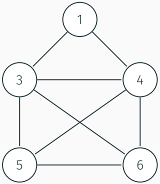
Pre-lecture brain teaser

Consider the following algorithm which takes in a undirected graph (G) and a vertex s

```
FindClique (G, s)
    C = s
    for each vertex v ∈ V
        flag = 1
        for each vertex u ∈ C
            if (u, v) ∉ E
                flag = 0
        if flag == 1
            C = C ∪ {v}
    return C
```

The algorithm is a represents a greedy algorithm which finds a clique depending on a start vertex s.

- How fast is this algorithm?



ECE-374-B: Lecture 21 - P/NP and NP-completeness

Instructor: Nickvash Kani

April 11, 2023

University of Illinois at Urbana-Champaign

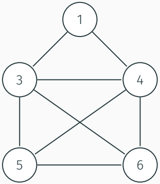
Pre-lecture brain teaser

Consider the following algorithm which takes in a undirected graph (G) and a vertex s

```
FindClique (G, s)
    C = s
    for each vertex v ∈ V
        flag = 1
        for each vertex u ∈ C
            if (u, v) ∉ E
                flag = 0
        if flag == 1
            C = C ∪ {v}
    return C
```

The algorithm is a represents a greedy algorithm which finds a clique depending on a start vertex s.

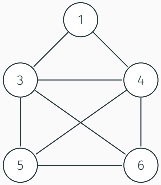
- How fast is this algorithm?



Pre-lecture brain teaser

Consider the following algorithm which takes in a undirected graph (G) and a vertex s

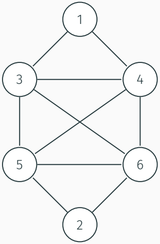
```
FindClique (G, s)
    C = s
    for each vertex v ∈ V
        flag = 1
        for each vertex u ∈ C
            if (u, v) ∉ E
                flag = 0
        if flag == 1
            C = C ∪ {v}
    return C
```



The Clique-problem is NP-complete. But this algorithm provides us with the maximal clique containing s. If we run it $|V|$ times, does that solve the clique-problem.

Consider the following algorithm which takes in a undirected graph (G) and a vertex s

```
FindClique (G, s)
    C = s
    for each vertex v ∈ V
        flag = 1
        for each vertex u ∈ C
            if (u, v) ∉ E
                flag = 0
        if flag == 1
            C = C ∪ {v}
    return C
```



The Satisfiability Problem (SAT)

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

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

Propositional Formulas

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

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

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

CNF is universal

Every boolean formula $f : \{0,1\}^n \to \{0,1\}$ can be written as a CNF formula.

x_1	x_2	x_3	x_4	x_5	x_6	$f(x_1, x_2, \ldots, x_6)$	$\overline{x_1} \vee x_2\overline{x_3} \vee x_4 \vee \overline{x_5} \vee x_6$
0	0	0	0	0	0	$f(0, \ldots, 0, 0)$	1
0	0	0	0	0	1	$f(0, \ldots, 0, 1)$	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	0	1	0	0	1	?	1
1	0	1	0	1	0	0	0
1	0	1	0	1	1	?	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
1	1	1	1	1	1	$f(1, \ldots, 1)$	1

For every row that f is zero compute corresponding CNF clause.

Take the and ($\bigwedge$) of all the CNF clauses computed

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 3CNF 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

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

3SAT
Given a 3CNF formula φ, is there a truth assignment to variables such that φ evaluates to true?

Importance of SAT and 3SAT

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

$z = \overline{x}$

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

$z = \overline{x}$: Solution

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

x	y	$z = \overline{x}$
0	0	0
0	1	1
1	0	1
1	1	0

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

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

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

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

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

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

$z = x \wedge y$

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

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

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

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

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

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

x	y	z	$z = x \wedge y$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Reducing SAT to 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.

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

- Pad short clauses so they have 3 literals.
- Break long clauses into shorter clauses.
- Repeat the above till we have a 3CNF.

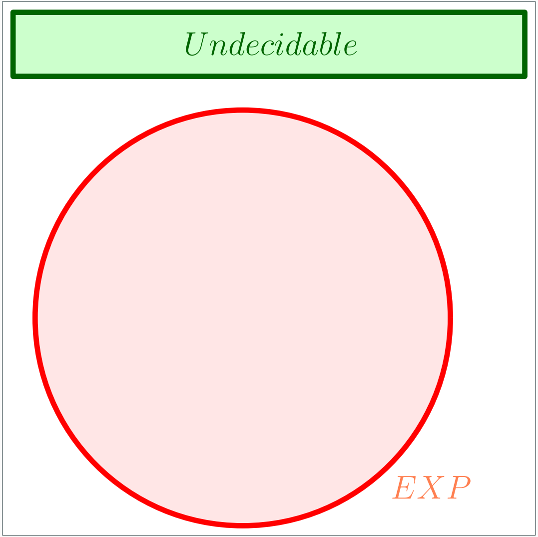
Proof of this in Prof. Har-Peled's async lectures!

