INT202
Complexity of Algorithms

NP-Completeness

Year 2020-2021

Hard Computational Problems

Some computational problems seem *hard* to solve.

Despite numerous attempts we do not *know* any *efficient* algorithms for these problems.

However, we are also far away from a *proof* that these problems are indeed hard to solve. We simply “don’t know” or are unsure right now.

Hard Computational Problems (cont.)

In more formal language, we don't know if $\mathrm{NP} = \mathrm{P}$ or $\mathrm{NP} \neq \mathrm{P}$.

This is an important question in theoretical computer science! And this lecture is designed to shed some light on this question.

This question is one of the seven Millennium Prizes set out by the Clay Institute, and can earn someone $1 million dollars for its solution (as well as a place in math/computer science history). See http://www.claymath.org/millennium-problems/p-vs-np-problem for their description of this problem.

Hard Computational Problems (cont.)

P vs NP Problem

Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical.

Hard Computational Problems (cont.)

P vs NP Problem

Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students.

However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer.

Example already encountered

We have already seen one example of these difficult problems where no efficient algorithm (i.e. polynomial-time) is known (i.e. one that will work on *all* inputs in polynomial-time).

This is the $\{0,1\}$ Knapsack problem.

The $\{0, 1\}$ Knapsack problem

The $\{0,1\}$ Knapsack problem can be defined in the following way:

$\{0,1\}$ **Knapsack**
Input: A collection of items $\{i_1, i_2, \ldots\ i_n\}$ where item i_j has integer weight $w_j > 0$ and gives integer benefit b_j. Also, a maximum weight W.

Goal: Find a subset of the items whose total weight does not exceed W and that maximizes the total benefit (taking fractional parts of items is *not allowed*).

Note: The dynamic programming algorithm for the $\{0,1\}$ Knapsack problem runs in time $\theta(nW)$, which is *not* polynomial in the *size* of the problem.

The {0, 1} Knapsack problem

The {0,1} Knapsack problem can be defined in the following way:

{0,1} **Knapsack**

Note: The dynamic programming algorithm for the {0,1} Knapsack problem runs in time $\theta(nW)$, which is *not* polynomial in the *size* of the problem.

$\theta(nW)$ looks like a polynomial time, but it is not, it is pseudo-polynomial.

Time complexity measures the time that an algorithm takes as a function of the length in bits of its input. The dynamic programming solution is indeed linear in the value of W, but exponential in the length of W!

The { 0, 1} Knapsack problem

The size of input is log(W) bits for the weight (and O(n) for "value" and "weight" arrays).

So, the input size of weight is j = log(W) (and not mere W).
So, $W = 2^j$ (as binary is used).

The final complexity is O(n * W)

This O(n * W) can be rewritten as $O(n * 2^j)$, which exponential in the size of the input.

So, this solution is not polynomial.

Decision/Optimization problems

A *decision problem* is a computational problem for which the output is either *yes* or *no*.

In an *optimization problem* we try to *maximize* or *minimize* some value.

An optimization problem can often be turned into a decision problem by adding a parameter k, and then asking whether the value of the function (from the optimization problem) is *at most* or *at least* k.

If a decision problem is *hard*, then its related optimization version must also be *hard*.

Decision/Optimization problems (cont.)

Example:

Optimization problem: Let G be a connected graph with integer weights on its edges. Find the weight of a minimum spanning tree in G.

Decision problem: Let G be a connected graph with integer weights on its edges, and let k be an integer. Does G have a spanning tree of weight at most k?

Decision/Optimization problems (cont.)

$\{0,1\}$ **Knapsack (decision version)**

Input: A collection of items $\{i_1, i_2, \ldots\ i_n\}$ where item i_j has integer weight $w_j > 0$ and gives integer benefit $b_j > 0$. Also, a maximum weight W and an integer k.

Question: Is there a subset of the items whose total weight does not exceed W, and whose total benefit is at least k?

Note that being able to efficiently *answer* decision questions allows us to efficiently *solve* the related optimization problem.

We can use a type of "binary search" to find the optimal solution.

