Math class is tough!
— Teen Talk Barbie (1992)

That's why I like it!
— What she should have said next

The Manhattan-based Barbie Liberation Organization claims to have performed corrective surgery on 300 Teen Talk Barbies and Talking Duke G.I. Joes—switching their sound chips, repackaging the toys, and returning them to store shelves. Consumers reported their amazement at hearing Barbie bellow, 'Eat lead, Cobra!' or 'Vengeance is mine!,' while Joe chirped, 'Will we ever have enough clothes?' and 'Let's plan our dream wedding!'
— Mark Dery, "Hacking Barbie's Voice Box: Vengeance is Mine!", *New Media* (May 1994)

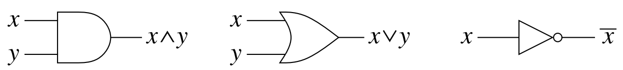
16 NP-Hard Problems (December 3 and 5)

16.1 'Efficient' Problems

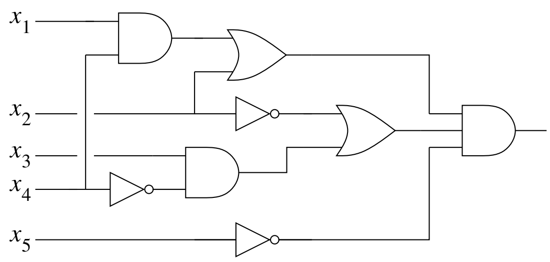
A long time ago[1], theoretical computer scientists like Steve Cook and Dick Karp decided that a minimum requirement of any efficient algorithm is that it runs in polynomial time: $O(n^c)$ for some constant c. People recognized early on that not all problems can be solved this quickly, but we had a hard 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 proved 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.

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

16.2 P, NP, and co-NP

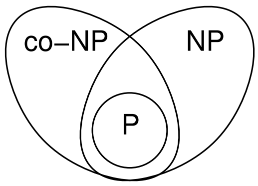
Let me define three classes of problems:

- **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.
- **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.

16.3 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:

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

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.

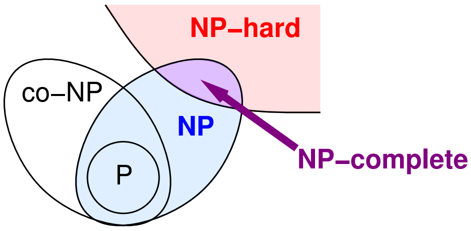
[2]Technically, I should be talking about *languages*, which are just sets of bit strings. The language associated with a yes/no problem is the set of bit strings for which the answer is yes. For example, if the problem is 'Is the input graph connected?', then the corresponding language is the set of 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.

Saying that a problem is NP-hard is like saying 'If I own a dog, then it can speak fluent English.' You probably don't know whether or not I own a dog, but you're probably pretty sure that I don't own a *talking* dog. Nobody has a mathematical *proof* that dogs can't speak English—the fact that no one has ever heard a dog speak English is evidence, as are the hundreds of examinations of dogs that lacked the proper mouth shape and brainpower, but mere evidence is not a proof. Nevertheless, no sane person would believe me if I said I owned a dog that spoke fluent English. So the statement 'If I own a dog, then it can speak fluent English' has a natural corollary: No one in their right mind should believe that I own a dog! 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.

16.4 Reductions (again) and SAT

To prove that a problem is NP-hard, we use a reduction argument, *exactly* like we're trying to prove a lower bound. So we can use a special case of that statement about reductions the you tattooed on the back of your hand last time.

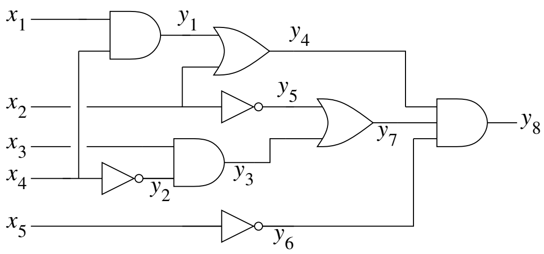
To prove that problem A is NP-hard, reduce a known NP-hard problem to A.

