



Design and analysis of algorithms

Lecture 31 & 32 & 33

Edyta Szymańska

edyta@cc.gatech.edu

Tractable and Intractable Problems

So far so good:

*almost all problems that we have studied have had **polynomial** running times, i.e. their running time was $T(n) = O(n^k)$ for some constant k .*

Tractable and Intractable Problems

So far so good:

*almost all problems that we have studied have had **polynomial** running times, i.e. their running time was $T(n) = O(n^k)$ for some constant k . Typically k was small, 3 or less.*

Tractable and Intractable Problems

So far so good:

*almost all problems that we have studied have had **polynomial** running times, i.e. their running time was $T(n) = O(n^k)$ for some constant k .*

Typically k was small, 3 or less.

Let \mathbf{P} denote the class of all problems whose solution can be computed in polynomial time, i.e. $O(n^k)$, for some fixed k .

Tractable and Intractable Problems

So far so good:

*almost all problems that we have studied have had **polynomial** running times, i.e. their running time was $T(n) = O(n^k)$ for some constant k .*

Typically k was small, 3 or less.

Let \mathbf{P} denote the class of all problems whose solution can be computed in polynomial time, i.e. $O(n^k)$, for some fixed k .

*We consider all problems in \mathbf{P} efficiently solvable or **tractable**.*

Tractable and Intractable Problems

So far so good:

*almost all problems that we have studied have had **polynomial** running times, i.e. their running time was $T(n) = O(n^k)$ for some constant k .*

Typically k was small, 3 or less.

Let \mathbf{P} denote the class of all problems whose solution can be computed in polynomial time, i.e. $O(n^k)$, for some fixed k .

We consider all problems in \mathbf{P} efficiently solvable or *tractable*.

A problem is considered *intractable* if it is not solvable in polynomial time.

NP-Complete Problems

Properties of **NP**-complete problems

- ⑥ known methods for solving these problems have *exponential* running time, i.e. $O(2^{n^k})$ for some k
- ⑥ if we could solve one **NP**-complete problem in polynomial time, then there is a way to solve *every* **NP**-complete problem in polynomial time.

Why study NP-Complete Problems ?

Recognizing an NP-complete problem is helpful:

Why study NP-Complete Problems ?

Recognizing an NP-complete problem is helpful:

- ⑥ you can use a known algorithm for it, and accept the fact that it will take a long time to solve it if n is large;

Why study NP-Complete Problems ?

Recognizing an NP-complete problem is helpful:

- ⑥ you can use a known algorithm for it, and accept the fact that it will take a long time to solve it if n is large;
- ⑥ you can settle for *approximating* the solution, i.e. finding a nearly best solution rather than the optimum

Why study NP-Complete Problems ?

Recognizing an NP-complete problem is helpful:

- ⑥ you can use a known algorithm for it, and accept the fact that it will take a long time to solve it if n is large;
- ⑥ you can settle for *approximating* the solution, i.e. finding a nearly best solution rather than the optimum
- ⑥ you can reformulate your problem so that it is in **P**, for example by restricting to work only on a subset of simpler problems.

Why study NP-Complete Problems ?

Recognizing an NP-complete problem is helpful:

- ⑥ you can use a known algorithm for it, and accept the fact that it will take a long time to solve it if n is large;
- ⑥ you can settle for *approximating* the solution, i.e. finding a nearly best solution rather than the optimum
- ⑥ you can reformulate your problem so that it is in **P**, for example by restricting to work only on a subset of simpler problems.

To understand (solve?) the most famous open problem in computer science: "**P=NP ?**"

Why study NP-Complete Problems ?

Recognizing an NP-complete problem is helpful:

- ⑥ you can use a known algorithm for it, and accept the fact that it will take a long time to solve it if n is large;
- ⑥ you can settle for *approximating* the solution, i.e. finding a nearly best solution rather than the optimum
- ⑥ you can reformulate your problem so that it is in **P**, for example by restricting to work only on a subset of simpler problems.

To understand (solve?) the most famous open problem in computer science: "**P=NP ?**"

NP and NP-Complete Problems (still informally)

Class **NP** : the class of problems where, if someone hands you a potential solution, then you can *check* whether it is a solution in polynomial time.

NP and NP-Complete Problems (still informally)

Class **NP** : the class of problems where, if someone hands you a potential solution, then you can *check* whether it is a solution in polynomial time.

Example: Does a graph have a cycle on n vertices ?

NP and NP-Complete Problems (still informally)

Class **NP** : the class of problems where, if someone hands you a potential solution, then you can *check* whether it is a solution in polynomial time.

Example: Does a graph have a cycle on n vertices ?
Observe that **P** \subseteq **NP**.

NP and NP-Complete Problems (still informally)

Class **NP** : the class of problems where, if someone hands you a potential solution, then you can *check* whether it is a solution in polynomial time.

Example: Does a graph have a cycle on n vertices ?
Observe that **P** \subseteq **NP**.

Intuition: checking the validity of a solution is easier than coming up with a solution

NP and NP-Complete Problems (still informally)

Class **NP** : the class of problems where, if someone hands you a potential solution, then you can *check* whether it is a solution in polynomial time.

Example: Does a graph have a cycle on n vertices ?

Observe that $\mathbf{P} \subseteq \mathbf{NP}$.

