NP Completeness and Cook-Levin Theorem

Lecture 22

April 19, 2011

P and NP and Turing Machines

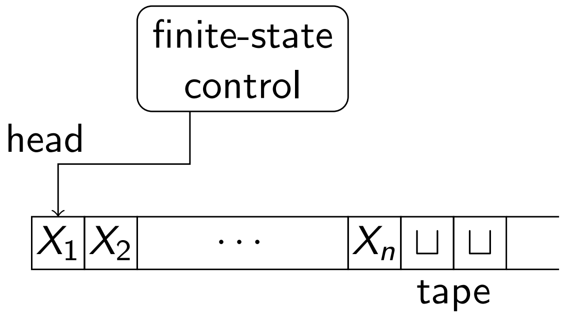
- **P**: set of decision problems that have polynomial time algorithms
- **NP**: set of decision problems that have polynomial time non-deterministic algorithms

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.

Turing Machines: Recap



- Infinite tape
- Finite state control
- Input at beginning of tape
- Special tape letter "blank" $\sqcup$
- Head can move only one cell to left or right

Turing Machines: Formally

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

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

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

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

P via TMs

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 $L = L(M)$.

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

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

Non-deterministic TMs vs certifiers

Two definition of **NP**:

- **L** is in **NP** iff **L** has a polynomial time certifier $\mathbf{C}(\cdot, \cdot)$.
- **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 $|\mathbf{s}|$ and then acts as a *deterministic* TM.

Non-determinism, guessing and verification

- 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.

Algorithms: TMs vs RAM Model

Why do we use TMs some times 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 $\mathbf{X}$ is said to be **NP-Complete** if

- $\mathbf{X} \in \mathbf{NP}$
- (Hardness) For any $\mathbf{Y} \in \mathbf{NP}$, $\mathbf{Y} \leq_P \mathbf{X}$

Solving NP-Complete Problems

Proposition

Suppose $\mathbf{X}$ *is* **NP-Complete**. *Then* $\mathbf{X}$ *can be solved in polynomial time iff* $\mathbf{P} = \mathbf{NP}$

Proof.

$\Rightarrow$ Suppose $\mathbf{X}$ can be solved in polynomial time

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

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

NP-Hard Problems

Definition

A problem $\mathbf{X}$ is said to be **NP-Hard** if

- (Hardness) For any $\mathbf{Y} \in \mathbf{NP}$, $\mathbf{Y} \leq_P \mathbf{X}$

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 $\mathbf{X}$ is **NP-Complete**

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

$\mathbf{X}$ is unlikely to be efficiently solvable.

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

NP-Complete Problems

Question

Are there any problems that are **NP-Complete**?

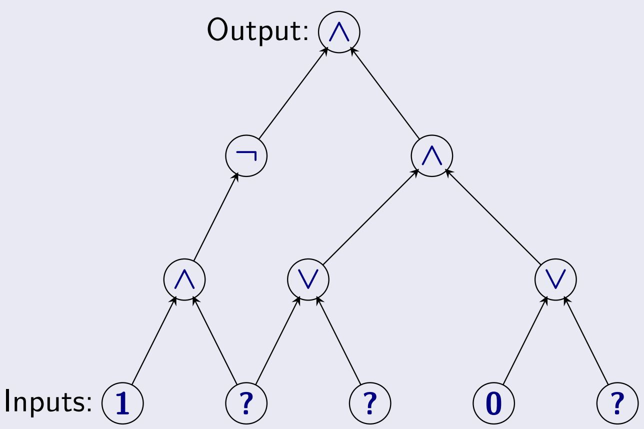
Answer

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 $\vee$, $\wedge$ 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

- **CSAT** is in **NP**
- *every* **NP** problem **X** reduces to **CSAT**.

CSAT: Circuit Satisfaction

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 $\mathbf{X}$ reduces to CSAT.

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

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

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

Reducing X to CSAT

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

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

Reducing X to CSAT

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

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

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

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

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

Reducing X to CSAT

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

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

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

- Convert $\mathbf{C}(\mathbf{s}, \mathbf{t})$ into a circuit $\mathbf{G}$ with $\mathbf{t}$ as unknown inputs (rest is known including $\mathbf{s}$)
- We know that $|\mathbf{t}| = \mathbf{p}(|\mathbf{s}|)$ so express boolean string $\mathbf{t}$ as $\mathbf{p}(|\mathbf{s}|)$ variables $\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_k$ where $\mathbf{k} = \mathbf{p}(|\mathbf{s}|)$.
- Asking if there is a proof $\mathbf{t}$ that makes $\mathbf{C}(\mathbf{s}, \mathbf{t})$ say YES is same as whether there is an assignment of values to "unknown" variables $\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_k$ that will make $\mathbf{G}$ evaluate to true/YES.

