The wonderful thing about standards is that
there are so many of them to choose from.

— Real Admiral Grace Murray Hopper

If a problem has no solution, it may not be a problem, but a fact —
not to be solved, but to be coped with over time.

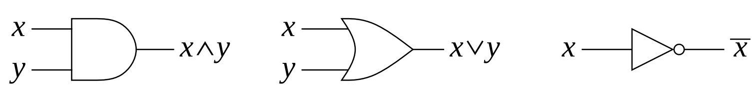
— Shimon Peres

24 NP-Hard Problems

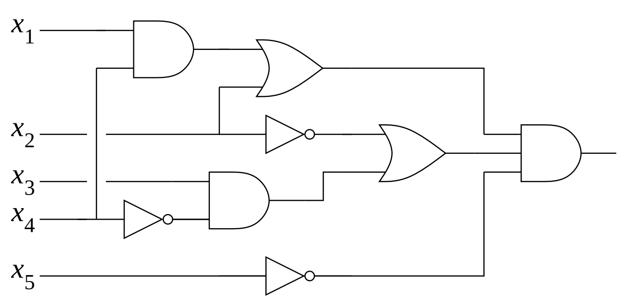
24.1 'Efficient' Problems

A generally-accepted minimum requirement for an algorithm to be considered 'efficient' is that its running time is polynomial: $O(n^c)$ for some constant c, where n is the size of the input.[1] Researchers 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. there are several so-called *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 of the circuit in polynomial (actually, *linear*) time using depth-first-search, since we can compute the output of a k-input gate in $O(k)$ time.



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.

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!

[1]This notion of efficiency was independently formalized by Alan Cobham (The intrinsic computational difficulty of functions. *Logic, Methodology, and Philosophy of Science (Proc. Int. Congress)*, 24–30, 1965), Jack Edmonds (Paths, trees, and flowers. *Canadian Journal of Mathematics* 17:449–467, 1965), and Michael Rabin (Mathematical theory of automata. *Proceedings of the 19th ACM Symposium in Applied Mathematics*, 153–175, 1966), although similar notions were considered more than a decade earlier by Kurt Gödel and John von Neumann.

24.2 P, NP, and co-NP

A *decision problem* is a problem whose output is a single boolean value: YES or NO.[2] Let me define three classes of decision problems:

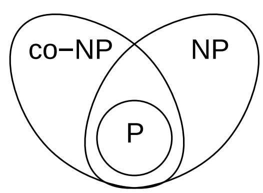
- ***P*** is the set of decision problems that can be solved in polynomial time.[3] Intuitively, P is the set of problems that can be solved quickly.
- ***NP*** is the set of decision problems with the following property: If the answer is YES, then there is a *proof* of this fact that can be checked in polynomial time. Intuitively, NP is the set of decision problems where we can verify a YES answer quickly if we have the solution in front of us.
- ***co-NP*** is the 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.

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. It is widely believed that circuit satisfiability is *not* in P or in co-NP, but nobody actually knows.

Every decision problem in P is also in NP. If a problem is in P, we can verify YES answers in polynomial time recomputing the answer from scratch! Similarly, any problem in P is also in co-NP.

One of the most important open questions in theoretical computer science is whether or not P = NP. Nobody knows. Intuitively, it should be obvious that P $\neq$ NP; the homeworks and exams in this class and others have (I hope) convinced you that problems can be incredibly hard to solve, even when the solutions are obvious in retrospect. But nobody knows how to prove it.

A more subtle but still open question is whether NP and co-NP are different. Even if we can verify every YES answer quickly, there's no reason to think that we can also verify NO answers quickly. For example, as far as we know, there is no short proof that a boolean circuit is *not* satisfiable. It is generally believed that NP $\neq$ co-NP, but nobody knows how to prove it.



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:

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

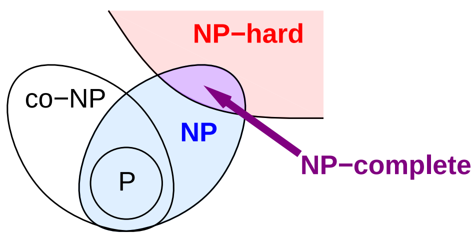
[2]Technically, I should be talking about *languages*, which are just sets of bit strings. The language associated with a decision problem is the set of bit strings for which the answer is YES. For example, for the problem is 'Is the input graph connected?', the corresponding language is the set of connected graphs, where each graph is represented as a bit string (for example, its adjacency matrix).

[3]More formally, P is the set of languages that can be recognized in polynomial time by a single-tape Turing machine. If you want to learn more about Turing machines, take CS 579.

Intuitively, 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.[4]

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 mathematical 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.

Finally, a problem is ***NP-complete*** if it is both NP-hard and an element of NP (or '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.

It is not immediately clear that *any* decision problems are NP-hard or NP-complete. NP-hardness is already a lot to demand of a problem; insisting that the problem also have a nondeterministic polynomial-time algorithm seems almost completely unreasonable. The following remarkable theorem was first published by Steve Cook in 1971 and independently by Leonid Levin in 1973.[5] I won't even sketch the proof, since I've been (deliberately) vague about the definitions.

