**Topic 23 (continued): NP-Completeness**

(CLSS 34)

CPS 230, Fall 2001

The theory of NP-completeness involves much formalism.

Basic idea:
Some problems (seem) much harder than others
Want to abstract, talk about *classes* of problems
Class we’ve seen: $P$ (polynomial-time problems)
New class: NP-Complete
(seemingly) super-poly-time problems
E.g., max-clique, $k$-clique, $k$-coloring, TSP, ...
Hundreds of others (Garey and Johnson)

Situation:
No one can give a poly-time alg. for these, but...
No one can show a super-poly lower bound, either!
(Most would be surprised by poly-time alg.)

Complexity theory addresses *decision problems*
Simplest output: a single bit, yes or no
(OK, because can recast optimization as decision)

We want to make statements like the following
for decision problems B and C:

If problem B has a polynomial-time algorithm,
then so does C.

If problem C has no polynomial-time algorithm,
then neither does B.

---

**Encodings & Formal Languages**

Encodings
Must agree on concrete representation of input
in order to relate running time to input size
Just have to be careful that encoding is reasonable
(i.e., poly-size in all other reasonable encodings)

Formal Languages
A language $L$ is any set of strings
made from an alphabet
Take simplest interesting alphabet, $\Sigma = \{0, 1\}$

Decision problems and formal languages
$\Sigma^*$ comprises all possible input instances
Decision problem is characterized by those strings
that produce 1.
View this set of strings as a formal language $L$.

Decision problems, Accepted, Decided

Decision problems and algorithms that solve them:
Inputs are simply arbitrary strings fed to routines.
Algorithm *accepts* string if it produces 1
Algorithm *rejects* string if it produces 0
But... it might loop forever.

Decides:
Algorithm *decides* language $L$ if every
binary string is either accepted or rejected

What’s the difference?
To accept a language, the algorithm needs only
“worry” about outputting 1 when it sees $x \in L$
If it sees a string $x \notin L$, it can loop forever.

To decide a language, algorithm must accept
or reject *every* string in $\{0, 1\}^*$.

There are some problems (e.g. halting problem)
that have an accepting, but no deciding, algorithm
Problem/Language Classes

Class $P$ consists of those languages for which some algorithm $A$ decides $L$ in polynomial time.

Class $NP$ consists of those languages for which a solution can be verified in polynomial time. This involves a certificate or witness to solution (which need not be unique)

(If no solution, then no certificate!)

Fundamental open question: does $P = NP$?

Most believe no; that is, that there are languages (i.e., decision problems) in $NP$, but not in $P$.

The GRAPH ISOMORPHISM problem (given two graphs, is there a permutation of the vertices of one graph that makes the two equivalent?) is in $NP$ but is not known to be either NP-complete or in $P$.

Complexity Theory

Recall:

decision problem = language = subset of $\{0,1\}^*$

$P = \{ \text{poly-time decidable languages} \}$

We say $L \in P$ if $\exists$ poly-time alg. $A$ such that

$L = \{ x \in \{0,1\}^* : A(x) = 1 \}$

$NP = \{ \text{poly-time verifiable languages} \}$

We say $L \in NP$ if $\exists 2$-arg. poly-time alg. $A$ so that

$L = \{ x \in \{0,1\}^* : \exists y, |y| = \text{poly}(|x|) \text{ such that } A(x,y) = 1 \}$

Reducibility

**Definition.** Let $L_1, L_2 \subseteq \{0,1\}^*$.

Then $L_1$ is **polynomial-time reducible** to $L_2$, written $L_1 \leq_p L_2$, if $\exists$ poly-time computable function

$f : \{0,1\}^* \rightarrow \{0,1\}^*$

such that $\forall x \in \{0,1\}^*$,

$x \in L_1$ if and only if $f(x) \in L_2$.

Any problem instance for $L_1$ can be easily rephrased as a problem instance for $L_2$.

$L_1 \leq_p L_2$ implies the following:

We can use $L_2$ to solve $L_1$!

Solving $L_1$ is no harder than solving $L_2$!

Solving $L_2$ is at least as hard as solving $L_1$!

