NP Completeness and Cook-Levin Theorem

Lecture 22
April 19, 2011
**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

**Question**: What is an algorithm? Depends on the model of computation!

What is our model of computation?

Formally speaking our model of computation is Turing Machines.
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

**Question**: What is an algorithm? Depends on the model of computation!

What is our model of computation?

Formally speaking our model of computation is Turing Machines.
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

**Question**: What is an algorithm? Depends on the model of computation!

What is our model of computation?

Formally speaking our model of computation is Turing Machines.
**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

**Question**: What is an algorithm? Depends on the model of computation!

What is our model of computation?

Formally speaking our model of computation is Turing Machines.
**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

**Question**: What is an algorithm? Depends on the model of computation!

What is our model of computation?

Formally speaking our model of computation is Turing Machines.
Turing Machines: Recap

- Infinite tape
- Finite state control
- Input at beginning of tape
- Special tape letter “blank” □
- Head can move only one cell to left or right
Turing Machines: Formally

A Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$:

- $Q$ is set of states in finite control
- $q_0$ start state, $q_{\text{accept}}$ is accept state, $q_{\text{reject}}$ is reject state
- $\Sigma$ is input alphabet, $\Gamma$ is tape alphabet (includes $\square$)
- $\delta : Q \times \Gamma \rightarrow \{L, R\} \times \Gamma \times Q$ is transition function
  - $\delta(q, a) = (q', b, L)$ means that $M$ in state $q$ and head seeing $a$ on tape will move to state $q'$ while replacing $a$ on tape with $b$ and head moves left.

$L(M)$: language accepted by $M$ is set of all input strings $s$ on which $M$ accepts; that is:

- $TM$ is started in state $q_0$.
- Initially, the tape head is located at the first cell.
- The tape contain $s$ on the tape followed by blanks.
- The $TM$ halts in the state $q_{\text{accept}}$. 
**Definition**

M is a polynomial time TM if there is some polynomial \( p(\cdot) \) such that on all inputs \( w \), M halts in \( p(|w|) \) steps.

**Definition**

L is a language in P iff there is a polynomial time TM M such that \( L = L(M) \).
Definition

$L$ is an **NP** language iff there is a *non-deterministic* polynomial time **TM** $M$ such that $L = L(M)$.

Non-deterministic **TM**: each step has a choice of moves

- $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$.
  - Example: $\delta(q, a) = \{(q_1, b, L), (q_2, c, R), (q_3, a, R)\}$ means that $M$ can non-deterministically choose one of the three possible moves from $(q, a)$.

- $L(M)$: set of all strings $s$ on which there exists some sequence of valid choices at each step that lead from $q_0$ to $q_{\text{accept}}$
**NP via TMs**

**Definition**

$L$ is an NP language iff there is a *non-deterministic* polynomial time TM $M$ such that $L = L(M)$.

Non-deterministic TM: each step has a choice of moves

- $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$.

  Example: $\delta(q, a) = \{(q_1, b, L), (q_2, c, R), (q_3, a, R)\}$ means that $M$ can non-deterministically choose one of the three possible moves from $(q, a)$.

- $L(M)$: set of all strings $s$ on which there exists some sequence of valid choices at each step that lead from $q_0$ to $q_{accept}$.
Two definition of \textbf{NP}:

- $L$ is in \textbf{NP} iff $L$ has a polynomial time certifier $C(\cdot, \cdot)$.
- $L$ is in \textbf{NP} iff $L$ is decided by a non-deterministic polynomial time \textbf{TM} $M$.

\textbf{Claim}: Two definitions are equivalent. Why?

Informal proof idea: the certificate $t$ for $C$ corresponds to non-deterministic choices of $M$ and vice-versa. In other words $L$ is in \textbf{NP} iff $L$ is accepted by a \textbf{NTM} which first guesses a proof $t$ of length poly in input $|s|$ and then acts as a deterministic \textbf{TM}.
Non-deterministic TMs vs certifiers

Two definition of NP:

- \( L \) is in \( NP \) iff \( L \) has a polynomial time certifier \( C(\cdot, \cdot) \).
- \( L \) is in \( NP \) iff \( L \) is decided by a non-deterministic polynomial time TM \( M \).

