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 () 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.

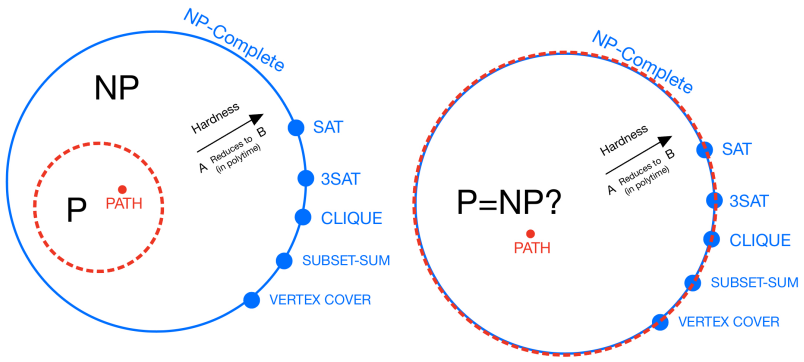

Figure 1: On the left, we have a schema for when P and NP may not be equal. On the right, we have a schema for when $P = NP$.

- 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 number sets 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, written in base 10. 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. Calculating each element of the table takes polynomial time and the size of our table is $O((2n + 2m)^2)$, which is polynomial in the size of ϕ, as desired.

□

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:*

$UHAMPATH = \{\langle G, s, t\rangle |$ *G is an undirected graph with a Hamiltonian path between s and t*$\}$

Theorem 3. $UHAMPATH$ *is* NP*-complete.*

Proof. As usual, in order to prove $UHAMPATH$ is NP-complete, we must show the following:

1. $UHAMPATH \in$ NP:
   We can prove this by providing a certificate that can be checked in polynomial time. The certificate is just a list of n nodes, where n is the number of nodes in G. A verifier would only have to check that there are no repetitions and that there is an edge between adjacent pairs in the list. These can both be done in polynomial time.

2. $UHAMPATH$ is NP-hard:
   We reduce from $HAMPATH$; in other words, we show that $HAMPATH \leq_p UHAMPATH$. Let $\langle G, s, t\rangle$ be a $HAMPATH$ instance. We would like to construct a corresponding $UHAMPATH$ instance $\langle G', s', t'\rangle$ such that

   $$\langle G, s, t\rangle \in HAMPATH \iff \langle G', s', t'\rangle \in UHAMPATH.$$

   Construction
   We write $G = (V, E)$ and $G' = (V', E')$. We construct V' and E' as follows. For each node $v \in V$ 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'. Finally, we let $s' = s_{out}$ and $t' = t_{in}$.

   Correctness
   Let's prove the correctness of this construction:

   - $\langle G, s, t\rangle \in HAMPATH \implies \langle G', s', t'\rangle \in UHAMPATH.$
     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.

- $\langle G', s', t' \rangle \in UHAMPATH \implies \langle G, s, t \rangle \in HAMPATH$.
  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.

Runtime

Finally, we show that we can construct $\langle G', s', t' \rangle$ in polynomial time. To see this, note that graph G' has less than $3|V|$ nodes and $3|E|$ edges so its size is polynomial in the size of G.

□

Note how we used a pretty different reduction approach to the usual gadget-based construction from 3SAT. There is still a sense of gadgets here: node gadgets and edge gadgets. As an exercise, try to identify what parts of the construction could be considered node gadgets and which could be considered edge gadgets.