By “no harder” and “at least as hard”, we mean “up to a poly-time factor in the running time”.
Poly-time reduction

\begin{align*}
\{0,1\}^* & \quad \{0,1\}^* \\
L_1 \xrightarrow{f} & \quad L_2
\end{align*}

\[ x \in L_1 \iff f(x) \in L_2. \]

**Lemma:**
If \( L_1 \leq_P L_2 \) and \( L_2 \in P \), then \( L_1 \in P \).

**Proof:** Let \( F \) be poly-time alg. that computes \( f \), and let \( A_2 \) be poly-time alg. deciding \( L_2 \).

We now construct poly-time \( A_1 \) deciding \( L_1 \):

**Claim:** \( A_1 \) runs in polynomial time.

**Proof:**
\( F \) runs in \( O(n^a) \) time on input of length \( n \).
\( A_2 \) runs in \( O(n^b) \) time on input of length \( n \).
\[ |f(x)| = O(|x|^a) \]
\[ A_2 \text{ runs in } O(|f(x)|^b) = O(|x|^{ab}) \text{ time} \]
\[ \implies A_1 \text{ runs in } O(|x|^{ab}) \text{ time (i.e., poly-time).} \]

**NP-completeness**

**Definition.** \( L \) is NP-complete if
1. \( L \in NP \)
2. \( \forall L' \in NP, L' \leq_P L \).

NP-complete problems are hardest in NP. (If only 2. holds, problem is called “NP-hard.”)

**Theorem:**
\( L \in NPC \text{ and } L \in P \implies P=NP. \)

**Proof:**
\( \forall L' \in NP, \text{ we have } L' \leq_P L. \)

Lemma and \( L \in P \implies L' \in P. \)
Circuit satisfiability

A boolean combinational circuit:

\[ x_1 \rightarrow \neg x_2 \rightarrow x_3 \]

Decision problem:

Is there an assignment to the input that makes the circuit evaluate to TRUE?

CKT-SAT = \{\langle \text{CKT} \rangle: \text{CKT has satisfying assignment} \}

What is running time of the naive algorithm?

Theorem:

CKT-SAT is NP-complete.

Proof: Part 1. CKT-SAT \( \in \) NP.

\( A(\langle \text{CKT} \rangle, \langle \text{state of all wires in satisfying assignment} \rangle) \)

- \( A \) checks that wire values are correctly computed
- if output = 1, then \( A \) outputs 1
  certificate size is \( \text{poly}(|\langle \text{CKT} \rangle|) \)
  \( \implies A \) is poly-time.

Part 2. CKT-SAT is NP-hard.

Proof is involved; uses a simulation argument. For complete details, see CLRS §34.3.

Proving CKT-SAT NP-Complete: The Hard Way

Let \( L \) be any language in NP.

We will construct poly-time algorithm \( F \) that computes function \( f \) such that \( x \in L \) iff \( f(x) \in \text{CKT-SAT} \).

Let \( A \) be a poly-time algorithm that recognizes \( L \).

Let \( T(n) \) be the running time of \( A \).

\( T(n) = O(n^k) \), for some \( k \).

Basic idea of proof: We represent \( A \)'s computation as a sequence of configurations.

Config. \( t \) (at time \( t \)) represents (in binary format)
1. Algorithm \( A \),
2. Program counter PC,
3. State of all variables of \( A \),
4. Working storage,
5. Input \( x \) to \( A \),
6. Certificate \( y \),

We construct a combinatorial circuit \( M \) that takes configuration \( t \) as input and produces configuration \( t + 1 \) as output.

We can simulate \( A \) by pasting together \( T(n) \) copies of \( M \).

We call the resulting circuit \( C \). See CLRS Figure 34.9.

In the initial configuration, we “hardwire” the values of items 1–5 to the appropriate values.

