NP and NP Completeness

Lecture 23

April 20, 2023

Part I

NP

P and NP and Turing Machines

1. ***P***: set of decision problems that have polynomial time algorithms.
2. ***NP***: set of decision problems that have polynomial time *non-deterministic* algorithms.

- Many natural problems we would like to solve are in ***NP***.
- Every problem in ***NP*** has an exponential time algorithm
- $P \subseteq NP$
- Some problems in ***NP*** are in ***P*** (example, shortest path problem)

P and NP and Turing Machines

1. P: set of decision problems that have polynomial time algorithms.
2. NP: set of decision problems that have polynomial time *non-deterministic* algorithms.

- Many natural problems we would like to solve are in NP.
- Every problem in NP has an exponential time algorithm
- $P \subseteq NP$
- Some problems in NP are in P (example, shortest path problem)

Big Question: Does every problem in NP have an efficient algorithm? Same as asking whether $P = NP$.

No known efficient algorithms for

Problems

1. **Independent Set**
2. **Vertex Cover**
3. **Set Cover**
4. **SAT**
5. **3SAT**

There are of course undecidable problems (no algorithm at all!) but many problems that we want to solve are of similar flavor to the above.

Question: What property do the above problems share?

Efficient Checkability

Above problems share the following feature:

Checkability (informal)

For any YES instance I_X *of* X *there is a proof/certificate/solution that is "short" and will convince an efficient checker. For a NO instance* I_X*, no proof will convince the checker.*

Efficient Checkability

Above problems share the following feature:

Checkability (informal)

For any YES instance I_X of X there is a proof/certificate/solution that is "short" and will convince an efficient checker. For a NO instance I_X, no proof will convince the checker.

Examples:

1. **SAT** formula φ: proof is a satisfying assignment. Verifier checks whether assignment satisfies all the clauses.
2. **Independent Set** in graph G and k: a subset S of vertices. Verifier checks whether S is independent in G
3. **Homework**

Sudoku

			2	5				
	3	6		4		8		
	4					1	6	
2								
7	6						1	9
								3
	1	5					7	
		9		8		2	4	
				3	7			

Given $n \times n$ sudoku puzzle, does it have a solution?

Certifiers

Definition

An algorithm $C(\cdot, \cdot)$ is a **certifier** for problem X if the following two conditions hold:

- For every $w \in X$ there is some string t such that $C(w, t) =$ "yes" (t is a certificate or proof or witness for w)
- If $w \notin X$, $C(w, t) =$ "no" for *every* string t.

Efficient (polynomial time) Certifiers

Definition (Efficient Certifier.)

A certifier C is an **efficient certifier** for problem X if there is a polynomial $p(\cdot)$ such that the following conditions hold:

- For every $w \in X$ there is some string t such that $C(w, t) =$ "yes" *and* $|t| \leq p(|s|)$.
- If $w \notin X$, $C(s, t) =$ "no" for every t.
- $C(w, t)$ runs in time polynomial in its input size $(|w| + |t|)$

Example: Independent Set

1. Problem: Given graph $G = (V, E)$, does G have an independent set of size $\geq k$?
   1. Certificate: Set $S \subseteq V$.
   2. Certifier: Check $|S| \geq k$ and no pair of vertices in S is connected by an edge.

Example: Vertex Cover

1. Problem: Does G have a vertex cover of size $\leq k$?
   1. Certificate: $S \subseteq V$.
   2. Certifier: Check $|S| \leq k$ and that for every edge at least one endpoint is in S.

Example: SAT

1. Problem: Does formula φ have a satisfying truth assignment?
   1. Certificate: Assignment $\boldsymbol{a}$ of $0/1$ values to each variable.
   2. Certifier: Check each clause under $\boldsymbol{a}$ and say "yes" if all clauses are true.

Example: Composites

Problem: Composite

Instance: A number n **Question:** Is the number n composite?

1. Problem: **Composite**.
    1. Certificate: A factor $t \leq n$ such that $t \neq 1$ and $t \neq n$.
    2. Certifier: Check that t divides n

Example: NFA Universality

Problem: NFA Universality

> **Instance:** Description of a NFA M.
> **Question:** Is $L(M) = \Sigma^*$, that is, does M accept all strings?

