# EE360C: Algorithms

## NP-Completeness

Spring 2018

Department of Electrical and Computer Engineering
University of Texas at Austin

# Introduction

# Algorithm Complexity

## Recap

Up until now, we've talked mainly about polynomial time algorithms. But not all problems can be solved in polynomial time.

## Coming Up

But there's also a class of problems *for which we don't know*. We know we can verify that a solution is correct in polynomial time, but we don't know if we can solve them in polynomial time.

These are the "NP-complete" problems. This has led to the $P \neq NP$ question of computer science lore

- But the kicker is... if you can solve one of the NP-Complete problems in polynomial time, you can solve them all!

# NP Complete Problems

OK. The NP-Complete problems are clearly hard, so they should *look* hard, right?

**Shortest vs. Longest Simple Paths**

We know how to solve the shortest path problem in $O(VE)$ time. Finding the *longest* simple path is much more difficult. In fact, just writing an algorithm to determine if a graph has a path of $k$ edges is an NP-Complete problem.

**Euler tour vs. Hamiltonian tour**

A Euler tour contains every edge in a graph; a Hamiltonian tour contains every vertex in the graph. We can determine whether a graph has a Euler tour in $O(E)$ time; determining whether a graph has a Hamiltonian tour is NP-Complete.

# The Classes P, NP, and NPC

## The Class P

The class $P$ contains problems that are solvable in polynomial time (specifically, in $O(n^k)$, for instance size $n$ and some constant $k$).

## The Class *NP*

The class *NP* contains problems whose solutions can be verified in polynomial time (e.g., given a purported 3-coloring of a graph, we can determine that it is or is not correct using DFS)

A problem is in NPC if it is in NP and is at least as hard as any other problem in NPC (it's unlikely that the problem could be solved in polynomial time  If, as an algorithm designer, you encounter a problem that is NP-complete, it's likely better to focus your attention on an approximation algorithm...

# A Summary

- **P**: problems that can be solved in polynomial time.
- **NP**: problems that can be verified in polynomial time (*nondeterministic polynomial*). Clearly $P \subseteq$ NP.
- **NP-Complete**: decision problems (those whose answers are "yes" or "no") that can be verified in polynomial time, but for which no known polynomial time solution is known. NP-Complete $\subseteq$ NP.
- **NP-Hard**: problems that are at least as hard as the hardest problem in NP
  - A problem $H$ is in NP-Hard iff there exists a problem $L \in$ NP-complete that is polynomial time reducible to $H$.
  - That is, $L$ can be solved in polynomial time given an oracle that can solve $H$.

# Showing a Problem to be NP-Complete

We're turning the tables here a little bit... In showing a problem is NP-complete, we're not trying to design an efficient implementation but to show that one is unlikely to exist

## Decision Problems vs. Optimization Problems

We're going to focus on decision problems (with a yes/no answer) instead of optimization problems

- decision problems are closely related to optimization problems (e.g., given a graph $G$, vertices $u$, $v$, and integer $k$, does there exist a path of length $\leq k$ from $u$ to $v$?)
- the optimization version is at least as hard as the decision version; if the decision problem is hard, it implies that the optimization problem is hard

# Showing a Problem to be NP-Complete (cont.)

To demonstrate a problem is NP-Complete, the key ingredient is a *reduction* that shows that the new problem is at least as hard as a problem known to be NP-complete.

- consider an instance of a decision problem *A* and another decision problem *B* for which we have a polynomial time algorithm
- suppose we have a procedure to transform any instance $\alpha$ of *A* into an instance $\beta$ of *B* such that
  - the transformation takes polynomial time
  - the answer to $\beta$ is yes if and only if the answer to $\alpha$ is yes
- this gives us a polynomial time algorithm for solving *A*

Suppose there ends up not to be a polynomial time algorithm for *A*, and *A* polynomial-time reduces to *B*, then there is no polynomial time procedure for *B*.

# Polynomial-time reducibility

# Polynomial-time reducibility

Intuitively, a problem $Y$ can be reduced to another problem $X$ if any instance of $Y$ can be easily rephrased as an instance of $X$, and the solution to $X$ provides a solution to the instance of $Y$.

- consider the problem $Y$ of solving linear equations (e.g., $ax + b = 0$). We can easily rephrase this as the problem $X$ of solving quadratic equations (e.g., $0x^2 + ax + b = 0$), where if we solve the latter, we have a solution to the former.

## Claim

Problem $X$ is at least as hard as problem $Y$ and write $Y \leq_P X$ (formally, $Y$ is polynomial-time reducible to $X$) if arbitrary instances of problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to a black-box that solves problem $X$.

## Corollary

If $X$, $Y$ are problems such that $Y \leq_p X$, then $X \in P$ implies $Y \in P$.

# Example of Polynomial-time Reducibility: Independent-Set $\leq_P$ Vertex-Cover

**Independent Set**

An **independent set** in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices with no edges between them. The **size** of an independent set is the number of vertices it includes.

The **independent set problem** is the optimization problem of finding an independent set of maximum size in a graph. As a decision problem, we ask if an independent set of a given size at least $k$ exists in the graph.

# Example of Polynomial-time Reducibility: Independent-Set $\leq_P$ Vertex-Cover

**Vertex Cover**

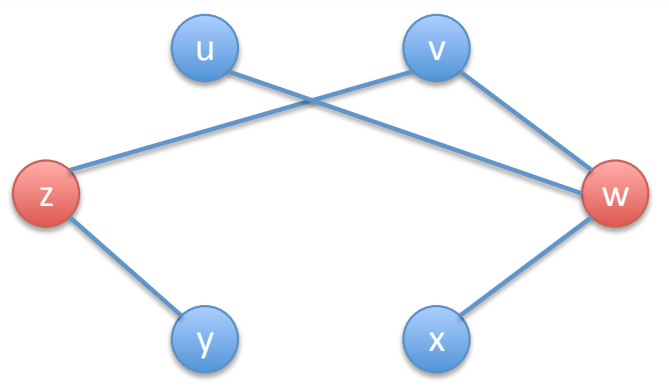
A **vertex-cover** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both). The **size** of a vertex cover is the number of vertices in it.

