small a vertex cover as possible

Vertex Cover is the decision problem that takes a graph G and an integer k and asks whether there is a vertex cover for G containing *at most* k vertices

Vertex Cover (VC)



Lemma 13.5: Vertex-Cover *is in* NP.

Proof: Suppose we are given an integer k and a graph G, with the vertices of G numbered from 1 to N. We can use repeated calls to the choose method to construct a collection C of k numbers that range from 1 to N. As a verification, we insert all the numbers of C into a dictionary and then we examine each of the edges in G to make sure that, for each edge (v,w) in G, v is in C or w is in C. If we ever find an edge with neither of its end-vertices in G, then we output "no." If we run through all the edges of G so that each has an end-vertex in C, then we output "yes." Such a computation clearly runs in polynomial time.

Note that if G has a vertex cover of size at most k, then there is an assignment of numbers to define the collection C so that each edge of G passes our test and our algorithm outputs "yes." Likewise, if our algorithm outputs "yes," then there must be a subset C of the vertices of size at most k, such that C is a vertex cover. Thus, Vertex-Cover is in NP. ■

Polynomial-time reducibility

Another important idea in the theory of N P-completeness is the idea of transforming one problem instance into another problem instance. We formalize this idea below.

- We say that a language L, defining some decision problem, is *polynomial-time reducible* to a language M, if: there is a function f , computable in polynomial time, that takes as an input $s \in L$, and transforms it into a input $f(s) \in M$ such that $s \in L$ if and only if $f(s) \in M$.

In other words, we can transform an input for one decision problem into an appropriate input for another decision problem. Furthermore, the first problem has a “yes” solution if and only if the second decision problem has a “yes” solution.

We use notation $L \overset{poly}{\rightarrow} M$ to signify that language L is polynomial time reducible to language M.

NP-hardness

We say that a language M, defining some decision problem, is NP-*hard* if every other language L in NP is polynomial-time reducible to M. So this means that

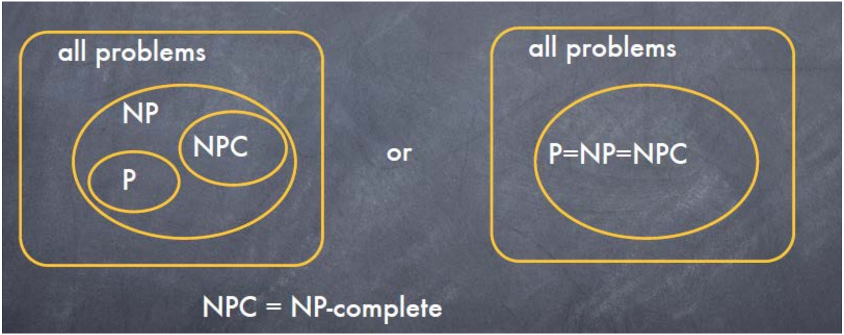
- M is NP-hard if for every $L \in NP$, $L \xrightarrow{poly} M$.

If a language M is NP-hard and it belongs to NP itself, then M is NP-*complete*.

NP-complete problems are some of the *hardest problems* in NP.

N P-complete

NP-complete problems is class of "hardest" problems in NP!
• If an NP-complete problem can be solved in polynomial time, then all problems in NP can be solved in polynomial time, and thus P = NP.



NP-complete

NP-complete problems is class of "hardest" problems in NP!

- If an NP-complete problem can be solved in polynomial time, then all problems in NP can be solved in polynomial time, and thus P = NP.

The question whether P=NP is open since the 70s. It is one of the central open problems in computer science. The question is of theoretical interest as well as of great practical importance.

Pragmatic approach: If your problem is NP-complete, then don't waste time looking for an efficient algorithm, focus on approximations or heuristics.

Cook-Levin Theorem

The Cook-Levin Theorem states that *Circuit-SAT* is

N P-complete. [Cook 1971, Levin 1973]

Proof:

1. CIRCUIT-SAT ∈ NP

Proof: Can verify an input assignment satisfies a circuit by computing the output of a finite number of gates, one of which will be the output of the circuit. This can be done in polynomial time. Thus, by definition of NP, CIRCUIT- SAT ∈ NP.

2. CIRCUIT-SAT ∈NP-Hard

I.e., $L \xrightarrow{poly}$ CIRCUIT-SAT for every L ∈NP

Proof: Complex

Show that any problem in NP can be computed using a Boolean combination circuit (i.e., a computer).

This circuit has a polynomial number of elements and can be constructed in polynomial time. Thus, $L \xrightarrow{poly}$ CIRCUIT-SAT for all L ∈NP.

Thus, CIRCUIT-SAT ∈NP-Hard.