Intuition: checking the validity of a solution is easier than coming up with a solution

NP-complete \subseteq **NP**

NP and NP-Complete Problems (still informally)

Class **NP** : the class of problems where, if someone hands you a potential solution, then you can *check* whether it is a solution in polynomial time.

Example: Does a graph have a cycle on n vertices ?
Observe that **$P \subseteq NP$** .

Intuition: checking the validity of a solution is easier than coming up with a solution
 $NP\text{-complete} \subseteq NP$

Decision Problems

We will only consider problems with YES/NO answers.

Decision Problems

We will only consider problems with YES/NO answers.

Example: Travelling Salesman Problem (TSP) on
 $G = (V, E, w)$, w -nonnegative integer edge weights

Decision Problems

We will only consider problems with YES/NO answers.

Example: Travelling Salesman Problem (TSP) on $G = (V, E, w)$, w -nonnegative integer edge weights

1. Given $G = (V, E, w)$, what is the minimum length cycle that visits each node exactly once ? *optimization problem*

Decision Problems

We will only consider problems with YES/NO answers.

Example: Travelling Salesman Problem (TSP) on $G = (V, E, w)$, w -nonnegative integer edge weights

1. Given $G = (V, E, w)$, what is the minimum length cycle that visits each node exactly once ? *optimization problem*
2. Given $G = (V, E, w)$ and an integer K , is there a cycle that visits each node exactly once, with weight at most K ? *decision problem*

Decision Problems

We will only consider problems with YES/NO answers.

Example: Travelling Salesman Problem (TSP) on $G = (V, E, w)$, w -nonnegative integer edge weights

1. Given $G = (V, E, w)$, what is the minimum length cycle that visits each node exactly once ? *optimization problem*
2. Given $G = (V, E, w)$ and an integer K , is there a cycle that visits each node exactly once, with weight at most K ? *decision problem*

If we show that Question 2 is *NP – complete* then that means that Question 1 is at least as hard.

Decision Problems

We will only consider problems with YES/NO answers.

Example: Travelling Salesman Problem (TSP) on $G = (V, E, w)$, w -nonnegative integer edge weights

1. Given $G = (V, E, w)$, what is the minimum length cycle that visits each node exactly once ? *optimization problem*
2. Given $G = (V, E, w)$ and an integer K , is there a cycle that visits each node exactly once, with weight at most K ? *decision problem*

If we show that Question 2 is *NP – complete* then that means that Question 1 is at least as hard.

Textbook setting

Formally:

Problems = Languages over a finite alphabet

Textbook setting

Formally:

Problems = Languages over a finite alphabet

Alphabet: nonempty, finite set of symbols. Usually,
 $\Sigma = \{0, 1\}$.

Textbook setting

Formally:

Problems = Languages over a finite alphabet

Alphabet: nonempty, finite set of symbols. Usually,

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

Σ^* - the set of all possible strings over the alphabet Σ .

Textbook setting

Formally:

Problems = Languages over a finite alphabet

Alphabet: nonempty, finite set of symbols. Usually,

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

Σ^* - the set of all possible strings over the alphabet Σ .

Language $L = \{x \in \Sigma^*\}$, any subset of Σ^* , for example

$L = \{0, 01, 011, 01110, \dots\}$.

Textbook setting

Formally:

Problems = Languages over a finite alphabet

Alphabet: nonempty, finite set of symbols. Usually,

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

Σ^* - the set of all possible strings over the alphabet Σ .

Language $L = \{x \in \Sigma^*\}$, any subset of Σ^* , for example

$L = \{0, 01, 011, 01110, \dots\}$.

Algorithm A **decides** language L , if A **accepts**

$(A(x) = YES/1)$ all strings $x \in L$ and **rejects** $(A(x) = NO/0)$

all strings $x \notin L$.

Textbook setting

Formally:

Problems = Languages over a finite alphabet

Alphabet: nonempty, finite set of symbols. Usually,

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

Σ^* - the set of all possible strings over the alphabet Σ .

Language $L = \{x \in \Sigma^*\}$, any subset of Σ^* , for example

$L = \{0, 01, 011, 01110, \dots\}$.

Algorithm A **decides** language L , if A **accepts**

$(A(x) = YES/1)$ all strings $x \in L$ and **rejects** $(A(x) = NO/0)$

all strings $x \notin L$.

Solving a decision problem in polynomial time = deciding the corresponding language in polynomial time.

Textbook setting

Complexity class: the set of languages(problems) which have a similar complexity measure (running time of an algorithm)

Textbook setting

Complexity class: the set of languages(problems) which have a similar complexity measure (running time of an algorithm)

Class $P = \{L \subseteq \Sigma^* :$

\exists an algorithm A that decides L in polynomial time}

Reductions

Let A and B be two decision problems.

Reductions

Let A and B be two decision problems.

A reduction from A to B is a polynomial time algorithm R which transforms every input of A to an equivalent input of B .

Reductions

Let A and B be two decision problems.

A reduction from A to B is a polynomial time algorithm R which transforms every input of A to an equivalent input of B .

In other words, R is a reduction from A to B if $\forall x$, we have $A(x) = B(R(x))$.

Reductions

Let A and B be two decision problems.

A reduction from A to B is a polynomial time algorithm R which transforms every input of A to an equivalent input of B .

In other words, R is a reduction from A to B if $\forall x$, we have $A(x) = B(R(x))$.

A reduction R from A to B , together with a polynomial time algorithm for B , constitute a polynomial algorithm for A .

