CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 23: NP and NP-completeness

Harvard SEAS - Fall 2021 *Nov. 23, 2021*

1 Announcements

- Happy Thanksgiving!
- Salil OH after class
- Revised guidelines for 2nd participation portfolio (due 12/5) are posted

Recommended Reading:

- MacCormick §12.0–12.3

2 Proving Intractability

Last time we defined the complexity classes $\mathsf{P}_{\mathsf{search}}$ and $\mathsf{EXP}_{\mathsf{search}}$ and their restrictions to decision problems, P and EXP. We *conjectured* that 3-Coloring, 3-SAT, Independent Set, and Longest Path are in $\mathsf{EXP}_{\mathsf{search}} - \mathsf{P}_{\mathsf{search}}$, but it could be that some of them or all of them are in $\mathsf{P}_{\mathsf{search}}$. So we have many possible worlds:

How can we *prove* that some of these problems are not in $\mathsf{P}_{\mathsf{search}}$?

1. Identify a particular problem Π in $\mathsf{EXP}_{\mathsf{search}} - \mathsf{P}_{\mathsf{search}}$. One example is deciding whether a Word-RAM program halts within 2^n steps on an input x of length n.
2. Show that Π reduces to the problems we are interested in, via a *polynomial-time reduction*.

Definition 2.1. 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).

Examples we've seen:
LongPath $\leq_p$ SAT
GraphColoring $\leq_p$ SAT

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

1. *If $\Gamma \in \mathsf{P}_{\mathsf{search}}$, then $\Pi \in \mathsf{P}_{\mathsf{search}}$.*
2. *If $\Pi \notin \mathsf{P}_{\mathsf{search}}$, then $\Gamma \notin \mathsf{P}_{\mathsf{search}}$.*

Proof.

1. Since Π reduces in polynomial time to Γ, there is some $O(n^c)$ time reduction R solving Π on inputs of size R given an oracle that solves Γ. We assume that Γ is in P, i.e. there is an $O(n^d)$ algorithm A solving Γ on inputs of length n.

   Then we can create an algorithm that, given an input x to Π, runs R on x. Whenever R calls the oracle on y_i, we instead evaluate $A(y_i)$. The runtime is

   $$O(n^c + \#\{\text{Calls made to oracle}\} \cdot B)$$

   where B is an upper bound on the size of the oracle calls. Then R calls the oracle at most $O(n^c)$ times. Furthermore, given that the reduction is in $O(n^c)$, each call is on an input of size of at most $O(n^c)$. This gives a total runtime bound of $O(n^c + n^c(n^c)^d) = O(n^{c\cdot(d+1)}$.

2. Contrapositive of 1.

These runtime blowups are acceptable because we are still inside P. If we had defined computationally efficient as e.g. quadratic time, we wouldn't be able to compose reductions in this way. □

Remark 2.3. We can compose reductions - if $\Pi \leq_p \Gamma$ and $\Gamma \leq_p \Theta$ then $\Pi \leq_p \Theta$.

Unfortunately, we don't know how to reduce the problems we know in $\mathsf{EXP}_{\mathsf{search}} - \mathsf{P}_{\mathsf{search}}$ (like Bounded Halting) to many of the problems we care about (like Independent Set, 3-Coloring, and Longest Path). These problems have additional structure, which will require us to define and study a different complexity class, NP.

3 NP

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?

It will be convenient to restrict attention to problems where the output symbol $\perp$ is always used to mean that there are no other solutions. (This is consistent with how we've been using $\perp$ all along in defining search problems.)

Definition 3.1. A computational problem $\Pi = (\mathcal{I}, f)$ is $\perp$-*proper* if for every $x \in \mathcal{I}$, we have $f(x) \neq \emptyset$ and either $f(x) = \{\perp\}$ or $\perp \notin f(x)$.

Definition 3.2. A computational problem $\Pi = (\mathcal{I}, f)$ is in $\mathsf{NP}_{\mathsf{search}}$ if it is $\perp$-proper and the following conditions hold:

1. 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. solutions are verifiable in polynomial time: Given $x \in \mathcal{I}$ and a potential solution $y \neq \perp$, we can decide whether $y \in f(x)$ in P.

(Remark on terminology: $\mathsf{NP}_{\mathsf{search}}$ is often called FNP in the literature, and is closely related to, but slightly more restricted than, the class $\mathsf{PolyCheck}$ defined in the MacCormick text.)

Examples:

1. Satisfiability:

$$\mathcal{I} = \{\text{Boolean forumlas } \phi(x_1, \ldots, x_n)\}$$
$$f(x) = \begin{cases} \{\alpha : \phi(\alpha) = 1\} & \phi \text{ is satisfiable} \\ \perp & \text{otherwise.} \end{cases}$$

   We can see this is $\perp$-proper by inspection. To check that it is in $\mathsf{NP}_{\mathsf{search}}$, we can verify if an assignment α satisfies ϕ in polynomial time by substituting in α and checking whether $\phi(\alpha) = 1$. Furthermore, we can check that the solutions are in polynomial length. Note that $|\alpha| \leq |\phi|$ so the solutions are not too long.

