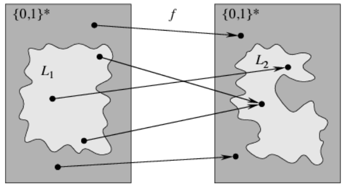
NP-Completeness

Some NP Problems Are As Hard As Any Other NP Problems!

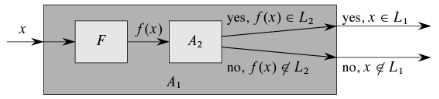
Reducing (Transforming) One Problem To Another

- Intuitively, a decision problem P is said to be "reducible" to another decision problem Q if and only if any instance of P (call it i_P) can be converted (algorithmically) to an instance of Q (call it i_Q), and the answer to i_P is always equal to the answer to i_Q.
- If using the formal language framework, it's a mapping (function) f from $\{0,1\}^*$ to $\{0,1\}^*$ that preserves the membership with the languages: (CLRS Figure 34.4 in pp. 1068)
  - If $x \in L_1$, then $f(x) \in L_2$.
  - If $x \notin L_1$, then $f(x) \notin L_2$.
- We say L_1 is "mapping reducible" to L_2, and this is denoted as $L_1 \leq L_2$.



Meaning Of $L_1 \leq L_2$

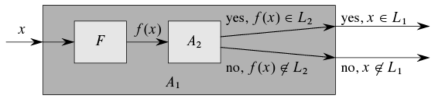
- Why $\leq$? This means L_1 is "no harder to decide (solve)" than L_2.
- That also means, using an algo. for L_2, we can build an algo. for L_1!
  - To see if $x \in L_1$, just compute $f(x)$ and run the decision algorithm for L_2 on $f(x)$, which will give yes or no. If the answer is yes, then $x \in L_1$. If it's no, then $x \notin L_1$.
- A bit confusing, though (in words)
  - Looks like $L_2 \geq L_1$, and yet we say L_1 is "reducible" (or "reduced") to L_2.
  - Just remember that we don't have $\geq$ in this theory. Only $\leq$ and from left to right.



CLRS Fig. 34.5 in pp.1069

Polynomial Time Reduction

- Such a mapping function f is called a "reduction function."
- An algorithm that computes f is called a "reduction algorithm."
- If the reduction algorithm for f is $O(n^k)$ for some constant k, then the reduction is called a "polynomial-time reduction."
- And L_1 is said to be "polynomial-time reducible" to L_2, and this is denoted as $L_1 \leq_P L_2$.
- This gives an interesting theorem (Lemma 34.3 in CLRS):
  - If $L_1 \leq_P L_2$ and $L_2 \in P$, then $L_1 \in P$!
    - Meaning F is $O(n^k)$ and A_2 is $O(n^l)$.
    - Thus A_1 is $F + A_2 = O(n^{\max(k,l)})$, which is still in polynomial time!



NP-Completeness

- An interesting concept arises out of this polynomial-time reduction on the class NP.
- If we can somehow find an NP problem (language, let's call it L) to which every other NP problem can be polynomial-time reducible,
  - That is, if, for every $L' \in NP$, $L' \leq_P L$,
- Then we can somehow consider L as a representative of all NP languages, because:
  - If L is decidable in polynomial time, then every $L' \in NP$ is also decidable in polynomial time (because of Lemma 34.3)!
  - In other words, if such an $L \in P$, then $P = NP$!
- We call such a language L "NP-Complete."

Formal Definition Of NP-Completeness

- A language $L \subseteq \{0,1\}^*$ is ***<u>NP-complete</u>*** if:
  - $L \in NP$, and
  - $L' \leq_P L$ for every $L' \in NP$.
- NPC: The class of all NP-complete languages.
- Theorem 34.4:
  - If there exists $L \in NPC$ such that $L \in P$ as well, then $P = NP$!
    - Obvious by definition of NPC and Lemma 34.3.
  - Equivalently, if $P \neq NP$, there must exist $L \in NPC$ such that $L \notin P$!
    - Just the contrapositive of the first statement.

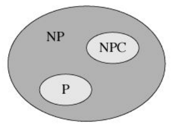


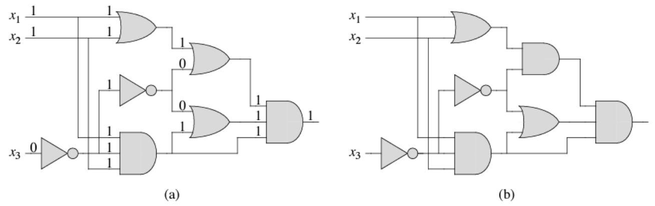
Figure 34.6 How most theoretical computer scientists view the relationships among P, NP, and NPC. Both P and NPC are wholly contained within NP, and P ∩ NPC = ∅.