The vertex-cover problem is to find a vertex cover of minimum size in a given graph. The decision version of this problem is to determine whether a graph has a vertex cover of a given size at most $k$.

# Example of Polynomial-time Reducibility: Independent-Set $\leq_P$ Vertex-Cover

**Proof**

We show that INDEPENDENT-SET$\leq_P$VERTEX-COVER. Note that a graph $G$ has an independent set (red nodes) of size at least $k$ if and only if the graph $G$ has a vertex-cover (blue nodes) of size at most $|V| - k$. Thus, given a black box for deciding whether a graph has a vertex cover of size at most $|V| - k$, we can decide whether a given graph $G$ has an independent set of size at least $k$.



# Showing a Problem to be NP-Complete (cont.)

To show a problem is NP-Complete, we'll reduce a known NP-Complete problem to it.

- We also have to show that it's in NP, but that's usually pretty easy.

We're also going to need a "first" NP-complete problem to start the reductions. We'll see that in a little bit...

# Polynomial-time verification (NP)

# Showing Membership in NP

What did it mean to be a member of NP? That a certificate to a *true instance* (i.e., a "yes" instance) of a problem can be verified in polynomial time.

- For example, consider the problem of determining whether a graph has a path of length $\leq k$.
- Given a purported solution path $p$, we can easily "walk" $p$ and count its edges in polynomial time.

**Hamiltonian Cycle**

A **Hamiltonian cycle** of an undirected graph is a simple cycle that contains every vertex in $V$.

- What's the certificate for the Hamiltonian cycle?
- just the cycle itself!
- it's easy to efficiently verify that it touches all vertices...

# NP or not NP?

## Problems in NP

- anything in $P$ (e.g., does there exist a subsequence of size $k$ for sequences $X$ and $Y$?)
- does there exist a 3-coloring of graph $G$?
- is there a partition of a set of integers $S$ into two subsets $S_0$ and $S_1$ such that the sum of elements in $S_0$ is equal to the sum of elements in $S_1$?

## Problems Unlikely to be in NP

- counting problems
- problems that involve $\forall x.\exists y$
- provably intractable/undecidable problems

Nothing is known about the relationships between P, NP, co-NP, NP-complete, etc.

# NP-Completeness

# NP-Completeness

## Definition

A problem $X$ is **NP-complete** if

1. $X \in NP$, and
2. $Y \leq_P X$ for every problem $Y \in$ NP.

If a problem satisfies the second property but not necessarily the first, then we say it is **NP-hard**.

The following theorem is a direct result:

## Theorem

If any NP-complete problem is polynomial time solvable, then $P = NP$. Equivalently, if any problem in NP-complete is not polynomial time solvable, then no NP-complete problem is polynomial time solvable.

# NP-Completeness

Most people believe that $P \neq NP$ simply because we've known about the NP-complete problems for a long time and no one has come up with an efficient solution for them.

- again, if we can solve any one of the NP-complete problems in polynomial time, we can solve them all in polynomial time
- on the other hand, if we can *prove* that no polynomial time algorithm exists for an NP-complete problem, then no algorithm exists for any of them

# So How Do I Do It?

The following is the general recipe for an NP-complete proof for showing a new problem $X$ is NP-complete:

1. Prove that $X \in$ NP.
2. Select a known NP-complete problem $Y$.
3. Prove that $Y \leq_P X$, namely: consider an arbitrary instance $s_Y$ of problem $Y$, and show how to construct, in polynomial time, an instance $s_X$ of problem $X$ that satisfies the following properties:
   - If $s_Y$ is a "yes" instance of $Y$, then $s_X$ is a "yes" instance of $X$;
   - If $s_X$ is a "yes" instance of $X$, then $s_Y$ is a "yes" instance of $Y$.

   (In other words, the instances $s_X$ and $s_Y$ have the same answer.)

# Our First NP-Complete problem: Circuit-SAT

# That First Problem

Once we have one NP-complete problem, we can reduce others to it. Our first problem is Circuit-Satisfiability, or Circuit-SAT. The proof that Circuit SAT is NP-complete is quite complex; we'll look at the intuition.

# Circuit Satisfiability Problem

A boolean combinational circuit is a collection of boolean combinational elements connected by wires.

- we assume three types of gates: NOT, AND, OR
- no cycles are allowed
- there are some *circuit inputs* and *circuit outputs*

The truth assignment for a circuit is a set of boolean input values.

A one-output boolean combinational circuit is **satisfiable** if it has a **satisfying assignment**: a truth assignment that causes the circuit's output to be 1.

# Circuit Satisfiability Problem

The circuit satisfiability problem is:

- Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?

This would be a nice problem to be able to solve efficiently; if there is a sub-circuit that is not satisfiable (i.e., always generates a 0 output), it can be optimized out and replaced with a “constant” 0

# The Obvious Solution

The obvious way to solve this is to try all of the possible inputs. But this is $O(2^k)$ for $k$ inputs.

Can we do better?

# Circuit SAT is in NP

**Lemma**

The circuit satisfiability problem belongs to the class NP.

**Proof**

Given a certificate (i.e., a purported correct input), we can easily verify its correctness in polynomial time just by running the circuit on the input.

# Circuit SAT is NP-Complete

## Lemma

The circuit satisfiability problem is NP-Complete.

## Proof (Intuition)

To perform this proof, we show that *every* language $L$ in NP-complete can be polynomial time reduced to Circuit-SAT (using the definition of NP-complete). If $L$ is in NP-complete, then there is a verification algorithm $A$ that runs in time $O(n^k)$, assuming strings in $L$ are of length $n$. We construct a single boolean circuit $M$ that maps one "configuration" of a machine that computes $A$ (recording, e.g., memory state, program counter, etc.) to the next "configuration." We hook together $O(n^k)$ of these circuits (because after all, that's the maximum possible number of steps required to compute $A$). This big, composed circuit ($C(x)$) is satisfiable by a certificate $y$ (associated with input $x$) if and only if $x$ was accepted in $L$. By lots more hand-waving, the size of this circuit is polynomial in $n$, and the transformation can be done in polynomial time.

# Examples

# Formula Satisfiability (SAT)

This is actually the first problem ever shown to be NP-complete.

