DISCLAIMER: These notes are not an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

- Overall grades, presentation, test 3 content, etc, etc...
- Review of $P$, $NP$,
- NP-completeness and reductions
- Clique $\leq_P$ Vertex-Cover
- Quiz 11
- 3-SAT $\leq_P$ Clique.

Last time we formally introduced $P$ and $NP$. $P$ is the set of all poly time decidable languages, is given by

$$P = \bigcup_k \text{TIME}(n^k).$$

We use $\text{NTIME}(g(n))$ to denote the set of all languages decidable by $O(g(n))$ time non-deterministic Turing machines. It also gives the definition of $NP$, non-deterministic polynomial time:

$$NP = \bigcup_k \text{NTIME}(n^k).$$

NP can also be characterized as:

**Definition 0.1.** A verifier of a language $A$ is an algorithm $V$ where

$$A = \{ w \mid V \text{ accepts } w, c \text{ for some string } c \}.$$

**Theorem 0.2.** $NP$ is precisely the languages with poly-time verifiers. That is, a verifier $V$ that runs in time $O(n^k)$ when $w$ has length $n$.

This is because we can pass in all the non-deterministic choices made by the Turing-machine ahead of time via the string $c$. 


1 P vs. NP and NP-Completeness

Note that because we can simulate over all \( c \) by brute force, we also have

\[
NP \subseteq \bigcup_k \text{TIME}(2^k).
\]

Somewhat surprisingly, this is the best that’s known to date in terms of the relation between \( P \) and \( NP \).

What we do know instead, is a large class of problems such that solving any one of them in \( P \) will imply \( P = NP \). \( NP \)-completeness is defined via \( NP \)-hardness, which in turn requires the notion of poly-time reducibility.

**Definition 1.1.** \( A \) is poly-time reducible to \( B \), or \( A \leq_P B \) if there is a poly-time function \( f \) such that

\[
w \in A \iff f(w) \in B.
\]

Note that this in particular means that if \( B \) is in \( P \), \( A \) would be in \( P \) as well: we simply run \( B \) on \( f(w) \), whose length is at most \( \text{poly}(|w|) \) because \( f \) runs in poly-time. This leads to the notion of \( NP \)-hard:

**Definition 1.2.** \( A \) is \( NP \)-hard if for every \( B \in NP \), we have \( B \leq_P A \).

Furthermore, a language is \( NP \)-complete if it’s both in \( NP \) and also \( NP \)-hard.

Note that if \( A \) is \( NP \)-hard, and we have \( A \leq_P B \), then \( B \) is also \( NP \)-hard. This gives the general strategy for carving out a space of \( NP \)-hard problems: starting with one problem, and reduce it to other problems.

The problem in question is 3-SAT: it asks whether there is an assignment of variables \( x_1 \ldots x_n \) that satisfies all clauses simultaneously. A clause is the OR of three literals, and each literal is either some variable, or its negation.

The Cook-Levin theorem gives that 3-SAT is \( NP \)-hard.

**Theorem 1.3** (Cook-Levin Theorem). *For any problem \( A \) in \( NP \), we have \( A \leq_P 3 \)-SAT.*

As 3-SAT is in \( NP \), it is also \( NP \)-complete.

2 Some NP-Hard Problems

We will show later that 3-SAT \( \leq_P \) CLIQUE, which means CLIQUE is also \( NP \)-hard.

Here it’s not even clear how maximum clique, which is an optimization problem, can be defined as a language. Formally, we define:

\[
\text{CLIQUE} = \{(G, k) \mid G \text{ has a clique of size } k\}.
\]

Note that given such a routine, we can use it to produce a clique by:
1. Delete a vertex.

2. If the maximum clique got smaller, then put this vertex back.

Similarly, we can define the decision version of the max-independent set problem, which seeks to find the maximum number of pairwise disjoint vertices.

\[
\text{INDEPENDENT-SET} = \{(G, k) \mid G \text{ contains } k \text{ vertices such that no pair is connected by an edge.}\}
\]

To show that \text{INDEPENDENT-SET} is also NP-hard, it suffices to show

\[
\text{Clique} \leq_p \text{INDEPENDENT-SET}.
\]

To do so, note that the definition of a clique is exactly opposite of the definition of an independent set: in one, every pair is connected, while in the other, every pair is disconnected.

So it suffices to take the complement graph: for some \(G\), we create \(\overline{G}\) where \(uv \in E(G)\) if and only if \(uv \notin E(G)\). Then the mapping function is

\[
f((G, k)) = (\overline{G}, k),
\]

which takes poly-time since it just checks every edge against the edge list of \(G\).

Note that the same function can also be used to show

\[
\text{INDEPENDENT-SET} \leq_p \text{Clique}.
\]

However, in general, it’s a very bad idea to mix up these two.

3 \ NP-Hardness of Clique

The main idea is that the structure of 3-SAT is rich enough fo the literals/clauses to be interpreted as (groups) of vertices. This then allows us to convert instances of 3-SAT to instances of graph theoretic problems.

To do so, we start with the easier direction: Note that given a candidate set \(S\), we can check whether the size of \(y\) is at least \(k\), and whether all pairs of vertices in \(y\) are connected. Therefore it’s easier to have a verifier \(C'((G, k), S)\).

For the harder direction we want:

\[
\text{3-SAT} \rightarrow \text{Clique}.
\]

That is, we want to create a function that takes any 3-SAT instance, and output a CLIQUE instance that’s true iff the 3-SAT instance is true.

Given a 3-SAT instance with clauses \(c_1 \ldots c_m\) and variables \(x_1 \ldots x_n\), we create a graph with \(3m\) vertices as follows:
1. For each clause $C_r = l_{r,1} \lor l_{r,2} \lor l_{r,3}$, create one vertex for each of $l_{r,1}$, $l_{r,2}$ and $l_{r,3}$.

2. Place an edge between two vertices $l_{r,i}$ and $l_{s,j}$ if and only if:
   - $r \neq s$, the literals are from different clauses, and
   - $l_{r,i} \neq \neg l_{s,j}$, aka. they are consistent.

Since each the 3 vertices corresponding to each clause are not connected from each other, at most one of them can be in a clique at a time. This means the maximum clique size is at most $m$. We now show that a clique of size $m$ exist if and only if the 3-SAT instance is satisfiable.

1. If we have such a clique, then take the vertices involved. Since none of them conflict with each other, we can set the variables accordingly. As we have one such literal per clause, each clause would then have a satisfied literal, making this a satisfying assignment.

2. If we have an satisfying assignment, then each clause has one literal that’s satisfied. Since none of these literals conflict with each other, there are edges between them, giving the clique of size $m$. 

4