

**Revision: The complexity class  $NP$**

- Decisions problems are all in the form “is this input an element of set  $L$ ”? Such a set is called a (formal) language.
- By extending the notion of computations, we defined Non-deterministic algorithms.
- Such algorithms have a facility to Guess something.
- The algorithm is allowed to reject something when the Guesses are not right.
- The algorithm must not accept something that it should reject, whatever the Guesses are. Thus it is not symmetric.
- If there is a non-deterministic algorithm running in polynomial time for a language, we say the problem is in  $NP$ .

ALG/TITCS L05-1

**$NP$  completeness**

- If any instance of problem  $A$  can be transformed, in polynomial time, into an instance of problem  $B$  so that the two instances (of two different problems) have the same answer, we say  $A$  is polynomial-time reducible to  $B$ .
- This means that if there is a polynomial algorithm for  $B$ , we can use that solution to solve  $A$  in polynomial time.
- In terms of formal languages, we write  $L_A \leq_P L_B$ . This captures the intuitive idea that  $A$  is *no more difficult* than  $B$ .
- Formal definition:  $L_A \leq_P L_B$  iff for all  $x$ , we can compute  $f(x)$  in polynomial time so that  $x \in L_A$  iff  $f(x) \in L_B$ .
- If a language  $L$  satisfies  $L' \leq_P L$  for all  $L' \in NP$ , we say  $L$  is  **$NP$ -hard**.
- If a language is in  $NP$  and is  $NP$ -hard, it is  **$NP$ -complete**.

ALG/TITCS L05-2

**What's meant by something is  $NP$ -complete?**

- It is not immediately clear whether  $NP$ -complete languages exist. We will show one such language, the circuit satisfiability problem.
- Therefore, if circuit satisfiability (or any  $NP$ -complete problem) can be solved in polynomial time (without using non-determinism), all problems in  $NP$  can be solved in polynomial time.
- Theorists trying to prove  $P = NP$  will try to solve one such problems in polynomial time.
- However, unless  $P = NP$ ,  $NP$ -complete problems cannot be solved in polynomial time. This is our general belief, although unproven.
- Thus showing a problem to be  $NP$ -complete means that the problem is probably **too difficult to be solved in polynomial time**. We say these problems are intractable.

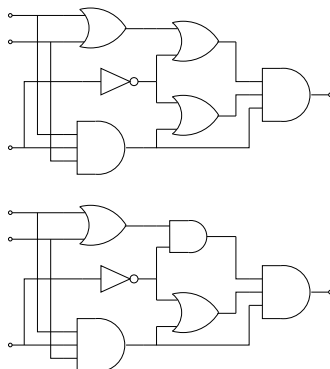
ALG/TITCS L05-3

**Problem: Circuit satisfiability (CIRCUIT\_SAT)**

- We are given a boolean circuit.
- The circuit contains some input. Each of them can be set to either 1 or 0 (i.e. true or false).
- There are some logic gates in the circuit. The input of the circuit is connected to the gates, and the output of the gate can connect to other gates. It is known that there is no cycle.
- The output of one gate is destined as the output of the circuit.
- Question: is there a way to set the inputs, so that the output is 1?

ALG/TITCS L05-4

**Example**



ALG/TITCS L05-5

**Circuit satisfiability is in  $NP$**

It is trivial to write a polynomial-time non-deterministic algorithm for circuit satisfiability:

- Use Guess() to get  $n$  values that are either 0 or 1.
- Treat the value obtained as the input for the circuit, and evaluate the circuit.
- If the output of the circuit is 1, accept. Otherwise, reject.

Clearly, if the circuit is satisfiable, there is a way to Guess() it. And if the circuit is not satisfiable, there is no way to Guess() it. So the algorithm is a correct (non-deterministic) algorithm for the problem.

This means it must be possible to solve circuit satisfiability in exponential time: try all possible sequence of Guess() outputs!

ALG/TITCS L05-6

**Circuit satisfiability is NP-hard**

- Can we do better? Yes. But can we do polynomial time? We don't know.
- However, we can show one thing: if it can be solved in polynomial time, every problem in *NP* can be solved in polynomial time.
- The basic technique is simple: reduction. We are going to reduce every problem in *NP* to circuit satisfiability. I.e. we will transform instances of every problem in *NP* as an instance of circuit satisfiability problem.
- The formal proof is quite complicated. We will thus settle on a less rigorous proof, which rely on our knowledge of the computers, which is used to execute these algorithms.
- In our proof, we make use of the definition of *NP* as a polynomial-time verifiable problem.

ALG/TITCS L05-7

**The idea**

- Every language *L* in *NP* is verifiable in polynomial time.
- I.e. there is a polynomial-time algorithm that takes the input and a certificate that is of length polynomial to the input, so that it accepts the input for some certificate if the input is in *L*, and it never accepts the input for any certificate if the input is not in *L*.
- Such algorithms can be implemented by a computer, so that it requires a polynomial number of steps to decide on the input and the certificate.
- The computer itself is a circuit with size polynomial to the size of memory, except there is a "clock" so that the outputs of the gates can be connected back to the gates.