Problems and Languages

Here we start giving a more formal definition of the classes of problems called P and NP.

- The input to a computational problem will be encoded as a finite binary string s. The length of the string is denoted as $|s|$.
- We *identify* a decision problem with a set of strings for which the answer is "yes".
- We say that the algorithm A for the decision problem *accepts* an input string s if A outputs "yes" on input s.

Problems and Languages (cont.)

For some decision problem X , let $L(X)$ denote the set of (binary) strings that should be accepted by an algorithm for that problem. We often refer to $L(X)$ as a *language*.

- We say that an algorithm A *accepts* a language $L(X)$ if A outputs “yes” for each $s \in L(X)$ and outputs “no” otherwise.

Language Examples

Now consider the $\{0,1\}$ Knapsack Problem (the decision version).

The *encoding* of the problem will be the binary representations of the integer weights, benefits, the maximum weight W, and the parameter k. (It takes $\log_2 k$ bits to represent an integer k.)

The *language* for an algorithm that would solve the $\{0,1\}$ knapsack problem consists of *all* collections of weight $\{w_i\}$, benefit $\{b_i\}$, total weight W, k parameters for which there is an allowable subset of the given items (i.e. total weight of the selected items in at most W) with total benefit at least k.

Complexity Class P

The *complexity class* P is the set of all decision problems X that can be solved in *worst-case* polynomial time. (Equivalently, P consists of all languages $L(X)$ that can be accepted in polynomial time.)

- That is, they are solvable in O(p(n)), where p(n) is a polynomial on n
- A deterministic algorithm is (essentially) one that always computes the correct answer

That is, there is an algorithm A that if $s \in L(X)$, then on input s, A outputs "yes" in time $p(|s|)$, where $|s|$ is the *length* of s, and $p(\cdot)$ is a *polynomial*.

The Class P

Single-source-shortest-paths problem is in P

Given a weighted graph G, a source vertex s, and a value k, does there exist shortest paths from s to every other vertex whose path length is at most k?

Theorem: Given a weighted graph with n vertices and m edges, each with non-negative weight, Dijkstra's algorithm solves the SSSP in $O(m\log n)$ time.

Then run Dijkstra's algorithm (polynomial time) and if the paths found have lengths at most k, then answer is "Yes"

Complexity Class P (cont.)

Many, many problems are in the class P.

Examples of problems that we have seen in this course include (decision versions of)

- fractional knapsack,
- shortest paths in graphs with non-negative edge weights,
- task scheduling.

Efficient certification

Another important idea when we discuss decision problemsis the idea of *certification*.

Example: Sudoku

The problem is given as an n^2 x n^2 array which is divided into blocks of n x n squares. Some array entries are filled with an integer in the range [1.. n^2].

The goal is to complete the array such that each row, column, and block contains each integer from [1.. n^2].

Problem

			4
3		2	
	1		3
4			

Solution

1	2	3	4
3	4	2	1
2	1	4	3
4	3	1	2

The Sudoku decision problem is whether a given Sudoku problem has a solution. Finding the solution might be difficult, but verifying the solution is easy.

Efficient certification

Another important idea when we discuss decision problems is the idea of *certification*.

Example: $\{0,1\}$ Knapsack Problem
Given a collection of weights, benefits, and parameters W and k for an instance of the $\{0,1\}$ Knapsack Problem, if I propose a subset of these items, it is *easy* to *check* if those items have total weight at most W and if the total benefit is at least k.

If both of those conditions are satisfied, then we say that 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."

The idea of *efficient certification* is what is used to define the class of problems called NP.

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.

Complexity class NP

0/1 Knapsack problem is in NP

we can check in polynomial time if a proposed subset of items whose weight is at most W and whose value is at least k

Circuit-SAT is in NP (to be introduced later)

we can check in polynomial time if proposed values lead to a final output value of 1

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."

