# NP-Hardness and Approximation Algorithms

Jared Saia
University of New Mexico

# Today's Outline

- P, NP and NP-Hardness
- Reductions
- Approximation Algorithms

# Efficient Algorithms

- We use polynomial time as our formal notion of efficiency: $O(n^c)$ for a constant $c$, where $n$ is the input length.
- A decision problem asks a yes/no question about an input.
- Our goal: understand how the difficulty of one problem can be related to the difficulty of another.

# NP-Hard Problems

- An NP-hard problem is at least as hard as every problem in NP, in the sense of polynomial-time reductions.
- A polynomial-time algorithm for any NP-hard problem would give polynomial-time algorithms for every problem in NP, implying $P = NP$.
- For NP-complete problems, no polynomial-time algorithm or super-polynomial worst-case lower bound is known.
- NP-hard problems need not belong to NP. We will distinguish NP-hardness from NP-completeness.

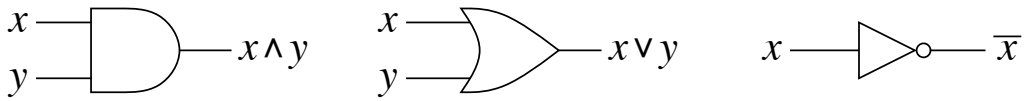
# Boolean Circuits

- A Boolean circuit is a directed acyclic graph of AND, OR, and NOT gates, with input bits $x_1, \ldots, x_m$ and one output bit.
- For reductions, split multi-input AND/OR gates into two-input gates; NOT gates have one input.
- Given an assignment to the input bits, evaluate the gates in topological order, using constant time per gate.
- Evaluation takes $O(N)$ time, where $N$ is the number of gates and wires in the circuit.

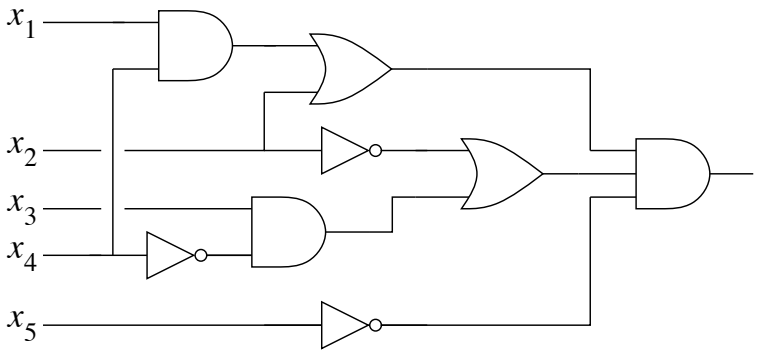
# Circuit Satisfiability

- **Circuit SAT:** Given a circuit, is there an input assignment that makes its output True? This problem is NP-complete.
- **Complementary question (Circuit UNSAT):** Does the circuit output False for every input assignment? This problem is co-NP-complete.
- Exhaustive search tries all $2^m$ assignments to the $m$ input bits, taking exponential time.
- No polynomial-time algorithm for Circuit SAT is known; proving that none exists would prove $P \neq NP$.

# Example




An and gate, an or gate, and a not gate.




A boolean circuit. Inputs enter from the left, and the output leaves to the right.

# P, NP, and co-NP

- **P:** decision problems solvable in time polynomial in the input length $n$.
- **NP:** decision problems with a verifier and a polynomial bound on certificate length. The verifier runs in time polynomial in $n$.
- A yes-instance has at least one accepting certificate; a no-instance has no accepting certificate.
- **co-NP:** complements of problems in NP. No-instances have polynomial-length certificates verifiable in polynomial time; yes-instances have no accepting certificates for a no answer.

# Example: Circuit SAT Is in NP

- **Certificate:** the $m$ input bits of a proposed satisfying assignment.
- **Verifier:** evaluate the circuit and accept exactly when its output is True.
- **Yes-instance:** a satisfying input assignment makes the verifier accept.
- **No-instance:** every input assignment makes the verifier reject.
- The certificate length and verification time are polynomial in the size of the circuit.

# P,NP, and co-NP

- If a problem is in P, then it is also in NP — to verify that the answer is yes in polynomial time, we can just throw away the proof and recompute the answer from scratch
- Similarly, any problem in P is also in co-NP
- In this sense, problems in P can only be easier than problems in NP and co-NP

# Examples

- The problem: "For a certain circuit and a set of inputs, is the output **True**?" is in P (and in NP and co-NP)
- The problem: "Does a certain circuit have an input that makes the output **True**?" is in NP
- The problem: "Does a certain circuit always have output true for any input?" is in co-NP

# P Examples

Most problems we've seen in this class so far are in P including:

- "Does there exist a path of distance $\leq d$ from $u$ to $v$ in the graph $G$?"
- "Does there exist a minimum spanning tree for a graph $G$ that has cost $\leq c$?"
- "Does there exist an alignment of strings $s_1$ and $s_2$ which has cost $\leq c$?"

# NP Examples

There are also several problems that are in NP (but probably not in P) including:

- **Circuit Satisfiability**
- **Coloring**: "Can we color the vertices of a graph $G$ with $c$ colors such that every edge has two different colors at its endpoints ($G$ and $c$ are inputs to the problem)
- **Clique**: "Is there a clique of size $k$ in a graph $G$?" ($G$ and $k$ are inputs to the problem)
- **Hamiltonian Path**: "Does there exist a path for a graph $G$ that visits every vertex exactly once?"

# The $1 Million Question

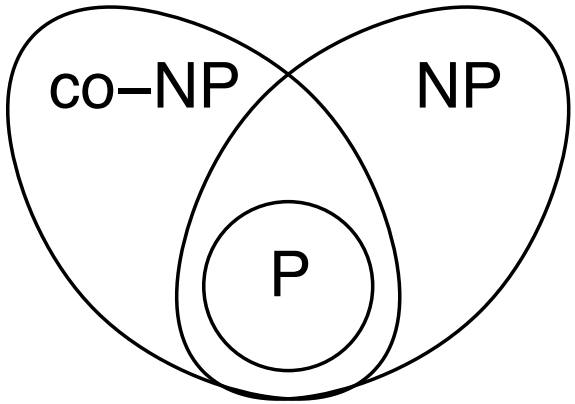
- The most important question in computer science (and one of the most important in mathematics) is: "Does P=NP?"
- Nobody knows.
- Intuitively, it *seems* obvious that P≠NP; in this class you've seen that some problems can be very difficult to solve, even though the solutions are obvious once you see them
- But nobody has proven that P≠NP

# NP and co-NP

- Notice that the definition of NP (and co-NP) is not symmetric.
- Just because we can verify every yes answer quickly doesn't mean that we can check no answers quickly
- For example, as far as we know, there is no short proof that a boolean circuit is *not* satisfiable
- In other words, we know that Circuit Satisfiability is in NP but we don't know if its in co-NP

# Conjectures

- We conjecture that P≠NP and that NP≠co-NP
- Here's a picture of what we *think* the world looks like:



# Polynomial-Time Reductions

For decision problems $A$ and $B$, write $A \leq_p B$ if a polynomial-time transformation $f$ maps instances of $A$ to instances of $B$ such that

