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

Lecture 24: NP-completeness

Harvard SEAS - Fall 2021 *Nov. 30, 2021*

1 Announcements

Recommended Reading:

- MacCormick Ch. 13, 14, 17
- PS11 is now a half pset due Sunday 12/5
- Next Tuesday 12/7 9:45-11 review session (taught by Salil)
- TFs will hold review sessions over exam period
- Model participation portfolio examples posted
- Last active learning Thursday!

2 More NP-complete Problems

Last time we saw:

Theorem 2.1 (Cook–Levin Theorem)**.** *SAT is* $\mathsf{NP}_{\mathsf{search}}$*-complete.*

Just like with Unsolvability, once we have one NP-complete problem, we can get others via reductions from it.

Theorem 2.2. *3-SAT is* $\mathsf{NP}_{\mathsf{search}}$*-complete.*

Proof.

1. 3SAT is in $\mathsf{NP}_{\mathsf{search}}$, since our verifier can check if an assignment α satisfies the 3CNF formula (the exact same verifier works 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. Inutitively, 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(φ) :
  Input  : A CNF formula φ
  Output : A 3-CNF formula φ'
2 φ' = φ
3 while φ' has a clause C = (ℓ_0 ∨ ... ∨ ℓ_k) of length k > 3 do
4 |   Remove C
5 |   Add clauses (y ∨ ℓ_0 ∨ ℓ_1) and (¬y ∨ ℓ_2 ... ℓ_k).
6 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!

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.

We now need to show that if φ was satisfiable, so is φ'.

Claim 2.3. *If φ is satisfiable then φ' is satisfiable.*

Proof of claim. Suppose α is a satisfying assigment to φ' at the start of a loop iteration. Then suppose we break up $C = (\ell_0 \vee ... \vee \ell_k)$. 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' 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 2.4. *If α' satisfies φ', then α'_x also satisfies φ, where α'_x is the restriction of the assignment α' to the x variables.*

Proof of claim. We again look at each iteration of the loop. Suppose φ' satisfies $(y \vee \ell_0 \vee \ell_1) \wedge (\neg y \vee \ell_2 \vee \ldots \vee \ell_k)$. Then one of y and $\neg y$ is false (since α' assigns it 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.

□

Theorem 2.5. *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 (similarly to how you encoded 2-SAT using Reachability on ps8).

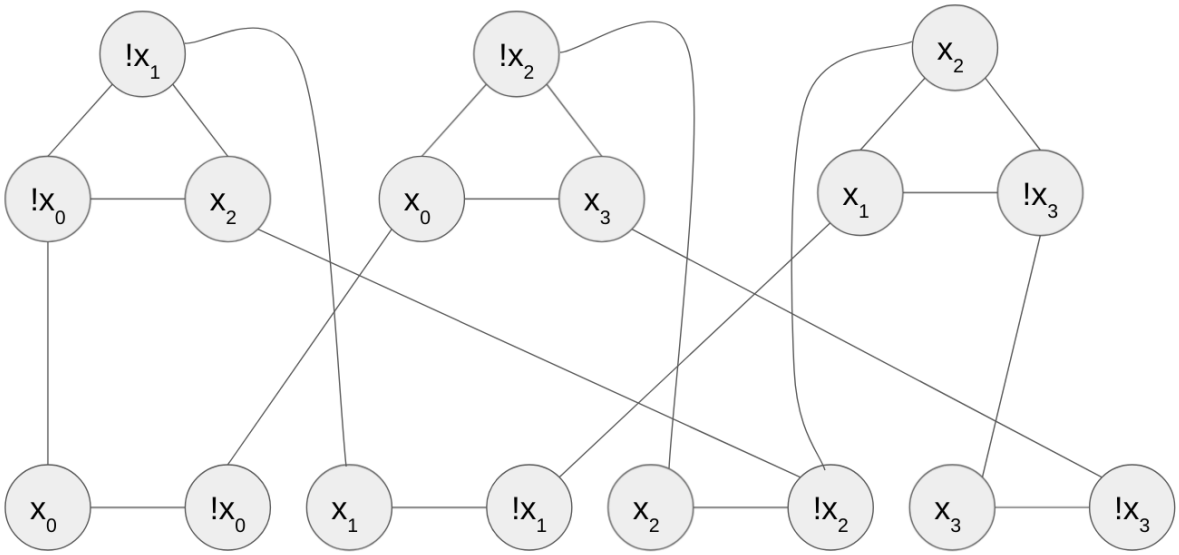
Our reduction $R(\varphi)$ takes in a CNF and produces a graph G and a size k.

