SAT, NP, NP-Completeness

Lecture 22
April 13, 2018

Most slides are courtesy Prof. Chekuri

Part I

Reductions Continued

II: Partition and subset sum?

Problem: Partition

Instance: A set S of n numbers.
Question: Is there a subset $T \subseteq S$ s.t. $\sum_{t \in T} t = \sum_{s \in S \setminus T} s$?

Problem: Subset Sum

Instance: S - set of positive integers, t: - an integer number (target).
Question: Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

Assume that we can solve **Partition** in polynomial time, then we can solve **Subset Sum** in polynomial time. This statement is

(A) True.

(B) Mostly true.

(C) False.

(D) Mostly false.

Polynomial Time Reduction

Karp reduction

$X \leq_P Y$: *algorithm* $\mathcal{A}$ reduces problem X to problem Y in polynomial-time:

1. given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y
2. $\mathcal{A}$ runs in time $poly(|I_X|) \Rightarrow |I_Y| = poly(|I_X|)$
3. Answer to I_X YES *iff* answer to I_Y is YES.

Polynomial Time Reduction

Karp reduction

$X \leq_P Y$: *algorithm* $\mathcal{A}$ reduces problem X to problem Y in polynomial-time:

1. given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y
2. $\mathcal{A}$ runs in time $poly(|I_X|) \Rightarrow |I_Y| = poly(|I_X|)$
3. Answer to I_X YES *iff* answer to I_Y is YES.

Consequences:

- poly-time algorithm for $Y \Rightarrow$ poly-time algorithm for X.
- X is "hard" $\Rightarrow$ Y is "hard".

Polynomial Time Reduction

Karp reduction

$X \leq_P Y$: *algorithm* $\mathcal{A}$ reduces problem X to problem Y in polynomial-time:

1. given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y
2. $\mathcal{A}$ runs in time $poly(|I_X|) \Rightarrow |I_Y| = poly(|I_X|)$
3. Answer to I_X YES *iff* answer to I_Y is YES.

Consequences:

- poly-time algorithm for $Y \Rightarrow$ poly-time algorithm for X.
- X is "hard" $\Rightarrow$ Y is "hard".

Note. $X \leq_P Y \quad \not\Longrightarrow \quad Y \leq_P X$

A More General Reduction

Turing Reduction

Definition (Turing reduction.)

Problem X polynomial time reduces to Y if there is an algorithm $\mathcal{A}$ for X that has the following properties:

1. on any given instance I_X of X, $\mathcal{A}$ uses polynomial in $|I_X|$ "steps"
2. a step is either a standard computation step, or
3. a sub-routine call to an algorithm that solves Y.

This is a **Turing reduction**.

A More General Reduction

Turing Reduction

Definition (Turing reduction.)

Problem X polynomial time reduces to Y if there is an algorithm $\mathcal{A}$ for X that has the following properties:

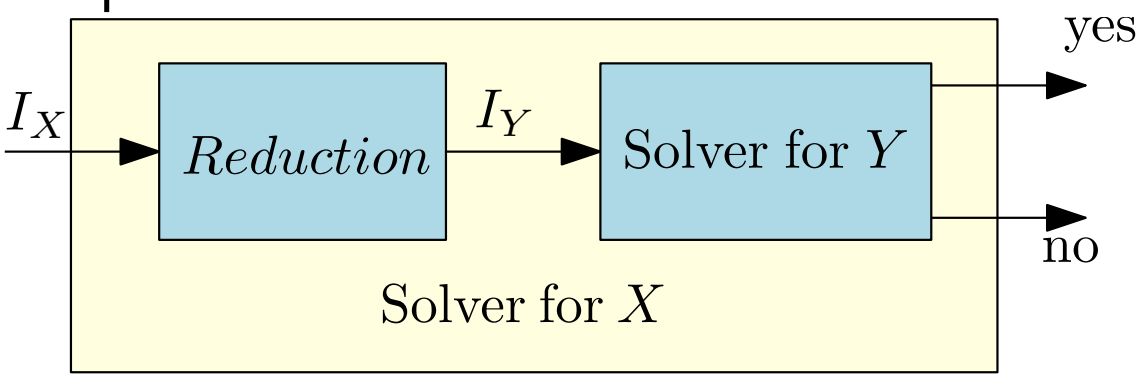
1. on any given instance I_X of X, $\mathcal{A}$ uses polynomial in $|I_X|$ "steps"
2. a step is either a standard computation step, or
3. a sub-routine call to an algorithm that solves Y.

