Announcements
1. Homework 11 due Wednesday 12/7 (a day that will live in infamy).

The Cook-Levin Theorem
1. Last class, we completed the proof of the Cook-Levin Theorem:

   **Theorem:** SAT is NP-Complete. That is, SAT ∈ NP and for any A ∈ NP, A ≤ₚ SAT.

   But we just barely completed it. So, I would like to start today with a quick review of the proof designed to give you a chance to ask questions about the proof.

2. We want to show that for any A ∈ NP, A ≤ₚ SAT.

3. All this tells us about A is that for some non-deterministic TM, M_A, A = L(M_A) and there is some polynomial function p(n) such that on an input w, M_A runs for at most p(|w|) steps (whether it accepts or not).

4. We need to show that given descriptions of M_A and p(n), we can identify a function f which for any input w will produce a boolean expression φ in such a way that φ is satisfiable iff w ∈ A = L(M_A).

5. As a result, rather than really describing f, we described a meta-function F such that F(M_A, p) = f_A where f_A it a polynomial-time computable function such that f_A(w) = φ is satisfiable iff w ∈ A.

6. The approach we took to describe F is based on computation histories. We want f_A(w) to be a formula which encodes the requirements for a table of TM configurations to represent a valid, accepting computation of M_A starting on input w that lasts for less than p(|w|) steps.

7. We are using the fact that a computation history can be organized in a table like:

   ```
   ( S $ 1 0 1 0 $ 1 0 # 1 0 1 # 1 1 1 # )
   ( $ S 1 0 1 0 $ 1 0 # 1 0 1 # 1 1 1 # )
   ( $ 1 0 S 1 0 $ 1 0 # 1 0 1 # 1 1 1 # )
   ( $ 1 0 1 S 0 $ 1 0 # 1 0 1 # 1 1 1 # )
   ( $ 1 0 1 0 S $ 1 0 # 1 0 1 # 1 1 1 # )
   ( $ 1 0 1 0 $ C 1 0 # 1 0 1 # 1 1 1 # )
   ( $ 1 0 1 0 $ # Z 0 # 1 0 1 # 1 1 1 # )
   ( $ 1 0 1 0 $ # # Z # 1 0 1 # 1 1 1 # )
   ( $ 1 0 1 0 $ # # # C 1 0 1 # 1 1 1 # )
   ```

8. Given this tabular view of computation histories, we can create a set of boolean variables that completely describe a computation history. For each cell in the computation history table there would be one boolean variable for each symbol that could appear in that cell. That is, we would have a variable

   \[ x_{r,c,s} \]

   for every row r, column c and symbol s in the union of our Turing machines tape alphabet Γ, its state set Q, and any other symbols we use to encode the configurations (like the parentheses we use to mark the beginning and end of each configuration) with \( x_{r,c,s} = \text{true} \iff s \text{ appeared in the } c\text{th position of the } r\text{th row of the table}. \)

9. We have described a formula happy_r,c that is true iff the variables reflect that a unique symbol is associated with cell r,c.

   - We can capture the need for some symbol to be associated with the cell with the formula:

     \[ \text{assigned}_{r,c} = \bigvee_{s \in \Gamma \cup Q \cup \{()\}} x_{r,c,s} \]

   - We can capture the need for some the assignment to be unique with with the formula:

     \[ \text{unique}_{r,c} = \bigwedge_{s,t \in \Gamma \cup Q \cup \{()\}} (x_{r,c,s} \lor x_{r,c,t}) \]
• Combining these we then get:

\[
\text{happy}_{r,c} = \text{assigned}_{r,c} \land \text{unique}_{r,c}
\]

10. Capturing the initial configuration is easy. Assuming the input is \( w = w_1 w_2 \ldots w_n \), we define:

\[
\text{start} = x_{0,0} ( \land x_{0,1,q_0} \land ( \bigwedge_{1 \leq i \leq n} x_{0,i+1,w_i} ) \land x_{0,n+2}) \land ( \bigwedge_{n+2 \leq i \leq p(n)} x_{0,i})
\]

11. Making sure that the computation accepts simply require making sure that some cell in the table contains the accept state:

\[
\text{accept} = \bigvee_{1 \leq r,c < p(n)} x_{r,c,\text{accept}}
\]

12. The tricky part is making sure that the configuration in each row of the table yields the configuration in the next row.