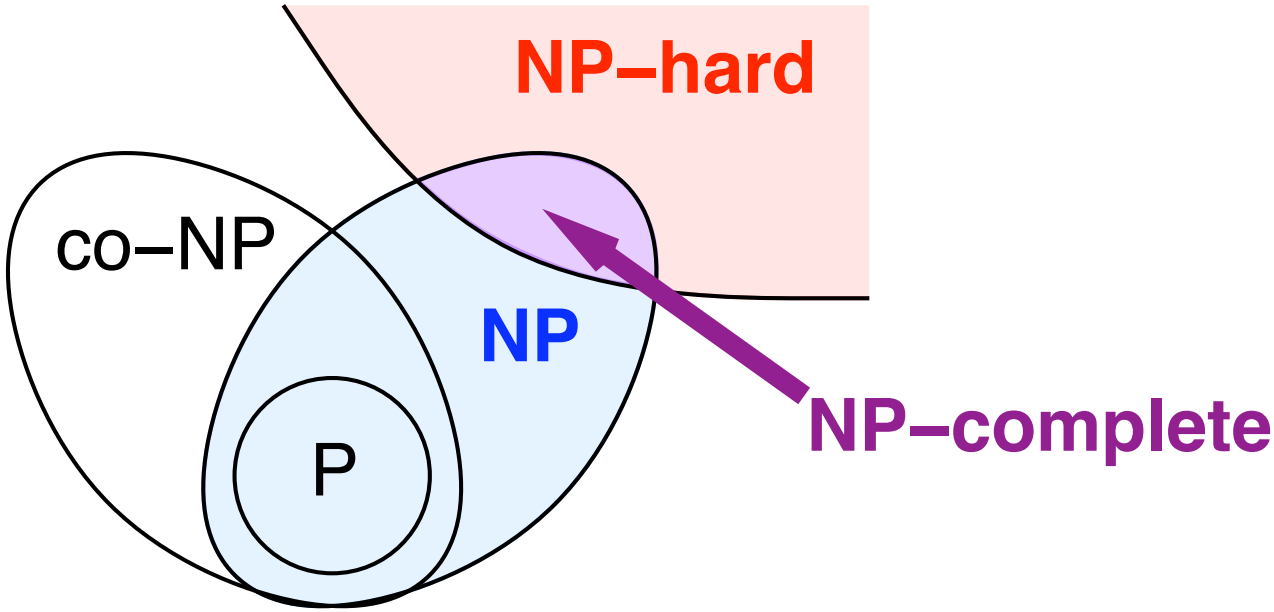
$x$ is a yes-instance of $A \iff f(x)$ is a yes-instance of $B$.

- An algorithm for $B$ would solve $A$: compute $f(x)$, run the algorithm for $B$, and return its answer.
- A decision problem $B$ is **NP-hard** if $A \leq_p B$ for every problem $A \in$ NP.
- Thus, a polynomial-time algorithm for $B$ would imply $P = NP$.

# NP-Complete

- A decision problem is **NP-complete** if it is both **NP-hard** and **in NP**.
- These are the hardest problems in NP under polynomial-time reductions.
- A polynomial-time algorithm for one NP-complete problem would give one for every problem in NP.
- Thousands of problems are known to be NP-complete. Their reductions connect many apparently different algorithmic problems.

# Example




A more detailed picture of what we *think* the world looks like.

# Proving NP-Hardness

**Cook–Levin theorem:** Circuit SAT is NP-complete.

To prove that a target decision problem $B$ is NP-hard:

1. Start with a **known NP-hard problem** $A$.
2. Construct $f(x)$, an instance of $B$, in polynomial time.
3. Prove both directions:

$$x \in A \quad \Longleftrightarrow \quad f(x) \in B.$$

The direction matters: **reduce the known-hard problem to the target**, $A \leq_p B$.

# SAT

- Consider the *formula satisfiability* problem (aka *SAT*)
- The input to SAT is a boolean formula like

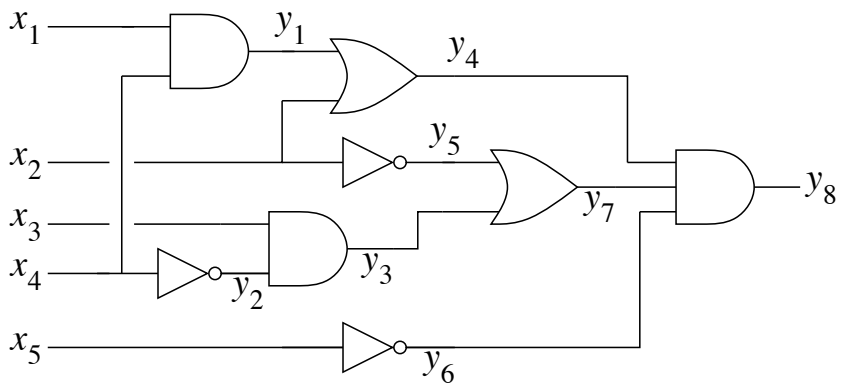
$$(a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee \overline{(\bar{a} \Rightarrow d)} \vee (c \neq a \wedge b)),$$

- The question is whether it is possible to assign boolean values to the variables $a, b, c, \ldots$ so that the formula evaluates to TRUE
- To show that SAT is NP-Hard, we need to show that we can use a solution to SAT to solve Circuit Satisfiability

# Circuit SAT $\leq_p$ SAT: Construction

- **Input:** a Boolean circuit. Introduce a variable for the output of each gate.
- Write a constraint saying that each gate's output equals the Boolean function of its inputs.
- Conjoin all gate constraints **and require the circuit's output variable to be True**.
- This produces a formula of linear size in the circuit's number of gates and wires.

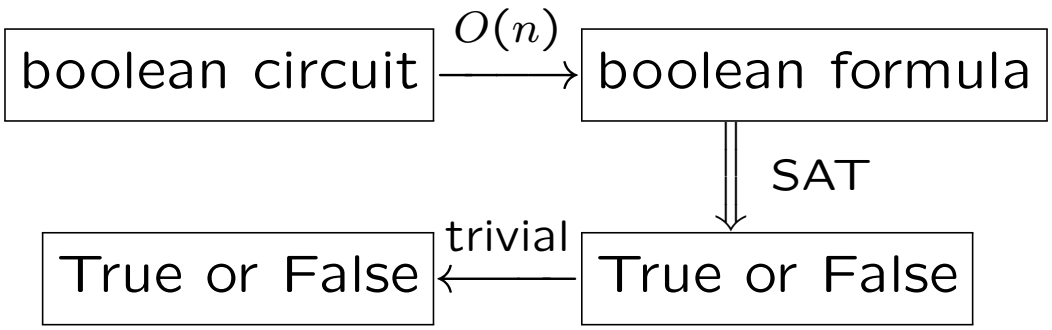
# Example




$$(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge (y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \wedge (y_8 = y_4 \wedge y_7 \wedge y_6) \wedge y_8$$

A boolean circuit with gate variables added, and an equivalent boolean formula.

# Reduction Picture



# Circuit SAT $\leq_p$ SAT: Correctness

**Claim:** The circuit is satisfiable iff the constructed formula is satisfiable.

**Proof.** If an input assignment makes the circuit output True, assign each gate variable its actual output. All gate constraints and the final output constraint hold.

Conversely, a satisfying assignment to the formula obeys every gate constraint. Evaluating the circuit in topological order therefore gives the assigned values at all gates, including True at the output. □

The construction takes polynomial time, so SAT is NP-hard.