Claim: Two definitions are equivalent. Why?

Informal proof idea: the certificate \( t \) for \( C \) corresponds to non-deterministic choices of \( M \) and vice-versa. In other words \( L \) is in \( NP \) iff \( L \) is accepted by a NTM which first guesses a proof \( t \) of length poly in input \( |s| \) and then acts as a deterministic TM.
A non-deterministic machine has choices at each step and accepts a string if there exists a set of choices which lead to a final state.

Equivalently the choices can be thought of as guessing a solution and then verifying that solution. In this view all the choices are made a priori and hence the verification can be deterministic. The “guess” is the “proof” and the “verifier” is the “certifier”.

We reemphasize the asymmetry inherent in the definition of non-determinism. Strings in the language can be easily verified. No easy way to verify that a string is not in the language.
Why do we use TMs sometimes and RAM Model other times?

- **TMs** are very simple: no complicated instruction set, no jumps/pointers, no explicit loops etc.
  - Simplicity is useful in proofs.
  - The “right” formal bare-bones model when dealing with subtleties.

- **RAM** model is a closer approximation to the running time/space usage of realistic computers for reasonable problem sizes
  - Not appropriate for certain kinds of formal proofs when algorithms can take super-polynomial time and space
“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
1. $X \in \text{NP}$
2. (Hardness) For any $Y \in \text{NP}$, $Y \leq_p X$
Proposition

Suppose $X$ is NP-Complete. Then $X$ can be solved in polynomial time iff $P = NP$

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 $X$
NP-Hard Problems

**Definition**

A problem $X$ is said to be **NP-Hard** if

- *(Hardness)* For any $Y \in \text{NP}$, $Y \leq_p X$

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 $X$. 

Question
Are there any problems that are NP-Complete?

Answer
Yes! Many, many problems are NP-Complete.
A circuit is a directed *acyclic* graph with

- **Input** vertices (without incoming edges) labelled with 0, 1 or a distinct variable
- Every other vertex is labelled ∨, ∧ or ¬
- Single node output vertex with no outgoing edges
Circuits

Definition

A circuit is a directed *acyclic* graph with

- **Input** vertices (without incoming edges) labelled with 0, 1 or a distinct variable
- Every other vertex is labelled $\lor$, $\land$ or $\neg$
- Single node output vertex with no outgoing edges
A circuit is a directed *acyclic* graph with

- **Input** vertices (without incoming edges) labelled with 0, 1 or a distinct variable
- Every other vertex is labelled ∨, ∧ or ¬
- Single node **output** vertex with no outgoing edges
Cook-Levin Theorem

Definition (Circuit Satisfaction (CSAT).)

Given a circuit as input, is there an assignment to the input variables that causes the output to get value 1?

Theorem (Cook-Levin)

CSAT is NP-Complete.

Need to show

- CSAT is in NP
- every NP problem X reduces to CSAT.
Cook-Levin Theorem

Definition (Circuit Satisfaction (CSAT).)
Given a circuit as input, is there an assignment to the input variables that causes the output to get value \(1\)?

Theorem (Cook-Levin)

\textbf{CSAT is NP-Complete.}

Need to show

- \textbf{CSAT} is in \textbf{NP}
- every \textbf{NP} problem \textbf{X} reduces to \textbf{CSAT}. 
Cook-Levin Theorem

Definition (Circuit Satisfaction (CSAT).)
Given a circuit as input, is there an assignment to the input variables that causes the output to get value 1?

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

Need to show
- CSAT is in NP
- every NP problem X reduces to CSAT.
Claim

**CSAT** \textit{is in NP}.

- **Certificate**: assignment to input variables
- **Certifier**: evaluate the value of each gate in a topological sort of DAG and check the output gate value
Claim

**CSAT** is in **NP**.

- **Certificate**: assignment to input variables
- **Certifier**: evaluate the value of each gate in a topological sort of DAG and check the output gate value
CSAT is NP-hard: Idea

Need to show that every NP problem $X$ reduces to CSAT.

What does it mean that $X \in \text{NP}$?