This is a **Turing reduction**.

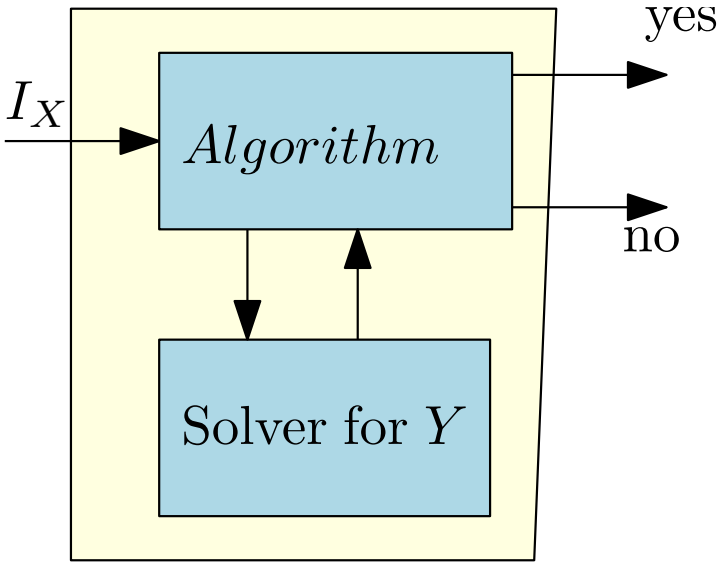
Note: In making sub-routine call to algorithm to solve Y, $\mathcal{A}$ can only ask questions of size polynomial in $|I_X|$. Why?

Comparing reductions

1. Karp reduction:



2. Turing reduction:



Turing reduction

1. Algorithm to solve X can call solver for Y $poly(|I_x|)$ many times.
2. Input to every call is of size $poly(|I_x|)$.

Example of Turing Reduction

Problem (Independent set in circular arcs graph.)

Input: *Collection of arcs on a circle.*
Goal: *Compute the maximum number of non-overlapping arcs.*

Reduced to the following problem:?

Problem (Independent set of intervals.)

Input: *Collection of intervals on the line.*
Goal: *Compute the maximum number of non-overlapping intervals.*

How?

Example of Turing Reduction

Problem (Independent set in circular arcs graph.)

Input: *Collection of arcs on a circle.*
Goal: *Compute the maximum number of non-overlapping arcs.*

Reduced to the following problem:?

Problem (Independent set of intervals.)

Input: *Collection of intervals on the line.*
Goal: *Compute the maximum number of non-overlapping intervals.*

How? Used algorithm for interval problem multiple times.

Turing vs Karp Reductions

1. Turing reductions more general than Karp reductions.
2. Turing reduction useful in obtaining algorithms via reductions.
3. Karp reduction is simpler and easier to use to prove hardness of problems.
4. Perhaps surprisingly, Karp reductions, although limited, suffice for most known **NP-Complete**ness proofs.
5. Karp reductions allow us to distinguish between **NP** and **co-NP** (more on this later).

Part II

The Satisfiability Problem (SAT)

Propositional Formulas

Definition

Consider a set of boolean variables $x_1, x_2, \ldots x_n$.

1. A **literal** is either a boolean variable x_i or its negation $\neg x_i$.

Propositional Formulas

Definition

Consider a set of boolean variables $x_1, x_2, \ldots x_n$.

1. A **literal** is either a boolean variable x_i or its negation $\neg x_i$.
2. A **clause** is a disjunction of literals.
   For example, $x_1 \vee x_2 \vee \neg x_4$ is a clause.

Propositional Formulas

Definition

Consider a set of boolean variables $x_1, x_2, \ldots x_n$.

1. A **literal** is either a boolean variable x_i or its negation $\neg x_i$.
2. A **clause** is a disjunction of literals.
   For example, $x_1 \vee x_2 \vee \neg x_4$ is a clause.
3. A **formula in conjunctive normal form** (CNF) is propositional formula which is a conjunction of clauses
   1. $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is a CNF formula.

Propositional Formulas

Definition

Consider a set of boolean variables $x_1, x_2, \ldots x_n$.