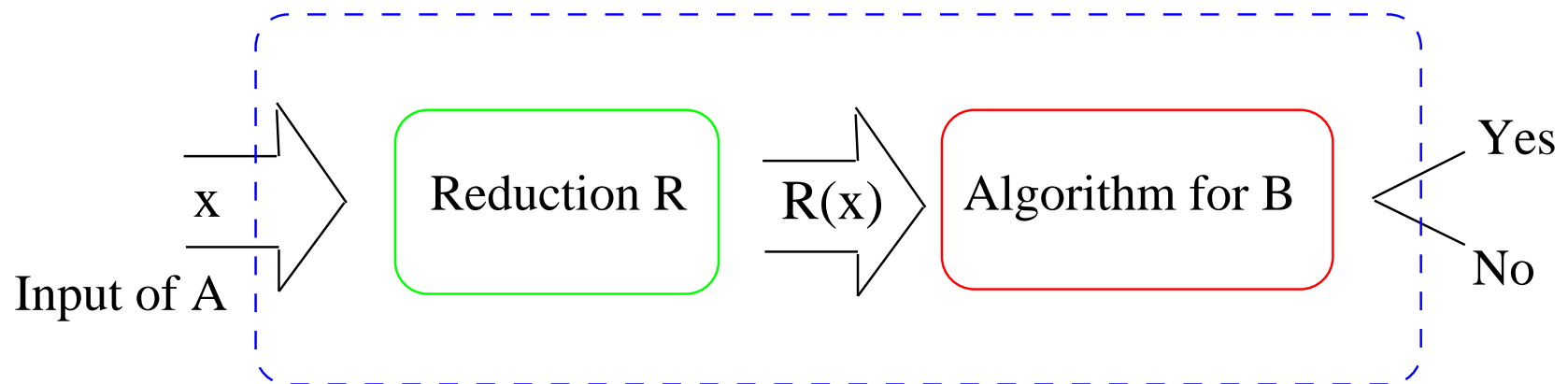
Reductions

Let A and B be two decision problems.

A reduction from A to B is a polynomial time algorithm R which transforms every input of A to an equivalent input of B .

In other words, R is a reduction from A to B if $\forall x$, we have $A(x) = B(R(x))$.

A reduction R from A to B , together with a polynomial time algorithm for B , constitute a polynomial algorithm for A .



Algorithm for A

Reductions

Example: bipartite matching to max-flow

Reductions

Example: bipartite matching to max-flow

If we reduce A to B , then it means that " A is no harder than B ."

Reductions

Example: bipartite matching to max-flow

If we reduce A to B , then it means that " A is no harder than B ."

Notation: $A \leq B$, which implies:

- ⌚ if B is easy then A is easy
- ⌚ if A is hard, then B is also hard.

NP and NP-Completeness

A problem A is in **NP** (*Nondeterministic Polynomial*) if there exists a polynomial p and a polynomial time algorithm V such that x is a YES-input for problem A if and only if there exists a solution y , with $length(y) \leq p(length(x))$ such that $V(x, y)$ outputs YES.

NP and NP-Completeness

A problem A is in **NP** (*Nondeterministic Polynomial*) if there exists a polynomial p and a polynomial time algorithm V such that x is a YES-input for problem A if and only if there exists a solution y , with $length(y) \leq p(length(x))$ such that $V(x, y)$ outputs YES.

Example: Hamiltonian cycle: Given a graph on n vertices, is there a cycle on n vertices ?

NP and NP-Completeness

A problem A is in **NP** (*Nondeterministic Polynomial*) if there exists a polynomial p and a polynomial time algorithm V such that x is a YES-input for problem A if and only if there exists a solution y , with $length(y) \leq p(length(x))$ such that $V(x, y)$ outputs YES.

Example: Hamiltonian cycle: Given a graph on n vertices, is there a cycle on n vertices ?

HAM-CYCLE \in **NP**

NP and NP-Completeness

A problem A is in **NP** (*Nondeterministic Polynomial*) if there exists a polynomial p and a polynomial time algorithm V such that x is a YES-input for problem A if and only if there exists a solution y , with $length(y) \leq p(length(x))$ such that $V(x, y)$ outputs YES.

Example: Hamiltonian cycle: Given a graph on n vertices, is there a cycle on n vertices ?

HAM-CYCLE \in NP

$V(x, y) : x$ is a hamiltonian graph G , y is a cycle C in G on n vertices.

NP and NP-Completeness

A problem A is in **NP** (*Nondeterministic Polynomial*) if there exists a polynomial p and a polynomial time algorithm V such that x is a YES-input for problem A if and only if there exists a solution y , with $length(y) \leq p(length(x))$ such that $V(x, y)$ outputs YES.

Example: Hamiltonian cycle: Given a graph on n vertices, is there a cycle on n vertices ?

HAM-CYCLE \in NP

$V(x, y)$: x is a hamiltonian graph G , y is a cycle C in G on n vertices.

Non-HAM-CYCLE: given a graph G , is it true that G does not contain an hamiltonian cycle?

NP and NP-Completeness

A problem A is in **NP** (*Nondeterministic Polynomial*) if there exists a polynomial p and a polynomial time algorithm V such that x is a YES-input for problem A if and only if there exists a solution y , with $length(y) \leq p(length(x))$ such that $V(x, y)$ outputs YES.