Overview of Complexity Classes

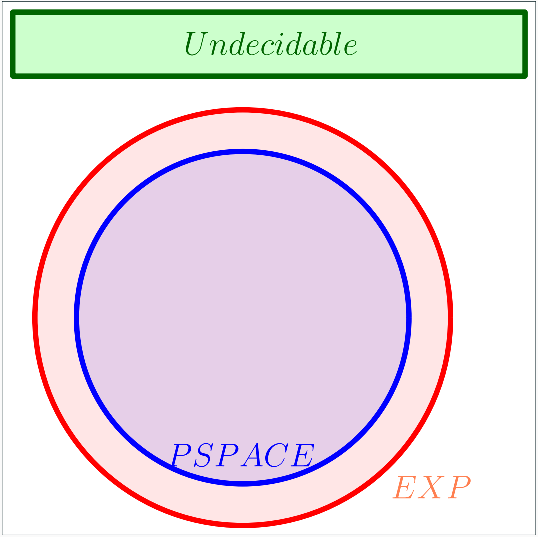
In the beginning...

In the beginning...

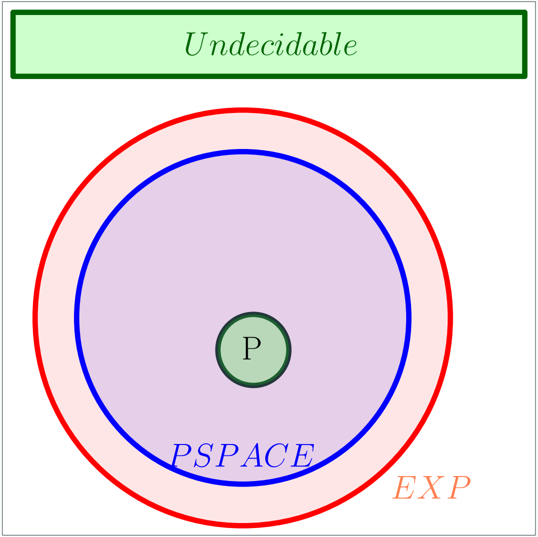
In the beginning...



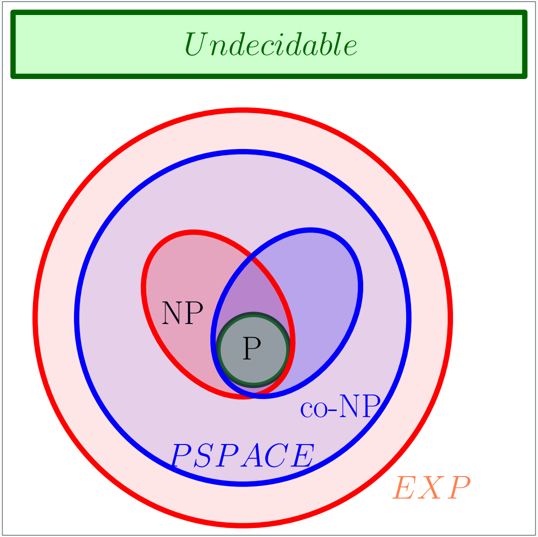
In the beginning...



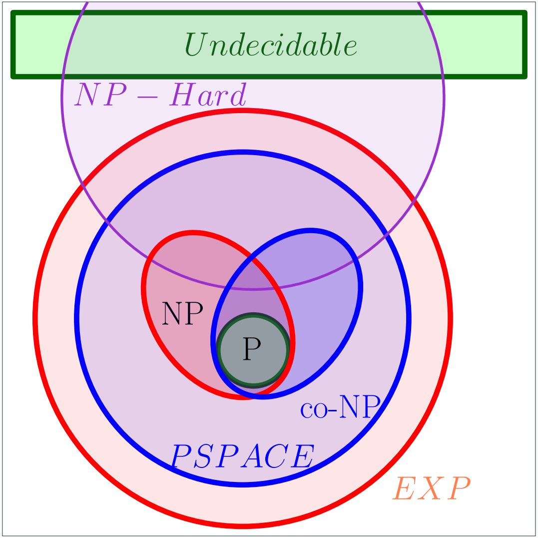
In the beginning...



