SAT is NP-complete

Suppose \((D, Q) \in \text{NP}\). We shall construct a reduction \(Q \leq^p \text{SAT}\).

Outline of proof:

Given an instance \(d \in D\), design a boolean formula \(\phi_d\) which can be satisfied if its variables describe the successful executions of an NRM checking \(Q\). The machine can be polynomially bounded, so the size of \(\phi_d\) is polynomial in the size of \(d\).

The rest is detail.
We have an instance \( d \in D \).

Since \((D, Q) \in \text{NP}\), there is a polynomially bounded NRM \( M = (R_0, \ldots, R_{m-1}, I_0 \ldots I_{n-1})\) that computes \( Q \). Let \( p(x) \) be the polynomial bound, and let \( s = p(|d|) \). (\( s \) ‘number of steps’) (Note: the empty instruction \( I_n \) is the halting state.)

So \( M \) takes at most \( s \) steps.

How big can the values in the registers get? Worst case: start with \( 2^{|d|} \), and execute \( k \text{ ADD}(0, 0) \) instructions, giving \( 2^{|d|+s} \). I.e. at most \( |d| + s \) bits. Assume wlog \( s \geq |d| \), so at most \( 2s \) bits.

We now define a large, but \( O(\text{poly}(s)) \), number of variables, with intended interpretations.

‘Intended’ because we will design \( \phi_d \) so that these interpretations do what we want.
Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>How many?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{tj}$</td>
<td>program counter at step $t$ is on $l_j$</td>
<td>$s \cdot n$</td>
</tr>
<tr>
<td>$R_{tik}$</td>
<td>$k$th bit of $R_i$ at step $t$</td>
<td>$s \cdot m \cdot 2s$</td>
</tr>
</tbody>
</table>

Formulas

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>How big?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\chi_{\text{one}}$</td>
<td>prog cntr in one place</td>
<td>$s \cdot n \cdot n$</td>
</tr>
<tr>
<td>$\chi_t$</td>
<td>step $t + 1$ follows from step $t$</td>
<td>??</td>
</tr>
<tr>
<td>$\rho_{\text{init}}$</td>
<td>initial register values</td>
<td>??</td>
</tr>
</tbody>
</table>

The formula $\rho_{\text{init}} \land \chi_{\text{one}} \land \bigwedge_t \chi_t$ is then satisfied by assignments corresponding to valid executions of the machine.

The $\chi_{\text{one}}$ formula is easy:

$$\bigwedge_t \bigvee \left( C_{tj} \land \bigwedge_j \neg C_{tj'} \right)$$
Coding $\chi_t$ is a bit tedious . . . step $t + 1$ follows from step $t$ if the control flow is correct and if the registers have changed correctly.

For control flow: the formula is $\bigvee_j (C_{tj} \land \nu_{tj})$, where $\nu_{tj}$ is:

\[
\begin{align*}
C_{t+1,j+1} & \quad \text{if } l_j \text{ is INC, ADD or SUB} \\
C_{t+1,j+1} \lor C_{t+1,j'} & \quad \text{if } l_j \text{ is MAYBE}(j') \\
((\bigvee_k R_{tik}) \land C_{t+1,j+1}) \lor ((\bigwedge_k \neg R_{tik}) \land C_{t+1,j'}) & \quad \text{if } l_j \text{ is DEC}(i,j')
\end{align*}
\]

To code the register changes, we need to express the addition of $(2s)$-bit numbers by boolean operations on the bits. See your hardware course!

**Exercise:** write the formula $\rho_{tii'}^+$, which says that at step $t + 1$, the value of $R_i$ is the sum of the values of $R_i$ and $R_i'$ at step $t$.

Despite the very tedious nature of these formulas, it’s clear that the whole thing is $O(s^3)$ in length.

And so the theorem is proved.
More NP-complete problems

There are many hundreds of natural problems that are NP-complete. Look at the Wikipedia article ‘List of NP-complete problems’ for a range of examples, or Sipser for more details on a smaller range.

Starting from SAT, we use reductions to build a library of NP-complete problems, for use in tackling new potential problems.

Sometimes, the reductions from SAT (or other known problem) require considerable ingenuity. E.g., showing NP-completeness of HPP is quite tricky.

We’ll do a couple of examples.
3SAT

There are special forms of SAT that turn out to be particularly handy. A literal is either a propositional variable $P$ or the negation $\neg P$ of one.

$\phi$ is in conjunctive normal form (CNF) if it is of the form $\land_i \lor_j p_{ij}$ where each $p_{ij}$ is a literal.

$\phi$ is in $k$-CNF if each clause $\lor_j p_{ij}$ has at most $k$ literals.

3SAT is the problem of whether a satisfying assignment exists for a formula in 3-CNF.