$X \in \text{NP}$ implies that there are polynomials $p()$ and $q()$ and certifier/verifier program $C$ such that for every string $s$ the following is true:

- If $s$ is a YES instance ($s \in X$) then there is a proof $t$ of length $p(|s|)$ such that $C(s, t)$ says YES.
- If $s$ is a NO instance ($s \not\in X$) then for every string $t$ of length at $p(|s|)$, $C(s, t)$ says NO.
- $C(s, t)$ runs in time $q(|s| + |t|)$ time (hence polynomial time)
CSAT is NP-hard: Idea

Need to show that every NP problem X reduces to CSAT.

What does it mean that X ∈ NP?

X ∈ NP implies that there are polynomials p() and q() and certifier/verifier program C such that for every string s the following is true:

- If s is a YES instance (s ∈ X) then there is a proof t of length p(|s|) such that C(s, t) says YES.
- If s is a NO instance (s ∉ X) then for every string t of length at p(|s|), C(s, t) says NO.
- C(s, t) runs in time q(|s| + |t|) time (hence polynomial time)
CSAT is $\textbf{NP}$-hard: Idea

Need to show that every $\textbf{NP}$ problem $X$ reduces to $\text{CSAT}$.

What does it mean that $X \in \textbf{NP}$?

$X \in \textbf{NP}$ implies that there are polynomials $p()$ and $q()$ and certifier/verifier program $C$ such that for every string $s$ the following is true:

- If $s$ is a YES instance ($s \in X$) then there is a proof $t$ of length $p(|s|)$ such that $C(s, t)$ says YES.
- If $s$ is a NO instance ($s \not\in X$) then for every string $t$ of length at $p(|s|)$, $C(s, t)$ says NO.
- $C(s, t)$ runs in time $q(|s| + |t|)$ time (hence polynomial time)
CSAT is **NP**-hard: Idea

Need to show that every **NP** problem $X$ reduces to **CSAT**.

What does it mean that $X \in \textbf{NP}$?

$X \in \textbf{NP}$ implies that there are polynomials $p()$ and $q()$ and certifier/verifier program $C$ such that for every string $s$ the following is true:

- If $s$ is a YES instance ($s \in X$) then there is a proof $t$ of length $p(|s|)$ such that $C(s, t)$ says YES.
- If $s$ is a NO instance ($s \not\in X$) then for every string $t$ of length at $p(|s|)$, $C(s, t)$ says NO.
- $C(s, t)$ runs in time $q(|s| + |t|)$ time (hence polynomial time)
Reducing $X$ to CSAT

$X$ is in $\textbf{NP}$ means we have access to $p(), q(), C(\cdot, \cdot)$. What is $C(\cdot, \cdot)$? It is a program or equivalently a Turing Machine! How are $p()$ and $q()$ given? As numbers. Example: if 3 is given then $p(n) = n^3$.

Thus an $\textbf{NP}$ problem is essentially a three tuple $< p, q, C >$ where $C$ is either a program or a $\textbf{TM}$. 
Reducing $X$ to CSAT

$X$ is in **NP** means we have access to $p(), q(), C(\cdot, \cdot)$.

What is $C(\cdot, \cdot)$? It is a program or equivalently a Turing Machine!

How are $p()$ and $q()$ given? As numbers.

Example: if 3 is given then $p(n) = n^3$.

Thus an **NP** problem is essentially a three tuple $< p, q, C >$ where $C$ is either a program or a **TM**.
Reducing $\Sigma$ to CSAT

$\Sigma$ is in NP means we have access to $p(), q(), C(\cdot, \cdot)$. What is $C(\cdot, \cdot)$? It is a program or equivalently a Turing Machine! How are $p()$ and $q()$ given? As numbers.

Example: if 3 is given then $p(n) = n^3$.

Thus an NP problem is essentially a three tuple $< p, q, C >$ where $C$ is either a program or a TM.
Reducing $X$ to CSAT

$X$ is in $\text{NP}$ means we have access to $p(), q(), C(\cdot, \cdot)$.

What is $C(\cdot, \cdot)$? It is a program or equivalently a Turing Machine!

How are $p()$ and $q()$ given? As numbers.

Example: if 3 is given then $p(n) = n^3$.

