Theory of Computation: Assignment 11 Solutions

Arjun Chandrasekhar

1. Because L_1 and L_2 are in NP, there exist nondeterministic TMs M_1 and M_2 that recognize L_1 and L_2 respectively in time $O(n^c)$. There also exist deterministic verifiers V_1, V_2 for L_1 and L_2 respectively that run in time $O(n^c)$.

   (a) **Approach 1:** We'll construct a machine M to recognize $L_1 \cup L_2$ in nondeterministic polynomial time. On input w, M nondeterministically guess whether to run M_1 or M_2 and accepts if the guessed machine accepts. Because nondeterministically guessing which machine to run takes $O(1)$ time, and because both machines run in nondeterministic time $O(n^c)$, the overall runtime is $O(n^c)$. Thus, $L_1 \cup L_2 \in \text{NTIME}(n^c)$.

   Approach 2: We'll construct a polynomial-time verifier V. The verifier takes as input a string w and two certificates c_1, c_2. V runs V_1 on (w, c_1), and then runs V_2 on (w, c_2). If either verifier accepts, we accept. Both certificates are polynomially bounded in length, so the overall length of $\langle c1, c2 \rangle$ is polynomially bounded. Because both verifiers run in time $O(n^c)$, the overall runtime is $O(n^c) + O(n^c) = O(n^c)$. Thus, $L_1 \cup L_2$ can be verified in time $O(n^c)$, meaning $L_1 \cup L_2 \in \text{NTIME}(n^c)$.

   (b) **Approach 1:** We'll construct a nondeterministic machine M to recognize $L_1 \circ L_2$ in $O(n^c)$ time. On input w, M nondeterministically guesses how to split up $w = xy$. It then runs M_1 on x and M_2 on y and accepts if both machines accept. It takes $O(1)$ time to nondeterministically guess how to split up the string. After that, both M_1 and M_2 run in $O(n^c)$; thus the overall runtime is $O(1) + O(n^c) + O(n^c)$. Thus, $L_1 \circ L_2 \in \text{NTIME}(n^c)$.

   Note: It might be tempting to have M deterministically try all possible ways of splitting up $w = xy$. However, this will increase our runtime by a factor of $O(n)$, meaning our overall runtime will be $O(n^{c+1})$. By taking advantage of nondeterminism we can reduce the runtime to $O(n^c)$.

   Approach 2: We'll construct polynomial-time verifier V. The verifier takes four inputs: a string w, two certificates c_1, c_2, and a way of splitting up $w = xy$. The verifier checks that xy is a valid way to split up w. It then checks that V_1 accepts (w, c_1), and it checks that V_2 accepts (w, c_2). It accepts if both verifiers accept.

   Both certificates are polynomially bounded in length, so the overall length of $\langle c1, c2 \rangle$ is polynomially bounded. It takes $O(n)$ time to verify that xy is a valid split of w, and both verifiers run in time $O(n^c)$. Thus V runs in polynomial time. Thus, $L_1 \circ L_2$ can be verified in time $O(n^c)$, meaning $L_1 \circ L_2 \in \text{NTIME}(n^c)$.

   (c) To prove that NP is closed under union and concatenation, we would need to show that if L_1 and L_2 are in NP, then $L_1 \cup L_2$ and $L_1 \circ L_2$ are also be in NP. If L_1 and L_2 are in NP, then L_1 and L_2 are in $\text{NTIME}(n^c) \subseteq \text{NP}$. for some constant c. From parts (a) and (b), we know that $L_1 \cup L_2$ and $L_1 \circ L_2$ are in $\text{NTIME}(n^c) \subseteq \text{NP}$. This proves that $L_1 \cup L_2$ and $L_1 \circ L_2$ are both in NP, thus establishing the desired closure properties.

2. (a) Suppose $L \in \text{P}$. There is a machine M that recognizes L in deterministic polynomial time.