## The Problem

An instance of SAT is a boolean formula $\phi$ composed of:

- $n$ boolean variables: $x_1, x_2, \ldots, x_n$
- $m$ boolean connectives: any boolean function with one or two inputs and one output, such as AND ($\wedge$), OR ($\vee$), NOT ($\neg$), implication ($\rightarrow$), if and only if ($\leftrightarrow$)
- parentheses (we assume there are no redundant parentheses)

A **truth assignment** for $\phi$ is a set of values for the variables; a **satisfying assignment** is one that returns true. A formula with a satisfying assignment is **satisfiable**.

SAT asks whether a formula is satisfiable.

# The Naïve SAT Algorithm

What's the naïve way to solve SAT?

**Easy Solution**

Enumerate all of the possible assignments for the variables and test to see if any of them return true.

**Easy is Bad**

What's the running time? $O(2^n)$ for a formula with $n$ variables.

# SAT is NP-Complete

**Theorem**

Satisfiability of boolean formulas is NP-complete.

**Proof: Part I**

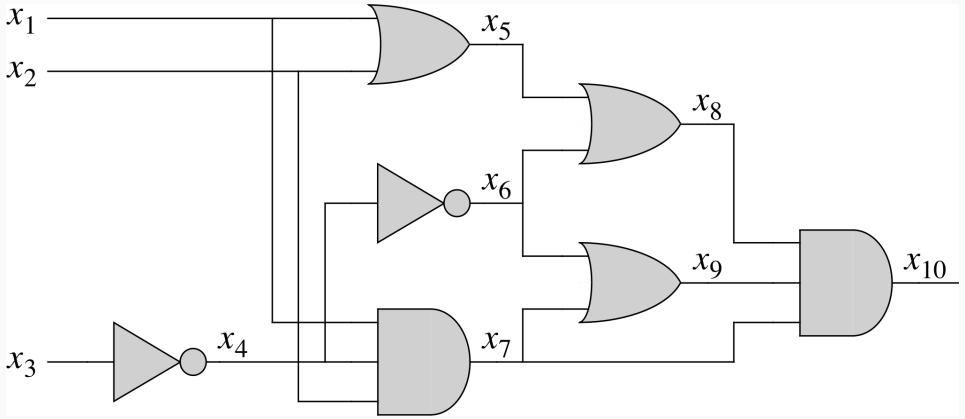
SAT$\in$NP. The certificate consisting of a satisfying assignment for $\phi$ can be verified in polynomial time by replacing each variable with its value in the input and then evaluating the expression.

# SAT is NP-Complete (cont.)

## Proof: Part II

We show that CIRCUIT-SAT$\leq_P$SAT, i.e., that any instance of circuit satisfiability can be reduced in polynomial time to an instance of SAT. We might think we can just replace any boolean circuit as a boolean formula. Unfortunately, when gates have large fanouts, the formula can grow exponentially. Instead, in addition to having variables from inputs of each the circuit, we create new variables, one per gate in the circuit. We encode the functioning of each gate by a small formula, e.g., an AND gate with inputs $x$ and $y$ and output variable $w$ is encoded as $(w \leftrightarrow (x \wedge y))$. Then we take the conjunction of all of these formulas and in addition $z$, where $z$ is the output of the circuit. The resulting formula is satisfiable if and only if the circuit is satisfiable. If the circuit has a satisfying assignment, each wire of the circuit has a well-defined value, and the circuit's output is 1. Therefore the assignment of wire values to variables in $\phi$ makes each clause of $\phi$ evaluate to 1, and thus the conjunction evaluates to 1.

# SAT Reduction Example




$$
\begin{aligned}
\Phi = x_{10} \quad & \wedge \quad (x_4 \leftrightarrow \neg x_3) \\
& \wedge \quad (x_5 \leftrightarrow (x_1 \vee x_2)) \\
& \wedge \quad (x_6 \leftrightarrow \neg x_4) \\
& \wedge \quad (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
& \wedge \quad (x_8 \leftrightarrow (x_5 \vee x_6)) \\
& \wedge \quad (x_9 \leftrightarrow (x_6 \vee x_7)) \\
& \wedge \quad (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))
\end{aligned}
$$

# 3-SAT

Reducing from SAT can be painful because the reduction has to handle any input formula. So it's useful to reduce from a more restricted language. That's what 3-SAT is used for.

A **literal** in a boolean formula is an occurrence of a variable or its negation. A formula is in **conjunctive normal form** if it expressed as an AND of clauses, each of which is an OR of one or more literals.

**3-SAT**

A boolean formula is an instance of **3-SAT** if each clause has exactly three distinct literals. An example:

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

# 3-SAT is NP-Complete

**Theorem**

3-SAT is NP-Complete.

**Proof: Part I**

The same argument that SAT$\in$NP can be given as was given for SAT$\in$NP (i.e., 3-SAT formulas are just special cases of formulas)

# 3-SAT is NP-Complete (cont.)

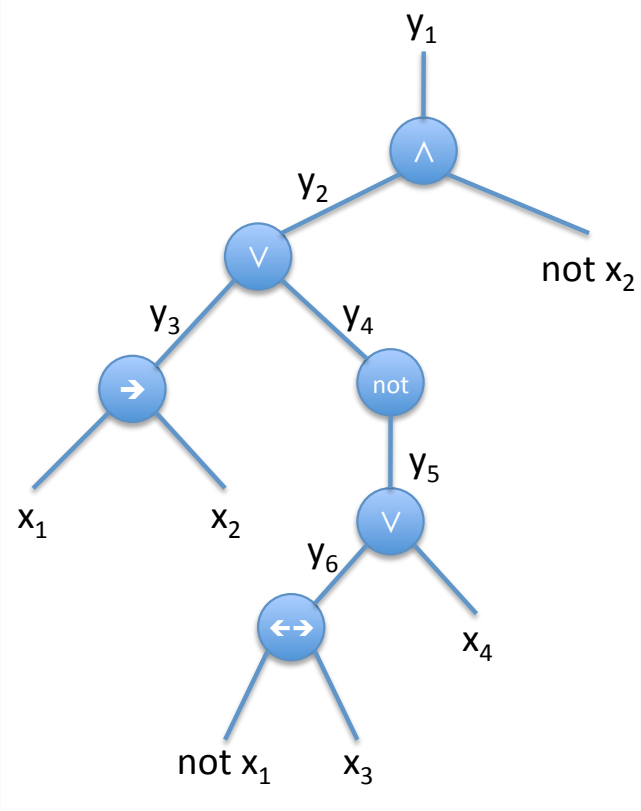
**Proof: Part II**

First, given a formula $\phi$, construct a "parse" tree for $\phi$ with literals as leaves and connectives as internal nodes. This parse tree is basically a circuit built directly from $\phi$. A node in the tree can have at most 2 inputs; we can adjust $\phi$ slightly to account for this by adding parenthesizations. We introduce a variable $y_i$ for the output of each node.