Thus an $\text{NP}$ problem is essentially a three tuple $\langle p, q, C \rangle$ where $C$ is either a program or a TM.
Reducing $X$ to $\text{CSAT}$

Thus an $\textbf{NP}$ problem is essentially a three tuple $< p, q, C >$ where $C$ is either a program or $\text{TM}$.

**Problem X:** Given string $s$, is $s \in X$?

Same as the following: is there a proof $t$ of length $p(|s|)$ such that $C(s, t)$ says YES.

How do we reduce $X$ to $\text{CSAT}$? Need an algorithm $A$ that

- takes $s$ (and $< p, q, C >$) and creates a circuit $G$ in polynomial time in $|s|$ (note that $< p, q, C >$ are fixed).
- $G$ is satisfiable if and only if there is a proof $t$ such that $C(s, t)$ says YES.
Reducing $X$ to CSAT

Thus an NP problem is essentially a three tuple $< p, q, C >$ where $C$ is either a program or TM.

Problem X: Given string $s$, is $s \in X$?

Same as the following: is there a proof $t$ of length $p(|s|)$ such that $C(s, t)$ says YES.

How do we reduce $X$ to CSAT? Need an algorithm $A$ that

- takes $s$ (and $< p, q, C >$) and creates a circuit $G$ in polynomial time in $|s|$ (note that $< p, q, C >$ are fixed).
- $G$ is satisfiable if and only if there is a proof $t$ such that $C(s, t)$ says YES.
Reducing $X$ to CSAT

Thus an NP problem is essentially a three tuple $< p, q, C >$ where $C$ is either a program or TM.

Problem X: Given string $s$, is $s \in X$?

Same as the following: is there a proof $t$ of length $p(|s|)$ such that $C(s, t)$ says YES.

How do we reduce $X$ to CSAT? Need an algorithm $A$ that

- takes $s$ (and $< p, q, C >$) and creates a circuit $G$ in polynomial time in $|s|$ (note that $< p, q, C >$ are fixed).
- $G$ is satisfiable if and only if there is a proof $t$ such that $C(s, t)$ says YES.
Reducing \( X \) to \textbf{CSAT}

Thus an \textbf{NP} problem is essentially a three tuple \( < p, q, C > \) where \( C \) is either a program or \textbf{TM}.

**Problem X:** Given string \( s \), is \( s \in X \)?

Same as the following: is there a proof \( t \) of length \( p(|s|) \) such that \( C(s, t) \) says \textbf{YES}.

How do we reduce \( X \) to \textbf{CSAT}? Need an algorithm \( \mathcal{A} \) that

- takes \( s \) (and \( < p, q, C > \)) and creates a circuit \( G \) in polynomial time in \( |s| \) (note that \( < p, q, C > \) are fixed).
- \( G \) is satisfiable if and only if there is a proof \( t \) such that \( C(s, t) \) says \textbf{YES}.
Reducing $X$ to CSAT

Thus an NP problem is essentially a three tuple $< p, q, C >$ where $C$ is either a program or TM.

Problem $X$: Given string $s$, is $s \in X$?

Same as the following: is there a proof $t$ of length $p(|s|)$ such that $C(s, t)$ says YES.

How do we reduce $X$ to CSAT? Need an algorithm $A$ that

- takes $s$ (and $< p, q, C >$) and creates a circuit $G$ in polynomial time in $|s|$ (note that $< p, q, C >$ are fixed).

- $G$ is satisfiable if and only if there is a proof $t$ such that $C(s, t)$ says YES.
Reducing $X$ to CSAT

How do we reduce $X$ to CSAT? Need an algorithm $A$ that
- takes $s$ (and $<p, q, C>$) and creates a circuit $G$ in polynomial time in $|s|$ (note that $<p, q, C>$ are fixed).
- $G$ is satisfiable if and only if there is a proof $t$ such that $C(s, t)$ says YES

Simple but Big Idea: Programs are essentially the same as Circuits!
- Convert $C(s, t)$ into a circuit $G$ with $t$ as unknown inputs (rest is known including $s$)
- We know that $|t| = p(|s|)$ so express boolean string $t$ as $p(|s|)$ variables $t_1, t_2, \ldots, t_k$ where $k = p(|s|)$.
- Asking if there is a proof $t$ that makes $C(s, t)$ say YES is same as whether there is an assignment of values to “unknown” variables $t_1, t_2, \ldots, t_k$ that will make $G$ evaluate to true/YES.
Reducing X to CSAT

How do we reduce X to CSAT? Need an algorithm A that

- takes s (and < p, q, C >) and creates a circuit G in polynomial time in |s| (note that < p, q, C > are fixed).
- G is satisfiable if and only if there is a proof t such that C(s, t) says YES

Simple but Big Idea: Programs are essentially the same as Circuits!

