Lakehead
UNIVERSITY

COMP 4433: Algorithm Design and Analysis

Dr. Y. Gu

March 29 2023 (Lecture 19)

NP Completeness (Part II) (Chapter 34)

Computer Rep. for Problem Solving

An encoding of a set S of abstract objects is a mapping e from S to the set of binary strings. For example, we are all familiar with encoding the natural numbers N = {0, 1, 2, 3, · · · } as the strings {0, 1, 10, 11, . . . }. Using this encoding, e(17) = 10001.

Thus, a computer algorithm that "solves" some abstract decision problem actually takes an encoding of a problem instance as input. We call a problem whose instance set is the set of binary strings a **concrete problem**.

Formal Language Theorem

Let's review some definitions from formal-language theory.

An alphabet Σ is a finite set of symbols.

A language L over Σ is any set of strings made up of symbols from Σ.

For example, if $\Sigma = \{0, 1\}$, the set $L = \{10, 11, 101, 111, 1011, \cdots\}$ is the language of binary representation of prime numbers.

We denote the empty string by ϵ (*epsilon*), the empty language by $\varnothing$, and the language of all strings over Σ by Σ^* .

For example, if $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \cdots\}$ is the set of all binary strings.

Formal Language Theorem

We can perform a variety of operations on languages.

Set-theoretic operations, such as **union** and **intersection**, follow directly from the set-theoretic definitions.

We define the **complement** of L by $\bar{L} = \Sigma^* - L$.

The **concatenation** L_1L_2 of two languages L_1 and L_2 is the language $L = \{x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\}$.

The **closure** or **Kleene star** of a language L is the language $L^* = \{\epsilon\} \cup L \cup L_2 \cup L_3 \cup \cdots$, where L_k is the language obtained by concatenation L to itself k times.

Formal Language Theorem

The formal-language framework allows us to express **concisely** the relation between decision problems and algorithms that solve them.

We say that an algorithm A accepts a string $x \in \{0, 1\}$ if, given input x, the algorithm's output A(x) is 1.

The language accepted by an algorithm A is the set of strings $L = \{x \in \{0, 1\}^* : A(x) = 1\}$, that is, the set of strings that the algorithm accepts.

An algorithm A rejects a string x if A(x) = 0.

Formal Language Theorem

A language L is decided (or say, can be verified) by an algorithm A if every binary string in L is accepted by A and every binary string not in L is rejected by A.

A language L is accepted in polynomial time by an algorithm A if it is accepted by A and if in addition there exists a constant k such that for any length-n string $x \in L$, algorithm A accepts x in time $O(n^k)$.

A language L is decided in polynomial time by an algorithm A if there exists a constant k such that for any length-n string $x \in \{0, 1\}$, the algorithm correctly decides whether $x \in L$ in time $O(n^k)$.

Formal Definition of P

Using this language-theoretic framework, we can provide an alternative definition of the complexity class P:

$P = \{L \subseteq \{0, 1\}^* : \text{there exists an algorithm A that decides L in polynomial time}\}$.

Formal Definition of NP

The complexity class NP is the class of languages that can be verified by a polynomial-time algorithm.

More precisely, a language **L ∈ NP** if and only if there exist a two-input polynomial-time algorithm A and a constant c such that

$L = \{x \in \{0, 1\}^* : \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x, y) = 1\}$.

We say that algorithm A verifies language L in polynomial time.

Formal Definition of NP

We say that a language L_1 is polynomial-time reducible to a language L_2, written $L_1 \leq_P L_2$, if

there exists a polynomial-time computable function $f : \{0, 1\}^* \rightarrow \{0, 1\}$ such that for all $x \in \{0, 1\}_*$, $x \in L_1$ if and only if $f(x) \in L_2$.

We call the function f the reduction function, and a polynomial-time algorithm F that computes f is a reduction algorithm.

Formal Definition of NP-Complete

Now we can give a more formal definition.

A language $L \subseteq \{0, 1\}^*$ is NP-complete if

1. $L \in NP$, and

2. $L' \leq_P L$ for every $L' \in NP$.

If a language L satisfies property 2, but not necessarily property 1, we say that L is NP-hard.

We also define **NPC** to be the class of NP-complete languages.

Formal Definition of NP-Complete

Theorem:

If any NP-complete problem is polynomial-time solvable, then P = NP.

Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.

Formal Definition of NP-Complete

To explain the name of NP (nondeterministic polynomial), we need to introduce an operation " choose" for algorithms.

- choose(b): this operation chooses in a nondeterministic way a bit and assigns it to b.

When an algorithm A uses the choose primitive operation, then we say A is nondeterministic.