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 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

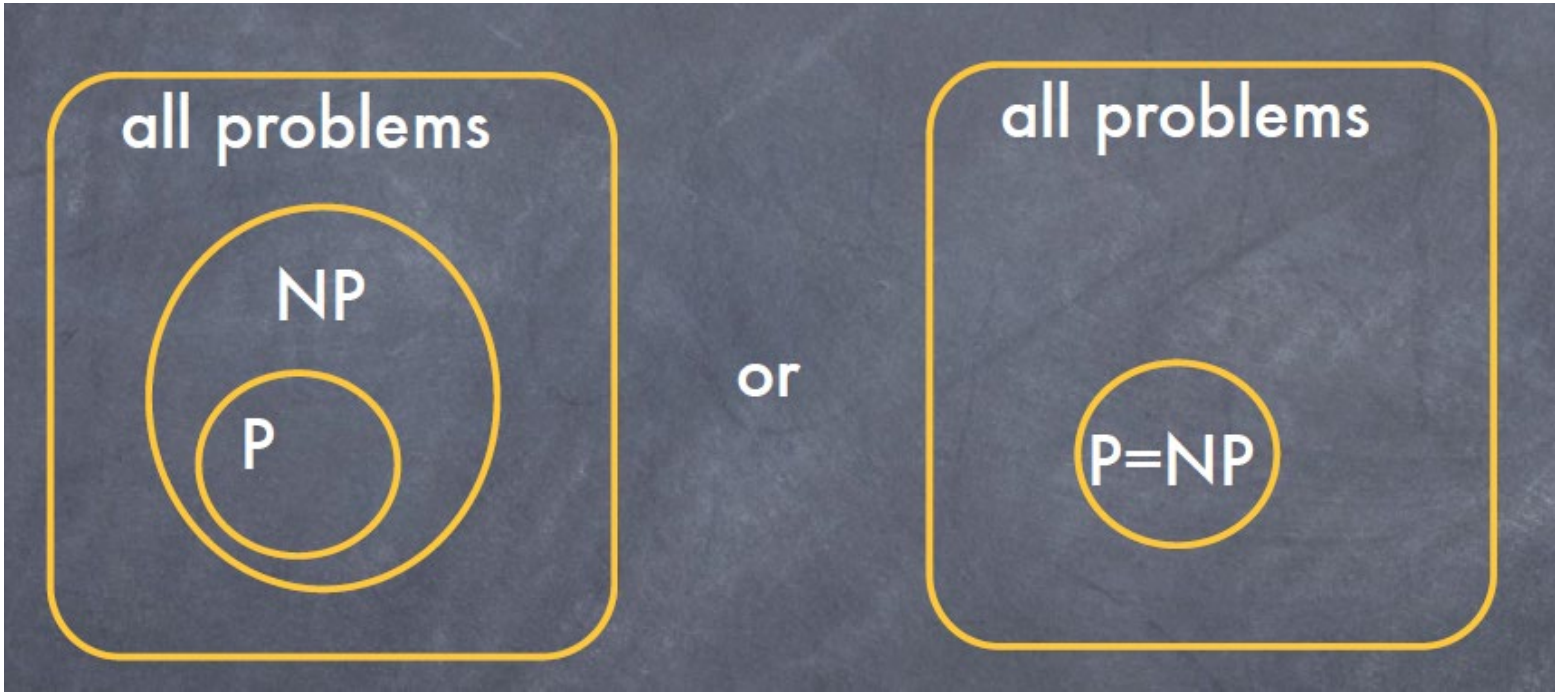
Note that $\mathrm{P} \subseteq \mathrm{NP}$. Basically, if there is a polynomial-time algorithm *A* to solve a problem, then we can "ignore" any proposed solution *t* and return the answer that the algorithm *A* gives (i.e. whatever *A* returns, either "yes" or "no", we give that same answer).

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.



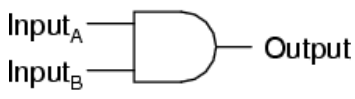
Boolean Circuit

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.