2. GraphColoring:

$$f(G, k) = \begin{cases} \{c : V \to [k] \text{ a proper } k \text{ coloring}\} & G \text{ is } k\text{-colorable} \\ \perp & \text{otherwise.} \end{cases}$$

   Our verifier takes in $c : V \to [k]$ and checks that for every edge (u, v), $c(u) \neq c(v)$, which runs in time $O(m)$. Equivalently, we can check that every color class defines an independent set. Furthermore, $|c| = n\lceil \log k \rceil$, so the solution is not too long.

Non-Example:

1. MinGraphColoring:

$$f(G) = \{\text{proper coloring } c : V \to [k] \text{ s.t. } k \text{ is the minimum number of colors needed}\}$$

   Here, it's not clear how to check that a valid k-coloring has the smallest possible number of colors. For instance, if we're given a valid 4-coloring of G, how can we determine if there isn't a valid 3-coloring?

Even though this problem does not appear to be in $\mathsf{NP}_{\mathsf{search}}$, it reduces in polynomial time to a problem in $\mathsf{NP}_{\mathsf{search}}$.

For instance, MinGraphColoring $\leq_p$ GraphColoring, since we can first ask our GraphColoring oracle if there is a 1-coloring, then a 2-coloring, and continue on until the answer is TRUE.

Every problem in $\mathsf{NP}_{\mathsf{search}}$ can be solved in exponential time:

Proposition 3.3. $\mathsf{NP}_{\mathsf{search}} \subseteq \mathsf{EXP}_{\mathsf{search}}$.

Proof.
Exhaustive search! We can enumerate over all possible solutions and check if any is a valid solution.

```
ExhaustiveSearch
  Input     : x
for y ≠⊥ such that |y| ≤ p(|x|) do
    if V(x, y) = accept then
        return y
return ⊥
```

This has runtime $O(2^{p(n)}(n + p(n))^2)$ which is certainly exponential.

□

Every problem in $\mathsf{NP}_{\mathsf{search}}$ has a corresponding decision problem (deciding whether or not there is a solution). The class of such decision problems is called NP and we will study it more next time.

4 NP-Completeness

Unfortunately, although it is widely conjectured, we do not know how to prove that $\mathsf{NP}_{\mathsf{search}} \not\subseteq \mathsf{P}_{\mathsf{search}}$. As we will see next time, 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 NP-*complete*:

Definition 4.1 (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 4.2. *Suppose* Π *is* $\mathsf{NP}_{\mathsf{search}}$*-complete. Then* $\Pi \in \mathsf{P}_{\mathsf{search}}$ *iff* $\mathsf{NP}_{\mathsf{search}} \subseteq \mathsf{P}_{\mathsf{search}}$.

Remarkably, there are natural NP-complete problems. The first one is Satisfiability:

Theorem 4.3 (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. We won't cover (or expect you to know) the proof of the Cook–Levin Theorem, but in case you are interested, a proof sketch is in Section 5.

Just like with Unsolvability, once we have one NP-complete problem, we can get others via reductions from it, as we'll see next time.

5 Proof Sketch of the Cook–Levin Theorem

We have already seen that Satisfiability is in $\mathsf{NP}_{\mathsf{search}}$, so to prove Theorem **??**, we only need to prove that every problem Γ in $\mathsf{NP}_{\mathsf{search}}$ reduces to Satisfiability. You've already seen how some problems in $\mathsf{NP}_{\mathsf{search}}$ (like k-Coloring and LongPath) can be reduced to Satisfiability. The general proof of this is somewhat technical, so we will just outline the ideas here.

1. Since Γ is in NP, there is a polynomial-time Word-RAM program V that verifies valid solutions for Π.
2. Given an instance x of length N and a potential solution y of length at most $p(N)$, the running time of $V(x, y)$ is at most $T = N^c$ for a constant c.
3. Given the bound of $T = N^c$ on V's runtime, we can unroll V's loops (i.e. make all GOTOs go to a higher line number) and also assume without loss of generality that V's word size is bounded by $w = O(\log T) = O(\log N)$.
4. Similarly to the Write-Free RAM problem on Problem Set 10, the problem of finding a solution y that makes $V(x, y)$ accept can be reduced to a SMT problem, but over the Theory of Bitvectors, i.e. w-bit words, instead of the Theory of Natural Numbers.

5. We can reduce the constraints involving words like $\texttt{var}_i = \texttt{var}_j\texttt{opvar}_k$ to constraints over propositional, by introducing w propositional variables for the individual bits of each theory variables and then turning each constraints like $\texttt{var}_i = \texttt{var}_j\texttt{opvar}_k$ into w CNF constraints (one for each bit of $\texttt{var}_i$), each with at most 2^{2w} clauses (since there $2w$ bits among $\texttt{var}_j$ and $\texttt{var}_k$).