Polynomial Time Reductions

Lecture 22

April 18, 2017

Part I

(Polynomial Time) Reductions

Reductions

A reduction from Problem X to Problem Y means (informally) that if we have an algorithm for Problem Y, we can use it to find an algorithm for Problem X.

Reductions

A reduction from Problem X to Problem Y means (informally) that if we have an algorithm for Problem Y, we can use it to find an algorithm for Problem X.

Using Reductions

1. We use reductions to find algorithms to solve problems.

Reductions

A reduction from Problem X to Problem Y means (informally) that if we have an algorithm for Problem Y, we can use it to find an algorithm for Problem X.

Using Reductions

1. We use reductions to find algorithms to solve problems.
2. We also use reductions to show that we can't find algorithms for some problems. (We say that these problems are hard.)

Reductions for decision problems/languages

For languages L_X, L_Y, a **reduction from L_X to L_Y** is:

1. An algorithm ...
2. Input: $w \in \Sigma^*$
3. Output: $w' \in \Sigma^*$
4. Such that:

$$\boxed{w \in L_Y} \iff \boxed{w' \in L_X}$$

Reductions for decision problems/languages

For languages L_X, L_Y, a **reduction from L_X to L_Y** is:

1. An algorithm ...
2. Input: $w \in \Sigma^*$
3. Output: $w' \in \Sigma^*$
4. Such that:

$$\boxed{w \in L_Y} \iff \boxed{w' \in L_X}$$

(Actually, this is only one type of reduction, but this is the one we'll use most often.) There are other kinds of reductions.

Reductions for decision problems/languages

For decision problems X, Y, a **reduction from X to Y** is:

1. An algorithm ...
2. Input: I_X, an instance of X.
3. Output: I_Y an instance of Y.
4. Such that:
   $\boxed{I_Y \text{ is YES instance of } Y} \iff \boxed{I_X \text{ is YES instance of } X}$

Using reductions to solve problems

1. $\mathcal{R}$: Reduction $X \to Y$
2. $\mathcal{A}_Y$: algorithm for Y:

Using reductions to solve problems

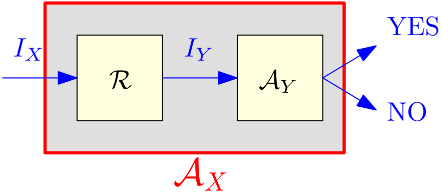
1. $\mathcal{R}$: Reduction $X \to Y$
2. $\mathcal{A}_Y$: algorithm for Y:
3. $\implies$ New algorithm for X:

```
A_X(I_X):
        // I_X: instance of X.
        I_Y ⇐ R(I_X)
        return A_Y(I_Y)
```

Using reductions to solve problems

1. $\mathcal{R}$: Reduction $X \to Y$
2. $\mathcal{A}_Y$: algorithm for Y:
3. $\implies$ New algorithm for X:

```
A_X(I_X):
        // I_X: instance of X.
        I_Y <= R(I_X)
        return A_Y(I_Y)
```



If $\mathcal{R}$ and $\mathcal{A}_Y$ polynomial-time $\implies$ $\mathcal{A}_X$ polynomial-time.

Comparing Problems

1. "Problem X is no harder to solve than Problem Y".
2. If Problem X reduces to Problem Y (we write $X \leq Y$), then X cannot be harder to solve than Y.
3. $X \leq Y$:
    1. X is no harder than Y, or
    2. Y is at least as hard as X.

Part II

Examples of Reductions

Independent Sets and Cliques

Given a graph G, a set of vertices V' is:

Independent Sets and Cliques

Given a graph G, a set of vertices V' is:

1. **independent set**: no two vertices of V' connected by an edge.

Independent Sets and Cliques

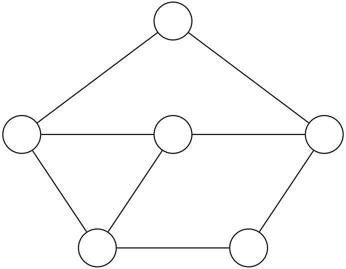
Given a graph G, a set of vertices V' is:

1. **independent set**: no two vertices of V' connected by an edge.
2. **clique**: *every* pair of vertices in V' is connected by an edge of G.

