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

Lecture 19: NP and NP-completeness

Harvard SEAS - Fall 2022 *Nov. 8, 2022*

1 Announcements

- Happy Election Day! Local polls close 8pm
- PS7 due tomorrow, PS8 due Friday 11/18
- PS9 due Friday 12/2
- Next SRE on Tuesday 11/15
- Stay tuned for updates to office hours and section next week

Recommended Reading:

- MacCormick §12.0–12.3, Ch. 13

2 Recap

Recall that $\mathsf{TIME}_{\mathsf{search}}(T(N))$ is the class of computational problems $\Pi = (\mathcal{I}, \mathcal{O}, f)$ such that there is a Word-RAM program solving Π in time $O(T(N))$ on inputs of bit-length N. $\mathsf{TIME}(T(N))$ is the class of decision problems in $\mathsf{TIME}_{\mathsf{search}}(T(N))$. We can define classes for $\mathsf{P}_{\mathsf{search}}$, P and $\mathsf{EXP}_{\mathsf{search}}$, EXP as follows:

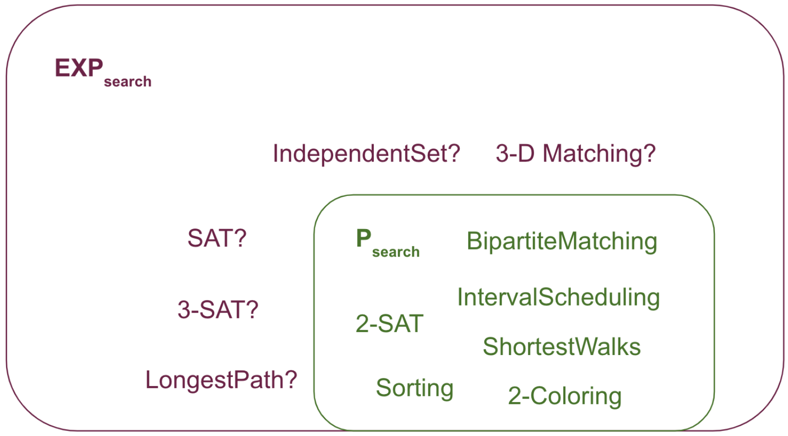
- (Polynomial time)

$$\mathsf{P}_{\mathsf{search}} = \bigcup_c \mathsf{TIME}_{\mathsf{search}}(n^c), \qquad \mathsf{P} = \bigcup_c \mathsf{TIME}(n^c)$$

- (Exponential time)

$$\mathsf{EXP}_{\mathsf{search}} = \bigcup_c \mathsf{TIME}_{\mathsf{search}}\left(2^{n^c}\right), \qquad \mathsf{EXP} = \bigcup_c \mathsf{TIME}\left(2^{n^c}\right).$$

The following diagram captures our current understanding of the complexity classifications of computational problems we have seen (or will see) in CS120.



The question marks indicate that we don't *know* that the problems in red are actually outside $\mathsf{P}_{\text{search}}$; we just have not found polynomial-time algorithms for them. To try to get a handle on these questions, we will introduce a new complexity class $\mathsf{NP}_{\text{search}}$ that captures some shared structure that they all have.

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?

Definition 3.1. A computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ is in $\mathsf{NP}_{\text{search}}$ 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 a polynomial-time verifier V that, given $x \in \mathcal{I}$ and a potential solution y,[1] decides whether $y \in f(x)$.

(Remark on terminology: $\mathsf{NP}_{\text{search}}$ 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:

$$\mathcal{I} = \{\text{Boolean formulas } \varphi(x_1, \ldots, x_n), n \in N\}$$

$$\mathcal{O} = \{\text{Assignments } \alpha \in \{0,1\}^n, n \in \mathbb{N}\}$$

$$f(x) = \{\alpha : \phi(\alpha) = 1\}$$

[1]Note that we do not assume $y \in \mathcal{O}$, so the verifier should reject if $y \notin \mathcal{O}$, i.e. y is ill-formed.

We can verify if a potential assignment α satisfies ϕ in polynomial time by (a) checking that α is indeed a valid assignment (i.e. an array of 0's and 1's), and (b) substituting α into φ and checking whether $\varphi(\alpha) = 1$. Note that $|\alpha| = n \leq |\varphi|$ so the solutions are of polynomial length.