1. A **literal** is either a boolean variable x_i or its negation $\neg x_i$.
2. A **clause** is a disjunction of literals.
   For example, $x_1 \vee x_2 \vee \neg x_4$ is a clause.
3. A **formula in conjunctive normal form** (CNF) is propositional formula which is a conjunction of clauses
   1. $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is a CNF formula.
4. A formula φ is a 3CNF:
   A CNF formula such that every clause has **exactly** 3 literals.
   1. $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_1)$ is a 3CNF formula, but $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is not.

Satisfiability

Problem: SAT

Instance: A CNF formula φ.
Question: Is there a truth assignment to the variable of φ such that φ evaluates to true?

Problem: 3SAT

Instance: A $3\mathrm{CNF}$ formula φ.
Question: Is there a truth assignment to the variable of φ such that φ evaluates to true?

Satisfiability

SAT

Given a CNF formula φ, is there a truth assignment to variables such that φ evaluates to true?

Example

1. $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is satisfiable; take $x_1, x_2, \ldots x_5$ to be all true
2. $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ is not satisfiable.

Importance of SAT and 3SAT

1. **SAT** and **3SAT** are basic constraint satisfaction problems.
2. Many different problems can reduced to them because of the simple yet powerful expressively of logical constraints.
3. Arise naturally in many applications involving hardware and software verification and correctness.
4. As we will see, it is a fundamental problem in theory of **NP-Completeness**.

3SAT $\leq_P$ SAT

1. **3SAT** $\leq_P$ **SAT**.
2. Because...
   A **3SAT** instance is also an instance of **SAT**.

SAT $\leq_P$ 3SAT

Claim

SAT $\leq_P$ 3SAT.

SAT $\leq_P$ 3SAT

Claim

SAT $\leq_P$ 3SAT.

Given φ a **SAT** formula we create a **3SAT** formula φ' such that

1. φ is satisfiable iff φ' is satisfiable.
2. φ' can be constructed from φ in time polynomial in $|\varphi|$.

SAT $\leq_P$ 3SAT

How SAT is different from 3SAT?

In **SAT** clauses might have arbitrary length: $\mathbf{1, 2, 3, \ldots}$ variables:

$$\Big(x \vee y \vee z \vee w\Big) \wedge \Big(\neg x \vee \neg y \vee \neg z \vee w \vee u\Big) \wedge \Big(\neg x\Big)$$

In **3SAT** every clause must have **exactly 3** different literals.

$SAT \leq_P 3SAT$

How SAT is different from 3SAT?

In SAT clauses might have arbitrary length: $1, 2, 3, \ldots$ variables:

$$\left(x \vee y \vee z \vee w\right) \wedge \left(\neg x \vee \neg y \vee \neg z \vee w \vee u\right) \wedge \left(\neg x\right)$$

In 3SAT every clause must have **exactly 3** different literals.

Consider $(x \vee y \vee z \vee w)$

Replace it with $\left(x \vee y \vee \alpha\right) \wedge \left(\neg\alpha \vee w \vee u\right)$

SAT $\leq_P$ 3SAT

How SAT is different from 3SAT?

In SAT clauses might have arbitrary length: $1, 2, 3, \ldots$ variables:

$$\Big(x \vee y \vee z \vee w\Big) \wedge \Big(\neg x \vee \neg y \vee \neg z \vee w \vee u\Big) \wedge \Big(\neg x\Big)$$

In 3SAT every clause must have **exactly 3** different literals.

Consider $(x \vee y \vee z \vee w)$

Replace it with $\Big(x \vee y \vee \alpha\Big) \wedge \Big(\neg\alpha \vee w \vee u\Big)$

1. Pad short clauses so they have 3 literals.
2. Break long clauses into shorter clauses. (Need to add new variables)
3. Repeat the above till we have a 3CNF.

What about **2SAT**?

What about **2SAT**?

2SAT can be solved in polynomial time! (specifically, linear time!)

What about **2SAT**?

2SAT can be solved in polynomial time! (specifically, linear time!)

No known polynomial time reduction from **SAT** (or **3SAT**) to **2SAT**. If there was, then **SAT** and **3SAT** would be solvable in polynomial time.

What about 2SAT?

2SAT can be solved in polynomial time! (specifically, linear time!)

No known polynomial time reduction from **SAT** (or **3SAT**) to **2SAT**. If there was, then **SAT** and **3SAT** would be solvable in polynomial time.

Why the reduction from 3SAT to 2SAT fails?