Independent Sets and Cliques

Given a graph G, a set of vertices V' is:

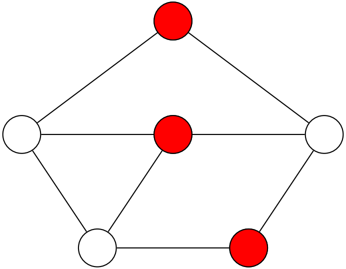
1. **independent set**: no two vertices of V' connected by an edge.
2. **clique**: *every* pair of vertices in V' is connected by an edge of G.



Independent Sets and Cliques

Given a graph G, a set of vertices V' is:

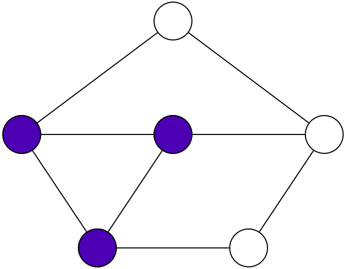
1. **independent set**: no two vertices of V' connected by an edge.
2. **clique**: *every* pair of vertices in V' is connected by an edge of G.



Independent Sets and Cliques

Given a graph G, a set of vertices V' is:

1. **independent set**: no two vertices of V' connected by an edge.
2. **clique**: *every* pair of vertices in V' is connected by an edge of G.



The Independent Set and Clique Problems

Problem: **Independent Set**

Instance: A graph G and an integer k.
Question: Does G has an independent set of size $\geq k$?

The **Independent Set** and **Clique** Problems

Problem: Independent Set

Instance: A graph G and an integer k.
Question: Does G has an independent set of size $\geq k$?

Problem: Clique

Instance: A graph G and an integer k.
Question: Does G has a clique of size $\geq k$?

Recall

For decision problems X, Y, a reduction from X to Y is:

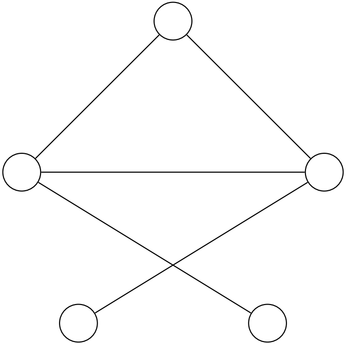
1. An algorithm . . .
2. that takes I_X, an instance of X as input . . .
3. and returns I_Y, an instance of Y as output . . .
4. such that the solution (YES/NO) to I_Y is the same as the solution to I_X.

Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph G and an integer k.

Reducing **Independent Set** to **Clique**

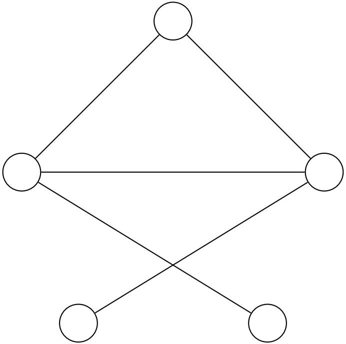
An instance of **Independent Set** is a graph G and an integer k.



Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph G and an integer k.

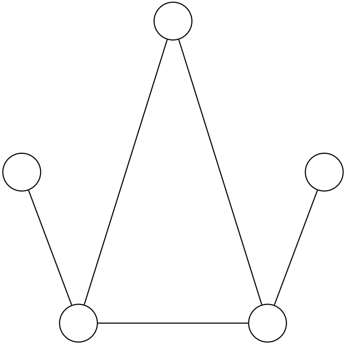
Reduction given $< G, k >$ outputs $< \overline{G}, k >$ where $\overline{G}$ is the *complement* of G. $\overline{G}$ has an edge (u, v) if and only if (u, v) is not an edge of G.



Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph G and an integer k.

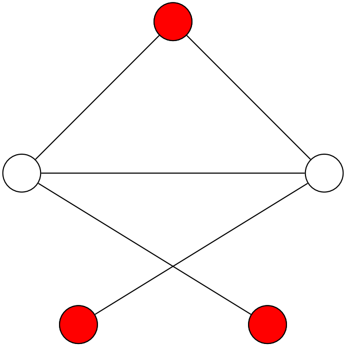
Reduction given $< G, k >$ outputs $< \overline{G}, k >$ where $\overline{G}$ is the *complement* of G. $\overline{G}$ has an edge (u, v) if and only if (u, v) is not an edge of G.



Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph G and an integer k.

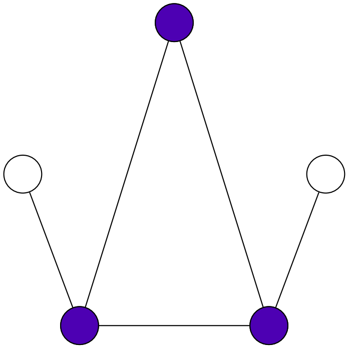
Reduction given $< G, k >$ outputs $< \overline{G}, k >$ where $\overline{G}$ is the *complement* of G. $\overline{G}$ has an edge (u, v) if and only if (u, v) is not an edge of G.



Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph G and an integer k.

Reduction given $< G, k >$ outputs $< \overline{G}, k >$ where $\overline{G}$ is the *complement* of G. $\overline{G}$ has an edge (u, v) if and only if (u, v) is not an edge of G.



Correctness of reduction

Lemma

G has an independent set of size k if and only if $\overline{G}$ has a clique of size k.

Proof.

Need to prove two facts:

G has independent set of size at least k implies that $\overline{G}$ has a clique of size at least k.

$\overline{G}$ has a clique of size at least k implies that G has an independent set of size at least k.

Easy to see both from the fact that $S \subseteq V$ is an independent set in G if and only if S is a clique in $\overline{G}$. □

Independent Set and Clique

1. Independent Set $\leq$ Clique.

Independent Set and **Clique**

1. **Independent Set** $\leq$ **Clique**.
   What does this mean?
2. If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.

Independent Set and **Clique**

1. **Independent Set** $\leq$ **Clique**.
   What does this mean?
2. If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.
3. **Clique** is *at least as hard as* **Independent Set**.

Independent Set and **Clique**

1. **Independent Set** $\leq$ **Clique**.
   What does this mean?
2. If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.
3. **Clique** is *at least as hard as* **Independent Set**.
4. Also... **Clique** $\leq$ **Independent Set**. Why? Thus **Clique** and **Independent Set** are polnomial-time equivalent.

Independent Set and Clique

Assume you can solve the **Clique** problem in $T(n)$ time. Then you can solve the **Independent Set** problem in

(A) $O(T(n))$ time.

(B) $O(n \log n + T(n))$ time.

(C) $O(n^2 T(n^2))$ time.

(D) $O(n^4 T(n^4))$ time.

(E) $O(n^2 + T(n^2))$ time.

(F) Does not matter - all these are polynomial if $T(n)$ is polynomial, which is good enough for our purposes.

DFA Universality

A DFA M is universal if it accepts every string.
That is, $L(M) = \Sigma^*$, the set of all strings.

DFA Universality

A DFA M is universal if it accepts every string.
That is, $L(M) = \Sigma^*$, the set of all strings.

Problem (DFA universality)

Input: *A DFA M.*
Goal: *Is M universal?*

DFA Universality

A DFA M is universal if it accepts every string.
That is, $L(M) = \Sigma^*$, the set of all strings.

Problem (DFA universality)

Input: *A DFA M.*
Goal: *Is M universal?*

How do we solve **DFA Universality**?

DFA Universality

A DFA M is universal if it accepts every string.
That is, $L(M) = \Sigma^*$, the set of all strings.

Problem (DFA universality)

Input: *A DFA M.*
Goal: *Is M universal?*

How do we solve **DFA Universality**?
We check if M has *any* reachable non-final state.

NFA Universality

An NFA N is said to be universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.

Problem (NFA universality)

Input: *A NFA M.*
Goal: *Is M universal?*

How do we solve **NFA Universality**?

NFA Universality

An NFA N is said to be universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.

Problem (NFA universality)

Input: *A NFA M.*
Goal: *Is M universal?*

How do we solve **NFA Universality**?
Reduce it to **DFA Universality**?

NFA Universality

An NFA N is said to be universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.

Problem (NFA universality)

Input: *A* NFA M.
Goal: *Is* M *universal?*

How do we solve **NFA Universality**?
Reduce it to **DFA Universality**?
Given an NFA N, convert it to an equivalent DFA M, and use the **DFA Universality** Algorithm.

NFA Universality

An NFA N is said to be universal if it accepts every string. That is, $L(N) = \Sigma^*$, the set of all strings.

Problem (NFA universality)

Input: *A* NFA M.
Goal: *Is* M *universal?*

How do we solve **NFA Universality**?
Reduce it to **DFA Universality**?
Given an NFA N, convert it to an equivalent DFA M, and use the **DFA Universality** Algorithm.
The reduction takes exponential time!
NFA Universality is known to be PSPACE-Complete and we do not expect a polynomial-time algorithm.

Polynomial-time reductions

We say that an algorithm is **efficient** if it runs in polynomial-time.

Polynomial-time reductions

We say that an algorithm is **efficient** if it runs in polynomial-time.