# SAT Is NP-Complete

- We have proved that SAT is NP-hard.
- **Certificate:** an assignment of one bit to each variable.
- **Verifier:** evaluate the formula's parse tree and accept exactly when the result is True.
- A satisfiable formula has an accepting certificate; an unsatisfiable formula has none.
- The certificate has polynomial length, and evaluation takes linear time in the formula size. Thus SAT is in NP and is NP-complete.

# An NP-Completeness Proof

**Membership in NP.** Describe a polynomial-length certificate and a polynomial-time verifier. Show that yes-instances have an accepting certificate and no-instances have none.

**NP-hardness.** Choose a known NP-hard problem $A$ and reduce it to the target $B$. Give the construction, bound its running time, and prove

$$x \in A \quad \Longleftrightarrow \quad f(x) \in B.$$

**Conclusion.** $B$ is NP-hard and in NP, hence NP-complete.

# 3-SAT

- A boolean formula is in *conjunctive normal form* (CNF) if it is a conjunction (and) of several *clauses*, each of which is the disjunction (or) of several *literals*, each of which is either a variable or its negation. For example:

$$\overbrace{(a \vee b \vee c \vee d)}^{\text{clause}} \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b})$$

- A *3CNF* formula is a CNF formula with exactly three literals per clause; repeated literals are allowed.
- The 3-SAT problem is just: "Is there any assignment of variables to a 3CNF formula that makes the formula evaluate to true?"

# 3-SAT

- 3-SAT is just a restricted version of SAT
- Surprisingly, 3-SAT also turns out to be NP-Complete
- 3-SAT is very useful in proving NP-Hardness results for other problems, we'll see how it can be used to show that CLIQUE is NP-Hard

# SAT to CNF: Gate Constraints

Represent the formula by a circuit with two-input AND/OR gates and one-input NOT gates. Introduce a fresh output variable $y$ per gate.

| Gate | CNF constraints |
|---|---|
| $y = a \wedge b$ | $(\neg y \vee a) \wedge (\neg y \vee b)$ |
| | $\wedge\, (y \vee \neg a \vee \neg b)$ |
| $y = a \vee b$ | $(y \vee \neg a) \wedge (y \vee \neg b)$ |
| | $\wedge\, (\neg y \vee a \vee b)$ |
| $y = \neg a$ | $(y \vee a) \wedge (\neg y \vee \neg a)$ |

AND together the constraints for every gate and the unit clause ($y_{\text{out}}$). The result has linear size.

# SAT to CNF: Correctness

**Claim:** The original formula is satisfiable iff the new CNF formula is satisfiable.

**Proof.** A satisfying assignment to the original variables extends to the gate variables by evaluating the circuit. Every gate constraint holds, and the output is True.

Conversely, the CNF constraints force each gate variable to equal its gate's output. The unit output clause forces the original formula to evaluate to True. □

# CNF to 3-SAT: Longer Clauses

For completeness, a CNF clause with $k \geq 4$ literals $(\ell_1 \vee \cdots \vee \ell_k)$ can also be split directly.

1. Introduce fresh variables $z_i$ for $i \in \{1, \ldots, k-3\}$. Use separate fresh variables for each original clause.
2. Replace the original clause by ANDing together these $k-2$ clauses:

$$(\ell_1 \vee \ell_2 \vee z_1),$$

$$(\neg z_i \vee \ell_{i+2} \vee z_{i+1}) \quad \text{for } i \in \{1, \ldots, k-4\},$$

$$(\neg z_{k-3} \vee \ell_{k-1} \vee \ell_k).$$

When $k = 4$, there are no middle clauses.

The total size remains linear in the number of literal occurrences.

# Why the Clause Chain Works

**Claim:** An assignment satisfies the original clause iff it extends to an assignment satisfying the chain.

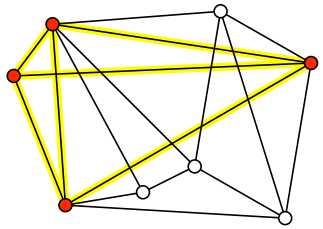
**Proof.** If the original clause is true, set $z_i$ to True exactly when all of $\ell_1, \ldots, \ell_{i+1}$ are false, for each $i \in \{1, \ldots, k-3\}$. This satisfies the first clause, each middle clause, and the last clause.

Conversely, if all original literals are false, the first clause forces $z_1$ to be True. Successive middle clauses force every $z_i$ to be True. The last clause is then false. □

Short clauses are padded by repeating literals. An empty clause makes the instance unsatisfiable and can be replaced by $(t \vee t \vee t) \wedge (\neg t \vee \neg t \vee \neg t)$.

# CLIQUE

- The problem CLIQUE asks "Is there a clique of size $k$ in a graph $G$?"
- Example graph with clique of size 4:



- We'll show that Clique is NP-Hard using a reduction from 3-SAT. (the proof that Clique is in NP is left as an exercise)

# 3-SAT $\leq_p$ CLIQUE: Construction

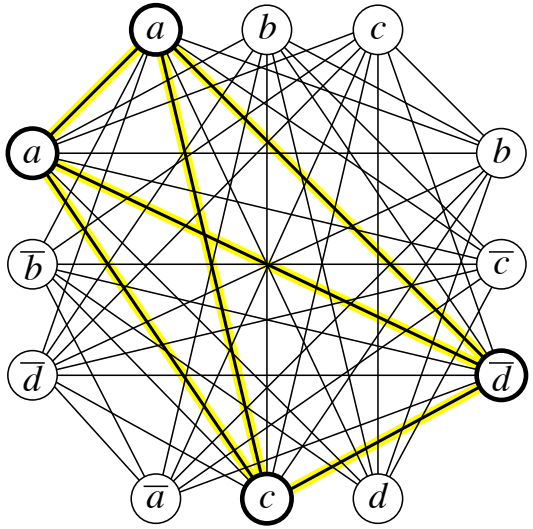
Given a 3-CNF formula $F$ with $k$ clauses, output $(G, k)$:

- Create a separate vertex for every literal occurrence: $3k$ vertices in total.
- Join two vertices iff their literals occur in different clauses and do not contradict each other.
- There are $O(k^2)$ candidate pairs, so construction takes $O(k^2)$ time and produces at most $O(k^2)$ edges.

We will prove that $F$ is satisfiable iff $G$ has a $k$-clique.