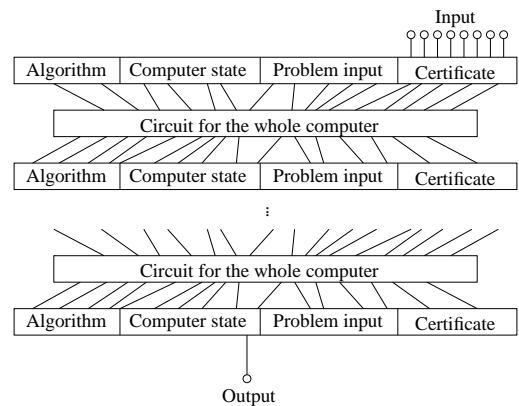
ALG/TITCS L05-8

**The idea (cont'd)**

- We can thus simulate a single step of a computer by using that circuit.
- By replicating the whole circuit, we can simulate the whole computation.
- The circuit itself is of size polynomial to the memory size, and can be computed in polynomial time.
- The number of times to replicate the circuit is polynomial, since the verification algorithm is polynomial.
- The place storing the answer can be made "true" if and only if there is a way the verification algorithm would accept the input, so the circuit satisfiability problem of this circuit is equivalent to the original problem.
- We thus have reduced the language *L* to a circuit satisfiability problem.

ALG/TITCS L05-9

**The circuit**



ALG/TITCS L05-10

**What would we do if we really want formal proof?**

- To really formally prove that this work, we would not want to use the "computer", which we cannot really specify rigorously.
- Instead, we would build a trimmed-down version of the computer that still encapsulate what it can do. This ends up with a Turing machine.
- A Turing machine can perform very few operations, thus we could actually argue exactly how the input is computed.
- On the other hand, things that can be done in polynomial time using computers can still be done in polynomial time using Turing machines (although much more slowly).
- Finally, we would use a problem that is easier to specify. We will use the formula satisfiability problem instead of the circuit satisfiability problem.

ALG/TITCS L05-11

**Showing other problems are NP-complete**

- Once we have at least one problem shown to be *NP* complete, we do not need reduce every problem in *NP* to a new problem in order to prove that the new problem is *NP*-hard.
- We know all problems in *NP* are polynomial-time reducible to the circuit satisfiability problem, so we just need to reduce the circuit satisfiability problem to the new problem.
- If we know other problems which are *NP*-hard, we can also use that in place of the circuit satisfiability problem.
- Therefore, it is important to learn a reasonable number of different *NP*-complete problems, so that when you notice something similar you know what problem to reduce.

ALG/TITCS L05-12

### Formula satisfiability

As an example, we show the *NP*-completeness of formula satisfiability.

#### Problem (SAT)

Given a boolean formula built using boolean operators like "and" and "or", and connecting "true", "false" and some variables  $x_i$  where  $i = 1, 2, \dots, k$ , determine whether there is an assignment for each of  $x_i$  so that the formula results in "true".

E.g.  $(x_1 \wedge x_2) \vee (x_1 \wedge x_3 \wedge (\neg x_1 \vee x_2))$

The problem is clear in *NP*: you can always Guess() the  $x_i$ 's. After that, it is trivial to evaluate in polynomial time the formula, in order to check whether the value is "true".

ALG/TITCS L05-13

### *NP*-completeness

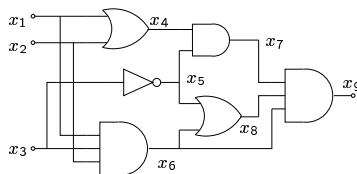
To show that SAT is *NP*-hard, we show  $\text{CIRCUIT\_SAT} \leq_P \text{SAT}$ . Three steps:

- Each instance  $A$  of CIRCUIT\_SAT can be transformed to another instance  $B$  of SAT. (Note the direction!)
- Whenever  $A$  is a satisfiable circuit, then  $B$  is a satisfiable formula.
- Whenever  $B$  is a satisfiable formula, then  $A$  is a satisfiable circuit.

So far, step 2 and step 3 are so similar that they are arrived at the same time. We will soon see reductions that one is much harder than the other.

ALG/TITCS L05-14

### The transformation



Give a variable to each circuit input and each gate output. For each gate, write out the formulae that need to be satisfied. e.g.:

- $(x_1 \vee x_2) \leftrightarrow x_4$
- $x_3 \leftrightarrow \neg x_5$

"And" all of them together with the variable correspond to the output gate. This forms a formula, using polynomial time.

ALG/TITCS L05-15

### Equivalence of the two problems

It leaves to show the two problems are equivalent.

- If the circuit is satisfiable, then there are some assignment of the circuit input that eventually give 1 to the output value. Trace through the circuit and find all the outputs of the gates. Use these as values for the variables of the formula, then the formula is satisfied.
- If the formula is satisfiable, then there must be some assignment to the variables that make the formula satisfied. If we map each value to the circuit input and gate output, and make the output gate 1, all the gates are consistent. So the values of the variables corresponding to the circuit input would make the circuit satisfied.

ALG/TITCS L05-16