NP-Completeness

P: is the set of decision problems (or languages) that are solvable in polynomial time.
NP: is the set of decision problems (or languages) that can be verified in polynomial time.

Polynomial reduction: \( L_1 \preceq_p L_2 \) means that there is a polynomial time computable function \( f \) such that \( x \in L_1 \) if and only if \( f(x) \in L_2 \). A more intuitive to think about this, is that if we had a subroutine to solve \( L_2 \) in polynomial time, then we could use it to solve \( L_1 \) in polynomial time.

Lemma: If \( L_1 \preceq_p L_2 \) and \( L_2 \in P \) then \( L_1 \in P \).
Lemma: If \( L_1 \preceq_p L_2 \) and \( L_1 \notin P \) then \( L_2 \notin P \).

• Polynomial reductions are transitive, that is, if \( L_1 \preceq_p L_2 \) and \( L_2 \preceq_p L_3 \), then \( L_1 \preceq_p L_3 \).

NP-Hard: \( L \) is NP-hard if for all \( L' \in NP \), \( L' \preceq_p L \). Thus, if we could solve \( L \) in polynomial time, we could solve all NP problems in polynomial time.

NP-Complete: \( L \) is NP-complete if (1) \( L \in NP \) and (2) \( L \) is NP-hard.

• The importance of NP-complete problems should now be clear. If any NP-complete problem (and generally any NP-hard problem) is solvable in polynomial time, then every NP-complete problem (and in fact every problem in NP) is also solvable in polynomial time.

• Conversely, if we can prove that any NP-complete problem cannot be solved in polynomial time, then every NP-complete problem (and generally every NP-hard problem) cannot be solved in polynomial time.

• Thus all NP-complete problems are equivalent to one another (in that they are either all solvable in polynomial time, or none are).

Theory of Computation, Feodor F. Dragan, Kent State University

NP-Completeness (cont.)

• The figure below illustrates one way that the sets \( P \), \( NP \), \( NP\text{-hard} \), and \( NP\text{-complete} \) (NPC) might look. We say might because we do not know whether all of these complexity classes are distinct or whether they are all solvable in polynomial time.

• One is Graph Isomorphism, which asks whether two graphs are identical up to a renaming of their vertices. It is known that this problem is in \( NP \), but it is not known to be in \( P \).

• The other is QBF, which stands for Quantified Boolean Formulas. In this problem you are given a Boolean formula with quantifiers (\( \exists \) and \( \forall \)) and you want to know whether the formula is true or false.

• An alternative way to show that a problem is NPC is to use transitivity of \( \preceq_p \).

Lemma: \( L \) is \( NP\text{-complete} \) if (1) \( L \in NP \) and (2) \( L' \preceq_p L \) for some \( NP\text{-complete} \) language \( L' \).

Note: The known \( NP\text{-complete} \) problem \( L' \) is being reduced to candidate \( NP\text{-complete} \) problem \( L \). Keep this order in mind.

Theory of Computation, Feodor F. Dragan, Kent State University
Cook-Levin's Theorem and Reductions

- Unfortunately, to use this lemma, we need to have at least one NP-complete problem to start the ball rolling. Stephen Cook and Leonid Levin showed (≈ 1970) that such a problem existed. Cook-Levin's theorem is rather complicated to prove. First we'll try to give a brief intuitive argument as to why such a problem might exist.
- For a problem to be in NP, it must have an efficient verification procedure.
- Virtually all NP problems can be stated in the form, "does there exist X such that P(X)?" where X is some structure (e.g. a set, a path, a partition, an assignment, etc.) and P(X) is some property that X must satisfy (e.g. the set of objects must fill the knapsack, or the path must visit every vertex, or you may use at most k colors and no two adjacent vertices can have the same color).
- In showing that such a problem is in NP, the certificate consists of giving X, and the verification involves testing that P(X) holds.
- In general, any set X can be described by choosing a set of objects, which in turn can be described as choosing the values of some Boolean variables.
- Similarly, the property P(X) that you need to satisfy, can be described as a Boolean formula.
- Cook and Levin were looking for the most general possible property he could, since this should represent the hardest problem in NP to solve.
- They reasoned that computers (which represent the most general type of computational devices known) could be described entirely in terms of Boolean circuits, and hence in terms of Boolean formulas.
- If any problem were hard to solve, it would be one in which X is an assignment of Boolean values (true/false, 0/1) and P(X) could be any Boolean formula. This suggests the following problem, called the Boolean satisfiability problem.

Boolean Satisfiability Problem

SAT: Given a Boolean formula, is there some way to assign truth values (0/1, true/false) to the variables of the formula, so that the formula evaluates to true?

- A Boolean formula is a logical formula which consists of variables \( x_i \), and the logical operations \( \overline{x} \) meaning the negation of \( x \), Boolean-or \( (x \vee y) \) and Boolean-and \( (x \wedge y) \).

- Given a Boolean formula, we say that it is satisfiable if there is a way to assign truth values (0 or 1) to the variables such that the final result is 1. (As opposed to the case where no matter how you assign truth values the result is always 0.)

