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

Lecture 19: NP and NP-completeness

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

1 Announcements

Recommended Reading:

- MacCormick §12.0–12.3, Ch. 13, 14, 17

2 Recap

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 $\mathsf{NP}_{\mathsf{search}}$ 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 $y \in \mathcal{O}$, decides whether $y \in f(x)$.

(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:
$$\mathcal{I} = \{\text{Boolean formulas } \phi(x_1, \ldots, x_n), n \in N\}$$
$$\mathcal{O} = \{\text{Assignments } \alpha \in \{0,1\}^n, \in \mathbb{N}\}$$
$$f(x) = \{\alpha : \phi(\alpha) = 1\}$$
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:
$$f(G, k) = \{c : V \to [k] \text{ a proper } k \text{ coloring}\}$$
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, $|c| = n\lceil \log k \rceil$, 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    : $x \in \mathcal{I}$
for $y \in \mathcal{O}$ such that $|y| \le p(|x|)$ do
    if $V(x, y) = accept$ then
        return $y$
return $\perp$
```

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

□

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 $\Gamma \in \mathsf{NP}_{\mathsf{search}}$, $\Gamma \le_p \Pi$.

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 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 an assignment α' (or $\perp$ if none exists). Finally:

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

In this case, we need to give the reduction R. Intuitively, when we have long clause $(x_1 \vee x_2 \vee ... \vee x_k)$ 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:

```
1 $R(\varphi)$ :
  Input  : A CNF formula $\varphi$
  Output : A 3-CNF formula $\varphi'$
2 $\varphi' = \varphi$
3 while $\varphi'$ has a clause $C = (\ell_0 \vee \ldots \vee \ell_k)$ of length $k > 3$ do
4     Remove $C$
5     Add clauses $(y \vee \ell_0 \vee \ell_1)$ and $(\neg y \vee \ell_2 \ldots \ell_k)$, where $y$ is a new variable
6 return $\varphi'$
```

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!

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 length of the clause.

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

Proof of claim. Assume that φ is satisfiable. We prove that φ' is satisfiable throughout the algorithm by induction on the number of loop iterations.

Base case: Initially $\varphi' = \varphi$ and we assumed that φ is satisfiable.

Induction step: By the induction hypothesis, we can assume that φ' is satisfiable after m loop iterations, and now we need to show that it will remain satisfiable after $m+1$ iterations:

Suppose α is a satisfying assigment to φ' after the m^{th} loop iteration. And suppose we break up $C = (\ell_0 \vee ... \vee \ell_k)$ in the $(m+1)$'st loop iteration. 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' that satisfies the new formula. 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 $\varphi' = R(\varphi)$, then α'_x also satisfies φ, where α'_x is the restriction of the assignment α' to the x variables.*

Proof of claim. We prove that α' satisfies φ' throughout the algorithm by *backward* induction on the loop. By assumption, we know that it satisfies φ' at the end of the last loop (this our base case) and we want to prove that it satisfies φ at the beginning of the first loop.

For the induction step: suppose that α' satisfies φ' at the end of loop iteration m, and now we want to show that it satisfies φ' at the beginning of loop iteration m (equivalently, the end of loop iteration $m-1$). In loop iteration m we break up some clause C 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. One of y and $\neg y$ is false under α' (since α' assigns y a single value). If y is false, α' must satisfy $(\ell_0 \vee \ell_1)$, and so satisfies the original pre-split clause C. An analogous argument holds if $\neg y$ is false.

□

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

Theorem 5.4. *IndependentSet is $\mathsf{NP}_{\mathsf{search}}$-complete.*

Proof. We'll do this proof less formally.

1. In $\mathsf{NP}_{\mathsf{search}}$: The verifier checks if the set S (claimed to be an iset of size at least k in G) is actually of size at least k and is actually independent, which can be done in polynomial time.

2. $\mathsf{NP}_{\mathsf{search}}$-hard: We will show 3SAT $\leq_p$ IndSet.

We've previously encoded many other problems in SAT, but here we're going in the other direction and showing a graph problem can encode SAT.

Our reduction $R(\varphi)$ takes in a CNF and produces a graph G and a size k.

$$\varphi(x_0, x_1, x_2, x_3) = (\neg x_0 \vee \neg x_1 \vee x_2) \wedge (x_0 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3).$$

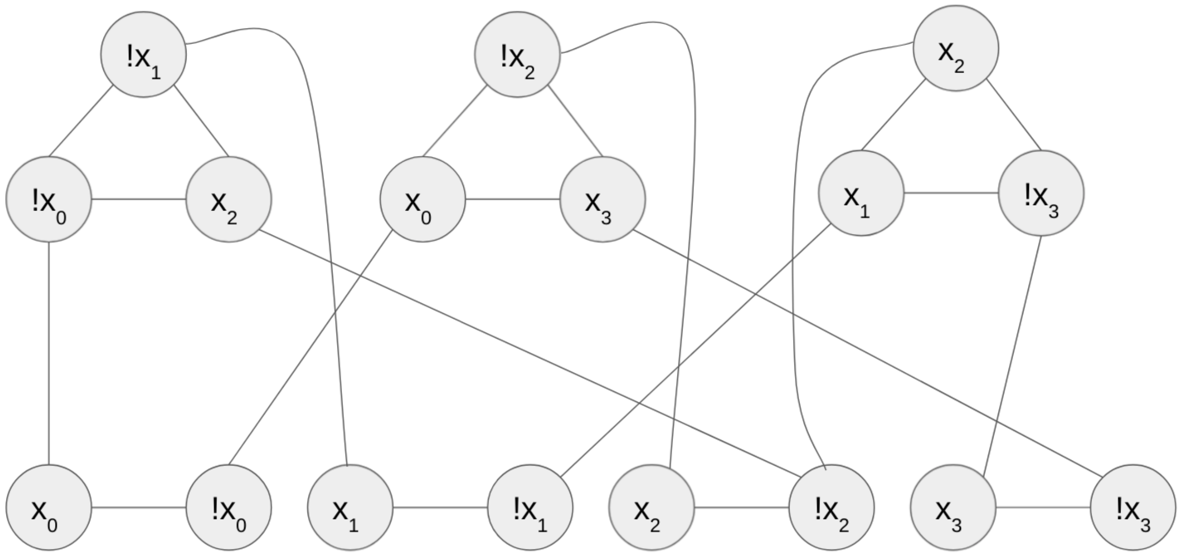
Our graph G consists of:

- Variable gadgets: these are dumbbells (two vertices connected by an edge) labelled by a variable x and its negation $\neg x$, capturing the fact that only one of these two literals can be true.

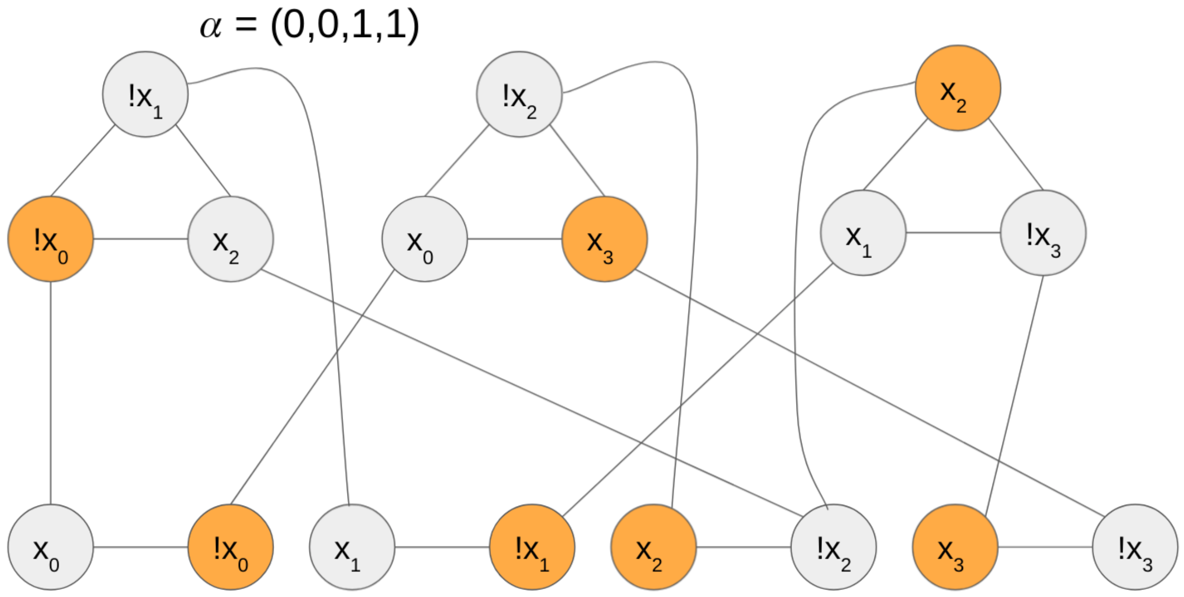
- Clause gadgets: these are triangles whose vertices are labelled by the literals in a clause, and will capture the fact that a sastifying assignment must satisfy at least one element of each clause.

- Consistency edges: We place edges between the vertices of variable gadgets and clause gadgets that are labelled by opposite-signed literals, which enforce the relation between an assignment to variables and satisfying the clauses.

An algorithm R can create this graph in polynomial time given φ. Here is an example for the formula φ above (using $!x$ to mean $\neg x$):



Note that (analogously to the SAT to 3SAT case) the correspondence between 3SAT and ISET does not exactly preserve the set of satisfying solutions (they aren't even the same problem) but we can go from solutions to one to solutions to the other:



Claim 5.5. *G has an independent set of size $k = n + m$ if and only if φ is satisfiable. Moreover, we can map independent sets of size k to satisfying assignments of φ in polynomial time.*

Proof of claim.
Given a satisfying assignment α to φ, we can pick one vertex in each variable and clause gadget and have them all be independent. For each variable gadget, pick the vertex corresponding to the assignment in α. For each clause gadget, pick a single vertex that corresponds to a literal satisfied by α. (If α satisfies more than one literal in the clause, we can pick one arbitrarily. We can't pick more than one since an independent set can only have one vertex from any triangle.)

A similar proof in the other direction shows that given an independent set of size $n + m = k$ in G, we can recover a satisfying assignment to φ. Specifically, if we have an independent set of size $n + m$ in G, it must contain exactly one vertex from each variable gadget and exactly one vertex from each clause gadget (else it would be of size smaller than $n+m$). Then we take our assignment α according to the vertices chosen from the variable gadget. The vertices chosen from the clause gadgets certify that at least one literal is satisfied in each clause.

$\square$

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