To find efficient algorithms for problems, we are only interested in polynomial-time reductions. Reductions that take longer are not useful.

Polynomial-time reductions

We say that an algorithm is **efficient** if it runs in polynomial-time.

To find efficient algorithms for problems, we are only interested in polynomial-time reductions. Reductions that take longer are not useful.

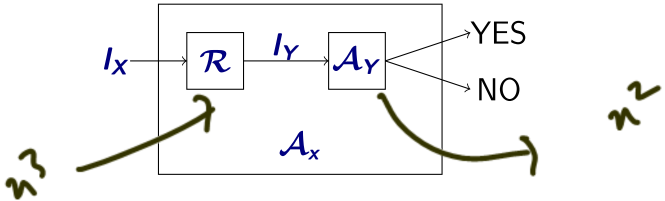
If we have a polynomial-time reduction from problem X to problem Y (we write $X \leq_P Y$), and a poly-time algorithm $\mathcal{A}_Y$ for Y, we have a polynomial-time/efficient algorithm for X.

Polynomial-time reductions

We say that an algorithm is **efficient** if it runs in polynomial-time.

To find efficient algorithms for problems, we are only interested in polynomial-time reductions. Reductions that take longer are not useful.

If we have a polynomial-time reduction from problem X to problem Y (we write $X \leq_P Y$), and a poly-time algorithm $\mathcal{A}_Y$ for Y, we have a polynomial-time/efficient algorithm for X.



Polynomial-time Reduction

A polynomial time reduction from a *decision* problem X to a *decision* problem Y is an *algorithm* $\mathcal{A}$ that has the following properties:

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

Proposition

If $X \leq_P Y$ then a polynomial time algorithm for Y implies a polynomial time algorithm for X.

Such a reduction is called a **Karp reduction**. Most reductions we will need are Karp reductions.Karp reductions are the same as mapping reductions when specialized to polynomial time for the reduction step.

Reductions again...

Let X and Y be two decision problems, such that X can be solved in polynomial time, and $X \leq_P Y$. Then

- **(A)** Y can be solved in polynomial time.
- **(B)** Y can NOT be solved in polynomial time.
- **(C)** If Y is hard then X is also hard.
- **(D)** None of the above.
- **(E)** All of the above.

Polynomial-time reductions and hardness

For decision problems X and Y, if $X \leq_P Y$, and Y has an efficient algorithm, X has an efficient algorithm.

Polynomial-time reductions and hardness

For decision problems X and Y, if $X \leq_P Y$, and Y has an efficient algorithm, X has an efficient algorithm.

If you believe that **Independent Set** does not have an efficient algorithm, why should you believe the same of **Clique**?

Polynomial-time reductions and hardness

For decision problems X and Y, if $X \leq_P Y$, and Y has an efficient algorithm, X has an efficient algorithm.

If you believe that **Independent Set** does not have an efficient algorithm, why should you believe the same of **Clique**?

Because we showed **Independent Set** $\leq_P$ **Clique**. If **Clique** had an efficient algorithm, so would **Independent Set**!

Polynomial-time reductions and hardness

For decision problems X and Y, if $X \leq_P Y$, and Y has an efficient algorithm, X has an efficient algorithm.

If you believe that **Independent Set** does not have an efficient algorithm, why should you believe the same of **Clique**?

Because we showed **Independent Set** $\leq_P$ **Clique**. If **Clique** had an efficient algorithm, so would **Independent Set**!

If $X \leq_P Y$ and X does not have an efficient algorithm, Y cannot have an efficient algorithm!

Polynomial-time reductions and instance sizes

Proposition

Let $\mathcal{R}$ be a polynomial-time reduction from X to Y. Then for any instance I_X of X, the size of the instance I_Y of Y produced from I_X by $\mathcal{R}$ is polynomial in the size of I_X.

Polynomial-time reductions and instance sizes

Proposition

Let $\mathcal{R}$ be a polynomial-time reduction from X to Y. Then for any instance I_X of X, the size of the instance I_Y of Y produced from I_X by $\mathcal{R}$ is polynomial in the size of I_X.

Proof.

$\mathcal{R}$ is a polynomial-time algorithm and hence on input I_X of size $|I_X|$ it runs in time $p(|I_X|)$ for some polynomial $p()$.
I_Y is the output of $\mathcal{R}$ on input I_X.
$\mathcal{R}$ can write at most $p(|I_X|)$ bits and hence $|I_Y| \leq p(|I_X|)$. □

Polynomial-time reductions and instance sizes

Proposition

Let $\mathcal{R}$ be a polynomial-time reduction from X to Y. Then for any instance I_X of X, the size of the instance I_Y of Y produced from I_X by $\mathcal{R}$ is polynomial in the size of I_X.