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 separated by and. For example, we could transform the example circuit into a formula as follows:



$$(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge$$
$$(y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \wedge (y_8 = y_4 \wedge y_7 \wedge y_6) \wedge y_8$$

A boolean circuit with gate variables added, and an equivalent boolean formula.

Now the original circuit is satisfiable if and only if the resulting 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.

We can 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:

$$\begin{tikzcd}
\boxed{\text{boolean circuit}} \arrow[r, "O(n)"] & \boxed{\text{boolean formula}} \arrow[d, Rightarrow, "\text{SAT}"] \\
\boxed{\text{TRUE or FALSE}} & \boxed{\text{TRUE or FALSE}} \arrow[l, "\text{trivial}"']
\end{tikzcd}$$

$$T_{\text{CSAT}}(n) \leq O(n) + T_{\text{SAT}}(O(n)) \quad \Longrightarrow \quad T_{\text{SAT}}(n) \geq T_{\text{CSAT}}(\Omega(n)) - O(n)$$

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.

16.5 3SAT

A special case of SAT that is incredibly useful in proving NP-hardness results is *3SAT* (or as [CLRS] 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) of several *literals*, each of which is either a variable or its negation. For example:

$$\overbrace{(a \vee b \vee c \vee d)}^{\text{clause}} \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b})$$

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 new variables into each one- and two-literal clause, and expand it into two clauses as follows:

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

For example, if we start with the same example circuit we used earlier, we obtain the following 3CNF formula. Although this may look a lot more ugly and complicated than the original circuit at first glance, it's actually only a constant factor larger. Even if the formula were larger than the circuit by a *polynomial*, like n^{373}, we would have a valid reduction.

$$\begin{aligned} &(y_1 \vee \overline{x_1} \vee \overline{x_4}) \wedge (\overline{y_1} \vee x_1 \vee z_1) \wedge (\overline{y_1} \vee x_1 \vee \overline{z_1}) \wedge (\overline{y_1} \vee x_4 \vee z_2) \wedge (\overline{y_1} \vee x_4 \vee \overline{z_2}) \\ \wedge\, &(y_2 \vee x_4 \vee z_3) \wedge (y_2 \vee x_4 \vee \overline{z_3}) \wedge (\overline{y_2} \vee \overline{x_4} \vee z_4) \wedge (\overline{y_2} \vee \overline{x_4} \vee \overline{z_4}) \\ \wedge\, &(y_3 \vee \overline{x_3} \vee \overline{y_2}) \wedge (\overline{y_3} \vee x_3 \vee z_5) \wedge (\overline{y_3} \vee x_3 \vee \overline{z_5}) \wedge (\overline{y_3} \vee y_2 \vee z_6) \wedge (\overline{y_3} \vee y_2 \vee \overline{z_6}) \\ \wedge\, &(\overline{y_4} \vee y_1 \vee x_2) \wedge (y_4 \vee \overline{x_2} \vee z_7) \wedge (y_4 \vee \overline{x_2} \vee \overline{z_7}) \wedge (y_4 \vee \overline{y_1} \vee z_8) \wedge (y_4 \vee \overline{y_1} \vee \overline{z_8}) \\ \wedge\, &(y_5 \vee x_2 \vee z_9) \wedge (y_5 \vee x_2 \vee \overline{z_9}) \wedge (\overline{y_5} \vee \overline{x_2} \vee z_{10}) \wedge (\overline{y_5} \vee \overline{x_2} \vee \overline{z_{10}}) \\ \wedge\, &(y_6 \vee x_5 \vee z_{11}) \wedge (y_6 \vee x_5 \vee \overline{z_{11}}) \wedge (\overline{y_6} \vee \overline{x_5} \vee z_{12}) \wedge (\overline{y_6} \vee \overline{x_5} \vee \overline{z_{12}}) \\ \wedge\, &(\overline{y_7} \vee y_3 \vee y_5) \wedge (y_7 \vee \overline{y_3} \vee z_{13}) \wedge (y_7 \vee \overline{y_3} \vee \overline{z_{13}}) \wedge (y_7 \vee \overline{y_5} \vee z_{14}) \wedge (y_7 \vee \overline{y_5} \vee \overline{z_{14}}) \\ \wedge\, &(y_8 \vee \overline{y_4} \vee \overline{y_7}) \wedge (\overline{y_8} \vee y_4 \vee z_{15}) \wedge (\overline{y_8} \vee y_4 \vee \overline{z_{15}}) \wedge (\overline{y_8} \vee y_7 \vee z_{16}) \wedge (\overline{y_8} \vee y_7 \vee \overline{z_{16}}) \\ \wedge\, &(y_9 \vee \overline{y_8} \vee \overline{y_6}) \wedge (\overline{y_9} \vee y_8 \vee z_{17}) \wedge (\overline{y_9} \vee y_8 \vee \overline{z_{17}}) \wedge (\overline{y_9} \vee y_6 \vee z_{18}) \wedge (\overline{y_9} \vee y_6 \vee \overline{z_{18}}) \\ \wedge\, &(y_9 \vee z_{19} \vee z_{20}) \wedge (y_9 \vee \overline{z_{19}} \vee z_{20}) \wedge (y_9 \vee z_{19} \vee \overline{z_{20}}) \wedge (y_9 \vee \overline{z_{19}} \vee \overline{z_{20}}) \end{aligned}$$

