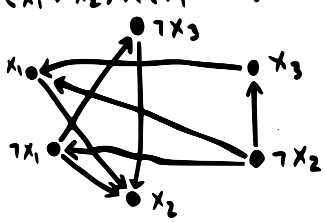
2-SAT is easy

From a 2-SAT expression
$$c_1 \wedge c_2 \wedge \dots \wedge c_m,$$
we form a directed graph G.

The vertices of G are the variables and their negations. Their is an edge $u \rightarrow w$ if and only if
$$(\neg u \vee w) \equiv (u \Rightarrow w)$$
occurs in the expression.

Example: The graph for
$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$$
is



Proposition: The expression is satisfiable if and only if there is no variable x such that G contains both:

- a directed path from x to $\neg x$,
- a directed path from $\neg x$ to x.

Proof: First suppose that such paths exist for some variable x, but that there is some assignment of {true, false} to the variables that nonetheless satisfies the expression.

Color the vertices of G by this assignment.

If x is true, then $\neg x$ is false. Therefore in the directed path from x to $\neg x$, there must be an edge $u \rightarrow w$ where u is true and w is false. However, since this is an edge, our expression includes the clause $(\neg u \vee w)$, a contradiction.

In the case where x is false, repeat this argument looking at the path from $\neg x$ to x.

Now suppose that G does not contain such paths for any variable. We want to construct a satisfying truth assignment.

To do this, we repeat the following process.

Pick a vertex u (i.e., a variable or its negation) which has yet to be assigned a truth value, and such that there is no path from u to $\neg u$ in G.

(We can do this because if u hasn't been assigned a truth value, then neither has $\neg u$, so at least one of these vertices does not have a path to the other.)

Now set u and all vertices reachable from u true.

Also set all negations of these vertices false.

Why can we do this?

If there is an edge $u \to w$ in G, then there's also an edge $\neg w \to \neg u$. So if there was a directed path from u to w, there would be a directed path from $\neg w$ to $\neg u$.

So, if there are directed paths from u to both w and $\neg w$, then there would be a directed path from u to $\neg u$, a contradiction.

Furthermore, if there were a path from u to w where w is set false (from a previous step), then there is a path from $\neg w$ to $\neg u$, so u would have been set false before.

Now simply repeat this until done. ∎

<u>Corollary</u>: 2SAT can be solved in polynomial time.

<u>Proof</u>: It takes polynomial time to check for a path from x to $\neg x$. Repeat this $2n$ times, for each vertex. ∎

Deterministic vs. Non-deterministic

What we have defined are <u>deterministic</u> Turing machines. By the Church-Turing Thesis, they are a good model of computation.

We now define an unrealistic model of computation. Our reasons will be explained later.

A non-deterministic Turing Machine is one in which there may be more than one appropriate "next step" in a computation.

Formally, δ maps to sets of actions.

Non-deterministic Turing Machines

An input is accepted if <u>some</u> branch of this computation yields "yes". (Even if other branches yield "no"!)

Suppose we have a family of problems P.

We say the non-deterministic Turing machine N can decide P <u>in time $f(n)$</u> if given any input $x \in P$ (the encoded problem), N has no computation paths longer than $f(n)$.

<u>Note</u>: The total amount of computation may be exponentially larger than $f(n)$!

<u>Fact</u>: SAT can be solved by a non-deterministic Turing machine in polynomial time.

<u>Proof</u>: Branch on whether $x_1 = 0$ or 1. Then branch on $x_2, \dots, x_n$. Once we've "guessed" truth assignments for $x_1, \dots, x_n$, a branch returns "yes" if its assignment satisfies the expression, and "no" otherwise. ■

You might feel like this proof is "cheating". It's not though — the part that is cheating is using a non-deterministic Turing machine in the first place!

Complexity Classes

P = decidable by a deterministic Turing machine in polynomial time.
E.g.: 2SAT

NP = decidable by a non-deterministic Turing machine in polynomial time.
E.g.: SAT

Note that $P \subseteq NP$.

Question: Does $P = NP$?

Prize: \$1 million from the Clay Mathematics Institute.

Why we shouldn't expect P=NP

1. At some level, P=NP means that creating = checking. Based on human experience, that seems wrong.
2. P=NP would end cryptography.
3. P=NP would mean that SAT can be solved <u>much</u> faster than brute force. Right now, the best algorithms run at 1.3^n.

One caveat: it could be that P=NP is true, but that solving SAT takes $n^{1000000}$ or something. The affects of this would be less.

The self-referential argument for P≠NP

If P=NP, then there is a proof.

Proofs are "easy" to check, so the problem of finding that proof is in NP.

But if P=NP, then we can find it in polynomial time.

So why haven't we found it?

SAT is NP-complete

The SAT problem is not only NP, but it is <u>NP-complete</u>, meaning that <u>every</u> NP problem can be viewed as a SAT problem.

In other words, a polynomial time algorithm to SAT (on a deterministic Turing machine) would prove P=NP and earn you $1 million.

Cook's Theorem: SAT is NP-complete.

Sketch of proof: To give a formal proof, we would need to get into ugly details of non-deterministic Turing machines, so instead we'll just try to give the main idea.

Suppose your class of problems X can be solved in time n^k by a non-determistic Turing machine.

Then if you feed any problem $x \in X$ into this machine, it has a computation tree of height at most n^k.

It is possible to convert this into a binary tree, although we omit the details.

Therefore, each time the machine has a choice to make, there are two options, say 0 or 1.

So, the computation of this machine can be described by a function

$$f : \text{choices} \to \{\text{yes, no}\},$$

i.e.,

$$f : \{0,1\}^{n^k} \to \{\text{yes, no}\},$$

i.e., a Boolean n^k-ary function!

The ultimate result is then "yes" if f is satisfiable and "no" otherwise. ■

Subset-sum Problem

Given a set of integers, does some subset sum to 0?

E.g.: $S = \{-2, -3, 7, 15, -10, 14\}$

$$-2 + -3 + -10 + 15 = 0.$$

This is NP-complete.

Hamiltonian cycle problem

Does the graph G have a Hamiltonian cycle? (A cycle that visits each vertex precisely once.)

This is NP-complete.