- Convert C(s, t) into a circuit G with t as unknown inputs (rest is known including s)
- We know that |t| = p(|s|) so express boolean string t as p(|s|) variables t_1, t_2, ..., t_k where k = p(|s|).
- Asking if there is a proof t that makes C(s, t) say YES is same as whether there is an assignment of values to “unknown” variables t_1, t_2, ..., t_k that will make G evaluate to true/YES.
Reducing X to CSAT

How do we reduce X to CSAT? Need an algorithm $A$ that

- takes $s$ (and $<p, q, C>$) and creates a circuit $G$ in polynomial
time in $|s|$ (note that $<p, q, C>$ are fixed).
- $G$ is satisfiable if and only if there is a proof $t$ such that $C(s, t)$
says YES

Simple but Big Idea: Programs are essentially the same as Circuits!

- Convert $C(s, t)$ into a circuit $G$ with $t$ as unknown inputs (rest
is known including $s$)
- We know that $|t| = p(|s|)$ so express boolean string $t$ as $p(|s|)$
variables $t_1, t_2, \ldots, t_k$ where $k = p(|s|)$.
- Asking if there is a proof $t$ that makes $C(s, t)$ say YES is same
as whether there is an assignment of values to “unknown”
variables $t_1, t_2, \ldots, t_k$ that will make $G$ evaluate to true/YES.
Reducing X to CSAT

How do we reduce X to CSAT? Need an algorithm $\mathcal{A}$ that
- takes $s$ (and $<p, q, C>$) and creates a circuit $G$ in polynomial time in $|s|$ (note that $<p, q, C>$ are fixed).
- $G$ is satisfiable if and only if there is a proof $t$ such that $C(s, t)$ says YES

Simple but Big Idea: Programs are essentially the same as Circuits!
- Convert $C(s, t)$ into a circuit $G$ with $t$ as unknown inputs (rest is known including $s$)
- We know that $|t| = p(|s|)$ so express boolean string $t$ as $p(|s|)$ variables $t_1, t_2, \ldots, t_k$ where $k = p(|s|)$.
- Asking if there is a proof $t$ that makes $C(s, t)$ say YES is same as whether there is an assignment of values to “unknown” variables $t_1, t_2, \ldots, t_k$ that will make $G$ evaluate to true/YES.
Reducing \( X \) to \textbf{CSAT}

How do we reduce \( X \) to \textbf{CSAT}? Need an algorithm \( \mathcal{A} \) that

- takes \( s \) (and \( < p, q, C > \)) and creates a circuit \( G \) in polynomial time in \( |s| \) (note that \( < p, q, C > \) are fixed).
- \( G \) is satisfiable if and only if there is a proof \( t \) such that \( C(s, t) \) says YES

Simple but Big Idea: Programs are essentially the same as Circuits!

- Convert \( C(s, t) \) into a circuit \( G \) with \( t \) as unknown inputs (rest is known including \( s \))
- We know that \( |t| = p(|s|) \) so express boolean string \( t \) as \( p(|s|) \) variables \( t_1, t_2, \ldots, t_k \) where \( k = p(|s|) \).
- Asking if there is a proof \( t \) that makes \( C(s, t) \) say YES is same as whether there is an assignment of values to “unknown” variables \( t_1, t_2, \ldots, t_k \) that will make \( G \) evaluate to true/YES.
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

Formally, why is **Independent Set** in **NP**?
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

Formally, why is **Independent Set** in **NP**?
Example: Independent Set

Formally why is Independent Set in NP?