The Cook-Levin Theorem. *Circuit satisfiability is NP-complete.*

[4]More formally, a problem Π is NP-hard if and only if, for any problem Π' in NP, there is a polynomial-time Turing reduction from Π' to Π—a Turing reduction just means a reduction that can be executed on a Turing machine. Polynomial-time Turing reductions are also called *Cook reductions*.

Complexity theorists prefer to define NP-hardness in terms of polynomial-time *many-one* reductions, which are also called *Karp reductions*. A *many-one* reduction from one language Π' to another language Π is an function $f : \Sigma^* \to \Sigma^*$ such that $x \in \Pi'$ if and only if $f(x) \in \Pi$. Every Karp reduction is a Cook reduction, but not vice versa. Every reduction (between decision problems) in these notes is a Karp reduction. This definition is preferred primarily because NP is closed under Karp reductions, but believed *not* to be closed under Cook reductions. Moreover, the two definitions of NP-hardness are equivalent only if NP=co-NP, which is considered unlikely. In fact, there are natural problems that are (1) NP-hard with respect to Cook reductions, but (2) NP-hard with respect to Karp reductions only if P=NP! On the other hand, the Karp definition *only* applies to decision problems, or more formally, sets of bit-strings.

To make things even more confusing, both Cook and Karp originally defined NP-hardness in terms of *logarithmic-space* reductions. Every logarithmic-space reduction is a polynomial-time reduction, but (we think) not vice versa. It is an open question whether relaxing the set of allowed (Cook or Karp) reductions from logarithmic-space to polynomial-time changes the set of NP-hard problems.

[5]Levin submitted his results, and discussed them in talks, several years before they were published. So naturally, in accordance with Stigler's Law, this result is often called 'Cook's Theorem'. Cook won the Turing award for his proof; Levin did not.

24.4 Reductions and SAT

To prove that a problem is NP-hard, we use a *reduction argument*. Reducing problem A to another problem B means describing an algorithm to solve problem A under the assumption that an algorithm for problem B already exists. You're already used to doing reductions, only you probably call it something else, like writing subroutines or utility functions, or modular programming. To prove something is NP-hard, we describe a similar transformation between problems, but not in the direction that most people expect.

You should tattoo the following rule of onto the back of your hand.

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

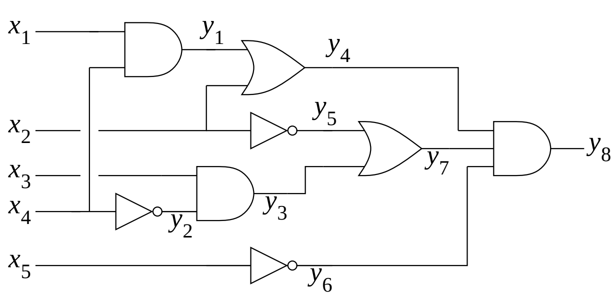
In other words, to prove that your problem is hard, you need to describe an algorithm to solve a *different* problem, which you already know is hard, using a mythical algorithm for *your* problem as a subroutine. The essential logic is a proof by contradiction. Your reduction shows implies that if your problem were easy, then the other problem would be easy, too. Equivalently, since you know the other problem is hard, your problem must also be hard.

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:

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

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

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.

24.5 3SAT (from SAT)

A special case of SAT that is particularly useful in proving NP-hardness results is called *3SAT*.

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:

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

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:

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

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—every binary gate in the original circuit has been transformed into at most five clauses. Even if the formula size were a large *polynomial* function (like n^{573}) of the circuit size, we would still 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}
$$

This process transforms the circuit into an equivalent 3CNF formula; the output formula is satisfiable if and only if the input circuit is satisfiable. As with the more general SAT problem, the formula is only a constant factor larger than 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:

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

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

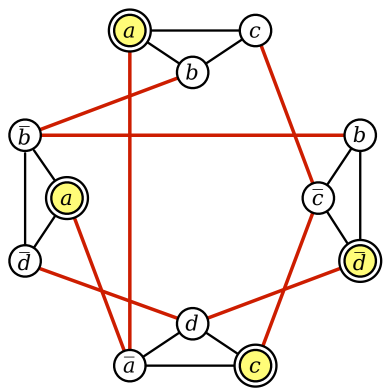
We conclude 3SAT is NP-hard. And because 3SAT is a special case of SAT, it is also in NP. Therefore, 3SAT is NP-complete.

24.6 Maximum Independent (from 3SAT)

For the next few problems we consider, the input is a simple, unweighted graph, and the problem asks for the size of the largest or smallest subgraph satisfying some structural property.

Let G be an arbitrary graph. An ***independent set*** in G is a subset of the vertices of G with no edges between them. The *maximum independent set* problem, or simply **MaxIndSet**, asks for the size of the largest independent set in a given graph.

I'll prove that MaxIndSet is NP-hard (but not NP-complete, since it isn't a decision problem) using a reduction from 3SAT. I'll describe a reduction from a 3CNF formula into a graph that has an independent set 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 the same clause, or (2) they correspond to a variable and its inverse. 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.