Example: Independent Set

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

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

Example: Independent Set

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

- Input:
  $< \mathbf{n}, \mathbf{y}_{1,1}, \mathbf{y}_{1,2}, \ldots, \mathbf{y}_{1,n}, \mathbf{y}_{2,1}, \ldots, \mathbf{y}_{2,n}, \ldots, \mathbf{y}_{n,1}, \ldots, \mathbf{y}_{n,n}, \mathbf{k} >$ encodes $< \mathbf{G}, \mathbf{k} >$.
  - $\mathbf{n}$ is number of vertices in $\mathbf{G}$
  - $\mathbf{y}_{i,j}$ is a bit which is $\mathbf{1}$ if edge $(\mathbf{i}, \mathbf{j})$ is in $\mathbf{G}$ and $\mathbf{0}$ otherwise (adjacency matrix representation)
  - $\mathbf{k}$ is size of independent set
- Certificate: $\mathbf{t} = \mathbf{t}_1\mathbf{t}_2\ldots\mathbf{t}_n$. Interpretation is that $\mathbf{t}_i$ is $\mathbf{1}$ if vertex $\mathbf{i}$ is in the independent set, $\mathbf{0}$ otherwise.

Certifier for **Independent Set**

Certifier $\mathbf{C(s, t)}$ for **Independent Set**:

```
if (t_1 + t_2 + ... + t_n < k) then
    return NO
else
    for each (i,j) do
        if (t_i ∧ t_j ∧ y_{i,j}) then
            return NO

return YES
```


Example: Independent Set

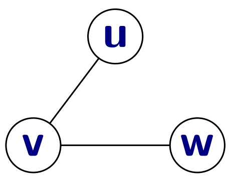
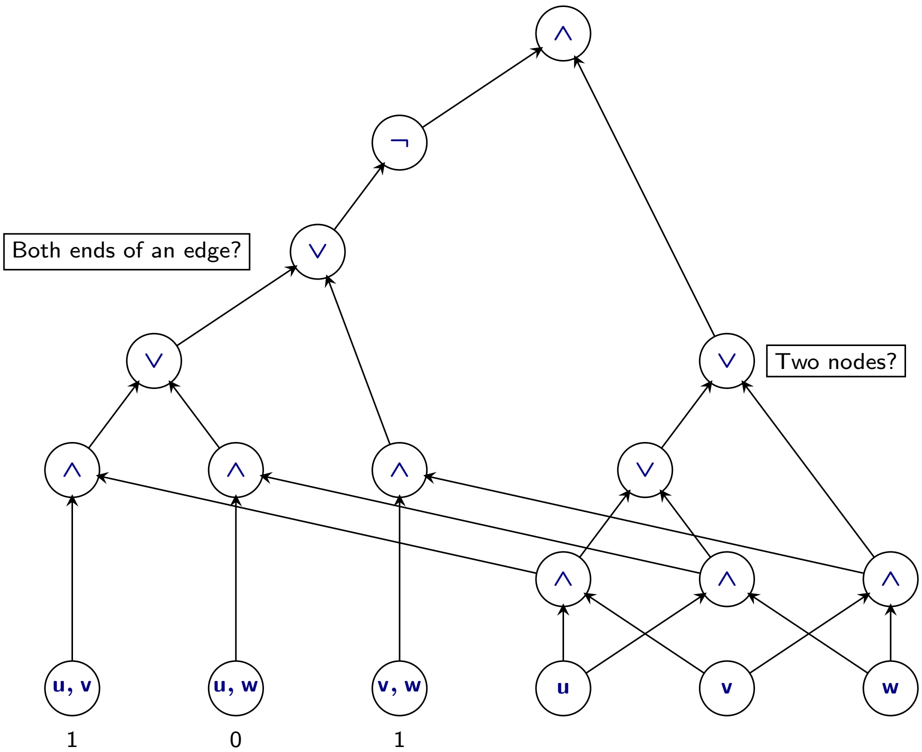


Figure: Graph $\mathbf{G}$ with $\mathbf{k = 2}$



Circuit from Certifier

Programs, Turing Machines and Circuits

Consider "program" $\mathbf{A}$ that takes $\mathbf{f(|s|)}$ steps on input string $\mathbf{s}$.