Example: Hamiltonian cycle: Given a graph on n vertices, is there a cycle on n vertices ?

HAM-CYCLE \in NP

$V(x, y)$: x is a hamiltonian graph G , y is a cycle C in G on n vertices.

Non-HAM-CYCLE: given a graph G , is it true that G does not contain an hamiltonian cycle?

Non-HAM-CYCLE \notin NP because there is no *polynomial certificate* y for non-hamiltonicity of G

NP and NP-Completeness

6 $P \subseteq NP$

NP and NP-Completeness

⑥ $P \subseteq NP$

⑥ We say that a problem A is **NP-hard** if for every problem $N \in NP$, N is reducible to A ($N \leq A$).

NP and NP-Completeness

⑥ **$P \subseteq NP$**

⑥ We say that a problem A is **NP-hard** if for every problem $N \in NP$, N is reducible to A ($N \leq A$).

⑥ We say that a problem A is **NP-complete** if A is **NP-hard** and $A \in NP$.

NP and NP-Completeness

⑥ $P \subseteq NP$

⑥ We say that a problem A is **NP-hard** if for every problem $N \in NP$, N is reducible to A ($N \leq A$).

⑥ We say that a problem A is **NP-complete** if A is **NP-hard** and $A \in NP$.

Lemma *If A is NP-complete, then A is in P if and only if $P=NP$.*

NP and NP-Completeness

Lemma *If A is NP-complete, then A is in P if and only if $P=NP$.*

NP and NP-Completeness

Lemma *If A is NP-complete, then A is in P if and only if $P=NP$.*

Thus, if we are dealing with a problem A that we can prove to be NP-complete, there are only two possibilities:

NP and NP-Completeness

Lemma *If A is NP-complete, then A is in P if and only if $P=NP$.*

Thus, if we are dealing with a problem A that we can prove to be NP-complete, there are only two possibilities:

- ⑥ A has no efficient algorithm
- ⑥ all the infinitely many problems in NP are in P.

NP and NP-Completeness

Lemma *If A is NP-complete, then A is in P if and only if $P=NP$.*

Thus, if we are dealing with a problem A that we can prove to be NP-complete, there are only two possibilities:

- ⑥ A has no efficient algorithm
- ⑥ all the infinitely many problems in NP are in P.

It is safe to assume that $P \neq NP$ (at least till the end of the semester.)

Some Problems in P, NP and NP-Complete Problems

- ⑥ **minimum spanning tree:** given $G = (V, E, w)$ and K , is there a spanning tree T in G such that $w(T) \leq K$?

Some Problems in P, NP and NP-Complete Problems

- ⑥ **minimum spanning tree:** given $G = (V, E, w)$ and K , is there a spanning tree T in G such that $w(T) \leq K$?
- ⑥ **travelling salesman problem:** given $G = (V, E, w)$ and K , is there a cycle C on n nodes of G such that $w(C) \leq K$?

Some Problems in P, NP and NP-Complete Problems

- ⑥ **minimum spanning tree:** given $G = (V, E, w)$ and K , is there a spanning tree T in G such that $w(T) \leq K$?
- ⑥ **travelling salesman problem:** given $G = (V, E, w)$ and K , is there a cycle C on n nodes of G such that $w(C) \leq K$?
- ⑥ **Eulerian graph:** given a graph, is there a cycle that visits each edge of the graph exactly once?

Some Problems in P, NP and NP-Complete Problems

- ⑥ **minimum spanning tree:** given $G = (V, E, w)$ and K , is there a spanning tree T in G such that $w(T) \leq K$?
- ⑥ **travelling salesman problem:** given $G = (V, E, w)$ and K , is there a cycle C on n nodes of G such that $w(C) \leq K$?
- ⑥ **Eulerian graph:** given a graph, is there a cycle that visits each edge of the graph exactly once?
- ⑥ **Hamiltonian graph:** given a graph, is there a cycle that visits each node of the graph exactly once?

Some Problems in P, NP and NP-Complete Problems

- ⑥ **circuit value:** given a Boolean circuit, and its inputs, is the output True?

Some Problems in P, NP and NP-Complete Problems

- ⑥ **circuit value:** given a Boolean circuit, and its inputs, is the output True?
- ⑥ **circuit SAT:** given a Boolean circuit, is there a way to set the inputs so that the output is True?

Some Problems in P, NP and NP-Complete Problems

- ⑥ **circuit value:** given a Boolean circuit, and its inputs, is the output True?
- ⑥ **circuit SAT:** given a Boolean circuit, is there a way to set the inputs so that the output is True?
- ⑥ **2SAT:** given a Boolean formula in **2-CNF**, is there a satisfying truth assignment to the input variables?

Some Problems in P, NP and NP-Complete Problems

- ⑥ **circuit value:** given a Boolean circuit, and its inputs, is the output True?
- ⑥ **circuit SAT:** given a Boolean circuit, is there a way to set the inputs so that the output is True?
- ⑥ **2SAT:** given a Boolean formula in **2-CNF**, is there a satisfying truth assignment to the input variables?
- ⑥ **3SAT:** given a Boolean formula in **3-CNF**, is there a satisfying truth assignment to the input variables?

Some Problems in P, NP and NP-Complete Problems

- ⑥ **matching:** given a bipartite graph, is there a perfect matching?