A graph derived from a 3CNF formula, and an independent set of size 4.
Black edges join literals from the same clause; red (heavier) edges join contradictory literals.

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

1. **independent set $\Longrightarrow$ satisfying assignment:** If the graph has an independent set of k vertices, then each vertex must come from a different clause. To obtain a satisfying assignment, we assign the value TRUE to each literal in the independent set. Since contradictory literals are connected by edges, this assignment is consistent. There may be variables that have no literal in the independent set; we can set these to any value we like. The resulting assignment satisfies the original 3CNF formula.

2. **satisfying assignment $\Longrightarrow$ independent set:** If we have a satisfying assignment, then we can choose one literal in each clause that is TRUE. Those literals form an independent set in the graph.

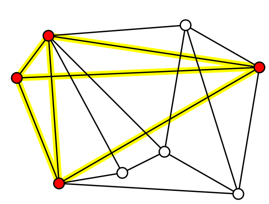
Thus, the reduction is correct. Since the reduction from 3CNF formula to graph takes polynomial time, we conclude that MAXINDSET is NP-hard. Here's a diagram of the reduction:

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

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

24.7 Clique (from Independent Set)

A *clique* is another name for a complete graph, that is, a graph where every pair of vertices is connected by an edge. 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.

There is an easy proof that MaxClique is NP-hard, using a reduction from MaxIndSet. Any graph G has an *edge-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 is independent in G if and only if the same vertices define a clique in $\overline{G}$. Thus, we can compute the largest independent in a graph simply by computing the largest clique in the complement of the graph.

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

24.8 Vertex Cover (from Independent Set)

A *vertex cover* of a graph is a set of vertices that touches every edge in the graph. The MinVertexCover 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.

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

24.9 Graph Coloring (from 3SAT)

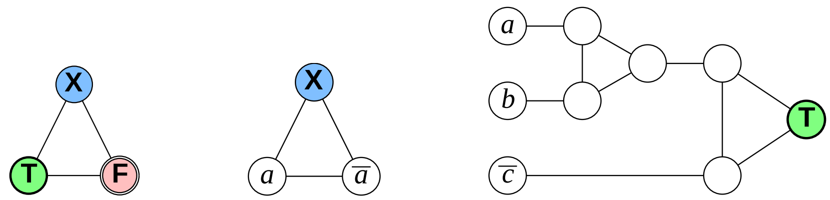
A *k-coloring* of a graph is a map $C : V \to \{1, 2, \ldots, k\}$ that assigns one of k 'colors' to each vertex, so that every edge has two different colors at its endpoints. The graph coloring problem is to find the smallest possible number of colors in a legal coloring. To show that this problem is 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.

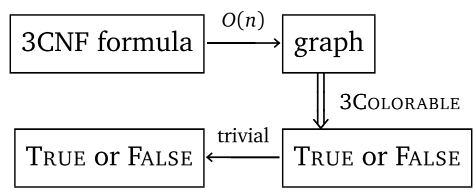
The variable gadget for a variable a is also a triangle joining two new nodes labeled a and $\overline{a}$ to node X in the truth gadget. Node a must be colored either True or False, and so node $\overline{a}$ must be colored either False or True, respectively.

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, and only *two* nodes a and $\bar{a}$ for each variable.

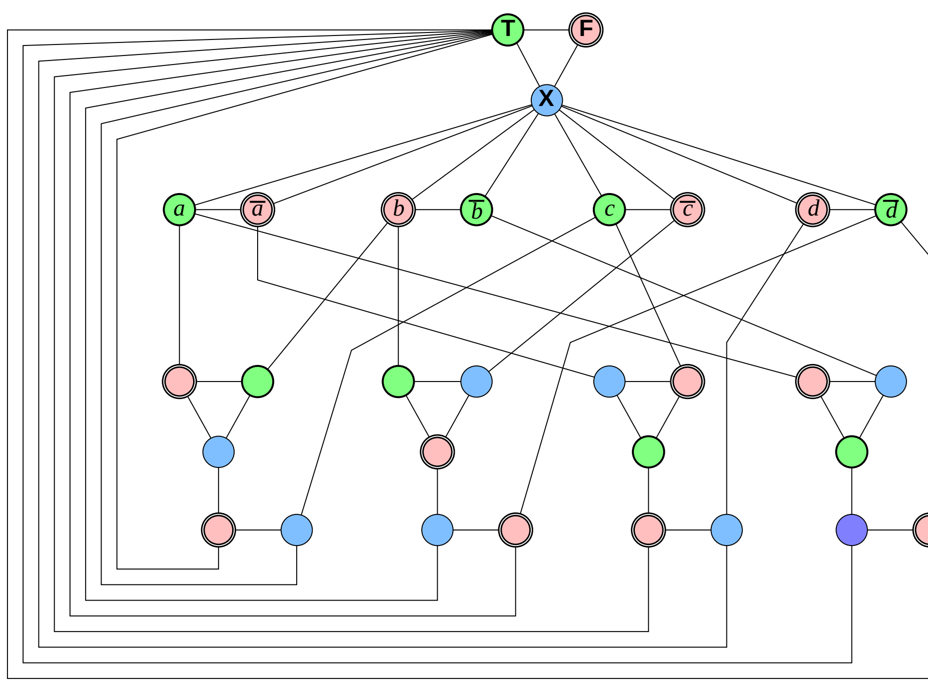


Gadgets for the reduction from 3SAT to 3-Colorability:
The truth gadget, a variable gadget for a, and a clause gadget for $(a \vee b \vee \bar{c})$.