We say that an algorithm A nondeterministically accept a string x if there exists a set of outcomes to the choose calls that A could make on input x such that A would ultimately output "yes". In other words, it is as if we consider all possible outcomes to choose and only select those that lead to acceptance if there is such a set of outcomes. Note that this is not the random choices.

Formal Definition of NP-Complete

Equivalently, we may use the following definition.

A **nondeterministic algorithm** is a two-stage procedure that takes as its input an instance I of a decision problem and does the following.

Nondeterministic ("guessing") stage: An arbitrary string *S* is generated that can be thought of as a candidate solution (certificate) to the given instance *I* (but may be complete gibberish as well).

Deterministic ("verification") stage: A deterministic algorithm takes both *I* and *S* as its input and outputs yes if *S* represents a solution to instance *I*. (If S is not a solution to instance *I* , the algorithm either returns no or is allowed not to halt at all.

Nondeterministic Algorithm

Equivalently, we may use the following definition.

A **nondeterministic algorithm** is a two-stage procedure that takes as its input an instance I of a decision problem and does the following.

Nondeterministic ("guessing") stage: An arbitrary string *S* is generated that can be thought of as a candidate solution (certificate) to the given instance *I* (but may be complete gibberish as well).

Deterministic ("verification") stage: A deterministic algorithm takes both *I* and *S* as its input and outputs yes if *S* represents a solution to instance *I*. (If S is not a solution to instance *I* , the algorithm either returns no or is allowed not to halt at all.)

Nondeterministic Algorithm

We say that a nondeterministic algorithm solves a decision problem if and only if for every yes instance of the problem it returns yes on some execution.

A nondeterministic algorithm is said to be nondeterministic polynomial if the time efficiency of its verification stage (deterministic stage) is polynomial.

Now we can define the class of NP as the class of decision problems that can be solved by nondeterministic polynomial algorithms.

NP-Completeness

Because the technique of reduction relies on having a problem already known to be NP-complete in order to prove a different problem NP-complete, we need a “first” NP-complete problem.

The problem we shall use is the circuit-satisfiability problem, in which we are given a boolean combinational circuit composed of AND, OR, and NOT gates, and we wish to know whether there exists some set of boolean inputs to this circuit that causes its output to be 1.

The Circuit-Satisfiability Problem

Because the technique of reduction relies on having a problem already known to be NP-complete in order to prove a different problem NP-complete, we need a “first” NP-complete problem.

The problem we shall use is the circuit-satisfiability problem, in which we are given a boolean combinational circuit composed of AND, OR, and NOT gates, and we wish to know whether there exists some set of boolean inputs to this circuit that causes its output to be 1.

The Circuit-Satisfiability Problem

CIRCUIT-SAT is closely related to Boolean satisfiability problem (SAT), and likewise, has been proven to be NP-complete.

It is a prototypical NP-complete problem; the Cook–Levin Theorem is sometimes proved on CIRCUIT-SAT instead of on the SAT, and then CIRCUIT-SAT can be reduced to the other satisfiability problems to prove their NP-completeness.

The Circuit-Satisfiability Problem

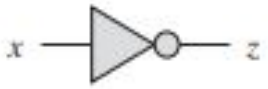
The boolean combinational elements that we use in the circuit-satisfiability problem are three basic logic gates:

the NOT gate (¬), the AND gate ($\wedge$), and the OR gate ($\vee$).

A boolean combinational circuit consists of one or more boolean combinational elements interconnected by wires.

A wire can connect the output of one element to the input of another, thereby providing the output value of the first element as an input value of the second.

The Circuit-Satisfiability Problem



x	$\neg x$
0	1
1	0

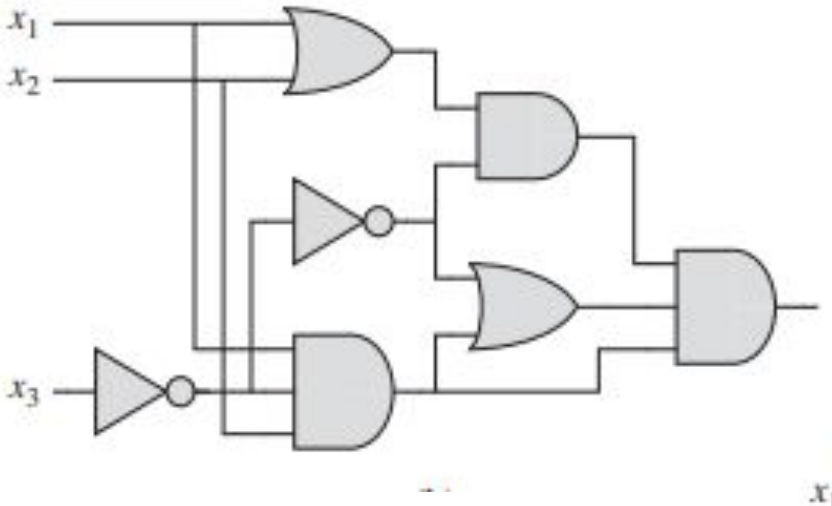
x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