# Reduction Example

- Let $F$ be the formula:
  $(a \vee b \vee c) \wedge (b \vee \overline{c} \vee \overline{d}) \wedge (\overline{a} \vee c \vee d) \wedge (a \vee \overline{b} \vee \overline{d})$
- This formula is transformed into the following graph:




(look for the edges that *aren't* in the graph)

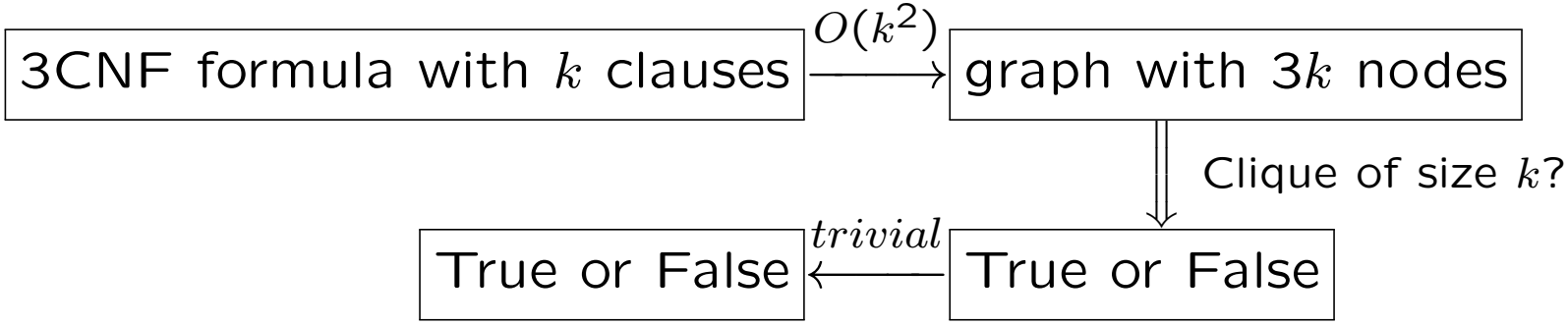
# 3-SAT $\leq_p$ CLIQUE: Correctness

**Claim:** $F$ is satisfiable iff $G$ has a clique of size $k$.

**Proof.** If $F$ is satisfiable, choose one true literal occurrence from each clause. These $k$ vertices belong to different clauses and no pair contradicts, so all pairs are adjacent.

Conversely, a $k$-clique contains one vertex from each of the $k$ clauses, since no two vertices of a clause are adjacent. Set its literals to True. The edges guarantee consistency; assign any remaining variables arbitrarily. Every clause now has a true literal. □

# 3-SAT $\leq_p$ CLIQUE



# In-Class Exercise

Consider the formula: $(a \vee b) \wedge (b \vee \overline{c}) \wedge (c \vee \overline{b})$

- Q1: Transform this formula into a graph, $G$, using the reduction just given.
- Q2: What is the maximum clique size in $G$? Give the vertices in this maximum clique.

# CLIQUE is NP-Complete

- To show CLIQUE is NP-Complete, must now show it is in NP
- An efficiently checkable proof (aka certificate) that there is a CLIQUE of size k in a graph $G = (V, E)$ is just a subset $V'$ of $V$ of size $k$
- To check this proof, need to go through every pair of nodes $u$ and $v$ in $V'$ and check that $(u, v) \in E$.
- This just takes polynomial time in the input size.

# Independent Set

- Independent Set is the following problem: "Does there exist a set of $k$ vertices in a graph $G$ with no edges between them?"
- It is easy to show that independent set is NP-Hard by a reduction from CLIQUE (will do now in class).
- Thus we can now use Independent Set to show that other problems are NP-Hard

# Vertex Cover

- A *vertex cover* of a graph is a set of vertices that touches every edge in the graph
- The problem *Vertex Cover* is: "Does there exist a vertex cover of size at most $k$ in a graph $G$?"
- We can prove this problem is NP-Hard by an easy reduction from Independent Set

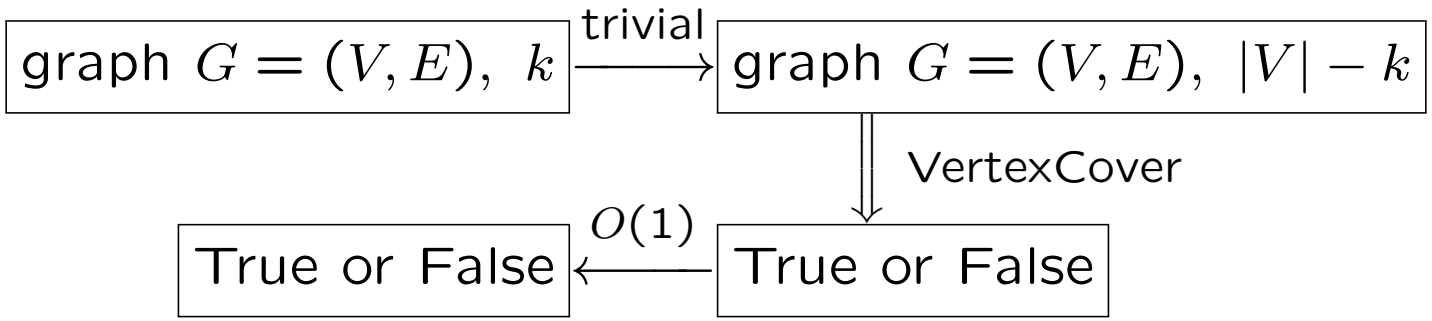
# Key Observation

- Key Observation: If $I$ is an independent set in a graph $G = (V, E)$, then $V - I$ is a vertex cover.
- Thus, there is an independent set of size $k$ iff there is a vertex cover of size $|V| - k$.
- For the reduction, we want to show that a polynomial time algorithm for Vertex Cover can give a polynomial time algorithm for Independent Set

# The Reduction

- We are given a graph $G = (V, E)$ and a value $k$ and we must determine if there is an independent set of size $k$ in $G$.
- To do this, we ask if there is a vertex cover of size $|V| - k$ in $G$.
- If so then we return that there *is* an independent set of size $k$ in $G$
- If not, we return that there *is not* an independent set of size $k$ in $G$

# The Reduction



# Graph Coloring

- A $c$-coloring of a graph $G$ is a map $C : V \to \{1, 2, \ldots, c\}$ that assigns one of $c$ "colors" to each vertex so that every edge has two different colors at its endpoints
- The graph coloring problem is: "Does there exist a $c$-coloring for the graph $G$?"
- Even when $c = 3$, this problem is hard. We call this problem *3Colorable* i.e. "Does there exist a 3-coloring for the graph $G$?"

# 3Colorable

- To show that 3Colorable is NP-hard, we will reduce from 3Sat
- This means that we want to show that a polynomial time algorithm for 3Colorable can give a polynomial time algorithm for 3Sat
- Recall that the 3-SAT problem is just: "Is there any assignment of variables to a 3CNF formula that makes the formula evaluate to true?"
- And a 3CNF formula is just a conjunct of a bunch of clauses, each of which contains exactly 3 literals e.g.

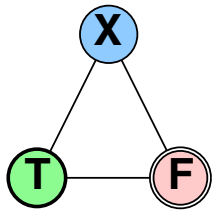
$$\overbrace{(a \vee b \vee c)}^{\text{clause}} \wedge (b \vee \overline{c} \vee \overline{d}) \wedge (\overline{a} \vee c \vee d) \wedge (a \vee \overline{b} \vee d)$$

# Reduction

- We are given a 3-CNF formula, $F$, and we must determine if it has a satisfying assignment
- To do this, we produce a graph as follows
- The graph contains one *truth* gadget, one *variable* gadget for each variable in the formula, and one *clause* gadget for each clause in the formula

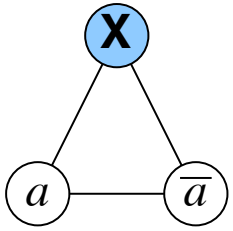
# The Truth Gadget

- The truth gadget is just a triangle with three vertices $T$, $F$ and $X$, which intuitively stand for **True**, **False**, and **other**
- Since these vertices are all connected, they must have different colors in any 3-coloring
- For the sake of convenience, we will name those colors **True**, **False**, and **Other**
- Thus when we say a node is colored "True", we just mean that it's colored the same color as the node $T$



# The Variable Gadgets

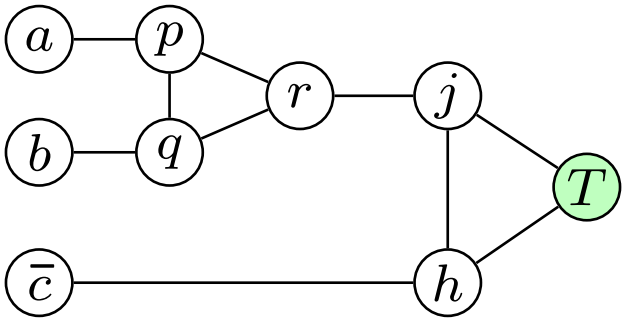
- The variable gadget for a variable $a$ is also a triangle joining two new nodes labeled $a$ and $\overline{a}$ to node $X$ in the truth gadget
- Node $a$ must be colored either "True" or "False", and so node $\overline{a}$ must be colored either "False" or "True", respectively.




