Teoria degli Algoritmi

Corso di Laurea Magistrale in Matematica Applicata
a.a. 2020-21

Gabriele Tolomei

Dipartimento di Informatica
Sapienza Università di Roma
tolomei@di.uniroma1.it

Lecture 7: NP Completeness

Table of Contents

Table of Contents

P vs. NP

- One important advance in the P vs. NP question came in the early 1970s thanks to Stephen Cook and Leonid Levin

P vs. *NP*

- One important advance in the *P* vs. *NP* question came in the early 1970s thanks to Stephen Cook and Leonid Levin
- They found that certain problems in *NP* whose complexity is related to that of the **entire** class

P vs. *NP*

- One important advance in the *P* vs. *NP* question came in the early 1970s thanks to Stephen Cook and Leonid Levin
- They found that certain problems in *NP* whose complexity is related to that of the **entire** class
- If a polynomial time algorithm exists for *any* of these problems, then **all** problems in *NP* would be solvable in polynomial time

P vs. *NP*

- One important advance in the *P* vs. *NP* question came in the early 1970s thanks to Stephen Cook and Leonid Levin
- They found that certain problems in *NP* whose complexity is related to that of the **entire** class
- If a polynomial time algorithm exists for *any* of these problems, then **all** problems in *NP* would be solvable in polynomial time
- These problems are called *NP*-**complete**

NP-complete Problems: Why Do We Care About Them?

- The existence of *NP*-complete problems are fundamental both from the theoretical and the practical perspective

NP-complete Problems: Why Do We Care About Them?

- The existence of *NP*-complete problems are fundamental both from the theoretical and the practical perspective
- In an attempt to prove that $P = NP$, one would "only" needs to find a polynomial time algorithm for an *NP*-complete problem

NP-complete Problems: Why Do We Care About Them?

- The existence of NP-complete problems are fundamental both from the theoretical and the practical perspective
- In an attempt to prove that $P = NP$, one would "only" needs to find a polynomial time algorithm for an NP-complete problem
- Vice versa, trying to prove $P \neq NP$ would require to find at least a problem that is in NP but not in P; in particular this is true for an NP-complete problem

NP-complete Problems: Why Do We Care About Them?

- The existence of *NP*-complete problems are fundamental both from the theoretical and the practical perspective
- In an attempt to prove that $P = NP$, one would "only" needs to find a polynomial time algorithm for an *NP*-complete problem
- Vice versa, trying to prove $P \neq NP$ would require to find at least a problem that is in *NP* but not in *P*; in particular this is true for an *NP*-complete problem
- On the practical side, even if we still don't know if $P \neq NP$, showing that a problem is *NP*-complete is a strong evidence of its non-polynomiality (as most people indeed think $P \neq NP$)

The First *NP*-complete Problem: Boolean Satisfiability

- The first *NP*-complete problem we present is called the **satisfiability problem**

The First *NP*-complete Problem: Boolean Satisfiability

- The first *NP*-complete problem we present is called the **satisfiability problem**
- Let's consider variables that can take on only **two** possible values: 1 (`TRUE`) or 0 (`FALSE`)

The First *NP*-complete Problem: Boolean Satisfiability

- The first *NP*-complete problem we present is called the **satisfiability problem**
- Let's consider variables that can take on only **two** possible values: 1 (`TRUE`) or 0 (`FALSE`)
- Those are normally called **boolean** or **binary** variables

The First *NP*-complete Problem: Boolean Satisfiability

- The first *NP*-complete problem we present is called the **satisfiability problem**
- Let's consider variables that can take on only **two** possible values: 1 (TRUE) or 0 (FALSE)
- Those are normally called **boolean** or **binary** variables
- On those variables, we define **3 operators**:
  - AND: $x \wedge y$
  - OR: $x \vee y$
  - NOT: $\neg x$ (also denoted as $\overline{x}$)

The First *NP*-complete Problem: Boolean Satisfiability

Just to remind how those operators work:

$0 \wedge 0 = 0$	$0 \vee 0 = 0$	$\overline{0} = 1$
$0 \wedge 1 = 0$	$0 \vee 1 = 1$	$\overline{1} = 0$
$1 \wedge 0 = 0$	$1 \vee 0 = 1$	
$1 \wedge 1 = 1$	$1 \vee 1 = 1$	

The First *NP*-complete Problem: Boolean Satisfiability

- A **boolean formula** is an expression containing boolean variables and the operators above, e.g.,:

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

The First *NP*-complete Problem: Boolean Satisfiability

- A **boolean formula** is an expression containing boolean variables and the operators above, e.g.,:

$$\phi = (\overline{x} \land y) \lor (x \land \overline{z})$$

- A boolean formula is **satisfiable** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1 (i.e., `TRUE`)

The First *NP*-complete Problem: Boolean Satisfiability

- A **boolean formula** is an expression containing boolean variables and the operators above, e.g.,:

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

- A boolean formula is **satisfiable** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1 (i.e., `TRUE`)
- The boolean formula ϕ of the example above is satisfiable because the assignment $x = 0$, $y = 1$, and $z = 0$ makes it `TRUE`

The First *NP*-complete Problem: Boolean Satisfiability

Definition (The Satisfiability Problem)

The Satisfiability Problem is to test whether a given boolean formula ϕ is satisfiable, namely whether it exists an assignment that satisfies it.

The First *NP*-complete Problem: Boolean Satisfiability

Definition (The Satisfiability Problem)

The Satisfiability Problem is to test whether a given boolean formula ϕ is satisfiable, namely whether it exists an assignment that satisfies it.
More formally, we define:

$$SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable boolean formula}\}$$

The First *NP*-complete Problem: Boolean Satisfiability

Definition (The Satisfiability Problem)

The Satisfiability Problem is to test whether a given boolean formula ϕ is satisfiable, namely whether it exists an assignment that satisfies it.
More formally, we define:

$$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable boolean formula}\}$$

We now introduce the **Cook-Levin theorem**, which links the complexity of *SAT* with that of **all** problems in *NP*

The Cook-Levin Theorem

Theorem (Cook-Levin)

$SAT \in P \Leftrightarrow P = NP$

The Cook-Levin Theorem

Theorem (Cook-Levin)

$SAT \in P \Leftrightarrow P = NP$

We (re-)introduce a technique that is crucial to prove the Cook-Levin theorem

Table of Contents

Polynomial Time Reducibility

- We already defined the concept of reducing one problem to another

Polynomial Time Reducibility

- We already defined the concept of reducing one problem to another
- When problem A reduces to problem B ($A \leq B$), a solution to B can be used to solve A

Polynomial Time Reducibility

- We already defined the concept of reducing one problem to another
- When problem A reduces to problem B ($A \leq B$), a solution to B can be used to solve A
- We now extend this idea by taking into account computational efficiency

Polynomial Time Reducibility

- We already defined the concept of reducing one problem to another
- When problem A reduces to problem B ($A \leq B$), a solution to B can be used to solve A
- We now extend this idea by taking into account computational efficiency
- Intuitively, when a problem A is **efficiently** reducible to a problem B, an efficient solution to B can be used to solve A also efficiently

Polynomial Time Reducibility

Definition (Polynomial Time Computable Function)

A function $f : \Sigma^* \mapsto \Sigma^*$ is a **polynomial time computable function** if some polynomial time Turing machine M exists that halts with just $f(x)$ on its tape, when it is given x as input

Polynomial Time Reducibility

Definition (Polynomial Time Mapping Reduction)

Language A is **polynomial time mapping reducible** to language B (denoted by $A \leq_P B$) if a polynomial time computable function $f : \Sigma^* \mapsto \Sigma^*$ exists, such that for every x:

$$x \in A \Leftrightarrow f(x) \in B$$

The function f is called the **polynomial time reduction** of A to B

Polynomial Time Reducibility

- Polynomial time reducibility is the efficient-analog to **mapping reducibility**

Polynomial Time Reducibility

- Polynomial time reducibility is the efficient-analog to **mapping reducibility**
- As with any other mapping reduction, a polynomial time reduction of A to B is a way to convert membership testing in A to that in B

Polynomial Time Reducibility

- Polynomial time reducibility is the efficient-analog to **mapping reducibility**
- As with any other mapping reduction, a polynomial time reduction of A to B is a way to convert membership testing in A to that in B
- Big plus: the conversion is now done "efficiently" (i.e., in polynomial time)

Polynomial Time Reducibility

- Polynomial time reducibility is the efficient-analog to **mapping reducibility**
- As with any other mapping reduction, a polynomial time reduction of A to B is a way to convert membership testing in A to that in B
- Big plus: the conversion is now done "efficiently" (i.e., in polynomial time)
- If one language is polynomial time reducible to another language B, which we already know a polynomial time solution for, then we obtain an overall polynomial time solution for A

Polynomial Time Reducibility

Theorem (Polynomial Time Reducibility)

If $A \leq_P B$ and $B \in P$ then $A \in P$

Polynomial Time Reducibility

Theorem (Polynomial Time Reducibility)

If $A \leq_P B$ and $B \in P$ then $A \in P$

Proof.

