NP-Completeness, part I

Christian Wulff-Nilsen
Advanced Algorithms and Data Structures
DIKU

December 12, 2022

Overview for today

- Problems and decision problems
- Polynomial-time solvable problems
- Definition of P
- Polynomial-time verifiable problems
- Definition of NP
- Reducibility
- NP-completeness
- The circuit-satisfiability problem

Definition of a problem

- Consider a set I of *instances* and a set S of *solutions*.
- An abstract *problem* is a binary relation between I and S, i.e., a subset of $I \times S$.
- For `SHORTEST-PATH`, an instance is a triple $\langle G, s, t \rangle$.
- A solution is a sequence of vertices forming a shortest s-to-t path.

Decision problems

- Unless otherwise stated, we only consider decision problems in this lecture and the next, i.e., problems with $1/0$ (yes/no) answers.
- Hence, $S = \{0, 1\}$.
- Example of a decision problem: $\texttt{PATH}$.
- $\texttt{PATH}(\langle G, u, v, k\rangle) = 1$ if there is a u-to-v path in G with at most k edges.
- Otherwise, $\texttt{PATH}(\langle G, u, v, k\rangle) = 0$.
- We can regard a decision problem as a mapping from instances to $S = \{0, 1\}$.
- Instances with solution 1 are called *yes*-instances.
- Instances with solution 0 are called *no*-instances.
- Optimization problems (like $\texttt{SHORTEST-PATH}$) can usually be turned into decision problems (like $\texttt{PATH}$).

Polynomial-time solvable problems

- We assume that instances of a problem are encoded as binary strings.
- An algorithm *solves* a problem in time $O(T(n))$ if for any instance of length n, the algorithm returns a solution (0 or 1) in time $O(T(n))$.
- If $T(n) = O(n^k)$ for some constant k, the problem is *polynomial-time solvable*.
- Suppose we define P as the class of polynomial-time solvable problems.
- What is missing in this definition? Which encoding of the input is assumed?

Which encoding to pick?

- Suppose an instance of some problem is a single number k.
- Suppose there is a $\Theta(k)$ time algorithm for the problem.
- We could choose an encoding of k in unary:

$$\overbrace{11\ldots1}^{k}.$$

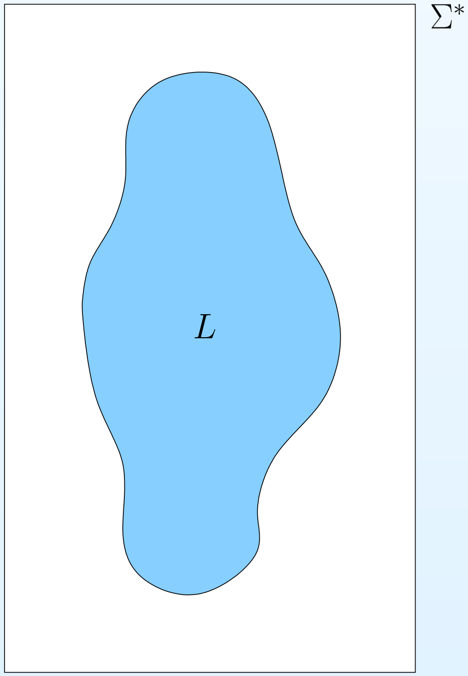
- In this case, the input size is $n = k$ and the algorithm runs in $\Theta(n)$ time which is polynomial in the input size.
- We could also choose a much more compact binary encoding, giving input size $n = \lfloor \lg k \rfloor + 1$.
- In this case, running time is $\Theta(k) = \Theta(2^n)$ which is exponential in the input size.
- These two ways of encoding k correspond to two different problems.

Which encoding to pick?

- In this lecture and the next, we consider problems with concise encodings.
- In particular, numbers are represented in binary, not unary.
- We use the notation $\langle x \rangle$ to refer to a chosen encoding of an instance x of a problem.
- Encodings are always binary strings in our setting.

