1 A New Problem: SAT

- Consider the set of all propositional Boolean formulas $\phi$ over the connectives $\land$, $\lor$, and $\neg$.
- **Example:**
  \[ \phi = (x \land y) \lor (\neg x \land z) \]
  - Each propositional variable may be assigned a value of true or false.
  - Depending on values assigned to vars, a formula may be true or false.
  - If assignment $A$ of values to variables makes formula $\phi$ true, we say that $A$ satisfies $\phi$.
- **Example:** if $x = \text{false}$, $y = \text{false}$, and $z = \text{true}$, then $\phi$ is true.
- Not every formula has a satisfying assignment!
- **Example:**
  \[ \psi = ((x \land y) \lor (\neg x \land z)) \land \neg(y \lor z) \]
  is unsatisfiable.
- **Problem** (SAT): given a Boolean formula $\phi$ on variable set $X = \{x_1 \ldots x_n\}$, does there exist an assignment to $X$ that satisfies $\phi$?

We can consider restricted versions of the SAT problem in which the formula has a special form.

- A formula is in *conjunctive normal form* (CNF) if it is written as an “and” of a bunch of *clauses*, which are “or”’s of *literals* (a variable or its complement).
- **Example** (four clauses, three literals per clause):
  \[ \phi = (x \lor y \lor \neg z) \land (\neg x \lor y \lor \neg z) \land (x \lor \neg y \lor w) \land (x \lor y \lor \neg w) \]
- We can further restrict CNF formulas by specifying how many literals are in each clause.
- In particular, a 3CNF formula has exactly three literals per clause.
- 3SAT is the restriction of the SAT problem in which the input is a 3CNF formula.
2 3SAT Is No Harder Than INDEPENDENT-SET

SAT and 3SAT are classic “hard problems,” as we’ll discuss. I suggested earlier that INDEPENDENT-SET is also a hard problem. Are they related?

* Theorem: 3SAT $\leq_p$ INDEPENDENT-SET

* Proof: Let $\phi$ be a 3CNF formula. Given $\phi$, we construct a graph $G$ as follows.

  - For each literal $\ell$ in $\phi$, create a vertex $v_{\ell}$.
  - Hence, if $\phi$ has $k$ clauses, $G$ has $3k$ vertices.
  - Now, for any two literals $\ell_i$ and $\ell_j$, we connect their vertices by an edge in $G$ iff
    1. $\ell_i$ and $\ell_j$ are in the same clause, or
    2. $\ell_i$ is the logical inverse of $\ell_j$ (i.e. $\ell_i = \neg \ell_j$).

* Example:

  - Clearly, if $\phi$ has a total of $m$ literals, we can build $G$ in time $O(m^2)$, so this construction requires time polynomial in $|\phi|$.
  - Claim: Suppose $\phi$ has $k$ clauses. $\phi$ is satisfiable iff $G$ contains an independent set of size $\geq k$.
  - In what follows, we leverage the fact that a truth assignment to $\phi$ is satisfying iff it makes at least one literal in every clause of $\phi$ (hence, at least $k$ total literals) true.
  - ($\Rightarrow$) let $A$ be a satisfying assignment for $\phi$.
  - In each clause $C_s$ of $\phi$, $1 \leq s \leq k$, $A$ makes at least one literal $\ell_i^s$ true.
  - Now consider two true literals $\ell_i^s$ and $\ell_j^t$ from distinct clauses.
  - No truth assignment makes both $\ell$ and $\neg \ell$ true, so $\ell_i^s$ and $\ell_j^t$ are not logical inverses.
  - Hence, their vertices do not have an edge between them.
  - Conclude that picking one true literal under $A$ from each of $\phi$’s $k$ clauses forms an independent set of size $k$ in $G$.
  - ($\Leftarrow$) Conversely, suppose $G$ has an independent set of size $\geq k$. Then it has such a set $X$ of size exactly $k$.
  - No two vertices in $X$ are generated from the same clause of $\phi$, since all such pairs have an edge between them.
  - Hence, $X$ contains one vertex $v_s$ generated from each clause $C_s$ of $\phi$. 
Moreover, no two vertices in $X$ correspond to logically inverse literals in $\phi$, since again all such pairs would be connected by an edge.

Conclude that some truth assignment $A$ can simultaneously make all the literals corresponding to vertices in $X$ true.

But $A$ makes at least one literal true in each clause, hence satisfies $\phi$. QED

This kind of construction is called a gadget reduction – we build a “gadget” graph to model the logical relationships among variables in the formula, so that we can solve satisfiability by solving a graph problem.

3 A General Theory of Hardness

I’ve suggested that several problems are known to be “hard” in practice. Can we formalize this vague statement?

- Let $S$ be a computational decision problem.
- We say that $S$ is in the class $P$ if there exists an algorithm to decide any instance $s$ of $S$ in time $O(\text{poly}(|s|))$.
- Hence, $P$ is the set of (decision) problems solvable in polynomial time.
- Now, let’s consider a weaker property than fast decidability.
- Suppose I gave you a formula $\phi$ and claimed it was satisfiable.
- How could I prove that claim to you?
- Idea: show a satisfying assignment $A$.
- You can plug $A$ into $\phi$ and evaluate it in time... well, definitely $O(|\phi|^2)$, to check that $A$ does indeed satisfy $\phi$.
- Similarly, I could prove that a graph $G$ has an independent set of size $k$ by showing you the set.
- You can check that the set does indeed lack an edge between any two of its vertices in time $O(|G|^2)$.
- Even for problems that I don’t know how to solve quickly, I can often verify a solution for them quickly!

