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 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 $COMPOSITES \notin P$, 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 $A_{TM} \leq_m B$.

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 C be a language in NP. If B is NP-Complete and $B \leq_p C$, then C is NP-Complete.*

How do we prove this? Since B is $NP - COMPLETE$, for all $A \in NP$, $A \leq_p B$. If $B \leq_p C$, then for all $A \in NP$, $A \leq_p B \leq_p C$, satisfying the definition of NP-Completeness for C.

Thus, once we know at least one language B is NP-Complete, we may show that some other language C in NP is NP-Complete by giving a polynomial-time reduction from B to C. This means that B can be any language we have previously shown to be NP-Complete, but in practice we will often take $3SAT$, where the boolean formulas have a special 3CNF form: an AND of clauses, where each clause is an OR of three literals. See figure (1) for a diagram of P, NP, and where $NP - COMPLETE$ languages fall, in the cases of both $P \neq NP$ and $P = NP$.

Gadget Constructions

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

- Variable Gadgets: These simulate a variable by capturing whether it is set to True or False.

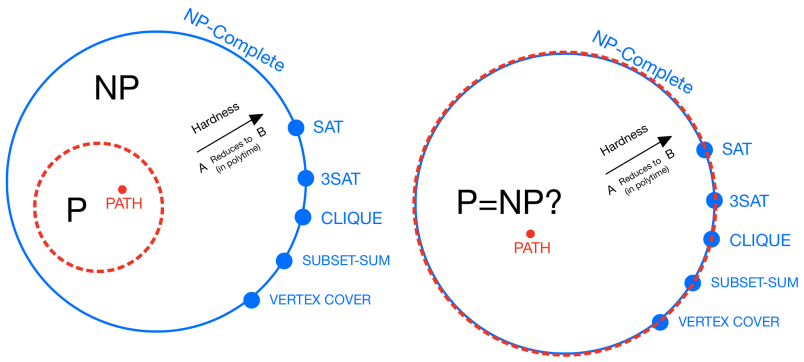


Figure 1: On the left, we have a schema for when P and NP may not be equal. On the right, we have a schema for when $P = NP$.

- 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. Two requirements we will repeatedly see are:

1. The reduction must enforce that each variable is assigned to true or false, but not both.
2. The reduction must enforce that each clause has at least one true literal.

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

SUBSET-SUM is NP-Complete

The $SUBSET - SUM$ language includes pairs of numbers and targets such that a subset of the numbers sums to the target.

Definition 2 (SUBSET-SUM). *Formally, we define the language:*

$$SUBSET - SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\} \text{ and for some } \{y_1, \ldots, y_l\} \subseteq S, \text{we have} \sum y_i = t\}$$

For example, $\langle\{5, 2, 3, 9\}, 11\rangle \in SUBSET - SUM$ because $9 + 2 = 11$. This problem is "hard" because the subset that works may be of any size.

Theorem 2. *$SUBSET - SUM$ is NP-Complete.*

Proof. In order to show that $SUBSET-SUM$ is NP-complete, we need to show the following:

1. $SUBSET-SUM \in NP$:
   We can prove this by providing a certificate that can be checked in polynomial time. The subset $c \subseteq S$ is the certificate, and we can have a verifier check whether or not all of the elements in c sum to t in polynomial time, so $SUBSET-SUM \in NP$. We could alternatively prove $SUBSET-SUM \in NP$ by providing a NTM which nondeterministically guesses a subset c, then tests if it sums to t and accepts if so.

2. $SUBSET-SUM$ *is NP-hard*:

   We reduce from $3SAT$; in other words, we show that $3SAT \leq_p SUBSET-SUM$. Let ϕ be a Boolean formula with n variables $x_1, \ldots, x_n$ and m clauses $c_1, \ldots, c_m$. We would like to construct a corresponding $SUBSET-SUM$ instance $\langle S, t\rangle$ such that

   $$\phi \in 3SAT \iff \langle S, t\rangle \in SUBSET-SUM.$$

   Construction
   We construct the reduction from ϕ to a $SUBSET-SUM$ instance $\langle\{y_1, z_1, \ldots, y_n, z_n, g_1, h_1, \ldots, g_m, h_m\}, t\rangle$ shown in the table below. Each row is one of the numbers in S. We hope to find a subset that sums to t, shown on the bottom.