- The variable gadget ensures that each of the literals is colored either "True" or "False"

# The Clause Gadget

Join literal vertices $a, b, \bar{c}$ to the shared vertex $T$ using five new vertices and ten edges.




**Claim:** The coloring extends to this gadget iff at least one of $a, b, \bar{c}$ is colored True.

# Clause Gadget: Two Inputs

In the first triangle, $p$ is adjacent to input $a$, $q$ to input $b$, and $p, q, r$ form a triangle. Let $T, F, X$ denote the three colors.

If both inputs are $F$, then $p, q$ use $T, X$, forcing $r = F$. Otherwise, we can choose $r = T$. These colorings show all cases:

| $a$ | $b$ | $p$ | $q$ | $r$ |
|---|---|---|---|---|
| $F$ | $F$ | $T$ | $X$ | $F$ |
| $F$ | $T$ | $X$ | $F$ | $T$ |
| $T$ | $F$ | $F$ | $X$ | $T$ |
| $T$ | $T$ | $F$ | $X$ | $T$ |

Thus we can make $r$ equal the OR of the two input truth values.

# Clause Gadget: Correctness

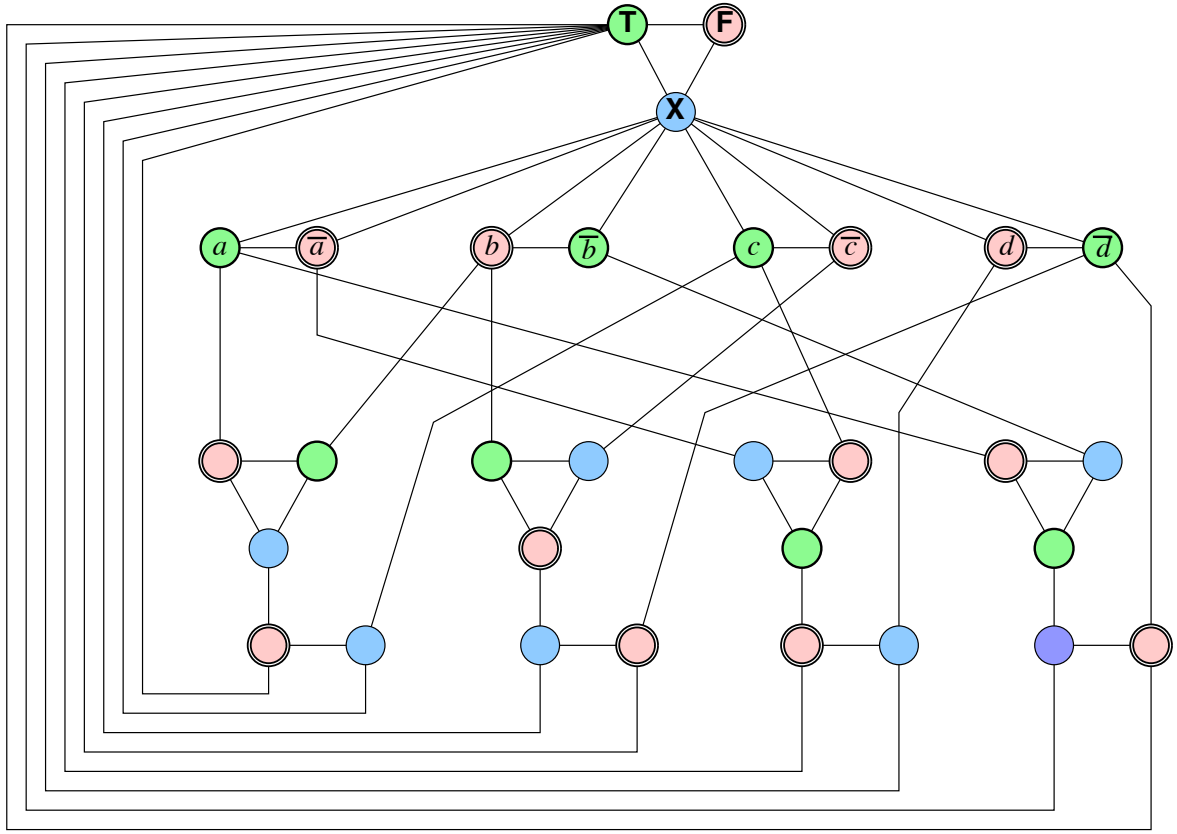
**Proof.** The second triangle is the same two-input component, with inputs $r, \bar{c}$, internal vertices $j, h$, and output fixed to $T$.

If $a, b, \bar{c}$ are all False, the first triangle forces $r = F$. The second then forces its output to be $F$, contradicting the fixed color of $T$.

If at least one literal is True, use the preceding table to color the first triangle with $r = a \vee b$. At least one of $r, \bar{c}$ is now True. Apply the same table to color the second triangle with output $T$. □

# Example

Consider the formula $(a \vee b \vee c) \wedge (b \vee \overline{c} \vee \overline{d}) \wedge (\overline{a} \vee c \vee d) \wedge (a \vee \overline{b} \vee \overline{d})$.
Following is the graph created by the reduction:



# Example

- Note that the 3-coloring of this example graph corresponds to a satisfying assignment of the formula
- Namely, $a = c =$ True, $b = d =$ False.
- Note that the final graph contains only *one* node $T$, only *one* node $F$, only *one* node $\bar{a}$ for each variable $a$ and so on

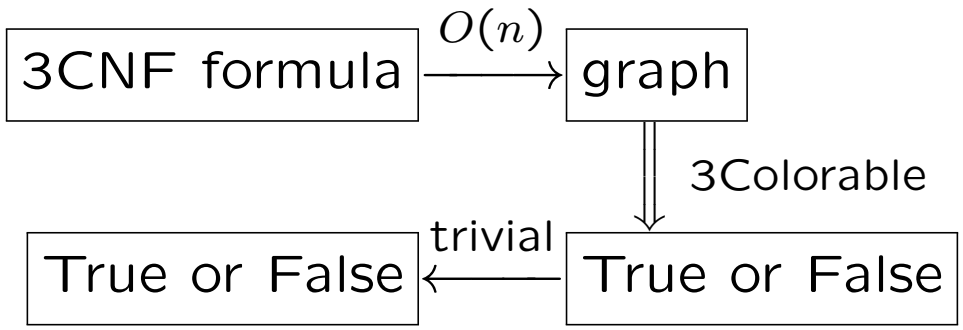
# 3-SAT $\leq_p$ 3-Coloring: Correctness

**Claim:** The constructed graph is 3-colorable iff $F$ is satisfiable.

**Proof.** A 3-coloring names the three colors using the truth triangle. Every variable gadget assigns opposite truth values to a variable and its negation. By the clause-gadget claim, each clause has a true literal, giving a satisfying assignment.

