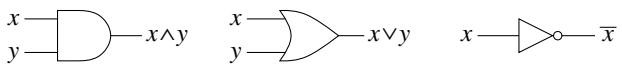
24 NP-Hard Problems (April 27)

24.1 "Efficient" Problems

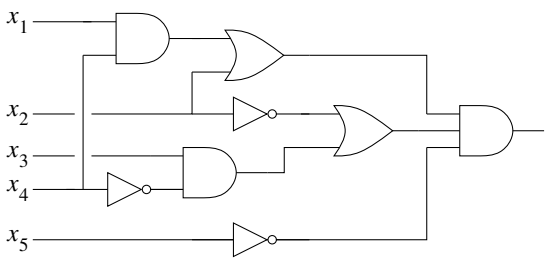
A long time ago[1], theoretical computer scientists like Steve Cook and Dick Karp decided that a minimum requirement of any efficient algorithm is that it runs in polynomial time: $O(n^c)$ for some constant c. People recognized early on that not all problems can be solved this quickly, but we had a hard time figuring out exactly which ones could and which ones couldn't. So Cook, Karp, and others, defined the class of *NP-hard* problems, which most people believe *cannot* be solved in polynomial time, even though nobody can prove a super-polynomial lower bound.

Circuit satisfiability is a good example of a problem that we don't know how to solve in polynomial time. In this problem, the input is a *boolean circuit*: a collection of and, or, and not gates connected by wires. We will assume that there are no loops in the circuit (so no delay lines or flip-flops). The input to the *circuit* is a set of m boolean (true/false) values $x_1, \ldots, x_m$. The output is a single boolean value. Given specific input values, we can calculate the output in polynomial (actually, *linear*) time using depth-first-search and evaluating the output of each gate in constant time.

The circuit satisfiability problem asks, given a circuit, whether there is an input that makes the circuit output "true", or conversely, whether the circuit *always* outputs "false". Nobody knows how to solve this problem faster than just trying all 2^m possible inputs to the circuit, but this requires exponential time. On the other hand, nobody has ever proven that this is the best we can do; maybe there's a clever algorithm that nobody has discovered yet!



An and gate, an or gate, and a not gate.



A boolean circuit. Inputs enter from the left, and the output leaves to the right.

24.2 P, NP, and co-NP

Let me define three classes of problems:

- **P** is the set of yes/no problems[2] that can be solved in polynomial time. Intuitively, P is the set of problems that can be solved quickly.

[1] ... in a galaxy far far away ...

[2] Technically, I should be talking about *languages*, which are just sets of bit strings. The language associated with a yes/no problem is the set of bit strings for which the answer is yes. For example, if the problem is "Is the input graph connected?", then the corresponding language is the set of connected graphs, where each graph is represented as a bit string (for example, its adjacency matrix). P is the set of languages that can be *recognized* in polynomial time by a single-tape Turing machine. Take 375 if you want to know more.

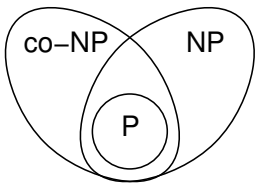
- **NP** is the set of yes/no problems with the following property: If the answer is yes, then there is a *proof* of this fact that can be checked in polynomial time. Intuitively, NP is the set of problems where we can verify a "yes" answer quickly if we have the solution in front of us. For example, the circuit satisfiability problem is in NP. If the answer is yes, then any set of m input values that produces "true" output is a proof of this fact; we can check the proof by evaluating the circuit in polynomial time.

- **co-NP** is the exact opposite of NP. If the answer to a problem in co-NP is *no*, then there is a proof of this fact that can be checked in polynomial time.

If a problem is in P, then it is also in NP — to verify that the answer is yes in polynomial time, we can just throw away the proof and recompute the answer from scratch. Similarly, any problem in P is also in co-NP.

The (or at least, *a*) central question in theoretical computer science is whether or not P=NP. Nobody knows. Intuitively, it should be obvious that P≠NP; the homeworks and exams in this class have (I hope) convinced you that problems can be incredibly hard to solve, even when the solutions are obvious once you see them. But nobody can prove it.

Notice that the definition of NP (and co-NP) is not symmetric. Just because we can verify every yes answer quickly, we may not be able to check no answers quickly, and vice versa. For example, as far as we know, there is no short proof that a boolean circuit is *not* satisfiable. But again, we don't have a proof; everyone believes that NP≠co-NP, but nobody really knows.



What we *think* the world looks like.

