Recall that a language $L$ is **NP-complete** if it is in NP and is NP-hard:

1. $L \in$ NP if there exists a poly-time verifier $V$ such that $x \in L$ if and only if $V(x,w)$ accepts. Equivalently, $L \in$ NP if there exists a poly-time NTM $N$ such that $L = L(N)$.

2. $L$ is NP-hard if for all $B \in$ NP, $B \leq_p L$. Note that in general, $A \leq_p B$ if there exists a poly-time computable function $f$ such that for all $x$, $x \in A$ if and only if $f(x) \in B$.

Some examples of NP-complete languages (which we will prove to be NP-complete) are:

- **SAT** = \{ $\phi(x_1, \ldots, x_n)$ is a Boolean formula such that there exists an assignment $a_1, \ldots, a_n$ such that $\phi(a_1, \ldots, a_n) = \text{‘true’}$ \}

- **3SAT** = \{ $\phi(x_1, \ldots, x_n)$ is a 3CNF Boolean formula such that there exists an assignment $a_1, \ldots, a_n$ such that $\phi(a_1, \ldots, a_n) = \text{‘true’}$ \}

- **CLIQUE** = \{ $(G,k)$ | $G$ is a graph that has a $k$-clique \}

**Topics Covered**

1. NP-Complete Reductions
2. The Cook-Levin Theorem

### 1 NP-Complete Reductions

**Lemma** If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.

**Proof** Let $f$ be the poly-time computable function such that $x \in A$ if and only if $f(x) \in B$. Let $g$ be the poly-time computable function such that $y \in B$ if and only if $g(y) \in C$. Define the function $h(x) = g(f(x))$. Then we have $x \in A \iff f(x) \in B \iff g(f(x)) \in C$. Thus $h$ is a poly-time function such that $x \in A$ if and only if $h(x) \in C$, implying that $A \leq_p C$.

**Theorem** If $L$ is NP-complete and $A \in$ NP such that $L \leq_p A$, then $A$ is NP-complete.
Proof To be NP-complete, \( A \) must (1) be in NP and (2) be NP-hard. Note that (1) is given. For (2), consider any language \( B \in \text{NP} \). Then \( B \leq_p L \) because \( L \) is NP-complete, and \( L \leq_p A \). By the lemma, \( B \leq_p A \). This holds for any \( B \in \text{NP} \), so \( A \) is NP-hard. As a result of (1) and (2), \( A \) is NP-complete.

Proof that 3SAT is NP-Complete The language 3SAT consists of 3CNF Boolean formulas such that a satisfying variable assignment exists. Recall that a 3CNF formula consists of clauses joined by conjunctions ('and'), where each clause consists of three literals joined by disjunctions ('or'). For example, \((x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3 \lor x_4) \land (x_3 \lor x_2 \lor x_5)\) is a 3CNF formula.

To show that 3SAT is NP-complete, we must show that it is (1) in NP and (2) NP-hard. Last class, we showed that 3SAT is in NP. To show that it is NP-hard, we will construct a reduction in which SAT \( \leq_p \) 3SAT. Since SAT is NP-complete (as we will later show), it follows that 3SAT is NP-hard. Consider the following poly-time computable function \( f \) where \( \langle \phi \rangle \in \text{SAT} \) if and only if \( f(\langle \phi \rangle) \in \text{3SAT} \):

\[
\begin{align*}
f, \text{ on input } \langle \phi \rangle: \\
1. & \text{Construct a tree corresponding to } \phi. \\
2. & \text{For each internal node } \phi_i \text{ of the tree, create a new variable } z_i. \text{ Let } \ell \text{ be the number of internal nodes.} \\
3. & \text{Construct } \phi' = \bigwedge_{i=1}^{\ell} (z_i \iff \text{child1}(z_i) \text{ OP child2}(z_i)). \\
4. & \text{Convert each } (z_i \iff \text{child1}(z_i) \text{ OP child2}(z_i)) \text{ term into 3CNF.} \\
5. & \text{Output the resulting 3CNF formula.}
\end{align*}
\]

