[I]n his short and broken treatise he provides an eternal example—not of laws, or even of method, for there is no method except to be very intelligent, but of intelligence itself swiftly operating the analysis of sensation to the point of principle and definition.

— T. S. Eliot on Aristotle, "The Perfect Critic", *The Sacred Wood* (1921)

The nice thing about standards is that you have so many to choose from; furthermore, if you do not like any of them, you can just wait for next year's model.

— Andrew S. Tannenbaum, *Computer Networks* (1981)

It is a rare mind indeed that can render the hitherto non-existent blindingly obvious. The cry "I could have thought of that" is a very popular and misleading one, for the fact is that they didn't, and a very significant and revealing fact it is too.

— Dirk Gently to Richard McDuff
in Douglas Adams' *Dirk Gently's Holistic Detective Agency* (1987)

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, as quoted by David Rumsfeld, *Rumsfeld's Rules* (2001)

12

NP-Hardness

12.1 A Game You Can't Win

Imagine that a salesman in a red suit, who looks suspiciously like Tom Waits, presents you with a black steel box with n binary switches on the front and a light bulb on top. The salesman tells you that the state of the light bulb is controlled by a complex *boolean circuit*—a collection of AND, OR, and NOT gates connected by wires, with one input wire for each switch and a single output wire for the bulb. He then asks you a simple question: Is it possible to set the switches so that the light bulb turns on? If you can answer this question correctly, he will give you ~~one million~~ one hundred billion dollars; if you answer incorrectly, or if you die without answering, he will take your soul.

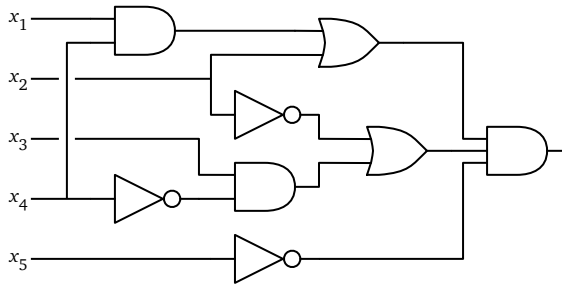**Figure 12.1.** An AND gate, an OR gate, and a NOT gate.

Figure 12.2. A boolean circuit. Inputs enter from the left, and the output leaves to the right.

As far as you can tell, the Adversary hasn't connected the switches to the light bulb at all, so no matter how you set the switches, the light bulb will stay off. If you declare that it *is* possible to turn on the light, the Adversary will open the box and reveal that there is no circuit at all. But if you declare that it is *not* possible to turn on the light, before testing all 2^n settings, the Adversary will magically create a circuit inside the box that turns on the light *if and only if* the switches are in one of the settings you haven't tested, and then flip the switches to that setting, turning on the light. (You can't detect the Adversary's cheating, because you can't see inside the box until the end.) The only way to *provably* answer the Adversary's question correctly is to try all 2^n possible settings. You quickly realize that this will take *far* longer than you expect to live, so you gracefully decline the Adversary's offer.

The Adversary smiles and says, in a growl like Heath Ledger's Joker after smoking a carton of Marlboros, "Ah, yes, of course, you have no reason to trust me. But perhaps I can set your mind at ease." He hands you a large roll of parchment—which you hope was made from *sheep* skin—with a circuit diagram drawn (or perhaps tattooed) on it. "Here are the complete plans for the circuit inside the box. Feel free to poke around inside the box to make sure the plans are correct. Or build your own circuit from these plans. Or write a computer program to simulate the circuit. Whatever you like. If you discover that the plans don't match the actual circuit in the box, you win the hundred billion bucks." A few spot checks convince you that the plans have no obvious flaws; subtle cheating appears to be impossible.

But you should still decline the Adversary's "generous" offer. The problem that the Adversary is posing is called **circuit satisfiability** or *CircuitSat*: Given a boolean circuit, is there a set of inputs that makes the circuit output True, or conversely, whether the circuit *always* outputs False. For any *particular* input setting, we can calculate the output of the circuit in polynomial (actually, *linear*) time using depth-first-search. But nobody knows how to solve CircuitSat faster than trying all 2^n possible inputs to the circuit by brute force, which requires exponential time. Admittedly, nobody has actually formally *proved* that we can't

beat brute force—maybe, just *maybe*, there's a clever algorithm that just hasn't been discovered yet—but nobody has actually formally proved that anti-gravity unicorns don't exist, either. For all practical purposes, it's safe to assume that there is no fast algorithm for CIRCUITSAT.

You tell the salesman no. He smiles and says, "You're smarter than you look, kid," and then flies away on his anti-gravity unicorn.

12.2   P versus NP

A minimal requirement for an algorithm to be considered "efficient" is that its running time is bounded by a polynomial function of the input size: $O(n^c)$ 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 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.

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

- *P* is the set of decision problems that can be solved in polynomial time. 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 essentially 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 a given boolean circuit is satisfiable, then any set of m input values that produces TRUE output is a proof that the circuit is satisfiable; 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, every problem in P is also in co-NP.