24.3 NP-hard, NP-easy, and NP-complete

A problem Π is **NP-hard** if a polynomial-time algorithm for Π would imply a polynomial-time algorithm for *every problem in NP*. In other words:

$$\boxed{\Pi \text{ is NP-hard} \iff \text{If } \Pi \text{ can be solved in polynomial time, then P=NP}}$$

Intuitively, this is like saying that if we could solve one particular NP-hard problem quickly, then we could quickly solve *any* problem whose solution is easy to understand, using the solution to that one special problem as a subroutine. NP-hard problems are at least as hard as any problem in NP.

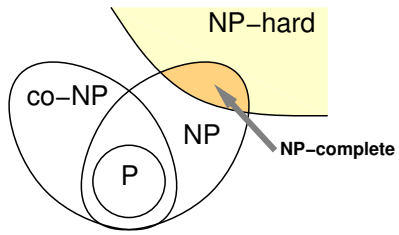
Saying that a problem is NP-hard is like saying "If pigs can whistle, then horses can fly." A whistling pig would be somewhat surprising, but not entirely unbelievable. A flying horse, on the other hand, is something that everyone (sane) "knows" can't exist. We don't have a mathematical *proof* that horses can't fly—the fact that no one has ever seen a flying horse is evidence, but not a proof—but no sane person would believe me if I said I owned a flying horse. So the statement "If pigs can whistle, then horses can fly" has a natural corollary: No one in their right mind should

believe that pigs can whistle! Likewise, if a problem is NP-hard, no one in their right mind should believe it can be solved in polynomial time.

The following theorem was proved by Steve Cook in 1971. I won't even sketch the proof (since I've been deliberately vague about the definitions). If you want more details, take CS 375 next semester.

Cook's Theorem: *circuit satisfiability is NP-hard.*

Finally, a problem is **NP-complete** if it is both NP-hard and an element of NP ("NP-easy"). NP-complete problems are the hardest problems in NP. If anyone finds a polynomial-time algorithm for even one NP-complete problem, then that would imply a polynomial-time algorithm for *every* NP-complete problem. Literally *thousands* of problems have been shown to be NP-complete, so a polynomial-time algorithm for one (*i.e.*, all) of them seems incredibly unlikely.



More of what we *think* the world looks like.

24.4 Reductions (again) and SAT

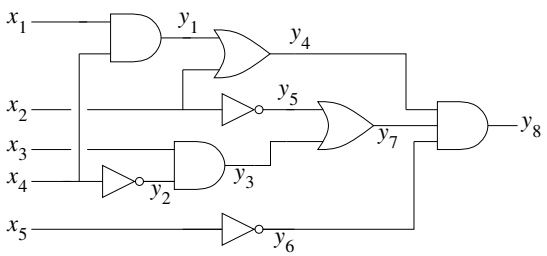
To prove that a problem is NP-hard, we use a reduction argument, just like we're trying to prove a lower bound. For example, consider the *formula satisfiability* problem, usually just called *SAT*. The input to SAT is a boolean *formula* like

$$(a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee \overline{(\bar{a} \Rightarrow d)} \vee (c \neq a \wedge b)),$$

and the question is whether it is possible to assign boolean values to the variables $a, b, c, \ldots$ so that the formula evaluates to "true".

To show that SAT is NP-hard, we need to give a reduction from a known NP-hard problem. The only problem we know is NP-hard so far is circuit satisfiability, so let's start there. Given a boolean circuit, we can transform it into a boolean formula by creating new output variables for each gate, and then just writing down the list of gates. For example, we could transform the circuit below into the following formula:

$$(y_1 = x_1 \wedge x_2) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge \cdots \wedge (y_8 = y_4 \wedge y_7 \wedge y_6)$$



A boolean circuit with gate variables.

Now the circuit is satisfiable if and only if the formula is satisfiable. Given a satisfying input to the circuit, we can get a satisfying assignment for the formula by computing the output of every gate. Given a satisfying assignment for the formula, we can get a satisfying input the the circuit by just ignoring the gate variables y_i.

It is possible to transform any boolean circuit into a formula in linear time using depth-first search, and the size of the resulting formula is only a constant factor larger than the size of the circuit. Thus, we have a polynomial-time reduction from circuit satisfiability to SAT:

$$T_{CSAT}(n) \leq O(n) + T_{SAT}(O(n))$$

The reduction implies that if we had a polynomial-time algorithm for SAT, then we'd have a polynomial-time algorithm for circuit satisfiability, which would imply that P=NP. So SAT is NP-hard.

To prove that a boolean formula is satisfiable, we only have to specify an assignment to the variables that makes the formula true. We can check the proof in linear time just by reading the formula from left to right, evaluating as we go. So SAT is also in NP, and thus is actually NP-complete.

24.5 3SAT

A special case of SAT that is incredibly useful in proving NP-hardness results is *3SAT* (or as [CLR] insists on calling it, *3-CNF-SAT*).