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:

$$\begin{tikzcd}
\boxed{\text{boolean circuit}} \arrow[r, "O(n)"] & \boxed{\text{3CNF formula}} \arrow[d, Rightarrow, "\text{3SAT}"] \\
\boxed{\textsc{True} \text{ or } \textsc{False}} & \boxed{\textsc{True} \text{ or } \textsc{False}} \arrow[l, "\text{trivial}"']
\end{tikzcd}$$

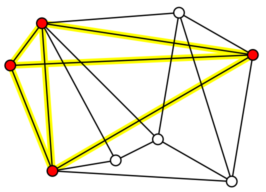
$$T_{\text{CSAT}}(n) \leq O(n) + T_{\text{3SAT}}(O(n)) \qquad \Longrightarrow \qquad T_{\text{3SAT}}(n) \geq T_{\text{CSAT}}(\Omega(n)) - O(n)$$

So 3SAT is NP-hard.

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

16.6 Maximum Clique Size (from 3SAT)

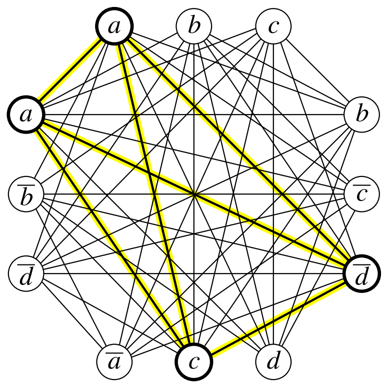
The last problem I'll consider in this lecture is a graph problem. A *clique* is another name for a complete graph. The *maximum clique size* problem, or simply MAXCLIQUE, is to compute, given a graph, the number of nodes in its largest complete subgraph.



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. Here's a diagram of the reduction:

$$
\begin{tikzcd}
\boxed{\text{3CNF formula with } k \text{ clauses}} \arrow[r, "O(n)"] & \boxed{\text{graph with } 3k \text{ nodes}} \arrow[d, Rightarrow, "\textsc{Clique}"] \\
\boxed{\textsc{True} \text{ or } \textsc{False}} & \boxed{\text{maximum clique size}} \arrow[l, "O(1)"']
\end{tikzcd}
$$

$$T_{\text{3SAT}}(n) \le O(n) + T_{\textsc{MaxClique}}(O(n)) \quad \Longrightarrow \quad T_{\textsc{MaxClique}}(n) \ge T_{\text{3SAT}}(\Omega(n)) - O(n)$$

16.7 Independent Set (from Clique)

An *independent set* is a collection of vertices is a graph with no edges between them. The IndependentSet problem is to find the largest independent set in a given graph.