- **Input:**
  \[ \langle n, y_{1,1}, y_{1,2}, \ldots, y_{1,n}, y_{2,1}, \ldots, y_{2,n}, \ldots, y_{n,1}, \ldots, y_{n,n}, k \rangle \]
  encodes \( \langle G, k \rangle \).
  - \( n \) is number of vertices in \( G \)
  - \( y_{i,j} \) is a bit which is 1 if edge \((i, j)\) is in \( G \) and 0 otherwise (adjacency matrix representation)
  - \( k \) is size of independent set

- **Certificate:** \( t = t_1 t_2 \ldots t_n \). Interpretation is that \( t_i \) is 1 if vertex \( i \) is in the independent set, 0 otherwise.
Certifier \( C(s, t) \) for Independent Set:

\[
\text{if } (t_1 + t_2 + \ldots + t_n < k) \text{ then } \\
\quad \text{return NO} \\
\text{else} \\
\quad \text{for each } (i, j) \text{ do} \\
\quad \quad \text{if } (t_i \land t_j \land y_{i,j}) \text{ then} \\
\quad \quad \quad \text{return NO} \\
\text{return YES}
\]
Example: Independent Set

Figure: Graph $G$ with $k = 2$
Circuit from Certifier
Consider “program” $A$ that takes $f(|s|)$ steps on input string $s$.

**Question:** What computer is the program running on and what does step mean?

Real computers difficult to reason with mathematically because
- instruction set is too rich
- pointers and control flow jumps in one step
- assumption that pointer to code fits in one word

**Turing Machines**
- simpler model of computation to reason with
- can simulate real computers with polynomial slow down
- all moves are local (head moves only one cell)
Consider “program” $A$ that takes $f(|s|)$ steps on input string $s$.

**Question:** What computer is the program running on and what does *step* mean?

Real computers difficult to reason with mathematically because
- instruction set is too rich
- pointers and control flow jumps in one step
- assumption that pointer to code fits in one word

**Turing Machines**
- simpler model of computation to reason with
- can simulate real computers with *polynomial* slow down
- all moves are *local* (head moves only one cell)
Certifiers that at TMs

Assume $C(\cdot, \cdot)$ is a (deterministic) Turing Machine $M$

Problem: Given $M$, input $s$, $p$, $q$ decide if there is a proof $t$ of length $p(|s|)$ such that $M$ on $s$, $t$ will halt in $q(|s|)$ time and say YES.

There is an algorithm $A$ that can reduce above problem to CSAT mechanically as follows.

1. $A$ first computes $p(|s|)$ and $q(|s|)$.
2. Knows that $M$ can use at most $q(|s|)$ memory/tape cells
3. Knows that $M$ can run for at most $q(|s|)$ time
4. Simulates the evolution of the state of $M$ and memory over time using a big circuit.
Think of $M$’s state at time $\ell$ as a string $x^\ell = x_1 x_2 \ldots x_k$ where each $x_i \in \{0, 1, B\} \times Q \cup \{q_{-1}\}$.

At time 0 the state of $M$ consists of input string $s$ a guess $t$ (unknown variables) of length $p(|s|)$ and rest $q(|s|)$ blank symbols.

At time $q(|s|)$ we wish to know if $M$ stops in $q_{\text{accept}}$ with say all blanks on the tape.

We write a circuit $C_\ell$ which captures the transition of $M$ from time $\ell$ to time $\ell + 1$.

Composition of the circuits for all times 0 to $q(|s|)$ gives a big (still poly) sized circuit $C$.

The final output of $C$ should be true if and only if the entire state of $M$ at the end leads to an accept state.
NP-Hardness of Circuit Satisfaction

Key Ideas in reduction:
- Use TMs as the code for certifier for simplicity
- Since $p()$ and $q()$ are known to $A$, it can set up all required memory and time steps in advance
- Simulate computation of the TM from one time to the next as a circuit that only looks at three adjacent cells at a time

Note: Above reduction can be done to SAT as well. Reduction to SAT was the original proof of Steve Cook.
NP-Hardness of Circuit Satisfaction

Key Ideas in reduction:

- Use $\text{TMs}$ as the code for certifier for simplicity
- Since $p()$ and $q()$ are known to $A$, it can set up all required memory and time steps in advance
- Simulate computation of the $\text{TM}$ from one time to the next as a circuit that only looks at three adjacent cells at a time

Note: Above reduction can be done to $\text{SAT}$ as well. Reduction to $\text{SAT}$ was the original proof of Steve Cook.
SAT is NP-Complete