As an example, consider the formula \( \phi = (x_1 \land (\neg x_1 \lor x_2)) \lor (x_2 \land x_3) \). We let \( z_0 = \phi \), \( z_1 = (x_1 \land (\neg x_1 \lor x_2)) \), \( z_2 = \neg(x_2 \land x_3) \), \( z_3 = (\neg x_1 \lor x_2) \), \( z_4 = \neg x_1 \), and \( z_5 = (x_2 \land x_3) \). Then the tree constructed in step (1) is:
To analyze this reduction, first observe that the function $f$ takes polynomial time. If $\langle \phi \rangle$ is in SAT, then there exists an assignment $a_1, \ldots, a_n$ that satisfies $\phi$. We can compute the appropriate assignments to the $z_i$ variables from $a_1, \ldots, a_n$ (i.e. by traversing the tree from the leaves upward). By construction, $\phi'$ is also satisfied. For the other direction, if $\langle \phi' \rangle$ is in 3SAT, let $a_1, \ldots, a_n, b_1, \ldots, b_\ell$ be its satisfying assignment. Then $a_1, \ldots, a_n$ satisfy the original formula $\phi$. Thus, the reduction is correct. In particular, SAT $\leq_p$ 3SAT, so 3SAT is NP-complete.

**Proof that CLIQUE is NP-Complete**  
The language CLIQUE consists of encodings $\langle G, k \rangle$ such that $G$ is a graph with a $k$-clique. We saw last class that CLIQUE is in NP; we will show it is NP-hard by a reduction from 3SAT. That is, we will construct a reduction such that 3SAT $\leq_p$ CLIQUE. Consider the following function $f$:

---

$f$, on input $\langle \phi \rangle$:

1. Construct the graph $G = (V, E)$ where:
2. Output \( \langle G, k \rangle \), where \( k \) is the number of clauses in \( \phi \).

For example, suppose we have the 3CNF formula \( \phi = (x_1 \lor x_2 \lor x_3) \land (\neg x_1 \lor \neg x_2 \lor x_4) \). Then our graph \( G \) has 3 vertices per clause, and edges connecting vertices in different clauses whose corresponding literals do not contradict each other. Specifically, we have:

To analyze this reduction, first note that \( f \) is a poly-time computable function. If \( \phi \) has a satisfying assignment, then there is a \( k \)-clique connecting one true-valued literal from each clause. That is, there is a set of literals, one from each clause, that do not contradict one another. By our construction of \( G \), this set forms a \( k \)-clique. For the other direction, if \( G \) contains a \( k \)-clique, then we can assign variables so that the literals corresponding to the vertices in the clique are true. There must be one vertex from the clique in each clause, and there are no contradictions in the corresponding variable assignment. Thus, every clause is satisfied so \( \phi \) is satisfied overall. Therefore, this reduction shows that 3SAT \( \leq_p \) CLIQUE, proving that CLIQUE is NP-complete. \( \blacksquare \)

2 The Cook-Levin Theorem

Cook-Levin Theorem  SAT is NP-complete.

Proof  SAT is in NP, as we can verify a truth assignment in polynomial time. To show that SAT is NP-hard, let \( B \) be any language in NP. We will show that \( B \leq_p \) SAT. To do so, let \( N \) be a poly-time NTM for \( B \). Let \( t(n) \) be its runtime. Our reduction from \( B \) to SAT “knows” \( N \) and \( t(n) \), which can in some sense be hardwired into the reduction. On input \( w \), we want to construct \( \phi \) such that \( N \) has an accepting computation history on \( w \)
if and only if $\phi$ has a satisfying variable assignment.

We represent the computation history of $N$ on input $w$ as a “tableau” with $t(n) + 1$ rows and $t(n) + 3$ columns. Note that the number of columns is bounded by $t(n)$ because a TM can move at most $t(n)$ spaces in $t(n)$ time. If $N$ halts in fewer than $t(n)$ steps, then rows are repeated. In general, a tableau is of the form:

<table>
<thead>
<tr>
<th>time 0:</th>
<th>#</th>
<th>$q_0$</th>
<th>$w_0$</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>$w_n$</th>
<th>$_________$</th>
<th>$_________$</th>
<th>$_________$</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>time 1:</td>
<td>#</td>
<td>$a$</td>
<td>$q_1$</td>
<td>$w_1$</td>
<td>...</td>
<td>...</td>
<td>$w_n$</td>
<td>$_________$</td>
<td>$_________$</td>
<td>$_________$</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td></td>
<td>#</td>
<td></td>
<td></td>
<td>#</td>
<td></td>
<td></td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td></td>
<td>#</td>
<td></td>
<td></td>
<td>#</td>
<td></td>
<td></td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td></td>
<td>#</td>
<td></td>
<td></td>
<td>#</td>
<td></td>
<td></td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td></td>
<td>#</td>
<td></td>
<td></td>
<td>#</td>
<td></td>
<td></td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>time $t(n)$:</td>
<td>#</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A tableau represents a valid and accepting computation history of $N$ on $w$ if the following conditions hold:

1. Row 0 contains $\#q_0w_0...w_n...\#$.

2. For all $1 \leq i \leq t(n)$, row $i$ follows from row $i - 1$ via a valid transition of $N$.

3. The state $q_{accept}$ appears in the tableau.

Now it remains to encode a tableau as a Boolean formula. To do so, we will use the set of variables:

$$\{X_{i,j,u} \mid 0 \leq i \leq t(n) \text{ is the row, } 1 \leq j \leq t(n) + 3 \text{ is the column, and}$$

$$u \in \Gamma \cup Q \cup \{\#\} \text{ is a possible content of cell } (i,j)\}$$

The number of variables is $(t(n) + 1) \cdot (t(n) + 3) \cdot (|\Gamma| + |Q| + 1)$, which is polynomial with respect to $n$. In general, the variable $X_{i,j,u}$ is true if cell $(i,j)$ contains the symbol $u$. To construct the formula $\phi$, we take the conjunction of several component formulas:

$$\phi = \phi_{cell} \land \phi_{start} \land \phi_{move} \land \phi_{accept}$$

In particular, $\phi_{cell}$ is satisfied iff there is exactly one element $u$ in each cell, $\phi_{start}$ is satisfied iff the first row correctly corresponds to the initial configuration of $N$, $\phi_{move}$ is satisfied iff
row $i$ follows from row $i-1$, and $\phi_{\text{accept}}$ is satisfied iff the tableau contains $q_{\text{accept}}$.

We construct $\phi_{\text{cell}}$ to check that every cell contains exactly one element:

$$
\phi_{\text{cell}} = \bigwedge_{i=0}^{t(n)-1} \bigwedge_{j=1}^{t(n)+3} \phi_{\text{cell}(i,j)}
$$

$$
\phi_{\text{cell}(i,j)} = \left( \bigvee_{u \in \Gamma \cup Q \cup \{\#\}} X_{i,j,u} \right) \land \neg \left( \bigvee_{u \neq u'} X_{i,j,u} \land X_{i,j,u'} \right)
$$

We construct $\phi_{\text{start}}$ to check that the first row is the starting configuration of $N$:

$$
\phi_{\text{start}} = X_{0,1,\#} \land X_{0,2,\#} \land X_{0,3,\#} \land X_{0,4,\#} \land \cdots \land X_{0,t(n),\#}
$$

We construct $\phi_{\text{accept}}$ to check that $q_{\text{accept}}$ appears in the tableau:

$$
\phi_{\text{accept}} = \bigvee_{i=1}^{t(n)} \bigvee_{j=1}^{t(n)+3} X_{i,j,q_{\text{accept}}}
$$

Finally, we construct $\phi_{\text{move}}$ to check that for every row, row $i$ follows from row $i-1$. It is sufficient to check two-by-three windows in the tableau, rather than an entire row at a time. For example, the following is a valid window:

```
  a  b  c
  a  b  c
```

On the other hand, something fishy is going on in this window, where the tape head disappears:

```
  a  q  c
  a  a  c
```

We let $W$ be the set of valid two-by-three windows $\{a,b,c,d,e,f\}$, given $N$’s transition function. In general, represent a window as:

```
  a  b  c
  d  e  f
```

Then we have:

$$
\phi_{\text{move}} = \bigwedge_{i=0}^{t(n)-1} \bigwedge_{j=1}^{t(n)+1} \bigvee_{(a,b,c,d,e,f) \in W} X_{i,j,a} \land X_{i,j+1,b} \land X_{i,j+2,c} \land X_{i+1,j,d} \land X_{i+1,j+1,e} \land X_{i+1,j+2,f}
$$

In an analysis of this reduction, we would prove that $\phi$ has a satisfying assignment if and only if $N$ has an accepting computation history on input $w$, and that the reduction takes polynomial time. We could then conclude that SAT is NP-complete. ■