Conversely, a satisfying assignment colors the truth and variable gadgets. By the clause-gadget claim, each clause gadget can be completed. Its internal vertices are private to that clause, so these completions do not conflict. □

# Reduction Picture



# Wrap Up

- We've just shown that if 3Colorable can be solved in polynomial time then 3-SAT can be solved in polynomial time
- This shows that 3Colorable is NP-Hard
- To show that 3Colorable is in NP, we just need to note that we can easily verify that a graph has been correctly 3-colored in linear time: just compare the endpoints of every edge
- Thus, 3Coloring is NP-Complete.
- This implies that the more general graph coloring problem is also NP-Complete

# In-Class Exercise

Consider the problem *4Colorable*: "Does there exist a 4-coloring for a graph $G$?"

- Q1: Show this problem is in NP by showing that there exists an efficiently verifiable proof of the fact that a graph is 4 colorable.
- Q2: Show the problem is NP-Hard by a reduction from the problem 3Colorable. In particular, show the following:
  - Given a graph $G$, you can create a graph $G'$ such that $G'$ is 4-colorable iff $G$ is 3-colorable.
  - Creating $G'$ from $G$ takes polynomial time

Note: You've now shown that 4Colorable is NP-Complete!

# Hamiltonian Cycle

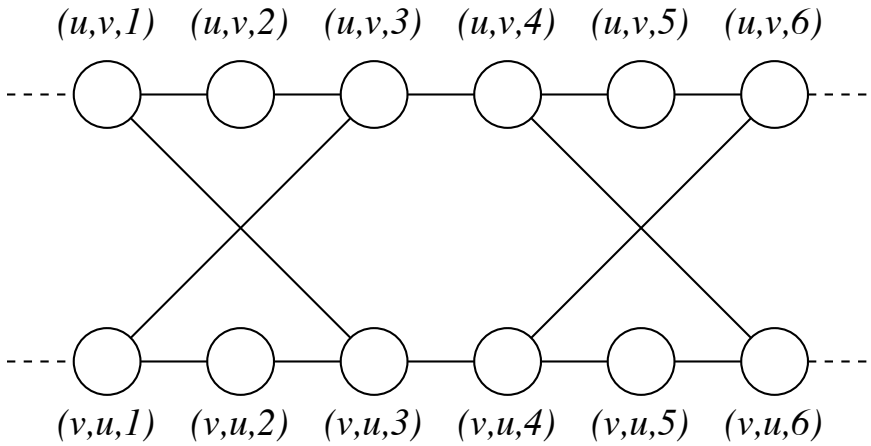
- A *Hamiltonian Cycle* in a graph is a cycle that visits every vertex exactly once (note that this is very different from an *Eulerian cycle* which visits every *edge* exactly once)
- The Hamiltonian Cycle problem is to determine if a given graph $G$ has a Hamiltonian Cycle
- We will show that this problem is NP-Hard by a reduction from the vertex cover problem.

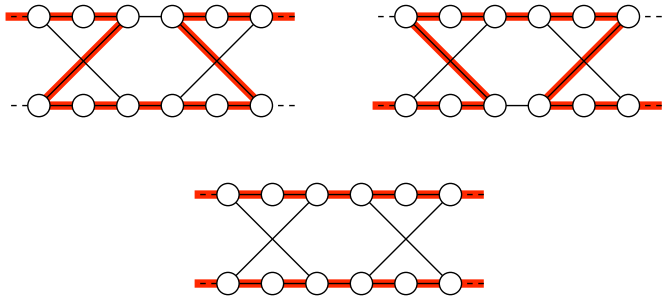
# The Reduction

- To do the reduction, we need to show that we can solve Vertex Cover in polynomial time if we have a polynomial time solution to Hamiltonian Cycle.
- Given a graph $G$ and an integer $k$, we will create another graph $G'$ such that $G'$ has a Hamiltonian cycle iff $G$ has a vertex cover of size $k$
- $G'$ contains $k$ *cover vertices*, numbered 1 through $k$.
- As in the last reduction, our transformation will consist of putting together several "gadgets"

# Edge Gadget and Cover Vertices

- For each edge $(u, v)$ in $G$, we have an *edge gadget* in $G'$ consisting of twelve vertices and fourteen edges, as shown below



An edge gadget for $(u, v)$ and the only possible Hamiltonian paths through it.

# Edge Gadget

- The four corner vertices $(u, v, 1)$, $(u, v, 6)$, $(v, u, 1)$, and $(v, u, 6)$ each have an edge leaving the gadget
- A Hamiltonian cycle can only pass through an edge gadget in one of the three ways shown in the figure
- These paths through the edge gadget will correspond to one or both of the vertices $u$ and $v$ being in the vertex cover.

# Vertex Chains

- For each vertex $u$ in $G$ with positive degree, we string together all the edge gadgets for edges $(u, v)$ into a single *vertex chain* and then connect the ends of the chain to all the cover vertices
- Specifically, suppose $u$ has $d \geq 1$ neighbors $v_1, v_2, \ldots, v_d$. Then $G'$ has the following edges:
  - $d - 1$ edges between $(u, v_i, 6)$ and $(u, v_{i+1}, 1)$ (for all $i$ between 1 and $d - 1$)
  - $k$ edges between the cover vertices and $(u, v_1, 1)$
  - $k$ edges between the cover vertices and $(u, v_d, 6)$

# The Reduction

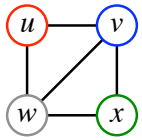
- It's not hard to prove that if $\{v_1, v_2, \ldots, v_k\}$ is a vertex cover of $G$, then $G'$ has a Hamiltonian cycle
- To get this Hamiltonian cycle, we start at cover vertex 1, traverse through the vertex chain for $v_1$, then visit cover vertex 2, then traverse the vertex chain for $v_2$ and so forth, until we eventually return to cover vertex 1
- Conversely, one can prove that any Hamiltonian cycle in $G'$ alternates between cover vertices and vertex chains, and that the vertex chains correspond to the $k$ vertices in a vertex cover of $G$