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.

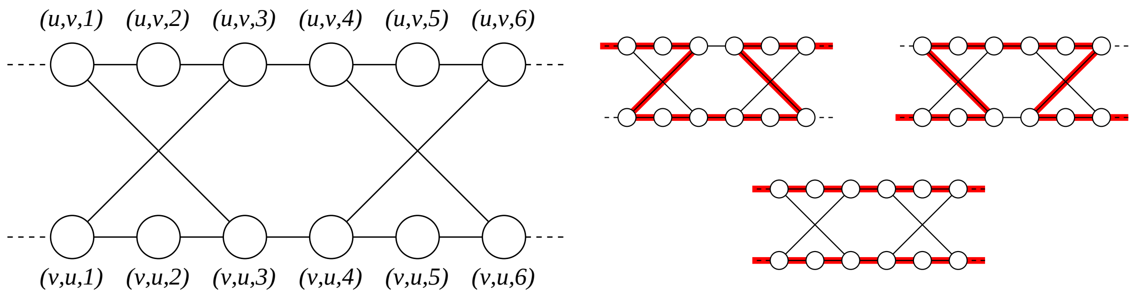
We can easily verify that a graph has been correctly 3-colored in linear time: just compare the endpoints of every edge. Thus, 3COLORING is in NP, and therefore NP-complete. Moreover, since 3COLORING is a special case of the more general graph coloring problem—What is the minimum number of colors?—the more problem is also NP-hard, but *not* NP-complete, because it's not a decision problem.

24.10 Hamiltonian Cycle (from Vertex Cover)

A *Hamiltonian cycle* in 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 uses several gadgets.

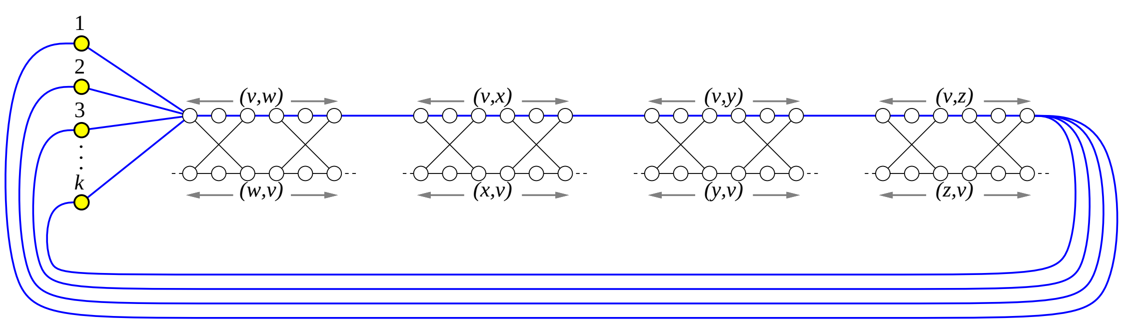
- For each edge (u,v) in G, we have an *edge gadget* in G' consisting of twelve vertices and fourteen edges, as shown below. The four corner vertices $(u,v,1)$, $(u,v,6)$, $(v,u,1)$, and $(v,u,6)$ each have an edge leaving the gadget. A Hamiltonian cycle can only pass through an edge gadget in one of 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.

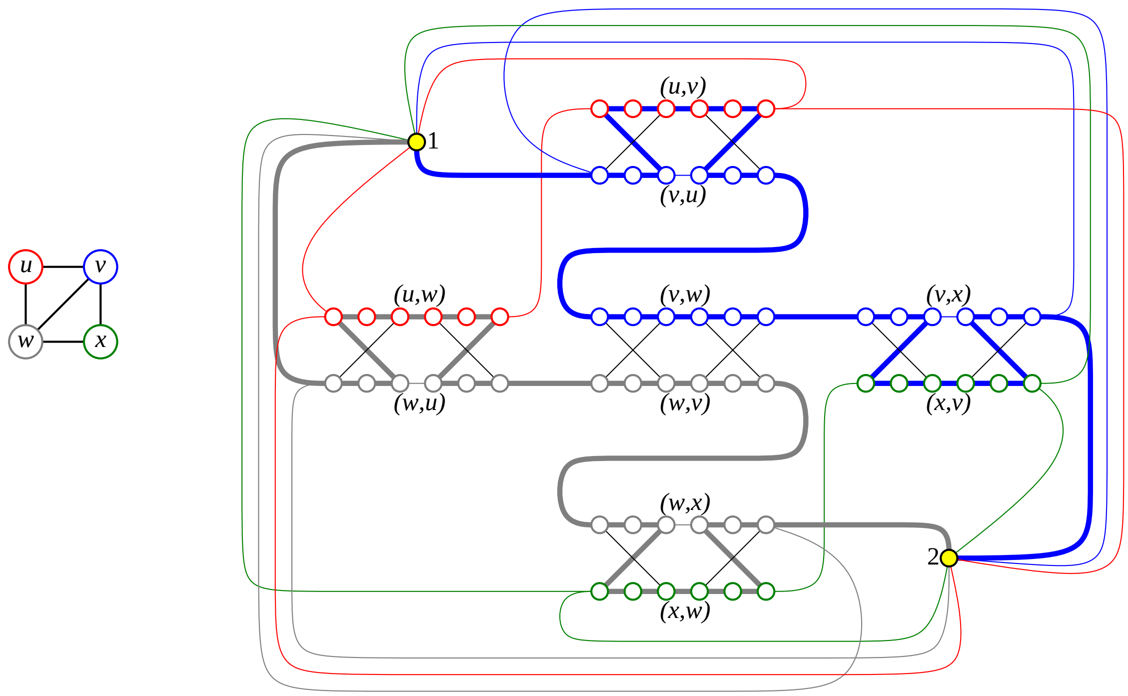
- G' also contains k *cover vertices*, simply numbered 1 through k.