2. GraphColoring:
$$f(G, k) = \{c : V \to [k] \text{ a proper } k \text{ coloring}\}$$
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, $|c| = n\lceil \log k \rceil \leq |(G, k)|^2$, so the solution is not too long.

Non-Example:

1. IndependentSet-OptimizationSearch:
$$f(G) = \{S \subseteq V : S \text{ is an independent set in } G \text{ of maximum } \textit{size}\}$$

Even though this problem does not appear to be in $\mathsf{NP}_{\mathsf{search}}$ (why?), you saw on Problem Set 5 that it reduces in polynomial time to a problem in $\mathsf{NP}_{\mathsf{search}}$. (Which one?)

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

Proposition 3.2. $\mathsf{NP}_{\mathsf{search}} \subseteq \mathsf{EXP}_{\mathsf{search}}$.

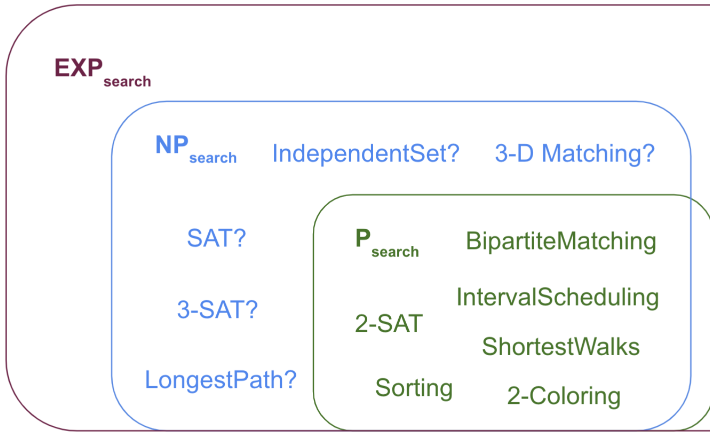
Proof.
Exhaustive search! We can enumerate over all possible solutions and check if any is a valid solution.

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

This has runtime $O(2^{p(n)} \cdot (n + p(n))^c)$ which is bounded by the exponential $O(2^{n^d})$, where $d = \deg(p) + 1$.

$\square$

So now our diagram of complexity classes looks like this:



(Note that $\mathsf{P}_{\mathsf{search}} \not\subseteq \mathsf{NP}_{\mathsf{search}}$. This due to artificial examples that you may see on PS9, but most natural problems in $\mathsf{P}_{\mathsf{search}}$ are also in $\mathsf{NP}_{\mathsf{search}}$ (like all of the green problems in the above diagram).) 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 week.

We still have question marks next to all of the blue problems; we don't know whether they (and thousands of other important problems in $\mathsf{NP}_{\mathsf{search}}$) are in $\mathsf{P}_{\mathsf{search}}$ or not. We will now try to get a handle on these questions.

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 CNF-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 we may provide you a proof sketch in the last set of lecture notes in the course.

5 More $\mathsf{NP}_{\mathsf{search}}$-complete Problems

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

Theorem 5.1. *3-SAT is* $\mathsf{NP}_{\mathsf{search}}$*-complete.*

Proof.

1. 3SAT is in $\mathsf{NP}_{\mathsf{search}}$: Our verifier can check if an assignment α satisfies the 3CNF formula (the same verifier as for SAT).
2. 3SAT is $\mathsf{NP}_{\mathsf{search}}$-hard: Since every problem in NP reduces to SAT, all we need to show is SAT $\leq_p$ 3SAT (since reductions are transitive).

For part (2) we follow a general reduction template. First, we transform the problem from what we want to solve to what we have an oracle for.

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

Then we feed the instance φ' to our 3SAT oracle and obtain a satisfying assignment α' to φ' or $\perp$ if none exists. If we get $\perp$ from the oracle, we return $\perp$, else we transform α' into a satisfying assignment to φ.