N P-complete

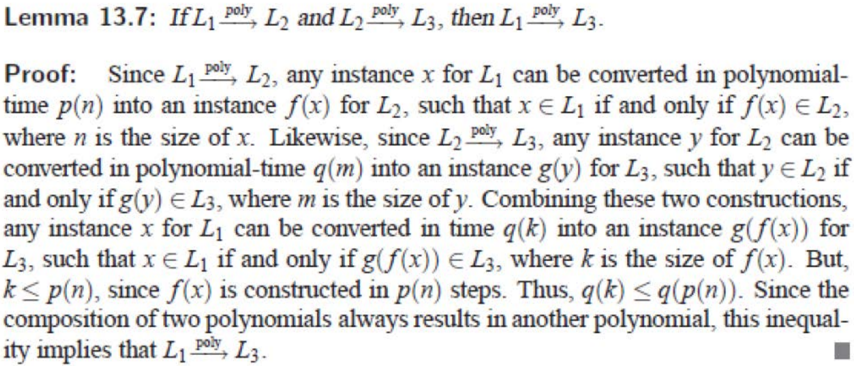
We have just noted that there is *at least* one N P-complete problem.

Using polynomial time reducibility we can show existence of other N P-complete problems.

The following useful result also helps to prove N P-completeness in many cases.

Lemma: If $L_1 \overset{poly}{\rightarrow} L_2$ and $L_2 \overset{poly}{\rightarrow} L_3$ then $L_1 \overset{poly}{\rightarrow} L_3$

N P-complete



Lemma 13.7: *If $L_1 \xrightarrow{\text{poly}} L_2$ and $L_2 \xrightarrow{\text{poly}} L_3$, then $L_1 \xrightarrow{\text{poly}} L_3$.*

Proof: Since $L_1 \xrightarrow{\text{poly}} L_2$, any instance x for L_1 can be converted in polynomial-time $p(n)$ into an instance $f(x)$ for L_2, such that $x \in L_1$ if and only if $f(x) \in L_2$, where n is the size of x. Likewise, since $L_2 \xrightarrow{\text{poly}} L_3$, any instance y for L_2 can be converted in polynomial-time $q(m)$ into an instance $g(y)$ for L_3, such that $y \in L_2$ if and only if $g(y) \in L_3$, where m is the size of y. Combining these two constructions, any instance x for L_1 can be converted in time $q(k)$ into an instance $g(f(x))$ for L_3, such that $x \in L_1$ if and only if $g(f(x)) \in L_3$, where k is the size of $f(x)$. But, $k \le p(n)$, since $f(x)$ is constructed in $p(n)$ steps. Thus, $q(k) \le q(p(n))$. Since the composition of two polynomials always results in another polynomial, this inequality implies that $L_1 \xrightarrow{\text{poly}} L_3$. ■

NP-complete

Suppose that we have a new problem X and we think that X is NP-complete. How can we do this?

- First, show that $X \in \mathrm{NP}$, i.e. show that X has a polynomial-time nondeterministic algorithm, or equivalently, show that X has an efficient certifier.
- Secondly, take a *known* NP-complete problem Y, and demonstrate a polynomial time reduction from Y to X, i.e. show that $Y \overset{poly}{\to} X$.

Conjunctive Normal Form

A Boolean formula is in *Conjunctive Normal Form* (CNF) if it is formed as a collection of *clauses* combined using the operator *AND* ($\wedge$), where each clause is formed by *literals* (variables x_i or their negations $\overline{x_i}$) combined using the operator *OR* ($\vee$).

Some examples are:

$$(\overline{x_1} \vee \overline{x_2} \vee x_4 \vee \overline{x_6}) \wedge (\overline{x_2} \vee x_4 \vee \overline{x_5} \vee x_3)$$

$$(x_2 \vee \overline{x_3} \vee \overline{x_1}) \wedge (x_1 \vee x_3 \vee x_4) \wedge (\overline{x_4} \vee \overline{x_2} \vee \overline{x_5})$$

$$(\overline{x_2} \vee x_1) \wedge (\overline{x_1} \vee x_4 \vee x_3) \wedge (x_4 \vee x_3) \wedge (\overline{x_3} \vee x_2 \vee \overline{x_5} \vee \overline{x_1})$$

CNF-SAT and 3-SAT

CNF-SAT
Input: A Boolean formula in CNF.

Question: Is there an assignment of Boolean values to its variables so that the formula evaluates to 1 (i.e. is the formula *satisfiable*)?

3-SAT
Input: A CNF $C = c_1 \wedge c_2 \wedge \ldots \wedge c_m$ of clauses dependent on variables $x_1, x_2, \ldots, x_m$ such that each c_i is of the form $x_{i_1} \vee x_{i_2} \vee x_{i_3}$ for $1 \leq i \leq m$.

Question: Is there a truth assignment for variables x_i that satisfies C?

3-SAT and Vertex Cover (VC)

3-SAT is CNF-SAT in which each clause has exactly three literals.

Theorem: 3-SAT is N P-complete.

VC

Input: An undirected graph G and a positive integer k.

Question: Is there a set C of at most k vertices such that every edge of G is incident to at least one node of C ? (Such a set C is called a vertex cover of G.)

Theorem: VC is N P-complete.

3-Coloring of Graphs

3**-COL:**
Input: A graph *G*.

Question: Is *G* 3-colorable? In other words, is there an assignment of the labels $\{1, \ 2, \ 3\}$ to the *vertices* of *G* so that any two vertices that are adjacent are assigned *different* labels?

3-COL is part of the more general class of problems *k* **-COL**, where the vertices of a graph are assigned labels from the set $\{1, 2, \ldots, k\}$, and the goal is to assign labels so that adjacent vertices have different labels.

If $k \geq 3$, then *k* **-COL** is NP-complete.