# 3-SAT is NP-Complete (cont.)

For example, the tree for the formula:

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

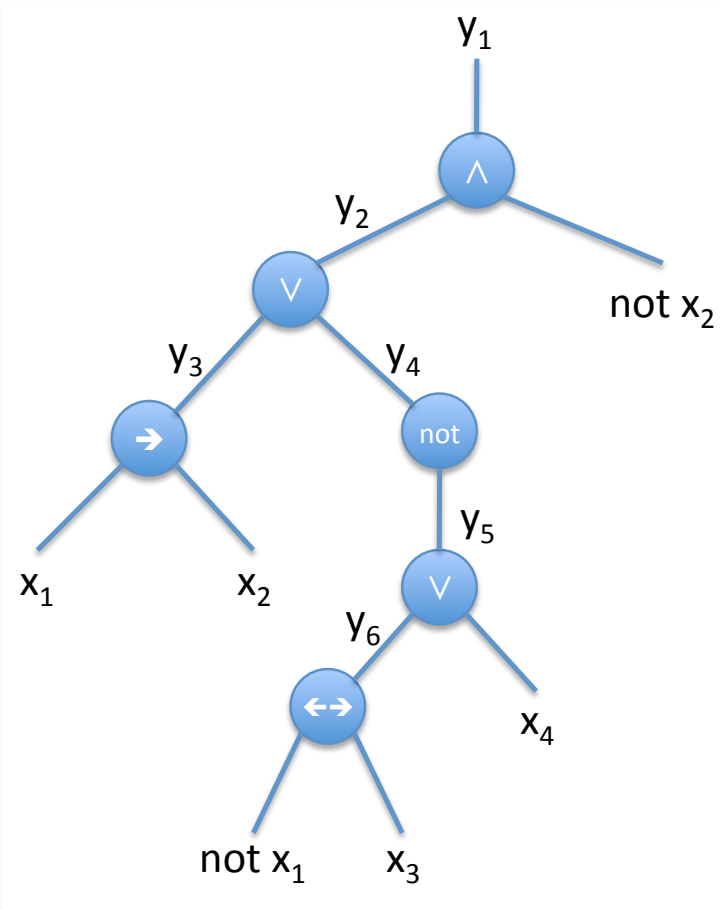


# 3-SAT is NP-Complete (cont.)

**Proof: Part II**

We rewrite the original formula $\phi$ as the AND of the root variable and the conjunction of clauses describing the operation at each node. Call this new formula $\phi'$. $\phi'$ is satisfiable if and only if $\phi$ is satisfiable. Each clause in $\phi'$ has no more than 3 literals.

# 3-SAT is NP-Complete (cont.)




$$\begin{aligned}
\phi' = y_1 \quad & \wedge \quad (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\
& \wedge \quad (y_2 \leftrightarrow (y_3 \vee y_4)) \\
& \wedge \quad (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\
& \wedge \quad (y_4 \leftrightarrow \neg y_5) \\
& \wedge \quad (y_5 \leftrightarrow (y_6 \vee x_4)) \\
& \wedge \quad (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))
\end{aligned}$$

# 3-SAT is NP-Complete (cont.)

- We need to convert our $\phi'$ to 3-SAT form.
- We first create a truth table for each clause in $\phi'$.
- We use the truth table to build a formula in *disjunctive normal form* that is equivalent to $\neg\phi'_i$ (i.e., we just OR all of the 0 values from the truth table).
- Then we can just easily apply DeMorgan's laws to convert $\neg\phi'_i$ to $\phi''_i$.
- The resulting clauses each have at most 2 literals. If they only have two, create a new dummy variable, $p$, and form two clauses, one with $p$, and one with $\neg p$.
- This final formula is satisfiable if and only if the original $\phi$ is satisfiable.
- Oh, and we can compute this reduction in polynomial time, which is the last part of the proof.

# 3-SAT Example

**Example**

Consider $\phi_1' = (y_1 \leftrightarrow (y_2 \wedge \neg x_2)$. Construct the truth table (the value of the clause for all eight combinations of $y_1$, $y_2$, and $x_2$.

Then the DNF for $\neg\phi_1'$ is

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

We apply DeMorgan's Laws, which complements all of the literals and converts all OR's into AND's and all AND's into OR's. Then $\phi_1''$ is:

$$(\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

Which is in 3-SAT form.

# The Clique Problem

**The Problem**

A **clique** in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in $E$. The **size** of a clique is the number of vertices it includes.

The **clique problem** is the optimization problem of finding a clique of maximum size in a graph. As a decision problem, we ask if a clique of a given size $k$ exists in the graph.

# Clique is NP-Complete

**Theorem**

The clique problem is NP-complete.

**Proof: Part I**

Demonstrating CLIQUE$\in$NP is simple; use $V'$ as a certificate and just check to see that every member of $V'$ is connected to every other member.

# Clique is NP-Complete (cont.)

## Proof: Part II

Next show 3-SAT$\leq_P$CLIQUE. Let $\phi = C_1 \wedge C_2 \wedge \cdots C_k$ be a boolean formula in 3-SAT form and $k$ clauses. We construct a graph $G = (V, E)$ such that for each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$, we add vertices $v_1^r$, $v_2^r$, and $v_3^r$ in $V$. We put an edge between $v_i^r$ and $v_j^s$ if $r \neq s$ and $l_i^r$ is not the negation of $l_j^s$. We can easily do this transformation in polynomial time.