Let M_B be the polynomial time algorithm deciding B and f be the polynomial time reduction from A to B. We can describe a polynomial time algorithm M_A that decides A as follows:

Polynomial Time Reducibility

Theorem (Polynomial Time Reducibility)

If $A \leq_P B$ and $B \in P$ then $A \in P$

Proof.

Let M_B be the polynomial time algorithm deciding B and f be the polynomial time reduction from A to B. We can describe a polynomial time algorithm M_A that decides A as follows:

$M_A =$ "On input x:

1. Compute $f(x)$;
2. Run M_B on $f(x)$ and output whatever M_B does."

□

Polynomial Time Reducibility

- Before showing the power of polynomial time reductions, let's introduce a special case of the SAT problem before, called $3SAT$

Polynomial Time Reducibility

- Before showing the power of polynomial time reductions, let's introduce a special case of the SAT problem before, called $3SAT$
- $3SAT$ assumes the formula ϕ which we must test the satisfiability of has a particular form

Polynomial Time Reducibility

- Before showing the power of polynomial time reductions, let's introduce a special case of the SAT problem before, called $3SAT$
- $3SAT$ assumes the formula ϕ which we must test the satisfiability of has a particular form
- We call a **literal** any boolean variable (x) or its negated ($\overline{x}$)

Polynomial Time Reducibility

- Before showing the power of polynomial time reductions, let's introduce a special case of the *SAT* problem before, called 3*SAT*
- 3*SAT* assumes the formula ϕ which we must test the satisfiability of has a particular form
- We call a **literal** any boolean variable (x) or its negated $(\overline{x})$
- A **clause** is several literals connected with $\vee$s, e.g., $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$

Polynomial Time Reducibility

- Before showing the power of polynomial time reductions, let's introduce a special case of the *SAT* problem before, called 3*SAT*
- 3*SAT* assumes the formula ϕ which we must test the satisfiability of has a particular form
- We call a **literal** any boolean variable (x) or its negated ($\overline{x}$)
- A **clause** is several literals connected with $\vee$s, e.g., $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$
- A boolean formula is in **conjunctive normal form** (CNF) if it comprises several clauses connected with $\wedge$s:

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6})$$

3SAT

Definition (3SAT)

Let ϕ be a 3-CNF boolean formula, i.e., a CNF boolean formula where each clause has exactly 3 literals, e.g.:

$$\phi = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6)$$

3*SAT*

Definition (3*SAT*)

Let ϕ be a 3-CNF boolean formula, i.e., a CNF boolean formula where each clause has exactly 3 literals, e.g.:

$$\phi = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6)$$

We define 3*SAT* as the problem of testing whether a 3-CNF formula is satisfiable, i.e.:

$$3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3-CNF formula}\}$$

$3SAT$

Definition ($3SAT$)

Let ϕ be a 3-CNF boolean formula, i.e., a CNF boolean formula where each clause has exactly 3 literals, e.g.:

$$\phi = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6)$$

We define $3SAT$ as the problem of testing whether a 3-CNF formula is satisfiable, i.e.:

$$3SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable 3-CNF formula}\}$$

Note

In a satisfiable CNF formula, each clause must contain **at least one** literal whose assignment equals to 1 (`TRUE`)

Polynomial Time Reducibility: $3SAT \leq_P CLIQUE$

Theorem ($3SAT \leq_P CLIQUE$)

$3SAT$ is polynomial time reducible to $CLIQUE$

Polynomial Time Reducibility: $3SAT \leq_P CLIQUE$

Theorem ($3SAT \leq_P CLIQUE$)

$3SAT$ is polynomial time reducible to $CLIQUE$

Proof.

A sketch of the proof can be the following.
The polynomial time reduction f that we look for must convert 3-CNF boolean formulas to graphs. Graphs are constructed so as cliques of a specified size correspond to satisfying assignments of the formula. Structures within the graph ar designed to mimic the behavior of literals and clauses. □

$3SAT \leq_P CLIQUE$: Proof

- Let ϕ be a 3-CNF formula with k clauses as follows:

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$$

$3SAT \leq_P CLIQUE$: Proof

- Let ϕ be a 3-CNF formula with k clauses as follows:

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$$

- The reduction f must generate the string $\langle G, k \rangle$, i.e., the encoding of an undirected graph G

$3SAT \leq_P CLIQUE$: Proof

- Let ϕ be a 3-CNF formula with k clauses as follows:

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$$

- The reduction f must generate the string $\langle G, k \rangle$, i.e., the encoding of an undirected graph G
- The nodes of G are organized into k groups of three nodes each, called **triples**: $t_1, t_2, \ldots, t_k$

$3SAT \leq_P CLIQUE$: Proof

- Let ϕ be a 3-CNF formula with k clauses as follows:

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$$

- The reduction f must generate the string $\langle G, k \rangle$, i.e., the encoding of an undirected graph G
- The nodes of G are organized into k groups of three nodes each, called **triples**: $t_1, t_2, \ldots, t_k$
- Each triple t_i represents a clause of the original formula ϕ, and each node in a triple is a literal of the associated clause

$3SAT \leq_P CLIQUE$: Proof

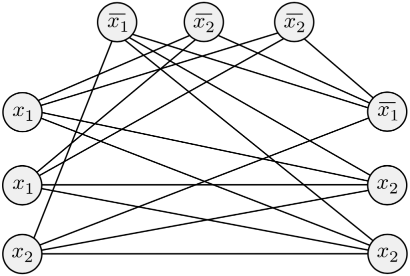
- The edges of G connect all but two types of pair of nodes

$3SAT \leq_P CLIQUE$: Proof

- The edges of G connect all but two types of pair of nodes
- In particular, no edge exists between nodes in the same triple and no edge is present between two nodes having contradictory labels

$3SAT \leq_P CLIQUE$: Proof

- The edges of G connect all but two types of pair of nodes
- In particular, no edge exists between nodes in the same triple and no edge is present between two nodes having contradictory labels



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

$3SAT \leq_P CLIQUE$: Proof

We will show that ϕ is satisfiable iff G has a k-clique

$3SAT \leq_P CLIQUE$: Proof

We will show that ϕ is satisfiable iff G has a k-clique

- $(\Rightarrow)$ Suppose that ϕ has a satisfying assignment

$3SAT \leq_P CLIQUE$: Proof

We will show that ϕ is satisfiable iff G has a k-clique

- ($\Rightarrow$) Suppose that ϕ has a satisfying assignment
- Thus, at least one literal is true in every clause

$3SAT \leq_P CLIQUE$: Proof

We will show that ϕ is satisfiable iff G has a k-clique

- ($\Rightarrow$) Suppose that ϕ has a satisfying assignment
- Thus, at least one literal is true in every clause
- In each triple of G we select one node corresponding to a `TRUE` literal in the existing satisfying assignment

$3SAT \leq_P CLIQUE$: Proof

We will show that ϕ is satisfiable iff G has a k-clique

- ($\Rightarrow$) Suppose that ϕ has a satisfying assignment
- Thus, at least one literal is true in every clause
- In each triple of G we select one node corresponding to a `TRUE` literal in the existing satisfying assignment
- If more than one literal is `TRUE` in a clause (triple), we choose one of them uniformly at random

$3SAT \leq_P CLIQUE$: Proof

We will show that ϕ is satisfiable iff G has a k-clique

- ($\Rightarrow$) Suppose that ϕ has a satisfying assignment
- Thus, at least one literal is true in every clause
- In each triple of G we select one node corresponding to a `TRUE` literal in the existing satisfying assignment
- If more than one literal is `TRUE` in a clause (triple), we choose one of them uniformly at random
- The nodes just selected form a k-clique!

$3SAT \leq_P CLIQUE$: Proof

Why do the selected nodes form a k-clique?

$3SAT \leq_P CLIQUE$: Proof

Why do the selected nodes form a k-clique?

- First of all, we select k nodes, i.e., one for each of the k triples

$3SAT \leq_P CLIQUE$: Proof

Why do the selected nodes form a k-clique?

- First of all, we select k nodes, i.e., one for each of the k triples
- Each pair of nodes selected are connected by an edge because no pair fits one of the exceptions described before

$3SAT \leq_P CLIQUE$: Proof

Why do the selected nodes form a k-clique?

- First of all, we select k nodes, i.e., one for each of the k triples
- Each pair of nodes selected are connected by an edge because no pair fits one of the exceptions described before
- *(i)* They cannot be part of the same triple because we select only one node per triple

$3SAT \leq_P CLIQUE$: Proof

Why do the selected nodes form a k-clique?