There is an easy proof that IndependentSet is NP-hard, using a reduction from Clique. Any graph G has a *complement* $\overline{G}$ with the same vertices, but with exactly the opposite set of edges—(u, v) is an edge in $\overline{G}$ if and only if it is *not* an edge in G. A set of vertices forms a clique in G if and only if the same vertices are an independent set in $\overline{G}$. Thus, we can compute the largest clique in a graph simply by computing the largest independent set in the complement of the graph.

$$
\begin{tikzcd}
\boxed{\text{graph } G} \arrow[r, "O(n)"] & \boxed{\text{complement graph } \overline{G}} \arrow[d, Rightarrow, "\textsc{IndependentSet}"] \\
\boxed{\text{largest clique}} & \boxed{\text{largest independent set}} \arrow[l, "\text{trivial}"']
\end{tikzcd}
$$

16.8 Vertex Cover (from Independent Set)

A *vertex cover* of a graph is a set of vertices that touches every edge in the graph. The VertexCover problem is to find the smallest vertex cover in a given graph.

Again, the proof of NP-hardness is simple, and relies on just one fact: If I is an independent set in a graph $G = (V, E)$, then $V \setminus I$ is a vertex cover. Thus, to find the *largest* independent set, we just need to find the vertices that aren't in the *smallest* vertex cover of the same graph.

$$
\begin{tikzcd}
\boxed{\text{graph } G = (V, E)} \arrow[r, "\text{trivial}"] & \boxed{\text{graph } G = (V, E)} \arrow[d, Rightarrow, "\textsc{VertexCover}"] \\
\boxed{\text{largest independent set } V \setminus S} & \boxed{\text{smallest vertex cover } S} \arrow[l, "O(n)"']
\end{tikzcd}
$$

16.9 Graph Coloring (from 3SAT)

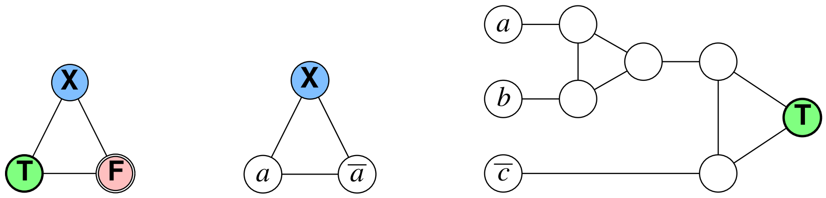
A *c-coloring* of a graph is a map $C : V \to \{1, 2, \ldots, c\}$ that assigns one of c 'colors' to each vertex, so that every edge has two different colors at its endpoints. The graph coloring problem is to find the smallest possible number of colors in a legal coloring. To show that this problem is NP-hard, it's enough to consider the special case 3Colorable: Given a graph, does it have a 3-coloring?

To prove that 3Colorable is NP-hard, we use a reduction from 3sat. Given a 3CNF formula, we produce a graph as follows. The graph consists of a *truth* gadget, one *variable* gadget for each variable in the formula, and one *clause* gadget for each clause in the formula.

The truth gadget is just a triangle with three vertices T, F, and X, which intuitively stand for True, false, and other. Since these vertices are all connected, they must have different colors in any 3-coloring. For the sake of convenience, we will name those colors True, False, and Other. Thus, when we say that a node is colored True, all we mean is that it must be colored the same as the node T.