	1	2	3	4	$\cdots$	n	c_1	c_2	$\cdots$	c_m
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
$\vdots$				$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	
y_n					$\cdots$	1	0	0	$\cdots$	1
z_n					$\cdots$	1	0	0	$\cdots$	1
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_m										1
h_m										1
t	1	1	1	1	1	1	3	3	3	3

   In this construction we have variable gadgets and clause gadgets:

- **Variable gadgets:** For each variable x_i in ϕ, we have two numbers $y_i, z_i \in S$, corresponding to a true or false assignment of x_i, respectively. Both have a 1 in the ith column, enforcing that we can only select either y_i (representing a true assignment) or z_i (representing a false assignment) for our subset, but not both. For each y_i and z_i, we also have a 1 in each clause column c_j where x_i or $\overline{x_i}$, respectively, occurs.
- **Clause gadgets:** For each clause c_i, we create numbers $g_i, h_i \in S$, where both g_i and h_i have a 1 in column c_i. These act as fillers.

We choose a subset of the numbers such that each column sums to the corresponding entry in t. Since each clause can have at most three literals, each column can have at most five 1's, and summing columns will never result in a "carry."

Correctness
Let's prove the correctness of this construction:

- $\phi \in 3SAT \implies \langle S, t \rangle \in SUBSET - SUM$.
  Given the certificate of a satisfiable assignment to ϕ, we must find a subset of S summing to t. For each x_i, if x_i is true add y_i to the subset. Otherwise, add z_i to the subset. Since we only select either y_i or z_i for each variable, the first n columns in our construction will each sum to 1, as desired. Additionally, since ϕ is satisfied, each clause must have at least one true literal, and therefore at least one 1 in the top right quadrant. For each clause, select one or both of g_i and h_i such that each clause column will sum to 3 in t, as desired.
- $\langle S, t \rangle \in SUBSET - SUM \implies \phi \in 3SAT$.
  Given the certificate of a valid subset, we must find a satisfying assignment to ϕ. Only one of y_i or z_i can be in our subset. If we have y_i, let x_i be true. Otherwise, let x_i be false. By nature of our subset construction, each clause column must have at least one 1, and thus each clause must have at least one true literal. Thus, our assignment is valid and satisfies ϕ.

Runtime
Finally, let's ensure we can construct $\langle S, t \rangle$ in polynomial time. The size of our table is $O((2n + 2m)^2)$, which is polynomial in the size of ϕ, as desired.

□

VERTEX COVER is NP-Complete

Given an undirected graph G, a **vertex cover** of G is a subset of its nodes touching every edge in G.

Definition 3 (VC). *Formally, we define the language*

$$VC = \{\langle G, k\rangle \mid G \text{ has a vertex cover of size at most } k\}.$$

Theorem 3. *VC is* NP*-complete.*

Proof. As usual, in order to prove VC is NP-Complete, we must show the following:

1. $VC \in NP$:
   We first realize that $VC \in \mathsf{NP}$ because we can have a certificate c be a set of k vertices that forms a vertex cover. A verifier can easily check that every edge has at least one endpoint from those k vertices from the certificate.

2. VC is NP-hard:
   We reduce from $3SAT$; in other words, we show that $3SAT \leq_p VC$. Let ϕ be a Boolean formula with n variables $x_1, \ldots, x_n$ and m clauses $c_1, \ldots, c_m$. We would like to construct a corresponding VC instance $\langle G, k\rangle$ such that

   $$\phi \in 3SAT \iff \langle G, k\rangle \in VC.$$

   Construction
   Again, we will make both variable gadgets and clause gadgets.

   - **Variable gadgets:** For each variable x_i, we make a connected component with a node labeled x_i^v and a node labeled $\overline{x_i^v}$, connected by an edge. Note that we only need to select *one of* x_i^v or $\overline{x_i^v}$ to cover this edge, representing how we only assign each variable to either true or false.

Figure 2: A variable gadget for x_1.

   - **Clause gadgets:** For each clause c_i, we make a connected component with one node for each of its three literals, connected in a triangle. See figure (3) for an example.

   To connect these components, we draw edges between each x_i^v and x_i^c, and each $\overline{x_i^v}$ and $\overline{x_i^c}$. Finally, we choose $k = n + 2m$ because for each of the n variables we add one vertex to the cover and for each

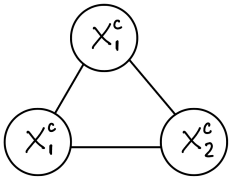


Figure 3: A clause gadget for $(x_1 \vee x_1 \vee x_1)$.

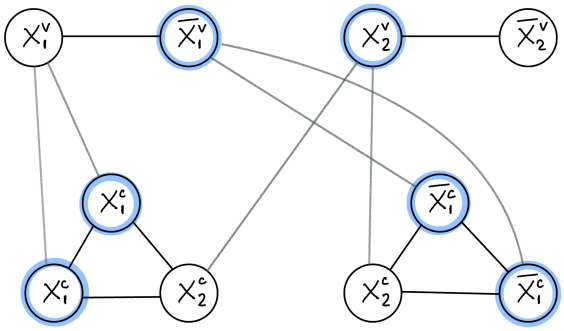


Figure 4: Our construction for $\phi = (x_1 \vee x_2 \vee x_1) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_1})$. A satisfying assignment sets x_1 to false and x_2 to true. The nodes in the corresponding vertex cover are outlined in blue.

of the m clauses we add two vertices to the cover. See figure (4) for an example construction.