- First of all, we select k nodes, i.e., one for each of the k triples
- Each pair of nodes selected are connected by an edge because no pair fits one of the exceptions described before
- *(i)* They cannot be part of the same triple because we select only one node per triple
- *(ii)* They cannot have contradictory labels because the associated literals must be both `TRUE` in the satisfying assignment

$3SAT \leq_P CLIQUE$: Proof

Why do the selected nodes form a k-clique?

- First of all, we select k nodes, i.e., one for each of the k triples
- Each pair of nodes selected are connected by an edge because no pair fits one of the exceptions described before
- *(i)* They cannot be part of the same triple because we select only one node per triple
- *(ii)* They cannot have contradictory labels because the associated literals must be both `TRUE` in the satisfying assignment
- Therefore, G has a k-clique

$3SAT \leq_P CLIQUE$: Proof

We will show that ϕ is satisfiable iff G has a k-clique

$3SAT \leq_P CLIQUE$: Proof

We will show that ϕ is satisfiable iff G has a k-clique

- ($\Leftarrow$) Suppose that G has a k-clique

$3SAT \leq_P CLIQUE$: Proof

We will show that ϕ is satisfiable iff G has a k-clique

- ($\Leftarrow$) Suppose that G has a k-clique
- Thus, no two of the clique's nodes occur in the same triple because nodes in the same triples are not connected by construction

$3SAT \leq_P CLIQUE$: Proof

We will show that ϕ is satisfiable iff G has a k-clique

- ($\Leftarrow$) Suppose that G has a k-clique
- Thus, no two of the clique's nodes occur in the same triple because nodes in the same triples are not connected by construction
- Therefore, each of the k triples contains exactly one of the k nodes which the k-clique is made of

$3SAT \leq_P CLIQUE$: Proof

We will show that ϕ is satisfiable iff G has a k-clique

- ($\Leftarrow$) Suppose that G has a k-clique
- Thus, no two of the clique's nodes occur in the same triple because nodes in the same triples are not connected by construction
- Therefore, each of the k triples contains exactly one of the k nodes which the k-clique is made of
- We assign truth values to the variables of ϕ so that each literal labeling a clique node is set to `TRUE`

$3SAT \leq_P CLIQUE$: Proof

Why can we make such assignment?

$3SAT \leq_P CLIQUE$: Proof

Why can we make such assignment?

- Two nodes labeled in a contradictory way are not connected and therefore cannot be part of the k-clique!

$3SAT \leq_P CLIQUE$: Proof

Why can we make such assignment?

- Two nodes labeled in a contradictory way are not connected and therefore cannot be part of the k-clique!
- Such assignment trivially satisfies ϕ because each triple contains a clique node

$3SAT \leq_P CLIQUE$: Proof

Why can we make such assignment?

- Two nodes labeled in a contradictory way are not connected and therefore cannot be part of the k-clique!
- Such assignment trivially satisfies ϕ because each triple contains a clique node
- Hence, each clause contains at least a literal that is assigned `TRUE`

$3SAT \leq_P CLIQUE$: Proof

Why can we make such assignment?

- Two nodes labeled in a contradictory way are not connected and therefore cannot be part of the k-clique!
- Such assignment trivially satisfies ϕ because each triple contains a clique node
- Hence, each clause contains at least a literal that is assigned `TRUE`
- Therefore, ϕ is satisfiable!

$3SAT \leq_P CLIQUE$: Considerations

- The mapping $\phi \mapsto \langle G, k \rangle$ is indeed polynomial time computable

$3SAT \leq_P CLIQUE$: Considerations

- The mapping $\phi \mapsto \langle G, k \rangle$ is indeed polynomial time computable
- Suppose we are given with a 3-CNF formula ϕ having k clauses and m variables

$3SAT \leq_P CLIQUE$: Considerations

- The mapping $\phi \mapsto \langle G, k \rangle$ is indeed polynomial time computable
- Suppose we are given with a 3-CNF formula ϕ having k clauses and m variables
- The graph $G = (V, E)$ of $\langle G, k \rangle$ we build has:
  - $|V| = 3k$ nodes
  - $|E| < \binom{3k}{2} = \frac{3k(3k-1)}{2} = O(k^2)$

$3SAT \leq_P CLIQUE$: Considerations

- The mapping $\phi \mapsto \langle G, k \rangle$ is indeed polynomial time computable
- Suppose we are given with a 3-CNF formula ϕ having k clauses and m variables
- The graph $G = (V, E)$ of $\langle G, k \rangle$ we build has:
    - $|V| = 3k$ nodes
    - $|E| < \binom{3k}{2} = \frac{3k(3k-1)}{2} = O(k^2)$
- The size of the graph G is polynomial in the size of the 3-CNF formula ϕ

$3SAT \leq_P CLIQUE$: Consequences

- The last theorem we just proved tells us that, if $CLIQUE$ is solvable in polynomial time so is $3SAT$

$3SAT \leq_P CLIQUE$: Consequences

- The last theorem we just proved tells us that, if $CLIQUE$ is solvable in polynomial time so is $3SAT$
- At first glance, this may sound odd since the two problems are indeed quite different

$3SAT \leq_P CLIQUE$: Consequences

- The last theorem we just proved tells us that, if *CLIQUE* is solvable in polynomial time so is $3SAT$
- At first glance, this may sound odd since the two problems are indeed quite different
- Polynomial time reducibility allows us to link their complexities

$3SAT \leq_P CLIQUE$: Consequences

- The last theorem we just proved tells us that, if *CLIQUE* is solvable in polynomial time so is $3SAT$
- At first glance, this may sound odd since the two problems are indeed quite different
- Polynomial time reducibility allows us to link their complexities
- In the following, we show how to link the complexities of an **entire** class of problems

Table of Contents

NP-completeness: Definition

Definition (*NP*-completeness)

A language B is *NP*-**complete** if it satisfies the following two conditions:

1. $B \in NP$

NP-completeness: Definition

Definition (*NP*-completeness)

A language B is *NP*-**complete** if it satisfies the following two conditions:

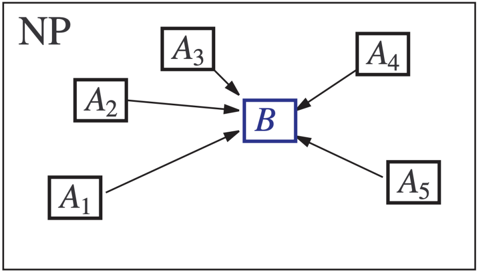
1. $B \in NP$
2. Every $A \in NP$ is polynomial time reducible to B

NP-completeness: Definition

Definition (*NP*-completeness)

A language B is *NP*-**complete** if it satisfies the following two conditions:

1. $B \in NP$
2. Every $A \in NP$ is polynomial time reducible to B



NP-completeness: Definition

Definition (*NP*-completeness)

A language B is *NP*-**complete** if it satisfies the following two conditions:

1. $B \in NP$
2. Every $A \in NP$ is polynomial time reducible to B

Note

- *NP*-complete problems are the "most difficult" problems in NP

NP-completeness: Definition

Definition (*NP*-completeness)

A language B is *NP*-**complete** if it satisfies the following two conditions:

1. $B \in NP$
2. Every $A \in NP$ is polynomial time reducible to B

Note

- *NP*-complete problems are the "most difficult" problems in NP
- If we omit first requirement (i.e., $B \in NP$), then we say that B is *NP*-**hard**

NP-completeness: Consequences

Theorem

If B is NP-complete and $B \in P$ then $P = NP$

NP-completeness: Consequences

Theorem

If B is NP-complete and $B \in P$ then $P = NP$

Proof.

From the definition above, if B is NP-complete it means that $B \in NP$ and **every** other language/problem $A \in NP$ is polynomially reducible to B.

NP-completeness: Consequences

Theorem

If B is NP-complete and $B \in P$ then $P = NP$

Proof.

From the definition above, if B is *NP*-complete it means that $B \in NP$ and **every** other language/problem $A \in NP$ is polynomially reducible to B. Now, if we know that a solver for B exists and it runs in polynomial time (i.e., $B \in P$) then we can solve **every** other problem $A \in NP$ by:

1. applying the polynomial time reduction from A to B

NP-completeness: Consequences

Theorem

If B is NP-complete and $B \in P$ then $P = NP$

Proof.

From the definition above, if B is *NP*-complete it means that $B \in NP$ and **every** other language/problem $A \in NP$ is polynomially reducible to B.
Now, if we know that a solver for B exists and it runs in polynomial time (i.e., $B \in P$) then we can solve **every** other problem $A \in NP$ by:

1. applying the polynomial time reduction from A to B
2. running the polynomial time solver for B

NP-completeness: Consequences

Theorem

If B is NP-complete and $B \in P$ then $P = NP$

Proof.