---

[1] This notion of efficiency was independently formalized by Alan Cobham in 1965, Jack Edmonds in 1965, and Michael Rabin in 1966, although similar notions were considered more than a decade earlier by Kurt Gödel, John Nash, and John von Neumann.

Perhaps the single most important unanswered question in theoretical computer science—if not all of computer science—if not all of **science**—is whether the complexity classes P and NP are actually different. Intuitively, it seems obvious to most people that P ≠ NP; the homeworks and exams in your algorithms and data structures classes have (I hope) convinced you that problems can be incredibly hard to solve, even when the solutions are simple in retrospect. It's completely obvious; *of course* solving problems from scratch is harder than verifying that a given solution is correct. We can reasonably accept—and most algorithm designers *do* accept—the statement "P ≠ NP" as a law of nature, similar to other laws of nature like Maxwell's equations, general relativity, and the sun rising tomorrow morning that are strongly supported by evidence, but have no mathematical proof.

But if we're being mathematically rigorous, we have to admit that nobody knows how to *prove* that that P ≠ NP. In fact, there has been little or no real progress toward a proof for decades.[2] The Clay Mathematics Institute lists P-versus-NP as the first of its seven Millennium Prize Problems, offering a \$1,000,000 reward for its solution. And yes, in fact, several people *have* lost their souls, or at least their sanity, attempting to solve this problem.

A more subtle but still open question is whether the complexity classes NP and co-NP are different. Even if we can verify every Yes answer quickly, there's no reason to believe 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 ≠ co-NP, but again, nobody knows how to prove it.
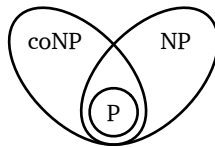

Figure 12.3. What we *think* the world looks like.

12.3 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:

> Π is NP-hard ⟺ If Π can be solved in polynomial time, then P=NP

---

[2]Perhaps the most significant progress has taken the form of *barrier* results, which imply that entire avenues of attack are doomed to fail. In a very real sense, not only do we have no idea how to prove P ≠ NP, but we can actually *prove* that we have no idea how to prove P ≠ NP!

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 every problem in NP.

Finally, a problem is ***NP-complete*** if it is both NP-hard and an element of NP (or "NP-easy"). Informally, NP-complete problems are the hardest problems in NP. A polynomial-time algorithm for even one NP-complete problem would immediately 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 (and therefore all) of them seems incredibly unlikely.

Calling a problem 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 I bet you're 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.[3] 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! Similarly, if a problem is NP-hard, no one in their right mind should believe it can be solved in polynomial time.
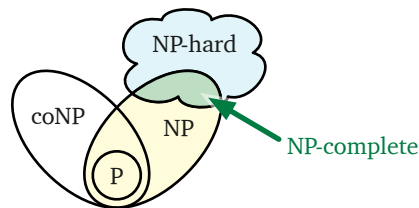

Figure 12.4. More of what we *think* the world looks like.

It is not immediately obvious that *any* problems are NP-hard. The following remarkable theorem was first published by Stephen Cook in 1971 and independently by Leonid Levin in 1973.[4]

---

[3]... The department chair shakes his head sadly and says, "Oh, come on, that just sounds like barking. Let *me* ask a question. Who was the greatest complexity theorist of all time?" The dog cocks his head, pauses for a few seconds, and then says "Karp!" After the chair chases them out of his office, the dog turns to its owner and says, "Maybe I should have said Impagliazzo?"

[4]Levin first reported his results at seminars in Moscow in 1971, while still a PhD student. News of Cook's result did not reach the Soviet Union until at least 1973, after Levin's announcement of his results had been published; in accordance with Stigler's Law, this result is often called "Cook's Theorem". Levin was denied his PhD at Moscow University for political reasons; he emigrated to the US in 1978 and earned a PhD at MIT a year later. Cook was denied tenure by the Berkeley

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

I won't even sketch a proof here, because I've been (deliberately) vague about the definitions.[5]

♥12.4   Formal Definitions (*HC SVNT DRACONES*)

Formally, the complexity classes P, NP, and co-NP are defined in terms of *languages* and *Turing machines*. A language is a set of strings over some finite alphabet Σ; without loss of generality, we can assume that $\Sigma = \{0, 1\}$. A Turing machine is a very restrictive type of computer—crudely, a finite-state machine with an unbounded memory tape—whose precise definition is surprisingly unimportant. P is the set of languages that can be decided in *P*olynomial time by a deterministic single-tape Turing machine. Similarly, NP is the set of all languages that can be decided in polynomial time by a nondeterministic Turing machine; NP is an abbreviation for *N*ondeterministic *P*olynomial-time.

The requirement of polynomial time is sufficiently crude that we do not have to specify the precise form of Turing machine (number of tapes, number of heads, number of tracks, size of the tape alphabet, and so on). In fact, any algorithm that runs on a random-access machine[6] in $T(n)$ time can be simulated by a single-tape, single-track, single-head Turing machine that runs in $O(T(n)^4)$ time. This simulation result allows us to argue formally about computational complexity in terms of standard high-level programming constructs like arrays and loops and recursion, instead of describing everything directly in terms of Turing machines.