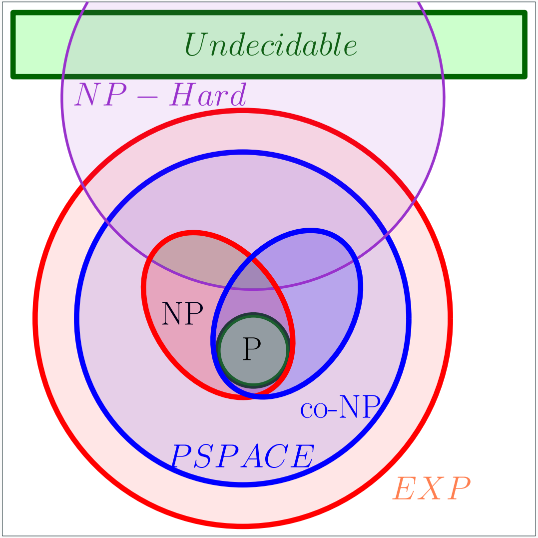
In the beginning...



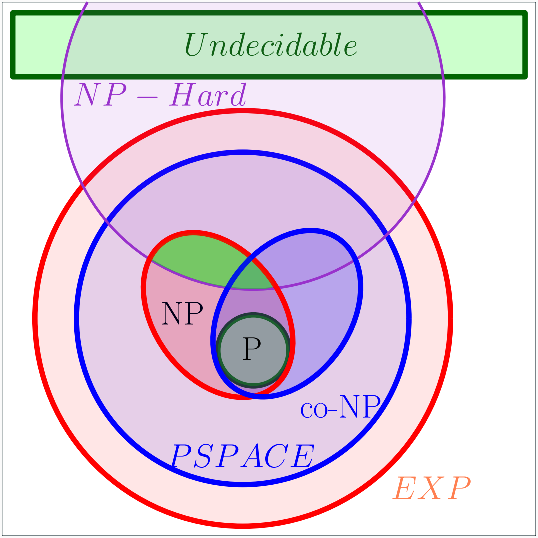
In the beginning...



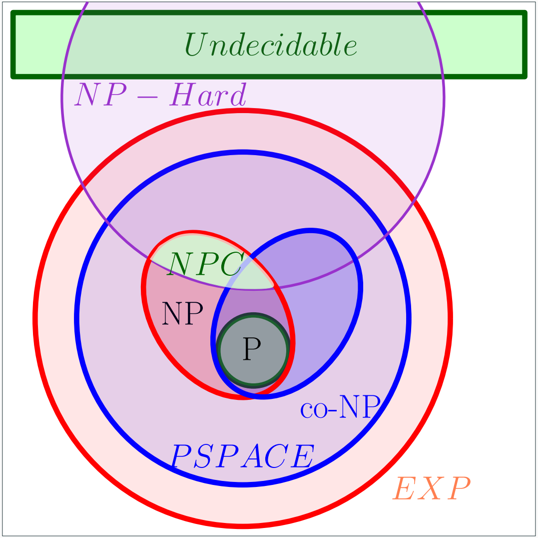
In the beginning...



In the beginning...



In the beginning...



Non-deterministic polynomial time - NP

P and NP and Turing Machines

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

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

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

Problems with no known deterministic polynomial time algorithms

Problems

- Independent Set
- Vertex Cover
- Set Cover
- SAT

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?

Problems with no known deterministic polynomial time algorithms

Problems

- Independent Set
- Vertex Cover
- Set Cover
- SAT

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?

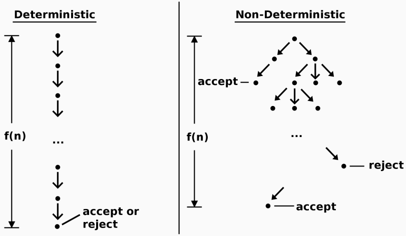
They can all be solved via a non-deterministic computer in polynomial time!

Non-determinism in computing

Non-determinism is a special property of algorithms.

An algorithm that is capable of taking multiple states concurrently. Whenever it reaches a choice, it takes both paths.

If there is a path for the string to be accepted by the machine, then the string is part of the language.



Problems with no known deterministic polynomial time algorithms

Problems

- Independent Set & Vertex Cover - Can build algorithm to check all possible collection of vertices
- Set Cover - Can check all possible collection of sets
- SAT -Can build a non-deterministic algorithm that checks every possible boolean assignment.

But we don't have access to a non-deterministic computer. So how can a deterministic computer verify that a algorithm is in NP?

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:

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

Certifiers

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

- For every $s \in X$ there is some string t such that $C(s, t) =$ "yes"
- If $s \notin X$, $C(s, t) =$ "no" for every t.