However, M *is* a nondeterministic machine that simply chooses not to use any nondeterminism. Thus M recognizes L in nondeterministic polynomial time. Thus, $L \in \text{NP}$

(b) Because $L \in \text{NP}$ there exists a polynomial time verifier V such that $w \in L \Leftrightarrow (w, c)$ is accepted by V for some polynomial-length certificate c. We'll construct a machine M that recognizes L in exponential time. On string w, M simply tries out all possible certificates c. If V accepts (w, c) for any c, then we accept w.

Because the certificate c must be polynomial in length, we have to try at most $O\left(2^{n^c}\right)$ certificates. Each certificate can be verified in exponential (in fact, polynomial) time. Thus the overall runtime is exponential.

3. For this problem let n be the number of vertices and m be the number of edges in G.

(a) **Approach 1:** We'll construct a machine that recognizes VERTEX-COVER nondeterministic polynomial time. On input $\langle G, k\rangle$, M nondeterministically guesses a vertex cover C of size k. It then verifies that every edge in the graph touches one of the vertices in C. Guessing C will take $O(k)$ steps; verifying that C is a valid vertex cover takes $O(m \cdot k)$ steps. Thus, the runtime is nondeterministic polynomial. Thus, VERTEX-COVER $\in$ NP

Approach 2: We'll construct a verifier V that verifies VERTEX-COVER in polynomial time. Our verifier takes $\langle G, k\rangle$ as input; it also takes a vertex cover C as a certificate. The length of the certificate is $O(k)$ because C can't have more than k vertices. We then check that $|C| = k$ (which takes $O(k)$ steps), and we check that C touches every edge in G (which takes $O(k \cdot m) steps$). Thus, we can verify a proposed vertex cover in polynomial. Thus, VERTEX-COVER $\in$ NP

(b) First, suppose that C is a vertex cover. Let u, v be any two vertices in $V \backslash C$. We know that u and v cannot be connected by an edge, because every edge has at least one endpoint in C. Thus, $V \backslash C$ represents an independent set.

Next, suppose $V \backslash C$ is an independent set. Take any edge $u, v \in E$. Because $V \backslash C$ is an independent set, then either $u \notin V \backslash C$, or $v \notin V \backslash C$. This means that either $u \in C$ or $v \in C$; thus, u, v has at least one endpoint in C. The same argument applies to every other edge; thus, C is a vertex cover.

(c) Suppose we are given $\langle G, k\rangle$ as input. From part (b), we know that a set C is a vertex cover if and only if $V \backslash C$ is an independent set. Thus, G has an independent set of size k if and only if G has a vertex cover of size $n - k$. Thus, our reduction is

$$\langle G, k\rangle \mapsto \langle G, n - k\rangle$$

.

Yes maps to yes: Suppose G has an independent set I of size k. Then from part (b), we know that $C = V \backslash I$ forms a vertex cover of size $n - k$.

No maps to no: Suppose G has a vertex cover C of size $n - k$. Thne from part (b), we know that $I = V \backslash C$ forms a vertex cover of size $n - (n - k) = k$.

We can compute $n - k$ in polynomial time, thus the reduction is polynomial. This proves that IND-SET $\leq_{\text{poly}}$ VERTEX-COVER, which implies that VERTEX-COVER is NP-Hard (and thus NP-complete).

Figure 1 gives a diagram of what it means for VERTEX-COVER to be NP-Complete.