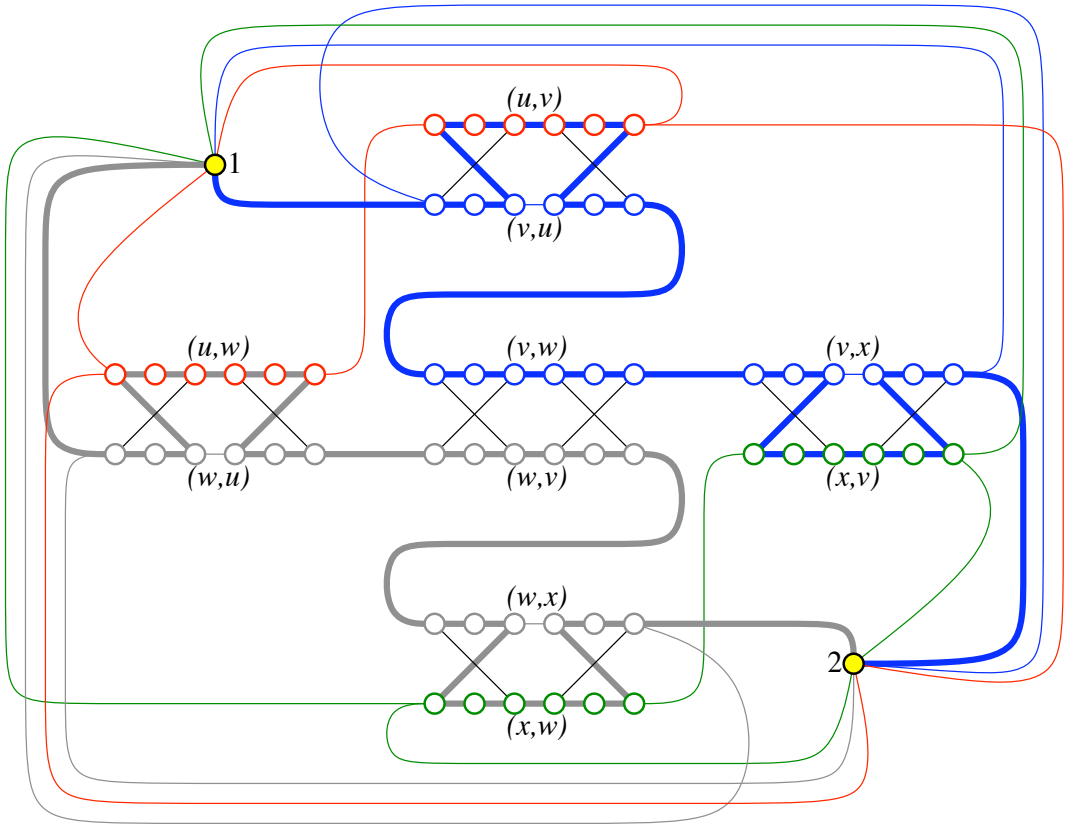
Thus, $G$ has a vertex cover of size $k$ iff $G'$ has a Hamiltonian cycle

# The Reduction

- The transformation from $G$ to $G'$ takes at most $O(|V|^2)$ time, so the Hamiltonian cycle problem is NP-Hard
- Moreover we can easily verify a Hamiltonian cycle in linear time, thus Hamiltonian Cycle is also in NP
- Thus Hamiltonian Cycle is NP-Complete

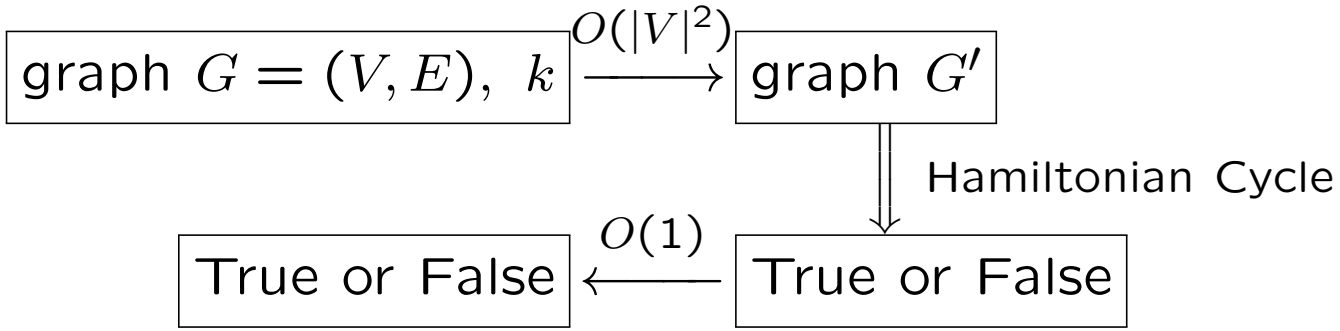
# Example




The original graph $G$ with vertex cover $\{v, w\}$, and the transformed graph $G'$ with a corresponding Hamiltonian cycle (bold edges).
Vertex chains are colored to match their corresponding vertices.

# The Reduction



# Hamiltonian Cycle $\leq_p$ TSP

Given an undirected graph $G = (V, E)$ with $n$ vertices, build a complete weighted graph on $V$. For distinct $u, v \in V$, set

$$w(u,v) = \begin{cases} 1 & \text{if } \{u,v\} \in E, \\ 2 & \text{otherwise.} \end{cases}$$

**Claim:** $G$ has a Hamiltonian cycle iff the new graph has a tour of cost at most $n$.

**Proof.** An original Hamiltonian cycle uses $n$ weight-1 edges. Conversely, a tour has $n$ edges of weight at least 1; cost at most $n$ forces every edge to belong to $E$. □

Construction takes $O(n^2)$ time. An exact TSP optimizer would answer the decision question, so optimization TSP is NP-hard.

# NP-Hard Games

- In 1999, Richard Kaye proved that the solitaire game Minesweeper is NP-Hard, using a reduction from Circuit Satisfiability.
- Subsequently, people have shown that Sudoku, Donkey Kong, Zelda, Pac-Man, Tetris, Mario, Lemmings are NP-Hard

# Challenge Problem

- Consider the *optimization* version of, say, the graph coloring problem: "Given a graph $G$, what is the smallest number of colors needed to color the graph?" (Note that unlike the *decision* version of this problem, this is not a yes/no question)
- Show that the optimization version of graph coloring is also NP-Hard by a reduction from the decision version of graph coloring.
- Is the optimization version of graph coloring also NP-Complete?

# Challenge Problem

- Consider the problem 4SAT which is: "Is there any assignment of variables to a 4CNF formula that makes the formula evaluate to true?"
- Is this problem NP-Hard? If so, give a reduction from 3SAT that shows this. If not, give a polynomial time algorithm which solves it.

# Fixed-Size CLIQUE

Consider the problem: "Does an input graph $G$ contain a clique of size 5?"

- Give an algorithm and analyze its running time in terms of $n = |V|$.
- Explain why fixing the clique size changes the complexity compared with CLIQUE, where $k$ is part of the input.

# Vertex Cover

- A *vertex cover* of a graph is a set of vertices that touches every edge in the graph
- The decision version of *Vertex Cover* is: "Does there exist a vertex cover of size $k$ in a graph $G$?" .
- We've proven this problem is NP-Hard by an easy reduction from Independent Set
- The *optimization* version of *Vertex Cover* is: "What is the minimum size vertex cover of a graph $G$?"
- We can prove this problem is NP-Hard by a reduction from the decision version of Vertex Cover (left as an exercise).

# Approximation Algorithms

For a minimization problem with nonnegative costs, a $\rho$-**approximation algorithm** runs in polynomial time and returns a feasible solution of cost at most $\rho\, OPT$, for every instance.

- $OPT$ is the cost of an optimal solution; $\rho \geq 1$ is the approximation factor.
- We will give a 2-approximation for minimum vertex cover.
- We will also give a 2-approximation for TSP under the triangle-inequality assumption.