We’re going to formalize the idea of “quick verification.”

- Defn: Let $S$ be a decision problem. A certificate for an instance $s$ of $S$ is a piece of information sufficient to prove that $s$ is a “true” instance.
- (We just saw two examples of certificates.)
- Defn: a verifier for $S$ is an algorithm $V$ that takes in an instance $s$ of $S$ and a certificate $c$ for $s$ and checks that $s$ is a “true” instance.
• (Again, we just saw two examples of verifier algorithms.)

• **Defn:** a problem $S$ is in the class $NP$ if there exists a scheme for creating certificates for true instances of $S$, and a verifier $V$ that takes an instance $s$ and certificate $c$ generated according to this scheme, such that
  
  – For any true $s \in S$, there exists a certificate of size $O(\text{poly}(|s|))$.
  
  – The verifier $V(s,c)$ runs in time $O(\text{poly}(|s|)\text{poly}(|c|))$.

• Informally, a problem is in $NP$ if there exists a proof for each true instance, such that we can check that proof in time polynomial in the instance’s size.

How are the classes $P$ and $NP$ related?

• First, I claim that every problem in $P$ is also in $NP$; that is, $P \subseteq NP$.

• Indeed, suppose that a problem $S$ has some algorithm $A$ that decides it in polynomial time.

• If you give me an instance $s$ that you claim to be true, I can simply apply $A$ to check that it is true.

• In other words, the required certificate is *the empty string*, and the verifier is “discard the certificate and run $A$ on $s$."

• Hence, $S$ is in $NP$.

• **Important Question:** does the opposite inclusion hold? That is, is every problem in $NP$ also in $P$?

• Intuitively, you would probably say “No way!” It’s much easier to check a proof (e.g. grade your homework) than to come up with one in the first place (e.g. write your homework).

• Formally, however, *we don’t know whether or not $NP \subseteq P$* (and hence whether $P = NP$).

• In 50+ years of trying, nobody has been able to prove this inclusion one way or the other.

• This is this famous $P$ vs $NP$ question!

4 **The Hardest Problems in NP**

We can leverage the practical difficulty of settling the equivalence of $P$ and $NP$ to show that other problems are practically hard.

• Let $S$ be a decision problem.

• We say that $S$ is $NP$-hard if, for every problem $T$ in $NP$, $T \leq_p S$.

• If we could solve $S$ efficiently, then we could solve every problem in $NP$ efficiently.
• Hence, $S$ is “at least as hard” as every problem in $NP$.
• If $S$ itself is also in $NP$, we say that $S$ is $NP$-complete.
• NP-complete problems are thus “the hardest problems in $NP$.”
• **Lemma**: if any NP-complete problem is in $P$, then $P = NP$.
• **Pf**: Let $S$ be an NP-complete problem, and suppose $S$ is in $P$.
• Let $A$ be an algorithm to decide $S$ in polynomial time.
• For any other problem $T$ in $NP$, $T \leq_p S$.
• Hence, to decide an instance $t$ of $T$, we can use the implied reduction to convert $t$ to an instance $s$ of $S$ in time $O(poly(|t|))$, then apply $A$ to decide $s$.
• Conclude that $T$ is also in $P$. QED

Consequences?

• Proving whether or not $P = NP$ is (practically speaking) really, really hard.
• If we know that a problem $S$ is NP-complete, then finding a polynomial-time algorithm for $S$ is equivalent to proving that $P = NP$, so it must also be really, really hard.
• That doesn’t mean that such an algorithm doesn’t exist... but we would be shocked if we found one, and it’s not likely that you or I would be able to do so.
• So, showing that a problem is NP-complete is basically a way to say “don’t expect me to find a polytime algorithm for this problem!”
• Moreover, NP-completeness is transferable from one problem to another.
• **Lemma**: Let $S$ be an NP-complete problem, and suppose that $T \in NP$. If $S \leq_p T$, then $T$ is also NP-complete.
• **Pf**: Because $S$ is NP-complete, for any problem $U \in NP$, we can reduce $U$ to $S$ in polynomial time.
• But if $S \leq_p T$, then we can reduce $S$ to $T$ in polynomial time.
• Hence, we can reduce $U$ to $T$ in two steps, each of which requires only polynomial time, and so $U \leq_p T$.
• Conclude that $T$ is NP-complete. QED

Wait... is there such a thing as an NP-complete problem?

• **Theorem** (Cook-Levin): SAT is an NP-complete problem.
• We’re not going to prove this, because the proof is too hairy.
• Roughly, Cook and Levin showed that Boolean formulas are powerful enough to compactly describe the behavior of an arbitrary algorithm.
• Hence, given a problem $S$ in $NP$, which therefore has a small certificate scheme and a fast verifier $V(s, c)$, we can write a small formula $\phi_V$ to describe the behavior of $V$ given $s$ and $c$.

• For a fixed $s$, formula $\phi_V$ is satisfiable iff there exists certificate $c$ that proves that $s$ is true; i.e. iff $s$ is a true instance of $S$.

• For a careful proof of this theorem, take CSE 547.

• Pretty much all known NP-complete problems (and there are lots!) were proved by reductions starting from SAT.

• In particular, one can prove that SAT $\leq_p$ 3SAT, and so this restricted form of satisfiability is also NP-complete.

• From there, we just showed that INDEPENDENT-SET is NP-complete by reduction from 3SAT, and that VERTEX-COVER is NP-complete by reduction from INDEPENDENT-SET.

• You’ll get to practice finding reductions of your own on your homework.