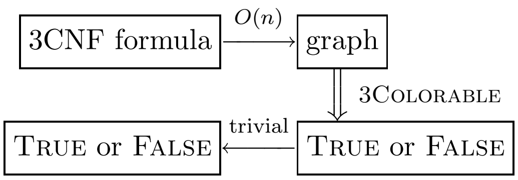
The variable gadget for a variable a is also a triangle joining two new nodes labeled a and $\overline{a}$ to node X in the truth gadget. Node a must be colored either True or False, and so node $\overline{a}$ must be colored either False or True, respectively.

Finally, each clause gadget joins three literal nodes to node T in the truth gadget using five new unlabeled nodes and ten edges; see the figure below. If all three literal nodes in the clause gadget are colored False, then the rightmost vertex in the gadget cannot have one of the three colors. Since the variable gadgets force each literal node to be colored either True or False, in any valid 3-coloring, at least one of the three literal nodes is colored True. I need to emphasize here that the final graph contains only *one* node T, only *one* node F, only *one* node $\bar{a}$ for each variable a, and so on.

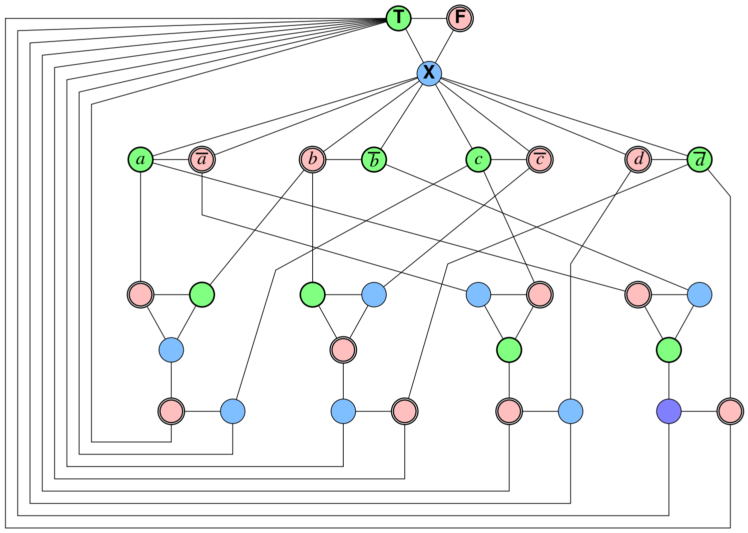


Gadgets for the reduction from 3SAT to 3-Colorability:
The truth gadget, a variable gadget for a, and a clause gadget for $(a \vee b \vee \bar{c})$.

The proof of correctness is just brute force. If the graph is 3-colorable, then we can extract a satisfying assignment from any 3-coloring—at least one of the three literal nodes in every clause gadget is colored True. Conversely, if the formula is satisfiable, then we can color the graph according to any satisfying assignment.



For example, the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ that I used to illustrate the MaxClique reduction would be transformed into the following graph. The 3-coloring is one of several that correspond to the satisfying assignment $a = c =$ True, $b = d =$ False.



A 3-colorable graph derived from a satisfiable 3CNF formula.

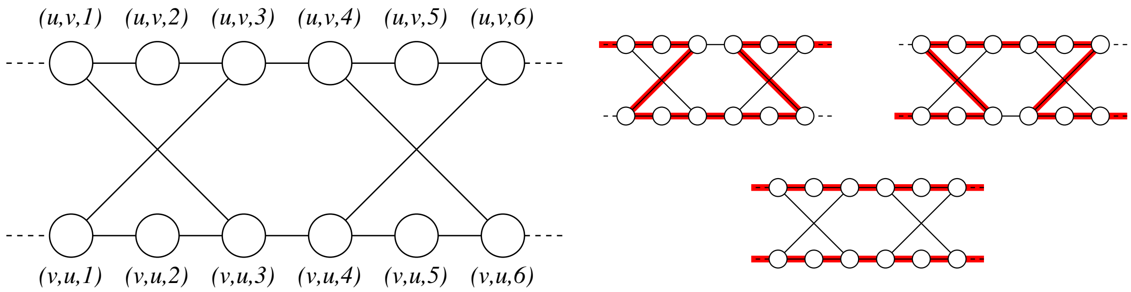
We can easily verify that a graph has been correctly 3-colored in linear time: just compare the endpoints of every edge. Thus, 3Coloring is in NP, and therefore NP-complete. Moreover, since 3Coloring is a special case of the more general graph coloring problem—What is the minimum number of colors?—the more problem is also NP-hard, but *not* NP-complete, because it's not a yes/no problem.