Consider a clause $(x \vee y \vee z)$. We need to reduce it to a collection of **2**CNF clauses. Introduce a face variable α, and rewrite this as

$$(x \vee y \vee \alpha) \wedge (\neg\alpha \vee z) \quad \text{(bad! clause with 3 vars)}$$

$$\text{or} \quad (x \vee \alpha) \wedge (\neg\alpha \vee y \vee z) \quad \text{(bad! clause with 3 vars).}$$

(In animal farm language: **2SAT** good, **3SAT** bad.)

What about 2SAT?

A challenging exercise: Given a 2SAT formula show to compute its satisfying assignment...

Look in books etc.

Independent Set

Problem: Independent Set

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

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

Later (if time permits)

Part III

Definition of P and NP

Recap . . .

Problems

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

Recap . . .

Problems

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

Relationship

3SAT $\leq_P$ Independent Set

Recap . . .

Problems

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

Relationship

3SAT $\leq_P$ **Independent Set** $\overset{\leq_P}{\geq_P}$ **Vertex Cover**

Recap . . .

Problems

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

Relationship

$$\textbf{3SAT} \leq_P \textbf{Independent Set} \overset{\leq_P}{\geq_P} \textbf{Vertex Cover} \leq_P \textbf{Set Cover}$$

Recap . . .

Problems

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

Relationship

3SAT $\leq_P$ Independent Set $\overset{\leq_P}{\geq_P}$ Vertex Cover $\leq_P$ Set Cover
3SAT $\leq_P$ SAT $\leq_P$ 3SAT

Problems and Algorithms: Formal Approach

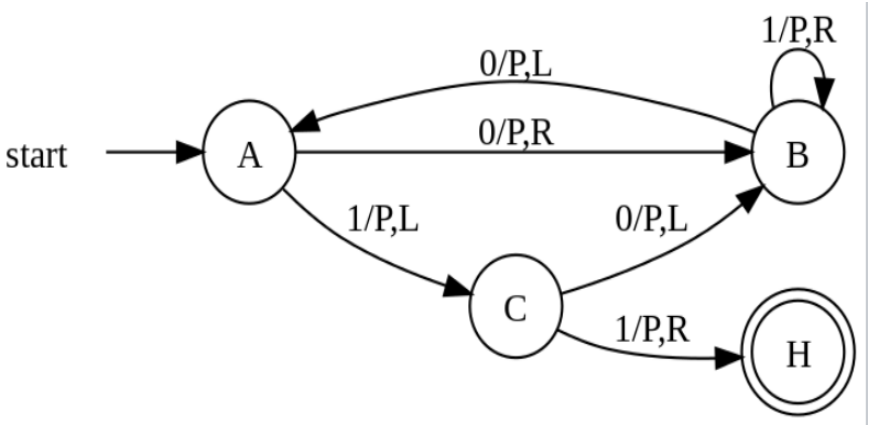
Decision Problems

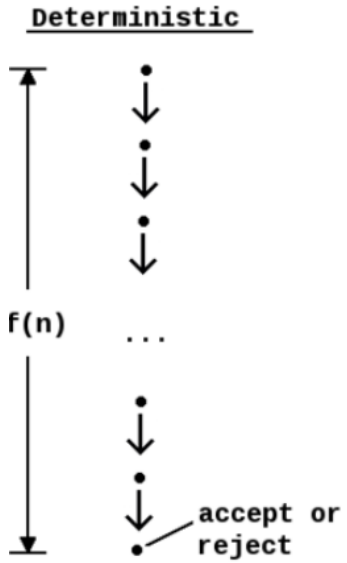
1. Problem Instance: Binary string s, with size $|s|$
2. Problem: A set X of strings on which the answer should be "yes"; we call these YES instances of X. Strings not in X are NO instances of X.

Definition

1. A is an algorithm for problem X if $A(I_x) =$ "yes" iff $I_x \in X$.
2. A is said to have a polynomial running time if there is a polynomial $p(\cdot)$ such that for every string I_x, $A(I_x)$ terminates in at most $O(p(|I_x|))$ steps.