1. Problem: **NFA Universality**.
   1. Certificate: A DFA M' equivalent to M
   2. Certifier: Check that $L(M') = \Sigma^*$ (all states accept)

Example: NFA Universality

Problem: NFA Universality

> **Instance:** Description of a NFA M.
> **Question:** Is $L(M) = \Sigma^*$, that is, does M accept all strings?

1. Problem: **NFA Universality**.
   1. Certificate: A DFA M' equivalent to M
   2. Certifier: Check that $L(M') = \Sigma^*$ (all states accept)

Certifier is efficient but certificate is not necessarily short! We do not know if the problem is in NP.

Example: A String Problem

Problem: PCP

Instance: Two sets of binary strings $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$
Question: Are there indices $i_1, i_2, \ldots, i_k$ such that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

1. Problem: **PCP**
   1. Certificate: A sequence of indices $i_1, i_2, \ldots, i_k$
   2. Certifier: Check that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

Example: A String Problem

Problem: PCP

Instance: Two sets of binary strings $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$
Question: Are there indices $i_1, i_2, \ldots, i_k$ such that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

1. Problem: **PCP**
    1. Certificate: A sequence of indices $i_1, i_2, \ldots, i_k$
    2. Certifier: Check that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

PCP = Posts Correspondence Problem and it is undecidable!
Implies no finite bound on length of certificate!

Nondeterministic Polynomial Time

Definition

Nondeterministic Polynomial Time (denoted by **NP**) is the class of all problems that have efficient certifiers.

Nondeterministic Polynomial Time

Definition

Nondeterministic Polynomial Time (denoted by **NP**) is the class of all problems that have efficient certifiers.

Example

Independent Set, **Vertex Cover**, **Set Cover**, **SAT**, **3SAT**, and **Composite** are all examples of problems in **NP**.

Why is it called...

Nondeterministic Polynomial Time

A certifier is an algorithm $C(w, t)$ with two inputs:

1. w: instance.
2. t: proof/certificate that the instance is indeed a YES instance of the given problem.

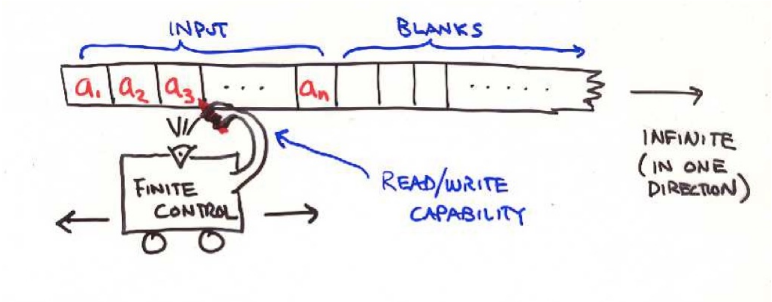
One can think about C as an algorithm for the original problem, if:

1. Given w, the algorithm *guesses* (non-deterministically, and who knows how) a certificate t.
2. The algorithm now verifies the certificate t for the instance w.

NP can be equivalently described using Turing machines.

Non-Deterministic TMs

- NFA with infinite tap
- One move: read, write, move one cell, change state but now NFA allows multiple choices



Non-deterministic TM M accepts w if *there exist some choice of moves* leading to M halting in an accept state.

Asymmetry in Definition of NP

Note that only YES instances have a short proof/certificate. NO instances need not have a short certificate.

Example

SAT formula φ. No easy way to prove that φ is NOT satisfiable!

More on this and **co-NP** later on.

P versus NP

Proposition

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

P versus NP

Proposition

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

For a problem in **P** no need for a certificate!

Proof.

Consider problem $X \in \mathbf{P}$ with algorithm A. Need to demonstrate that X has an efficient certifier:

1. Certifier C on input s, t, runs $A(s)$ and returns the answer.
2. C runs in polynomial time.
3. If $s \in X$, then for every t, $C(s, t) =$ "yes".
4. If $s \not\in X$, then for every t, $C(s, t) =$ "no". □

Exponential Time

Definition

Exponential Time (denoted **EXP**) is the collection of all problems that have an algorithm which on input s runs in exponential time, i.e., $O(2^{\mathrm{poly}(|s|)})$.