Languages

- *Alphabet*: finite set Σ of symbols.
- *Language* L over Σ: a set of strings of symbols from Σ.
- Example: $\Sigma = \{a, b, c\}$ and $L = \{a, ba, cab, bbac, \ldots\}$.
- We also allow an empty string and denote it by ϵ.
- The empty language is denoted $\emptyset$ (it does not contain ϵ).
- Σ^* denotes the language of all strings (including ϵ).
- Any language L over Σ is a subset of Σ^*.



Languages and decision problems

- Recall that we encode instances of a decision problem as binary strings.
- Also recall that we may view a decision problem as a mapping $Q(x)$ from instances x to $\Sigma = \{0, 1\}$.
- Q can be specified by the binary strings that encode yes-instances of the problem.
- Thus, we can view Q as a language L:

$$L = \{x \in \Sigma^* | Q(x) = 1\}.$$

- For instance, `PATH` is the language of binary strings $\langle G, u, v, k\rangle$ where G is a graph, u and v are vertices of G, and there is a u-to-v path in G with at most k edges.

Language accepted/decided by an algorithm

- Let A be an algorithm for a decision problem and denote by $A(x) \in \{0, 1\}$ its output (if any) on input x.
- A *accepts* a string x if $A(x) = 1$.
- A *rejects* a string x if $A(x) = 0$.
- There may be strings that A neither accepts nor rejects.
- The language *accepted* by A is:

$$L = \{x \in \{0, 1\}^* | A(x) = 1\}.$$

- Suppose in addition that all strings not in L are rejected by A, i.e., $A(x) = 0$ for all $x \in \{0, 1\}^* \setminus L$.
- Then we say that L is *decided* by A.
- Deciding a language is stronger than accepting it.

Accepting/deciding in polynomial time

- Language L is *accepted by an algorithm* A *in polynomial time* if A accepts L and runs in polynomial time on strings from L.
- L is *decided by* A *in polynomial time* if A decides L and runs in polynomial time on all strings.
- Example: `PATH` can both be accepted and decided in polynomial time.
- We can now define the complexity class P:

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

P in terms of acceptance

- Lemma:

$$
\begin{aligned}
P &\stackrel{\text{def}}{=} \{L \subseteq \{0,1\}^* \mid \text{there exists an algorithm that} \\
&\qquad\qquad\qquad\qquad \text{decides } L \text{ in polynomial time}\} \\
&= \{L \subseteq \{0,1\}^* \mid \text{there exists an algorithm that} \\
&\qquad\qquad\qquad\qquad \text{accepts } L \text{ in polynomial time}\}.
\end{aligned}
$$

- $\subseteq$: straightforward.
- $\supseteq$: need to show that if L is accepted by a polynomial-time algorithm A, it is decided by a polynomial-time algorithm A'.

P in terms of acceptance

- Need to show: if L is accepted by a polynomial-time algorithm A, it is decided by a polynomial-time algorithm A'.
- Since A accepts L, it runs in at most cn^k steps before halting on any n-length string from L, where c and k are constants.
- Now let s be any string in Σ^*.
- A' simulates A with input s for at most $c|s|^k$ steps.
- If the simulation has not halted after this many steps, A' halts and outputs 0.
- Otherwise, A' outputs whatever A outputs.
- A' decides L and runs in polynomial time.

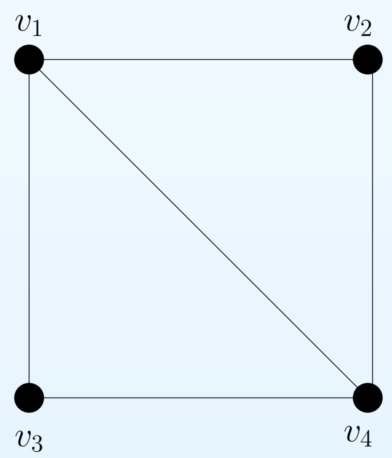
Verification

- Let L be a language.
- We might not have an efficient algorithm that accepts L.
- Consider an algorithm A taking two parameters, $x, c \in \Sigma^*$.
- Instead of trying to find a solution to x (which may take long time), A instead *verifies* that c is a solution to x.