- We have seen that SAT ∈ NP
- To show NP-Hardness, we will reduce Circuit Satisfiability (CSAT) to SAT

Instance of CSAT (we label each node):

```
Inputs:
1, a
? , b
? , c
0, d
? , e

Output:
¬, i
∧, j
∧, k
∧, f
∨, g
∨, h
```

Sariel (UIUC)
CS473
Spring 2011
Converting a circuit into a \textbf{CNF} formula

Label the nodes

\textbf{(A) Input circuit}

\textbf{(B) Label the nodes.}
Introduce a variable for each node

(B) Label the nodes.

(C) Introduce var for each node.
Converting a circuit into a **CNF** formula

Write a sub-formula for each variable that is true if the var is computed correctly.

\[ x_k \quad \text{(Demand a sat’ assignment!)} \]
\[ x_k = x_i \land x_k \]
\[ x_j = x_g \land x_h \]
\[ x_i = \neg x_f \]
\[ x_h = x_d \lor x_e \]
\[ x_g = x_b \lor x_c \]
\[ x_f = x_a \land x_b \]
\[ x_d = 0 \]
\[ x_a = 1 \]

(C) Introduce var for each node.

(D) Write a sub-formula for each variable that is true if the var is computed correctly.
Converting a circuit into a **CNF** formula

Convert each sub-formula to an equivalent CNF formula

<table>
<thead>
<tr>
<th>$x_k$</th>
<th>$x_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_k = x_i \land x_j$</td>
<td>$(\neg x_k \lor x_i) \land (\neg x_k \lor x_j) \land (x_k \lor \neg x_i \lor \neg x_j)$</td>
</tr>
<tr>
<td>$x_j = x_g \land x_h$</td>
<td>$(\neg x_j \lor x_g) \land (\neg x_j \lor x_h) \land (x_j \lor \neg x_g \lor \neg x_h)$</td>
</tr>
<tr>
<td>$x_i = \neg x_f$</td>
<td>$(x_i \lor x_f) \land (\neg x_i \lor x_f)$</td>
</tr>
<tr>
<td>$x_h = x_d \lor x_e$</td>
<td>$(x_h \lor \neg x_d) \land (x_h \lor \neg x_e) \land (\neg x_h \lor x_d \lor x_e)$</td>
</tr>
<tr>
<td>$x_g = x_b \lor x_c$</td>
<td>$(x_g \lor \neg x_b) \land (x_g \lor \neg x_c) \land (\neg x_g \lor x_b \lor x_c)$</td>
</tr>
<tr>
<td>$x_f = x_a \land x_b$</td>
<td>$(\neg x_f \lor x_a) \land (\neg x_f \lor x_b) \land (x_f \lor \neg x_a \lor \neg x_b)$</td>
</tr>
<tr>
<td>$x_d = 0$</td>
<td>$\neg x_d$</td>
</tr>
<tr>
<td>$x_a = 1$</td>
<td>$x_a$</td>
</tr>
</tbody>
</table>
Converting a circuit into a CNF formula

Take the conjunction of all the CNF sub-formulas

We got a CNF formula that is satisfiable if and only if the original circuit is satisfiable.
Reduction: \( \text{CSAT} \leq_p \text{SAT} \)

- For each gate (vertex) \( v \) in the circuit, create a variable \( x_v \).
- **Case** \( \neg \): \( v \) is labeled \( \neg \) and has one incoming edge from \( u \) (so \( x_v = \neg x_u \)). In **SAT** formula generate, add clauses \( (x_u \lor x_v) \), \( (\neg x_u \lor \neg x_v) \). Observe that

\[
x_v = \neg x_u \text{ is true } \iff (x_u \lor x_v) \land (\neg x_u \lor \neg x_v) \text{ both true.}
\]
Case $\lor$: So $x_v = x_u \lor x_w$. In SAT formula generated, add clauses $(x_v \lor \neg x_u)$, $(x_v \lor \neg x_w)$, and $(\neg x_v \lor x_u \lor x_w)$. Again, observe that

$$x_v = x_u \lor x_w \text{ is true } \iff (x_v \lor \neg x_u), \quad (x_v \lor \neg x_w), \quad \text{all true.}$$
Reduction: $\text{CSAT} \leq_p \text{SAT}$

Continued...