Formally, a problem Π is NP-hard if and only if, for every language $\Pi' \in \text{NP}$, there is a polynomial-time ***Turing reduction*** from Π' to Π. A Turing reduction means a reduction that can be executed on a Turing machine; that is, a Turing machine M that can solve Π' using another Turing machine M' for Π as a black-box subroutine. Turing reductions are also called *oracle reductions*; polynomial-time Turing reductions are also called *Cook reductions*.

---

mathematics department in 1970, just one year before publishing his seminal paper; he (but not Levin) later won the Turing award for his proof.

[5]Interested readers find a proof in my lecture notes on nondeterministic Turing machines at http://algorithms.wtf, or in Boaz Barak's excellent *Introduction to Theoretical Computer Science*.

[6]Random-access machines are a model of computation that more faithfully models physical computers. A standard random-access machine has unbounded random-access memory, modeled as an unbounded array $M[0 .. \infty]$ where each address $M[i]$ holds a single w-bit integer, for some fixed integer w, and can read to or write from any memory addresses in constant time. RAM algorithms are formally written in assembly-like language, using instructions like **ADD** i, j, k (meaning "$M[i] \leftarrow M[j] + M[k]$"), **INDIR** i, j (meaning "$M[i] \leftarrow M[M[j]]$"), and **IF0GOTO** i, ℓ (meaning "if $M[i] = 0$, go to line ℓ"), but the precise instruction set is surprisingly irrelevant. By definition, each instruction executes in unit time. In practice, RAM algorithms can be faithfully described using higher-level pseudocode, as long as we're careful about arithmetic precision.

Researchers in complexity theory 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 $L' \subseteq \Sigma^*$ to another language $L \subseteq \Sigma^*$ is a function $f : \Sigma^* \to \Sigma^*$ such that $x \in L'$ if and only if $f(x) \in L$. Then we can define a *language L* to be NP-hard if and only if, for any language $L' \in$ NP, there is a many-one reduction from L' to L that can be computed in polynomial time.

Every Karp reduction "is" a Cook reduction, but not vice versa. Specifically, any Karp reduction from one decision problem Π to another decision Π' is equivalent to transforming the input to Π into the input for Π', invoking an oracle (that is, a subroutine) for Π', and then returning the answer verbatim. However, as far as we know, not every Cook reduction can be simulated by a Karp reduction.

Complexity theorists prefer Karp reductions primarily because NP is closed under Karp reductions, but is *not* closed under Cook reductions (unless NP=co-NP, which is considered unlikely). 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. One trivial example of such a problem is UNSAT: Given a boolean formula, is it *always false*? On the other hand, many-one reductions apply *only* to decision problems (or more formally, to languages); formally, no optimization or construction problem is Karp-NP-hard.

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 (as far as we know) 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.

Fortunately, none of these subtleties rear their ugly heads in practice—in particular, every reduction described in this chapter can be formalized as a logarithmic-space many-one reduction—so you can wake up now.

12.5   Reductions and SAT

To prove that any problem other than circuit satisfiability 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've already been doing reduction for years, even before starting this book, only you probably called them something else, like subroutines or utility functions or modular programming or using a calculator. 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, right next to your mom's birthday and the *actual* rules of Monopoly.[7]

> **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 efficient algorithm to solve a *different* problem, which you already know is hard, using an hypothetical efficient algorithm for *your* problem as a black-box subroutine. The essential logic is a proof by contradiction. The reduction implies that if your problem were easy, then the other problem would be easy, which it ain't. Equivalently, since you know the other problem is hard, the reduction implies that your problem must also be hard; your hypothetical efficient algorithm does not actually exist.

As a canonical example, consider the *formula satisfiability* problem, usually just called *S**AT***. The input to S**AT** is a boolean *formula* like

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

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

To prove that S**AT** 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 C**IRCUIT**S**AT**, so let's start there.

Let K be an arbitrary boolean circuit. We can transform (or more accurately, *transcribe*) K into a boolean formula Φ as follows. First, label each interior wire by a new variable y_j, and label the output wire with a new variable z. The formula Φ consists of a list of equations, one for each gate, separated by A**NDS**, followed by a final $\wedge z$. Figure 12.5 shows the resulting transcription for our example circuit.

Now we claim that the original circuit K is satisfiable ***if and only if*** the resulting formula Φ is satisfiable. Like every other "if and only if" statement, we prove this claim in two steps:

---

[7]If a player lands on an available property and declines (or is unable) to buy it, that property is immediately auctioned off to the highest bidder; the player who originally declined the property may bid, and bids may be arbitrarily higher or lower than the list price. Players in Jail can still buy and sell property, buy and sell houses and hotels, and collect rent. The game has 32 houses and 12 hotels; once they're gone, they're gone. In particular, if all houses are already on the board, you cannot downgrade a hotel to four houses; you must raze all the hotels in the group to the ground. Players can sell or exchange *undeveloped* properties with each other, but cannot sell property back to the bank; on the other hand, players can sell buildings to the bank (at half price), but cannot sell or exchange buildings with each other. All penalties are paid directly to the bank. A player landing on Free Parking does not win anything. A player landing on Go gets exactly $200, no more. Railroads are not magic transporters. Finally, Jeff *always* gets the boot. No, not the T-Rex or the penguin—the *boot*, dammit.
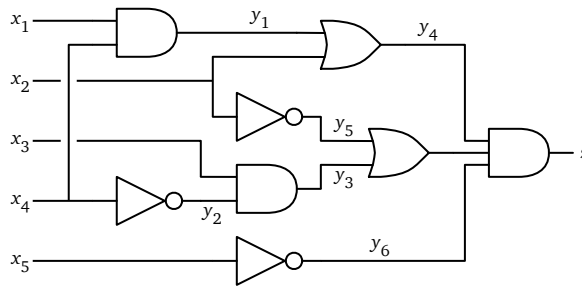
$$(y_1 = x_1 \land x_4) \land (y_2 = \overline{x_4}) \land (y_3 = x_3 \land y_2) \land (y_4 = y_1 \lor x_2) \land$$
$$(y_5 = \overline{x_2}) \land (y_6 = \overline{x_5}) \land (y_7 = y_3 \lor y_5) \land (z = y_4 \land y_7 \land y_6) \land z$$

Figure 12.5. Transcribing a boolean circuit as a boolean formula.

$\Rightarrow$ Given a set of inputs that satisfy the circuit K, we can derive a satisfying assignment for the formula Φ by computing the output of every gate in K.

$\Leftarrow$ Given a satisfying assignment for the formula Φ, we can obtain a satisfying input the circuit by simply ignoring the internal wire variables y_i and the output variable z.

The entire transformation from circuit to formula can be carried out in linear time. Moreover, the size of the resulting formula is at most a constant factor larger than any reasonable representation of the circuit.

Now suppose, for the sake of argument, there is an algorithm that can determine in polynomial time whether a given boolean formula is satisfiable. Then given any boolean circuit K, we can decide whether K is satisfiable by first transforming K into a boolean formula Φ as described above, and then asking our magical mystery SAT algorithm whether Φ is satisfiable, as suggested by the following cartoon. Each box represents an algorithm. The red box on the left is the transformation subroutine. The box on the right is the hypothetical magic SAT algorithm. It *must* be magic, because it has a *rainbow* on it.[8]
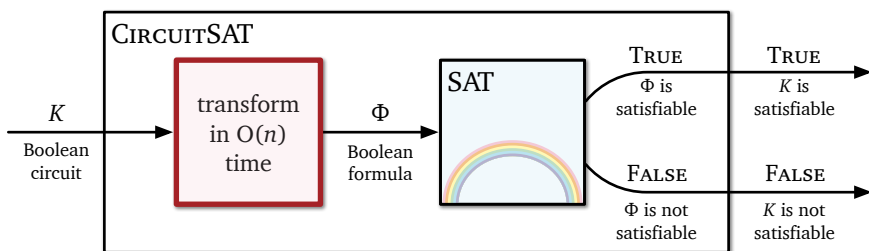
If you prefer magic pseudocode to magic boxes:

---

[8]Kay Erickson, personal communication, 2011. For those of you reading black-and-white printed copies: Yes, that round thing is a rainbow.

```
CircuitSat(K):
    transcribe K into a boolean formula Φ
    return Sat(Φ)        ⟪ ⋆★⋆★MAGIC★⋆★⋆ ⟫
```

Transcribing K into Φ requires only polynomial time (in fact, only *linear* time, but whatever), so the entire CircuitSat algorithm also runs in polynomial time.

$$T_{\text{CircuitSat}}(n) \leq O(n) + T_{\text{Sat}}(O(n))$$

We conclude that any polynomial-time algorithm for Sat would give us a polynomial-time algorithm for CircuitSat, which in turn would imply P=NP. So Sat is NP-hard!

12.6   3Sat (from CircuitSat)

A special case of Sat that is particularly useful in proving NP-hardness results is called *3CNF-Sat* or more often simply *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 the restriction of Sat 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 actually easier to start over from scratch, by reducing directly from CircuitSat.
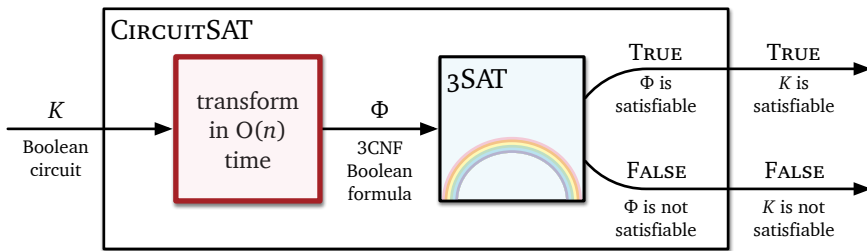

Figure 12.6. A polynomial-time reduction from CircuitSat to 3Sat.

Given an arbitrary boolean circuit K, we transform K into an equivalent 3CNF formula in several stages. Except for the very last stage, this reduction

was actually described by Grigorii Tseitin in 1966, five years before Cook and Levin reported their proofs of the Cook-Levin Theorem. (In the same 1966 paper, Tseitin described the problem we now call CNF-Sat, possibly for the first time.) As we describe each stage, we will also prove that stage is correct.