Proof.

$\mathcal{R}$ is a polynomial-time algorithm and hence on input I_X of size $|I_X|$ it runs in time $p(|I_X|)$ for some polynomial $p()$.
I_Y is the output of $\mathcal{R}$ on input I_X.
$\mathcal{R}$ can write at most $p(|I_X|)$ bits and hence $|I_Y| \leq p(|I_X|)$. □

Note: Converse is not true. A reduction need not be polynomial-time even if output of reduction is of size polynomial in its input.

Polynomial-time Reduction

A polynomial time reduction from a *decision* problem X to a *decision* problem Y is an *algorithm* $\mathcal{A}$ that has the following properties:

1. Given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y.
2. $\mathcal{A}$ runs in time polynomial in $|I_X|$. This implies that $|I_Y|$ (size of I_Y) is polynomial in $|I_X|$.
3. Answer to I_X YES *iff* answer to I_Y is YES.

Proposition

If $X \leq_P Y$ then a polynomial time algorithm for Y implies a polynomial time algorithm for X.

Transitivity of Reductions

Proposition

$X \leq_P Y$ and $Y \leq_P Z$ implies that $X \leq_P Z$.

Note: $X \leq_P Y$ does not imply that $Y \leq_P X$ and hence it is very important to know the FROM and TO in a reduction.

To prove $X \leq_P Y$ you need to show a reduction FROM X TO Y
That is, show that an algorithm for Y implies an algorithm for X.

Vertex Cover

Given a graph $G = (V, E)$, a set of vertices S is:

Vertex Cover

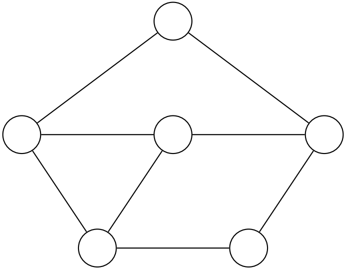
Given a graph $G = (V, E)$, a set of vertices S is:

1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.

Vertex Cover

Given a graph $G = (V, E)$, a set of vertices S is:

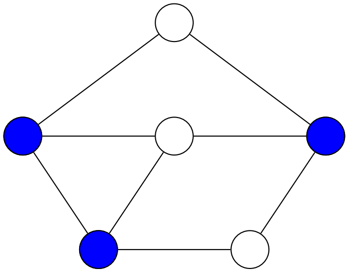
1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



Vertex Cover

Given a graph $G = (V, E)$, a set of vertices S is:

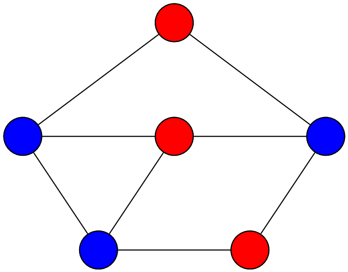
1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



Vertex Cover

Given a graph $G = (V, E)$, a set of vertices S is:

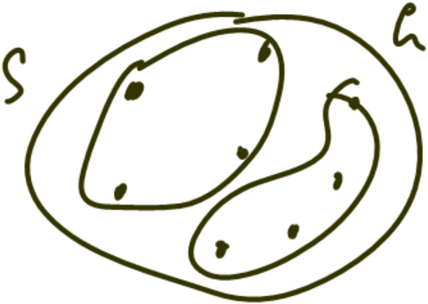
1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



The **Vertex Cover** Problem

Problem (**Vertex Cover**)

Input: *A graph G and integer k.*
Goal: *Is there a vertex cover of size $\leq k$ in G?*



The **Vertex Cover** Problem

Problem (**Vertex Cover**)

Input: *A graph G and integer k.*
Goal: *Is there a vertex cover of size $\leq k$ in G?*

Can we relate **Independent Set** and **Vertex Cover**?

Relationship between...

Vertex Cover and Independent Set

Proposition

Let $G = (V, E)$ be a graph. S is an independent set if and only if $V \setminus S$ is a vertex cover.

Proof.

($\Rightarrow$) Let S be an independent set

1. Consider any edge $uv \in E$.
2. Since S is an independent set, either $u \notin S$ or $v \notin S$.
3. Thus, either $u \in V \setminus S$ or $v \in V \setminus S$.
4. $V \setminus S$ is a vertex cover.

