Intractability and Reductions

Lecture 22
April 18, 2023

Course Outline

- Part I: models of computation (reg exps, DFA/NFA, CFGs, TMs)
- Part II: (efficient) algorithm design
- Part III: intractability via reductions
  - Undecidablity: problems that have no algorithms
  - NP-Completeness: problems unlikely to have efficient algorithms unless $\boldsymbol{P = NP}$

Part I

Intractability and Lower Bounds

Turing Machines and Church-Turing Thesis

Turing defined TMs as a machine model of computation

Church-Turing thesis: any function that is computable can be computed by TMs

Efficient Church-Turing thesis: any function that is computable can be computed by TMs with only a polynomial slow-down

Computability and Complexity Theory

- What functions can and *cannot* be computed by TMs?
- What functions/problems can and cannot be solved *efficiently*?

Why?

- Foundational questions about computation
- Pragmatic: Can we solve our problem or not?
- Are we not being clever enough to find an efficient algorithm or should we stop because there isn't one or likely to be one?

Lower Bounds and Impossibility Results

Prove that given problem cannot be solved (efficiently) on a TM. Informally we say that the problem is "hard".

Generally quite difficult: algorithms can be very non-trivial and clever.

Reductions to Prove Intractability

A general methodology to prove impossibility results.

- Start with some *known* hard problem X
- *Reduce* X to your favorite problem Y

If Y can be solved then so can $X \Rightarrow Y$ is also *hard*

Reductions to Prove Intractability

A general methodology to prove impossibility results.

- Start with some *known* hard problem X
- *Reduce* X to your favorite problem Y

If Y can be solved then so can $X \Rightarrow Y$ is also *hard*

Caveat: In algorithms we reduce new problem to known solved one!

Reductions to Prove Intractability

A general methodology to prove impossibility results.

- Start with some *known* hard problem X
- *Reduce* X to your favorite problem Y

If Y can be solved then so can X $\Rightarrow$ Y is also *hard*

Caveat: In algorithms we reduce new problem to known solved one!

Who gives us the initial hard problem?

- Some clever person (Cantor/Gödel/Turing/Cook/Levin ...) who establish hardness of a fundamental problem
- Assume some core problem is hard because we haven't been able to solve it for a long time. This leads to *conditional* results

Reductions to Prove Intractability

A general methodology to prove impossibility results.

- Start with some *known* hard problem X
- *Reduce* X to your favorite problem Y

If Y can be solved then so can $X \Rightarrow Y$ is also *hard*

Caveat: In algorithms we reduce new problem to known solved one!

Who gives us the initial hard problem?

- Some clever person (Cantor/Gödel/Turing/Cook/Levin ...) who establish hardness of a fundamental problem
- Assume some core problem is hard because we haven't been able to solve it for a long time. This leads to *conditional* results

Reduction is a powerful and unifying tool in Computer Science

Decision Problems, Languages, Terminology

When proving hardness we limit attention to *decision* problems

- A decision problem Π is a collection of instances (strings)
- For each instance I of Π, answer is YES or NO
- Equivalently: boolean function $f_\Pi : \Sigma^* \rightarrow \{0, 1\}$ where $f(I) = 1$ if I is a YES instance, $f(I) = 0$ if NO instance
- Equivalently: language $L_\Pi = \{I \mid I \text{ is a YES instance}\}$

Decision Problems, Languages, Terminology

When proving hardness we limit attention to *decision* problems

- A decision problem Π is a collection of instances (strings)
- For each instance I of Π, answer is YES or NO
- Equivalently: boolean function $f_\Pi : \Sigma^* \to \{0, 1\}$ where $f(I) = 1$ if I is a YES instance, $f(I) = 0$ if NO instance
- Equivalently: language $L_\Pi = \{I \mid I \text{ is a YES instance}\}$

Notation about encoding: distinguish I from encoding $\langle I \rangle$

- n is an integer. $\langle n \rangle$ is the encoding of n in some format (could be unary, binary, decimal etc)
- G is a graph. $\langle G \rangle$ is the encoding of G in some format
- M is a TM. $\langle M \rangle$ is the encoding of TM as a string according to some fixed convention

Examples

- Given directed graph G, is it strongly connected? $\langle G \rangle$ is a YES instance if it is, otherwise NO instance
- Given number n, is it a prime number? $L_{PRIMES} = \{\langle n \rangle \mid n \text{ is prime}\}$
- Given number n is it a composite number? $L_{COMPOSITE} = \{\langle n \rangle \mid n \text{ is a composite}\}$
- Given $G = (V, E), s, t, B$ is the shortest path distance from s to t at most B? Instance is $\langle G, s, t, B \rangle$

Part II

(Polynomial Time) Reductions

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 will use for hardness.) There are other kinds of reductions.