The reduction from unrestricted SAT to 3-SAT is also a bit tricky, mostly because normal boolean logic conversion to CNF introduces exponential blowup. The Tseitin encoding is used to produce a not equivalent but equisatisfiable CNF formula.

We are paying a price here for having done Cook–Levin with RMs. The original version with TMs manages to produce a CNF formula describing machine executions. In fact, SAT is often defined to mean CNF-SAT.
CLIQUE

Given a graph $G = (V, E)$ and a number $k$, a $k$-clique is a $k$-sized subset $C$ of $V$, such that every vertex in $C$ has an edge to every other. ($C$ forms a complete subgraph.) The CLIQUE problem is to decide whether $G$ has a $k$-clique.

The problem has applications in chemistry and biology (and of course sociology).

It is NP-complete, by the following reduction from 3SAT:

Let $\phi = \bigwedge_{1 \leq i \leq k} (x_{i1} \lor x_{i2} \lor x_{i3})$ be an instance of 3SAT, so each $x_{ij}$ is a literal. Construct a graph thus: each $x_{ij}$ is a vertex. Make an edge between $x_{ij}$ and $x_{i'j'}$ just in case $i \neq i'$ and $x_{i'j'}$ is not the negation of $x_{ij}$. (I.e., we connect literals in different clauses just when they are not inconsistent with each other.)

Since the vertices in one clause are disconnected, finding a $k$-clique amounts to finding one literal for each clause, such that they are all consistent – and so represent a satisfying assignment. Conversely, any satisfying assignment generates a $k$-clique.
We have said that NP-COMPLETE problems are effectively exponential. But do we know this? Is it possible that we’re just stupid, and there’s a PTIME algorithm for SAT?

We don’t know.

It is possible that NP = P. If you find a polynomial algorithm for SAT or any other NP-complete problem . . . hire bodyguards, because most web/banking security depends on such problems being hard.

Solving the problem is worth a million US dollars: it is one of the seven problems chosen by the Clay Institute for their Millennium Prizes.

Many of the results in complexity theory are therefore conditional: ‘if P ≠ NP, then very difficult theorem’.

E.g. if P ≠ NP, then there are problems that are neither P nor NP-complete. There are very few candidates: best known is the Graph Isomorphism Problem.
Handling NP

As far as we know, NP problems are just hard: need exponential search, so $O(\text{poly}(n) \cdot 2^n)$. So how do we solve them in practice? Huge industry ...

Randomized algorithms are often useful. Allow algorithms to toss a coin. Surprisingly one can get randomized algorithms that solve e.g. 3SAT in time $O(\text{poly}(n) \cdot 1.33^n)$ (Why is this useful? $2^{100} \approx 10^{31}$, while $1.33^{100} \approx 10^{12}$) Catch: (really) small probability of error!

In many special classes (e.g. sparse graphs, or almost-complete graphs), heuristics lead to fast results.

See http://satcompetition.org/ for the state of the art.
Beyond NP

So what do we know about hard problems?
As noted earlier, $\text{ExpTime}$ is strictly harder than P or NP: a polynomially bounded machine simply doesn’t have time or memory to explore $2^n$ steps. Formal proof via the Time Hierarchy Theorem.

Similarly $\text{2-ExpTime} \supset \text{ExpTime}$.

Also applies to N versions.
Space-bounded machines

An RM/TM is $f(n)$-space-bounded if it may use only $f$ (input size) space. For TMs, space means cells on tape; for RMs, number of bits in registers. \textbf{PSPACE} is the class of problems solvable by polynomially-space-bounded machines.

The following are obvious (\textbf{Exercise: why?}):

- $\text{PSPACE} \supseteq \text{PTime}$
- $\text{PSPACE} \supseteq \text{NPTime}$
- $\text{PSPACE} \subseteq \text{ExpTime}$
The polynomial hierarchy

Recall the notion of *oracle*. Suppose we have an oracle for NP problems. What then can we compute?

The polynomial hierarchy is defined thus:

- Let $\Delta^P_0 = \Sigma^P_0 = \Pi^P_0 = \text{P}$.
- Let $\Delta^P_{n+1} = \text{P}^{\Sigma^P_n}$, $\Sigma^P_{n+1} = \text{NP}^{\Sigma^P_n}$, $\Pi^P_{n+1} = \text{co-NP}^{\Sigma^P_n}$.

where co-$X$ is problems whose negation is in $X$.

So $\Sigma^P_1 = \text{NP}^\text{P} = \text{NP}$, and $\Sigma^P_2 = \text{NP}^{\text{NP}}$. *Why is NP$^{\text{NP}}$ not just NP?*

This hierarchy could *collapse* at any level – generalization of $\text{P} \equiv \text{NP}$. We don’t know.

The whole hierarchy is inside PSPACE.