The output of the resulting circuit \( C \) is set to the special variable or memory location corresponding to the output of \( A \).
Circuit $C$:

![Circuit Diagram]

Proving CKT-SAT NP-Complete: The Hard Way

**Lemma:** The reduction $F$ runs in polynomial time as a function of $|x| = n$.

**Proof:** Consider any string $x$ of length $n$.

- The input to the circuit $C$ has polynomial size:
  1. Algorithm $A$ can be encoded in $O(1)$ bits.
  2. Program $A$ can be encoded in $O(\log n)$ bits.
  3. State of variables $A$ can be encoded in $O(n^k)$ bits.
  4. Working storage can be encoded in $O(n^k)$ bits.
  5. Input $x$ to $A$ can be encoded in $n$ bits.
  6. Certificate $y$ can be encoded in $O(n^k)$ bits.

The circuit $M$ has size polynomial in the configuration length, which is therefore polynomial in $n$.

The number of copies of $M$ in $C$ is $T(n) = O(n^k)$.

Therefore, the entire construction of the circuit $C$ can be done in polynomial time.

Proving CKT-SAT NP-Complete: The Hard Way

Two remaining things to prove in order to show that CKT-SAT is NP-Complete:

1. $F$ correctly computes a reduction function $f$.
2. $F$ runs in polynomial time.

**Lemma:** Given any string $x$, circuit $C$ is satisfiable if and only if there is a poly certificate $y$ such that $A(x, y) = 1$.

**Proof:** Consider any string $x$ of length $n$.

If there is a certificate $y$ of length $O(n^k)$ such that $A(x, y) = 1$, then if we apply the bits of $y$ to the inputs of $C$ (i.e., configuration 0), the output of $C$ is 1 and thus $C$ is satisfiable.

If $C$ is satisfiable, then there is some setting for $y$ that makes the circuit produce 1.

Since $C$ is simulating $A$, it follows that $A(x, y) = 1$.

Proving problems NP-Complete: The Easy Way

Now that we have established a problem to be NP-Complete (namely, CKT-SAT), we have an easier way to prove other problems to be NP-Complete.

**General recipe:**

To show that $L \in$ NPC:

1. Show that $L \in$ NP.
2. Pick some $L' \in$ NPC.
   - Show that $L' \leq_p L$.

By definition of NP-Complete, since $L' \in$ NPC, then $\forall L'' \in$ NP, $L'' \leq_p L' \leq_p L$ and thus $L'' \leq_p L$.

Therefore, $L$ satisfies the definition of NP-Complete.

**Bottom line:** Since we can use an algorithm for $L$ to solve problem $L'$ that we know to be NP-complete, which in turn can be used to solve any NP problem $L''$, then we can conclude that $L$ is itself NP-complete.
**Formula satisfiability problem**

A boolean formula:
\[ \phi = ((x_1 \to x_2) \land (\overline{x_1} \lor x_2 \lor x_3)) \to (x_1 \land \overline{x_2}) \]

Note: We use \( \overline{x} \) or \( -x \) to represent the negation of \( x \).

\[ \text{SAT} = \{ \langle \phi \rangle : \phi \text{ has a satisfying assignment} \} \]

**Theorem:** SAT is NP-complete.

**Proof:** Part 1. SAT \( \in \) NP.
Certificate is the “truth assignment”.
Algorithm merely has to verify, in polynomial time, that the truth assignment produces TRUE.

Part 2. CKT-SAT \( \leq_p \) SAT.

Given \( \langle \text{CKT} \rangle \), produce formula \( \langle \phi \rangle = f(\langle \text{CKT} \rangle) \)
in poly time such that
\[ \langle \text{CKT} \rangle \in \text{CKT-SAT} \iff \langle \phi \rangle \in \text{SAT}. \]

---

**3CNF-SAT (a.k.a. 3SAT)**

**Definitions.**

A literal is \( x_i \) or \( \overline{x_i} \).

A clause is \( l_1 \lor l_2 \lor \ldots \lor l_k \), where \( l_i = \) literal.

A formula is in conjunctive normal form if it has form \( c_1 \land c_2 \land \ldots \land c_r \), where \( c_i = \) clause.

3CNF-SAT = \{ \langle \phi \rangle : \phi \text{ is a satisfiable formula in cnf with 3 literals per clause} \}

3CNF-SAT is a special case of SAT.
The instances of 3CNF-SAT are those instances of SAT that are in conjunctive normal form (i.e., AND of ORs) and such that there are three literals per clause.