Reductions for decision problems/languages

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

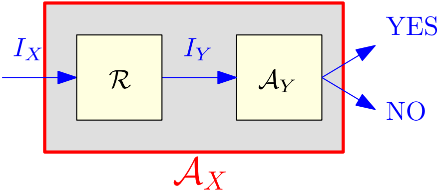
1. An algorithm ...
2. Input: I_X, an instance of X.
3. Output: I_Y an instance of Y.
4. Such that:

$$\boxed{I_Y \text{ is YES instance of } Y} \iff \boxed{I_X \text{ is YES instance of } X}$$

Reductions

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

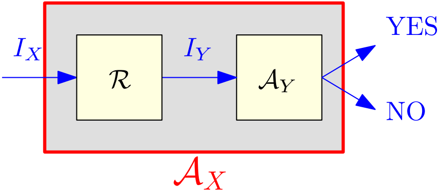
```
$\mathcal{A}_X(I_X)$:
        // $I_X$: instance of $X$.
        $I_Y \Leftarrow \mathcal{R}(I_X)$
        return $\mathcal{A}_Y(I_Y)$
```



Reductions

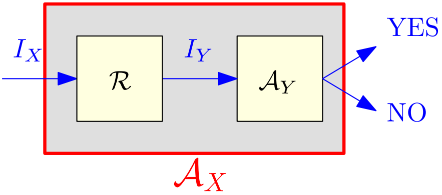
1. $\mathcal{R}$: Reduction $X \to Y$
2. $\mathcal{A}_Y$: algorithm for Y:
3. $\implies$ New algorithm for X:

```
$\mathcal{A}_X(I_X)$:
        // $I_X$: instance of $X$.
        $I_Y \Leftarrow \mathcal{R}(I_X)$
        return $\mathcal{A}_Y(I_Y)$
```



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

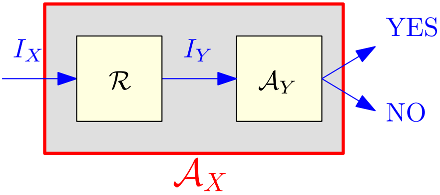
Reductions and running time



$R(n)$: running time of $\mathcal{R}$
$Q(n)$: running time of $\mathcal{A}_Y$

Question: What is running time of $\mathcal{A}_X$?

Reductions and running time



$R(n)$: running time of $\mathcal{R}$
$Q(n)$: running time of $\mathcal{A}_Y$

Question: What is running time of $\mathcal{A}_X$? $O(R(n) + Q(R(n)))$. Why?

- If I_X has size n, $\mathcal{R}$ creates an instance I_Y of size at most $R(n)$
- $\mathcal{A}_{\mathcal{Y}}$'s time on I_Y is by definition at most $Q(|I_Y|) \leq Q(R(n))$.

Example: If $R(n) = n^2$ and $Q(n) = n^{1.5}$ then $\mathcal{A}_X$ is $O(n^3)$

Notation and Implication of Reductions

1. If Problem X reduces to Problem Y we write $X \le Y$
2. If Problem X reduces to Problem Y where reduction $\mathcal{R}$ is an efficient (polynomial-time algorithm) we write $X \le_P Y$.

Notation and Implication of Reductions

1. If Problem X reduces to Problem Y we write $X \leq Y$
2. If Problem X reduces to Problem Y where reduction $\mathcal{R}$ is an efficient (polynomial-time algorithm) we write $X \leq_P Y$.