($\Leftarrow$) Let $V \setminus S$ be some vertex cover:

1. Consider $u, v \in S$
2. uv is not an edge of G, as otherwise $V \setminus S$ does not cover uv.
3. $\implies$ S is thus an independent set. □

Independent Set $\leq_P$ Vertex Cover

1. G: graph with n vertices, and an integer k be an instance of the **Independent Set** problem.

Independent Set $\leq_P$ Vertex Cover

1. G: graph with n vertices, and an integer k be an instance of the **Independent Set** problem.
2. G has an independent set of size $\geq k$ iff G has a vertex cover of size $\leq n - k$

Independent Set $\leq_P$ Vertex Cover

1. G: graph with n vertices, and an integer k be an instance of the **Independent Set** problem.
2. G has an independent set of size $\geq k$ iff G has a vertex cover of size $\leq n - k$
3. (G, k) is an instance of **Independent Set** , and $(G, n - k)$ is an instance of **Vertex Cover** with the same answer.

Independent Set $\leq_P$ Vertex Cover

1. G: graph with n vertices, and an integer k be an instance of the **Independent Set** problem.
2. G has an independent set of size $\geq k$ iff G has a vertex cover of size $\leq n - k$
3. (G, k) is an instance of **Independent Set** , and $(G, n - k)$ is an instance of **Vertex Cover** with the same answer.
4. Therefore, **Independent Set** $\leq_P$ **Vertex Cover**. Also **Vertex Cover** $\leq_P$ **Independent Set**.

Proving Correctness of Reductions

To prove that $X \leq_P Y$ you need to give an algorithm $\mathcal{A}$ that:

1. Transforms an instance I_X of X into an instance I_Y of Y.
2. Satisfies the property that answer to I_X is YES iff I_Y is YES.
    1. typical easy direction to prove: answer to I_Y is YES if answer to I_X is YES
    2. typical difficult direction to prove: answer to I_X is YES if answer to I_Y is YES (equivalently answer to I_X is NO if answer to I_Y is NO).
3. Runs in **polynomial** time.

Part III

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 $\neg x_i$.
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 $3\mathrm{CNF}$:
   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 $3\mathrm{CNF}$ 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. $(x_1 \lor x_2 \lor \neg x_4) \land (x_2 \lor \neg x_3) \land x_5$ is satisfiable; take $x_1, x_2, \ldots x_5$ to be all true
2. $(x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \land (x_1 \lor x_2)$ is not satisfiable.

3SAT

Given a $3\mathrm{CNF}$ formula φ, is there a truth assignment to variables such that φ evaluates to true?

(More on **2SAT** in a bit...)

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-Complete**ness.

$z = \overline{x}$

Given two bits x, z which of the following **SAT** formulas is equivalent to the formula $z = \overline{x}$:

(A) $(\overline{z} \vee x) \wedge (z \vee \overline{x})$.

(B) $(z \vee x) \wedge (\overline{z} \vee \overline{x})$.

(C) $(\overline{z} \vee x) \wedge (\overline{z} \vee \overline{x}) \wedge (\overline{z} \vee \overline{x})$.

(D) $z \oplus x$.

(E) $(z \vee x) \wedge (\overline{z} \vee \overline{x}) \wedge (z \vee \overline{x}) \wedge (\overline{z} \vee x)$.

$z = x \wedge y$

Given three bits x, y, z which of the following **SAT** formulas is equivalent to the formula $z = x \wedge y$:

(A) $(\overline{z} \vee x \vee y) \wedge (z \vee \overline{x} \vee \overline{y})$.

(B) $(\overline{z} \vee x \vee y) \wedge (\overline{z} \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee \overline{y})$.

(C) $(\overline{z} \vee x \vee y) \wedge (\overline{z} \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee \overline{y})$.

(D) $(z \vee x \vee y) \wedge (\overline{z} \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee \overline{y})$.

(E) $(z \vee x \vee y) \wedge (z \vee x \vee \overline{y}) \wedge (z \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee \overline{y}) \wedge (\overline{z} \vee x \vee y) \wedge (\overline{z} \vee x \vee \overline{y}) \wedge (\overline{z} \vee \overline{x} \vee y) \wedge (\overline{z} \vee \overline{x} \vee \overline{y})$.

$z = x \vee y$

Given three bits x, y, z which of the following **SAT** formulas is equivalent to the formula $z = x \vee y$:

(A) $(\overline{z} \vee x \vee y) \wedge (\overline{z} \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee \overline{y})$.