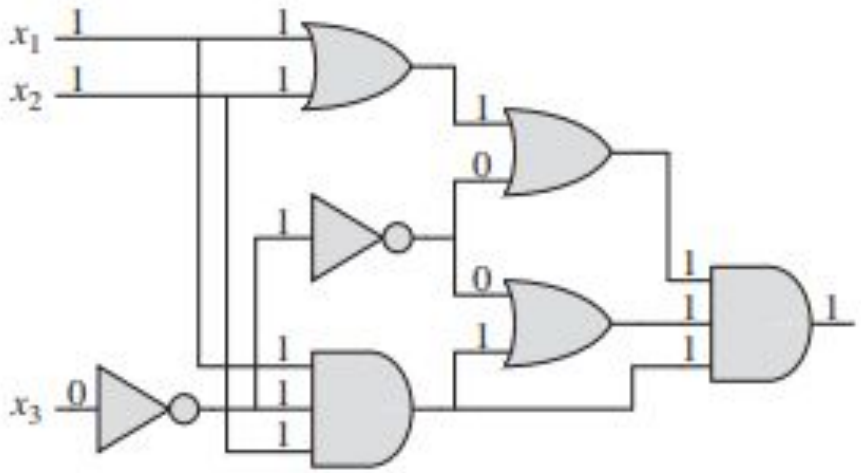
x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Boolean combinational circuits contain no cycles.

In other words, suppose we create a directed graph G = (V, E) with one vertex for each combinational element and with k directed edges for each wire whose fan-out is k; the graph contains a directed edge (u, v) if a wire connects the output of element u to an input of element v. Then G must be acyclic.

The Circuit Example





The Circuit-Satisfiability Problem

A truth assignment for a boolean combinational circuit is a set of boolean input values.

We say that a one-output boolean combinational circuit is satisfiable if it has a satisfying assignment:

A truth assignment that causes the output of the circuit to be 1.

The Circuit-Satisfiability Problem

The **circuit-satisfiability problem** is, "Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?"

In order to pose this question formally, however, we must agree on a standard encoding for circuits. The size of a boolean combinational circuit is the number of boolean combinational elements plus the number of wires in the circuit. We could devise a graph like encoding that maps any given circuit C into a binary string $\langle C \rangle$ whose length is polynomial in the size of the circuit itself. As a formal language, we can therefore define

CIRCUIT-SAT =

$\{\langle C \rangle : C \text{ is a satisfiable boolean combinational circuit}\}$.

The Circuit-Satisfiability Problem

The main idea to prove that the CIRCUIT-SAT is NP-complete is that:

for any language L in NP, we can provide some polynomial-time algorithm F computing a reduction function f that maps every binary string x to a circuit C = f(x) such that $x \in L$ *if and only if* $C \in$ CIRCUIT-SAT.

The formal detailed proof is omitted here.

One Way of Proof (informal)

Given a circuit and a satisfying set of inputs, one can compute the output of each gate in constant time. Hence, the output of the circuit is verifiable in polynomial time. Thus Circuit SAT belongs to complexity class NP.

To show NP-hardness, it is possible to construct a reduction from 3-SAT to Circuit SAT.

One Way of Proof (informal)

Suppose the original 3-SAT formula has variables x_1, x_2, .., x_n, and operators (AND, OR, NOT) . Design a circuit such that it has an input corresponding to every variable and a gate corresponding to every operator. Connect the gates according to the 3-SAT formula. For instance, if the 3-SAT formula is $(\neg x_1 \vee x_2) \wedge x_3$, the circuit will have 3 inputs, one AND, one OR, and one NOT gate. The input corresponding to x_1 will be inverted before sending to an OR gate with x_2, and the output of the AND gate will be sent to an AND gate with x_3.

One Way of Proof (informal)

Notice that the 3-SAT formula is equivalent to the circuit designed above, hence their output is same for same input. Hence, if the 3-SAT formula has a satisfying assignment, then the corresponding circuit will output 1, and vice versa. So, this is a valid reduction, and Circuit SAT is NP-hard.

This completes the proof that Circuit SAT is NP-Complete.

After Class

After class reading: Part VII 34.3.