NP-Completeness, part I

Christian Wulff-Nilsen

Department of Computer Science

University of Copenhagen

Denmark

Overview for today

- Introduction

Overview for today

- Introduction
- Polynomial-time solvable problems

Overview for today

- Introduction
- Polynomial-time solvable problems
- Definition of P

Overview for today

- Introduction
- Polynomial-time solvable problems
- Definition of P
- Polynomial-time verifiable problems

Overview for today

- Introduction
- Polynomial-time solvable problems
- Definition of P
- Polynomial-time verifiable problems
- Definition of NP

Overview for today

- Introduction
- Polynomial-time solvable problems
- Definition of P
- Polynomial-time verifiable problems
- Definition of NP
- Reducibility

Overview for today

- Introduction
- Polynomial-time solvable problems
- Definition of P
- Polynomial-time verifiable problems
- Definition of NP
- Reducibility
- NP-completeness

Overview for today

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

Overview for today

- Introduction
- 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*.

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 $I \times S$.

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 $\langle G, s, t \rangle$.

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.

Decision problems

- We only consider decision problems.

Decision problems

- We only consider decision problems.
- Hence, $S = \{0, 1\}$.

Decision problems

- We only consider decision problems.
- Hence, $S = \{0, 1\}$.
- Example of a decision problem: PATH.

Decision problems

- We only consider decision problems.
- Hence, $S = \{0, 1\}$.
- Example of a decision problem: PATH.
- PATH$(\langle G, u, v, k\rangle) = 1$ if there is a u-to-v path in G with at most k edges.

Decision problems

- We only consider decision problems.
- Hence, $S = \{0, 1\}$.
- Example of a decision problem: PATH.
- PATH$(\langle G, u, v, k\rangle) = 1$ if there is a u-to-v path in G with at most k edges.
- Otherwise, PATH$(\langle G, u, v, k\rangle) = 0$.

Decision problems

- We only consider decision problems.
- Hence, $S = \{0, 1\}$.
- Example of a decision problem: PATH.
- PATH$(\langle G, u, v, k\rangle) = 1$ if there is a u-to-v path in G with at most k edges.
- Otherwise, PATH$(\langle G, u, v, k\rangle) = 0$.
- We can regard a decision problem as a mapping from instances to $S = \{0, 1\}$.

Decision problems

- We only consider decision problems.
- Hence, $S = \{0, 1\}$.
- Example of a decision problem: PATH.
- PATH$(\langle G, u, v, k\rangle) = 1$ if there is a u-to-v path in G with at most k edges.
- Otherwise, 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.

Decision problems

- We only consider decision problems.
- Hence, $S = \{0, 1\}$.
- Example of a decision problem: PATH.
- PATH$(\langle G, u, v, k\rangle) = 1$ if there is a u-to-v path in G with at most k edges.
- Otherwise, 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.

Decision problems

- We only consider decision problems.
- Hence, $S = \{0, 1\}$.
- Example of a decision problem: PATH.
- PATH$(\langle G, u, v, k\rangle) = 1$ if there is a u-to-v path in G with at most k edges.
- Otherwise, 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 can usually be turned into decision problems.

Polynomial-time solvable problems

- We assume that instances of a problem are encoded as binary strings.

Polynomial-time solvable problems

- We assume that instances of a problem are encoded as binary strings.
- An algorithm *solves* an instance of length n in time $O(T(n))$ if it returns a solution (0 or 1) in $O(T(n))$ time for some constant k.

Polynomial-time solvable problems

- We assume that instances of a problem are encoded as binary strings.
- An algorithm *solves* an instance of length n in time $O(T(n))$ if it returns a solution (0 or 1) in $O(T(n))$ time for some constant k.
- If $T(n) = n^k$, the problem is *polynomial-time solvable*.

Polynomial-time solvable problems

- We assume that instances of a problem are encoded as binary strings.
- An algorithm *solves* an instance of length n in time $O(T(n))$ if it returns a solution (0 or 1) in $O(T(n))$ time for some constant k.
- If $T(n) = n^k$, the problem is *polynomial-time solvable*.
- We define P as the class of polynomial-time solvable problems.

Polynomial-time solvable problems

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

Which encoding to pick?

- Suppose an instance of a problem is a single number k.

Which encoding to pick?

- Suppose an instance of a problem is a single number k.
- Suppose there is a $\Theta(k)$ time algorithm.

Which encoding to pick?

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

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

Which encoding to pick?

- Suppose an instance of a problem is a single number k.
- Suppose there is a $\Theta(k)$ time algorithm.
- 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.

Which encoding to pick?