The $\texttt{HAM-CYCLE}$ problem

- An undirected graph G is hamiltonian if it contains a simple cycle containing every vertex of G.
- We define

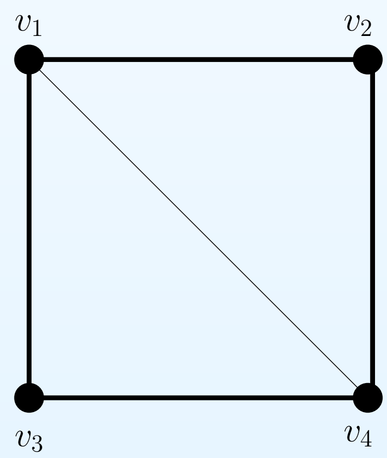
$$\texttt{HAM-CYCLE} = \{\langle G \rangle | G \text{ is Hamiltonian}\}.$$



The HAM-CYCLE problem

- An undirected graph G is hamiltonian if it contains a simple cycle containing every vertex of G.
- We define

$$\texttt{HAM-CYCLE} = \{\langle G\rangle | G \text{ is Hamiltonian}\}.$$



The $\texttt{HAM-CYCLE}$ problem

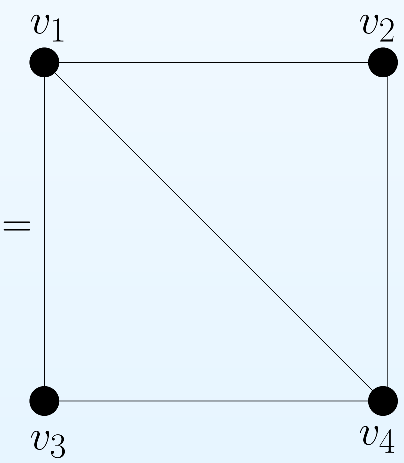
- An undirected graph G is hamiltonian if it contains a simple cycle containing every vertex of G.
- We define

$$\texttt{HAM-CYCLE} = \{\langle G \rangle | G \text{ is Hamiltonian}\}.$$

- It is open whether $\texttt{HAM-CYCLE}$ can be decided in polynomial time.
- However, it is easy to show (next slide) that $\texttt{HAM-CYCLE}$ can be verified in polynomial time.

Verifying `HAM-CYCLE`

- Consider instead an algorithm A_{ham} taking two parameters, $\langle G \rangle$ and $\langle C \rangle$.
- A_{ham} checks that $\langle G \rangle$ defines an undirected graph G and that $\langle C \rangle$ encodes a cycle C containing every vertex of G exactly once.
- If so, A_{ham} outputs 1, otherwise 0.

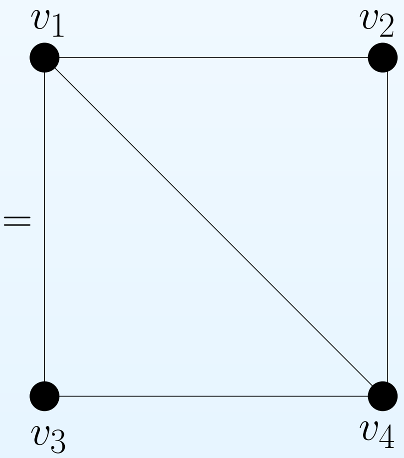
$G =$ 

$$C = [v_1, v_2, v_3, v_4]$$

- What is $A_{ham}(\langle G \rangle, \langle C \rangle)$?

Verifying `HAM-CYCLE`

- Consider instead an algorithm A_{ham} taking two parameters, $\langle G \rangle$ and $\langle C \rangle$.
- A_{ham} checks that $\langle G \rangle$ defines an undirected graph G and that $\langle C \rangle$ encodes a cycle C containing every vertex of G exactly once.
- If so, A_{ham} outputs 1, otherwise 0.