- *Make sure every AND and OR gate in K has exactly two inputs.* If any gate has $k > 2$ inputs, replace it with a binary tree of $k - 1$ binary gates. Call the resulting circuit K'. The circuits K and K' are logically equivalent circuits, so every satisfying input for K is a satisfying input for K' and vice versa.

- *Transcribe K' into a boolean formula Φ_1 with one clause per gate,* exactly as in our previous reduction to Sat. We already proved that every satisfying input for K' can be transformed into a satisfying assignment for Φ_1 and vice versa.

- *Replace each clause in Φ_1 with a CNF formula.* There are only three types of clauses in Φ_1, one for each type of gate in K':

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

Call the resulting CNF formula Φ_2. Because Φ_1 and Φ_2 are logically equivalent formulas, every satisfying assignment for Φ_1 is also a satisfying assignment for Φ_2, and vice versa.

- *Replace each clause in Φ_2 with a 3CNF formula.* Every clause in Φ_2 has *at most* three literals, but we need clauses with *exactly* three literals. To obtain a 3CNF formula, we expand each two-literal clause in Φ_2 into two three-literal clauses by introducing one new variable, and we expand the final one-literal clause in Φ_2 into four three-literal clauses by introducing two new variables.

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

Call the resulting 3CNF formula Φ_3. Every satisfying assignment for Φ_2 can be transformed into a satisfying assignment for Φ_3 by assigning *arbitrary* values to the new variables (x and y). Conversely, every satisfying assignment for Φ_3 can be transformed into a satisfying assignment for Φ_2 by ignoring the new variables.

For example, our example circuit is transformed into the following 3CNF formula; compare with Figure 12.5.

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

Yikes! At first glance, this formula might look a *lot* more complicated than the original circuit, but in fact, it's only larger by a constant factor. Specifically, the simplified circuit K' has at most twice as many wires as the original circuit K, each binary gate in K' is transformed into at most five clauses in Φ_3. Even if the formula size were a large *polynomial* function (like n^{374}) of the circuit size, we would still have a valid reduction.

Our reduction transforms an arbitrary boolean circuit K into a 3CNF formula Φ_3 in polynomial time (in fact, in linear time). Moreover, any satisfying input for the input circuit K can be transformed into a satisfying assignment for Φ_3, and any satisfying assignment for Φ_3 can be transformed into a satisfying input for K. In other words, K is satisfiable if and only if Φ_3 is satisfiable. Thus, if 3Sat can be solved in polynomial time, then CircuitSat can be solved in polynomial time, which implies that P = NP. We conclude that 3Sat is NP-hard.

12.7 Maximum Independent Set (from 3Sat)

For the next few problems we consider, the input is a simple, unweighted, undirected 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, which I'll abbreviate ***MaxIndSet***, asks for the size of the largest independent set in a given graph. I will prove that MaxIndSet is NP-hard using a reduction from 3Sat, as suggested by Figure 12.7.

Given an arbitrary 3CNF formula Φ, we construct a graph G as follows. Let k denote the number of clauses in Φ. The graph G contains exactly $3k$ vertices, one for each literal in Φ. Two vertices in G are connected by an edge if and only if either (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 graph shown in Figure 12.8.
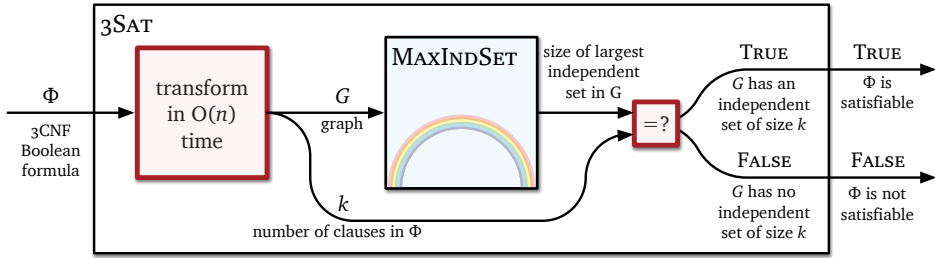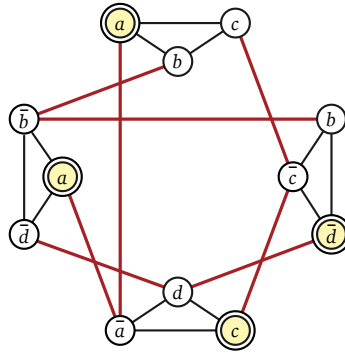
Figure 12.7. A polynomial-time reduction from 3Sat to MaxIndSet.

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

Figure 12.8. A graph derived from the satisfiable 3CNF formula with 4 clauses, and an independent set of size 4.

Each independent set in G contains at most one vertex from each clause triangle, because any two vertices in each triangle are connected. Thus, the largest independent set in G has size *at most k*. I claim that G contains an independent set of size *exactly k* if and only if the original formula Φ is satisfiable. As usual for "if and only if" statements, the proof consists of two parts.

$\Rightarrow$ Suppose Φ is satisfiable. Fix an arbitrary satisfying assignment. By definition, each clause in Φ contains at least one True literal. Thus, we can choose a subset S of k vertices in G that contains exactly one vertex per clause triangle, such that the corresponding k literals are all True. Because each triangle contains at most one vertex in S, no two vertices in S are connected by a triangle edge. Because every literal corresponding to a vertex in S is True, no two vertices in S are connected by a negation edge. We conclude that S is an independent set of size k in G.