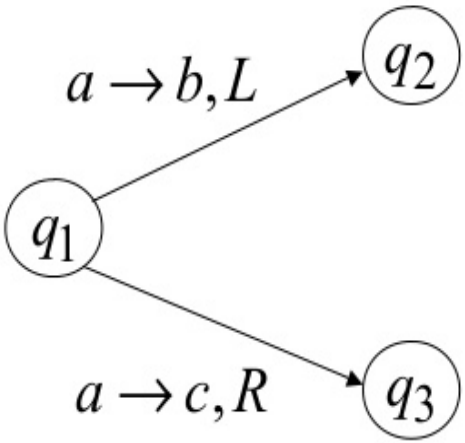
Deterministic Turing Machine



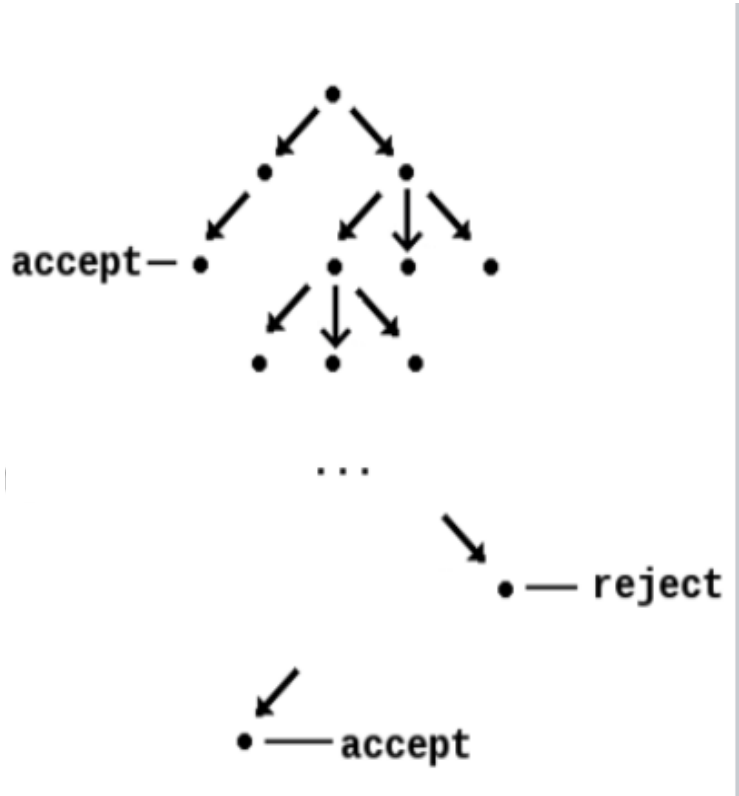


P (polynomial-time): problems that deterministic TM *solves* in polynomial time.

Nondeterministic Turing Machine



Non Deterministic Choice



NP (nondeterministic polynomial time): problems that nondeterministc TM *solves* in polynomial time.

Polynomial Time

Definition

Polynomial time (denoted by **P**) is the class of all (decision) problems that have an algorithm that solves it in polynomial time.

Polynomial Time

Definition

Polynomial time (denoted by **P**) is the class of all (decision) problems that have an algorithm that solves it in polynomial time.

Example

Problems in **P** include

1. Is there a shortest path from s to t of length $\leq k$ in G?
2. Is there a flow of value $\geq k$ in network G?
3. Is there an assignment to variables to satisfy given linear constraints?

Problems with no known polynomial time algorithms

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 is common to above problems?

Efficient Checkability

Above problems share the following feature:

Checkability

For any YES instance I_X of X there is a proof/certificate/solution that is of length poly($|I_X|$) such that given a proof one can efficiently check that I_X is indeed a YES instance.

Efficient Checkability

Above problems share the following feature:

Checkability

For any YES instance I_X of X there is a proof/certificate/solution that is of length poly($|I_X|$) such that given a proof one can efficiently check that I_X is indeed a YES instance.

Examples:

1. **SAT** formula φ: proof is a satisfying assignment.
2. **Independent Set** in graph G and k:

Efficient Checkability

Above problems share the following feature:

Checkability

For any YES instance I_X of X there is a proof/certificate/solution that is of length poly($|I_X|$) such that given a proof one can efficiently check that I_X is indeed a YES instance.

Examples:

1. **SAT** formula φ: proof is a satisfying assignment.
2. **Independent Set** in graph G and k: a subset S of vertices.

Certifiers

Definition

An algorithm $C(\cdot, \cdot)$ is a **certifier** for problem X if for every $I_x \in X$ there is some string t such that $C(I_x, t) =$ "yes", and conversely, if for some I_x and t, $C(I_x, t) =$ "yes" then $I_x \in X$.
The string t is called a certificate or proof for s.