Some Problems in P, NP and NP-Complete Problems

- ⑥ **matching:** given a bipartite graph, is there a perfect matching?
- ⑥ **3D matching:** given a three-partite graph, is there a perfect matching (a set of disjoint triangles that covers all nodes)?

Some Problems in P, NP and NP-Complete Problems

- ⑥ **matching:** given a bipartite graph, is there a perfect matching?
- ⑥ **3D matching:** given a three-partite graph, is there a perfect matching (a set of disjoint triangles that covers all nodes)?
- ⑥ **unary subset-sum:** given integers a_1, \dots, a_n and K in unary, is there a subset of these integers that sum exactly to K ?

Some Problems in P, NP and NP-Complete Problems

- ⑥ **matching:** given a bipartite graph, is there a perfect matching?
- ⑥ **3D matching:** given a three-partite graph, is there a perfect matching (a set of disjoint triangles that covers all nodes)?
- ⑥ **unary subset-sum:** given integers a_1, \dots, a_n and K in unary, is there a subset of these integers that sum exactly to K ?
- ⑥ **subset sum** given integers a_1, \dots, a_n and K in binary, is there a subset of these integers that sum exactly to K ?

Circuit Satisfiability

Does there exist an NP-complete problem?

Circuit Satisfiability

Does there exist an NP-complete problem?

Generic NP-complete problem:

consider a **Boolean combinatorial Circuit:**

Circuit Satisfiability

Does there exist an NP-complete problem?

Generic NP-complete problem:

consider a **Boolean combinatorial Circuit:**

logic gates: **NOT, AND, OR** and wires.

Circuit Satisfiability

Does there exist an NP-complete problem?

Generic NP-complete problem:

consider a **Boolean combinatorial Circuit:**

logic gates: **NOT, AND, OR** and wires.

Assume k inputs from $\{0, 1\}$ and one output from $\{0, 1\}$.

Circuit Satisfiability

Does there exist an NP-complete problem?

Generic NP-complete problem:

consider a **Boolean combinatorial Circuit:**

logic gates: **NOT, AND, OR** and wires.

Assume k inputs from $\{0, 1\}$ and one output from $\{0, 1\}$.

see Example on board/in textbook

Circuit Satisfiability

Does there exist an NP-complete problem?

Generic NP-complete problem:

consider a **Boolean combinatorial Circuit:**

logic gates: **NOT, AND, OR** and wires.

Assume k inputs from $\{0, 1\}$ and one output from $\{0, 1\}$.

see Example on board/in textbook

A one-output boolean combinatorial circuit is **satisfiable** if it has a *satisfying assignment* (a truth assignment of boolean input values) that causes the output of the circuit to be 1.

Circuit Satisfiability

The **Circuit-Satisfiability (CSAT) Problem** is to determine whether a given boolean combinatorial circuit is satisfiable.

Circuit Satisfiability

The **Circuit-Satisfiability (CSAT) Problem** is to determine whether a given boolean combinatorial circuit is satisfiable.

Theorem (Cook, Levin (1971)) *The CSAT is NP-complete.*

Circuit Satisfiability

The **Circuit-Satisfiability (CSAT) Problem** is to determine whether a given boolean combinatorial circuit is satisfiable.

Theorem (Cook, Levin (1971)) *The CSAT is NP-complete.*

Proof (sketch):

Circuit Satisfiability

Theorem (Cook, Levin (1971)) *The CSAT is NP-complete.*

Proof (sketch):

(1) $\text{CSAT} \in \text{NP}$

Circuit Satisfiability

Theorem (Cook, Levin (1971)) *The CSAT is NP-complete.*

Proof (sketch):

(1) **CSAT** \in **NP**

easy, the algorithm V has two inputs: the description of the circuit C (polynomial in the number of inputs) and the sequence of boolean values x_1, \dots, x_n and $V(C, x_1, \dots, x_n) = C(x_1, \dots, x_n)$, it simulates/computes C on the given sequence.

Circuit Satisfiability

Theorem (Cook, Levin (1971)) *The CSAT is NP-complete.*

Proof (sketch):

(1) **CSAT** \in **NP**

easy, the algorithm V has two inputs: the description of the circuit C (polynomial in the number of inputs) and the sequence of boolean values x_1, \dots, x_n and $V(C, x_1, \dots, x_n) = C(x_1, \dots, x_n)$, it simulates/computes C on the given sequence.

(2) **CSAT** is **NP-hard**, i.e. every decision problem $A \in \text{NP}$ reduces to CSAT in polynomial time

CSAT is NP-hard

Idea:



CSAT is NP-hard

Idea:

- ⑥ Let A be a problem in NP. $\Rightarrow A$ has a polynomial-time verifier V

CSAT is NP-hard

Idea:

- ⑥ Let A be a problem in **NP**. $\Rightarrow A$ has a polynomial-time verifier V
- ⑥ Convert the computation of V on an instance x of A and a solution $y : |y| \leq p(|x|)$ into a Nondeterministic Turing Machine M (a sequence of configurations)

CSAT is NP-hard

Idea:

- ⑥ Let A be a problem in **NP**. \Rightarrow A has a polynomial-time verifier V
- ⑥ Convert the computation of V on an instance x of A and a solution $y : |y| \leq p(|x|)$ into a Nondeterministic Turing Machine M (a sequence of configurations)
- ⑥ \Rightarrow A is decided by M

CSAT is NP-hard

Idea:

- ⑥ Let A be a problem in **NP**. $\Rightarrow A$ has a polynomial-time verifier V
- ⑥ Convert the computation of V on an instance x of A and a solution $y : |y| \leq p(|x|)$ into a Nondeterministic Turing Machine M (a sequence of configurations)
- ⑥ $\Rightarrow A$ is decided by M
- ⑥ Convert M to a boolean circuit

CSAT is NP-hard

Idea:

- ⑥ Let A be a problem in **NP**. $\Rightarrow A$ has a polynomial-time verifier V
- ⑥ Convert the computation of V on an instance x of A and a solution $y : |y| \leq p(|x|)$ into a Nondeterministic Turing Machine M (a sequence of configurations)
- ⑥ $\Rightarrow A$ is decided by M
- ⑥ Convert M to a boolean circuit

Proving NP-completeness

After establishing that **CSAT** is **NP-complete**, it becomes easier for other problems.

Proving NP-completeness

After establishing that **CSAT** is **NP-complete**, it becomes easier for other problems.

Tools: reduction and the following lemma.

Proving NP-completeness

After establishing that **CSAT** is **NP-complete**, it becomes easier for other problems.

Tools: reduction and the following lemma.

Lemma *Let B be an NP-complete problem and let A be a problem in NP. If we can prove that B reduces to A , then A is NP-complete.*

Proving NP-completeness

After establishing that **CSAT** is **NP-complete**, it becomes easier for other problems.

Tools: reduction and the following lemma.

Lemma *Let B be an NP-complete problem and let A be a problem in NP. If we can prove that B reduces to A , then A is NP-complete.*

Proof:

Proving NP-completeness

After establishing that **CSAT** is **NP-complete**, it becomes easier for other problems.

Tools: reduction and the following lemma.

Lemma *Let B be an NP-complete problem and let A be a problem in NP. If we can prove that B reduces to A , then A is NP-complete.*

Proof:

Reduction is transitive, i.e. $A \leq B$ and $B \leq C$ then $A \leq C$, which means that A is reducible to C in polynomial time.

Proving NP-completeness

After establishing that **CSAT** is **NP-complete**, it becomes easier for other problems.

Tools: reduction and the following lemma.

Lemma *Let B be an NP-complete problem and let A be a problem in NP. If we can prove that B reduces to A , then A is NP-complete.*

Proof:

Reduction is transitive, i.e. $A \leq B$ and $B \leq C$ then $A \leq C$, which means that A is reducible to C in polynomial time.

The lemma follows by the supposition and transitivity.

Recipe for Proving NP-completeness

1. Prove that $A \in \text{NP}$.

Recipe for Proving NP-completeness

1. Prove that $A \in \text{NP}$.
2. Select a known **NP-complete** problem B .

Recipe for Proving NP-completeness

1. Prove that $A \in \text{NP}$.
2. Select a known **NP-complete** problem B .
3. Describe a reduction algorithm R from B to A

Recipe for Proving NP-completeness

1. Prove that $A \in \text{NP}$.
2. Select a known **NP-complete** problem B .
3. Describe a reduction algorithm R from B to A
4. Prove that R runs in polynomial time.

Recipe for Proving NP-completeness

1. Prove that $A \in \text{NP}$.
2. Select a known **NP-complete** problem B .
3. Describe a reduction algorithm R from B to A
4. Prove that R runs in polynomial time.
5. Conclude A is **NP-complete**

Boolean Satisfiability (SAT Problem)

Boolean variables: $x_i, x_i \in \{0, 1\}$

connectives: \wedge, \vee, \neg

we will use $\neg x_i := \bar{x}_i$

x_i and \bar{x}_i are called literals

Boolean Satisfiability (SAT Problem)

Boolean variables: $x_i, x_i \in \{0, 1\}$

connectives: \wedge, \vee, \neg

we will use $\neg x_i := \bar{x}_i$

x_i and \bar{x}_i are called literals

CNF(Conjunctive normal form)

$C_1 \wedge C_2 \wedge \dots \wedge C_l$, C_i is a disjunction of literals

Boolean Satisfiability (SAT Problem)

Boolean variables: $x_i, x_i \in \{0, 1\}$

connectives: \wedge, \vee, \neg

we will use $\neg x_i := \bar{x}_i$

x_i and \bar{x}_i are called literals

CNF(Conjunctive normal form)

$C_1 \wedge C_2 \wedge \dots \wedge C_l$, C_i is a disjunction of literals

Note: *one variable may appear more than once within a C_i , and C_i 's might be duplicated*

Boolean Satisfiability (SAT Problem)

Boolean variables: $x_i, x_i \in \{0, 1\}$

connectives: \wedge, \vee, \neg

we will use $\neg x_i := \bar{x}_i$

x_i and \bar{x}_i are called literals

CNF(Conjunctive normal form)

$C_1 \wedge C_2 \wedge \dots \wedge C_l$, C_i is a disjunction of literals

Note: *one variable may appear more than once within a C_i , and C_i 's might be duplicated*

Example: ϕ -boolean formula in CNF

$$\phi_1 = (x_1 \vee x_3 \vee x_4) \wedge (\bar{x}_1 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_4 \vee \bar{x}_2) \wedge \bar{x}_3 \wedge (x_2 \vee \bar{x}_4)$$

$$\phi_2 = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (\bar{x}_3 \vee x_1) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

Boolean Satisfiability (SAT Problem)

