Recitation 8: NP-Completeness

This recitation will give you more insight into NP-Completeness by reviewing the intuition behind it as well as providing more examples of how to prove a problem is NP-Complete. Reductions are daunting at first and take some time to digest, but they are one of the most important concepts you will take away from this class. The problems you solve on the PSets will progressively give you more and more experience with this line of reasoning, and you’ll be able to tackle more abstract reductions as the semester progresses!

Definition and Intuition

Before giving the formal definition, we motivate the notion of NP-Completeness. The question of $P \text{ vs } NP$ has been central to complexity theory for a long time. Showing that $P = NP$ would require proving that every single language in $NP$ has a deterministic polynomial-time algorithm. Showing this directly would require putting in a lot of effort for every single $NP$ language. On the other hand, showing that $P \neq NP$ would require that at least one problem in $NP$ does not have a deterministic polynomial-time algorithm. But how would we choose this problem? Imagine spending years trying to prove that something like $\text{COMPOSITES} \not\in P$, only to have someone show that you can actually decide it in polynomial time!

If you’re curious about this, you should look up the AKS Primality check, which allows you to deterministically check whether or not a number is prime in polynomial time.

$NP$-Completeness addresses both of these issues. We find a set of the “hardest” problems in $NP$, meaning that a decider for one of these languages would yield a decider for all languages in $NP$ with only an additional polynomial overhead. Then, if we show one of these languages is in $P$, we immediately get that $P = NP$. For this reason, if one wants to show $P \neq NP$, an NP-Complete problem could be worth studying, since $P \neq NP$ would imply all NP-Complete problems do not have a deterministic poly-time algorithm!

One thing to keep in mind is that the tools for proving NP Completeness have strong parallels to what we used for proving undecidability. Our first time proving a language, $SAT$, is NP complete took substantial effort using a computation history method similar to that
in PCP. Similarly, our first time proving a language, \( A_{TM} \), was undecidable took considerable effort using diagonalization. However, after this is done, we use the power of reductions to leverage our initial result in more quickly proving any other language NP-Complete or undecidable. To prove \( A \) is NP-complete, we can show \( SAT \leq_p A \); to prove \( B \) is undecidable, we can show \( B \leq_m A_{TM} \).

Now, let’s move onto the formal definition of NP-Completeness.

**Definition 1 (NP-Completeness).** A language \( B \) is NP-Complete if it satisfies the two following properties:

1. \( B \in NP \)

2. For each language \( A \in NP \), we have \( A \leq_p B \)

Property 2 is known as NP-Hardness.

We have now formally defined what it would mean for a language to be one of the “hardest” in \( NP \). But how do we know such a language actually exists? The Cook-Levin Theorem tells us that the boolean satisfiability language, \( SAT \), actually has these properties. You probably remember the proof at a high level, and how tricky it was to make the approach general enough so that it works for any language in \( NP \). We don’t have to redo this approach every time we want to show some language \( B \) is NP-Complete, so we will generally exploit the following result.

**Theorem 1.** Let \( B \) be a language in \( NP \). If \( A \) is NP-Complete and \( A \leq_p B \), then \( B \) is NP-Complete.

Thus, once we know at least one language \( A \) is NP-Complete, we may show that some other language \( B \) is NP-Complete by giving a polynomial-time reduction from \( A \) to \( B \). This means that \( A \) can be any language we have previously shown to be in \( NP \), but in practice we will often take 3\( SAT \), where the boolean formulas have a special 3CNF form: an AND of clauses, where each clause is an OR of three literals.

**Gadget Constructions**

One of the reasons we often use 3\( SAT \) in reductions is that we can use a standarized reduction technique based on constructing “gadgets” that encode different parts of the 3\( SAT \) instance. When showing that 3\( SAT \) \( \leq_p B \) we usually construct two main types of gadgets:

- **Variable Gadgets:** These simulate a variable by capturing whether it is set to True or False.
- **Clause Gadgets:** These simulate a clause by capturing whether the clause is satisfied.
A useful strategy when constructing a mapping reduction \( f \) to show \( A \leq_p B \) for proving NP-completeness is considering how to relate the two instances’ certificates. You need to show \( w \in A \iff f(w) \in B \). Typically, you show this by taking a certificate for \( w \in A \) and constructing a certificate for \( f(w) \in B \). Then you show the reverse: take a certificate for \( f(w) \in B \) and construct a certificate for \( w \in A \). Often, the latter part involves analyzing the structure of possible certificates.