Algorithmic implication:

Lemma

- *If $X \leq Y$ and Y has an algorithm then X has an algorithm.*
- *If $X \leq_P Y$ and Y has a polynomial-time algorithm then X has a polynomial-time algorithm.*

Hardness Implications of Reductions

1. Problem X reduces to Problem Y: $X \leq Y$
2. Problem X efficiently reduces to Problem Y: $X \leq_P Y$.

Hardness implication:

Lemma

- *If* $X \leq Y$ **and** X *does not have an algorithm then* Y *does not have an algorithm.*
- *If* $X \leq_P Y$ **and** X *does not have a polynomial-time algorithm then* Y *does not have a polynomial-time algorithm.*

Hardness Implications of Reductions

1. Problem X reduces to Problem Y: $X \leq Y$
2. Problem X efficiently reduces to Problem Y: $X \leq_P Y$.

Hardness implication:

Lemma

- *If* $X \leq Y$ **and** X *does not have an algorithm then* Y *does not have an algorithm.*
- *If* $X \leq_P Y$ **and** X *does not have a polynomial-time algorithm then* Y *does not have a polynomial-time algorithm.*

Proof.

Suppose Y has an algorithm. Then X does too since $X \leq Y$. But contradicts assumption that X does not have an algorithm. Similarly for efficient reduction. □

Transitivity of Reductions

Proposition

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

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

Proving Correctness of Reductions

To prove that $X \leq Y$ 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. To prove $X \leq_P Y$, additionally show that $\mathcal{A}$ runs in **polynomial** time.

Remember, remember, remember

- **Algorithm design:** reduce new problem X to *known easy* problem Y
- **Hardness:** reduce *known hard* problem X to new problem Y

Tools to remember:

- Am I trying to design algorithm or prove hardness?
- What do I know about some standard problems? Easy or hard?

Part III

Examples of Reductions

Undecidability Reductions

Theorem (Turing)

Following languages are undecidable.

- $L_{HALT} = \{\langle M \rangle \mid M \text{ halts on blank input}\}$
- $L_{HALT,w} = \{\langle M, w \rangle \mid M \text{ halts on input } w\}$
- $L_u = \{\langle M, w \rangle \mid M \text{ accepts } w\}$

Used reduction from Halting to show several probles are also undecidable.

CS 125 assignment

Write a program that prints “Hello World”

```
main() {
    print(‘‘Hello World’’)
}
```

CS 125 assignment

Write a program that prints “Hello World”

```
main() {
    print(‘‘Hello World’’)
}
```

Question: Can we create an autograder?

CS 125 assignment

Write a program that prints “Hello World”

```
main() {
    print(‘‘Hello World’’)
}
```

Question: Can we create an autograder? No! Why?

Reducing Halting to Autograder

- **Halting problem:** given *arbitrary* program foo**()**, does it halt?
- **Reduction to CS125Autograder**: given foo**()** output foobar**()**

```
main() {
    foo()
    print(‘‘Hello World’’)
}
foo() {
    line 1
    line 2
    ...
}
```

Note: Reduction only needs to add a few lines of code to foo**()**

Reducing Halting to Autograder

- **Halting problem:** given *arbitrary* program foo**()**, does it halt?
- **Reduction to CS125Autograder**: given foo**()** output foobar**()**

```
main() {
    foo()
    print(‘‘Hello World’’)
}
foo() {
    line 1
    line 2
    ...
}
```

Note: Reduction only needs to add a few lines of code to foo**()**

- foobar**()** prints “Hello World” **if and only if** foo**()** halts!
- If we had CS125Autograder then we can solve Halting. But Halting is hard according to Turing. Hence ...

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

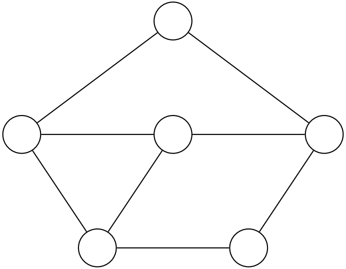
Given a graph G, a set of vertices V' is:

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:

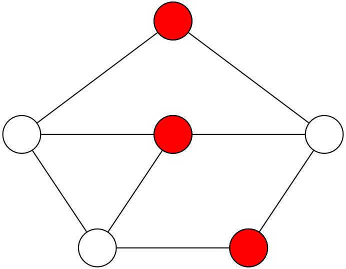
1. **independent set**: no two vertices of V' connected by an edge.
2. **clique**: *every* pair of vertices in V' is connected by an edge of G.



Independent Sets and Cliques

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

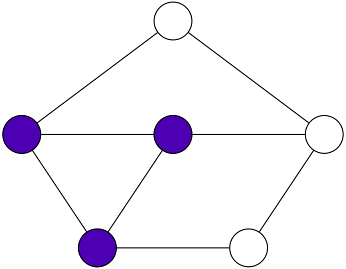
1. **independent set**: no two vertices of V' connected by an edge.
2. **clique**: *every* pair of vertices in V' is connected by an edge of G.