- 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 $v_1, v_2, \dots, v_d$. 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 $\{v_1, v_2, \dots, v_k\}$ 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.

24.11 Subset Sum (from Vertex Cover)

The last problem that we will prove NP-hard is the SubsetSum problem considered in the very first lecture on recursion: Given a set X of integers and an integer t, determine whether X has a subset whose elements sum to t.

To prove this problem is NP-hard, we apply a reduction from the vertex cover problem. Given a graph G and an integer k, we need to transform it into set of integers X and an integer t, such that X has a subset that sums to t if and only if G has an vertex cover of size k. Our transformation uses just two 'gadgets'; these are *integers* representing vertices and edges in G.

Number the *edges* of G arbitrarily from 0 to $m-1$. Our set X contains the integer $b_i := 4^i$ for each edge i, and the integer

$$a_v := 4^m + \sum_{i \in \Delta(v)} 4^i$$

for each vertex v, where $\Delta(v)$ is the set of edges that have v has an endpoint. Alternately, we can think of each integer in X as an $(m+1)$-digit number written in base 4. The mth digit is 1 if the integer represents a vertex, and 0 otherwise. For each $i < m$, the ith digit is 1 if the integer represents edge i or one of its endpoints, and 0 otherwise. Finally, we set the target sum

$$t := k \cdot 4^m + \sum_{i=0}^{m-1} 2 \cdot 4^i.$$

Now let's prove that the reduction is correct. First, suppose there is a vertex cover of size k in the original graph G. Consider the subset $X_C \subseteq X$ that includes a_v for every vertex v in the vertex cover, and b_i for every edge i that has *exactly one* vertex in the cover. The sum of these integers, written in base 4, has a 2 in each of the first m digits; in the most significant digit, we are summing exactly k 1's. Thus, the sum of the elements of X_C is exactly t.

On the other hand, suppose there is a subset $X' \subseteq X$ that sums to t. Specifically, we must have

$$\sum_{v \in V'} a_v + \sum_{i \in E'} b_i = t$$

for some subsets $V' \subseteq V$ and $E' \subseteq E$. Again, if we sum these base-4 numbers, there are no carries in the first m digits, because for each i there are only three numbers in X whose ith digit is 1. Each edge number b_i contributes only one 1 to the ith digit of the sum, but the ith digit of t is 2. Thus, for each edge in G, at least one of its endpoints must be in V'. In other words, V is a vertex cover. On the other hand, only vertex numbers are larger than 4^m, and $\lfloor t/4^m \rfloor = k$, so V' has at most k elements. (In fact, it's not hard to see that V' has *exactly* k elements.)

For example, given the four-vertex graph used on the previous page to illustrate the reduction to Hamiltonian cycle, our set X might contain the following base-4 integers:

$$\begin{aligned}
b_{uv} &:= 010000_4 = 256 \\
b_{uw} &:= 001000_4 = 64 \\
b_{vw} &:= 000100_4 = 16 \\
b_{vx} &:= 000010_4 = 4 \\
b_{wx} &:= 000001_4 = 1 \\
a_u &:= 111000_4 = 1344 \\
a_v &:= 110110_4 = 1300 \\
a_w &:= 101101_4 = 1105 \\
a_x &:= 100011_4 = 1029
\end{aligned}$$

If we are looking for a vertex cover of size 2, our target sum would be $t := 222222_4 = 2730$. Indeed, the vertex cover $\{v, w\}$ corresponds to the subset $\{a_v, a_w, b_{uv}, b_{uw}, b_{vx}, b_{wx}\}$, whose sum is $1300 + 1105 + 256 + 64 + 4 + 1 = 2730$.

The reduction can clearly be performed in polynomial time. Since VERTEXCOVER is NP-hard, it follows that SUBSETSUM is NP-hard.