From the definition above, if B is NP-complete it means that $B \in NP$ and **every** other language/problem $A \in NP$ is polynomially reducible to B. Now, if we know that a solver for B exists and it runs in polynomial time (i.e., $B \in P$) then we can solve **every** other problem $A \in NP$ by:

1. applying the polynomial time reduction from A to B
2. running the polynomial time solver for B

Since the process above is a composition of polynomial time algorithms and it holds for all $A \in NP$, we can state that
$\forall A \in NP, A \in P \Leftrightarrow P = NP$. □

NP-completeness: Consequences

Theorem

If B is NP-complete and $B \leq_P C$ for $C \in NP$, then C is NP-complete

NP-completeness: Consequences

Theorem

If B is NP-complete and $B \leq_P C$ for $C \in NP$, then C is NP-complete

Proof.

We know that $C \in NP$, so we must show that every other problem in NP is polynomial time reducible to C.

NP-completeness: Consequences

Theorem

If B is NP-complete and $B \leq_P C$ for $C \in NP$, then C is NP-complete

Proof.

We know that $C \in NP$, so we must show that every other problem in NP is polynomial time reducible to C.

Instead of proving this for every possible $A \in NP$, we can take advantage of knowing that B is NP-complete itself!

NP-completeness: Consequences

Theorem

If B is NP-complete and $B \leq_P C$ for $C \in NP$, then C is NP-complete

Proof.

We know that $C \in NP$, so we must show that every other problem in NP is polynomial time reducible to C.
Instead of proving this for every possible $A \in NP$, we can take advantage of knowing that B is NP-complete itself!
Because B is NP-complete, every other $A \in NP$ is polynomial time reducible to it, i.e., $A \leq_P B$, $\forall A \in NP$.

NP-completeness: Consequences

Theorem

If B is NP-complete and $B \leq_P C$ for $C \in NP$, then C is NP-complete

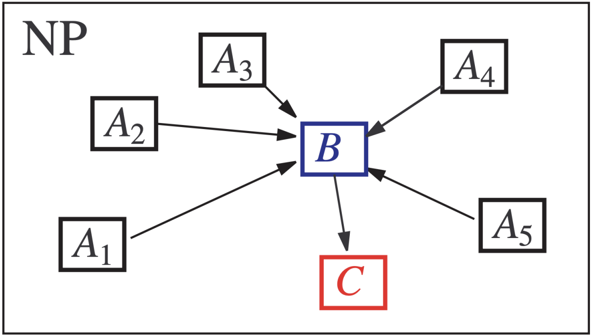
Proof.

We know that $C \in NP$, so we must show that every other problem in NP is polynomial time reducible to C.
Instead of proving this for every possible $A \in NP$, we can take advantage of knowing that B is NP-complete itself!
Because B is NP-complete, every other $A \in NP$ is polynomial time reducible to it, i.e., $A \leq_P B,\ \forall A \in NP$.
Plus, we know from the hypothesis that $B \leq_P C$, and due to composition of polynomial time reductions $A \leq_P B\ \forall A \in NP$ and $B \leq_P C$ then $A \leq_P C\ \forall A \in NP$. Therefore C is NP-complete! □

NP-completeness: Consequences

Theorem

If B is NP-complete and $B \leq_P C$ for $C \in NP$, then C is NP-complete



NP-completeness: The Cook-Levin Theorem

- Once we have **one** *NP*-complete problem, we may obtain others by polynomial time reduction from it

NP-completeness: The Cook-Levin Theorem

- Once we have **one** *NP*-complete problem, we may obtain others by polynomial time reduction from it
- However, finding such first *NP*-complete problem is really hard!

NP-completeness: The Cook-Levin Theorem

- Once we have **one** *NP*-complete problem, we may obtain others by polynomial time reduction from it
- However, finding such first *NP*-complete problem is really hard!
- Historically, we do so by showing that our original problem of boolean satisfiability (SAT) is *NP*-complete

NP-completeness: The Cook-Levin Theorem

Theorem

SAT is NP-complete

NP-completeness: The Cook-Levin Theorem

Theorem

SAT is NP-complete

Proof Sketch.

In order to show that *SAT* is *NP*-complete we must prove that:

1. $SAT \in NP$

NP-completeness: The Cook-Levin Theorem

Theorem

SAT is NP-complete

Proof Sketch.

In order to show that SAT is NP-complete we must prove that:

1. $SAT \in NP$
2. $\forall A \in NP, \; A \leq_P SAT$

□

NP-completeness: The Cook-Levin Theorem

Proof Sketch.

In order to show that *SAT* is *NP*-complete we must prove that:

1. $SAT \in NP$
2. $\forall A \in NP,\ A \leq_P SAT$

Proving 1 is straightforward:
A polynomial time NTM can guess assignment to a boolean formula ϕ and accept if that assignment satisfies ϕ

NP-completeness: The Cook-Levin Theorem

Proof Sketch.

In order to show that *SAT* is *NP*-complete we must prove that:

1. $SAT \in NP$
2. $\forall A \in NP,\ A \leq_P SAT$

Proving 1 is straightforward:
A polynomial time NTM can guess assignment to a boolean formula ϕ and accept if that assignment satisfies ϕ
In alternative, given ϕ along with a certificate (i.e., an assignment) we can design a polynomial time verifier that checks if the assignment satisfies ϕ □

NP-completeness: The Cook-Levin Theorem

Proof Sketch.

In order to show that SAT is NP-complete we must prove that:

1. $SAT \in NP$
2. $\forall A \in NP,\ A \leq_P SAT$

Proving 2 is harder! □

NP-completeness: The Cook-Levin Theorem

- Let $A \subseteq \Sigma^*$ be a language in *NP*

NP-completeness: The Cook-Levin Theorem

- Let $A \subseteq \Sigma^*$ be a language in NP
- We need to show that $A \leq_P SAT$

NP-completeness: The Cook-Levin Theorem

- Let $A \subseteq \Sigma^*$ be a language in NP
- We need to show that $A \leq_P SAT$
- For every $w \in \Sigma^*$, we want a boolean formula ϕ such that:
  1. $w \in A \Leftrightarrow f(w) = \langle \phi \rangle \in SAT$
  2. f is a polynomial time reduction

NP-completeness: The Cook-Levin Theorem

- Let $A \subseteq \Sigma^*$ be a language in NP
- We need to show that $A \leq_P SAT$
- For every $w \in \Sigma^*$, we want a boolean formula ϕ such that:
  1. $w \in A \Leftrightarrow f(w) = \langle \phi \rangle \in SAT$
  2. f is a polynomial time reduction
- Let N be polynomial time NTM that decides A in time at most n^k, where $n = |w|$

NP-completeness: The Cook-Levin Theorem

Outline of the basic approach:

$w \in A \Leftrightarrow$ NTM N accepts input w

NP-completeness: The Cook-Levin Theorem

Outline of the basic approach:

$w \in A \Leftrightarrow$ NTM N accepts input w

$\Leftrightarrow \exists$ an accepting computation history of N on w

NP-completeness: The Cook-Levin Theorem

Outline of the basic approach:

$w \in A \Leftrightarrow$ NTM N accepts input w

$\Leftrightarrow \exists$ an accepting computation history of N on w

$\Leftrightarrow \exists$ a boolean function ϕ and variables $x_1, \ldots, x_m$ such that $\phi(x_1, \ldots, x_m) = \texttt{TRUE}$

NP-completeness: The Cook-Levin Theorem

Outline of the basic approach:

$w \in A \Leftrightarrow$ NTM N accepts input w

$\Leftrightarrow \exists$ an accepting computation history of N on w

$\Leftrightarrow \exists$ a boolean function ϕ and variables $x_1, \ldots, x_m$ such that $\phi(x_1, \ldots, x_m) = \texttt{TRUE}$

Note

The basic intuition is to be able to show that **any** algorithm can be encoded as a boolean formula!