# Approximation Algorithm

- The approximation algorithm does the following until $G$ has no more edges:
- It chooses an arbitrary edge $(u, v)$ in $G$ and includes both $u$ and $v$ in the cover
- It then removes from $G$ all edges which are incident to either $u$ or $v$

# Approximate Vertex Cover

**Approx-Vertex-Cover**$(G = (V, E))$

$C \leftarrow \varnothing$; $E' \leftarrow E$

**while** $E' \neq \varnothing$ **do**

Choose any edge $\{u, v\} \in E'$

$C \leftarrow C \cup \{u, v\}$

Remove from $E'$ every edge incident to $u$ or $v$

**return** $C$

# Vertex Cover: Runtime

- Scan the edge list, selecting an edge only when neither endpoint has been selected before.
- Mark selected vertices. An edge incident to a marked vertex is skipped; no physical edge deletion is needed.
- Each edge is processed $O(1)$ times, with constant work per check.
- Initializing the marks and scanning the edges takes

$$\boxed{O(|V| + |E|)}.$$

# Vertex Cover: Guarantee

**Theorem 1:** The algorithm returns a vertex cover $C$ with $|C| \leq 2|C^*|$, where $C^*$ is an optimal cover.

**Proof.** Every removed edge has an endpoint in $C$. Since no edges remain, $C$ is a vertex cover.

Let $A$ be the set of chosen edges. No two edges of $A$ share an endpoint: choosing an edge removes every edge incident to its endpoints. Thus $A$ is a matching, and every cover must select a distinct endpoint for each edge of $A$. Hence $|C^*| \geq |A|$.

The algorithm selects both endpoints of every chosen edge, so

$$|C| = 2|A| \leq 2|C^*|. \qquad \square$$

# Metric TSP

- Given a complete, undirected graph $G$ with nonnegative edge weights satisfying the *triangle inequality*, find a minimum-weight Hamiltonian cycle.
- Completeness guarantees that every shortcut edge used by the approximation algorithm exists.
- This problem remains NP-hard: the earlier reduction's weights 1 and 2 satisfy the triangle inequality.
- Under these assumptions, we will give a 2-approximation algorithm.

# Triangle Inequality

- In many practical problems, it's reasonable to make the assumption that the weights, $c$, of the edges obey the *triangle inequality*
- The triangle inequality says that for all vertices $u, v, w \in V$:

$$c(u, w) \leq c(u, v) + c(v, w)$$

- In other words, the cheapest way to get from $u$ to $w$ is always to just take the edge $(u, w)$
- In the real world, this is usually a pretty natural assumption. For example it holds if the vertices are points in a plane and the cost of traveling between two vertices is just the euclidean distance between them.

# TSP: The Algorithm

1. Compute a minimum spanning tree $T$ and choose a root $r$.
2. Traverse $T$ by DFS. Record each vertex only on its **first visit**, producing an ordering $L$.
3. Visit the vertices in order $L$, then **return to** $r$ to close the tour.

The proof compares this tour with the closed DFS walk that traverses each tree edge twice.

# Approximate TSP

**Approx-TSP**$(G)$

$T \leftarrow \mathrm{MST}(G)$

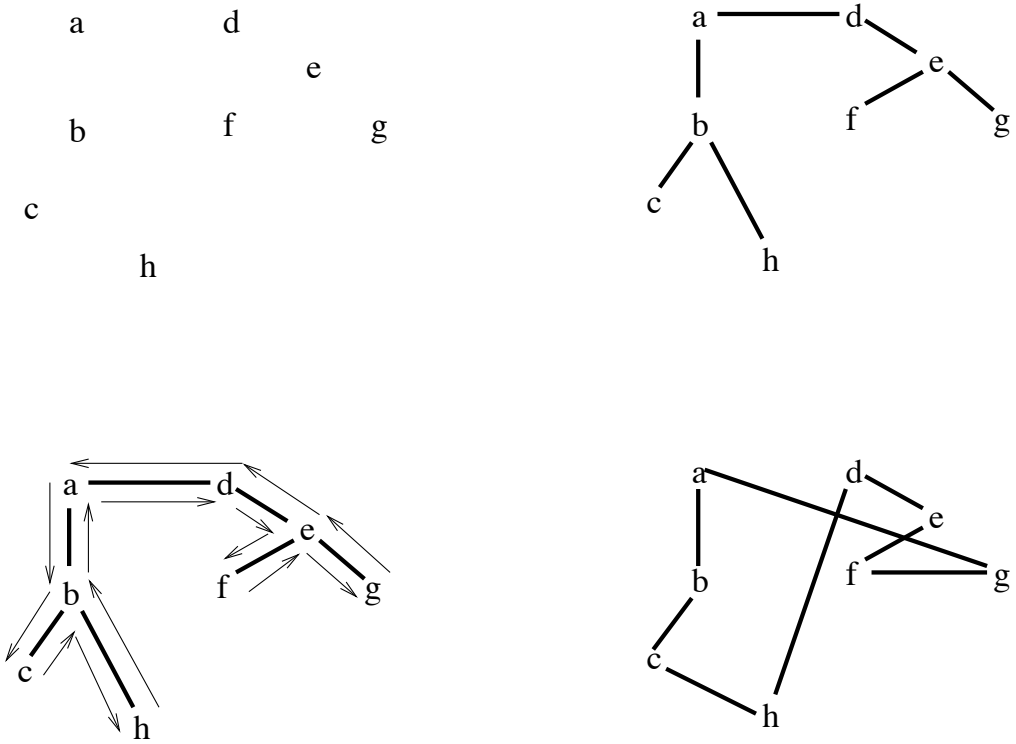
Choose a root $r \in V$

$L \leftarrow$ vertices of $T$ in DFS first-visit order from $r$

$H \leftarrow$ tour visiting $L$ in order and returning to $r$

**return** $H$

# TSP: Example Run




Top: points and MST. Bottom: closed DFS walk and shortcut tour.

$$H = (a, b, c, h, d, e, f, g, a).$$

# TSP: Runtime

- Prim's algorithm with a Fibonacci heap computes the MST in $O(m + n \log n)$ time, where $n = |V|$ and $m = |E|$.
- DFS and recording the first-visit order take $O(n)$ time.
- Forming the tour, including the closing edge, takes $O(n)$ time.
- Thus the total running time is

$$\boxed{O(m + n \log n)}.$$

# TSP: An MST Lower Bound

**Fact:** The MST costs at most an optimal tour:

$$w(T) \leq w(H^*).$$

**Proof.** Remove one edge from the optimal tour $H^*$ to obtain a spanning tree $T'$. Removing an edge of nonnegative weight does not increase cost. By minimality of the MST,

$$w(T) \leq w(T') \leq w(H^*). \qquad \square$$

# TSP: Approximation Guarantee

**Theorem 2:** For metric TSP, the algorithm returns a tour $H$ with $w(H) \leq 2w(H^*)$.

**Proof.** Let $W$ be the closed DFS walk on the MST $T$. Each tree edge is traversed twice, so $w(W) = 2w(T)$.

Shortcut every repeat visit, retaining the first visit to each vertex and the final return to the start. By the triangle inequality, each shortcut costs at most the segment it replaces. The resulting tour is exactly $H$, so $w(H) \leq w(W)$.

Using the MST lower bound,

$$w(H) \leq w(W) = 2w(T) \leq 2w(H^*). \qquad \square$$

# Take Away

- Many real-world problems can be shown to not have an efficient solution unless $P = NP$ (these are the NP-Hard problems)
- However, if a problem is shown to be NP-Hard, all hope is not lost!
- In many cases, we can come up with a provably good approximation algorithm for the NP-Hard problem.