Exponential Time

Definition

Exponential Time (denoted **EXP**) is the collection of all problems that have an algorithm which on input s runs in exponential time, i.e., $O(2^{\text{poly}(|s|)})$.

Example: $O(2^n)$, $O(2^{n \log n})$, $O(2^{n^3})$, ...

NP versus EXP

Proposition

NP $\subseteq$ **EXP**.

Proof.

Let $X \in$ **NP** with certifier C. Need to design an exponential time algorithm for X.

1. For every t, with $|t| \leq p(|s|)$ run $C(s, t)$; answer "yes" if any one of these calls returns "yes".
2. The above algorithm correctly solves X (exercise).
3. Algorithm runs in $O(q(|s| + |p(s)|)2^{p(|s|)})$, where q is the running time of C. □

Examples

1. **SAT**: try all possible truth assignment to variables.
2. **Independent Set**: try all possible subsets of vertices.
3. **Vertex Cover**: try all possible subsets of vertices.

Is NP efficiently solvable?

We know **P** $\subseteq$ **NP** $\subseteq$ **EXP**.

Is NP efficiently solvable?

We know **P** $\subseteq$ **NP** $\subseteq$ **EXP**.

Big Question

Is there are problem in **NP** that does not belong to **P**? Is **P** $=$ **NP**?

If P = NP

Or: If pigs could fly then life would be sweet.

- Many important optimization problems can be solved efficiently.
- The RSA cryptosystem can be broken.
- No security on the web.
- No e-commerce ...
- Creativity can be automated! Proofs for mathematical statement can be found by computers automatically (if short ones exist).

If P = NP this implies that...

- **Vertex Cover** can be solved in polynomial time.
- $\mathsf{P} = \mathsf{EXP}$.
- $\mathsf{EXP} \subseteq \mathsf{P}$.
- All of the above.

P versus NP

Status

Relationship between **P** and **NP** remains one of the most important open problems in mathematics/computer science.

Consensus: Most people feel/believe $\boldsymbol{P \neq NP}$.

Resolving **P** versus **NP** is a Clay Millennium Prize Problem. You can win a million dollars in addition to a Turing award and major fame!

Part II

NP-Completeness

“Hardest” Problems

Question

What is the hardest problem in **NP**? How do we define it?

Towards a definition

1. Hardest problem must be in **NP**.
2. Hardest problem must be at least as “difficult” as every other problem in **NP**.

NP Complete Problem

Definition

A problem X is said to be **NP-Complete** if

1. $X \in \mathbf{NP}$, and
2. (Hardness) For any $Y \in \mathbf{NP}$, $\mathbf{Y} \leq_P \mathbf{X}$.

Solving NP-Complete Problems

Proposition

*Suppose X is **NP-Complete**. Then X can be solved in polynomial time if and only if $\mathbf{P} = \mathbf{NP}$.*

Proof.

$\Rightarrow$ Suppose X can be solved in polynomial time

1. Let $Y \in \mathbf{NP}$. We know $Y \leq_P X$.
2. We showed that if $Y \leq_P X$ and X can be solved in polynomial time, then Y can be solved in polynomial time.
3. Thus, every problem $Y \in \mathbf{NP}$ is such that $Y \in P$; $NP \subseteq P$.
4. Since $\mathbf{P} \subseteq \mathbf{NP}$, we have $\mathbf{P} = \mathbf{NP}$.

$\Leftarrow$ Since $\mathbf{P} = \mathbf{NP}$, and $X \in \mathbf{NP}$, we have a polynomial time algorithm for X. □

NP-Hard Problems

Definition

A problem X is said to be **NP-Hard** if

1. (Hardness) For any $Y \in \mathbf{NP}$, we have that $\mathbf{Y} \leq_P \mathbf{X}$.

An **NP-Hard** problem need not be in **NP**!

Example: Halting problem is **NP-Hard** (why?) but not **NP-Complete**.