• The secret to capturing this is to notice that the contents of consecutive rows in the computation history must be identical except for the symbols before and after the cell holding the state in the earlier row. Therefore, if we verify the correctness of each 2 by 3 block of cells in our table, we can be sure all transitions are handled correctly.

• Given that the size of the tape alphabet and state set are fixed we know that there are a fixed number \((|\Gamma| + |Q| + 2)^6\) of possible configurations of a 2 by 3 subsection of the history table. The number of invalid 2 by 3 subsection must be smaller that this.

• For each invalid configuration, we can produce a boolean expression that verifies that the bad configuration does not appear at a particular point in our table. For example, the subsection:

<table>
<thead>
<tr>
<th>0</th>
<th>q4</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>q4</td>
</tr>
</tbody>
</table>

would be invalid for any machine (since the symbol 0 which was not under the tape head has changed into a 1).

• If we name this configuration \( c \), then the expression

\[
\text{notbad}_c(r,c) =
\]

\[x_{r,c,0} \lor x_{r,c+1,q_4} \lor x_{r,c+2,1} \lor x_{r+1,c,1} \lor x_{r+1,c+1,1} \lor x_{r+1,c+2,q_4}\]

yields true only if the bad configuration does not appear at position \( r,c \).

• Better yet, this formula is a disjunction. So the formula:

\[
\text{notAllBad}(r,c) = \bigwedge_{c \in \text{bad configurations}} \text{notbad}_c(r,c)
\]

ensures us that everything is fine at \( r,c \) and is in CNF.

13. With these pieces, we can describe the formula we use to reduce \( A \) to SAT:

\[
\phi = ( \bigwedge_{0 \leq r,c \leq p(n)} \text{happy}(r,c)) \land ( \bigwedge_{0 \leq r,c \leq p(n) - \epsilon} \text{notAllBad}(r,c)) \land \text{start} \land \text{accept}
\]

**Problems on Graphs**

1. There are many other problems that are NP-Complete.

2. Given that we know that SAT and 3-SAT are NP-complete, we can show that another problem \( C \) is NP-complete by showing that 3 - SAT \( \leq_p \) \( C \) or SAT \( \leq_p \) \( C \) or \( B \leq_p \) \( C \) for any other \( B \) that we have already shown to be NP-complete.

• In general if \( B \) is NP-complete, \( B \leq_p C \Rightarrow C \) is NP-complete as long as \( C \in NP \).

• Basically, \( \leq_p \) is transitive. We know that if \( B \) is NP-complete then for any \( A, A \leq_p B \). This means that there exist a polynomial time computable function \( f_{A \rightarrow B} \) such that \( f_{A \rightarrow B}(w) \in B \) iff \( w \in A \). Now, if we learn or can show that \( B \leq_p C \) there must also be a polynomial time function \( f_{B \rightarrow C} \) such that \( f_{B \rightarrow C}(w) \in C \) iff \( w \in B \). The function obtained by composing these two functions, \( f_{A \rightarrow C}(w) = f_{B \rightarrow C}(f_{A \rightarrow B}(w)) \) must be polynomial computable
and have the property that $f_{A\to C}(w) \in C$ iff $w \in A$ which implies that $A \leq_p C$. Thus, all $A$ are polynomial reducible to $C$ which means that $C$ must be NP-complete.

- Given that we showed how to reduce 3-SAT to subset sum before we even showed that 3-SAT was NP-complete, we immediately know that subset sum is NP-complete.