Now it’s time to see these gadget constructions in practice.

### 3Color is NP-Complete

3COLOR is the set of graphs where a each vertex can be assigned one of three colors such that no edge connects two vertices of the same color. You can think of the vertices as jobs that need to be done and the jobs connected by an edge as those that can’t be done at the same time. If such a graph had a valid three coloring as described above, all the jobs can be completed in three steps: in each step, the jobs assigned to some particular color would be run.

**Definition 2 (3COLOR).** We define the language:

\[
3COLOR = \{\langle G \rangle | G \text{ is 3 colorable (no edge between same-color vertices)}\}
\]

**Theorem 2.** 3COLOR is NP-Complete.

**Proof.** First we show that 3COLOR \( \in \) NP. Note that in this case the certificate would be the coloring assignment to vertices, and the verifier would simply have to check that all the edges connect differently colored vertices.

Next, we will show that 3SAT \( \leq_p \) 3COLOR. The intuition here is that we will be able to think of one of the colors as true, another as false, and use the third color to force an assignment of true or false.

Given \( \phi \), a 3CNF formula, we will construct \( G \), a graph which has a valid 3 coloring iff \( \phi \) is satisfiable.

For the first part of our reduction, we will construct the variable gadget. Here we want to capture that each variable should be set to either true or false by using the third color to force the assignment. To build off of our true-false idea above, we can think of constructing two vertices for each variable in \( \phi \), one vertex for the true literal and the other for the false. If we connect these two vertices to each other and a third vertex, then all the vertices should have different colors. This aligns with our goal of having a color represent true and another false. Now if we do the same for all the other variables while keeping the third vertex the same, then the true and false colors are consistent amongst the literals. Note that which color corresponds to true or false is arbitrary, and we can think of having a palette of the true color T, the
false color F, and some other color C to capture this fact. The diagram below shows how the variables $x_1, x_2, x_3$ can be translated in a graph.

Now we will move to constructing the clause gadget. Since each clause consists of two or’s, we will first figure out how to simulate an or using vertices and edges. The following simulates an or:

We can see that the above indeed simulates an or by looking at the “truth” table of what the top, or output, vertex can be while maintaining 3-colorability:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$Z$ possibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>C,T,F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>C,T,F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

So, the output vertex can be colored true only if the at least one of the input nodes is colored true. We can chain these or gadgets together to create a clause gadget by having the inputs being the corresponding literal vertices from the clause. Now, we connect the final output vertex to the two non-true vertices in the palette to force the clause to be true. An example clause of $(x_1 \lor \overline{x}_2 \lor \overline{x}_3)$ is shown below.

We can add in variable gadgets for all the clauses in $\phi$ in the same fashion to finish $G$. 
Each variable and clause in $\phi$ gets translated to a constant number of vertices and edges in $G$ so the reduction is polynomial time.

Now we have to prove that $\phi$ is satisfiable iff $G \in 3\text{COLOR}$.

($\rightarrow$) If $\phi$ is satisfiable, we will use the satisfying assignment to set the corresponding literal in each variable gadget to be the true color (we can pick this color arbitrarily to be green). We then color the second half of the literal vertices to be say red. We will pick the third color to be blue and then color the palette accordingly. As we showed when describing the clause gadgets, the rest of the clause vertices can be colored to yield a valid 3 colored graph.