$\Leftarrow$ On the other hand, suppose G contains an independent set S of size k. Each vertex in S must lie in a different clause triangle. Suppose we assign the value True to each literal in S; because contradictory literals are connected by edges, this assignment is consistent. There may be variables x such that neither x nor $\bar{x}$ corresponds to a vertex in S; we can set these variables to

any value we like. Because S contains one vertex in each clause triangle, each clause in Φ contains (at least) one TRUE literal. We conclude that Φ is satisfiable.

Transforming the 3CNF formula Φ into the graph G takes polynomial time, even if we do everything by brute force. Thus, if we could solve MAXINDSET in polynomial time, then we could also solve 3SAT in polynomial time, by transforming the input formula Φ into a graph G and comparing the size of the largest independent set in G with the number of clauses in Φ. But that would imply P=NP, which is ridiculous! We conclude that MAXINDSET is NP-hard.

12.8 The General Pattern

All NP-hardness proofs—and more generally, all polynomial-time reductions—follow the same general outline. To reduce problem X to problem Y in polynomial time, we need to do three things:

1. Describe a polynomial-time algorithm to transform an ***arbitrary*** instance of x of X into a *special* instance y of Y.
2. Prove that if x is a "good" instance of X, then y is a "good" instance of Y.
3. Prove that if y is a "good" instance of Y, then x is a "good" instance of X. (This is usually the part that causes the most trouble.)

Of course, *developing* a correct reduction doesn't mean handling these three tasks one at a time. *First* writing down an algorithm that *seems* to work and *then* trying prove that it *actually* works is rarely successful, especially in time-limited settings like exams. We *must* develop the algorithm, the "if" proof, and the "only if" proof simultaneously.

To quote the late great Ricky Jay:[9] This is an *acquired* skill.

One point that confuses many students is that the reduction algorithm only "works one way"—from X to Y—but the correctness proof needs to "work both ways". But the correctness proofs are not actually symmetric. The "if" proof needs to handle *arbitrary* instances of X, but the "only if" only needs to handle the *special* instances of Y produced by the reduction algorithm. Exploiting this asymmetry is the key to successfully designing correct reductions.

I find it useful to think in terms of transforming *certificates*—proofs that a given instance is "good"—along with the instances themselves. For example, a certificate for CIRCUITSAT is a set of inputs that turns on the light bulb; a certificate for SAT or 3SAT is a satisfying assignment; a certificate for MAXINDSET is a large independent set. To reduce X to Y, we actually need to design *three* algorithms, one for each of the following tasks:

---

[9]from his 1996 off-Broadway show *Ricky Jay and his 52 Assistants*

- Transform an arbitrary instance x of X into a special instance y of Y in polynomial time.
- Transform an arbitrary certificate for x into a certificate for y, and
- Transform an arbitrary certificate for y into a certificate for x.

The second and third tasks refer to the input and output of the first algorithm. The *certificate* transformation needs to be reversible, not the *instance* transformation. We never have to transform instances of Y, and we don't need to think about *arbitrary* instances of Y at all. Only the first algorithm needs to run in polynomial time (although in practice, the second and third algorithms are almost always simpler than the first).

For example, our reduction from CircuitSat to 3Sat consists of three algorithms:

- The first transforms an *arbitrary* boolean circuit K into a special 3CNF boolean formula Φ_3, in polynomial time. (Encode each wire as a variable and each gate as a sub-formula, and then expand each sub-formula into 3CNF.)
- The second transforms an *arbitrary* satisfying input for K into a satisfying assignment for Φ_3. (Trace the input through the circuit, transfer values from each wire to the corresponding variable, and give any additional variables arbitrary values.)
- The third transforms an *arbitrary* satisfying assignment for Φ_3 into a satisfying input for K. (Transfer values from each wire variable in Φ_3 to the corresponding wire in K.)

The reduction works because the first algorithm *encodes* any boolean circuit K into a highly structured 3CNF formula Φ_3. The specific structure of Φ_3 restricts *how* it can be satisfied; every satisfying assignment for Φ_3 must "come from" some satisfying input for K. We don't have to think about arbitrary 3CNF formulas at all.

Similarly, our reduction from 3Sat to MaxIndSet consists of three algorithms:

- The first transforms an *arbitrary* 3CNF formula Φ into a special graph G and a specific integer k, in polynomial time.
- The second transforms an *arbitrary* satisfying assignment for Φ into an independent set in G of size k.
- The third transforms an *arbitrary* independent set in G of size k into a satisfying assignment for Φ.

Again, our first transformation *encodes* the input formula Φ into a highly structured graph G and a specific integer k. The structure of G ensures that every independent set of size k "comes from" a satisfying assignment for Φ. We don't consider arbitrary graphs or arbitrary independent set sizes at all.

