**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 TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$:

- $Q$ 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 $\downarrow$)
- $\delta : Q \times \Gamma \rightarrow \{L, R\} \times \Gamma \times Q$ 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 $q_{\text{accept}}$.
**P via TMs**

**Definition**

$M$ is a polynomial time $\text{TM}$ 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 $\text{TM}$ $M$ such that $L = L(M)$.

---

**NP via TMs**

**Definition**

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

**Non-deterministic $\text{TM}$:** each step has a choice of moves

1. $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$.
2. 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 $(q, a)$.
3. $L(M)$: set of all strings $s$ on which there exists some sequence of valid choices at each step that lead from $q_0$ to $q_{\text{accept}}$.

---

**Non-deterministic TMs vs certifiers**

Two definition of $\text{NP}$:

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

**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 $\text{NP}$ iff $L$ is accepted by a $\text{NTM}$ which first guesses a proof $t$ of length poly in input $|s|$ and then acts as a deterministic $\text{TM}$.

---

**Non-determinism, guessing and verification**

1. 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.
2. 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”.
3. 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.
Algorithms: **TM**s vs **RAM** Model

Why do we use **TM**s some times and **RAM** Model other times?

- **TM**s 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

---

**NP-Complete** Problems

**Definition**

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

- $X \in \text{NP}$, and
- *(Hardness)* For any $Y \in \text{NP}$, $Y \leq P X$.

---

“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**.

---

Solving **NP-Complete** Problems

**Proposition**

Suppose $X$ is **NP-Complete**. Then $X$ can be solved in polynomial time if and only if $P = \text{NP}$.

**Proof.**

⇒ Suppose $X$ can be solved in polynomial time

- Let $Y \in \text{NP}$. 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 $Y \in \text{NP}$ is such that $Y \in P$; $\text{NP} \subseteq P$.
- Since $P \subseteq \text{NP}$, we have $P = \text{NP}$.

⇐ Since $P = \text{NP}$, and $X \in \text{NP}$, 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 $Y \in \text{NP}$, we have that $Y \leq_p X$.

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

NP-Complete Problems

Question
Are there any problems that are NP-Complete?

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

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.
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
1. **CSAT** is in NP.
2. every NP problem X reduces to CSAT.

---

**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:

1. 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.
2. If s is a NO instance (s ∉ X) then for every string t of length at p(|s|), C(s, t) says NO.
3. C(s, t) runs in time q(|s| + |t|) time (hence polynomial time).

---

Reducing X to CSAT

X is in NP means we have access to p(), q(), C(·, ·).

What is C(·, ·)? 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

Thus an \( \text{NP} \) problem is essentially a three tuple \( \langle p, q, C \rangle \) 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 CSAT? Need an algorithm \( A \) that

1. takes \( s \) (and \( \langle p, q, C \rangle \)) and creates a circuit \( G \) in polynomial time in \( |s| \) (note that \( \langle p, q, C \rangle \) are fixed).
2. \( G \) 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!

1. Convert \( C(s, t) \) into a circuit \( G \) with \( t \) as unknown inputs (rest is known including \( s \)).
2. We know that \( |t| = p(|s|) \) so express boolean string \( t \) as \( p(|s|) \) variables \( t_1, t_2, \ldots, t_k \) where \( k = p(|s|) \).
3. 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 \( \text{NP} \)?

1. **Input:** \( < n, y_{1,1}, y_{1,2}, \ldots, y_{1,n}, y_{2,1}, \ldots, y_{2,n}, \ldots, y_{n,1}, \ldots, y_{n,n}, k > \)
2. **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 for Independent Set

Certifier $C(s, t)$ for Independent Set:

if $(t_1 + t_2 + \ldots + t_n < k)$ then
    return NO
else
    for each $(i,j)$ do
        if $(t_i \wedge t_j \wedge y_{i,j})$ then
            return NO
    return YES

Example: Independent Set

A certifier circuit for Independent Set

Programs, Turing Machines and Circuits

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

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 $\text{CSAT}$ mechanically as follows.