The string s is the problem instance. (Example: particular graph in independent set problem) The string t is called a certificate or proof for s.

Efficient (polynomial time) Certifiers

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

- For every $s \in X$ there is some string t such that $C(s,t) = \text{"yes"}$ and $|t| \leq p(|s|)$.
- If $s \notin X$, $C(s,t) = \text{"no"}$ for every t.
- $C(\cdot,\cdot)$ runs in polynomial time.

Example: Independent Set

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

Example: SAT

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

Why is it called Nondeterministic Polynomial Time

A certifier is an algorithm $C(I, c)$ with two inputs:

- I: instance.
- c: proof/certificate that the instance is indeed a YES instance of the given problem.

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

- Given I, the algorithm guesses (non-deterministically, and who knows how) a certificate c.
- The algorithm now verifies the certificate c for the instance I.

NP can be equivalently described using Turing machines.

Cook-Levin Theorem

“Hardest” Problems

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

Towards a definition

- Hardest problem must be in NP.
- 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-Complete** if

- $X \in NP$, and
- (Hardness) For any $Y \in NP$, $Y \leq_P X$.

Lemma
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

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

$\Leftarrow$ Since $P = NP$, and $X \in NP$, we have a polynomial time algorithm for X. $\square$

NP-Hard Problems

Definition
A problem Y is said to be NP-Hard if

- (Hardness) For any $X \in NP$, we have that $X \leq_P Y$.

An NP-Hard problem need not be in NP!

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

Consequences of proving NP-Completeness

If X is NP-Complete

- Since we believe $P \neq NP$,
- and solving X implies $P = NP$.

X is unlikely to be efficiently solvable.

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

Consequences of proving NP-Completeness

If X is NP-Complete

- Since we believe $P \neq NP$,
- and solving X implies $P = NP$.

X is unlikely to be efficiently solvable.

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

(This is proof by mob opinion — take with a grain of salt.)

NP-Complete Problems

Question
Are there any problems that are NP-Complete?

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

Cook-Levin Theorem

Theorem (Cook-Levin)
SAT is NP-Complete.

Cook-Levin Theorem

Theorem (Cook-Levin)
SAT is NP-Complete.

Need to show

- SAT is in NP.
- every NP problem X reduces to SAT.

Steve Cook won the Turing award for his theorem.

Proving that a problem X is NP-Complete

To prove X is NP-Complete, show

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

Proving that a problem X is NP-Complete

To prove X is NP-Complete, show

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

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

Proving that a problem X is NP-Complete

To prove X is NP-Complete, show

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

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

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

3-SAT is NP-Complete

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

NP-Completeness via Reductions

- SAT is NP-Complete due to Cook-Levin theorem
- SAT $\leq_P$ 3-SAT
- 3-SAT $\leq_P$ Independent Set
- Independent Set $\leq_P$ Vertex Cover
- Independent Set $\leq_P$ Clique
- 3-SAT $\leq_P$ 3-Color
- 3-SAT $\leq_P$ Hamiltonian Cycle

NP-Completeness via Reductions

- SAT is NP-Complete due to Cook-Levin theorem
- SAT $\leq_P$ 3-SAT
- 3-SAT $\leq_P$ Independent Set
- Independent Set $\leq_P$ Vertex Cover
- Independent Set $\leq_P$ Clique
- 3-SAT $\leq_P$ 3-Color
- 3-SAT $\leq_P$ Hamiltonian Cycle

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

A surprisingly frequent phenomenon!

Reducing 3-SAT to Independent Set

Independent Set

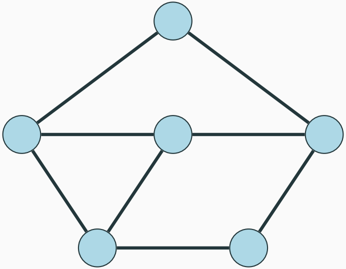
Problem: Independent Set

Instance: A graph G, integer k.

Question: Is there an independent set in G of size k?