16.10 Hamiltonian Cycle (from Vertex Cover)

A *Hamiltonian cycle* is a graph is a cycle that visits every vertex exactly once. This is very different from an *Eulerian cycle*, which is actually a closed *walk* that traverses every *edge* exactly once. Eulerian cycles are easy to find and construct in linear time using a variant of depth-first search. Finding Hamiltonian cycles, on the other hand, is NP-hard.

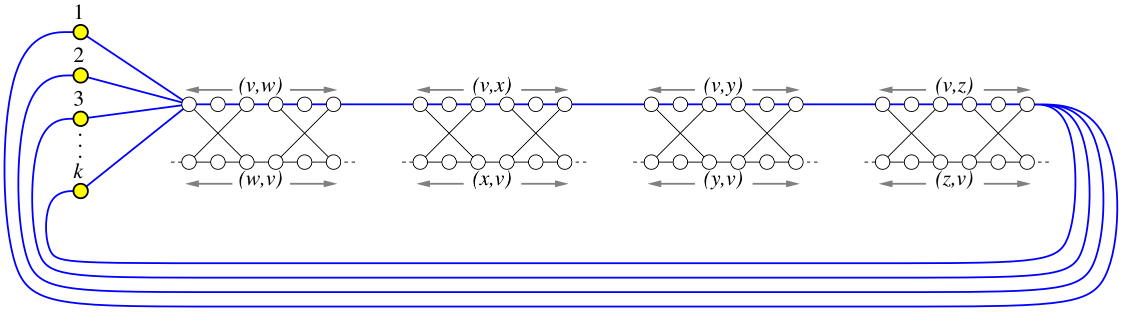
To prove this, we use a reduction from the vertex cover problem. Given a graph G and an integer k, we need to transform it into another graph G', such that G' has a Hamiltonian cycle if and only if G has a vertex cover of size k. As usual, our transformation consists of putting together several gadgets.

- For each edge (u, v) in G, we have an *edge gadget* in G' consisting of twelve vertices and fourteen edges, as shown below. The four corner vertices $(u, v, 1)$, $(u, v, 6)$, $(v, u, 1)$, and $(v, u, 6)$ each have an edge leaving the gadget. A Hamiltonian cycle can only pass through an edge gadget in one of three ways. Eventually, these will correspond to one or both of the vertices u and v being in the vertex cover.