Note: If \( L \) is a special case of \( L' \), then
\[ L \leq_p L' \]
(i.e., \( L' \) is at least as hard as \( L \))

---

**Proving 3CNF-SAT NP-Complete**

**Theorem:** 3CNF-SAT is NP-Complete.

**Proof:**

Part 1. Easy; the certificate is the “truth assignment”.
In fact, we know automatically that 3CNF-SAT \( \in \) NP
since the more general problem of SAT is in NP.

Part 2. SAT \( \leq_p \) 3CNF-SAT.

Starting with instance \( \phi \) of SAT,
we need to generate in polynomial time
an instance \( \phi' \) of 3CNF-SAT so that
\( \phi \) is satisfiable iff \( \phi' \) is satisfiable.
**Proving 3CNF-SAT NP-Complete**

**Step 1.** Think of $\phi$ as a boolean expression and represent it as a binary tree. Each node is an operation that gets the input from its two children and forwards the output to its parent.

Clauses of many literals, such as $(x_1 \lor \overline{x_2} \lor x_3 \lor x_4)$ can be represented in binary form as $(((x_1 \lor \overline{x_2}) \lor x_3) \lor x_4)$.

Introduce a new variable for the output and define a new formula $\phi'$ that for each node relates the two input edges with the one output edge.

**Proving 3CNF-SAT NP-Complete**

For example, if $\phi$ is the formula $(x_1 \rightarrow x_2) \leftrightarrow (x_2 \lor \overline{x_1})$, its binary tree representation is

![Binary Tree Diagram]

and the equivalent formula with three literals per clause is

$$\phi' = (y_2 \leftrightarrow (x_1 \rightarrow x_2))$$
$$\land (y_3 \leftrightarrow (x_2 \lor \overline{x_1}))$$
$$\land (y_1 \leftrightarrow (y_2 \leftrightarrow y_3))$$
$$\land y_1.$$

It should be clear that $\phi$ is satisfiable if and only if $\phi'$ is satisfiable.

**Step 2.** Convert each clause to disjunctive normal form.

We can use the truth table for each clause. For example, the truth table for $y_2 \leftrightarrow (x_1 \rightarrow x_2)$ is

<table>
<thead>
<tr>
<th>$y_2$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$(y_2 \leftrightarrow (x_1 \rightarrow x_2))$</th>
<th>prohibited</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\overline{y_2} \land \overline{x_1} \land \overline{x_2}$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\lor (\overline{y_2} \land \overline{x_1} \land x_2)$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$\lor (\overline{y_2} \land x_1 \land x_2)$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\lor (y_2 \land x_1 \land x_2)$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\lor (y_2 \land x_1 \land x_2)$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\lor (y_2 \land x_1 \land x_2)$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$\lor (y_2 \land x_1 \land x_2)$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$\lor (y_2 \land x_1 \land x_2)$</td>
</tr>
</tbody>
</table>

The negation of $y_2 \leftrightarrow (x_1 \rightarrow x_2)$ is equivalent to the disjunction of the conjunctions given in the “prohibited” column.

Therefore, $y_2 \leftrightarrow (x_1 \rightarrow x_2)$ is equivalent to the negation of that disjunction of conjunctions, which by de Morgan’s law is $(y_2 \lor x_1 \lor x_2) \land (y_2 \lor x_1 \lor \overline{x_2}) \land (y_2 \lor \overline{x_1} \lor x_2) \land (y_2 \lor \overline{x_1} \lor \overline{x_2})$.

**Proving 3CNF-SAT NP-Complete**

**Step 3.** The clauses with fewer than three literals can be expanded to three literals by adding new variables.

For example we expand the two-literall clause $a \lor b$ to $(a \lor b \lor p) \land (a \lor b \lor \overline{p})$.

We expand the singleton clause $(a)$ to $(a \lor p \lor q) \land (a \lor p \lor \overline{q}) \land (a \lor \overline{p} \lor q) \land (a \lor \overline{p} \lor \overline{q})$.