($\leftarrow$) If $G$ has a valid 3 coloring, we will create an assignment where we set each variable to true if its corresponding non-negated vertex is colored the same as the true vertex in the palette. Otherwise, we set that variable to false. $\phi$ will be true under this based on our reasoning above since a valid three coloring exists only when each clause gadget has its final vertex as the true color.

\[ \square \]

**UHampath is NP-Complete**

Recall the Hamiltonian Path problem, where given a directed graph $G = (V, E)$ and two nodes $s, t$ we want to determine if there is a path from $s$ to $t$ that goes through each node in $V$ exactly once. We will show that the undirected variant of this problem is also NP-Complete.

**Definition 3 (UHAMPATH).** We define the language:
**UHAMPATH** = \{ ⟨G, s, t⟩ | G is an undirected graph with a Hamiltonian path between s and t \}

**Theorem 3.** UHAMPATH is NP-Complete.

**Proof.** First we show that UHAMPATH ∈ NP. Note that in this case the certificate would be the path, and the verifier would simply have to check that all the edges in the path are actually in E.

Next, we will show that HAMPATH ≤ₚ UHAMPATH. The main intuition here is that we will be able to simulate each directed edge using a series of undirected edges between special nodes.

Let G = (V, E) be our directed graph. We will construct undirected graph G′ = (V′, E′) which will have a Hamiltonian Path if and only if G has one.

For each node v ∈ V that is not s or t, we will create three nodes v_{in}, v_{mid}, v_{out} in V′, along with two undirected edges (v_{in}, v_{mid}) and (v_{mid}, v_{out}) in E′. For s and t we just create s_{out} and t_{in}, since the path must start at s and end at t so we will not have edges going into s or coming out of t in it.

Next, for each directed edge u, v in E, we create an edge (u_{out}, v_{in}) in E′. We will now argue that G has a Hamiltonian path from s to t if and only if G′ has a Hamiltonian path from s_{out} to t_{in}.

(→) If there is a Hamiltonian Path in G, we can obtain one in G′ by simply following the edges, but for each intermediate node we will have v_{in}, v_{mid}, v_{out} in the path. (←) It is sufficient to argue that any Hamiltonian path in G′ always traverses each in-mid-out triple in that exact order. Assume there is a path that, for some v, visits v_{in}, then some other nodes, and then returns to v_{mid} and v_{out}. Note that the path would be stuck at v_{mid}, since it only has two edges: one to v_{in} and v_{out}. Since we assumed v_{in} had already been visited, and v_{mid} was reached from v_{out}, then there is no way to continue a Hamiltonian path from v_{mid} in this case.

This allows us to conclude that the path will consist of a sequence of triples of in-mid-out nodes. Then, since we copied the edges from E to E′ by connecting out-nodes to in-nodes, this means that the direction of the traversal respects that of the directed edges, so we can simply copy the sequence of triples into a sequence of nodes in G and we obtain a Hamiltonian Path.

This shows that UHAMPATH is NP-Complete. □

**SAT ≤ₚ 3SAT**

This section is meant to fill in a gap you may have noticed in our usual chain of NP-Completeness reductions. We know that SAT is NP-Complete from the Cook-Levin Theorem. Then, we have often
shown that other problems are NP-Complete by giving a reduction from \(3SAT\), which allows us to exploit the structure of 3CNF formulas. But how exactly does the NP-Completeness of \(3SAT\) follow from \(SAT\)?

**Theorem 4.** \(3SAT\) is NP-Complete.

**Proof.** Note that \(3SAT \in NP\) since we can give a satisfying assignment as a certificate, then the verifier just has to check that this assignment satisfies every clause.

We now show \(SAT \leq_p 3SAT\). To begin, we observe that we can interpret any boolean formula as a binary tree. If you are familiar with logical circuits, then that may be a useful way to think about this transformation. The key is that any AND and OR operation only acts on two logical values, and NOTs only apply to one, so we can establish a natural “order” on the operations based on any parentheses. For example, the formula

\[
\phi = (x_1 \lor x_2) \land (x_1 \land x_3)
\]

can be transformed into the following binary tree:

![Binary Tree Diagram]

To construct the 3CNF formula \(\phi'\), we will first introduce a new set of variables, one for each node in the binary tree:

![New Variable Diagram]