(B) $(\overline{z} \vee x \vee y) \wedge (\overline{z} \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee \overline{y})$.

(C) $(z \vee x \vee y) \wedge (\overline{z} \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee \overline{y})$.

(D) $(z \vee x \vee y) \wedge (z \vee x \vee \overline{y}) \wedge (z \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee \overline{y}) \wedge (\overline{z} \vee x \vee y) \wedge (\overline{z} \vee x \vee \overline{y}) \wedge (\overline{z} \vee \overline{x} \vee y) \wedge (\overline{z} \vee \overline{x} \vee \overline{y})$.

(E) $(\overline{z} \vee x \vee y) \wedge (z \vee \overline{x} \vee y) \wedge (z \vee x \vee \overline{y}) \wedge (z \vee \overline{x} \vee \overline{y})$.

$\text{SAT} \leq_P \text{3SAT}$

How SAT is different from 3SAT?

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

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

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

SAT $\leq_P$ 3SAT

How **SAT** is different from **3SAT**?

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

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

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

To reduce from an instance of **SAT** to an instance of **3SAT**, we must make all clauses to have exactly **3** variables...

Basic idea

1. Pad short clauses so they have **3** literals.
2. Break long clauses into shorter clauses.
3. Repeat the above till we have a $3\mathrm{CNF}$.

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

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 $|\varphi|$.

Idea: if a clause of φ is not of length $\mathbf{3}$, replace it with several clauses of length exactly $\mathbf{3}$.

SAT $\leq_P$ 3SAT

A clause with two literals

Reduction Ideas: clause with 2 literals

1. Case clause with 2 literals: Let $c = \ell_1 \vee \ell_2$. Let u be a new variable. Consider

$$c' = \left(\ell_1 \vee \ell_2 \vee u\right) \wedge \left(\ell_1 \vee \ell_2 \vee \neg u\right).$$

2. Suppose $\varphi = \psi \wedge c$. Then $\varphi' = \psi \wedge c'$ is satisfiable iff φ is satisfiable.

SAT $\leq_P$ 3SAT

A clause with a single literal

Reduction Ideas: clause with 1 literal

1. Case clause with one literal: Let c be a clause with a single literal (i.e., $c = \ell$). Let u, v be new variables. Consider

$$c' = \left(\ell \vee u \vee v\right) \wedge \left(\ell \vee u \vee \neg v\right) \wedge \left(\ell \vee \neg u \vee v\right) \wedge \left(\ell \vee \neg u \vee \neg v\right).$$

2. Suppose $\varphi = \psi \wedge c$. Then $\varphi' = \psi \wedge c'$ is satisfiable iff φ is satisfiable.

Reduction Ideas: clause with more than 3 literals

1. Case clause with five literals: Let $c = \ell_1 \lor \ell_2 \lor \ell_3 \lor \ell_4 \lor \ell_5$. Let u be a new variable. Consider

$$c' = \Big(\ell_1 \lor \ell_2 \lor \ell_3 \lor u\Big) \land \Big(\ell_4 \lor \ell_5 \lor \neg u\Big).$$

2. Suppose $\varphi = \psi \land c$. Then $\varphi' = \psi \land c'$ is satisfiable iff φ is satisfiable.

$l_1 \lor l_2 \lor l_3 \lor l_4 \lor l_5$ -

$(l_1 \lor l_2 \lor l_3 \lor u) \land (u \equiv l_4 \lor l_5)$

Reduction Ideas: clause with more than 3 literals

1. Case clause with $k > 3$ literals: Let $c = \ell_1 \lor \ell_2 \lor \ldots \lor \ell_k$. Let u be a new variable. Consider

$$c' = \Big(\ell_1 \lor \ell_2 \ldots \ell_{k-2} \lor u\Big) \land \Big(\ell_{k-1} \lor \ell_k \lor \neg u\Big).$$

2. Suppose $\varphi = \psi \land c$. Then $\varphi' = \psi \land c'$ is satisfiable iff φ is satisfiable.

Breaking a clause

Lemma

For any boolean formulas X *and* Y *and* z *a new boolean variable. Then*

$$X \vee Y \text{ is satisfiable}$$

if and only if, z *can be assigned a value such that*

$$\left(X \vee z\right) \wedge \left(Y \vee \neg z\right) \text{ is satisfiable}$$

(with the same assignment to the variables appearing in X *and* Y*).*

SAT $\leq_P$ 3SAT (contd)

