CS120: Intro. to Algorithms and their Limitations Hesterberg & Vadhan

Section 10: NP & NP-completeness

Harvard SEAS - Fall 2022 *Nov. 9, 2022*

1 NP and NP-completeness

Roughly speaking, NP consists of the computational problems where solutions can be *verified* in polynomial time. This is a very natural requirement; what's the point in searching for something if we can't recognize when we've found it?

Definition 1.1. A computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ is in $\mathsf{NP}_{\mathsf{search}}$ if the following conditions hold:

1. All solutions are of polynomial length: There is a polynomial p such that for every $x \in \mathcal{I}$ and every $y \in f(x)$, we have $|y| \leq p(|x|)$, where $|z|$ denotes the bitlength of z.
2. All solutions are verifiable in polynomial time: There's an algorithm in P that, given $x \in \mathcal{I}$ and a potential solution y, decides whether $y \in f(x)$.

Definition 1.2 (NP-completeness, search version)**.** A problem Π is $\mathsf{NP}_{\mathsf{search}}$*-complete* if:

1. Π is in $\mathsf{NP}_{\mathsf{search}}$
2. Π is $\mathsf{NP}_{\mathsf{search}}$*-hard*: For every computational problem $\Gamma \in \mathsf{NP}_{\mathsf{search}}$, $\Gamma \leq_p \Pi$.

We can think of the NP-complete problems as the "hardest" problems in NP. Indeed:

Proposition 1.3. *Suppose* Π *is* $\mathsf{NP}_{\mathsf{search}}$*-complete. Then* $\Pi \in \mathsf{P}_{\mathsf{search}}$ *iff* $\mathsf{NP}_{\mathsf{search}} \subseteq \mathsf{P}_{\mathsf{search}}$.

Theorem 1.4 (Cook–Levin Theorem)**.** *SAT is* $\mathsf{NP}_{\mathsf{search}}$*-complete.*

This can be interpreted as strong evidence that SAT is not solvable in polynomial time. If it were, then *every* problem in $\mathsf{NP}_{\mathsf{search}}$ would be solvable in polynomial time.

1.1 Proving that a problem is $\mathsf{NP}_{\mathsf{search}}$-complete [1]

Definition 1.5. For computational problems Π and Γ, we write $\Pi \leq_p \Gamma$ if there is a *polynomial-time* reduction R from Π to Γ. That is, on an input of length N the reduction should run in time $O(N^c)$, if we count the oracle calls as one time step (as usual).

To show that a problem is $\mathsf{NP}_{\mathsf{search}}$-complete via reduction, we typically show that (a) it is in $\mathsf{NP}_{\mathsf{search}}$, and (b) that it is "just as hard as" a problem that we already know to be $\mathsf{NP}_{\mathsf{search}}$-complete. NP-completeness proofs have the following structure:

What is the usual strategy for proving that a problem Γ is $\mathsf{NP}_{\mathsf{search}}$-complete?

[1] Every problem in $\mathsf{NP}_{\mathsf{search}}$ has a corresponding decision problem (deciding whether or not there is a solution). However, for this problem set, we will be focused on $\mathsf{NP}_{\mathsf{search}}$-completeness, not NP-completeness. Not every $\mathsf{NP}_{\mathsf{search}}$-complete problem's corresponding decision problem is NP-complete.

1. **In $\mathsf{NP}_{\mathsf{search}}$**: Show that the problem is in $\mathsf{NP}_{\mathsf{search}}$ by proving that solutions can be verified in polynomial time.

2. **$\mathsf{NP}_{\mathsf{search}}$-hard**:

   (a) Pick a known $\mathsf{NP}_{\mathsf{search}}$-complete problem Π to try to reduce to Γ. Typically, we might try to pick a problem Π that seems as similar as possible to Γ, or which has been used to prove that problems similar to Γ are $\mathsf{NP}_{\mathsf{search}}$-complete. Otherwise 3-SAT is often a good fallback option.

   (b) Come up with an algorithm R mapping instances x of Π to instances $R(x)$ of Γ. If Π is 3-SAT, this will often involve designing "variable gadgets" that force solutions to $R(x)$ to encode true/false assignments to variables of x and "clause gadgets" that force these assignments to satisfy each of the clauses of x.

   (c) Show that R runs in polynomial time.

   (d) Show that if x has a solution, then so does $R(x)$. That is, we can transform valid solutions to x to valid solutions to $R(x)$.

   (e) Conversely, show that if $R(x)$ has a solution, then so does x. Moreover, we can transform valid solutions to $R(x)$ into valid solutions to x in *polynomial time*. This transformation needs to be efficient (in contrast to Item 2d because it has to be carried out by our reduction).

Question 1.6. Consider the problems A and B. Are these statements true or false?

1. If A reduces to B and B is NP-hard, then A is NP-hard also.
2. If A reduces to B and A is NP-hard, then B is NP-hard also.
3. If A reduces to B and A is not NP-hard, then B must also not be NP-hard.
4. If A reduces to B and B is not NP-hard, then A must also not be NP-hard.
5. If A is NP-hard and B is not NP-hard, A will never reduce to B.