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

Most of the work is usually in coming up with the reduction R. Intuitively, when we have long clause $(\ell_0 \vee \ell_1 \vee ... \vee \ell_{k-1})$ for $k > 3$ we want to break it into multiple clauses of size 3. But simply breaking it up doesn't preserve information about φ being satisfiable. Our reduction R is as follows:

$R(\varphi)$:
Input : A CNF formula φ
Output : A 3-CNF formula φ'
$\varphi' = \varphi$
while φ' *has a clause* $C = (\ell_0 \vee \ldots \vee \ell_{k-1})$ *of length* $k > 3$ **do**
  Remove C
  Add clauses $(y \vee \ell_0 \vee \ell_1)$ and $(\neg y \vee \ell_2 \ldots \ell_{k-1})$, where y is a new variable
return φ'

This is **not** an equivalent formula to the original (we introduced potentially many dummy variables), but it preserves what we care about — φ' is satisfiable iff φ is (as we'll prove below). In fact, this reduction is the "reverse" of the Resolution rule! Indeed, $C = (y \vee \ell_0 \vee \ell_1) \diamond (\neg y \vee \ell_2 \ldots \ell_{k-1})$.

We need to check that R runs in polynomial time: At each iteration of the while loop, we take a clause of length k and produce clauses of length 3 and $k-1$. Thus, the total length of too-large clauses goes down by 1 at each step, so the procedure terminates. In fact, the number of iterations is bounded by $\sum_{C \in \varphi, |C|>3} |C| \leq nm$ where $|C|$ is the width of the clause.

Claim 5.2. *If* φ *is satisfiable then* $\varphi' = R(\varphi)$ *is satisfiable.*

Proof of claim. Assume that φ is satisfiable. Let $\varphi = \varphi_0, \varphi_1, \ldots, \varphi_t = R(\varphi)$ be the formula φ' as it evolves through the t loop iterations. We will prove by induction on i that φ_i is satisfiable for $i = 0, \ldots, t$. constructed through the t loop iterations.

Base case ($i = 0$): $\varphi_0 = \varphi$, which is satisfiable by hypothesis.

Induction step: By the induction hypothesis, we can assume that φ_{i-1} is satisfiable, and now we need to show that φ_i is satisfiable:

Suppose α_{i-1} is a satisfying assignment to φ_{i-1}, and we obtain φ_i from it by breaking up clause $C = (\ell_0 \vee ... \vee \ell_k)$. Then since α satisfies C, it satisfies at least one of $(\ell_0 \vee \ell_1)$ and $(\ell_2 \vee ... \vee \ell_k)$. If it satisfies the first, we can set $y = 0$ and obtain an assignment α_i that satisfies both $(y \vee \ell_0 \vee \ell_1)$ and $(\neg y \vee \ell_2 \ldots \ell_{k-1})$ and hence φ_i. φ_i. In the second case, we can set $y = 1$. Thus, we've maintained that a satisfying assignment exists. □

Finally, we need to show we can transform a satisyfing assignment α' to φ' into a satisfying assignment α to φ. Our S simply discards all introduced dummy y variables and takes the assignment to the x variables.

Claim 5.3. *If α' satisfies $R(\varphi)$, then $\alpha'|_x$ also satisfies φ, where $\alpha'|_x$ is the restriction of the assignment α' to the x variables.*

Proof of claim. We prove by "backwards induction" that α' satisfies φ_i for $i = t, \ldots, 0$. We can then drop the extra t variables that don't appear in φ without changing the satisfiability. (We call this "backwards induction" since our base cases is $i = t$.)

The base case $(i = t)$ follows because α' satisfies $R(\varphi) = \varphi_t$ by assumption.

For the induction step: Suppose by induction that α' satisfies φ_i, and now we want to show that it also satisfies φ_{i-1}. φ_i was constructed from φ_{i-1} by breaking up some clause $C = (\ell_0 \vee ... \vee \ell_k)$ into $(y \vee \ell_0 \vee \ell_1) \wedge (\neg y \vee \ell_2 \vee \ldots \vee \ell_k)$. By assumption α' satisfies the two new clauses. But then, by the soundness of the resolution rule, α' also satisfies $(y \vee \ell_0 \vee \ell_1) \diamond (\neg y \vee \ell_2 \vee \ldots \vee \ell_k) = C$.

□

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