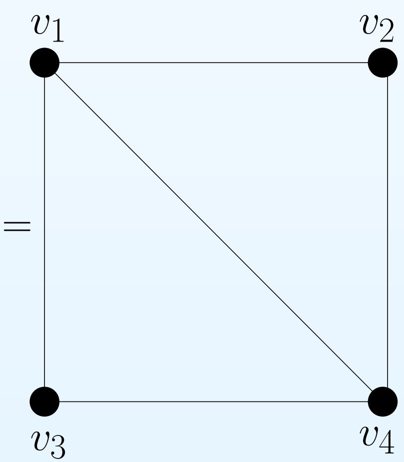
$G =$ 

$C = [v_1, v_2, v_3, v_4]$

- $A_{ham}(\langle G \rangle, \langle C \rangle) = 0$

Verifying `HAM-CYCLE`

- Consider instead an algorithm A_{ham} taking two parameters, $\langle G \rangle$ and $\langle C \rangle$.
- A_{ham} checks that $\langle G \rangle$ defines an undirected graph G and that $\langle C \rangle$ encodes a cycle C containing every vertex of G exactly once.
- If so, A_{ham} outputs 1, otherwise 0.

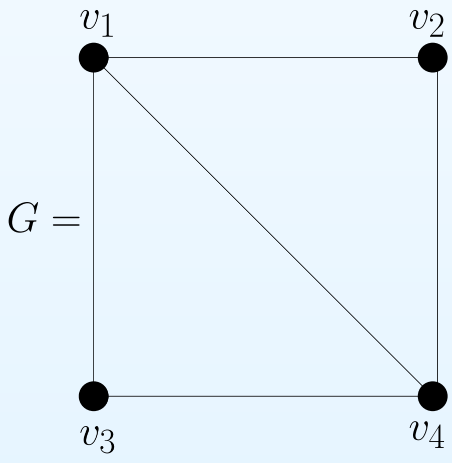
$G =$ 

$$C = [v_1, v_2, v_4, v_3]$$

- What is $A_{ham}(\langle G \rangle, \langle C \rangle)$?

Verifying `HAM-CYCLE`

- Consider instead an algorithm A_{ham} taking two parameters, $\langle G \rangle$ and $\langle C \rangle$.
- A_{ham} checks that $\langle G \rangle$ defines an undirected graph G and that $\langle C \rangle$ encodes a cycle C containing every vertex of G exactly once.
- If so, A_{ham} outputs 1, otherwise 0.



$C = [v_1, v_2, v_4, v_3]$

- $A_{ham}(\langle G \rangle, \langle C \rangle) = 1$

Verifying HAM-CYCLE

- Consider instead an algorithm A_{ham} taking two parameters, $\langle G \rangle$ and $\langle C \rangle$.
- A_{ham} checks that $\langle G \rangle$ defines an undirected graph G and that $\langle C \rangle$ encodes a cycle C containing every vertex of G exactly once.
- If so, A_{ham} outputs 1, otherwise 0.
- Designing A_{ham} to run in polynomial time is easy.
- Hence we can verify HAM-CYCLE in polynomial time.

Verifying a language

- A *verification algorithm* is an algorithm A taking two arguments, $x, y \in \{0,1\}^*$, where y is the *certificate*.
- A *verifies* a string x if there is a certificate y such that $A(x, y) = 1$.
- The language verified by A is

$$L = \{x \in \{0,1\}^* | \text{there is a } y \in \{0,1\}^* \text{ such that } A(x,y) = 1\}.$$

- Example:

$$\texttt{HAM-CYCLE} = \{x \in \{0,1\}^* | \text{there is a } y \in \{0,1\}^* \text{ such that } A_{ham}(x,y) = 1\}.$$

The complexity class NP

- NP is the class of languages that can be verified in polynomial time.
- More precisely, $L \in$ NP if and only if there is a polynomial-time verification algorithm A and a constant c such that

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

- We have seen that $\texttt{HAM-CYCLE} \in$ NP.
- If $L \in$ P then $L \in$ NP. Why?
- Hence, P $\subseteq$ NP.
- Big open problem: is P $=$ NP?