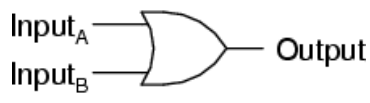
Incoming edges: inputs for its Boolean function
Outgoing edges: outputs

2-input AND gate



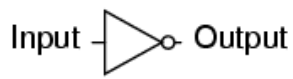
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

2-input OR gate



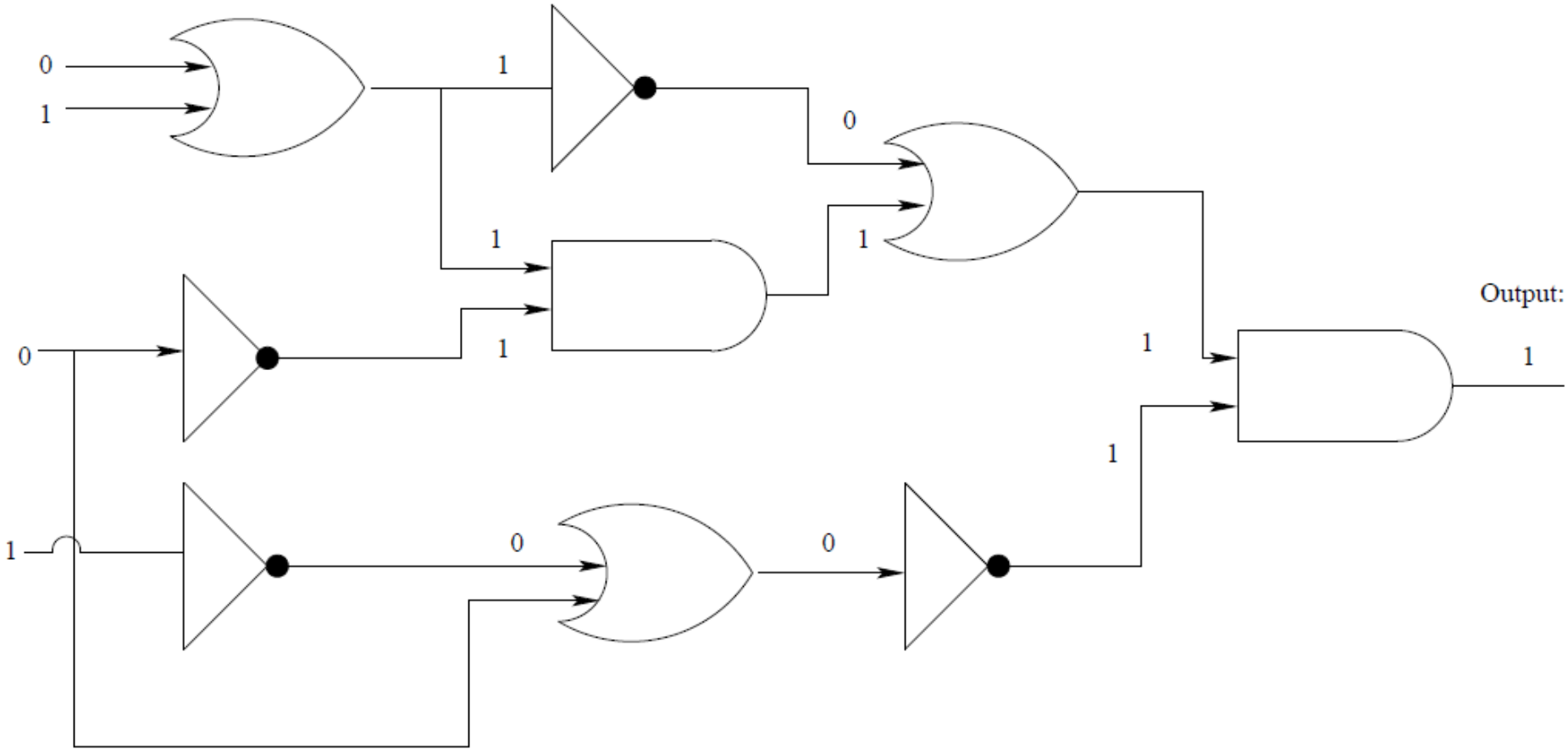
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