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 $\mathbf{C}(\cdot,\cdot)$ is a (deterministic) Turing Machine $\mathbf{M}$

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

There is an algorithm $\mathcal{A}$ that can reduce above problem to **CSAT** mechanically as follows.

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

Simulation of Computation via Circuit

- Think of $\mathbf{M}$'s state at time ℓ as a string $\mathbf{x}^\ell = \mathbf{x}_1\mathbf{x}_2\ldots\mathbf{x}_k$ where each $\mathbf{x}_i \in \{0,1,\mathbf{B}\} \times \mathbf{Q} \cup \{\mathbf{q}_{-1}\}$.
- At time $\mathbf{0}$ the state of $\mathbf{M}$ consists of input string $\mathbf{s}$ a guess $\mathbf{t}$ (unknown variables) of length $\mathbf{p}(|\mathbf{s}|)$ and rest $\mathbf{q}(|\mathbf{s}|)$ blank symbols.
- At time $\mathbf{q}(|\mathbf{s}|)$ we wish to know if $\mathbf{M}$ stops in $\mathbf{q}_{\mathbf{accept}}$ with say all blanks on the tape.
- We write a circuit $\mathbf{C}_\ell$ which captures the transition of $\mathbf{M}$ from time ℓ to time $\ell + 1$.
- Composition of the circuits for all times $\mathbf{0}$ to $\mathbf{q}(|\mathbf{s}|)$ gives a big (still poly) sized circuit $\mathcal{C}$
- The final output of $\mathcal{C}$ should be true if and only if the entire state of $\mathbf{M}$ at the end leads to an accept state.

NP-Hardness of Circuit Satisfaction

Key Ideas in reduction:

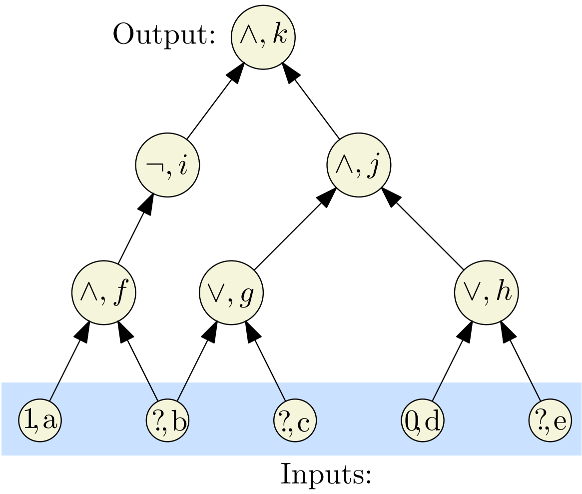
- Use TMs as the code for certifier for simplicity
- Since **p()** and **q()** are known to $\mathcal{A}$, it can set up all required memory and time steps in advance
- Simulate computation of the TM from one time to the next as a circuit that only looks at three adjacent cells at a time

Note: Above reduction can be done to **SAT** as well. Reduction to **SAT** was the original proof of Steve Cook.

SAT is NP-Complete

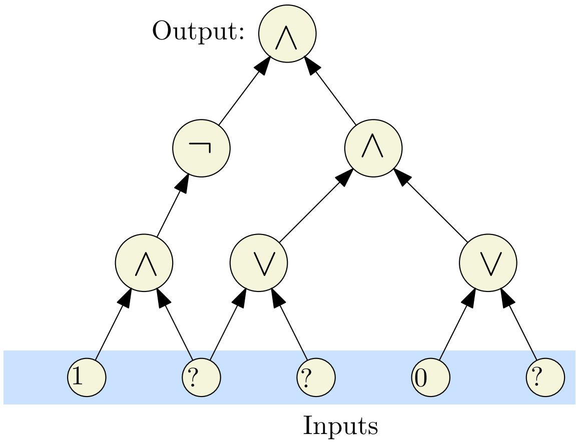
- We have seen that **SAT** $\in$ **NP**
- To show **NP-Hardness**, we will reduce Circuit Satisfiability (**CSAT**) to **SAT**

  Instance of **CSAT** (we label each node):