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

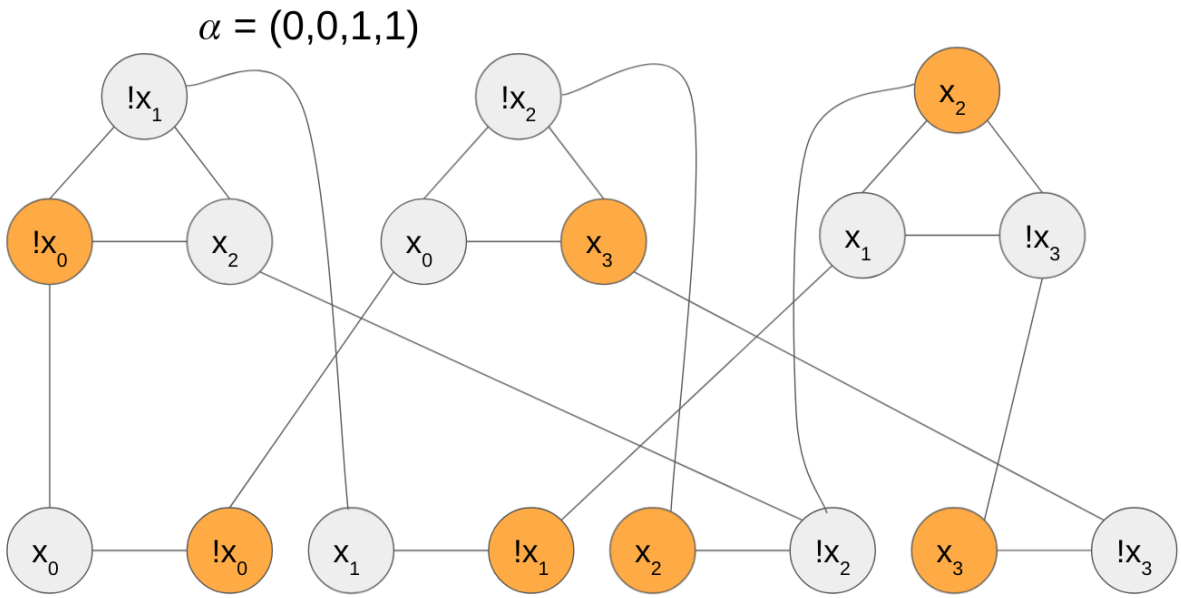
Our graph G consists of:

- 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.
- 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 2.6. *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. □

□

What is the usual strategy for proving that a problem Γ is $\mathsf{NP}_{\mathsf{search}}$-complete?

1. Pick a known $\mathsf{NP}_{\mathsf{search}}$-complete problem Π to try to reduce to Γ. Typically, we might try to pick a problem Π that seems as similar as possible to Γ, or which has been used to prove that problems similar to Γ are $\mathsf{NP}_{\mathsf{search}}$-complete. Otherwise 3-SAT is often a good fallback option.

2. Come up with an algorithm R mapping instances x of Π to instances $R(x)$ of Γ. If Π is 3-SAT, this will often involve designing "variable gadgets" that force solutions to $R(x)$ to encode true/false assignments to variables of x and "clause gadgets" that force these assignments to satisfy each of the clauses of x.

3. Show that R runs in polynomial time.

4. Show that if x has non-$\perp$ solutions, then so does $R(x)$. That is, we can transform valid solutions to x to valid solutions to $R(x)$.

5. Conversely, show that if $R(x)$ has non-$\perp$ solutions, then so does x. Moreover, we can transform valid solutions to $R(x)$ into valid solutions to x in *polynomial time*. This transformation needs to be efficient (in contrast to Item 4 because it has to be carried out by our reduction).

Reductions with the structure outlined in Items 2–5 are called *mapping reductions*, and they are what are typically used throughout the theory of NP-completeness. A formal definition follows (but we won't expect you to use this formalism, you can stick with the general definition of polynomial-time reductions):

Definition 2.7. Let $\Pi = (\mathcal{I}, f)$ and $\Gamma = (\mathcal{J}, g)$ be $\perp$-proper search problems. A *polynomial-time mapping reduction* from Π to Γ consists of two polynomial-time algorithms R and S such that for every $x \in \mathcal{I}$:

1. $R(x) \in \mathcal{J}$.

2. If $f(x) \neq \{\perp\}$, then $R(f(x)) \neq \{\perp\}$.

3. If $y \in g(R(f(x))$ and $y \neq \perp$, then $S(x, y) \in f(x)$ and $S(x, y) \neq \perp$.