1.1.1 Mapping Reductions

The reduction template for $\mathsf{NP}_{\mathsf{search}}$-completeness described above is formalized by the concept of a mapping reduction:

Definition 1.7. Let $\Pi = (\mathcal{I}, f)$ and $\Gamma = (\mathcal{J}, g)$ be search problems. A *polynomial-time mapping reduction* from Π to Γ consists of two polynomial-time algorithms R and S[2] such that for every $x \in \mathcal{I}$:

1. $R(x) \in \mathcal{J}$.
2. If $f(x) \neq \emptyset$, then $R(x) \neq \emptyset$.
3. If $y \in g(R(x))$, then $S(x, y) \in f(x)$.

The above conditions in the definition can be understood as:

1. The reduction, when applied to the input x to computational problem Π, meaning that it is in $\mathcal{I}$, will give a valid input to computational problem Γ, meaning it is in $\mathcal{J}$.
2. If the algorithm f run on x has a valid solution for Π, then the reduction applied to that input must also return a valid solution for Γ as well.
3. Running the algorithm g on input $R(x) \in \mathcal{J}$, we should be able to run the post-processing algorithm S on the result in order to obtain an output that is in the correct output set for Π, i.e., that it is in $f(x)$ for some input $x \in \mathcal{I}$.

Question 1.8. (Classic NP-completeness reduction) Show that VertexCover is NP-complete by reducing from IndependentSet.

Input	**:** A graph G and a number $k \in \mathbb{N}$
Output	**:** An set S of size k such that $(u, v) \in E \implies u \in S \vee v \in S$, if such a set exists

Computational Problem VertexCover

Example: If we formulate a graph shaped like a hexagon (with 6 vertices connected in a cycle), a valid vertex cover would be to take every other vertex in the hexagon. This is because for every edge in the graph, we now have one of its two endpoints in our vertex cover S. This latter sentence is a good informal approximation of the VertexCover output definition.

[2] In earlier iterations of class and section notes, you may have seen R and S be called R_{pre} and R_{post}, respectively, indicating that they are the pre- and post-processing parts of the reduction.

Question 1.9. (Subset Sum) The SubsetSum problem is the following:

Assuming this result, prove that the standard SubsetSum problem is also $\mathsf{NP}_{\mathsf{search}}$-complete:

Input	: Natural numbers $v_0, v_1, \ldots, v_{n-1}, t$
Output	: A subset $S \subseteq [n]$ such that $\sum_{i \in S} v_i = t$, if such a subset S exists.

Computational Problem SubsetSum

Example: Given the input $1, 5, 3, 8, 6, 2, 7$, a solution would be the subset $S = \{1, 6\}$ since $1 + 6 = t = 7$.

In the Sender–Receiver Exercise on November 15, you will prove that the following vector variant of SubsetSum is $\mathsf{NP}_{\mathsf{search}}$-complete:

Input	: Vectors $\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_{n-1} \in \{0,1\}^d$, $\vec{t} \in \mathbb{N}^d$
Output	: A subset $S \subseteq [n]$ such that $\sum_{i \in S} \vec{v}_i = \vec{t}$, if such a subset S exists

Computational Problem VectorSubsetSum

We will use the notation $\vec{v}[j]$ to denote the j'th entry of vector $\vec{v}$, so the condition $\sum_{i \in S} \vec{v}_i = \vec{t}$ means that for every $j = 0, 1, \ldots, d-1$, we have $\sum_{i \in S} \vec{v}_i[j] = \vec{t}[j]$.

Example: We have a small example with 3 vectors $\vec{v}_0, \vec{v}_1, \vec{v}_2, \vec{v}_3$ and $\vec{t} = [2, 1, 1, 2]$. We can write down all the vectors in a table like the following.

	0	1	2	3
$\vec{v}_0$	1	0	0	1
$\vec{v}_1$	1	0	0	1
$\vec{v}_2$	0	1	0	0
$\vec{v}_3$	0	1	1	0

We select vectors $\vec{v}_0, \vec{v}_1$, and $\vec{v}_3$ as our output.

Assuming the $\mathsf{NP}_{\mathsf{search}}$-completeness of VectorSubsetSum, prove that SubsetSum is $\mathsf{NP}_{\mathsf{search}}$-complete.

Question 1.10. ($\mathsf{NP}_{\mathsf{search}}$ vs. $\mathsf{NP}_{\mathsf{search}}$-complete) Show that if all problems in $\mathsf{NP}_{\mathsf{search}}$ are $\mathsf{NP}_{\mathsf{search}}$-complete, then $\mathsf{NP}_{\mathsf{search}} \subseteq \mathsf{P}_{\mathsf{search}}$. (You will prove the converse of this statement is on ps8, so it is actually an iff. Hint: 2-SAT is in $\mathsf{NP}_{\mathsf{search}}$.)

Question 1.11. (Programming Team, redux) Recall the Programming Team problem from Problem Set 7:

Input	: A finite set $L = \{\ell_0, \ldots, \ell_{m-1}\}$ of programming languages; a finite set $S = \{s_0, \ldots, s_{n-1}\}$ of students; for each student $s \in S$, a set $K(s) \subseteq L$ of languages that student s knows; and a team size $k \in \mathbb{N}$
Output	: A team $T \subseteq S$ of size k that collectively knows all of the programming languages in L (i.e. $\bigcup_{s \in T} K(s) = L$), if one exists

Computational Problem ProgrammingTeam

Prove that ProgrammingTeam is $\mathsf{NP}_{\mathsf{search}}$-complete, even in the special case where every language $\ell \in \mathcal{L}$ is known by exactly two students.