The complexity class co-NP

- co-NP is the class of languages L such that $\overline{L} \in$ NP.
- Does $L \in$ NP imply $L \in$ co-NP?
- For instance, is $\texttt{HAM-CYCLE} \in$ co-NP?
- Said differently, is $\overline{\texttt{HAM-CYCLE}} \in$ NP?
- In words, given a graph, can we easily verify that it does *not* have a simple cycle containing every vertex of G?
- What should we use as certificate? Not clear.
- It is open whether NP $=$ co-NP.
- What is known is that $P \subseteq$ NP $\cap$ co-NP.

NP-complete problems

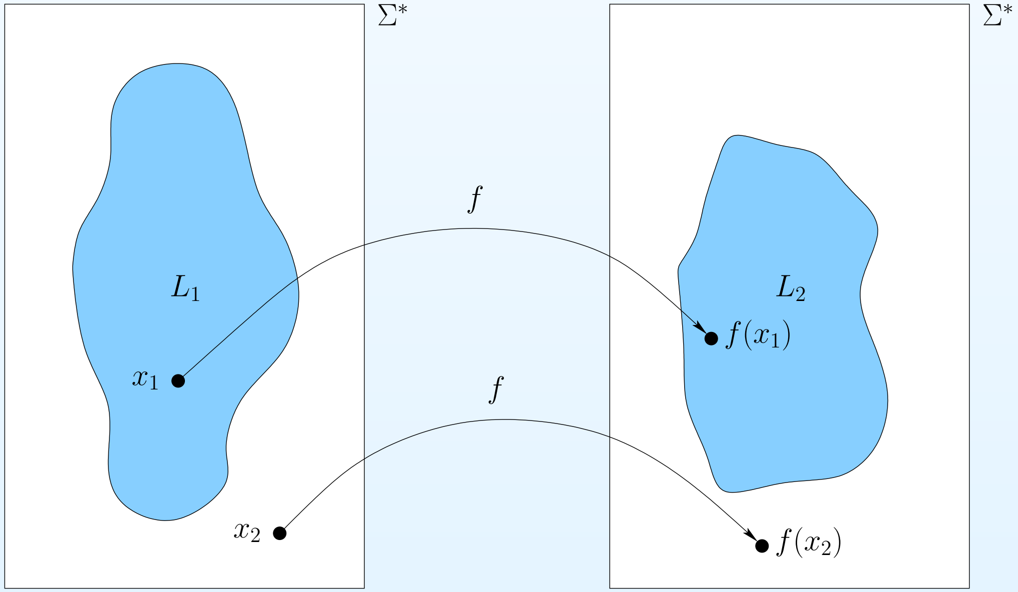
- There are problems in NP that are “the most difficult” in that class.
- If any one of them can be solved in polynomial time then *every* problem in NP can be solved in polynomial time.
- These difficult problems are called *NP-complete*.
- $\texttt{HAM-CYCLE}$ is NP-complete.
- Hence, if we could show $\texttt{HAM-CYCLE} \in \mathrm{P}$ then $\mathrm{P} = \mathrm{NP}$.
- We will see examples of several other NP-complete problems.
- To define NP-completeness, we need to first define polynomial-time reducibility.

Polynomial-time reducibility

- Language L_1 is polynomial-time *reducible* to language L_2 if there is a polynomial-time computible function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$$x \in L_1 \Leftrightarrow f(x) \in L_2.$$