## $G_\phi$ has a $k$-clique if and only if $\phi$ is satisfiable.

- SAT$\Rightarrow$ $k$-CLIQUE: Take any satisfying assignment for $\phi$. Then each clause $C_r$ must have some literal $l_i^r$ that is assigned 1, and each one of these corresponds to a vertex. Picking the true literal from each of the $k$ clauses yields a clique of size $k$.
- $k$-CLIQUE$\Rightarrow$SAT: Suppose that $G$ has a clique $V'$ of size $k$. No edges in $G$ connect vertices in the same triple, and so $V'$ contains exactly one edge per triple. We can assign a 1 to each of the literals corresponding to each of these vertices, generating a satisfying input for $\phi$.

# The Vertex-Cover Problem

**The Problem**

A **vertex-cover** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both). The **size** of a vertex cover is the number of vertices in it.

The vertex-cover problem is to find a vertex cover of minimum size in a given graph. The decision version of this problem is to determine whether a graph has a vertex cover of a given size $k$.

# Vertex-Cover is NP-Complete

**Theorem**

The vertex-cover problem is NP-Complete

**Proof: Part I**

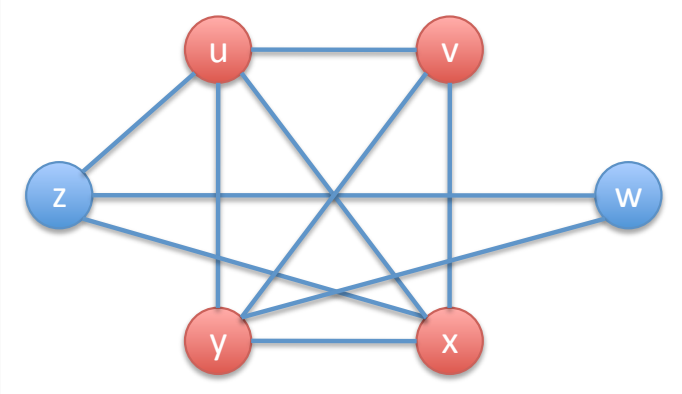
VERTEX-COVER$\in$NP. Given a certificate (i.e., a purported vertex cover), we can step through the list of vertices to ensure they're all covered by the solution.

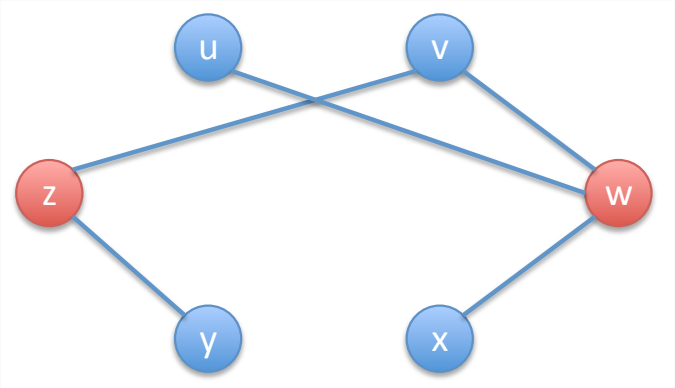
# Vertex-Cover is NP-Complete

**Proof: Part II**

We show that VERTEX-COVER is NP-Hard by showing CLIQUE$\leq_P$VERTEX-COVER. We compute the complement of the graph (i.e., the graph that contains all of the $(u, v)$ that are not in $E$.) In this reduction, the graph $G$ has a clique of size $k$ if and only if the graph $\overline{G}$ has a vertex-cover of size $|V| - k$.

# The Relationship of Clique and Vertex-Cover



# The Hamiltonian-Cycle Problem

**The Problem**

Remember the Hamiltonian Cycle problem: a Hamiltonian cycle of a graph $G = (V, E)$ is a simple cycle that contains every vertex in $V$.

The decision form of the Hamiltonian Cycle problem is the one of asking whether a Hamiltonian Cycle exists.

# HAM-CYCLE is NP-Complete

**Theorem 34.13**

The Hamiltonian Cycle problem is NP-Complete.

**Proof: Part I**

HAM-CYCLE$\in$NP. The certificate for a HAM-CYCLE is the sequence of vertices of length $|V|$ that defines the cycle. The verification just ensures that the all of the corresponding edges exist in the graph. This can easily be done in polynomial time.