First NP-Complete Language (Problem)

- So it sounds like a very fancy concept, but does there really exist such a language (problem) at all?
- Indeed, there is. In fact, there are very many!
- The first one is probably most difficult to prove.
  - After that, we can rely on $\leq_P$ (see the next lesson).
- For the first NP-complete problem, we must be able to reduce (transform) *<u>any</u>* NP problem to it in polynomial time.
- The idea is to represent any verification algorithm computation as a Boolean circuit (hardware)!

CIRCUIT-SAT: Definition, $\in NP$

- Boolean circuit, truth assignment, satisfiability (CLRS Fig. 34.8)



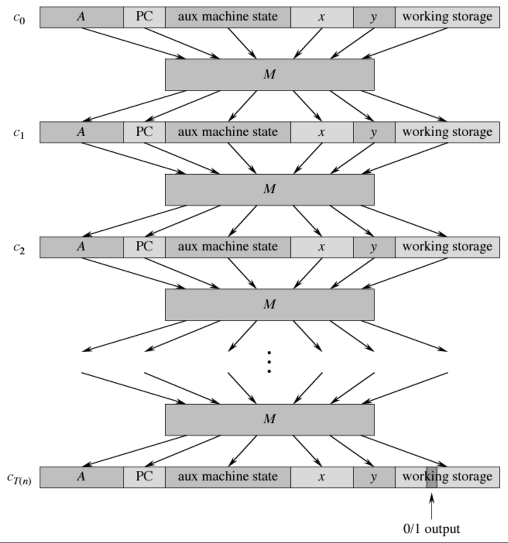
- CIRCUIT-SAT=$\{\langle C \rangle : C$ is a satisfiable Boolean combinational circuit$\}$.
  - Not too hard to see CIRCUIT-SAT $\in NP$.
    - Certificate is Boolean value assignments to all wires.
    - Verification algorithm just checks if the value on the output of any logic gate is equal to the Boolean logic output of the values of the input wires of the logic gate.

$L \leq_P$ CIRCUIT-SAT For Every $L \in NP$

- Key idea: For any $L \in NP$,
  - There must be a polynomial-time verification algorithm A for L.
  - The algorithm A can be implemented as a sequential logic circuit with a bunch of flip-flops and a big combinational logic circuit.
    - Digital logic design and/or computer organization/architecture course topic!
  - Cascade this combinational circuit $T(n) = O(n^k)$ times so that the sequential computation becomes a huge unwound combinational circuit.
  - Show that the size of the huge unwound combinational circuit is still in polynomial to n.
- Actual proof omitted, Read CLRS Section 34.3 if interested.

Unwound Sequential Computation (CLRS Fig. 34.9)

- The result of the sequential computation is one bit in the working storage after enough steps of configuration transitions.
- The size of the resulting unwound combinational circuit is still $O(n^l)$ for some constant l.
- Thus $L \leq_P$ CIRCUIT-SAT for any $L \in NP$!



Summary

- Therefore, CIRCUIT-SAT is NP-complete.
  - No need to remember or fully understand the proof.
  - Just remember that it's related to the hardware implementation of a sequential algorithm, and its realization with a Boolean logic circuit.
- This result will allow us to prove many other NP problems are also NP-complete.
  - Remember the Lemma 34.3: If $L_1 \leq_P L_2$ and $L_2 \in P$, then it must be that $L_1 \in P$!
  - What if you have an $L_1 \in NPC$, and then prove $L_1 \leq_P L_2$ for an $L_2 \in NP$?
    - Since, for every $L' \in NP$, $L' \leq_P L_1$ (definition of $L_1 \in NPC$), we effectively proved that $L' \leq_P L_2$ as well (because $\leq_P$ is transitive), concluding $L_2 \in NPC$ as well!

More NP-Complete Problems

No Need To Do The Same As The First NP-Complete Problem

Proving Subsequent NP-Complete Problems

- Do we still need to show that every NP problem is polynomial-time reducible to a target NP-complete problem?
  - Like we did for CIRCUIT-SAT?
- Technically yes, but can we leverage an already-proven NP-complete problem?
  - We already got a problem to which every NP problem is polynomial-time reducible.
  - What if we show that a chosen & known NP-complete problem is polynomial-time reducible to the target NP-complete problem?