Consequences of proving NP-Completeness

If X is **NP-Complete**

1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
2. and solving X implies $\mathbf{P} = \mathbf{NP}$.

X is unlikely to be efficiently solvable.

Consequences of proving NP-Completeness

If X is **NP-Complete**

1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
2. and solving X implies $\mathbf{P} = \mathbf{NP}$.

X is unlikely to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for X.

Consequences of proving NP-Completeness

If X is **NP-Complete**

1. Since we believe **P** $\neq$ **NP**,
2. and solving X implies **P** $=$ **NP**.

X is unlikely to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for X.

Consequences of proving NP-Completeness

If X is **NP-Complete**

1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
2. and solving X implies $\mathbf{P} = \mathbf{NP}$.

X is unlikely to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for X.
(This is proof by mob opinion — take with a grain of salt.)

NP Complete Problems

Question

Are there any problems that are **NP-Complete**?

Answer

Yes! Many, many problems are **NP-Complete**.

Cook-Levin Theorem

Theorem (Cook-Levin)

SAT *is* **NP-Complete***.*

Cook-Levin Theorem

Theorem (Cook-Levin)

SAT *is* **NP-Complete**.

Need to show

1. **SAT** is in **NP**.
2. *every* **NP** problem X reduces to **SAT**.

Steve Cook won the Turing award for his theorem.

Proving that a problem X is NP Complete

To prove X is **NP-Complete**, show

1. Show that X is in **NP**.
2. Give a polynomial-time reduction *from* a known **NP-Complete** problem such as **SAT** *to* X

Proving that a problem X is NP Complete

To prove X is **NP-Complete**, show

1. Show that X is in **NP**.
2. Give a polynomial-time reduction *from* a known **NP-Complete** problem such as **SAT** *to* X

SAT $\leq_P X$ implies that every **NP** problem $Y \leq_P X$. Why?

Proving that a problem X is NP Complete

To prove X is **NP-Complete**, show

1. Show that X is in **NP**.
2. Give a polynomial-time reduction *from* a known **NP-Complete** problem such as **SAT** *to* X

SAT $\leq_P X$ implies that every **NP** problem $Y \leq_P X$. Why? Transitivity of reductions:

$Y \leq_P SAT$ and $SAT \leq_P X$ and hence $Y \leq_P X$.

3-SAT is NP-Complete

- **3-SAT** is in NP
- **SAT** $\leq_P$ **3-SAT** as we saw

NP-Completeness via Reductions

1. **SAT** is **NP-Complete** due to Cook-Levin theorem
2. **SAT** $\leq_P$ **3-SAT**
3. **3-SAT** $\leq_P$ **Independent Set**
4. **Independent Set** $\leq_P$ **Vertex Cover**
5. **Independent Set** $\leq_P$ **Clique**
6. **3-SAT** $\leq_P$ **3-Color**
7. **3-SAT** $\leq_P$ **Hamiltonian Cycle**

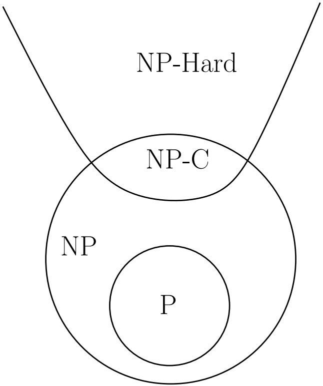
NP-Completeness via Reductions

1. **SAT** is **NP-Complete** due to Cook-Levin theorem
2. **SAT** $\leq_P$ **3-SAT**
3. **3-SAT** $\leq_P$ **Independent Set**
4. **Independent Set** $\leq_P$ **Vertex Cover**
5. **Independent Set** $\leq_P$ **Clique**
6. **3-SAT** $\leq_P$ **3-Color**
7. **3-SAT** $\leq_P$ **Hamiltonian Cycle**

Hundreds and thousands of different problems from many areas of science and engineering have been shown to be **NP-Complete**.

A surprisingly frequent phenomenon!

P, NP, NP-Complete, NP-Hard



Part III

3-SAT to Independent Set Reduction

Independent Set

Problem: Independent Set

Instance: A graph G, integer k.
Question: Is there an independent set in G of size k?