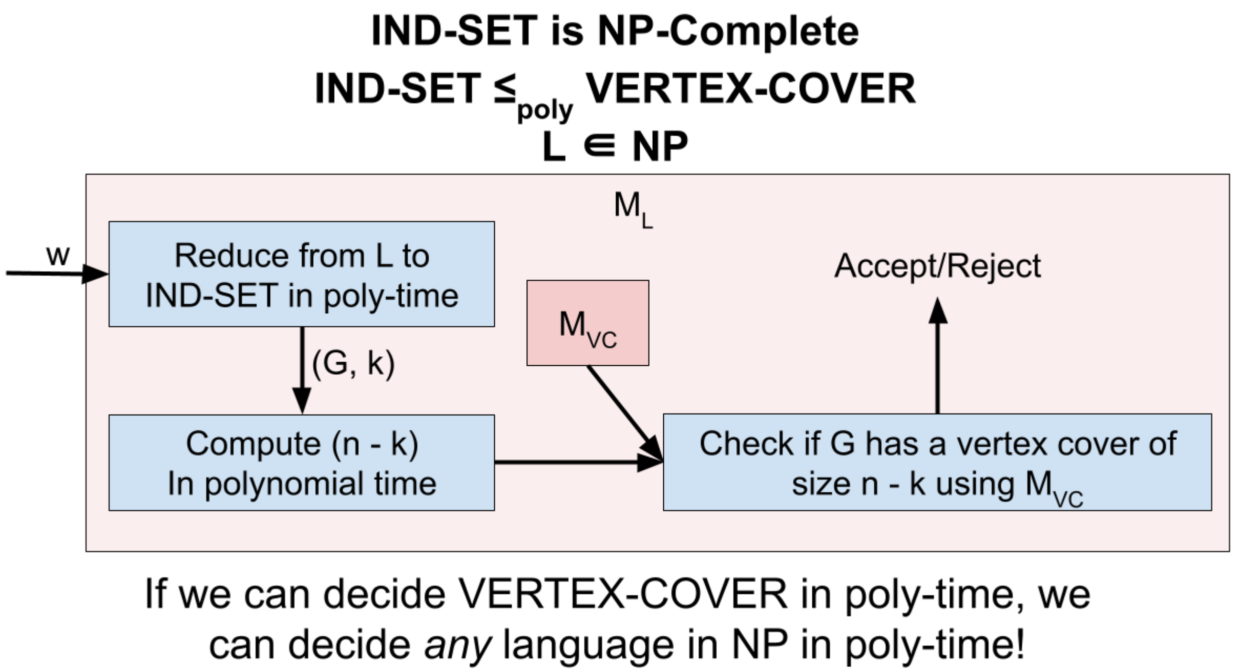

Figure 1: Vertex cover is NP-complete

4. (a) **Approach 1:** We'll construct a machine that recognizes PARTITION in nondeterministic polynomial time. On input $S = (s_1, \dots s_n)$, our machine nondeterministically guess a partition $T \subseteq S$ in $O(n)$ time. It then verifies that $\sum T = \sum S \backslash T$, which can be done in polynomial time. Thus, our machine recognizes L in nondetermiistic polynomial time, which establishes that PARTITION $\in$ NP.

   Approach 2: We'll construct a polynomial-time verifier V. The verifier takes $S = (s_1, \dots s_n)$ as input. It also takes a certificate T, a subset of the numbers, as input. The length of the certificate is at most $O(|S|)$, which is polynomial. The verifier first checks that T is a proper subset of S, which can be done in polynomial time. It then adds up all of the numbers in T and all the numbers not in T and checks that the sums are equal. This can be done in polynomial time as well. Thus, we can verify the certificate in polynomial time. This establishes that PARTITION has a polynomial-time verifier, and thus PARTITION $\in$ NP.

   (b) First note that For the forward direction, suppose there is a combination $(s_i, s_j, \dots s_k)$ that adds up to B. Then all of the remaining elements add up to $(\sum s_i - B) + T = (\sum s_i - B) + (2B - \sum s_i) = B$. Thus, both the two subsets form a partition.

   For the backward direction, suppose we have two subsets $R, S \backslash R$ that both add up to $\dfrac{(\sum S) + T}{2} = \dfrac{(\sum s_i) + (2B - \sum s_i)}{2} = \dfrac{2B}{2} = B$. One of those subsets does not include the new number T. This subset represents a combination of the original numbers that adds up to B - thus, a solution to the original SUBSET-SUM instance.