- **Case $\wedge$:** So $x_v = x_u \land x_w$. In $\text{SAT}$ formula generated, add clauses $(\neg x_v \lor x_u)$, $(\neg x_v \lor x_w)$, and $(x_v \lor \neg x_u \lor \neg x_w)$. Again observe that

\[ x_v = x_u \land x_w \text{ is true } \iff (\neg x_v \lor x_u), \quad (\neg x_v \lor x_w), \quad (x_v \lor \neg x_u \lor \neg x_w) \text{ all true.} \]
Reduction: \textbf{CSAT} \leq_{P} \textbf{SAT}

Continued...

- If $v$ is an input gate with a fixed value then we do the following. If $x_v = 1$ add clause $x_v$. If $x_v = 0$ add clause $\neg x_v$.
- Add the clause $x_v$ where $v$ is the variable for the output gate.
Correctness of Reduction

Need to show circuit $C$ is satisfiable iff $\varphi_C$ is satisfiable

$\Rightarrow$ Consider a satisfying assignment $a$ for $C$

- Find values of all gates in $C$ under $a$
- Give value of gate $v$ to variable $x_v$; call this assignment $a'$
- $a'$ satisfies $\varphi_C$ (exercise)

$\Leftarrow$ Consider a satisfying assignment $a$ for $\varphi_C$

- Let $a'$ be the restriction of $a$ to only the input variables
- Value of gate $v$ under $a'$ is the same as value of $x_v$ in $a$
- Thus, $a'$ satisfies $C$

Theorem

$\text{SAT}$ is NP-Complete.
Proving that a problem \( X \) is \textbf{NP-Complete}

To prove \( X \) is \textbf{NP-Complete}, show

- Show \( X \) is in \textbf{NP}.
  - certificate/proof of polynomial size in input
  - polynomial time certifier \( C(s, t) \)
- Reduction from a known \textbf{NP-Complete} problem such as \textbf{CSAT} or \textbf{SAT} to \( X \)

\( \text{SAT} \leq_p X \) implies that every \textbf{NP} problem \( Y \leq_p X \). Why?

Transitivity of reductions:

\( Y \leq_p \text{SAT} \) and \( \text{SAT} \leq_p X \) and hence \( Y \leq_p X \).
To prove \( X \) is **NP-Complete**, show

- Show \( X \) is in **NP**.
  - certificate/proof of polynomial size in input
  - polynomial time certifier \( C(s, t) \)
- Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to \( X \)

\( \text{SAT} \leq_P X \) implies that every \( \text{NP} \) problem \( Y \leq_P X \). Why?

Transitivity of reductions:

\( Y \leq_P \text{SAT} \) and \( \text{SAT} \leq_P X \) and hence \( Y \leq_P X \).
Proving that a problem $X$ is NP-Complete

To prove $X$ is NP-Complete, show

- Show $X$ is in NP.
  - certificate/proof of polynomial size in input
  - polynomial time certifier $C(s, t)$
- Reduction from a known NP-Complete problem such as CSAT or 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 $Y \leq_P X$. 
NP-Completeness via Reductions

- **CSAT** is NP-Complete
- **CSAT \( \leq_p \) SAT** and SAT is in NP and hence SAT is NP-Complete
- **SAT \( \leq_p \) 3-SAT** and hence 3-SAT is NP-Complete
- **3-SAT \( \leq_p \) Independent Set** (which is in NP) and hence Independent Set is NP-Complete
- Vertex Cover is NP-Complete
- Clique is NP-Complete

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

A surprisingly frequent phenomenon!
NP-Completeness via Reductions

- **CSAT** is **NP-Complete**
- **CSAT** $\leq_p$ **SAT** and **SAT** is in **NP** and hence **SAT** is **NP-Complete**
- **SAT** $\leq_p$ **3-SAT** and hence **3-SAT** is **NP-Complete**
- **3-SAT** $\leq_p$ Independent Set (which is in **NP**) and hence Independent Set is **NP-Complete**
- Vertex Cover is **NP-Complete**
- Clique is **NP-Complete**

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

A surprisingly frequent phenomenon!