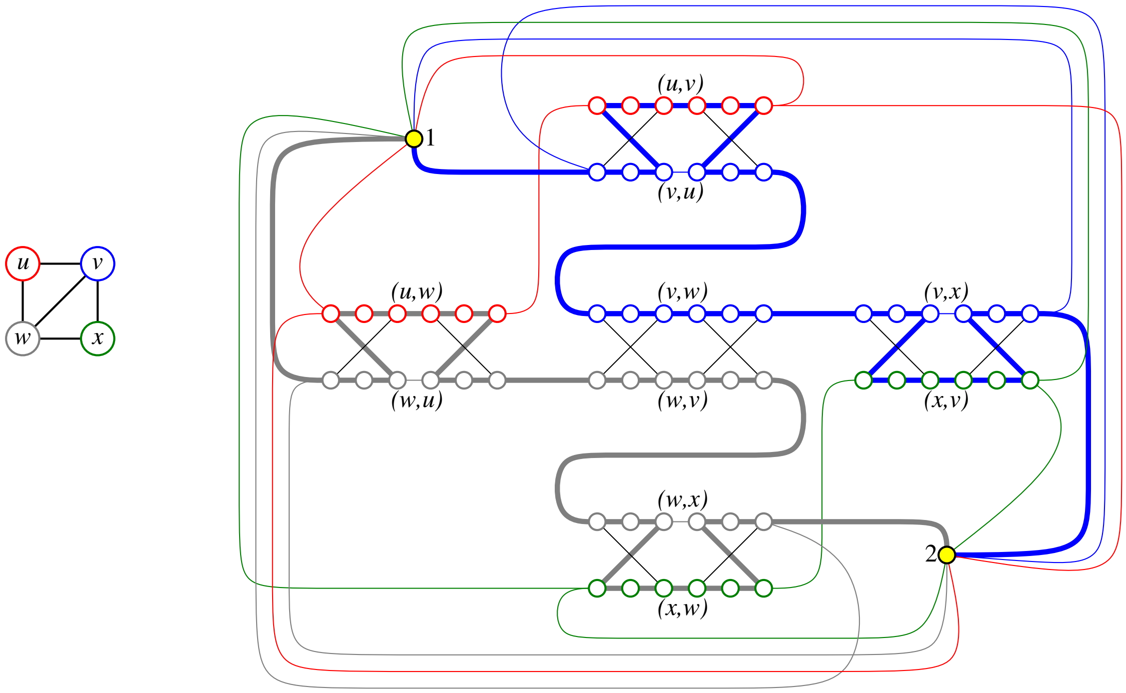
An edge gadget for (u, v) and the only possible Hamiltonian paths through it.

- G' also contains k *cover vertices*, simply numbered 1 through k.
- Finally, for each vertex u in G, we string together all the edge gadgets for edges (u, v) into a single *vertex chain*, and then connect the ends of the chain to all the cover vertices. Specifically, suppose u has d neighbors $v_1, v_2, \ldots, v_d$. Then G' has $d-1$ edges between $(u, v_i, 6)$ and $(u, v_{i+1}, 1)$, plus k edges between the cover vertices and $(u, v_1, 1)$, and finally k edges between the cover vertices and $(u, v_d, 6)$.



The vertex chain for v: all edge gadgets involving v are strung together and joined to the k cover vertices.

It's not hard to prove that if $\{v_1, v_2, \ldots, v_k\}$ is a vertex cover of G, then G' has a Hamiltonian cycle—start at cover vertex 1, through traverse the vertex chain for v_1, then visit cover vertex 2, then traverse the vertex chain for v_2, and so forth, eventually returning to cover vertex 1. Conversely, any Hamiltonian cycle in G' alternates between cover vertices and vertex chains, and the vertex chains correspond to the k vertices in a vertex cover of G. (This is a little harder to prove.) Thus, G has a vertex cover of size k if and only if G' has a Hamiltonian cycle.



The original graph G with vertex cover $\{v, w\}$, and the transformed graph G' with a corresponding Hamiltonian cycle. Vertex chains are colored to match their corresponding vertices.

The transformation from G to G' takes at most $O(n^2)$ time, so the Hamiltonian cycle problem is NP-hard. Moreover, since we can easily verify a Hamiltonian cycle in linear time, the Hamiltonian cycle problem is in NP, and therefore NP-complete.

A closely related problem to Hamiltonian cycles is the famous *traveling salesman problem*—Given a *weighted* graph G, find the shortest cycle that visits every vertex. Finding the shortest cycle is obviously harder than determining if a cycle exists at all, so the traveling salesman problem is also NP-hard.

16.11 Minesweeper (from Circuit SAT)

In 1999, Richard Kaye proved that the solitaire game Minesweeper is NP-complete, using a reduction from the original circuit satisfiability problem.[3] The reduction involves setting up gadgets for every possible feature of a boolean circuit: wires, AND gates, OR gates, NOT gates, wire crossings, and so forth. For all the gory details, see http://www.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.pdf!