NP-completeness: The Cook-Levin Theorem

Key Idea:

"Satisfying assignment of ϕ" $\Leftrightarrow$ "Accepting computation history of N on input w"

NP-completeness: The Cook-Levin Theorem

Key Idea:
"Satisfying assignment of ϕ" $\Leftrightarrow$ "Accepting computation history of N on input w"

- A computation of N (i.e., a list of configurations) on **some** branch of its computation tree is described by a $n^k \times n^k$ **tableau**

NP-completeness: The Cook-Levin Theorem

Key Idea:
"Satisfying assignment of ϕ" $\Leftrightarrow$ "Accepting computation history of N on input w"

- A computation of N (i.e., a list of configurations) on **some** branch of its computation tree is described by a $n^k \times n^k$ **tableau**
- The first row of the tableau is the starting configuration of N on w

NP-completeness: The Cook-Levin Theorem

Key Idea:

"Satisfying assignment of ϕ" $\Leftrightarrow$ "Accepting computation history of N on input w"

- A computation of N (i.e., a list of configurations) on **some** branch of its computation tree is described by a $n^k \times n^k$ **tableau**
- The first row of the tableau is the starting configuration of N on w
- Each row follows the previous one according to N's transition function

NP-completeness: The Cook-Levin Theorem

Key Idea:

"Satisfying assignment of ϕ" $\Leftrightarrow$ "Accepting computation history of N on input w"

- A computation of N (i.e., a list of configurations) on **some** branch of its computation tree is described by a $n^k \times n^k$ **tableau**
- The first row of the tableau is the starting configuration of N on w
- Each row follows the previous one according to N's transition function
- A tableau is **accepting** if any row of the tableau encodes an accepting configuration

NP-completeness: The Cook-Levin Theorem

Key Idea:
"Satisfying assignment of ϕ" $\Leftrightarrow$ "Accepting computation history of N on input w"

- A computation of N (i.e., a list of configurations) on **some** branch of its computation tree is described by a $n^k \times n^k$ **tableau**
- The first row of the tableau is the starting configuration of N on w
- Each row follows the previous one according to N's transition function
- A tableau is **accepting** if any row of the tableau encodes an accepting configuration
- Every accepting tableau for N on w corresponds to an accepting computation branch of N on w

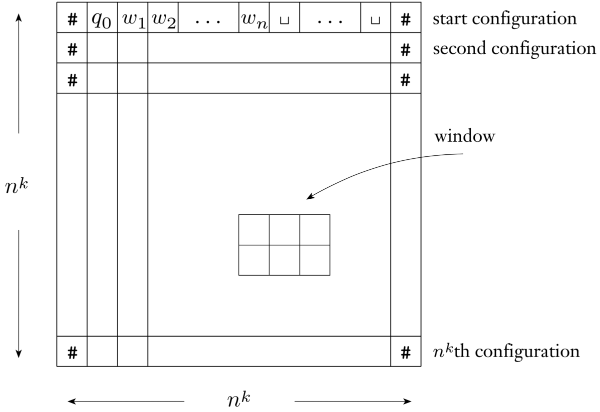
NP-completeness: The Cook-Levin Theorem

Key Idea:

"Satisfying assignment of ϕ" $\Leftrightarrow$ "Accepting computation history of N on input w"

- A computation of N (i.e., a list of configurations) on **some** branch of its computation tree is described by a $n^k \times n^k$ **tableau**
- The first row of the tableau is the starting configuration of N on w
- Each row follows the previous one according to N's transition function
- A tableau is **accepting** if any row of the tableau encodes an accepting configuration
- Every accepting tableau for N on w corresponds to an accepting computation branch of N on w
- The problem of determining whether N accepts w is **equivalent** to finding if an accepting tableau for N on w exists

NP-completeness: The Cook-Levin Theorem



NP-completeness: The Cook-Levin Theorem

Step 1: Describe computations of NTM N on w by boolean variables using the tableau

- If $|w| = n$, then any computation history of N on w has at most n^k configurations because we assumed N runs in time n^k

NP-completeness: The Cook-Levin Theorem

Step 1: Describe computations of NTM N on w by boolean variables using the tableau

- If $|w| = n$, then any computation history of N on w has at most n^k configurations because we assumed N runs in time n^k
- Let $C = Q \cup \Sigma \cup \{\#\}$, where Q is the set of states and Σ the alphabet of N

NP-completeness: The Cook-Levin Theorem

Step 1: Describe computations of NTM N on w by boolean variables using the tableau

- If $|w| = n$, then any computation history of N on w has at most n^k configurations because we assumed N runs in time n^k
- Let $C = Q \cup \Sigma \cup \{\#\}$, where Q is the set of states and Σ the alphabet of N
- Each cell (i, j) in the tableau contains one element of C

NP-completeness: The Cook-Levin Theorem

Step 1: Describe computations of NTM N on w by boolean variables using the tableau

- If $|w| = n$, then any computation history of N on w has at most n^k configurations because we assumed N runs in time n^k
- Let $C = Q \cup \Sigma \cup \{\#\}$, where Q is the set of states and Σ the alphabet of N
- Each cell (i, j) in the tableau contains one element of C
- For each $i, j \in \{1, \dots, n^k\}$ and for each $s \in C$ we associate a boolean variable $x_{i,j,s}$ of ϕ

NP-completeness: The Cook-Levin Theorem

Step 1: Describe computations of NTM N on w by boolean variables using the tableau

- If $|w| = n$, then any computation history of N on w has at most n^k configurations because we assumed N runs in time n^k
- Let $C = Q \cup \Sigma \cup \{\#\}$, where Q is the set of states and Σ the alphabet of N
- Each cell (i, j) in the tableau contains one element of C
- For each $i, j \in \{1, \ldots, n^k\}$ and for each $s \in C$ we associate a boolean variable $x_{i,j,s}$ of ϕ
- $x_{i,j,s} = 1$ means "cell (i, j) contains s"

NP-completeness: The Cook-Levin Theorem

N accepts w iff:

1. Each cell in the tableau is well-defined

NP-completeness: The Cook-Levin Theorem

N accepts w iff:

1. Each cell in the tableau is well-defined
2. The first row of the tableau is the initial configuration with w as the input

NP-completeness: The Cook-Levin Theorem

N accepts w iff:

1. Each cell in the tableau is well-defined
2. The first row of the tableau is the initial configuration with w as the input
3. Each row follows from the previous rows using the transition function given by N

NP-completeness: The Cook-Levin Theorem

N accepts w iff:

1. Each cell in the tableau is well-defined
2. The first row of the tableau is the initial configuration with w as the input
3. Each row follows from the previous rows using the transition function given by N
4. Some row has a cell that includes an accepting state q_{accept}

NP-completeness: The Cook-Levin Theorem

N accepts w iff:

1. Each cell in the tableau is well-defined
2. The first row of the tableau is the initial configuration with w as the input
3. Each row follows from the previous rows using the transition function given by N
4. Some row has a cell that includes an accepting state q_{accept}

Note

We design ϕ so that a satisfying assignment to its variables $x_{i,j,s}$ corresponds to an accepting tableau for N on w

NP-completeness: The Cook-Levin Theorem

Step 2: Express conditions for an accepting sequence of configurations of NTM N on w by a boolean formula ϕ as the AND of four parts:

$$\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

1. ϕ_{cell} = "for each cell (i, j), there is **exactly one** $s \in C$ with $x_{i,j,s} = 1$" (cell is well defined)

NP-completeness: The Cook-Levin Theorem

Step 2: Express conditions for an accepting sequence of configurations of NTM N on w by a boolean formula ϕ as the AND of four parts:

$$\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

1. $\phi_{\text{cell}} =$ "for each cell (i, j), there is **exactly one** $s \in C$ with $x_{i,j,s} = 1$" (cell is well defined)
2. $\phi_{\text{start}} =$ "first row of tableau is the starting configuration of N on w"

NP-completeness: The Cook-Levin Theorem

Step 2: Express conditions for an accepting sequence of configurations of NTM N on w by a boolean formula ϕ as the AND of four parts:

$$\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

1. ϕ_{cell} = "for each cell (i, j), there is **exactly one** $s \in C$ with $x_{i,j,s} = 1$" (cell is well defined)
2. ϕ_{start} = "first row of tableau is the starting configuration of N on w"
3. ϕ_{move} = "every 2×3 window is consistent with N's transition function"

NP-completeness: The Cook-Levin Theorem

Step 2: Express conditions for an accepting sequence of configurations of NTM N on w by a boolean formula ϕ as the AND of four parts:

$$\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

1. ϕ_{cell} = "for each cell (i, j), there is **exactly one** $s \in C$ with $x_{i,j,s} = 1$" (cell is well defined)
2. ϕ_{start} = "first row of tableau is the starting configuration of N on w"
3. ϕ_{move} = "every 2×3 window is consistent with N's transition function"
4. ϕ_{accept} = "at least one row of tableau is an accepting configuration of N on w"

NP-completeness: The Cook-Levin Theorem

$$\phi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right].$$

NP-completeness: The Cook-Levin Theorem

$$\phi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right].$$

- The outer $\wedge$ ensures the formula applies to every cell (i, j)

NP-completeness: The Cook-Levin Theorem

$$\phi_{\text{cell}} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right].$$

- The outer $\wedge$ ensures the formula applies to every cell (i, j)
- The first inner component guarantees that at least one cell is "on", i.e., at least one valid symbol is on a cell

NP-completeness: The Cook-Levin Theorem

$$\phi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right].$$

- The outer $\wedge$ ensures the formula applies to every cell (i, j)
- The first inner component guarantees that at least one cell is "on", i.e., at least one valid symbol is on a cell
- The second inner component says that in each pair of variables, at least one is turned off

