CS120: Intro. to Algorithms and their Limitations	Anshu & Hesterberg
Lecture 19: NP and NP-completeness	
Harvard SEAS - Fall 2023	*Nov. 9, 2023*

1 Announcements

- PS7 due Nov 15
- PS8 out Nov 14
- Next SRE on Thu Nov 16

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:

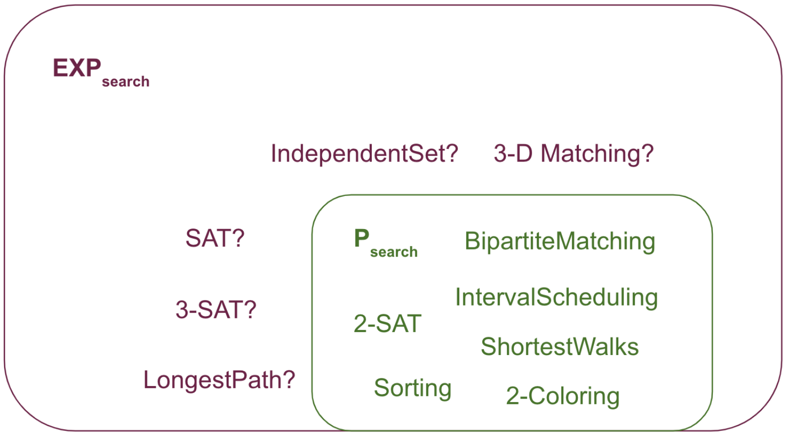
- (Polynomial time)

$$\mathsf{P}_{\mathsf{search}} = \qquad\qquad\qquad \mathsf{P} = \qquad\qquad\qquad .$$

- (Exponential time)

$$\mathsf{EXP}_{\mathsf{search}} = \qquad\qquad\qquad \mathsf{EXP} = \qquad\qquad\qquad .$$

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 $\mathsf{P}_{\mathsf{search}}$; 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 $\mathsf{NP}_{\mathsf{search}}$ that captures some shared structure that they all have.

3 NP

Figure 1: Can you find a cat?

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 $\mathsf{NP}_{\mathsf{search}}$ if the following conditions hold:

1. All solutions are of polynomial length:

2. All solutions are verifiable in polynomial time:

(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:

2. GraphColoring:

Non-Example:

1. IndependentSet-OptimizationSearch:

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?)

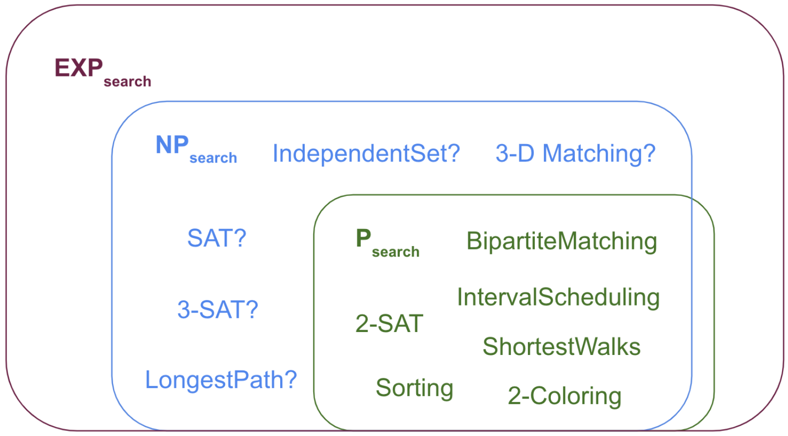
Every problem in $\mathsf{NP}_{\mathsf{search}}$ can be solved in exponential time:

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

Proof.

$\square$

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.

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.

2.

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 some intuition is as follows.

A sketch of the proof of Theorem 4.3: Given a computational problem $\Gamma \in \mathsf{NP}_{\mathsf{search}}$ and a $x \in \mathcal{I}$, lets define the following function that inputs y and outputs a bit:

$$g_x(y) = \begin{cases} 1 & \text{if } y \in f(x) \\ 0 & \text{otherwise} \end{cases}.$$

For this discussion, let's assume that x, y by itself are binary strings (if it's not, then we can include another function encoding them as binary strings, but we can ignore such nuances for now). If the number of bits of x is n, then g_x is a function $\{0,1\}^{p(n)} \to \{0,1\}$. In Lemma 2.3 in Lecture 15, we saw that there is a CNF ψ_x such that for all y, $\psi_x(y) = g_x(y)$. This is a reduction to SAT, but a major issue is that

However, note that there is a polynomial-time algorithm that solves the computational problem $(\{0,1\}^{p(n)}, \{0,1\}, g_x)$ -

At the heart of the Cook-Levin Theorem (Theorem 4.3) is a construction that significantly improves upon Lemma 2.3 from Lecture 15, giving a CNF of size polynomial in n, whenever there is a polynomial-time algorithm for the computational problem! We will skip the details, but as an illustrative example, suppose $p(n) = 3n$ and the verifier algorithm takes n different majorities $z_0 = \phi_{maj}(y_0, y_1, y_2)$, $z_1 = \phi_{maj}(y_3, y_4, y_5)$, $\ldots z_{n-1} = \phi_{maj}(y_{3n-3}, y_{3n-2}, y_{3n-1})$ and then outputs $(z_0 \vee z_1 \vee \ldots z_{n-1})$. Then the SAT instance is much shorter:

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 $\mathsf{NP}_{\mathsf{search}}$:

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
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).

We need to check that R runs in polynomial time:

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 ($i = 0$):

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:

$\square$

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:

$\square$

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