- Suppose an instance of a problem is a single number k.
- Suppose there is a $\Theta(k)$ time algorithm.
- 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.
- We could also choose a much more compact binary encoding, giving $n = \lfloor \lg k \rfloor + 1$.

Which encoding to pick?

- Suppose an instance of a problem is a single number k.
- Suppose there is a $\Theta(k)$ time algorithm.
- 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.
- We could also choose a much more compact binary encoding, giving $n = \lfloor \lg k \rfloor + 1$.
- In this case, running time is $\Theta(k) = \Theta(2^n)$ which is super-polynomial.

Which encoding to pick?

- We always pick concise encodings.

Which encoding to pick?

- We always pick concise encodings.
- In particular, numbers are represented in binary, not unary.

Which encoding to pick?

- We always pick concise encodings.
- In particular, numbers are represented in binary, not unary.
- Thus, we regard the algorithm on the previous slide as having super-polynomial $O(2^k)$ time.

Languages

- *Alphabet*: finite set Σ of symbols.

Languages

- *Alphabet*: finite set Σ of symbols.
- *Language* L over Σ: a set of strings of symbols from Σ.

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 $L = \{a, ba, cab, bbac, \ldots\}$.

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.

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 $\emptyset$.

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 $\emptyset$.
- Σ^* denotes the language of all strings (including ϵ).

Languages and decision problems

- Recall that we encode instances of a decision problem as binary strings.

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 to $\Sigma = \{0, 1\}$.

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 to $\Sigma = \{0, 1\}$.
- Q can be specified by the binary strings that encode yes-instances of the problem.

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 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:

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

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 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, given input x.

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, given input x.
- A *accepts* a string x if $A(x) = 1$.

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, given input x.
- A *accepts* a string x if $A(x) = 1$.
- A *rejects* a string x if $A(x) = 0$.

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, given 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.

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, given 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:

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

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, given 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:

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

- If all strings not in L are rejected by A, i.e., $A(x) = 0$ for all $x \in \{0, 1\}^* \setminus L$, then L is *decided* by A.

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, given 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:

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

- If all strings not in L are rejected by A, i.e., $A(x) = 0$ for all $x \in \{0, 1\}^* \setminus L$, then 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.

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.

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.

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 have:

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

P in terms of acceptance

- Lemma:

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

P in terms of acceptance

- Lemma:

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

- $\subseteq$: straightforward.

P in terms of acceptance

- Lemma:

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

- $\subseteq$: 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'.

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 string from L.

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 string from L.
- A' simulates A for at most cn^k steps.

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 string from L.
- A' simulates A for at most cn^k steps.
- If the simulation has not halted after this many steps, A' halts and outputs 0.

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 string from L.
- A' simulates A for at most cn^k steps.
- If the simulation has not halted after this many steps, A' halts and outputs 0.
- Then A' decides L and runs in polynomial time.

Verification

- Let L be a language.

Verification

- Let L be a language.
- We might not have an efficient algorithm that accepts L.

Verification

- Let L be a language.
- We might not have an efficient algorithm that accepts L.
- Consider instead an algorithm A taking two parameters, $x, c \in \Sigma^*$.

Verification

- Let L be a language.
- We might not have an efficient algorithm that accepts L.
- Consider instead 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.

Verification

- Let L be a language.
- We might not have an efficient algorithm that accepts L.
- Consider instead 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.
- What is simpler - solving an exercise yourself or checking that a given solution is correct?

The `HAM-CYCLE` problem

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

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 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 HAM-CYCLE can be decided in polynomial time.

Verifying `HAM-CYCLE`

- Consider instead an algorithm A_{ham} taking two parameters, $\langle G \rangle$ and $\langle C \rangle$.

Verifying HAM-CYCLE

- Consider instead an algorithm A_{ham} taking two parameters, $\langle G \rangle$ and $\langle C \rangle$.
- A_{ham} checks that G is undirected and that C is a simple cycle containing every vertex of G.

Verifying HAM-CYCLE

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

Verifying `HAM-CYCLE`

- Consider instead an algorithm A_{ham} taking two parameters, $\langle G \rangle$ and $\langle C \rangle$.
- A_{ham} checks that G is undirected and that C is a simple cycle containing every vertex of G.
- If so, A_{ham} outputs 1, otherwise 0.
- Designing A_{ham} to run in polynomial time is easy.

Verifying a language

- A *verification algorithm* is an algorithm A taking two arguments, $x, y \in \{0,1\}^*$, where y is the *certificate*.

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 $A(x, y) = 1$.

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

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

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.