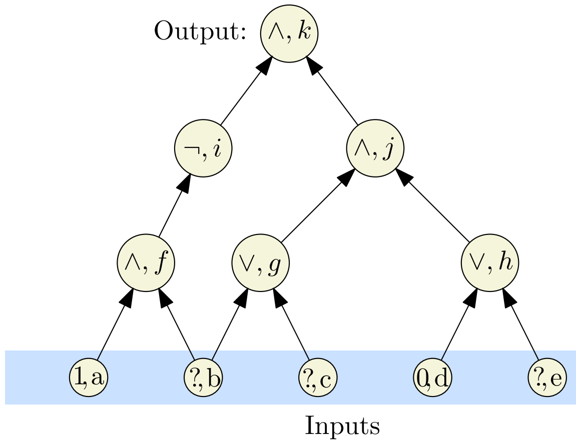


Converting a circuit into a CNF formula

Label the nodes



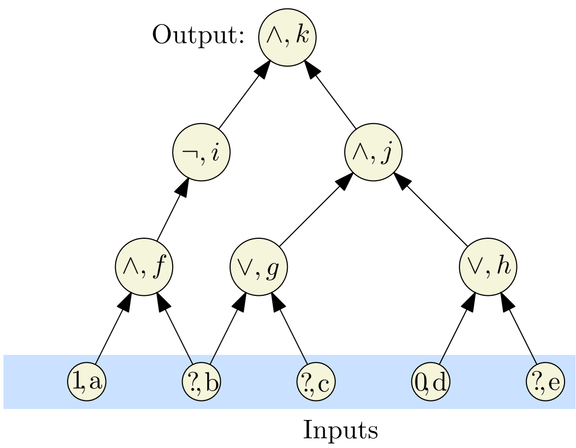
(A) Input circuit



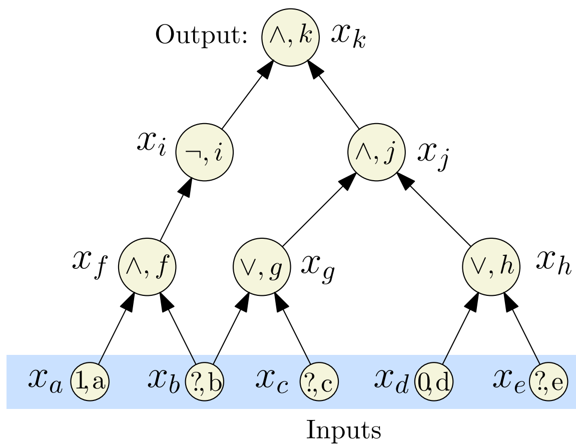
(B) Label the nodes.

Converting a circuit into a CNF formula

Introduce a variable for each node



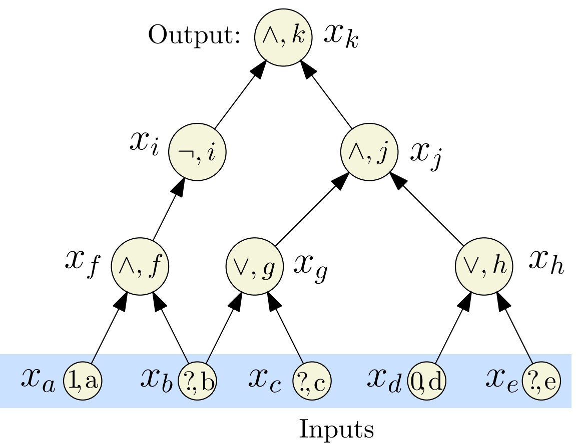
(B) Label the nodes.



(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.



(C) Introduce var for each node.

$\mathsf{x_k}$ (Demand a sat' assignment!)