12.9 Clique and Vertex Cover (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 MAXCLIQUE problem asks for the number of nodes in its largest complete subgraph in a given graph. A **vertex cover** of a graph is a set of vertices that touches every edge in the graph. The MINVERTEXCOVER problem asks for the size of the smallest vertex cover in a given graph.
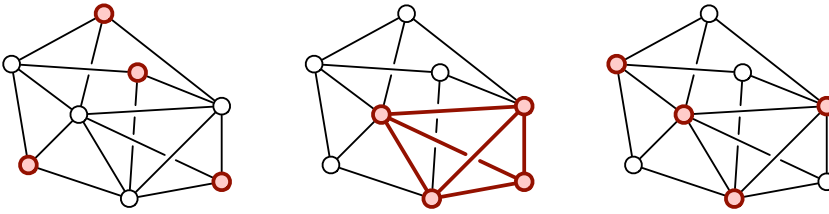

Figure 12.9. A graph whose largest independent set, largest clique, and smallest vertex cover all have size 4.

We can prove that MAXCLIQUE is NP-hard using the following easy reduction from MAXINDSET. Any graph G has an *edge-complement* $\overline{G}$ with the same vertices, but with exactly the opposite set of edges—uv is an edge in $\overline{G}$ if and only if uv 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, the largest independent in G has the same vertices (and thus the same size) as the largest clique in the complement of G.
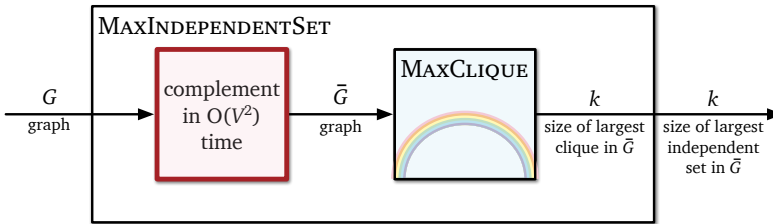

Figure 12.10. An easy reduction from MAXINDSET to MAXCLIQUE.

The proof that MINVERTEXCOVER is NP-hard is even simpler, because it relies on the following easy observation: I is an independent set in a graph $G = (V, E)$ if and only if its complement $V \setminus I$ is a vertex cover of the same graph G. Thus, the *largest* independent set in any graph is the complement of the *smallest* vertex cover of the same graph! Thus, if the smallest vertex cover in an n-vertex graph has size k, then the largest independent set has size $n - k$.
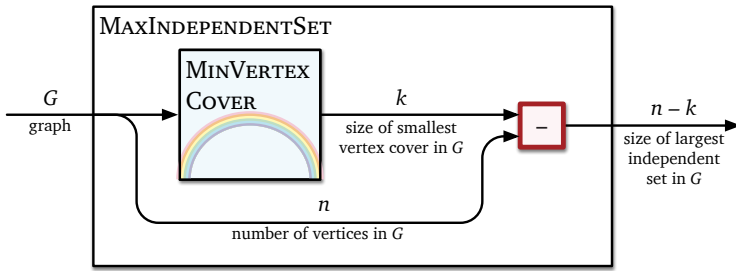
Figure 12.11. An even easier reductions from MaxIndSet to MinVertexCover.

12.10 Graph Coloring (from 3Sat)

A ***proper k-coloring*** of a graph $G = (V, E)$ is a function $C \colon V \to \{1, 2, \ldots, k\}$ that assigns one of k "colors" to each vertex, so that every edge has two different colors at its endpoints. (The "colors" are really arbitrary labels, which for simplicity we represent by small positive integers, rather than electromagnetic frequencies, CMYK vectors, or Pantone numbers, for example.) The graph coloring problem asks for the smallest possible number of colors in a legal coloring of a given graph.

To prove that graph coloring is NP-hard, it suffices to consider the decision problem ***3Color***: Given a graph, does it have a proper 3-coloring? We prove 3Color is NP-hard using a reduction from 3Sat. (Why 3Sat? Because it has a 3 in it. You probably think I'm joking, but I'm not.) Given a 3CNF formula Φ, we construct a graph G that is 3-colorable if and only if Φ is satisfiable, as suggested by the usual diagram.
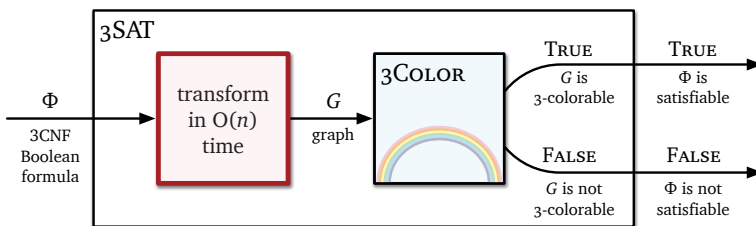
We describe the reduction using a standard strategy of decomposing the output graph G into ***gadgets***, subgraphs that enforce various semantics of the input formula Φ in the language of graph coloring. Decomposing reductions into separate gadgets is not only helpful for understanding existing reductions and proving them correct, but for designing new NP-hardness reductions.[10]

---

[10]Our reduction from CircuitSat to Sat encoding each gate in the input circuit as a clause in the output formula; these clauses are "gate gadgets". Similarly, our reduction from 3Sat to MaxIndSet used two type of gadgets: "clause gadgets" (triangles) and "variable gadgets" (edges between contradicting literals).