Certifiers

Definition

An algorithm $C(\cdot, \cdot)$ is a **certifier** for problem X if for every $I_x \in X$ there is some string t such that $C(I_x, t) =$ "yes", and conversely, if for some I_x and t, $C(I_x, t) =$ "yes" then $I_x \in X$.
The string t is called a certificate or proof for s.

Definition (Efficient Certifier.)

A certifier C is an **efficient certifier** for problem X if there is a polynomial $p(\cdot)$ such that for every string s, we have that

- $I_x \in X$ if and only if
- there is a string t:
  1. $|t| \leq p(|I_x|)$,
  2. $C(I_x, t) =$ "yes",
  3. and C runs in polynomial time in $|I_x|$.

Example: Independent Set

1. Problem: Does $G = (V, E)$ 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 $\leq k$?

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?

Example: SAT

1. Problem: Does formula φ have a satisfying truth assignment?
   1. Certificate: Assignment $\boldsymbol{a}$ of $\mathbf{0/1}$ 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 s.
Question: Is the number s a composite?

Example: Composites

Problem: Composite

> **Instance:** A number s.
> **Question:** Is the number s a composite?

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

Not composite?

Problem: Not Composite

Instance: A number s.
Question: Is the number s not a composite?

The problem **Not Composite** is

- **(A)** Can be solved in linear time.
- **(B)** in **P**.
- **(C)** Can be solved in exponential time.
- **(D)** Does not have a certificate or an efficient certifier.
- **(E)** The status of this problem is still open.

Nondeterministic Polynomial Time

Alternate definition

Definition

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

Nondeterministic Polynomial Time

Alternate definition

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**.

"Guess" the certificate and verify $\Rightarrow$ nondeterministic TM.

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.

1. Certifier C on input I_x, t, runs $A(I_x)$ and returns the answer.
   - C runs in polynomial time.
   - If $I_x \in X$, then for every t, $C(I_x, t) =$ "yes".
   - If $I_x \notin X$, then for every t, $C(I_x, t) =$ "no". □

Exponential Time

Definition

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

Exponential Time

Definition

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

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 $|t| \leq p(|I_x|)$ run $C(I_x, t)$; answer "yes" if any one of these calls returns "yes".
2. The above algorithm correctly solves X (exercise).
3. Algorithm runs in $O(q(|I_x| + p(|I_x|))2^{p(|I_x|)})$, 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 $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$.

Is NP efficiently solvable?

We know $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$.

Big Question

Is there a problem in **NP** that does not belong to **P**? Or is $\mathbf{P} = \mathbf{NP}$?

If P = NP . . .

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

1. Many important optimization problems can be solved efficiently.

If P = NP . . .

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

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.

If $\mathbf{P = NP}$ . . .

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

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.
3. No security on the web.

If P = NP . . .

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

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.
3. No security on the web.
4. No e-commerce . . .

If P = NP . . .

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

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

If P = NP this implies that...

(A) **Vertex Cover** can be solved in polynomial time.

(B) **P = EXP**.

(C) **EXP $\subseteq$ P**.

(D) 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 $P \neq NP$.

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 IV

NP-Completeness and Cook-Levin Theorem

"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 Problems

Definition

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

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

NP-Complete Problems

Definition

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

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

Definition

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

1. $X \in \mathbf{NP}$, and
2. X is **NP-Hard**

NP-Complete Problems

Definition

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

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

Definition

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

1. $X \in \mathbf{NP}$, and
2. X is **NP-Hard**

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

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

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. Then Y can be solved in polynomial time. $Y \in \mathbf{P}$.

Solving NP-Complete Problems

Proposition

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

Proof.

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

1. Let $Y \in NP$. We know $Y \leq_P X$.
2. Then Y can be solved in polynomial time. $Y \in P$.
3. Thus, $Y \in NP \Rightarrow Y \in P$; $NP \subseteq P$.
4. Since $P \subseteq NP$, we have $P = NP$.