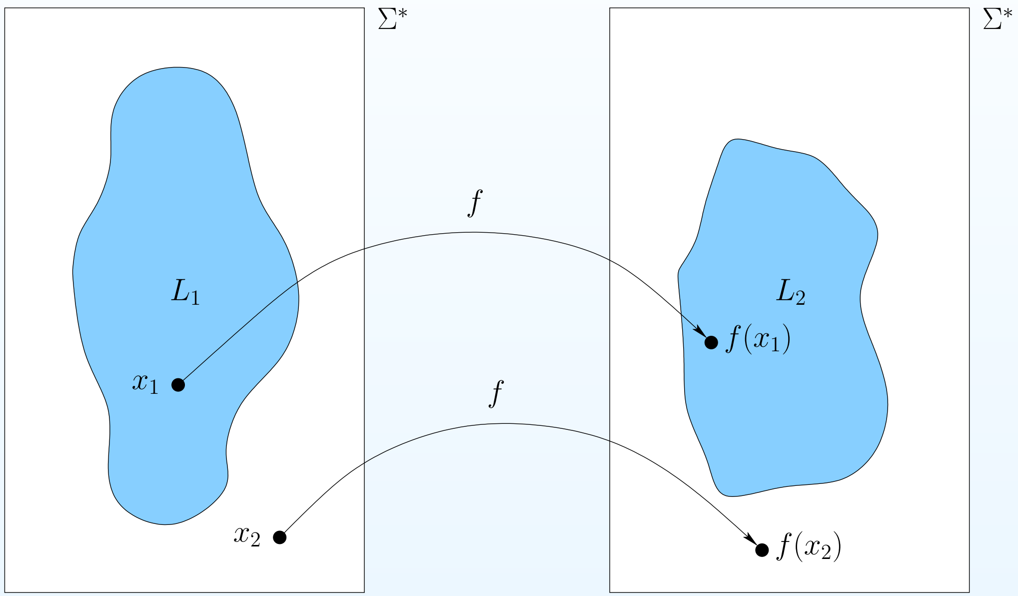
- In this case, we write $L_1 \leq_P L_2$.



- If $L_1 \leq_P L_2$ then L_1 is in a sense no harder to solve than L_2.

Polynomial-time reducibility

- If $L_1 \leq_P L_2$ then L_1 is in a sense no harder to solve than L_2.



- More precisely,

$$L_1 \leq_P L_2 \wedge L_2 \in P \Rightarrow L_1 \in P.$$

- This follows since any instance x of L_1 can be solved by transforming it in polynomial time to an instance $y = f(x)$ of L_2 and then solving y with a polynomial-time algorithm for L_2.

NP-complete languages

- Language L is *NP-complete* if
  1. $L \in$ NP and
  2. $L' \leq_P L$ for every $L' \in$ NP.
- L is *NP-hard* if property 2 holds (and possibly not property 1).
- The class of NP-complete languages is denoted NPC.
- If some language of NPC belongs to P then P $=$ NP. Why?
- It is not immediately clear from the definition that NP-complete languages even exist.
- In practice, why would it be useful to show that a problem is NP-complete?
- We next show that the circuit satisfiability problem is NP-complete.

An NP-complete problem: Circuit satisfiability

- A *boolean combinational circuit* consists of a collection of logic gates connected together with wires.
- The logic gates allowed are AND, OR, and NOT.
- Each wire has a value which is either 0 or 1.
- Some wires are specified by input values and the rest by the logic gates.
- Other wires specify output values.
- We can represent a circuit as an acyclic graph.

The circuit satisfiability problem

- Given a boolean combinational circuit C with one output wire.
- A *satisfying assignment* for C is an assignment of values to input wires of C causing an output of 1.
- The *circuit satisfiability problem* CIRCUIT-SAT is to decide if a given circuit has a satisfying assignment:

$$\texttt{CIRCUIT-SAT} = \{\langle C\rangle | C \text{ is a satisfiable boolean combinational circuit}\}.$$

- We will show that CIRCUIT-SAT is NP-complete.

Showing $\texttt{CIRCUIT-SAT} \in$ NP

- We construct algorithm A with inputs x and y.
- A checks that x represents a boolean combinational circuit C with one output wire and that y represents an assignment of truth values to the wires of C.
- If so, A checks that y represents a valid truth assignment.
- If so, A checks that the single output is 1.
- If this is the case, A returns 1; otherwise it returns 0.
- A is a verification algorithm for $\texttt{CIRCUIT-SAT}$ and can easily be made to run in polynomial time.
- Thus, $\texttt{CIRCUIT-SAT} \in \text{NP}$.