$\mathsf{x_k = x_i \wedge x_k}$

$\mathsf{x_j = x_g \wedge x_h}$

$\mathsf{x_i = \neg x_f}$

$\mathsf{x_h = x_d \vee x_e}$

$\mathsf{x_g = x_b \vee x_c}$

$\mathsf{x_f = x_a \wedge x_b}$

$\mathsf{x_d = 0}$

$\mathsf{x_a = 1}$

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

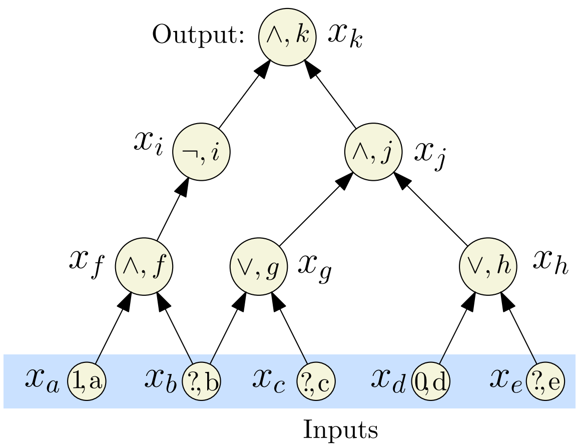
Converting a circuit into a CNF formula

Convert each sub-formula to an equivalent CNF formula

x_k	x_k
$x_k = x_i \wedge x_j$	$(\neg x_k \vee x_i) \wedge (\neg x_k \vee x_j) \wedge (x_k \vee \neg x_i \vee \neg x_j)$
$x_j = x_g \wedge x_h$	$(\neg x_j \vee x_g) \wedge (\neg x_j \vee x_h) \wedge (x_j \vee \neg x_g \vee \neg x_h) \wedge$
$x_i = \neg x_f$	$(x_i \vee x_f) \wedge (\neg x_i \vee x_f) \wedge$
$x_h = x_d \vee x_e$	$(x_h \vee \neg x_d) \wedge (x_h \vee \neg x_e) \wedge (\neg x_h \vee x_d \vee x_e)$
$x_g = x_b \vee x_c$	$(x_g \vee \neg x_b) \wedge (x_g \vee \neg x_c) \wedge (\neg x_g \vee x_b \vee x_c)$
$x_f = x_a \wedge x_b$	$(\neg x_f \vee x_a) \wedge (\neg x_f \vee x_b) \wedge (x_f \vee \neg x_a \vee \neg x_b)$
$x_d = 0$	$\neg x_d$
$x_a = 1$	x_a

Converting a circuit into a CNF formula

Take the conjunction of all the CNF sub-formulas



$$
\begin{aligned}
&x_k \wedge (\neg x_k \vee x_i) \wedge (\neg x_k \vee x_j)\\
&\wedge (x_k \vee \neg x_i \vee \neg x_j) \wedge (\neg x_j \vee x_g)\\
&\wedge (\neg x_j \vee x_h) \wedge (x_j \vee \neg x_g \vee \neg x_h)\\
&\wedge (x_i \vee x_f) \wedge (\neg x_i \vee x_f)\\
&\wedge (x_h \vee \neg x_d) \wedge (x_h \vee \neg x_e)\\
&\wedge (\neg x_h \vee x_d \vee x_e) \wedge (x_g \vee \neg x_b)\\
&\wedge (x_g \vee \neg x_c) \wedge (\neg x_g \vee x_b \vee x_c)\\
&\wedge (\neg x_f \vee x_a) \wedge (\neg x_f \vee x_b)\\
&\wedge (x_f \vee \neg x_a \vee \neg x_b) \wedge (\neg x_d) \wedge x_a
\end{aligned}
$$

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

Reduction: CSAT $\leq_P$ 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 $(x_u \vee x_v)$, $(\neg x_u \vee \neg x_v)$. Observe that

$$
x_v = \neg x_u \text{ is true} \iff \begin{array}{l}(x_u \vee x_v)\\(\neg x_u \vee \neg x_v)\end{array} \text{ both true.}
$$

Reduction: CSAT $\leq_P$ SAT

Continued...

- Case $\vee$: So $x_v = x_u \vee x_w$. In SAT formula generated, add clauses $(x_v \vee \neg x_u)$, $(x_v \vee \neg x_w)$, and $(\neg x_v \vee x_u \vee x_w)$. Again, observe that

$$x_v = x_u \vee x_w \text{ is true} \iff \begin{array}{l}(x_v \vee \neg x_u),\\ (x_v \vee \neg x_w),\\ (\neg x_v \vee x_u \vee x_w)\end{array} \text{ all true.}$$

Reduction: CSAT $\leq_P$ SAT

Continued...

- Case $\wedge$: So $x_v = x_u \wedge x_w$. In SAT formula generated, add clauses $(\neg x_v \vee x_u)$, $(\neg x_v \vee x_w)$, and $(x_v \vee \neg x_u \vee \neg x_w)$. Again observe that

$$x_v = x_u \wedge x_w \text{ is true} \iff \begin{array}{l}(\neg x_v \vee x_u),\\ (\neg x_v \vee x_w),\\ (x_v \vee \neg x_u \vee \neg x_w)\end{array} \text{ all true.}$$

Reduction: CSAT $\leq_P$ SAT

Continued...

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

Correctness of Reduction

Need to show circuit $\mathbf{C}$ is satisfiable iff $\varphi_{\mathbf{C}}$ is satisfiable

$\Rightarrow$ Consider a satisfying assignment $\mathbf{a}$ for $\mathbf{C}$

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

$\Leftarrow$ Consider a satisfying assignment $\mathbf{a}$ for $\varphi_{\mathbf{C}}$

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

Theorem

SAT *is* **NP-Complete**.

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 $\mathbf{C(s,t)}$
- 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!