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. Then Y can be solved in polynomial time. $Y \in \mathbf{P}$.
3. Thus, $Y \in \mathbf{NP} \Rightarrow Y \in P$; $NP \subseteq P$.
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. □

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 $\mathbf{P} \neq \mathbf{NP}$,
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 $\mathbf{P} \neq \mathbf{NP}$,
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

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

Cook-Levin Theorem

Theorem

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

Using reductions one can prove that many other problems are **NP-Complete**

Proving that a problem X is NP-Complete

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

1. Show X is in **NP**.
   1. certificate/proof of polynomial size in input
   2. polynomial time certifier $C(s, t)$
2. Reduction from a known **NP-Complete** problem such as **3SAT** or **SAT** to X

Proving that a problem X is NP-Complete

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

1. Show X is in **NP**.
   1. certificate/proof of polynomial size in input
   2. polynomial time certifier $C(s, t)$
2. Reduction from a known **NP-Complete** problem such as **3SAT** or **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 X is in **NP**.
   1. certificate/proof of polynomial size in input
   2. polynomial time certifier $C(s, t)$
2. Reduction from a known **NP-Complete** problem such as **3SAT** or **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 $Y \leq_P X$.

NP-Completeness via Reductions

1. **SAT** is **NP-Complete**.
2. **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
3. **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
4. **Clique** is **NP-Complete**
5. **Vertex Cover** is **NP-Complete**
6. **Set Cover** is **NP-Complete**
7. **Hamilton Cycle** is **NP-Complete**
8. **3-Color** is **NP-Complete**

NP-Completeness via Reductions

1. **SAT** is **NP-Complete**.
2. **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
3. **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
4. **Clique** is **NP-Complete**
5. **Vertex Cover** is **NP-Complete**
6. **Set Cover** is **NP-Complete**
7. **Hamilton Cycle** is **NP-Complete**
8. **3-Color** is **NP-Complete**

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

A surprisingly frequent phenomenon!

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a 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 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.