Boolean variables: $x_i, x_i \in \{0, 1\}$

connectives: \wedge, \vee, \neg

we will use $\neg x_i := \bar{x}_i$

x_i and \bar{x}_i are called literals

CNF(Conjunctive normal form)

$C_1 \wedge C_2 \wedge \dots \wedge C_l$, C_i is a disjunction of literals

Note: *one variable may appear more than once within a C_i , and C_i 's might be duplicated*

Example: ϕ -boolean formula in CNF

$$\phi_1 = (x_1 \vee x_3 \vee x_4) \wedge (\bar{x}_1 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_4 \vee \bar{x}_2) \wedge \bar{x}_3 \wedge (x_2 \vee \bar{x}_4)$$

$$\phi_2 = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (\bar{x}_3 \vee x_1) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

ϕ is **satisfiable** if some assignment of 0's and 1's evaluates it to 1.

Boolean Satisfiability (SAT Problem)

Boolean variables: $x_i, x_i \in \{0, 1\}$

connectives: \wedge, \vee, \neg

we will use $\neg x_i := \bar{x}_i$

x_i and \bar{x}_i are called literals

CNF(Conjunctive normal form)

$C_1 \wedge C_2 \wedge \dots \wedge C_l$, C_i is a disjunction of literals

Note: *one variable may appear more than once within a C_i , and C_i 's might be duplicated*

Example: ϕ -boolean formula in CNF

$$\phi_1 = (x_1 \vee x_3 \vee x_4) \wedge (\bar{x}_1 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_4 \vee \bar{x}_2) \wedge \bar{x}_3 \wedge (x_2 \vee \bar{x}_4)$$

$$\phi_2 = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (\bar{x}_3 \vee x_1) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

ϕ is **satisfiable** if some assignment of 0's and 1's evaluates it to 1. **SAT Problem:** Is the given formula ϕ satisfiable ?

Boolean Satisfiability (SAT and 3SAT)

Theorem *SAT is NP-complete.*

Boolean Satisfiability (SAT and 3SAT)

Theorem *SAT is NP-complete.*

Proof: A reduction from CSAT.(will not prove it.)

Boolean Satisfiability (SAT and 3SAT)

Theorem *SAT is NP-complete.*

Proof: A reduction from CSAT.(will not prove it.)

SAT is used to prove that other problems are NP-complete.

Boolean Satisfiability (SAT and 3SAT)

Theorem *SAT is NP-complete.*

Proof: A reduction from CSAT.(will not prove it.)

SAT is used to prove that other problems are **NP-complete**.

It is even more convenient to use a version of **SAT**, namely **3SAT**.

Boolean Satisfiability (SAT and 3SAT)

Theorem *SAT is NP-complete.*

Proof: A reduction from CSAT.(will not prove it.)

SAT is used to prove that other problems are **NP-complete**.

It is even more convenient to use a version of **SAT**, namely **3SAT**.

In **3SAT** all clauses have 3 literals, for example

$$(x_2 \vee \bar{x}_4 \vee x_5) \wedge (x_1 \vee \bar{x}_3 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee x_3 \vee x_5)$$

Boolean Satisfiability (SAT and 3SAT)

Theorem *SAT is NP-complete.*

Proof: A reduction from CSAT.(will not prove it.)

SAT is used to prove that other problems are **NP-complete**.

It is even more convenient to use a version of **SAT**, namely **3SAT**.

In **3SAT** all clauses have 3 literals, for example

$$(x_2 \vee \bar{x}_4 \vee x_5) \wedge (x_1 \vee \bar{x}_3 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee x_3 \vee x_5)$$

3SAT is NP-complete

Theorem *3SAT is NP-complete.*

3SAT is NP-complete

Theorem *3SAT is NP-complete.*

Proof: A reduction from CSAT.

3SAT is NP-complete

Theorem *3SAT is NP-complete.*

Proof: A reduction from CSAT.

3SAT \in **NP**

3SAT is NP-complete

Theorem *3SAT is NP-complete.*

Proof: A reduction from CSAT.

3SAT \in **NP**

We can verify in polynomial time that a proposed truth assignment satisfies the given set of clauses.

3SAT is NP-complete

Theorem *3SAT is NP-complete.*

Proof: A reduction from CSAT.

3SAT \in **NP**

We can verify in polynomial time that a proposed truth assignment satisfies the given set of clauses.

3SAT is NP-hard.

3SAT is NP-complete

Theorem *3SAT is NP-complete.*

Proof: A reduction from CSAT.

3SAT \in **NP**

We can verify in polynomial time that a proposed truth assignment satisfies the given set of clauses.

3SAT is NP-hard.

We will show it via a reduction from CSAT.

3SAT is NP-complete

Theorem *3SAT is NP-complete.*

Proof: A reduction from CSAT.

3SAT \in **NP**

We can verify in polynomial time that a proposed truth assignment satisfies the given set of clauses.

3SAT is NP-hard.

We will show it via a reduction from CSAT.

Idea of the reduction: 2 steps

3SAT is NP-complete

Theorem *3SAT is NP-complete.*

Proof: A reduction from CSAT.

3SAT \in NP

We can verify in polynomial time that a proposed truth assignment satisfies the given set of clauses.

3SAT is NP-hard.

We will show it via a reduction from CSAT.