Clauses with more than 3 literals

Let $c = \ell_1 \vee \cdots \vee \ell_k$. Let $u_1, \ldots u_{k-3}$ be new variables. Consider

$$c' = \left(\ell_1 \vee \ell_2 \vee u_1\right) \wedge \left(\ell_3 \vee \neg u_1 \vee u_2\right)$$
$$\wedge \left(\ell_4 \vee \neg u_2 \vee u_3\right) \wedge$$
$$\cdots \wedge \left(\ell_{k-2} \vee \neg u_{k-4} \vee u_{k-3}\right) \wedge \left(\ell_{k-1} \vee \ell_k \vee \neg u_{k-3}\right).$$

Claim

$\varphi = \psi \wedge c$ is satisfiable iff $\varphi' = \psi \wedge c'$ is satisfiable.

Another way to see it — reduce size of clause by one:

$$c' = \left(\ell_1 \vee \ell_2 \ldots \vee \ell_{k-2} \vee u_{k-3}\right) \wedge \left(\ell_{k-1} \vee \ell_k \vee \neg u_{k-3}\right).$$

An Example

Example

$$\varphi = \left(\neg x_1 \vee \neg x_4\right) \wedge \left(x_1 \vee \neg x_2 \vee \neg x_3\right) \wedge \left(\neg x_2 \vee \neg x_3 \vee x_4 \vee x_1\right) \wedge \left(x_1\right).$$

Equivalent form:

$$\psi = (\neg x_1 \vee \neg x_4 \vee z) \wedge (\neg x_1 \vee \neg x_4 \vee \neg z)$$

An Example

Example

$$\varphi = \left(\neg x_1 \vee \neg x_4\right) \wedge \left(x_1 \vee \neg x_2 \vee \neg x_3\right) \wedge \left(\neg x_2 \vee \neg x_3 \vee x_4 \vee x_1\right) \wedge \left(x_1\right).$$

Equivalent form:

$$\psi = (\neg x_1 \vee \neg x_4 \vee z) \wedge (\neg x_1 \vee \neg x_4 \vee \neg z) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$$

An Example

Example

$$\varphi = \Big(\neg x_1 \vee \neg x_4\Big) \wedge \Big(x_1 \vee \neg x_2 \vee \neg x_3\Big) \wedge \Big(\neg x_2 \vee \neg x_3 \vee x_4 \vee x_1\Big) \wedge \Big(x_1\Big).$$

Equivalent form:

$$\psi = (\neg x_1 \vee \neg x_4 \vee z) \wedge (\neg x_1 \vee \neg x_4 \vee \neg z) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3 \vee y_1) \wedge (x_4 \vee x_1 \vee \neg y_1)$$

An Example

Example

$$\varphi = \left(\neg x_1 \lor \neg x_4\right) \land \left(x_1 \lor \neg x_2 \lor \neg x_3\right)$$
$$\land \left(\neg x_2 \lor \neg x_3 \lor x_4 \lor x_1\right) \land \left(x_1\right).$$

Equivalent form:

$$\psi = (\neg x_1 \lor \neg x_4 \lor z) \land (\neg x_1 \lor \neg x_4 \lor \neg z)$$
$$\land (x_1 \lor \neg x_2 \lor \neg x_3)$$
$$\land (\neg x_2 \lor \neg x_3 \lor y_1) \land (x_4 \lor x_1 \lor \neg y_1)$$
$$\land (x_1 \lor u \lor v) \land (x_1 \lor u \lor \neg v)$$
$$\land (x_1 \lor \neg u \lor v) \land (x_1 \lor \neg u \lor \neg v).$$

Overall Reduction Algorithm

Reduction from **SAT** to **3SAT**

```
ReduceSATTo3SAT(φ):
    // φ: CNF formula.
    for each clause c of φ do
        if c does not have exactly 3 literals then
            construct c′ as before
        else
            c′ = c
    ψ is conjunction of all c′ constructed in loop
    return Solver3SAT(ψ)
```

Correctness (informal)

φ is satisfiable iff ψ is satisfiable because for each clause c, the new $3\mathrm{CNF}$ formula c' is logically equivalent to c.

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 $2\mathrm{CNF}$ 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...
(Hint: Create a graph with two vertices for each variable (for a variable x there would be two vertices with labels $x = \mathbf{0}$ and $x = \mathbf{1}$). For ever $\mathbf{2}\mathrm{CNF}$ clause add two directed edges in the graph. The edges are implication edges: They state that if you decide to assign a certain value to a variable, then you must assign a certain value to some other variable.
Now compute the strong connected components in this graph, and continue from there...)