Showing that $\texttt{CIRCUIT-SAT}$ is NP-hard

- Consider any language $L \in$ NP.
- We need to give a polynomial-time reduction from L to $\texttt{CIRCUIT-SAT}$.
- In other words, we need to find a polynomial-time algorithm A computing a function $f : \{0, 1\}^* \to \{0, 1\}^*$ such that

$$x \in L \Leftrightarrow f(x) \in \texttt{CIRCUIT-SAT}.$$

Showing that CIRCUIT-SAT is NP-hard

- Since $L \in \text{NP}$, there is a polynomial-time algorithm A such that

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

- Given string x, f outputs a circuit $C(x)$ with $O(|x|^c)$ input wires.
- We ensure that $C(x)$ has a satisfying assignment of its input wires if and only if $A(x,y) = 1$ for some y with $|y| = O(|x|^c)$.
- This way,

$$x \in L \Leftrightarrow f(x) = \langle C(x) \rangle \in \texttt{CIRCUIT-SAT}.$$

- Each y with $|y| = O(|x|^c)$ defines an input to $C(x)$.
- Intuition: Circuit $C(x)$ implements algorithm A on input (x,y) with x fixed.
- We ensure that $A(x,y) = 1$ if and only if y is a satisfying assignment.

Showing that `CIRCUIT-SAT` is NP-hard

- There is a constant k such that the worst-case running time $T(n)$ of A on an input (x, y) is $O(n^k)$ where $n = |x|$.
- The machine executing A has a certain *configuration* at each time step.
- The configuration gives a complete specification of the current memory, CPU state, and so on.
- When executing A on (x, y), the machine goes through a series of configurations $c_0, c_1, \ldots, c_{T(n)}$ (assume for simplicity that A runs for exactly $T(n)$ steps on (x, y)).
- Configuration c_0 specifies inputs x and y and the program code for A.
- One bit of the last configuration $c_{T(n)}$ specifies the $0/1$-output of A.

Showing that `CIRCUIT-SAT` is NP-hard

- Let M be the circuit implementing the hardware of the machine.
- We feed the initial configuration c_0 as input wires to M.
- M performs a single step of A and the new configuration c_1 is stored on output wires.
- These output wires feed into M which makes another step, forming c_2 as output, and so on.
- In total, we glue $T(n)$ copies of M together.
- This gives a BIG circuit representing the entire execution of A on input (x, y).
- The size of the circuit is still polynomial in n, however.

Showing that $\texttt{CIRCUIT-SAT}$ is NP-hard

- We modify the circuit by hard-wiring part of the input to that specified by binary string x and so that the only output wire is that corresponding to the output of A.
- The circuit now only takes inputs y.
- The resulting circuit $C(x)$ has a satisfying assignment y if and only if $A(x, y) = 1$.
- $C(x)$ can be computed from x in time polynomial in $|x|$.
- This shows that $L \leq_P \texttt{CIRCUIT-SAT}$.
- Thus, $\texttt{CIRCUIT-SAT}$ is NP-hard.
- Since also $\texttt{CIRCUIT-SAT} \in$ NP, it follows that $\texttt{CIRCUIT-SAT}$ is NP-complete.

Plan for next lecture

- Showing NP-completeness of other problems using polynomial-time reductions:
    - SAT
    - 3-CNF-SAT
    - CLIQUE
    - VERTEX-COVER
    - (HAM-CYCLE)
    - TSP
    - SUBSET-SUM

Showing NP-completeness using reductions

- Suppose L' is an NP-complete language.
- Consider another language L.
- If $L' \leq_P L$ then L is NP-hard. Why?
- If also $L \in \text{NP}$ then L is NP-complete.
- Next time we show:

$$\begin{aligned}
\texttt{CIRCUIT-SAT} &\leq_P \texttt{SAT} \leq_P \texttt{3-CNF-SAT} \\
&\leq_P \texttt{SUBSET-SUM}, \\
\texttt{3-CNF-SAT} &\leq_P \texttt{CLIQUE} \leq_P \texttt{VERTEX-COVER} \\
&\leq_P \texttt{HAM-CYCLE} \leq_P \texttt{TSP}
\end{aligned}$$

- We also show that all these languages are in NP and hence they are NP-complete.