16.12 Other Useful NP-hard Problems

Literally thousands of different problems have been proved to be NP-hard. I want to close this note by listing a few NP-hard problems that are useful in deriving reductions. I won't describe the NP-hardness for these problems, but you can find them in Garey and Johnson's *Angry Black Book of NP-Completeness.*[4]

- PlanarCircuitSAT: Given a boolean circuit that can be embedded in the plane so that no two wires cross, is there an input that makes the circuit output True? This can be proved NP-hard by reduction from the general circuit satisfiability problem, by replacing each crossing with a small series of gates. (This is an easy exercise.)
- NotAllEqual3SAT: Given a 3CNF formula, is there an assignment of values to the variables so that every clause contains at least one true literal *and* at least one false literal? This can be proved NP-hard by reduction from the usual 3SAT.
- Planar3SAT: Given a 3CNF boolean formula, consider a bipartite graph whose vertices are the clauses and variables, where an edge indicates that a variable (or its negation) appears in a clause. If this graph is planar, the 3CNF formula is also called planar. The Planar3SAT problem asks, given a planar 3CNF formula, whether it has a satifying assignment. This can be proven NP-hard by reduction from PlanarSat.
- PlanarNotAllEqual3SAT: You get the idea.
- Exact3DimensionalMatching or X3M: Given a set S and a collection of three-element subsets of S, called *triples*, is there a subcollection of disjoint triples that exactly cover S? This can be proved NP-hard by a reduction from 3SAT.
- Partition: Given a set S of n integers, are there subsets A and B such that $A \cup B = S$, $A \cap B = \varnothing$, and
$$\sum_{a \in A} a = \sum_{b \in B} b?$$
This can be proved NP-hard by a reduction from...

[3]Minesweeper is NP-complete. *Mathematical Intelligencer* 22(2):9–15, 2000.

[4]Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., 1979.

- SUBSETSUM: Given a set S of n integers and an integer T, is there a subset $A \subseteq S$ such that

$$\sum_{a \in A} a = S?$$

This is a generalization of PARTITION, where $T = (\sum S)/2$. SUBSETSUM can proved NP-hard by a (nontrivial!) reduction from either 3SAT or X3M.

- 3PARTITION: Given a set S with $3n$ elements, can it be partitioned into n disjoint subsets, each with 3 elements, such that every subset has the same sum. Note that this is *very* different from the PARTITION problem; I didn't make up the names. This can be proved NP-hard by reduction from X3M. The similar problem of dividing a set of $2n$ into n equal-weight *two*-element sets can be solved in $O(n \log n)$ time.

- SETCOVER: Given a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, find the smallest subcollection of S_i's that contains all the elements of $\bigcup_i S_i$. This is a generalization of both VERTEXCOVER and X3M.

- HITTINGSET: Given a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, find the minimum number of elements of $\bigcup_i S_i$ that hit every set in $\mathcal{S}$. This is also a generalization of VERTEXCOVER.

- LONGESTPATH: Given a non-negatively weighted graph G and two vertices u and v, what is the longest simple path from u to v in the graph? A path is *simple* if it visits each vertex at most once. This is a generalization of the HAMILTONIANPATH problem. Of course, the corresoinding *shortest* path problem is in P.

- STEINERTREE: Given a weighted, undirected graph G with some of the vertices marked, what is the minimum-weight subtree of G that contains every marked vertex? If *every* vertex is marked, the minimum Steiner tree is just the minimum spanning tree; if exactly two vertices are marked, the minimum Steiner tree is just the shortest path between them. This can be proved NP-hard by reduction to HAMILTONIANPATH.

- TETRIS: Given a Tetris board and a finite sequence of future pieces, can you survive? This was recently proved NP-hard by reduction from 3PARTITION.