   (c) We will reduce from SUBSET-SUM. Suppose we start with $\langle B, s_1, s_2, \dots s_n \rangle$. Following the hint, we compute $T = 2B - \sum s_i$. We then attempt to find a partition in $S \cup T = (s_1, s_2, \dots x_n, T)$.

Yes maps to yes: Suppose S has a subset that adds up to B. Then from part (b), $S \cup T$ has a partition.

No maps to no: Suppose $S \cup T$ has a partition. Then from part (b), S has a subset that adds up to B.

Computing $T = 2B - \sum s_i$ takes polynomial time, thus the reduction is polynomial.
Thus

$$(B, s_1, \ldots s_n) \mapsto \left(s_1, \ldots s_n, 2B - \sum s_i\right)$$

is a poly-time reduction from SUBSET-SUM to PARTITION. This establishes that PARTITION is NP-hard, and thus NP-complete.

Figure 2 gives a diagram of what it means for PARTITION to be NP-Complete.

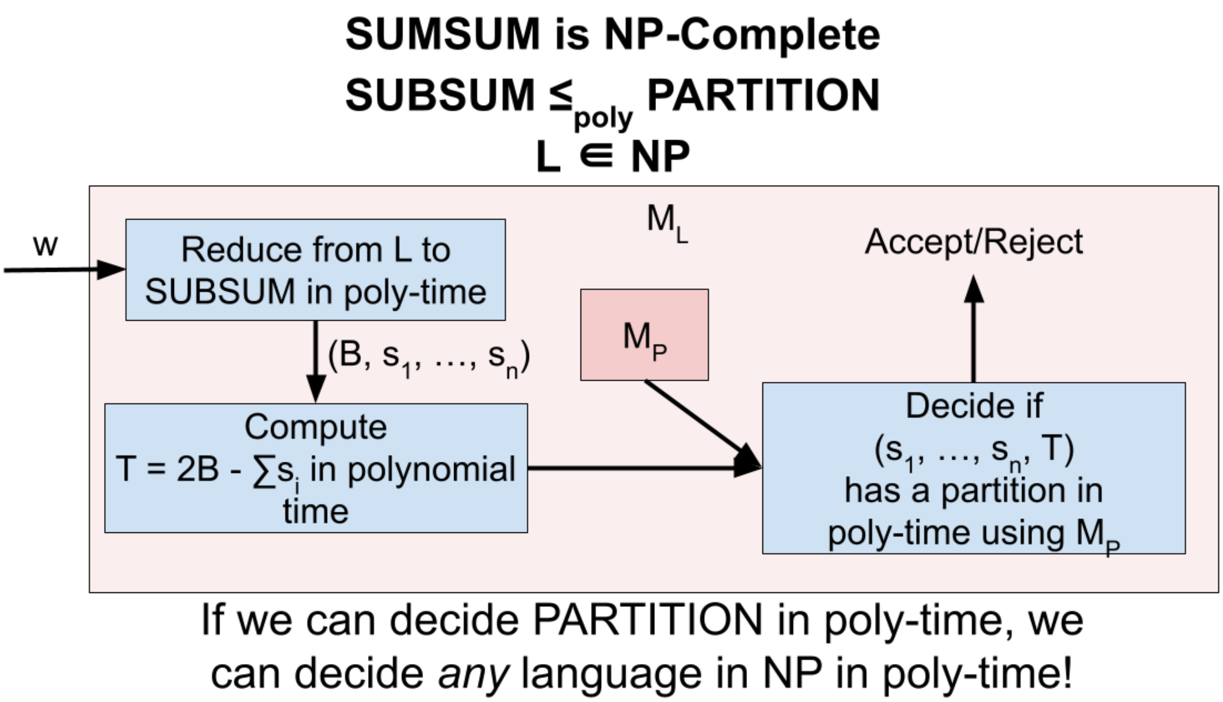

Figure 2: Partition is NP-complete