Each of Steps 1, 2, and 3 takes only polynomial time, and we end up with an equivalent formula in 3-conjunctive normal form.

Therefore, $\text{SAT} \leq_p 3\text{CNF-SAT}$. This completes the proof that 3CNF-SAT is NP-Complete.
**Proving CLIQUE NP-Complete**

**Theorem:** CLIQUE is NP-Complete.

**Proof:**

1. **Part 1.** Easy; the certificate is the list of vertices in the clique.

2. **Part 2.** We will prove 3CNF-SAT ≤ₚ CLIQUE.

That is, we want to show that a poly-time algorithm for CLIQUE would give us a poly-time algorithm for 3CNF-SAT.

So... starting with instance φ of 3CNF-SAT, we need to generate an instance of CLIQUE (namely, \( \langle G, k \rangle \)) in polynomial time so that φ is satisfiable iff G has a clique of size k.

**Example:**

\[
\phi = (x_1 \lor \overline{x}_2 \lor \overline{x}_3)(\overline{x}_1 \lor x_2 \lor x_3)(x_1 \lor x_2 \lor x_3)
\]

\[
C_1 = x_1 \lor \neg x_2 \lor \neg x_3
\]

\[
C_2 = \neg x_1 \lor x_2 \lor x_3
\]

\[
C_3 = x_1 \lor x_2 \lor x_3
\]

\[
C_4 = \neg x_1 \lor \overline{x}_2 \lor \overline{x}_3
\]

**Claim:** φ is satisfiable iff G has a clique of size c.

**Proof:**

( \( \implies \) ) If φ is satisfiable, then at least one literal in each clause is TRUE, and the corresponding vertices form a clique of size c.

( \( \iff \) ) If G has a clique of size c then

1. It must have exactly one vertex per triple.
2. It does not contain vertices with inconsistent labels. By assigning TRUE to each corresponding literal in the clique, we make φ satisfiable.

G is computable from φ in poly-time.
Other NP-Complete Problems

The INDEPENDENT-SET problem \( \langle G, k \rangle \) asks if the graph \( G \) has an independent set of size at least \( k \) (i.e., \( k \) vertices such that no two of them are neighbors).

**Theorem:** INDEPENDENT-SET \( \in \) NPC.

**Proof:** Part 1. The certificate represents the \( k \) independent vertices. The verification algorithm merely has to check that there are no edges connecting any two.

**Part 2.** CLIQUE \( \leq_p \) INDEPENDENT-SET.

\( W \subseteq V \) is independent iff \( W \) defines a clique in the complement graph \( \overline{G} = (V, (V/2) - E) \).

We transform instance \( \langle H, k \rangle \) of CLIQUE problem to instance \( \langle \overline{H}, k \rangle \) of INDEPENDENT-SET

\( \overline{H} \) has an independent set of size \( k \) or larger iff \( H \) has a clique of size \( k \) or larger.

---

Other NP-Complete Problems

The VERTEX-COVER problem \( \langle G, k \rangle \) asks if the graph \( G \) has a vertex cover of size \( k \) or less (i.e., whether there are \( k \) vertices such that every edge has at least one of them as an endpoint).

**Theorem:** VERTEX-COVER \( \in \) NPC.

**Proof:** Part 1. The certificate represents the \( k \) vertices in the vertex cover. The verification algorithm merely has to check that each edge in the graph has at least one endpoint among the \( k \) vertices.

**Part 2.** INDEPENDENT-SET \( \leq_p \) CLIQUE.

\( W \) is a vertex cover iff \( V - W \) is independent.
We transform instance \( \langle G, k \rangle \) of INDEPENDENT-SET to the instance \( \langle G, n - k \rangle \) of VERTEX-COVER.

\( G \) has an independent set of size \( k \) or larger iff it has a vertex cover of size \( n - k \) or smaller.

---

**Illustration of the reductions**

In the graph on the left, the four shaded vertices form an independent set.

In the complement graph on the right, the four shaded vertices form a clique.

In the graph on the left, the four white vertices form a vertex cover.