For example, \( (x_1 \wedge (x_2 \vee \overline{x}_3)) \wedge ((\overline{x}_2 \wedge \overline{x}_4) \vee \overline{x}_4) \)

is satisfiable, by the assignment \( x_1 = 1, x_2 = 0, x_3 = 0. \) On the other hand,

\( (\overline{x}_1 \vee (x_2 \wedge x_3)) \wedge (x_1 \vee (\overline{x}_2 \wedge \overline{x}_3)) \wedge (x_2 \vee x_3) \wedge (\overline{x}_2 \vee \overline{x}_3) \)

is not satisfiable. (Observe that the last two clauses imply that one of \( x_2 \) and \( x_3 \) must be true and the other must be false. This implies that neither of the subclauses involving \( x_2 \) and \( x_3 \) in the first two clauses can be satisfied, but \( x_1 \) cannot be set to satisfy them either.)

Cook-Levin's Theorem: SAT is NP-complete.

Proof will be given in class.
THE COOK-LEVIN THEOREM

**Theorem:** \( SAT \) is \( NP \)-complete.

**Proof** First, we show that \( SAT \) is in \( NP \). A nondeterministic polynomial time machine can guess an assignment to a given formula \( \phi \) and accept if the assignment satisfies \( \phi \).

Next, we take any language \( A \) in \( NP \) and show that \( A \) is polynomially reducible to \( SAT \).

The reduction for \( A \) takes a string \( w \) and produces a Boolean formula \( \phi \) that simulates the \( NP \) machine for \( A \) on input \( w \). If the machine accepts, \( \phi \) has a satisfying assignment that corresponds to the accepting computation. If the machine doesn't accept, no assignment satisfies \( \phi \). Therefore, \( w \) is in \( A \) if and only if \( \phi \) is satisfiable.

Let \( N \) be a nondeterministic Turing machine that decides \( A \) in \( n^k \) time for some constant \( k \).

---

**A tableau for** \( N \) on \( w \) is an \( n^k \times n^k \) table whose rows are the configurations of a branch of the computation of \( N \) on input \( w \), as shown in the following figure. For convenience later we assume that each configuration starts and ends with a \( \# \) symbol, so the first and last columns of a tableau are all \( \# \). The first row of the tableau is the starting configuration of \( N \) on \( w \), and each row follows the previous one according to \( N \)'s transition function. A tableau is accepting if any row of the tableau is an accepting configuration.

**Figure**

Every accepting tableau for \( N \) on \( w \) corresponds to a computation branch of \( N \) on \( w \). Thus the problem of determining whether \( N \) accepts \( w \) is equivalent to the problem of determining whether an accepting tableau for \( N \) on \( w \) exists.
Reduction $f$ from A to SAT

- Variables:
  \[ x_{ij} \in \{1, \ldots, n\}^{+}, \quad s \in C = \{0, 1\} \]
  \[ x_{ij} = \begin{cases} 1, & \text{if } c_{ij} \in S \\ 0, & \text{otherwise} \end{cases} \]

- We design $\phi$ so that a satisfying assignment to the variables does correspond to an accepting tableau for $N$ on $w$.

- We must guarantee, in order to obtain a correspondence between an assignment and a tableau, that the assignment turns on exactly one variable for each cell.

- \( \phi_{\text{start}} \) ensures this:
  \[ \phi_{\text{start}} = \bigwedge_{i \leq j \leq n} \left( \bigvee_{k \leq n} x_{i,k} \right) \land \left( \bigwedge_{i \leq j \leq n} \left[ \bigwedge_{k \leq n} \left( \bigvee_{l \leq m} x_{l,j} \right) \right] \right) \]

- \( \forall x_{i,j} \) is shorthand for:
  \[ x_{1,1}, x_{2,2}, \ldots, x_{i,j}, \ldots, x_{n,n} \]

Reduction $f$ from A to SAT (cont.)

- Parts $\phi_{\text{start}}, \phi_{\text{move}}, \phi_{\text{accept}}$ ensure that the table is actually an accepting tableau.

Formula $\phi_{\text{start}}$ ensures that the first row of the table is the starting configuration of $N$ on $w$ by explicitly stipulating that the corresponding variables are on:

- $\phi_{\text{start}} = x_{1,1,1} \land x_{1,1,0} \land x_{1,1,1} \land x_{1,1,0} \land \cdots \land x_{1,1,0} \land x_{1,1,1}$

Formula $\phi_{\text{accept}}$ guarantees that an accepting configuration occurs in the tableau. It ensures that $\phi_{\text{accept}}$, the symbol for the accept state, appears in one of the cells of the tableau, by stipulating that one of the corresponding variables is on:

\[ \phi_{\text{accept}} = \bigvee_{i \leq j \leq n} x_{i,j} \cdot \text{accept} \]
Reduction $f$ from A to SAT (cont.)

Finally, formula $\phi_{\text{more}}$ guarantees that each row of the table corresponds to a configuration that legally follows the preceding row’s configuration according to $\mathcal{N}$’s rules. It does so by ensuring that each $2 \times 3$ window of cells is legal. We say that a $2 \times 3$ window is legal if that window does not violate the actions specified by $\mathcal{N}$’s transition function. In other words, a window is legal if it might appear when one configuration correctly follows another.