There is one subtle point that needs to be emphasized here. Way back at the beginning of the semester, we developed a dynamic programming algorithm to solve SUBSETSUM in time $O(nt)$. Isn't this a polynomial-time algorithm? Nope. True, the running time is polynomial in n and t, but in order to qualify as a true polynomial-time algorithm, the running time must be a polynomial function *of the size of the input*. The *values* of the elements of X and the target sum t could be exponentially larger than the number of input bits. Indeed, the reduction we just described produces exponentially large integers, which would force our dynamic programming algorithm to run in exponential time. Algorithms like this are called *pseudo-polynomial-time*, and any NP-hard problem with such an algorithm is called *weakly* NP-hard.

24.12 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 most of them in Garey and Johnson's classic *Scary Black Book of NP-Completeness.*[6]

- 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[7] 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 satisfying assignment. This can be proved NP-hard by reduction from PlanarCircuitSAT.[8]

- Exact3DimensionalMatching or X3M: Given a set S and a collection of three-element subsets of S, called *triples*, is there a sub-collection 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 simple reduction from SubsetSum. Like SubsetSum, the Partition problem is only weakly NP-hard.

- 3Partition: Given a set S of $3n$ integers, can it be partitioned into n disjoint subsets, each with 3 elements, such that every subset has exactly 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. Unlike Partition, the 3Partition problem is *strongly* NP-hard, that is, it remains NP-hard even if the input numbers are less than some polynomial in n. The similar problem of dividing a set of $2n$ integers into n equal-weight *two*-element sets can be solved in $O(n \log n)$ time.

- SetCover: Given a collection of sets $\mathscr{S} = \{S_1, S_2, \ldots, S_m\}$, find the smallest sub-collection 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 $\mathscr{S} = \{S_1, S_2, \ldots, S_m\}$, find the minimum number of elements of $\bigcup_i S_i$ that hit every set in $\mathscr{S}$. 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 corresponding *shortest* path problem is in P.

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

[7] or at least *nondeterministically* easy

[8] Surprisingly, PlanarNotAllEqual3SAT is solvable in polynomial time!

- 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.

Most interesting puzzles and solitaire games have been shown to be NP-hard, or to have NP-hard generalizations. (Arguably, if a game or puzzle isn't at least NP-hard, it isn't interesting!) Some familiar examples include Minesweeper (by reduction from CircuitSAT)[9], Tetris (by reduction from 3Partition)[10], and Sudoku (by reduction from 3SAT)[11]. Most two-player games[12] like tic-tac-toe, reversi, checkers, chess, go, mancala—or more accurately, appropriate generalizations of these constant-size games to arbitrary board sizes—are not just NP-hard, but PSPACE-hard or even EXP-hard.[13]

[9] Richard Kaye. Minesweeper is NP-complete. *Mathematical Intelligencer* 22(2):9–15, 2000. http://www.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.pdf

[10] Ron Breukelaar*, Erik D. Demaine, Susan Hohenberger*, Hendrik J. Hoogeboom, Walter A. Kosters, and David Liben-Nowell*. Tetris is Hard, Even to Approximate. International Journal of Computational Geometry and Applications 14:41–68, 2004. The reduction was *considerably* simplified between its discovery in 2002 and its publication in 2004.

[11] Takayuki Yato and Takahiro Seta. Complexity and Completeness of Finding Another Solution and Its Application to Puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E86-A(5):1052–1060, 2003. http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.pdf.

[12] For a good (but now slightly dated) overview of known results on the computational complexity of games and puzzles, see Erik D. Demaine's survey "Playing Games with Algorithms: Algorithmic Combinatorial Game Theory" at http://arxiv.org/abs/cs.CC/0106019.

[13] PSPACE and EXP are the next two big steps above NP in the complexity hierarchy.

PSPACE is the set of decision problems that can be solved using polynomial space. Every problem in NP (and therefore in P) is also in PSPACE. It is generally believed that NP $\neq$ PSPACE, but nobody can even prove that P $\neq$ PSPACE. A problem Π is PSPACE-hard if, for any problem Π' that can be solved using polynomial *space*, there is a polynomial-*time* many-one reduction from Π' to Π. If any PSPACE-hard problem is in NP, then PSPACE=NP.

EXP (also called EXPTIME) is the set of decision problems that can be solved in exponential time: at most 2^{n^c} for some $c > 0$. Every problem in PSPACE (and therefore in NP (and therefore in P)) is also in EXP. It is generally believed that PSPACE $\subsetneq$ EXP, but nobody can even prove that NP $\neq$ EXP. We *do* know that P $\subsetneq$ EXP, but we do not know of a single natural decision problem in P $\setminus$ EXP. A problem Π is EXP-hard if, for any problem Π' that can be solved in *exponential* time, there is a *polynomial*-time many-one reduction from Π' to Π. If any EXP-hard problem is in PSPACE, then EXP=PSPACE.

Then there's NEXP, then EXPSPACE, then EEXP, then NEEXP, then EEXPSPACE, and so on ad infinitum. Whee!

Exercises

1. Consider the following problem, called BoxDepth: Given a set of n axis-aligned rectangles in the plane, how big is the largest subset of these rectangles that contain a common point?

   (a) Describe a polynomial-time reduction from BoxDepth to MaxClique.

   (b) Describe and analyze a polynomial-time algorithm for BoxDepth. *[Hint: $O(n^3)$ time should be easy, but $O(n \log n)$ time is possible.]*

   (c) Why don't these two results imply that P=NP?