Now the 3CNF formula will be equivalent to checking whether there is a consistent evaluation of this binary tree that makes the root True, meaning \(z_1 = 1\). This means that our first clause will just check for this, so we will have \((z_1 \lor z_1 \lor z_1)\) in \(\phi'\). Now, we need the \(z_i\)'s to agree with the operation they represent, so we need to create clauses.
that check for this consistency. The idea behind these will be to look at
the truth table of the operation, and then create 3CNF clauses that are
equivalent. For example, let’s create the clauses that enforce \( z_2 \) being
consistent with \( x_1 \lor \overline{x_2} \). The truth table is as follows:

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( z_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

We can rewrite this as four implications that must all be satisfied:

- \( \overline{x_1} \land \overline{x_2} \Rightarrow z_2 \)
- \( \overline{x_1} \land x_2 \Rightarrow \overline{z_2} \)
- \( x_1 \land \overline{x_2} \Rightarrow z_2 \)
- \( x_1 \land x_2 \Rightarrow z_2 \)

It turns out we can actually write each of these implications as a
3CNF clause. First we will rewrite the implication in Boolean logic,
and then use DeMorgan’s Law to turn it into a 3CNF clause. We will
show the procedure for \( \overline{x_1} \land \overline{x_2} \Rightarrow z_2 \). This implication tells us that
if the LHS is 1, the RHS must be 1, but if the LHS is 0 then the RHS
can be anything. The following formula is equivalent to this constraint

\[
(\overline{(x_1 \land \overline{x_2})} \lor z_2)
\]

since, if \( \overline{x_1} \land \overline{x_2} = 1 \), \( z_2 \) is forced to be equal to 1 for the formula to
evaluate to 1. Now, if we apply DeMorgan’s Law, we get:

\[
(x_1 \lor x_2 \lor z_2)
\]

which is exactly a 3CNF clause. We then convert the three other impli-
cations using the same procedure, then AND them, and get four CNF
clauses that capture the consistency of \( z_2 \) with \( x_1 \lor \overline{x_2} \).

We will now present another way of seeing how to generate the four
CNF clauses that might seem less magical. This involves looking at all
eight possible boolean assignments for a node and its two children; we
use one CNF clause to rule out each of the four incorrect assignments.
To make this more clear, we will use the same example as above: \( z_2 = x_1 \lor \overline{x_2} \). We present the possible boolean assignments below and color
in red those that aren’t consistent with \( z_2 = x_1 \lor \overline{x_2} \).
Now, for each of the red rows in the table, we will construct the clause which disallows them. The first row is the assignment of $x_1 = 0, x_2 = 0, z_2 = 0$, and $(x_1 \lor x_2 \lor z_2)$ disallows exactly that by saying at least one of the literals’ values must be flipped. Similarly, the clause we create to disallow the fourth row will be $(x_1 \lor \overline{x_2} \lor \overline{z_2})$. So for each operation node in the tree, we will get 4 CNF clauses with 3 literals each. We do this for all the nodes in the binary tree, and end up with a 3CNF formula.

If the total number of operations in our original boolean formula was $k$, then the total number of clauses we created is $4k + 1$, so this reduction runs in polynomial time.

All that remains is to argue that $\phi \in SAT \iff \phi' \in 3SAT$.

($\rightarrow$) If there is a satisfying assignment to $\phi$, we evaluate our binary tree and set the $z_i$ variables according to the result of the intermediate nodes. Since $\phi'$ checks for a consistent evaluation of $\phi$’s binary tree such that $z_1 = 1$, and we already know we have a satisfying assignment for $\phi$, this process gives us a satisfying assignment to $\phi'$.

($\leftarrow$) If there is a satisfying assignment for $\phi'$, this means there is a consistent evaluation of the binary tree representation of $\phi$ which results in $z_1 = 1$. Moreover, the satisfying assignment for $\phi'$ already has assigned values to $x_1, \cdots, x_n$ that result in such an evaluation. As a result, we can just copy these values to obtain the satisfying assignment to $\phi$. 

$\square$