The complexity class NP

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

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

The complexity class NP

- NP is the class of languages that can be verified in polynomial time.
- In other words, $L \in$ 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 `HAM-CYCLE` $\in$ NP.

The complexity class NP

- NP is the class of languages that can be verified in polynomial time.
- In other words, $L \in$ 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 `HAM-CYCLE` $\in$ NP.
- Furthermore, if $L \in$ P then $L \in$ NP.

The complexity class NP

- NP is the class of languages that can be verified in polynomial time.
- In other words, $L \in$ 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 `HAM-CYCLE` $\in$ NP.
- Furthermore, if $L \in$ P then $L \in$ NP.
- Hence, P $\subseteq$ NP.

The complexity class NP

- NP is the class of languages that can be verified in polynomial time.
- In other words, $L \in$ 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 `HAM-CYCLE` $\in$ NP.
- Furthermore, if $L \in$ P then $L \in$ NP.
- 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 $\overline{L} \in \text{NP}$.

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?

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 $\overline{\texttt{HAM-CYCLE}} \in$ 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 $\overline{\texttt{HAM-CYCLE}} \in$ NP?
- In other words, given a graph, can we easily verify that it does *not* have a simple cycle containing every vertex of G?

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 $\overline{\texttt{HAM-CYCLE}} \in$ NP?
- In other words, given a graph, can we easily verify that it does *not* have a simple cycle containing every vertex of G?
- It is open whether NP $=$ co-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 $\overline{\texttt{HAM-CYCLE}} \in$ NP?
- In other words, given a graph, can we easily verify that it does *not* have a simple cycle containing every vertex of G?
- 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.

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.

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*.

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*.
- `HAM-CYCLE` is NP-complete.

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*.
- `HAM-CYCLE` is NP-complete.
- Hence, if we could show $\texttt{HAM-CYCLE} \in \mathrm{P}$ then $\mathrm{P} = \mathrm{NP}$.

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*.
- `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.

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

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

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

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

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

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

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

- 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.

Reducibility

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

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

- 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.
- This follows since any instance I_1 of L_1 can be solved by transforming it in polynomial time to an instance I_2 of L_2 and then solving I_2.

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

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

- 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.
- This follows since any instance I_1 of L_1 can be solved by transforming it in polynomial time to an instance I_2 of L_2 and then solving I_2.
- Hence,

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

NP-complete languages

- Language L is *NP-complete* if

NP-complete languages

- Language L is *NP-complete* if

1. $L \in$ NP and

NP-complete languages

- Language L is *NP-complete* if
  1. $L \in$ NP and
  2. $L' \leq_P L$ for every $L' \in$ NP.

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 L satisfies property 2 (and possibly not property 1).

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 L satisfies property 2 (and possibly not property 1).
- The class of NP-complete languages is denoted NPC.

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 L satisfies property 2 (and possibly not property 1).
- The class of NP-complete languages is denoted NPC.
- We have that if any language of NPC belongs to P then P $=$ NP.

Circuit satisfiability

- A *boolean combinational circuit* consists of a collection of logic gates connected together with wires.

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.

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.

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.

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.

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.

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

- 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:

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

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:

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

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

Showing CIRCUIT-SAT $\in$ NP

- We construct algorithm A with inputs x and y.

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.

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

- 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.

Showing `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.

Showing `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.

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

- 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.

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, $\texttt{CIRCUIT-SAT} \in$ NP.

Showing that `CIRCUIT-SAT` is NP-hard

- Consider any language $L \in \mathrm{NP}$.

Showing that CIRCUIT-SAT is NP-hard

- Consider any language $L \in$ NP.
- We need to give a polynomial-time reduction from L to CIRCUIT-SAT.

Showing that CIRCUIT-SAT is NP-hard

- Consider any language $L \in$ NP.
- We need to give a polynomial-time reduction from L to 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

- We know that there is a polynomial-time algorithm A such that

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

Showing that CIRCUIT-SAT is NP-hard

- We know that 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 a string x, the idea is to build a circuit $C(x)$ with $|y|$ input wires.

Showing that CIRCUIT-SAT is NP-hard

- We know that 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 a string x, the idea is to build a circuit $C(x)$ with $|y|$ input wires.
- $C(x)$ should have a satisfying assignment if and only if $A(x,y) = 1$ for some y with $|y| = O(|x|^c)$.

Showing that CIRCUIT-SAT is NP-hard

- We know that 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 a string x, the idea is to build a circuit $C(x)$ with $|y|$ input wires.
- $C(x)$ should have a satisfying assignment if and only if $A(x,y) = 1$ for some y with $|y| = O(|x|^c)$.
- This way,

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

Showing that `CIRCUIT-SAT` is NP-hard

- There is a constant k such that the worst-case running time $T(n)$ of A is $O(n^k)$, where $n = |x|$.