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

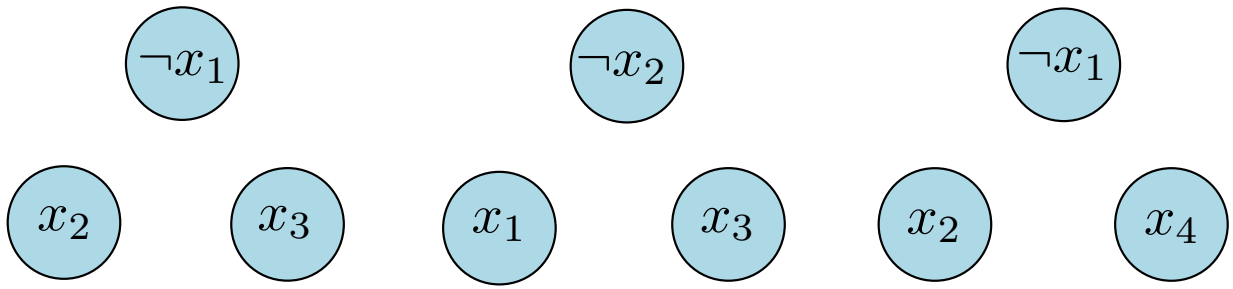


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

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

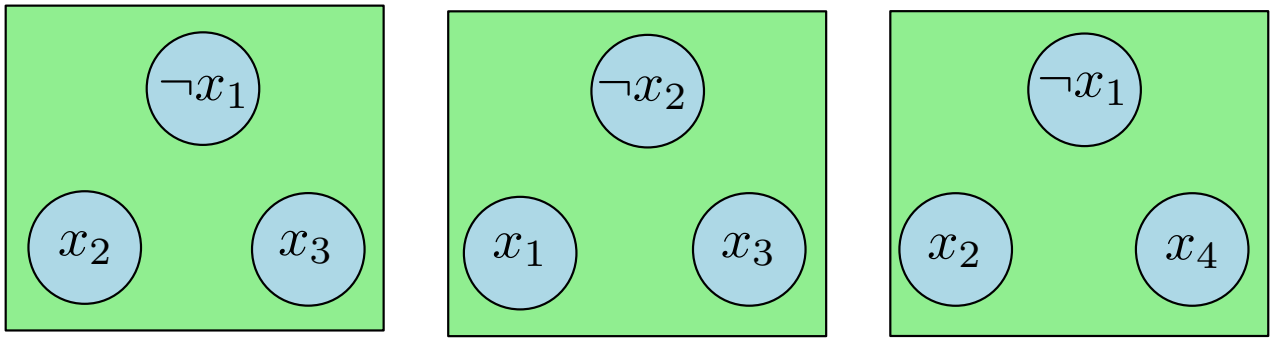


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

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

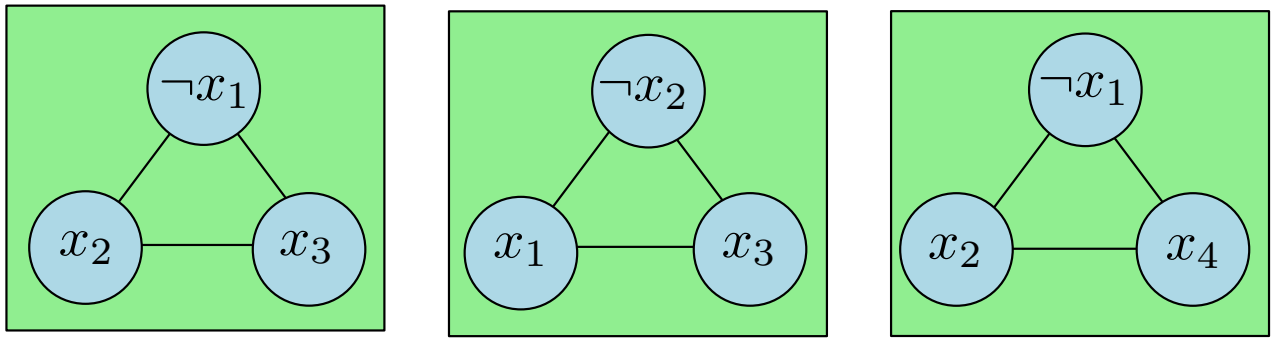


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

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

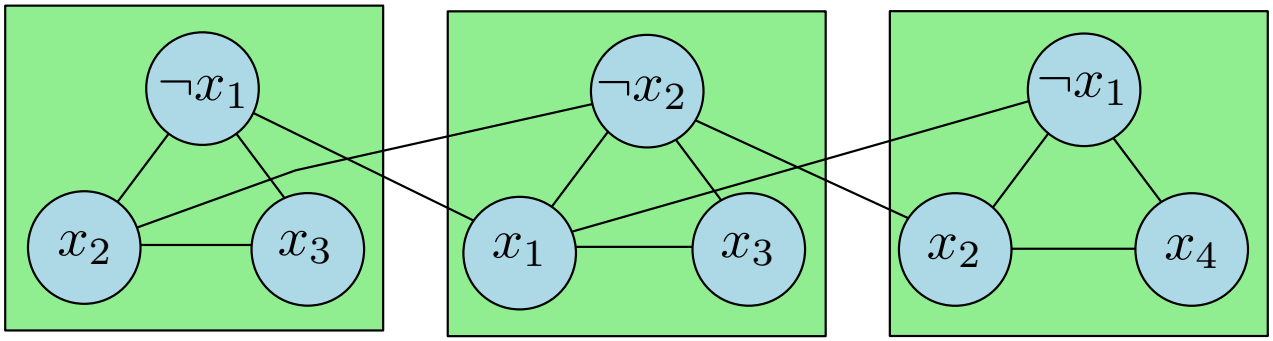


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

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

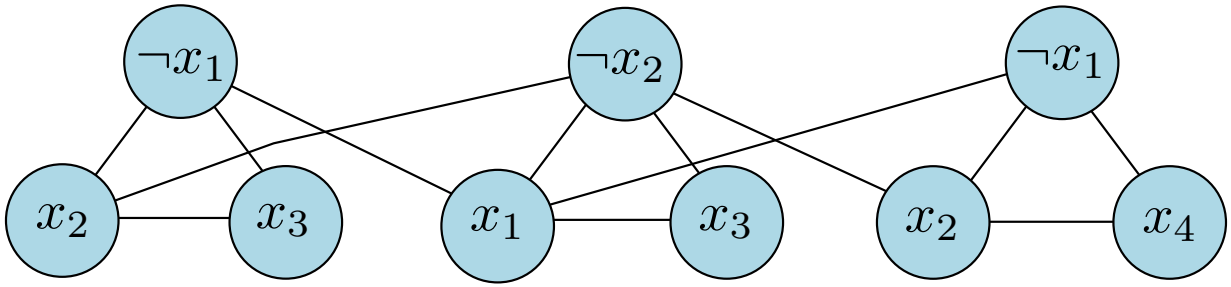


Figure: Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

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 $\square$

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 clauses
3. Thus, it is possible to obtain a truth assignment that makes in the literals in S true; such an assignment satisfies one literal in every clause $\square$