2. (a) Describe a polynomial-time reduction from Partition to SubsetSum.

   (b) Describe a polynomial-time reduction from SubsetSum to Partition.

3. (a) Describe and analyze and algorithm to solve Partition in time $O(nM)$, where n is the size of the input set and M is the sum of the absolute values of its elements.

   (b) Why doesn't this algorithm imply that P=NP?

4. A boolean formula is in *disjunctive normal form* (or *DNF*) if it consists of a *disjunction* (Or) or several *terms*, each of which is the conjunction (And) of one or more literals. For example, the formula

$$(\overline{a} \wedge b \wedge \overline{c}) \vee (b \wedge c) \vee (a \wedge \overline{b} \wedge \overline{c})$$

   is in disjunctive normal form. DNF-SAT asks, given a boolean formula in disjunctive normal form, whether that formula is satisfiable.

   (a) Show that DNF-SAT is in P.

   (b) What is the error in the following argument that P=NP?

   > Suppose we are given a boolean formula in conjunctive normal form with at most three literals per clause, and we want to know if it is satisfiable. We can use the distributive law to construct an equivalent formula in disjunctive normal form. For example,
   >
   > $$(a \vee b \vee \overline{c}) \wedge (\overline{a} \vee \overline{b}) \iff (a \wedge \overline{b}) \vee (b \wedge \overline{a}) \vee (\overline{c} \wedge \overline{a}) \vee (\overline{c} \wedge \overline{b})$$
   >
   > Now we can use the algorithms from part (a) to determine, in polynomial time, whether the resulting DNF formula is satisfiable. We have just solved 3SAT in polynomial time! Since 3SAT is NP-hard, we must conclude that P=NP.

5. (a) Describe and analyze a polynomial-time algorithm for 2Partition. Given a set S of $2n$ positive integers, your algorithm will determine in polynomial time whether the elements of S can be split into n disjoint pairs whose sums are all equal.

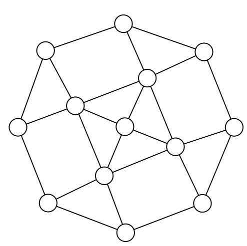
   (b) Describe and analyze a polynomial-time algorithm for 2Color. Given an undirected graph G, your algorithm will determine in polynomial time whether G has a proper coloring that uses only two colors.

   (c) Describe and analyze a polynomial-time algorithm for 2SAT. Given a boolean formula Φ in conjunctive normal form, with exactly *two* literals per clause, your algorithm will determine in polynomial time whether Φ has a satisfying assignment.

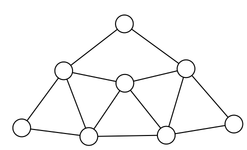
6. (a) Prove that PLANARCIRCUITSAT is NP-complete.
   (b) Prove that NOTALLEQUAL3SAT is NP-complete.
   (c) Prove that the following variant of 3SAT is NP-complete: Given a formula ϕ in conjunctive normal form where each clause contains at most 3 literals and each variable appears in at most 3 clauses, is ϕ satisfiable?

7. Jeff tries to make his students happy. At the beginning of class, he passes out a questionnaire that lists a number of possible course policies in areas where he is flexible. Every student is asked to respond to each possible course policy with one of "strongly favor", "mostly neutral", or "strongly oppose". Each student may respond with "strongly favor" or "strongly oppose" to at most five questions. Because Jeff's students are very understanding, each student is happy if (but only if) he or she prevails in just one of his or her strong policy preferences. Either describe a polynomial-time algorithm for setting course policy to maximize the number of happy students, or show that the problem is NP-hard.

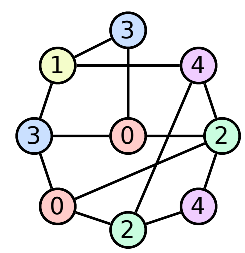
8. (a) Using the gadget on the right below, prove that deciding whether a given *planar* graph is 3-colorable is NP-complete. *[Hint: Show that the gadget can be 3-colored, and then replace any crossings in a planar embedding with the gadget appropriately.]*
   (b) Using part (a) and the middle gadget below, prove that deciding whether a planar graph *with maximum degree 4* is 3-colorable is NP-complete. *[Hint: Replace any vertex with degree greater than 4 with a collection of gadgets connected so that no degree is greater than four.]*



(a) Gadget for planar 3-colorability.



(b) Gadget for degree-4 planar 3-colorability.



(c) A careful 5-coloring.

9. Recall that a 5-coloring of a graph G is a function that assigns each vertex of G an 'color' from the set $\{0,1,2,3,4\}$, such that for any edge uv, vertices u and v are assigned different 'colors'. A 5-coloring is *careful* if the colors assigned to adjacent vertices are not only distinct, but differ by more than 1 (mod 5). Prove that deciding whether a given graph has a careful 5-coloring is NP-complete. *[Hint: Reduce from the standard* 5COLORABLE *problem.]*

10. Prove that the following problems are NP-complete.
    (a) Given two undirected graphs G and H, is G isomorphic to a subgraph of H?
    (b) Given an undirected graph G, does G have a spanning tree in which every node has degree at most 17?
    (c) Given an undirected graph G, does G have a spanning tree with at most 42 leaves?