NOT gate truth table



Input	Output
0	1
1	0

Boolean Circuit - Example



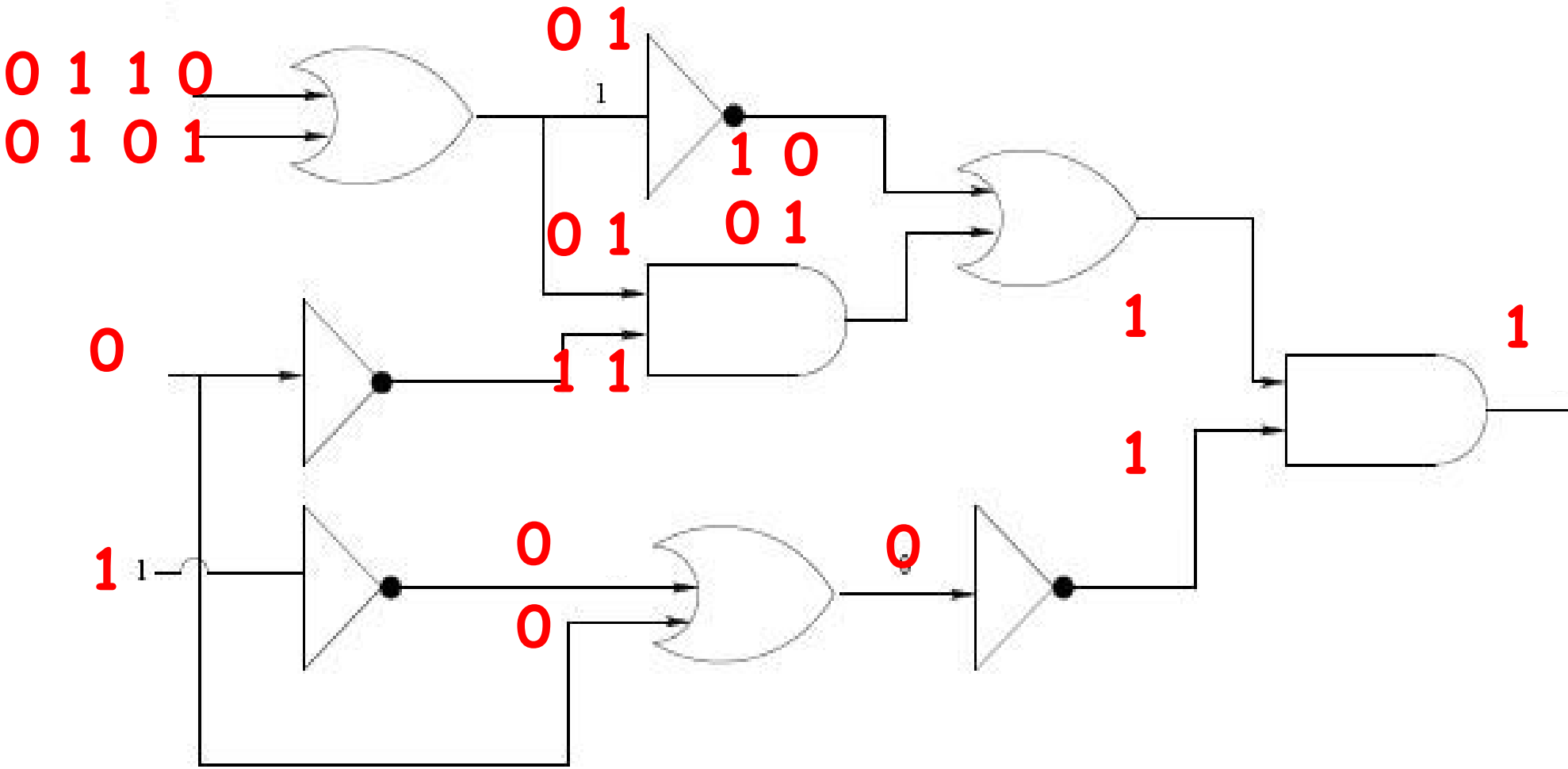
Circuit-SAT

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. ■