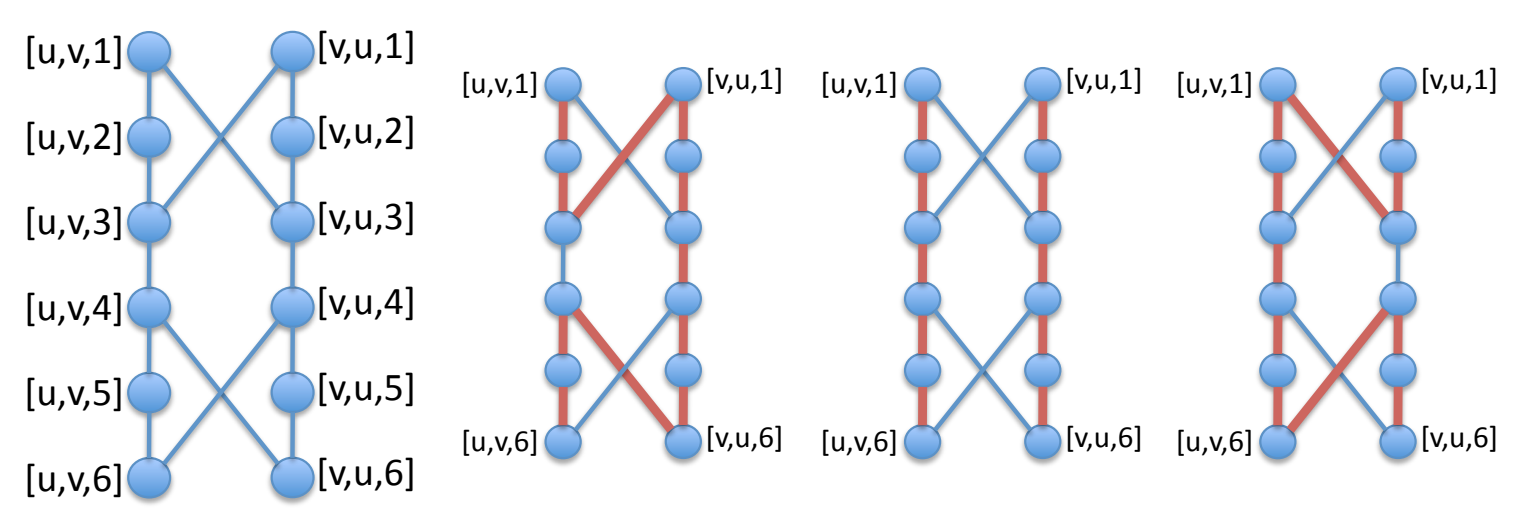
# HAM-CYCLE is NP-Complete

**Proof: Part II**

VERTEX-COVER$\leq_P$HAM-CYCLE. That is, given an undirected graph $G = (V, E)$ and an integer $k$, we construct an undirected graph $G' = (V', E')$ that has a Hamiltonian cycle if and only if $G$ has a vertex cover of size $k$. For every edge $(u, v) \in E$, we create a widget $W_{uv}$. $W_{uv}$ includes 12 vertices and 14 edges, such that there is a vertex $[u, v, i]$ and $[v, u, i]$ for $1 \leq i \leq 6$ in $W_{uv}$.

# HAM-CYCLE is NP-Complete (cont.)

For each edge $(u, v)$ in $G$, we have a widget $W_{uv}$ consisting of 12 vertices and 14 edges (that has the shown Hamiltonian Paths through it)



# HAM-CYCLE is NP-Complete (cont.)

**Proof: Part II (cont.)**

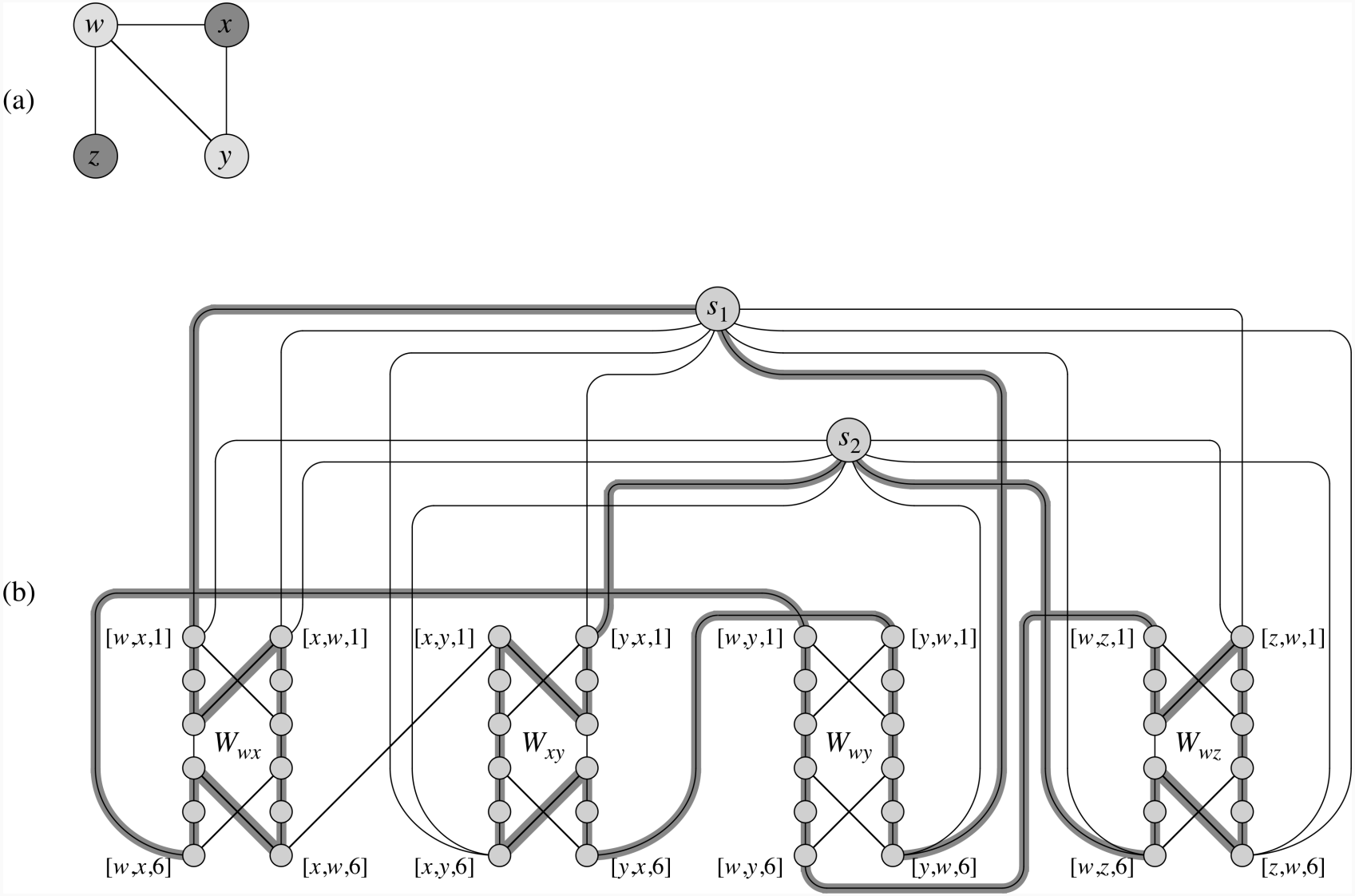
Only vertices $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$, and $[v, u, 6]$ can have edges in $E'$ that exit $W_{uv}$. Any Hamiltonian cycle of $G'$ will have to traverse the edges in $W_{uv}$ in one of the three ways shown previously. In addition to the vertices in all of the widgets, $V'$ also contains $k$ selector vertices $s_1, s_2, \ldots, s_k$. For each vertex $u \in V$, we string together the widgets for all edges incident to $u$ into a single *vertex chain* and connect the ends of the chain to the selector vertices. Specifically, suppose $u$ has $d$ neighbors $v_1, v_2, \ldots v_d$. Then $G'$ has the following edges: 1) $d - 1$ edges between $(u, v_i, 6)$ and $(u, v_{i+1}, 1)$ for all $i$ between 1 and $d - 1$; 2) $k$ edges between the selector vertices and $(u, v_1, 1)$; and 3) $k$ edges between the selector vertices and $(u, v_d, 6)$.