NP-completeness: The Cook-Levin Theorem

$$\phi_{\text{cell}} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right].$$

- The outer $\wedge$ ensures the formula applies to every cell (i, j)
- The first inner component guarantees that at least one cell is "on", i.e., at least one valid symbol is on a cell
- The second inner component says that in each pair of variables, at least one is turned off
- The accepting tableau must meet condition 1

NP-completeness: The Cook-Levin Theorem

$$\begin{aligned}\phi_{\text{start}} = \; & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}\,.\end{aligned}$$

NP-completeness: The Cook-Levin Theorem

$$\begin{aligned}\phi_{\text{start}} = \; & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} \,.\end{aligned}$$

- This formula ensures that the first row of the tableau is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on

NP-completeness: The Cook-Levin Theorem

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} \,.\end{aligned}$$

- This formula ensures that the first row of the tableau is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on
- The accepting tableau must meet condition 2

NP-completeness: The Cook-Levin Theorem

$$\phi_{\text{move}} = \bigwedge_{1 \leq i < n^k,\ 1 < j < n^k} \big(\text{the } (i, j)\text{-window is legal}\big).$$

$$\bigvee_{\substack{a_1, \ldots, a_6 \\ \text{is a legal window}}} \big(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}\big)$$

NP-completeness: The Cook-Levin Theorem

$$\phi_{\text{move}} = \bigwedge_{1 \leq i < n^k,\ 1 < j < n^k} \big(\text{the } (i,j)\text{-window is legal}\big).$$

$$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is a legal window}}} \left(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}\right)$$

- This formula ensures that each 2x3 window is legal according to *N*'s transition function (proof omitted here)

NP-completeness: The Cook-Levin Theorem

$$\phi_{\text{move}} = \bigwedge_{1 \le i < n^k,\ 1 < j < n^k} \big(\text{the } (i,j)\text{-window is legal}\big).$$

$$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is a legal window}}} \left(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}\right)$$

- This formula ensures that each 2x3 window is legal according to *N*'s transition function (proof omitted here)
- Intuitively, this is a disjunction over $|C|^6$ possible legal sequences

NP-completeness: The Cook-Levin Theorem

$$\phi_{\text{move}} = \bigwedge_{1 \leq i < n^k,\ 1 < j < n^k} \big(\text{the } (i,j)\text{-window is legal}\big).$$

$$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is a legal window}}} \left(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}\right)$$

- This formula ensures that each 2x3 window is legal according to *N*'s transition function (proof omitted here)
- Intuitively, this is a disjunction over $|C|^6$ possible legal sequences
- The accepting tableau must meet condition 3

NP-completeness: The Cook-Levin Theorem

$$\phi_{\text{accept}} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{\text{accept}}}.$$

NP-completeness: The Cook-Levin Theorem

$$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{\text{accept}}}.$$

- This formula ensures that an accepting configuration occurs in the tableau

NP-completeness: The Cook-Levin Theorem

$$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{\text{accept}}}.$$

- This formula ensures that an accepting configuration occurs in the tableau
- It guarantees that q_{accept}, the symbol for the accept state, appears in one of the cells of the tableau

NP-completeness: The Cook-Levin Theorem

$$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{\text{accept}}}.$$

- This formula ensures that an accepting configuration occurs in the tableau
- It guarantees that q_{accept}, the symbol for the accept state, appears in one of the cells of the tableau
- The accepting tableau must meet condition 4

NP-completeness: The Cook-Levin Theorem

Putting all together:

NP-completeness: The Cook-Levin Theorem

Putting all together:

- Given a non-deterministic Turing machine N and some input w we have shown that we can build a propositional formula ϕ:

$$\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

  ϕ is satisfiable if and only if N accepts w

NP-completeness: The Cook-Levin Theorem

Putting all together:

- Given a non-deterministic Turing machine N and some input w we have shown that we can build a propositional formula ϕ:

$$\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

  ϕ is satisfiable if and only if N accepts w
- The subformulas encode the 4 conditions needed there be an accepting tableau for the computation of N on input w

NP-completeness: The Cook-Levin Theorem

Putting all together:

- Given a non-deterministic Turing machine N and some input w we have shown that we can build a propositional formula ϕ:

$$\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

  ϕ is satisfiable if and only if N accepts w
- The subformulas encode the 4 conditions needed there be an accepting tableau for the computation of N on input w
- It only remains to show that the reduction from w to ϕ is computable in polynomial time

NP-completeness: The Cook-Levin Theorem

- We assumed that N runs in n^k time on inputs of length n

NP-completeness: The Cook-Levin Theorem

- We assumed that N runs in n^k time on inputs of length n
- Therefore, the tableau has n^k rows and n^k columns

NP-completeness: The Cook-Levin Theorem

- We assumed that N runs in n^k time on inputs of length n
- Therefore, the tableau has n^k rows and n^k columns
- Each $n^k * n^k = n^{2k}$ cell has $|C|$ variables associated with it, therefore the total number of literals is $|C| * n^{2k} = O(n^{2k})$

NP-completeness: The Cook-Levin Theorem

- We assumed that N runs in n^k time on inputs of length n
- Therefore, the tableau has n^k rows and n^k columns
- Each $n^k * n^k = n^{2k}$ cell has $|C|$ variables associated with it, therefore the total number of literals is $|C| * n^{2k} = O(n^{2k})$
- Globally, the total size of the formula ϕ is $O(n^{2k})$, which is clearly polynomial in the original input size $|w| = n$

NP-completeness: The Cook-Levin Theorem

- We assumed that N runs in n^k time on inputs of length n
- Therefore, the tableau has n^k rows and n^k columns
- Each $n^k * n^k = n^{2k}$ cell has $|C|$ variables associated with it, therefore the total number of literals is $|C| * n^{2k} = O(n^{2k})$
- Globally, the total size of the formula ϕ is $O(n^{2k})$, which is clearly polynomial in the original input size $|w| = n$
- For every $A \in NP$, this gives a polynomial time reduction from $w \in A$ to a boolean formula ϕ which is satisfiable iff from to $\phi \in SAT$

NP-completeness: The Cook-Levin Theorem

- We assumed that N runs in n^k time on inputs of length n
- Therefore, the tableau has n^k rows and n^k columns
- Each $n^k * n^k = n^{2k}$ cell has $|C|$ variables associated with it, therefore the total number of literals is $|C| * n^{2k} = O(n^{2k})$
- Globally, the total size of the formula ϕ is $O(n^{2k})$, which is clearly polynomial in the original input size $|w| = n$
- For every $A \in NP$, this gives a polynomial time reduction from $w \in A$ to a boolean formula ϕ which is satisfiable iff from to $\phi \in SAT$
- SAT in NP-complete!

NP-completeness: The Cook-Levin Theorem

- Showing the *NP*-completeness of other languages generally doesn't require such a hard proof

NP-completeness: The Cook-Levin Theorem

- Showing the *NP*-completeness of other languages generally doesn't require such a hard proof
- In fact, *NP*-completeness can be demonstrated by showing that a polynomial time reduction exists from a problem that is already known to be *NP*-complete (like *SAT*!)

NP-completeness: The Cook-Levin Theorem

- Showing the *NP*-completeness of other languages generally doesn't require such a hard proof
- In fact, *NP*-completeness can be demonstrated by showing that a polynomial time reduction exists from a problem that is already known to be *NP*-complete (like *SAT*!)
- Rather than using *SAT*, typically a reduction is shown from one of its variant, i.e., 3*SAT* or 3-CNF formulas

NP-completeness: The Cook-Levin Theorem

- Showing the *NP*-completeness of other languages generally doesn't require such a hard proof
- In fact, *NP*-completeness can be demonstrated by showing that a polynomial time reduction exists from a problem that is already known to be *NP*-complete (like *SAT*!)
- Rather than using *SAT*, typically a reduction is shown from one of its variant, i.e., 3*SAT* or 3-CNF formulas
- Before being able to do that, we need to show that 3*SAT* is also *NP*-complete

$3SAT$ is NP-complete

- To prove that $3SAT$ is NP-complete we must show that:
  1. $3SAT \in NP$
  2. $\forall A \in NP, A \leq_P 3SAT$

$3SAT$ is NP-complete

- To prove that $3SAT$ is NP-complete we must show that:
  1. $3SAT \in NP$
  2. $\forall A \in NP, A \leq_P 3SAT$
- Obviously, $3SAT \in NP$!