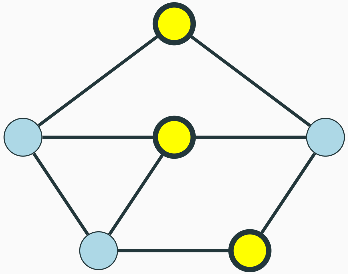
Problem: Independent Set

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



Problem: Independent Set

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



Interpreting 3SAT

There are two ways to think about 3SAT

- 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.
- 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

- 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
- 4- Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict
- 5- Take k to be the number of clauses

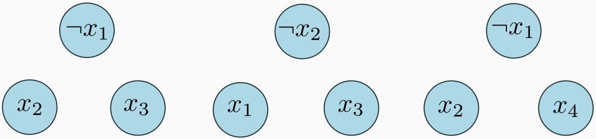


Figure 1: Graph for $\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

- 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
- 4- Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict
- 5- Take k to be the number of clauses

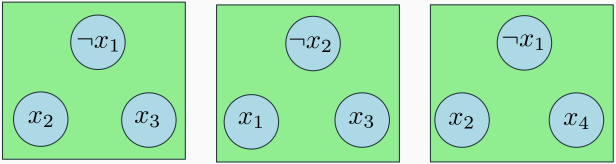


Figure 1: Graph for $\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

- 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
- 4- Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict
- 5- Take k to be the number of clauses

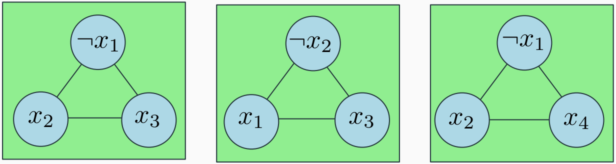


Figure 1: Graph for $\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

- 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
- 4- Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict
- 5- Take k to be the number of clauses

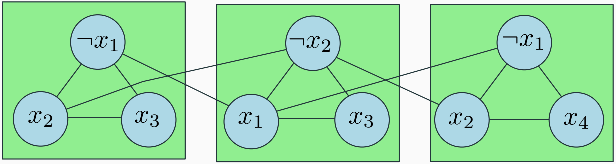


Figure 1: Graph for $\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

- 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
- 4- Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict
- 5- Take k to be the number of clauses

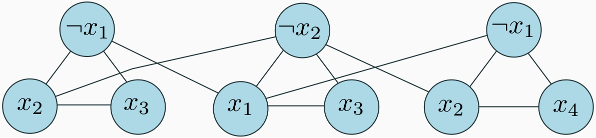


Figure 1: Graph for $\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

Lemma
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

- 2- Pick one of the vertices, corresponding to true literals under a, from each triangle. This is an independent set of the appropriate size. Why? $\square$

Correctness (contd)

Lemma
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

- S must contain exactly one vertex from each clause triangle
- S cannot contain vertices labeled by conflicting literals
- 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 $\square$

Other NP-Complete problems

Graph Coloring

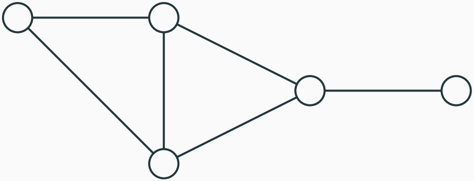
Problem: Graph Coloring

Instance: $G = (V, E)$: Undirected graph, integer k.
Question: Can the vertices of the graph be colored using k colors so that vertices connected by an edge do not get the same color?

Graph 3-Coloring

Problem: 3 Coloring

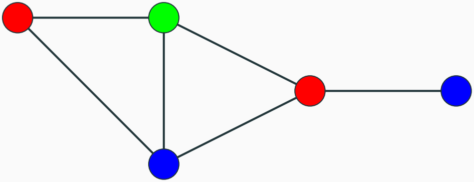
Instance: $G = (V, E)$: Undirected graph.
Question: Can the vertices of the graph be colored using 3 colors so that vertices connected by an edge do not get the same color?



‘

Problem: 3 Coloring

> **Instance**: $G = (V, E)$: Undirected graph.
> **Question**: Can the vertices of the graph be colored using 3 colors so that vertices connected by an edge do not get the same color?



'

Graph Coloring

Observation: If G is colored with k colors then each color class (nodes of same color) form an independent set in G. Thus, G can be partitioned into k independent sets iff G is k-colorable.

Graph 2-Coloring can be decided in polynomial time.

G is 2-colorable iff G is bipartite! There is a linear time algorithm to check if G is bipartite using Breadth-first-Search

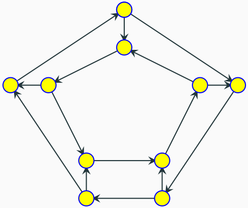
Hamiltonian Cycle

Directed Hamiltonian Cycle

Input Given a directed graph $G = (V, E)$ with n vertices

Goal Does G have a Hamiltonian cycle?

- 2- A Hamiltonian cycle is a cycle in the graph that visits every vertex in G exactly once



Directed Hamiltonian Cycle

Input Given a directed graph $G = (V, E)$ with n vertices

Goal Does G have a Hamiltonian cycle?

- 2- A Hamiltonian cycle is a cycle in the graph that visits every vertex in G exactly once