# HAM-CYCLE is NP-Complete (cont.)

**Proof: Part II (cont.)**

Now, if $\{v_1, v_2, \ldots, v_k\}$ is a vertex cover of $G$, then $G'$ has a Hamiltonian cycle. To list out the Hamiltonian cycle, we start at the first selector vertex ($s_1$) and traverse the vertex chain for $v_1$ (i.e., all of the widgets for the edges incident to $v_1$, then visit the second selector vertex ($s_2$), etc. In the other direction, if any Hamiltonian cycle in $G'$ alternates between cover vertices and vertex chains, then the vertex chains correspond to the $k$ vertices in a vertex cover of $G$.

# HAM-CYCLE is NP-Complete (an example)



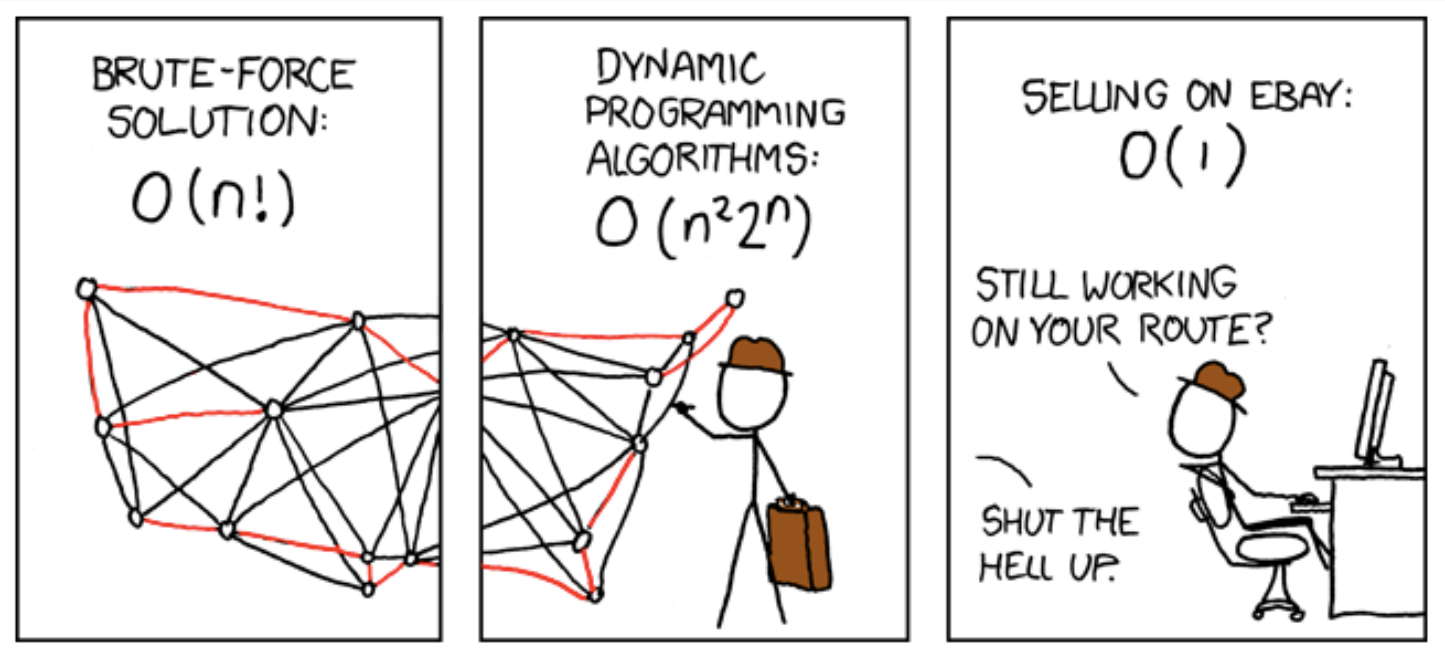
# The Traveling-Salesman Problem

**The Problem**

In the traveling salesman problem, a salesman wishes to make a **tour**, visiting each city exactly once and finishing at the starting city. There is an integer cost $C(i, j)$ to travel between $i$ and $j$, and the salesman wishes to make the least cost tour.

The traveling salesman problem asks whether a tour of cost at most $k$ exists.

# TSP is NP-Complete



# TSP is NP-Complete

**Theorem**

The traveling salesman problem is NP-complete.

**Proof: Part I**

TSP$\in$NP. Given a certificate, we can check that the purported solution goes to all of the vertices and has cost not more than $k$ but just traversing the edges.

# TSP is NP-Complete (cont.)

**Proof: Part II**

HAM-CYCLE$\leq_P$TSP. Let $G = (V, E)$ be an instance of HAM-CYCLE. We construct an instance of TSP as follows. Form the complete graph $G' = (V, E')$, where $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$, and we define the cost function by:

$$c(i,j) = \begin{cases} 0 & \text{if } (i,j) \in E \\ 1 & \text{if } (i,j) \notin E \end{cases}$$

Then $G$ has a Hamiltonian cycle if and only if $G'$ has a tour of cost at most 0.