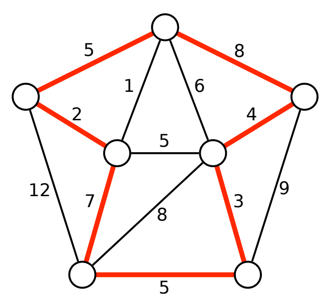
11. The RECTANGLETILING problem asks, given a 'large' rectangle R and several 'small' rectangles $r_1, r_2, \ldots, r_n$, whether the small rectangles can be placed inside the larger rectangle with no gaps or overlaps. Prove that RECTANGLETILING is NP-complete.

12. (a) A *tonian path* in a graph G is a path that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian path is NP-complete.
    (b) A *tonian cycle* in a graph G is a cycle that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian cycle is NP-complete. *[Hint: Use part (a).]*

13. For each problem below, either describe a polynomial-time algorithm or prove that the problem is NP-complete.
    (a) A *double-Eulerian circuit* in an undirected graph G is a closed walk that traverses every edge in G exactly twice. Given a graph G, does G have a double-Eulerian circuit?
    (b) A *double-Hamiltonian circuit* in an undirected graph G is a closed walk that visits every vertex in G exactly twice. Given a graph G, does G have a double-Hamiltonian circuit?

14. A boolean formula in *exclusive-or conjunctive normal form* (XCNF) is a conjunction (AND) of several *clauses*, each of which is the *exclusive*-or of several literals; that is, a clause is true if and only if it contains an odd number of true literals. The XCNF-SAT problem asks whether a given XCNF formula is satisfiable. Either describe a polynomial-time algorithm for XCNF-SAT or prove that it is NP-hard.

15. Let G be an undirected graph with weighted edges. A *heavy Hamiltonian cycle* is a cycle C that passes through each vertex of G exactly once, such that the total weight of the edges in C is at least half of the total weight of all edges in G. Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-complete.



A heavy Hamiltonian cycle. The cycle has total weight 34; the graph has total weight 67.

16. *Pebbling* is a solitaire game played on an undirected graph G, where each vertex has zero or more *pebbles*. A single *pebbling move* consists of removing two pebbles from a vertex v and adding one pebble to an arbitrary neighbor of v. (Obviously, the vertex v must have at least two pebbles before the move.) The PEBBLEDESTRUCTION problem asks, given a graph $G = (V, E)$ and a pebble count $p(v)$ for each vertex v, whether is there a sequence of pebbling moves that removes all but one pebble. Prove that PEBBLEDESTRUCTION is NP-complete.

17. The next time you are at a party, one of the guests will suggest everyone play a round of Three-Way Mumbletypeg, a game of skill and dexterity that requires three teams and a knife. The official Rules of Three-Way Mumbletypeg (fixed during the Holy Roman Three-Way Mumbletypeg Council in 1625) require that (1) each team *must* have at least one person, (2) any two people on the same team *must* know each other, and (3) everyone watching the game *must* be on one of the three teams. Of course, it will be a really *fun* party; nobody will want to leave. There will be several pairs of people at the party who don't know each other. The host of the party, having heard thrilling tales of your prowess in all things algorithmic, will hand you a list of which pairs of party-goers know each other and ask you to choose the teams, while he sharpens the knife.

    Either describe and analyze a polynomial time algorithm to determine whether the party-goers can be split into three legal Three-Way Mumbletypeg teams, or prove that the problem is NP-hard.

18. (a) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary weighted graph G, the length of the shortest Hamiltonian cycle in G. Describe and analyze a **polynomial-time** algorithm that computes, given an arbitrary weighted graph G, the shortest Hamiltonian cycle in G, using this magic black box as a subroutine.

    (b) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary graph G, the number of vertices in the largest complete subgraph of G. Describe and analyze a **polynomial-time** algorithm that computes, given an arbitrary graph G, a complete subgraph of G of maximum size, using this magic black box as a subroutine.

    (c) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary weighted graph G, whether G is 3-colorable. Describe and analyze a **polynomial-time** algorithm that either computes a proper 3-coloring of a given graph or correctly reports that no such coloring exists, using the magic black box as a subroutine. *[Hint: The input to the magic black box is a graph. Just a graph. Vertices and edges. Nothing else.]*

    (d) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary boolean formula Φ, whether Φ is satisfiable. Describe and analyze a **polynomial-time** algorithm that either computes a satisfying assignment for a given boolean formula or correctly reports that no such assignment exists, using the magic black box as a subroutine.

    *(e) Suppose you are given a magic black box that can determine **in polynomial time**, given an initial Tetris configuration and a finite sequence of Tetris pieces, whether a perfect player can play every piece. (This problem is NP-hard.) Describe and analyze a **polynomial-time** algorithm that computes the shortest Hamiltonian cycle in a given weighted graph in polynomial time, using this magic black box as a subroutine. *[Hint: Use part (a). You do not need to know anything about Tetris to solve this problem.]*