

## Lecture 20: NP-completeness

Harvard SEAS - Fall 2025

2025-11-11

## 1 Announcements

- Salil's in-person OH this week Thu 1-1:45pm SEC 3.327.
- Next SRE Thursday 11/13.

Recommended Reading:

- Hesterberg–Vadhan 21

## 2 Polynomial-Time Reductions

Although we do not know how to prove that many problems we care about are not in  $P_{\text{search}}$ , it turns out that we can get a much better understanding of their complexity via *reductions*. Since we have adopted “polynomial time” as our coarse notion of “efficiently solvable,” it makes sense to do the same for reductions:

**Definition 2.1.** For computational problems  $\Pi$  and  $\Gamma$ , we write  $\Pi \leq_p \Gamma$  if

Using polynomial-time reductions to compare problems fits nicely with the study of the classes  $P_{\text{search}}$  and  $P$ , since they are “closed” under such reductions:

**Lemma 2.2.** Let  $\Pi$  and  $\Gamma$  be computational problems such that  $\Pi \leq_p \Gamma$ . Then:

1.

2.

*Proof.* 1.

2. Contrapositive of Item 1

□

The vast majority of reductions we have seen in this book so far have been polynomial-time reductions. Typically, we have used these for *positive* results:

However, Item 2 shows that we can also use reductions for *negative* results, to give evidence that problems are not in  $\text{P}_{\text{search}}$ . As always, *the direction of the reduction is crucial!*

Another very useful feature of polynomial-time reductions is that they compose with each other:

**Lemma 2.3.** *If  $\Pi \leq_p \Gamma$  and  $\Gamma \leq_p \Theta$  then  $\Pi \leq_p \Theta$ .*

This follows by similar reasoning to Lemma 2.2.

### 3 Definition

Although it is widely conjectured, we unfortunately do not know how to prove that  $\text{NP}_{\text{search}} \not\subseteq \text{P}_{\text{search}}$ . As we will see in Chapter ??, this is an equivalent formulation of the famous P vs. NP problem, considered one of the most important open problems in computer science and mathematics. However, even without resolving the P vs. NP conjecture, we can give strong evidence that problems are not solvable in polynomial time by showing that they are  $\text{NP}_{\text{search}}$ -*complete*:

**Definition 3.1** (NP-completeness, search version). A problem  $\Gamma$  is  $\text{NP}_{\text{search}}$ -*complete* if:

1.

2.

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

**Proposition 3.2.** *Suppose  $\Gamma$  is  $\text{NP}_{\text{search}}$ -complete. Then  $\Gamma \in \text{P}_{\text{search}}$  iff  $\text{NP}_{\text{search}} \subseteq \text{P}_{\text{search}}$ .*

*Proof.*

□

In other words, if any  $\text{NP}_{\text{search}}$ -complete problem is in  $\text{P}_{\text{search}}$ , then all problems in  $\text{NP}_{\text{search}}$  are in  $\text{P}_{\text{search}}$ .

## 4 $\text{NP}_{\text{search}}$ -complete problems

Remarkably, there not only exist  $\text{NP}_{\text{search}}$ -complete problems, but some of them are quite natural. The first one we consider is SAT:

**Theorem 4.1** (Cook–Levin Theorem). *SAT is  $\text{NP}_{\text{search}}$ -complete.*

This can be interpreted as strong evidence that SAT is not solvable in polynomial time. If it were, then *every* problem in  $\text{NP}_{\text{search}}$  would be solvable in polynomial time. We will take the Cook–Levin Theorem on faith for now, and may sketch a proof for it in the lecture before Thanksgiving.

Once we have one  $\text{NP}_{\text{search}}$ -complete problem, we can get others via reductions from it.

**Theorem 4.2.** *3-SAT is  $\text{NP}_{\text{search}}$ -complete.*

*Proof.* The proof follows in two steps.

1. 3-SAT is in  $\text{NP}_{\text{search}}$ :

2. 3-SAT is  $\text{NP}_{\text{search}}$ -hard: Since every problem in  $\text{NP}_{\text{search}}$  reduces to SAT (by Theorem 4.1), all we need to show is  $\text{SAT} \leq_p \text{3-SAT}$  (since reductions compose, by Lemma 2.3).

The reduction algorithm from SAT to 3-SAT has the following components. First, we give an algorithm  $R$  which takes a SAT instance  $\varphi$  to a 3-SAT instance  $\varphi'$ .

$$\text{SAT instance } \varphi \xrightarrow{\text{polytime } R} \text{3SAT instance } \varphi'$$

Then we feed the instance  $\varphi'$  to our 3-SAT oracle and obtain a satisfying assignment  $\alpha'$  to  $\varphi'$  or  $\perp$  if none exists. If we get  $\perp$  from the oracle, we return  $\perp$ , else we transform  $\alpha'$  into a satisfying assignment to  $\varphi$  using another algorithm  $S$ .

$$\text{SAT assignment } \alpha \xleftarrow{\text{polytime } S} \text{3SAT assignment } \alpha'$$

Pictorially:

**Algorithm  $R$ :** The intuition behind this algorithm is that when we have a clause  $(\ell_0 \vee \ell_1 \vee \dots \vee \ell_{k-1})$  in the SAT instance  $\varphi$  (with too-large width  $k > 3$ ), we want to break it into multiple clauses of width  $\leq 3$ .