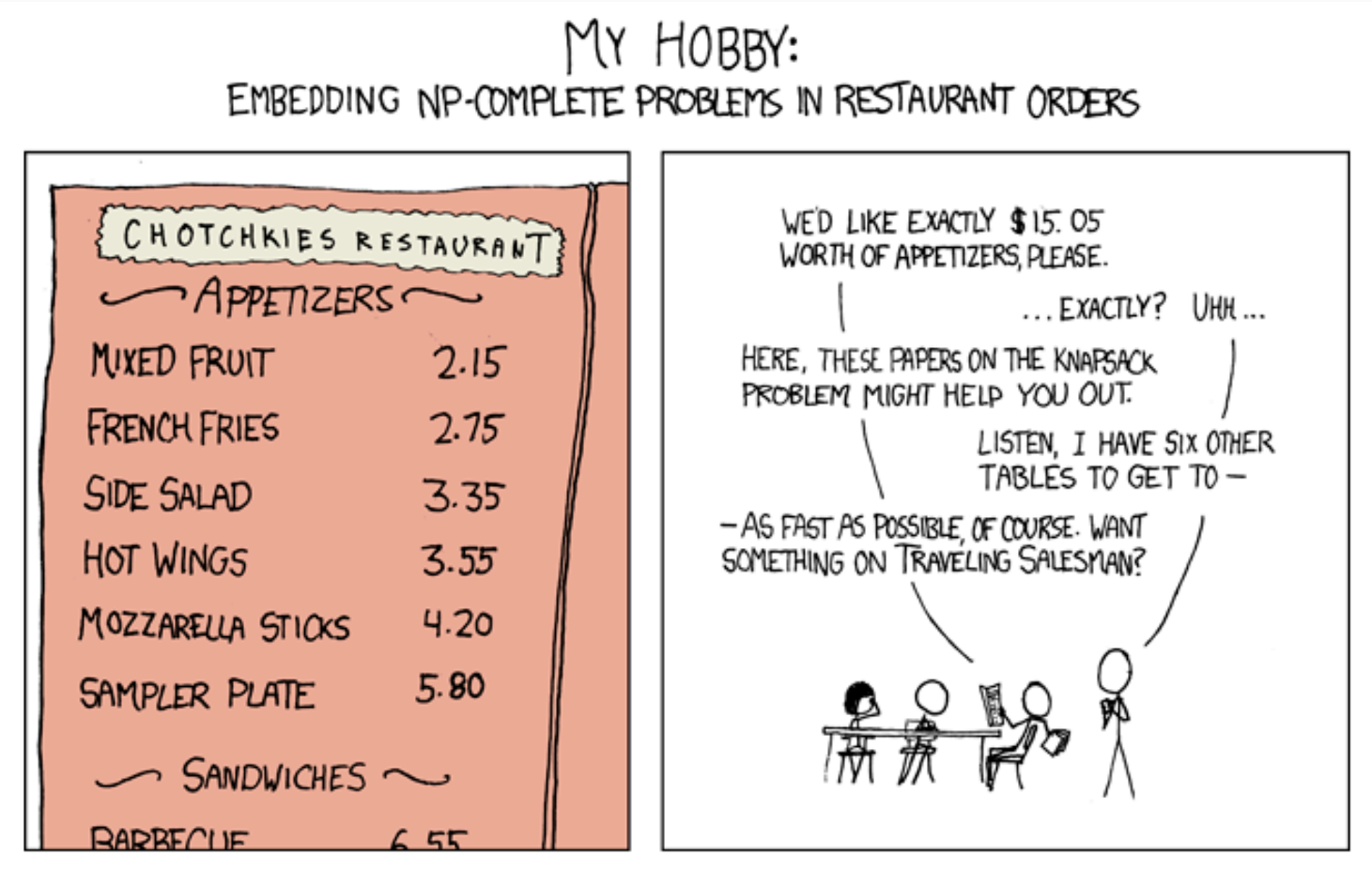
# The Subset-Sum Problem

**The Problem**

In the **subset sum problem**, we have a finite set $S \subset N$ and a **target** $W \in N$. We ask whether there is a subset $S' \subseteq S$ whose elements sum to $W$.

# Subset-Sum is NP-Complete

(And remember that 0-1 Knapsack problem? It is too!)



# Subset-Sum is NP-Complete

**Theorem**

The subset sum problem is NP-Complete

**Proof: Part I**

Subset-Sum$\in$ NP. Given a certificate of the subset sum problem (i.e., a subset of the given numbers), we can check that all of the numbers were in the original set and that they sum to $W$, and we can do these checks in polynomial time.

# Subset-Sum is NP-Complete (cont.)

**Proof: Part II**

We show that 3-Dimensional Matching $\leq_p$ Subset-Sum.

**Wait, what's 3-D Matching?**

**The 3-Dimensional Matching Problem.** *Given disjoint sets $X$, $Y$, and $Z$, each of size $n$, and given a set $T \subseteq X \times Y \times Z$ of ordered triples, does there exist a set of $n$ triples in $T$ so that each element of $X \cup Y \cup Z$ is contained in exactly one of these triples?*

# Subset-Sum is NP-Complete (cont.)

## Proof: Part II

We show that 3-Dimensional Matching $\leq_p$ Subset-Sum. Consider an instance of 3-D Matching specified by sets $X, Y, Z$, each of size $n$, and a set of $m$ triples $T \subseteq X \times Y \times Z$. For this problem, we represent each triple $t = (x_i, y_j, z_k) \in T$ by constructing a number $w_t$ with $3n$ digits that has a 1 in positions $i$, $n + j$, and $2n + k$. That is, for some base $d$, $w_t = d^{i-1} + d^{n+j-1} + d^{2n+k-1}$. We assume $d = m + 1$ (to ensure no "carries" when we "add" these numbers). We define $W$ to be the number in base $m + 1$ with $3n$ digits, where every digit is a 1. **The set $T$ of triples contains a perfect 3D matching iff there is a subset of the numbers $\{w_t\}$ that adds up to $W$.** Suppose there is a 3D matching consisting of triples $t_1, \ldots, t_n$. Then in the sum $w_{t_1} + \cdots + w_{t_n}$, there is a single 1 in each of the $3n$ digit positions, so the result is equal to $W$. Conversely, suppose there exists a set of numbers $w_{t_1}, \ldots, w_{t_k}$ that adds up to $W$. Then since each $w_{t_i}$ has three ones in it (by construction) and there are no carries, then $k = n$. Then for each of the $3n$ digits positions, exactly one of the $w_{t_i}$ has a 1 in that position and $t_1, \ldots, t_k$ constitute a perfect 3D matching.

# A Recap

The takeaways from all of this are:

- there is a class of (decision) problems that are all equivalently unlikely to be efficiently solvable (“tractable”)
- it’s important to be familiar with this class of problems since pounding your head against one of them is likley to be a waste of time
- your time would be better spent on developing an approximation algorithm

To prove a problem NP-Complete, we have to show:

- that it can be verified in polynomial time
- that it is NP-hard, i.e., that some other known NP-Hard problem is polynomial-time reducible to the target problem

Choosing the right problem to start from can make a really big difference in the complexity of the proof; there’s no obvious guidance except to use experience and err towards a problem that “looks” similar.

# Questions