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

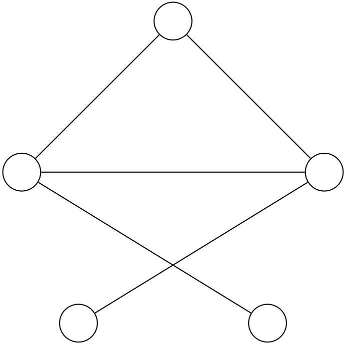
Instance: A graph G and an integer k.
Question: Does G has a clique of size $\geq k$?

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

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

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

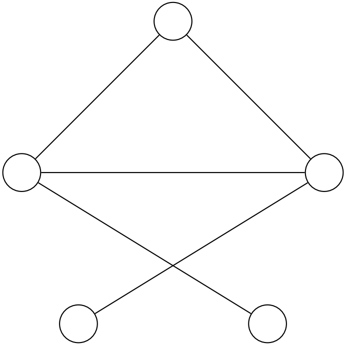
An instance of **Independent Set** is a graph G and an integer k.



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

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

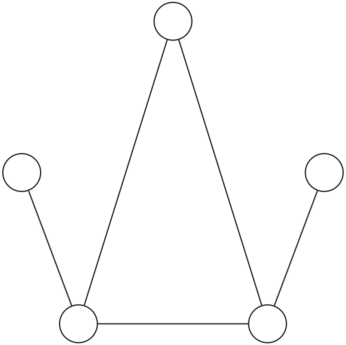
Reduction given $\langle G, k \rangle$ outputs $\langle \overline{G}, k \rangle$ where $\overline{G}$ is the *complement* of G. $\overline{G}$ has an edge (u, v) if and only if (u, v) is not an edge of G.



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

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

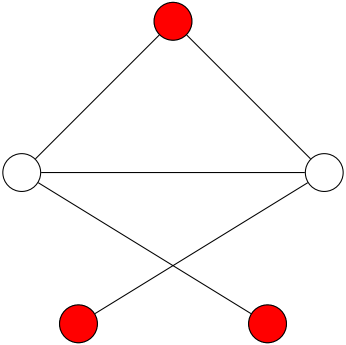
Reduction given $\langle G, k\rangle$ outputs $\langle \overline{G}, k\rangle$ where $\overline{G}$ is the *complement* of G. $\overline{G}$ has an edge (u, v) if and only if (u, v) is not an edge of G.



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

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

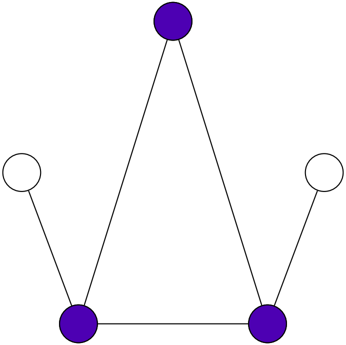
Reduction given $\langle G, k\rangle$ outputs $\langle \overline{G}, k\rangle$ where $\overline{G}$ is the *complement* of G. $\overline{G}$ has an edge (u, v) if and only if (u, v) is not an edge of G.



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

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

Reduction given $\langle G, k \rangle$ outputs $\langle \overline{G}, k \rangle$ 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

Independent Set $\leq_P$ **Clique**. What does this mean?

Independent Set and **Clique**

Independent Set $\leq_P$ **Clique**. What does this mean?

1. If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.
2. The reduction is efficient. Hence, if we have a poly-time algorithm for **Clique**, then we have a poly-time algorithm for **Independent Set**.
3. **Clique** is *at least as hard as* **Independent Set**.

Independent Set and **Clique**

Independent Set $\le_P$ **Clique**. What does this mean?

1. If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.
2. The reduction is efficient. Hence, if we have a poly-time algorithm for **Clique**, then we have a poly-time algorithm for **Independent Set**.
3. **Clique** is *at least as hard as* **Independent Set**.

Also... **Clique** $\le_P$ **Independent Set**. Why?

Independent Set and Clique

Independent Set $\leq_P$ **Clique**. What does this mean?