$3SAT$ is NP-complete

- To prove that $3SAT$ is NP-complete we must show that:
  1. $3SAT \in NP$
  2. $\forall A \in NP, A \leq_P 3SAT$
- Obviously, $3SAT \in NP$!
- One way to show 2 is to prove that SAT polynomial time reduces to $3SAT$

$3SAT$ is NP-complete

- To prove that $3SAT$ is NP-complete we must show that:
  1. $3SAT \in NP$
  2. $\forall A \in NP, A \leq_P 3SAT$
- Obviously, $3SAT \in NP$!
- One way to show 2 is to prove that SAT polynomial time reduces to $3SAT$
- Instead, we slightly adapt the proof we used to show that SAT is NP-complete to achieve this

3*SAT* is *NP*-complete

- The boolean formula ϕ used to show that *SAT* is *NP*-complete is almost in CNF

3*SAT* is *NP*-complete

- The boolean formula ϕ used to show that *SAT* is *NP*-complete is almost in CNF
- Let's see how each of ϕ's subformulas are organized

3*SAT* is *NP*-complete

- The boolean formula ϕ used to show that *SAT* is *NP*-complete is almost in CNF
- Let's see how each of ϕ's subformulas are organized
- ϕ_{cell} is made of a big AND of subformulas, each one containing a big OR and a big AND of ORs

$$\phi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right].$$

3*SAT* is *NP*-complete

- The boolean formula ϕ used to show that *SAT* is *NP*-complete is almost in CNF
- Let's see how each of ϕ's subformulas are organized
- ϕ_{cell} is made of a big AND of subformulas, each one containing a big OR and a big AND of ORs

$$\phi_{\text{cell}} = \bigwedge_{1\leq i,j\leq n^k} \left[\left(\bigvee_{s\in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t\in C \\ s\neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right].$$

- Thus, ϕ_{cell} is an AND of **clauses**, therefore already in CNF

3*SAT* is *NP*-complete

- The boolean formula ϕ used to show that *SAT* is *NP*-complete is almost in CNF
- Let's see how each of ϕ's subformulas are organized
- ϕ_{start} is just a big AND of variables

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} \,.\end{aligned}$$

3*SAT* is *NP*-complete

- The boolean formula ϕ used to show that *SAT* is *NP*-complete is almost in CNF
- Let's see how each of ϕ's subformulas are organized
- ϕ_{start} is just a big AND of variables

$$
\begin{aligned}
\phi_{\text{start}} = {} & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\
& x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\
& x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} \, .
\end{aligned}
$$

- Each variable can be considered as a "degenerate" clause of size 1 with a single literal

3*SAT* is *NP*-complete

- The boolean formula ϕ used to show that *SAT* is *NP*-complete is almost in CNF
- Let's see how each of ϕ's subformulas are organized
- ϕ_{start} is just a big AND of variables

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} \,.\end{aligned}$$

- Each variable can be considered as a "degenerate" clause of size 1 with a single literal
- Therefore, ϕ_{start} is also in CNF

3*SAT* is *NP*-complete

- The boolean formula ϕ used to show that *SAT* is *NP*-complete is almost in CNF
- Let's see how each of ϕ's subformulas are organized
- ϕ_{accept} is just a big OR of variables

$$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{\text{accept}}}.$$

3*SAT* is *NP*-complete

- The boolean formula ϕ used to show that *SAT* is *NP*-complete is almost in CNF
- Let's see how each of ϕ's subformulas are organized
- ϕ_{accept} is just a big OR of variables

$$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{\text{accept}}}.$$

- All those variables can be seen as the literals of a single, big clause

3*SAT* is *NP*-complete

- The boolean formula ϕ used to show that *SAT* is *NP*-complete is almost in CNF
- Let's see how each of ϕ's subformulas are organized
- ϕ_{accept} is just a big OR of variables

$$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{\text{accept}}}.$$

- All those variables can be seen as the literals of a single, big clause
- Therefore, ϕ_{accept} is also in CNF

3*SAT* is *NP*-complete

- The boolean formula ϕ used to show that *SAT* is *NP*-complete is almost in CNF
- Let's see how each of ϕ's subformulas are organized
- ϕ_{move} is the only one slightly problematic!

$$\phi_{\text{move}} = \bigwedge_{1 \leq i < n^k,\ 1 < j < n^k} \big(\text{the } (i,j)\text{-window is legal}\big).$$

3*SAT* is *NP*-complete

- The boolean formula ϕ used to show that *SAT* is *NP*-complete is almost in CNF
- Let's see how each of ϕ's subformulas are organized
- ϕ_{move} is the only one slightly problematic!

$$\phi_{\text{move}} = \bigwedge_{1 \leq i < n^k,\ 1 < j < n^k} \big(\text{the } (i,j)\text{-window is legal}\big).$$

- It is a big AND of subformulas, each containing an OR of ANDs describing all the possible windows

3*SAT* is *NP*-complete

- The boolean formula ϕ used to show that *SAT* is *NP*-complete is almost in CNF
- Let's see how each of ϕ's subformulas are organized
- ϕ_{move} is the only one slightly problematic!

$$\phi_{\text{move}} = \bigwedge_{1 \leq i < n^k,\ 1 < j < n^k} \big(\text{the } (i,j)\text{-window is legal}\big).$$

- It is a big AND of subformulas, each containing an OR of ANDs describing all the possible windows
- Using the **distributive law**, however, we can transform any OR of ANDs into an equivalent AND of ORs (i.e., CNF)

Converting to CNF

- Every propositional formula can be converted into an equivalent formula that is in CNF

Converting to CNF

- Every propositional formula can be converted into an equivalent formula that is in CNF
- This transformation is based on rules about logical equivalences:
  1. double negation elimination: $P \Leftrightarrow \neg(\neg P)$

Converting to CNF

- Every propositional formula can be converted into an equivalent formula that is in CNF
- This transformation is based on rules about logical equivalences:
  1. double negation elimination: $P \Leftrightarrow \neg(\neg P)$
  2. De Morgan's laws:
  $\neg(P \vee Q) \Leftrightarrow (\neg P) \wedge (\neg Q)$; $\neg(P \wedge Q) \Leftrightarrow (\neg P) \vee (\neg Q)$

Converting to CNF

- Every propositional formula can be converted into an equivalent formula that is in CNF
- This transformation is based on rules about logical equivalences:
  1. double negation elimination: $P \Leftrightarrow \neg(\neg P)$
  2. De Morgan's laws:
     $\neg(P \vee Q) \Leftrightarrow (\neg P) \wedge (\neg Q)$; $\neg(P \wedge Q) \Leftrightarrow (\neg P) \vee (\neg Q)$
  3. distributive law:
     $P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$; $P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$

3*SAT* is *NP*-complete

- Now, we have written ϕ in CNF but we still need to convert it to 3-CNF

3*SAT* is *NP*-complete

- Now, we have written ϕ in CNF but we still need to convert it to 3-CNF
- In each clause containing less than 3 literals, we just replicate one of the literals until getting a 3-literal clause

3*SAT* is *NP*-complete

- Now, we have written ϕ in CNF but we still need to convert it to 3-CNF
- In each clause containing less than 3 literals, we just replicate one of the literals until getting a 3-literal clause
- In each clause that has more than 3 literals, we need to split them into multiple 3-literal clauses preserving the satisfiability

3*SAT* is *NP*-complete

Example

Suppose our clause is made of 4 literals:

$$c = (a_1 \vee a_2 \vee a_3 \vee a_4)$$

$3SAT$ is NP-complete

Example

Suppose our clause is made of 4 literals:

$$c = (a_1 \vee a_2 \vee a_3 \vee a_4)$$

Then we can transform c into c' as follows:

$$c' = (a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$$

3*SAT* is *NP*-complete

Example

Suppose our clause is made of 4 literals:

$$c = (a_1 \vee a_2 \vee a_3 \vee a_4)$$

Then we can transform c into c' as follows:

$$c' = (a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$$

Here, z is a new variable (literal) and if some assignment of the a_i's satisifies c we can also find a setting of z that satisfies c'.

3*SAT* is *NP*-complete

Example

More generally, if the clause contains ℓ literals:

$$c = (a_1 \vee a_2 \vee \ldots \vee a_\ell)$$

We can replace it with $\ell - 2$ clauses as follows:

$$c' = (a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \ldots \wedge (\overline{z_{\ell-3}} \vee a_{\ell-1} \vee a_\ell))$$

3*SAT* is *NP*-complete

Example

More generally, if the clause contains ℓ literals:

$$c = (a_1 \vee a_2 \vee \ldots \vee a_\ell)$$

We can replace it with $\ell - 2$ clauses as follows:

$$c' = (a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \ldots \wedge (\overline{z_{\ell-3}} \vee a_{\ell-1} \vee a_\ell))$$

3*SAT* is *NP*-complete!

Proving *NP*-completeness: Summary

How to prove that a problem C is NP-complete?

Proving *NP*-completeness: Summary

How to prove that a problem C is NP-complete?