$R(\varphi)$ :	
<b>Input</b>	: A CNF formula $\varphi$
<b>Output</b>	: A CNF formula $\varphi'$ with each clause of width 3
<b>0</b>	$\varphi' = \varphi$
<b>1</b>	$i = 0$
<b>2</b>	<b>while</b> $\varphi'$ has a clause $C = (\ell_0 \vee \dots \vee \ell_{k-1})$ of width $k > 3$ <b>do</b>
<b>3</b>	Remove $C$
<b>4</b>	Add clauses _____
<b>5</b>	$i = i + 1$
<b>6</b>	<b>return</b> $\varphi'$

**Algorithm 4.1:  $R$**

Note that  $\varphi'$  is *not* an equivalent formula to  $\varphi$ . While  $\varphi$  is on variables  $z_0, \dots, z_{n-1}$ , the formula  $\varphi'$  is on variables  $z_0, \dots, z_{n-1}, y_0, \dots, y_{t-1}$ , where  $t$  is the number of iterations of the while loop.

**Algorithm  $S$ :** Given an assignment  $\alpha' = (\alpha_0, \dots, \alpha_{n-1}, \beta_0, \dots, \beta_{t-1})$  to  $\varphi'$ , the algorithm simply takes the first  $n$  bits, i.e.  $\alpha = (\alpha_0, \dots, \alpha_{n-1})$ .

Next we consider the runtime and correctness of the overall reduction algorithm.

**Runtime of the reduction algorithm:** We first consider the runtime of the algorithm  $R$ :

Then, we consider the runtime of the algorithm  $S$ , which is simply  $O(n)$ . Overall, the runtime of the reduction algorithm is  $O(nm)$ .

**Proof of correctness:** We need to show that if  $\varphi$  is satisfiable, then the reduction algorithm produces a satisfying assignment, and that if  $\varphi$  is unsatisfiable, the reduction algorithm outputs  $\perp$ . This proof relies on the following two claims.

**Claim 4.3.** *If  $\varphi$  is satisfiable then  $\varphi' = R(\varphi)$  is satisfiable.*

*Proof sketch.*

□

**Claim 4.4.** *If  $\alpha'$  satisfies  $\varphi' = R(\varphi)$ , then  $\alpha = S(\alpha')$  also satisfies  $\varphi$ .*

*Proof sketch.*

□

To finish the correctness proof, suppose  $\varphi$  is satisfiable. Then from Claim 4.3,  $\varphi'$  is also satisfiable. The 3-SAT oracle returns a satisfying assignment  $\alpha'$ , which is turned into a satisfying assignment for  $\varphi$  via the algorithm  $S$  (Claim 4.4). If  $\varphi$  is unsatisfiable, then by Claim 4.4,  $\varphi'$  is also unsatisfiable. In this case, the 3-SAT oracle returns  $\perp$ , so the reduction algorithm also returns  $\perp$ .

This completes the proof that 3-SAT is  $\text{NP}_{\text{search}}$ -complete.

## 5 Mapping Reductions

The usual strategy for proving that a problem  $\Gamma$  in  $\text{NP}_{\text{search}}$  is also  $\text{NP}_{\text{search}}$ -hard (and hence  $\text{NP}_{\text{search}}$ -complete) follows a structure similar to the proof of Theorem 4.2.

1. Pick a known  $\text{NP}_{\text{search}}$ -complete problem  $\Pi$  to try to reduce to  $\Gamma$ .
2. Come up with an algorithm  $R$  mapping instances  $x$  of  $\Pi$  to instances  $R(x)$  of  $\Gamma$ .
3. Show that  $R$  runs in polynomial time.
4. Show that if  $x$  has a valid output, then so does  $R(x)$ .
5. Conversely, show that if  $R(x)$  has an answer, then so does  $x$ . Moreover, we can transform valid answers to  $R(x)$  into valid answers to  $x$  through a *polynomial time* algorithm  $S$ .

Reductions with the structure outlined above are called *polynomial-time mapping reductions*, and they are what are typically used throughout the theory of  $\text{NP}$ -completeness. A formal definition is given in the textbook.

## 6 INDEPENDENT SET

Next we turn to INDEPENDENT SET, specifically the INDEPENDENT SET–THRESHOLD SEARCH version.

**Theorem 6.1.** INDEPENDENT SET is  $\text{NP}_{\text{search}}$ -complete.

*Proof.* We'll do this proof less formally than we did the proof of  $\text{NP}_{\text{search}}$ -completeness of 3-SAT.

1. In  $\text{NP}_{\text{search}}$ :

2.  $\text{NP}_{\text{search}}$ -hard: We will show  $\text{3-SAT} \leq_p \text{INDEPENDENT SET}$ .

Note that, whereas we've previously encoded many other problems in SAT, here we're going in the other direction and showing that SAT can be encoded by a graph problem.

Our reduction  $R(\varphi)$  takes in a CNF formula and produces a graph  $G$  and a size  $k$ . We'll use as a running example the formula

$$\varphi(z_0, z_1, z_2, z_3) = (\neg z_0 \vee \neg z_1 \vee z_2) \wedge (z_0 \vee \neg z_2 \vee z_3) \wedge (z_1 \vee z_2 \vee \neg z_3).$$

Our graph  $G$  consists of:

- Variable gadgets:

- Clause gadgets:

- Conflict edges:

We pick  $k = m + n$ , which implies that every independent set of size  $k$  must contain exactly one vertex from each variable gadget and one vertex from each clause gadget. An algorithm  $R$  can create this graph (and  $k$ ) in polynomial time given  $\varphi$ . The graph for the formula  $\varphi$  is below.

**Claim 6.2.** *G has an independent set of size  $k = n+m$  if and only if  $\varphi$  is satisfiable. Moreover, we can map independent sets in G of size  $k$  to satisfying assignments of  $\varphi$  in polynomial time.*

*Proof of claim.*

□

This completes the proof that INDEPENDENT SET is NP<sub>search</sub>-complete. □

It is also known that LONG PATH and 3-D MATCHING are NP<sub>search</sub>-complete; a proof of the latter is in optional reading in the textbook.