Finding a Polynomial-Time Reduction from a Known NP-Complete Problem is Enough

- Are two polynomial-time reductions equivalent to just one polynomial-time reduction?
- Yes, since the sum of two polynomials is still a polynomial!
  - Class of all polynomials is closed under addition.
- Therefore, if we show that a known NP-complete problem can be polynomial-time reducible to a target NP-complete problem, then we effectively show that the target problem is indeed NP-complete.
- CLRS Lemma 34.8: If $L' \le_P L$ for some $L' \subseteq NPC$ and $L \in NP$, then L is NP-complete.
  - Because for any $L'' \in NP$, $L'' \le_P L'$, and $L' \le_P L$, therefore $L'' \le_P L$ for every $L'' \in NP$.

Recipe To Prove NP-Completeness of Problem (Language L)

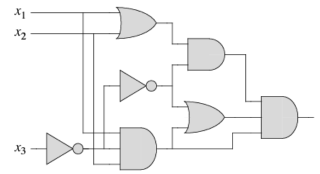
- Prove $L \in NP$ (Mostly straightforward)
- Select a known NP-complete language L'.
  - May have to repeat and choose something else.
- Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L.
- Show that f satisfies that $x \in L'$ iff $f(x) \in L$ for all $x \in \{0,1\}^*$.
- Show that the algorithm that computes f runs in polynomial time.
- NOTE: There's no point in showing $L \le_P L'$!
  - The correct direction is always $L' \le_P L$ for a known NP-complete language L'.

Proving SAT Is NP-Complete

- SAT is our next NP-complete problem that we will prove as such.
- SAT= $\{\langle\phi\rangle : \phi$ is a satisfiable Boolean formula$\}$
  - E.g., for $\phi = \big((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)\big) \wedge \neg x_2$,
  - There is a satisfying assignment $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$, since
    - $\phi = \big((0 \to 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)\big) \wedge \neg 0 = \big(1 \vee \neg(1 \vee 1)\big) \wedge 1 = (1 \vee 0) \wedge 1 = 1$.
- To prove SAT is NP-complete,
  - First, show that SAT $\in NP$.
  - Then choose a known NP-complete problem L.
    - Currently, there's only one: CIRCUIT-SAT !
  - And show that CIRCUIT-SAT $\le_P$ SAT !
    - Remember: It's NOT the other way around: ~~SAT $\le_P$ CIRCUIT-SAT~~

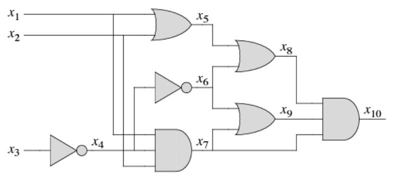
Actual Proof Of SAT's NP-Completeness

- Show SAT $\in NP$.
  - State a poly-time verification algorithm on any certificate.
    - Certificate: An assignment of Boolean values (0/1) to all variables $x_1, \dots, x_n$.
    - Verification algorithm: Replaces every occurrence of x_i with its actual Boolean value assignment in the certificate, evaluate the Boolean expression. Runs in poly-time.
- Show CIRCUIT-SAT $\leq_P$ SAT.
  - Unfortunately, straightforward translation wouldn't work because of duplicates (possibly exponentially many)!
  - E.g., $\phi = \big((x_1 \vee x_2) \wedge \neg\neg x_3\big)$
    $\wedge \big(\neg\neg x_3 \vee (x_1 \wedge x_2 \wedge \neg x_3)\big)$
    $\wedge (x_1 \wedge x_2 \wedge \neg x_3)$



Trick To Make Poly-Time Reduction

- Use intermediate auxiliary variables for internal wires!



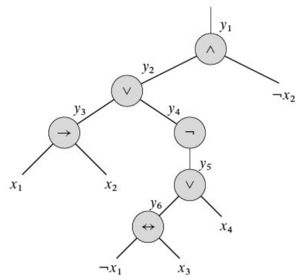
$$\begin{aligned}\phi = x_{10} &\wedge (x_4 \leftrightarrow \neg x_3) \\ &\wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\ &\wedge (x_6 \leftrightarrow \neg x_4) \\ &\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\ &\wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\ &\wedge (x_9 \leftrightarrow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

Proving 3-CNF-SAT's NP-Completeness

- 3-CNF (3-Conjunctive Normal Form): A restricted form of Boolean formulas. A Boolean formula is in 3-CNF if and only if:
  - The formula is a conjunction (AND) of ***clauses***,
  - Each of which is a disjunction (OR) of three ***literals***,
  - Each of which is a variable (x_i) or its negation ($\neg x_i$).
- E.g., $(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \wedge \neg x_4)$
- 3-CNF-SAT $= \{\langle\phi\rangle: \phi$ is a satisfiable 3-CNF Boolean formula$\}$.
  - Is NP-complete.
  - Proof by showing i) 3-CNF-SAT $\in NP$, and ii) SAT $\leq_P$ 3-CNF-SAT.
    - i) is easy (just a special case of SAT)