- Following the definition may be tedious as we need to show that:
  1. $C \in NP$ ("easy")
  2. C is NP-hard, i.e., $\forall A \in NP,\ A \leq_P C$

Proving *NP*-completeness: Summary

How to prove that a problem C is NP-complete?

- Following the definition may be tedious as we need to show that:
  1. $C \in NP$ ("easy")
  2. C is NP-hard, i.e., $\forall A \in NP,\ A \leq_P C$
- Recall that we proved that if B is NP-complete and $B \leq_P C$ then C is NP-complete

Proving *NP*-completeness: Summary

How to prove that a problem C is NP-complete?

- We therefore need to show that:
  1. $C \in NP$
  2. a well-known NP-complete problem B polynomial time reduces to C (e.g., $SAT \leq_P C$ or $3SAT \leq C$)
  3. the reduction actually takes polynomial time

CLIQUE is *NP*-complete

This results follows directly from the previous findings

CLIQUE is *NP*-complete

This results follows directly from the previous findings

- We showed that $CLIQUE \in NP$

CLIQUE is *NP*-complete

This results follows directly from the previous findings

- We showed that $CLIQUE \in NP$
- We showed that $3SAT \leq_P CLIQUE$

CLIQUE is *NP*-complete

This results follows directly from the previous findings

- We showed that $CLIQUE \in NP$
- We showed that $3SAT \leq_P CLIQUE$
- We showed that $3SAT$ is NP-complete

CLIQUE is *NP*-complete

This results follows directly from the previous findings

- We showed that $CLIQUE \in NP$
- We showed that $3SAT \leq_P CLIQUE$
- We showed that $3SAT$ is NP-complete
- Thus, $CLIQUE$ is NP-complete!

Karp's 21 *NP*-Complete Problems (1972)

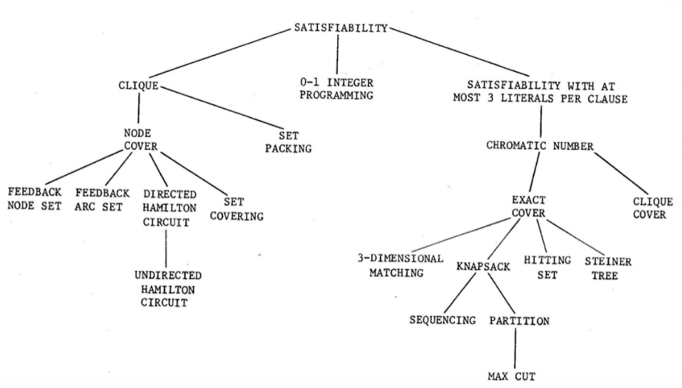


Figure: Karp's 21 *NP*-complete Problems

NP-hard Optimization Problems

- Decision problems have YES/NO answers

NP-hard Optimization Problems

- Decision problems have YES/NO answers
- Many decision problems have corresponding optimization version

NP-hard Optimization Problems

- Decision problems have YES/NO answers
- Many decision problems have corresponding optimization version
- Optimization version of *NP*-complete problems are *NP*-hard

Problem	Decision Version	Optimization Version
CLIQUE	Does a graph G have a clique of size k ?	Find largest clique
ILP	Does $\exists$ integer vector y such that $Ay \leq b$?	Find integer vector y to max $d^\top y$ s.t. $Ay \leq b$
TSP	Does a graph G have tour of length $\leq d$?	Find min length tour
Scheduling	Given set of tasks and constraints, can we finish all tasks in time d ?	Find min time schedule

Why are *NP*-complete and *NP*-hard Important?

- Suppose you are faced with a problem and you can't come up with an efficient algorithm for it

Why are *NP*-complete and *NP*-hard Important?

- Suppose you are faced with a problem and you can't come up with an efficient algorithm for it
- If you can prove the problem is *NP*-complete or *NP*-hard, then there is no known efficient algorithm to solve it
  - No known polynomial-time algorithms for *NP*-complete and *NP*-hard problems!

Why are *NP*-complete and *NP*-hard Important?

- Suppose you are faced with a problem and you can't come up with an efficient algorithm for it
- If you can prove the problem is *NP*-complete or *NP*-hard, then there is no known efficient algorithm to solve it
  - No known polynomial-time algorithms for *NP*-complete and *NP*-hard problems!
- How to deal with an *NP*-complete or *NP*-hard problem?
  - Approximation algorithm
  - Probabilistic algorithm
  - Special cases
  - Heuristic

Table of Contents

Summary

- Class P comprises problems that can be **decided** in polynomial time (e.g., $PATH \in P$)

Summary

- Class P comprises problems that can be **decided** in polynomial time (e.g., $PATH \in P$)
- Class NP contains problems that can be **verified** in deterministic polynomial time (or, equivalently, **decided** in non-deterministic polynomial time)

Summary

- Class P comprises problems that can be **decided** in polynomial time (e.g., $PATH \in P$)
- Class NP contains problems that can be **verified** in deterministic polynomial time (or, equivalently, **decided** in non-deterministic polynomial time)
- NP includes all problems that are in P plus, for example, $HAMPATH$, $CLIQUE$, $SUBSET\text{-}SUM$, $3SAT$, etc.

Summary

- Class P comprises problems that can be **decided** in polynomial time (e.g., $PATH \in P$)
- Class NP contains problems that can be **verified** in deterministic polynomial time (or, equivalently, **decided** in non-deterministic polynomial time)
- NP includes all problems that are in P plus, for example, $HAMPATH$, $CLIQUE$, $SUBSET\text{-}SUM$, $3SAT$, etc.
- P vs. NP question:
  1. We know that $P \subseteq NP$
  2. We **don't** know whether $P \neq NP$ or $P = NP$

Summary

- Polynomial-time mapping reducibility: $A \leq_P B$ if exists polynomial-time computable function f such that:

$$w \in A \Leftrightarrow f(w) \in B$$

Summary

- Polynomial-time mapping reducibility: $A \leq_P B$ if exists polynomial-time computable function f such that:

$$w \in A \Leftrightarrow f(w) \in B$$

- A language B is *NP*-complete if $B \in NP$ and $A \leq_P B$ **for all** $A \in NP$

Summary

- Polynomial-time mapping reducibility: $A \leq_P B$ if exists polynomial-time computable function f such that:

$$w \in A \Leftrightarrow f(w) \in B$$

- A language B is *NP*-complete if $B \in NP$ and $A \leq_P B$ **for all** $A \in NP$
- If any *NP*-complete language B is in P, then $P = NP$

Summary

- Polynomial-time mapping reducibility: $A \leq_P B$ if exists polynomial-time computable function f such that:

$$w \in A \Leftrightarrow f(w) \in B$$

- A language B is NP-complete if $B \in NP$ and $A \leq_P B$ **for all** $A \in NP$
- If any NP-complete language B is in P, then $P = NP$
- If any NP language B is **not** in P, then $P \neq NP$ (in particular, we could show it when B is also NP-complete)

Summary

- Polynomial-time mapping reducibility: $A \leq_P B$ if exists polynomial-time computable function f such that:

$$w \in A \Leftrightarrow f(w) \in B$$

- A language B is *NP*-complete if $B \in NP$ and $A \leq_P B$ **for all** $A \in NP$
- If any *NP*-complete language B is in P, then $P = NP$
- If any *NP* language B is **not** in P, then $P \neq NP$ (in particular, we could show it when B is also *NP*-complete)
- If B is *NP*-complete and $B \leq_P C$ for $C \in NP$, then C is *NP*-complete

Summary

- Polynomial-time mapping reducibility: $A \leq_P B$ if exists polynomial-time computable function f such that:

$$w \in A \Leftrightarrow f(w) \in B$$

- A language B is NP-complete if $B \in NP$ and $A \leq_P B$ **for all** $A \in NP$
- If any NP-complete language B is in P, then $P = NP$
- If any NP language B is **not** in P, then $P \neq NP$ (in particular, we could show it when B is also NP-complete)
- If B is NP-complete and $B \leq_P C$ for $C \in NP$, then C is NP-complete
- **Cook-Levin Theorem:** SAT is NP-complete

Summary

- Polynomial-time mapping reducibility: $A \leq_P B$ if exists polynomial-time computable function f such that:

$$w \in A \Leftrightarrow f(w) \in B$$

- A language B is *NP*-complete if $B \in NP$ and $A \leq_P B$ **for all** $A \in NP$
- If any *NP*-complete language B is in P, then $P = NP$
- If any *NP* language B is **not** in P, then $P \neq NP$ (in particular, we could show it when B is also *NP*-complete)
- If B is *NP*-complete and $B \leq_P C$ for $C \in NP$, then C is *NP*-complete
- **Cook-Levin Theorem:** *SAT* is *NP*-complete
- 3*SAT*, *CLIQUE*, *SUBSET-SUM*, *HAMPATH*, etc. are all *NP*-complete (via polynomial time reduction)