3. Many NP-complete problems involve graphs:
- $k$-clique
- $k$-colorability
- vertex cover
- Hamiltonian path
- ...

4. For an example, we will start with the clique problem.
- We say a set of nodes in a graph form a clique if there is an edge connecting every pair of nodes in the set.
- The $k$-clique problem asks whether a given graph contains at least one clique of size $k$.

\[ k\text{-CLIQUE} = \{\langle G, k \rangle | G \text{ is an undirected graph that contains a } k\text{-clique } \} \]

- $K$-CLIQUE $\in$ NP since if we are given a set of $k$ nodes we can verify that each node has all of the other nodes as neighbors in polynomial time.
- To show that $k$-CLIQUE is NP-complete, we will show that 3-SAT $\leq_p k$-CLIQUE.

5. To do this, we must show that given a formula $\phi$ in 3-conjunctive normal form we can construct a graph $G$ and determine a value $k$ such that $G$ has a clique of size $k$ if and only if $\phi$ is satisfiable.

6. In fact, given a 3-CNF formula with $k$ clauses, we will construct a graph with $3k$ nodes that has a clique of size $k$ iff the formula is satisfiable.

7. Our graph will contain one node for each literal that appears in any clause.

8. To illustrate the construction, we will use my favorite 3-CNF formula — the formula for even parity:

\[(x_1 \lor x_2 \lor \overline{x_3}) \land (x_1 \lor \overline{x_2} \lor x_3) \land (\overline{x_1} \lor x_2 \lor x_3) \land (\overline{x_1} \lor \overline{x_2} \lor \overline{x_3}) \]

9. The rules for connecting these nodes with edges will depend on both the variable name used in the literal and the clause to which it belongs. Therefore, it helps to label the nodes with the literals to which they correspond and to group them according to the clause to which they belong:

10. Our goal is to insert edges in such a way that if there is a satisfying assignment for $\phi$, then there will be a clique of size $k$ in the graph and also that if we find a clique of size $k$, then $\phi$ will be satisfiable.

11. To accomplish the first goal, we will add edges in such a way that if, given a satisfying assignment, we pick a node corresponding to one of the true literals in each clause, then these nodes will form a clique.

- If this was our only goal, we could accomplish it by simply inserting edges to connect every node to all of the other nodes associated with different clauses.

12. On the other hand, since we also need to make sure that there is no clique of size $k$ if $\phi$ is not satisfiable, we need to make sure that if there is a clique of size $k$, then $\phi$ must be satisfiable.
To ensure this we will first make sure that every clique contains one node from each clause by not including edges that would connect two nodes associated with the same clause.

This ensures that if there is a $k$-clique it contains one node associated with each clause.

Next, we need to make sure that it is possible to assign the value true to all of the literals associated with the nodes that form the clique. The only problem that could arise here is if for some variable $x_i$, nodes associated with both $x_i$ and $\overline{x_i}$ belong to the clique. To prevent this, we will make sure such nodes are not connected.

13. This logic leads to the following construction rule:

- Connect each node to every other node that is associated with a different clause and is not associated with the complement of the literal associated with source node.

14. For example, applying this rule to the node for the literal $x_1$ that appears in the first clause would produce:

15. Applying it to all of the nodes corresponding to literals in the first term yields:

16. And applying it to all of the nodes yields:

17. To understand how this construction works, consider the assignment $x_1 = \text{true}$, $x_2 = \text{false}$, $x_3 = \text{true}$. This is a satisfying assignment since $x_1$ appears in clause 1, $\overline{x_2}$ appears in clauses 2 and 4, and $x_3$ appears in clause 4. In the following figure, we have highlighted the nodes corresponding to true literal and the edges for the clique formed by four literals that produce true.
18. Because of the way we constructed the graph, for any satisfying assignment of \( \phi \), we can choose one node associated with a true literals for each clause of \( \phi \) and the resulting set must always produce a clique as desired.

19. At the same time, as explained in the construction, any clique that can be found in the graph corresponds to some satisfying assignment.