Showing that `CIRCUIT-SAT` is NP-hard

- There is a constant k such that the worst-case running time $T(n)$ of A is $O(n^k)$, where $n = |x|$.
- The machine executing A has a certain *configuration* at each time step.

Showing that `CIRCUIT-SAT` is NP-hard

- There is a constant k such that the worst-case running time $T(n)$ of A 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 memory, CPU and so on.

Showing that `CIRCUIT-SAT` is NP-hard

- There is a constant k such that the worst-case running time $T(n)$ of A 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 memory, CPU and so on.
- When executing A, the machine goes through a series of configurations $c_0, c_1, \ldots, c_{T(n)}$.

Showing that `CIRCUIT-SAT` is NP-hard

- There is a constant k such that the worst-case running time $T(n)$ of A 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 memory, CPU and so on.
- When executing A, the machine goes through a series of configurations $c_0, c_1, \ldots, c_{T(n)}$.
- Configuration c_0 specifies the inputs x and y and the program code for A.

Showing that `CIRCUIT-SAT` is NP-hard

- There is a constant k such that the worst-case running time $T(n)$ of A 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 memory, CPU and so on.
- When executing A, the machine goes through a series of configurations $c_0, c_1, \ldots, c_{T(n)}$.
- Configuration c_0 specifies the inputs x and y and the program code for A.
- 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.

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.

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.

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.

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.

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.

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.
- The size of the circuit is still polynomial in n, however.

Showing that `CIRCUIT-SAT` is NP-hard

- We modify the circuit so that the input corresponding to x is specified and so that the only output wire is that corresponding to the output of A.

Showing that `CIRCUIT-SAT` is NP-hard

- We modify the circuit so that the input corresponding to x is specified and so that the only output wire is that corresponding to the output of A.
- The resulting circuit $C(x)$ has a satisfying assignment y if and only if $A(x, y) = 1$.

Showing that `CIRCUIT-SAT` is NP-hard

- We modify the circuit so that the input corresponding to x is specified and so that the only output wire is that corresponding to the output of A.
- The resulting circuit $C(x)$ has a satisfying assignment y if and only if $A(x, y) = 1$.
- This shows that $L \leq_P$ `CIRCUIT-SAT`.

Showing that `CIRCUIT-SAT` is NP-hard

- We modify the circuit so that the input corresponding to x is specified and so that the only output wire is that corresponding to the output of A.
- The resulting circuit $C(x)$ has a satisfying assignment y if and only if $A(x,y) = 1$.
- This shows that $L \leq_P$ `CIRCUIT-SAT`.
- Thus, `CIRCUIT-SAT` is NP-hard.

Showing that CIRCUIT-SAT is NP-hard

- We modify the circuit so that the input corresponding to x is specified and so that the only output wire is that corresponding to the output of A.
- The resulting circuit $C(x)$ has a satisfying assignment y if and only if $A(x, y) = 1$.
- This shows that $L \leq_P$ CIRCUIT-SAT.
- Thus, CIRCUIT-SAT is NP-hard.
- Since also CIRCUIT-SAT $\in$ NP, it follows that CIRCUIT-SAT is NP-complete.

Plan for next lecture

- Showing NP-completeness of other problems using polynomial-time reductions:

Plan for next lecture

- Showing NP-completeness of other problems using polynomial-time reductions:
  - SAT

Plan for next lecture

- Showing NP-completeness of other problems using polynomial-time reductions:
  - `SAT`
  - `3-CNF-SAT`

Plan for next lecture

- Showing NP-completeness of other problems using polynomial-time reductions:
  - `SAT`
  - `3-CNF-SAT`
  - `CLIQUE`

Plan for next lecture

- Showing NP-completeness of other problems using polynomial-time reductions:
  - `SAT`
  - `3-CNF-SAT`
  - `CLIQUE`
  - `VERTEX-COVER`

Plan for next lecture

- Showing NP-completeness of other problems using polynomial-time reductions:
  - `SAT`
  - `3-CNF-SAT`
  - `CLIQUE`
  - `VERTEX-COVER`
  - `HAM-CYCLE`

Plan for next lecture

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

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.

Showing NP-completeness using reductions

- Suppose L' is an NP-complete language.
- Consider another language L.

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.

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.
- If also $L \in$ NP then L is NP-complete.

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.
- If also $L \in$ NP then L is NP-complete.
- Next time we show:

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

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.
- If also $L \in$ 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.