A boolean formula is in *conjunctive normal form* (CNF) if it is a conjunction (and) of several *clauses*, each of which is the disjunction (or) or several *literals*, each of which is either a variable or its negation. For example:

$$(a \vee b \vee c \vee d) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b})$$

A *3CNF* formula is a CNF formula with exactly three literals per clause; the previous example is not a 3CNF formula, since its first clause has four literals and its last clause has only two. 3SAT is just SAT restricted to 3CNF formulas — given a 3CNF formula, is there an assignment to the variables that makes the formula evaluate to true?

We could prove that 3SAT is NP-hard by a reduction from the more general SAT problem, but it's easier just to start over from scratch, with a boolean circuit. We perform the reduction in several stages.

1. *Make sure every and and or gate has only two inputs.* If any gate has $k > 2$ inputs, replace it with a binary tree of $k - 1$ two-input gates.

2. *Write down the circuit as a formula, with one clause per gate.* This is just the previous reduction.

3. *Change every gate clause into a CNF formula.* There are only three types of clauses, one for each type of gate:

$$\begin{aligned} a = b \wedge c &\longmapsto (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c) \\ a = b \vee c &\longmapsto (\bar{a} \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (a \vee \bar{c}) \\ a = \bar{b} &\longmapsto (a \vee b) \wedge (\bar{a} \vee \bar{b}) \end{aligned}$$

4. *Make sure every clause has exactly three literals.* Introduce a new variable for each two-literal clause, and expand it into two clauses as follows:

$$(a \vee b) \longmapsto (a \vee b \vee x) \wedge (a \vee b \vee \bar{x})$$

At the end of this process, we've transformed the circuit into an equivalent 3CNF formula. The formula is satisfiable if and only if the original circuit is satisfiable. As with the more general SAT problem, the formula is only a constant factor larger than then any reasonable description of the original circuit, and the reduction can be carried out in polynomial time. Thus, we have a polynomial-time reduction from circuit satisfiability to 3SAT:

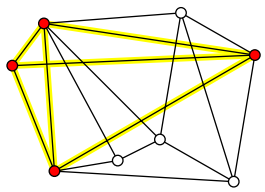
$$T_{CSAT}(n) \leq O(n) + T_{3SAT}(O(n))$$

So 3SAT is NP-hard.

Moreover, since 3SAT is a special case of SAT, it is also in NP, so 3SAT is NP-complete.

24.6 Maximum Clique Size

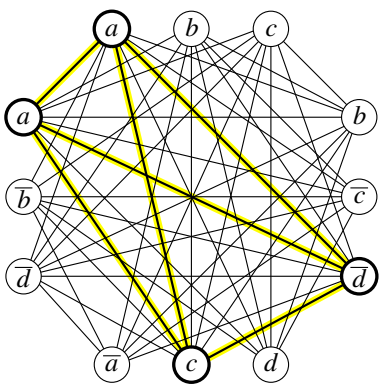
The last problem I'll consider in this lecture is a graph problem. A *clique* is another name for a complete graph. The *maximum clique size* problem, or simply MAXCLIQUE, is to compute, given a graph, the number of nodes in its largest complete subgraph.



A graph with maximum clique size 4.

I'll prove that MAXCLIQUE is NP-hard (but not NP-complete, since it isn't a yes/no problem) using a reduction from 3SAT. I'll describe a reduction algorithm that transforms a 3CNF formula into a graph that has a clique of a certain size if and only if the formula is satisfiable.

The graph has one node for each instance of each literal in the formula. Two nodes are connected by an edge if (1) they correspond to literals in different clauses and (2) those literals do not contradict each other. In particular, all the nodes that come from the same literal (in different clauses) are joined by edges. For example, the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ is transformed into the following graph. (Look for the edges that *aren't* in the graph.)



A graph derived from a 3CNF formula, and a clique of size 4.

Now suppose the original formula had k clauses. Then I claim that the formula is satisfiable if and only if the graph has a clique of size k.

1. **k-clique $\Longrightarrow$ satisfying assignment:** If the graph has a clique of k vertices, then each vertex must come from a different clause. To get the satisfying assignment, we declare that each literal in the clique is true. Since we only connect non-contradictory literals with edges, this declaration assigns a consistent value to several of the variables. There may be variables that have no literal in the clique; we can set these to any value we like.

2. **satisfying assignment $\Longrightarrow$ k-clique:** If we have a satisfying assignment, then we can choose one literal in each clause that is true. Those literals form a clique in the graph.

Thus, the reduction is correct. Since the reduction from 3CNF formula to graph can be done in polynomial time, so MaxClique is NP-hard.