1. If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.
2. The reduction is efficient. Hence, if we have a poly-time algorithm for **Clique**, then we have a poly-time algorithm for **Independent Set**.
3. **Clique** is *at least as hard as* **Independent Set**.

Also... **Clique** $\leq_P$ **Independent Set**. Why?
Caveat: in general $X \leq Y$ does not mean that $Y \leq X$.

Vertex Cover

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

Vertex Cover

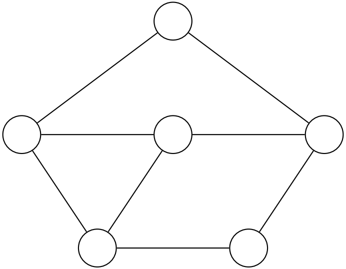
Given a graph $G = (V, E)$, a set of vertices S is:

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:

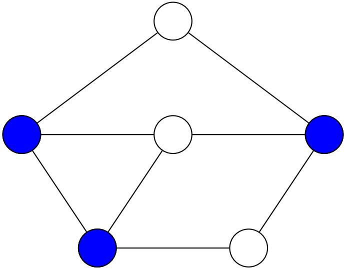
1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



Vertex Cover

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

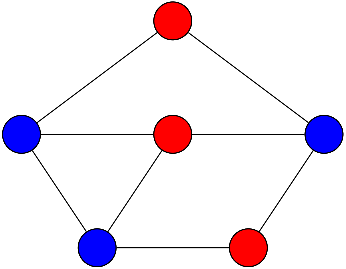
1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



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 $u \not\in S$ or $v \not\in S$.
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.
2. Reduction: given (G, k), an instance of **Independent Set** , ouput $(G, n - k)$ as an instance of **Vertex Cover**.
3. G has an independent set of size $\geq k$ iff G has a vertex cover of size $\leq n - k$ which proves correctness.
4. Easy to see reduction is efficient.
5. Therefore, **Independent Set** $\leq_P$ **Vertex Cover**. Also **Vertex Cover** $\leq_P$ **Independent Set**.

Part IV

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. $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ 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.

3SAT

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

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.

SAT $\leq_P$ 3SAT

How SAT is different from 3SAT?

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

$$\left(x \lor y \lor z \lor w \lor u\right) \land \left(\neg x \lor \neg y \lor \neg z \lor w \lor u\right) \land \left(\neg x\right)$$

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

Suppose $\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (x_3 \vee \neg x_5)$

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

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

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

Suppose $\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (x_3)$

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

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

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

SAT $\leq_P$ 3SAT

A clause with more than 3 literals

Suppose $\varphi = (x_1 \lor x_2 \lor \neg x_3) \land (x_3 \lor \neg x_5 \lor x_6 \lor \neg x_7 \lor x_8)$

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.

SAT $\leq_P$ 3SAT

A clause with more than 3 literals

Reduction Ideas: clause with more than 3 literals

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

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

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

Breaking a clause

Lemma

Let X and Y be boolean formulas in some variables and z a new boolean variable. Then

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

if and only if

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

Proof.

Exercise. ☐

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

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

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

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

Equivalent form:

$$\psi = (\neg x_1 \lor \neg x_4 \lor z) \land (\neg x_1 \lor \neg x_4 \lor \neg z)$$

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

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

Equivalent form:

$$\begin{aligned}\psi = {} & (\neg x_1 \vee \neg x_4 \vee z) \wedge (\neg x_1 \vee \neg x_4 \vee \neg z) \\ & \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \\ & \wedge (\neg x_2 \vee \neg x_3 \vee y_1) \wedge (x_4 \vee x_1 \vee \neg y_1)\end{aligned}$$

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:

$$\begin{aligned}\psi = &(\neg x_1 \lor \neg x_4 \lor z) \land (\neg x_1 \lor \neg x_4 \lor \neg z)\\ &\land (x_1 \lor \neg x_2 \lor \neg x_3)\\ &\land (\neg x_2 \lor \neg x_3 \lor y_1) \land (x_4 \lor x_1 \lor \neg y_1)\\ &\land (x_1 \lor u \lor v) \land (x_1 \lor u \lor \neg v)\\ &\land (x_1 \lor \neg u \lor v) \land (x_1 \lor \neg u \lor \neg v).\end{aligned}$$

Overall Reduction Algorithm

Reduction from SAT to 3SAT

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

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.

Algorithm for 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...)