Correctness Let's prove the correctness of this construction:

- $\phi \in 3SAT \implies \langle G, k \rangle \in VC$.
  Given a certificate of a satisfying assignment, we must find a size k vertex cover. For each x_i, if x_i is true add node x_i^v to the cover. This will cover the edges within the variable gadgets. Since we have a satisfying assignment, we know at least one literal in each clause is true. For each clause, choose some true literal. Add the other two to the vertex cover. These will cover the edges in the clause gadgets. Now we have $n + 2m = k$ vertices in our cover.
  Let's consider the edges between the variable and clause gadgets. Those connecting true literals will be covered by vertices from the variable gadgets. Those connecting false literals must be covered by vertices in the clause gadgets. Therefore, we have a valid vertex cover of size k.
- $\langle G, k \rangle \in VC \implies \phi \in 3SAT$.
  We look at the variable gadgets to select an assignment. For each x_i^v in the vertex cover, assign x_i to true. For each $\overline{x_i^v}$ is in the vertex cover, assign x_i to false. By our construction and because $k = n + 2m$, we cannot have a vertex cover containing both x_i^v and $\overline{x_i^v}$ (both a false and true assignment). Furthermore, since we can add at most two nodes from each clause gadget to the cover, we must have at least one true literal in each clause. Thus we have a valid satisfying assignment.

Runtime
Finally, we show that we can construct $\langle G, k \rangle$ in polynomial time. To see this, note that graph G has $2n + 3m$ nodes and at most $(2n + 3m)^2$ edges, so it is polynomial in terms of the input and can be constructed in polynomial time.

□

$SAT \leq_p 3SAT$

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?

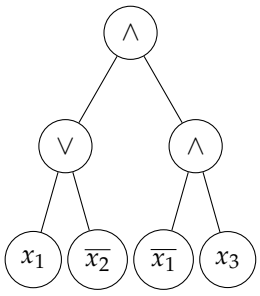
Theorem 4. *$3SAT$ is NP-Complete.*

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.

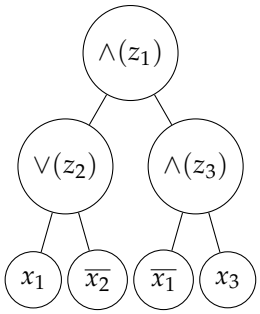
We now show $SAT \leq_p 3SAT$. To begin, we observe that we can interpret any boolean formula as a binary tree. If you are familiar with logical circuits, then that may be a useful way to think about this transformation. The key is that any AND and OR operation only acts on two logical values, and NOTs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

$$\phi = (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \wedge x_3)$$

can be transformed into the following binary tree:



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



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 $(z_1 \vee z_1 \vee z_1)$ in ϕ'. 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 $x_1 \vee \overline{x_2}$. The truth table is as follows:

x_1	x_2	z_2
0	0	1
0	1	0
1	0	1
1	1	1

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

- $\overline{x_1} \wedge \overline{x_2} \implies z_2$
- $\overline{x_1} \wedge x_2 \implies \overline{z_2}$
- $x_1 \wedge \overline{x_2} \implies z_2$
- $x_1 \wedge x_2 \implies z_2$

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 $\overline{x_1} \wedge \overline{x_2} \implies z_2$. 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

$$(\overline{(\overline{x_1} \wedge \overline{x_2})} \vee z_2)$$

since, if $\overline{x_1} \wedge \overline{x_2} = 1$, 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:

$$(x_1 \vee x_2 \vee z_2)$$

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

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: $z_2 = x_1 \vee \overline{x_2}$. We present the possible boolean assignments below and color in red those that aren't consistent with $z_2 = x_1 \vee \overline{x_2}$.

x_1	x_2	z_2
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

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 $(x_1 \vee x_2 \vee z_2)$ 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 $(x_1 \vee \overline{x_2} \vee \overline{z_2})$. 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 ϕ, we evaluate our binary tree and set the z_i variables according to the result of the intermediate nodes. Since ϕ' checks for a consistent evaluation of ϕ's binary tree such that $z_1 = 1$, and we already know we have a satisfying assignment for ϕ, this process gives us a satisfying assignment to ϕ'.

($\leftarrow$) If there is a satisfying assignment for ϕ', this means there is a consistent evaluation of the binary tree representation of ϕ which results in $z_1 = 1$. Moreover, the satisfying assignment for ϕ' 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 ϕ.

□