Idea of the reduction: 2 steps

$K \in \text{CSAT} \Rightarrow \phi \in \text{SAT (with } |C_i| \leq 3, \forall i) \Rightarrow \phi' \in \text{3SAT}$
(with $|C_i| = 3, \forall i$)

3SAT is NP-hard

Consider an arbitrary boolean circuit K , and assume that every AND and OR gate has 2 inputs and NOT gate has 1 input.

3SAT is NP-hard

Consider an arbitrary boolean circuit K , and assume that every AND and OR gate has 2 inputs and NOT gate has 1 input.

\forall gate $v \in K$ associate

- ⊗ a variable x_v (to encode the truth value that K holds at v)
- ⊗ certain clauses, which depend on the gate

Reduction from CSAT to SAT

- ⑥ if v is a NOT gate of u , then $(x_v \vee x_u) \wedge (\bar{x}_v \vee \bar{x}_u)$
- ⑥ if v is the OR gate of u and w , then $(x_v \vee \bar{x}_u) \wedge (x_v \vee \bar{x}_w) \wedge (\bar{x}_v \vee x_u \vee x_w)$
- ⑥ if v is the AND gate of u and w , then $(\bar{x}_v \vee x_u) \wedge (\bar{x}_v \vee x_w) \wedge (x_v \vee \bar{x}_u \vee \bar{x}_w)$
- ⑥ if v is a source, then we have a single-variable clause x_v OR \bar{x}_v
- ⑥ if v is the output node, then we include the single-variable clause x_0

Reduction from CSAT to SAT

- ⑥ if v is a NOT gate of u , then $(x_v \vee x_u) \wedge (\bar{x}_v \vee \bar{x}_u)$
- ⑥ if v is the OR gate of u and w , then $(x_v \vee \bar{x}_u) \wedge (x_v \vee \bar{x}_w) \wedge (\bar{x}_v \vee x_u \vee x_w)$
- ⑥ if v is the AND gate of u and w , then $(\bar{x}_v \vee x_u) \wedge (\bar{x}_v \vee x_w) \wedge (x_v \vee \bar{x}_u \vee \bar{x}_w)$
- ⑥ if v is a source, then we have a single-variable clause x_v OR \bar{x}_v
- ⑥ if v is the output node, then we include the single-variable clause x_0

Reduction from CSAT to SAT

Proving the equivalence:

The circuit K is satisfiable if and only if the formula $\phi \in R(K)$ is satisfiable.

Reduction from CSAT to SAT

Proving the equivalence:

The circuit K is satisfiable if and only if the formula $\phi \in R(K)$ is satisfiable.

(\Rightarrow) If K is satisfiable, then ϕ is satisfied by the construction. (see example.)

Reduction from CSAT to SAT

Proving the equivalence:

The circuit K is satisfiable if and only if the formula $\phi \in R(K)$ is satisfiable.

- (\Rightarrow) If K is satisfiable, then ϕ is satisfied by the construction. (see example.)
- (\Leftarrow) If ϕ is satisfiable then K is satisfiable because all the variables corresponding to K 's inputs constitute a satisfying assignment. (gates are correctly set and $x_0 = 1$).

Reduction from SAT to 3SAT

We need for all C_i : $|C_i| = 3$.

Reduction from SAT to 3SAT

We need for all C_i : $|C_i| = 3$.

Create new variables

⑥ four variables z_1, z_2, z_3, z_4

⑥ need $z_1 = z_2 = 0$ and add the following clauses:

$$(\bar{z}_i \vee z_3 \vee z_4) \wedge (\bar{z}_i \vee \bar{z}_3 \vee z_4) \wedge (\bar{z}_i \vee z_3 \vee \bar{z}_4) \wedge (\bar{z}_i \vee \bar{z}_3 \vee \bar{z}_4) \quad \forall i = 1, 2.$$

Reduction from SAT to 3SAT

We need for all $C_i : |C_i| = 3$.

Create new variables

⑥ four variables z_1, z_2, z_3, z_4

⑥ need $z_1 = z_2 = 0$ and add the following clauses:

$$(\bar{z}_i \vee z_3 \vee z_4) \wedge (\bar{z}_i \vee \bar{z}_3 \vee z_4) \wedge (\bar{z}_i \vee z_3 \vee \bar{z}_4) \wedge (\bar{z}_i \vee \bar{z}_3 \vee \bar{z}_4) \quad \forall i = 1, 2.$$

Correctness:

The resulting 3SAT formula ϕ' is equivalent to $\phi \in \text{SAT}$.

Reduction from SAT to 3SAT

We need for all $C_i : |C_i| = 3$.

Create new variables

- ⑥ four variables z_1, z_2, z_3, z_4
- ⑥ need $z_1 = z_2 = 0$ and add the following clauses:
 $(\bar{z}_i \vee z_3 \vee z_4) \wedge (\bar{z}_i \vee \bar{z}_3 \vee z_4) \wedge (\bar{z}_i \vee z_3 \vee \bar{z}_4) \wedge (\bar{z}_i \vee \bar{z}_3 \vee \bar{z}_4) \quad \forall i = 1, 2.$

Correctness:

The resulting 3SAT formula ϕ' is equivalent to $\phi \in \text{SAT}$.

- ⑥ if $C = t$, where $t = x$ or $t = \bar{x}$, then $C' := (t \vee z_1 \vee z_2)$
- ⑥ if $C = t \vee t'$, then $C' := (t \vee t' \vee z_1)$