Assume $\mathcal{N}$ has $\delta(q_1, a) = \{(q_1, b, R)\}$ and $\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$.

Examples of legal windows for this machine are shown in the following figure.

**FIGURE**
Examples of legal windows

(a) \[
\begin{array}{ccc}
    a & q_1 & b \\
    q_1 & a & c \\
\end{array}
\]
(b) \[
\begin{array}{ccc}
    a & q_1 & b \\
    a & a & q_1 \\
\end{array}
\]
(c) \[
\begin{array}{ccc}
    a & a & q_1 \\
    a & a & b \\
\end{array}
\]

**FIGURE**
Examples of illegal windows

(d) \[
\begin{array}{ccc}
    \# & b & a \\
    \# & b & a \\
\end{array}
\]
(e) \[
\begin{array}{ccc}
    a & b & a \\
    a & b & q_1 \\
\end{array}
\]
(f) \[
\begin{array}{ccc}
    b & b & b \\
    c & b & b \\
\end{array}
\]

Reduction $f$ from A to SAT (cont.)

**CLAIM** If the top row of the table is the start configuration and every window in the table is legal, each row of the table is a configuration that legally follows the preceding one.

- $\phi_{\text{more}}$ stipulates that all the windows in the tableau are legal

$\phi_{\text{more}} = \bigwedge_{1 \leq i \leq n^b, 1 \leq j \leq n^h} (\text{the } (i, j) \text{ window is legal})$

We replace the text "the $(i, j)$ window is legal" in this formula with the following formula. We write the contents of six cells of a window as $a_1, \ldots, a_6$.

$$\bigvee_{\phi_{a_1, \ldots, a_6} \text{ is a legal window}} (x_{i,j-1,a_1} \land x_{i,j,a_2} \land x_{i,j+1,a_3} \land x_{i+1,j-1,a_4} \land x_{i+1,j,a_5} \land x_{i+1,j+1,a_6}).$$
Complexity of the Reduction $f$

- Complexity $O(n^{2^k})$.
  - $n^{2^k}$ cells
  - $2^k$ variables associated with each cell
  - $e$ depends only on $N$, not on $n = 1 + e$
  - $O(n^{2^k})$ variables
  - Each cell has size $O(n^{2^k})$
  - Each ant has size $O(n^k)$
  - Each domino, template have size $O(n^{2^k})$

3-Conjunctive Normal Form (3-CNF)

- In fact, it turns out that a even more restricted version of the satisfiability problem is NP-complete.
- A literal is a variable or its negation, $x$ or $\overline{x}$.
- A formula is in 3-conjunctive normal form (3-CNF) if it is the Boolean-and of clauses where each clause is the Boolean-or of exactly 3 literals.
- For example, $(x_1 \lor x_2 \lor \overline{x}_3) \land (\overline{x}_1 \lor x_3 \lor x_4) \land (x_2 \lor x_3 \lor \overline{x}_4)$ is in 3-CNF form.
- 3SAT is the problem of determining whether a formula in 3-CNF is satisfiable.
- It turns out that it is possible to modify the proof of Cook’s theorem to show that 3SAT is also NP-complete.
- As an aside, note that if we replace the 3 in 3SAT with a 2, then everything changes. If a Boolean formula is given in 2SAT, then it is possible to determine its satisfiability in polynomial time. (It turns out that the problem can be reduced to computing the strong components in a directed graph.)
- Thus, even a seemingly small change can be the difference between an efficient algorithm and none.
\[ \text{SAT} \rightarrow 3\text{SAT} \]

- Use distributive laws
  \[ x \land (y \lor z) = (x \land y) \lor (x \land z) \]
  \[ x \lor (y \land z) = (x \lor y) \land (x \lor z) \]
- \( \phi \) is already almost in CNF
  - \( \phi \) is in CNF
  - \( \phi \) is start in CNF (many clauses, each in a single)
  - \( \phi \) is accept in CNF (1 clause)
  - \( \phi \) is more \( \rightarrow \) CNF using distributive law

- Thus, \( \phi' \) is in CNF
  - In each clause that has 1 or 2 literals, replicate one of the literals to get 3 literals.
  - In each clause that has 3 literals, split it into several clauses using additional variables.

\[ (a_1 \lor a_2 \lor a_3) \rightarrow (a_1 \lor a_2 \lor z) \land (z \lor a_3) \]

\[ (a_1 \lor a_2 \lor \cdots \lor a_l) \quad \ell \text{ variables} \]
\[ \downarrow \quad \ell-2 \text{ clauses} \]
\[ (a_1 \lor a_2 \lor z_1) \land (z_1 \lor a_3 \lor z_2) \land (z_2 \lor a_4 \lor z_3) \land \cdots \land (z_{l-3} \lor a_{l-1} \lor a_l). \]

**More NP-completeness proofs**

Now that we know that 3SAT is NP-complete, we can use this fact to prove that other problems are NP-complete.