SAT $\leq_P$ 3-CNF-SAT

- Translate an arbitrary Boolean formula to a 3-CNF Boolean formula!
  - And yet show that the resulting length (longer) is still in polynomial to the original length.
- Use parse tree and introduce auxiliary variables!
- E.g., for $\phi = \Big((x_1 \rightarrow x_2) \wedge \neg\big((\neg x_1 \leftrightarrow x_3) \vee x_4\big)\Big) \wedge \neg x_2$ (CLRS Fig. 34.11),

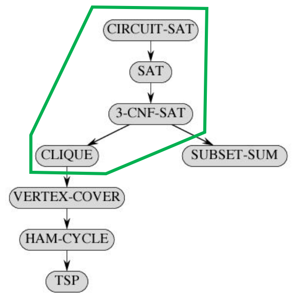


$$\begin{aligned}\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ &\wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ &\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ &\wedge (y_4 \leftrightarrow \neg y_5) \\ &\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ &\wedge (y_6 \leftrightarrow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

Not quite 3-CNF yet, but convertible to 3-CNF not too hard, still in poly-time & size! (omitted)

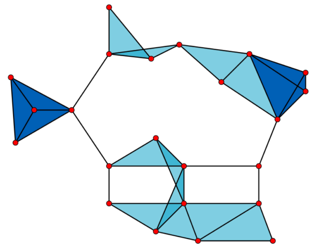
Proving Just One More NP-Complete Problem

- Turns out there are quite many NP-complete problems.
  - In Boolean logic, graphs, arithmetic, network design, scheduling, number theory, games, puzzles, …
- Proofs of interesting NP-complete problems in graph theory are presented in CLRS Section 34.5.
  - We study only one: CLIQUE.
- Note the structure of NP-completeness proofs (CLRS Fig. 34.13)!



Clique: A Complete Sub-Graph Of A Graph

- 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 contains.
  - A clique is called k-clique if its size is k.
  - 1-cliques, 2-cliques are not much interesting.
  - 3-cliques, 4-cliques in Wikipedia example:
- CLIQUE$= \{\langle G, k\rangle$: G is a graph containing a clique of size $k\}$.



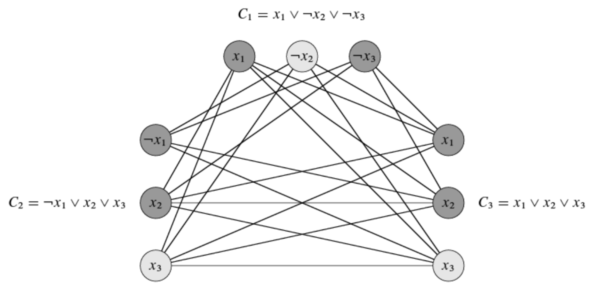
https://en.wikipedia.org/wiki/Clique_(graph_theory)

CLIQUE ∈ NPC

- CLIQUE ∈ NP
  - Pick k vertices from V (call it V'), treat it as a certificate for G. Checking if V' is a clique is $O(n^2)$.
- 3-CNF-SAT $\leq_P$ CLIQUE
  - Given an arbitrary 3-CNF Boolean formula ϕ (with k clauses),
  - Construct a graph G (in poly-time and size) that satisfies:
    - G will have a k-clique if and only ϕ is satisfiable.
- Ingenious trick to construct such a graph.
  - Will just see an example here.

3-CNF-SAT $\leq_P$ CLIQUE

- For $\phi = C_1 \wedge C_2 \wedge C_3$ where $C_1 = x_1 \vee \neg x_2 \vee \neg x_3$, $C_2 = \neg x_1 \vee x_2 \vee x_3$, and $C_3 = x_1 \vee x_2 \vee x_3$, construct a graph G as follows (CLRS Fig. 34.14):
- See ϕ is satisfiable if and only if G has a 3-clique!
- Also note that G can be built in poly-time and size!


$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$
x_1
$\neg x_2$
$\neg x_3$
$C_2 = \neg x_1 \vee x_2 \vee x_3$
$\neg x_1$
x_2
x_3
$C_3 = x_1 \vee x_2 \vee x_3$
x_1
x_2
x_3

Summary

- You might encounter in your career some problems that are pretty hard to solve exactly (optimally).
    - E.g., travelling salesperson problem, general scheduling, …
- Consider checking if those problems are NP-complete.
- If so, you won’t likely find an algorithm that gives exact (optimal) solutions.
    - Better not try to find one.
    - Or if you find one, you win the great fame and the million-dollar prize.
- Time to investigate approximation algorithms for those problems
    - CLRS Ch. 35
    - Beyond scope of our course