- $A$ first computes $p(|s|)$ and $q(|s|)$.
- Knows that $M$ can use at most $q(|s|)$ memory/tape cells
- Knows that $M$ can run for at most $q(|s|)$ time
- Simulates the evolution of the state of $M$ and memory over time using a big circuit.
Simulation of Computation via 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.

SAT is NP-Complete

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

Instance of CSAT (we label each node):

Converting a circuit into a CNF formula

Label the nodes

(A) Input circuit

(B) Label the nodes.
Converting a circuit into a CNF formula

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 (Demand a sat' assignment!)

x_k = x_i ∨ x_j

x_j = x_g ∨ x_h

x_i = ¬x_f

x_h = x_d ∨ x_e

x_g = x_b ∨ x_c

x_f = x_a ∧ x_b

x_d = 0

x_a = 1

We got a CNF formula that is satisfiable if and only if the original circuit is satisfiable.

Converting a circuit into a CNF formula

Convert each sub-formula to an equivalent CNF formula

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

\[
\begin{array}{|c|c|}
\hline
x_k & x_k \\
\hline
x_k = x_i \land x_j & (\neg x_k \lor x_i) \land (\neg x_k \lor x_j) \land (x_k \land \neg x_i \land \neg x_j) \\
x_j = x_g \land x_h & (\neg x_j \lor x_g) \land (\neg x_j \land x_h) \land (x_j \lor \neg x_g \land \neg x_h) \\
x_i = \neg x_f & (x_i \lor x_f) \land (\neg x_i \lor x_f) \\
x_h = x_d \lor x_e & (x_h \lor \neg x_d) \land (\neg x_h \land x_d \lor x_e) \\
x_g = x_b \lor x_c & (x_g \lor \neg x_b) \land (\neg x_g \land x_b \lor x_c) \\
x_f = x_a \land x_b & (\neg x_f \lor x_a) \land (\neg x_f \land x_b) \land (x_f \land \neg x_a \land \neg x_b) \\
x_d = 0 & \neg x_d \\
x_a = 1 & x_a \\
\hline
\end{array}
\]
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 $x_v = \neg x_u$ is true $\iff$ $(x_u \lor x_v)$ both true.

Reduction: $\text{CSAT} \leq_p \text{SAT}$

Continued...

- **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 $x_v = x_u \lor x_w$ is true $\iff$ $(x_v \lor \neg x_u)$, $(x_v \lor \neg x_w)$, all true. $(\neg x_v \lor x_u \lor x_w)$

Reduction: $\text{CSAT} \leq_p \text{SAT}$

Continued...

- **Case $\land$**: So $x_v = x_u \land x_w$. In 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 $x_v = x_u \land x_w$ is true $\iff$ $(\neg x_v \lor x_u)$, $(\neg x_v \lor x_w)$, all true. $(x_v \lor \neg x_u \lor \neg x_w)$

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

$\Leftarrow$ Consider a satisfying assignment $a$ for $\varphi_C$
  1. Let $a'$ be the restriction of $a$ to only the input variables
  2. Value of gate $v$ under $a'$ is the same as value of $x_v$ in $a$
  3. Thus, $a'$ satisfies $C$

Theorem

$SAT$ is NP-Complete.

Proving that a problem $X$ is NP-Complete

To prove $X$ is NP-Complete, show
  1. Show $X$ is in NP.
     - certificate/proofof polynomial size in input
     - polynomial time certifier $C(s, t)$
  2. 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

  1. $CSAT$ is NP-Complete.
  2. $CSAT \leq_p SAT$ and $SAT$ is in NP and hence $SAT$ is NP-Complete.
  3. $SAT \leq_p 3$-SAT and hence 3-SAT is NP-Complete.
  4. $3$-SAT $\leq_p$ Independent Set (which is in NP) and hence Independent Set is NP-Complete.
  5. Vertex Cover is NP-Complete.
  6. 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!