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a $\mathrm{3CNF}$ formula φ
Goal: Construct a graph G_φ and number k such that G_φ has an independent set of size k if and only if φ is satisfiable.

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a $3\mathrm{CNF}$ formula φ
Goal: Construct a graph G_φ and number k such that G_φ has an independent set of size k if and only if φ is satisfiable.
G_φ should be constructable in time polynomial in size of φ

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a $3\mathrm{CNF}$ formula φ
Goal: Construct a graph G_φ and number k such that G_φ has an independent set of size k if and only if φ is satisfiable.
G_φ should be constructable in time polynomial in size of φ

Importance of reduction: Although **3SAT** is much more expressive, it can be reduced to a seemingly specialized Independent Set problem.

Interpreting 3SAT

There are two ways to think about **3SAT**

Interpreting 3SAT

There are two ways to think about **3SAT**

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.

Interpreting 3SAT

There are two ways to think about **3SAT**

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.
2. Pick a literal from each clause and find a truth assignment to make all of them true

Interpreting 3SAT

There are two ways to think about **3SAT**

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.
2. Pick a literal from each clause and find a truth assignment to make all of them true. You will fail if two of the literals you pick are in conflict, i.e., you pick x_i and $\neg x_i$

We will take the second view of **3SAT** to construct the reduction.

The Reduction

1. G_φ will have one vertex for each literal in a clause

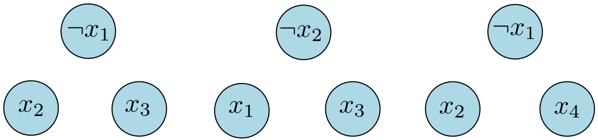


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true

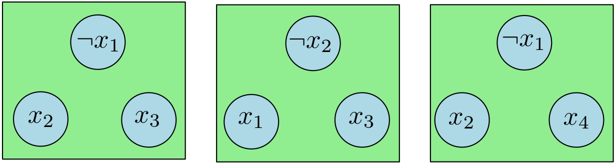


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true

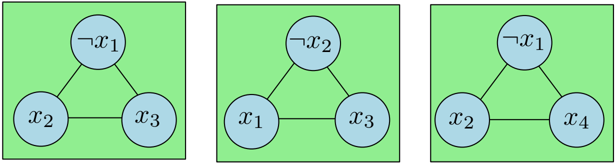


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict

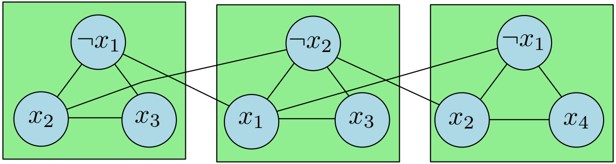


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict
4. Take k to be the number of clauses

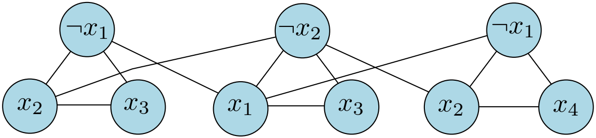


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

Reduction

Lemma

Reduction is efficient. That is, there is a polynomial-time algorithm that given a 3SAT formula φ *outputs* $\langle G_\varphi, k \rangle$.

Easy to see.

Correctness

Proposition

φ is satisfiable iff G_φ has an independent set of size k ($=$ number of clauses in φ).

Proof.

$\Rightarrow$ Let a be the truth assignment satisfying φ

Correctness

Proposition

φ is satisfiable iff G_φ has an independent set of size k (= number of clauses in φ).

Proof.

$\Rightarrow$ Let a be the truth assignment satisfying φ

1. Pick one of the vertices, corresponding to true literals under a, from each triangle. This is an independent set of the appropriate size. Why? □

Correctness (contd)

Proposition

φ is satisfiable iff G_φ has an independent set of size k (= number of clauses in φ).

Proof.

$\Leftarrow$ Let S be an independent set of size k

1. S must contain *exactly* one vertex from each clause
2. S cannot contain vertices labeled by conflicting literals
3. Thus, it is possible to obtain a truth assignment that makes the literals in S true; such an assignment satisfies one literal in every clause □