Our formula-to-graph reduction uses three types of gadgets:

- There is a single *truth gadget*: 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, we mean that it has the same color as vertex T.

- For each variable a, the graph contains a *variable gadget*, which is a triangle joining two new nodes labeled a and $\bar{a}$ to node X in the truth gadget. Node a must be colored either TRUE or FALSE, and therefore node $\bar{a}$ must be colored either FALSE or TRUE, respectively.
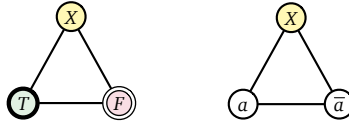

Figure 12.12. The truth gadget and a variable gadget for a.

- Finally, for each clause in Φ, the graph contains a *clause gadget*. Each clause gadget joins three literal nodes (from the corresponding variable gadgets) to node T (from the truth gadget) using five new unlabeled nodes and ten edges, as shown below.
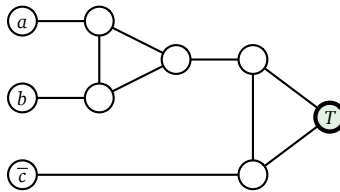

Figure 12.13. A clause gadget for $(a \lor b \lor \bar{c})$.

In effect, each triangle in the clause gadget behaves like a "majority gate". In any valid 3-coloring, if the two vertices to the left of the triangle have the same color, the rightmost vertex of the triangle must have the same color; on the other hand, if the two left vertices have different colors, the color of the right vertex can be chosen arbitrarily. See Figure 12.14.

It follows that there is no valid 3-coloring of a clause gadget where all three literal nodes are colored FALSE. On the other hand, any coloring of the literal nodes with more than one color can be extended to a valid 3-coloring of the clause gadget. The variable gadgets force each literal node to be colored either TRUE or FALSE; thus, in any valid 3-coloring of the clause gadget, at least one literal node is colored TRUE.

The final graph G contains exactly *one* node T, exactly *one* node F, and exactly *two* nodes a and $\bar{a}$ for each variable. For example, Figure 12.15 shows the graph
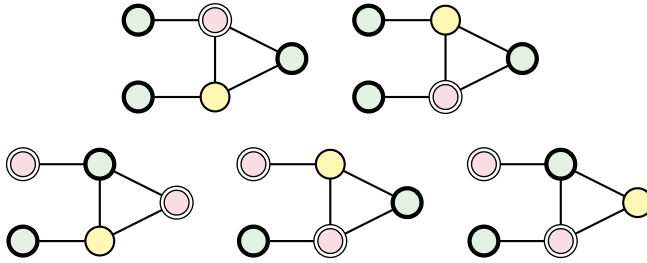
Figure 12.14. All valid 3-colorings of a "half-gadget", up to permutations of the colors

that results from the same 3CNF formula $(a \lor b \lor c) \land (b \lor \bar{c} \lor \bar{d}) \land (\bar{a} \lor c \lor d) \land (a \lor \bar{b} \lor \bar{d})$ that we previously used to illustrate the MaxIndSet reduction in Figure 12.8. The 3-coloring is one of several that correspond to the satisfying assignment $a = c = \text{True}$, $b = d = \text{False}$.
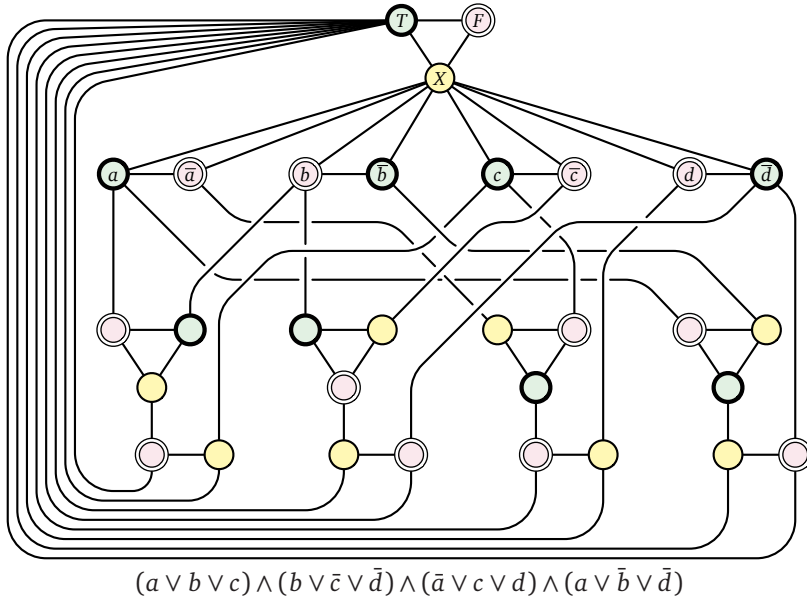

$$(a \lor b \lor c) \land (b \lor \bar{c} \lor \bar{d}) \land (\bar{a} \lor c \lor d) \land (a \lor \bar{b} \lor \bar{d})$$

Figure 12.15. The 3-colorable graph derived from a satisfiable 3CNF formula.

We've already done most of the work for a proof of correctness. If the formula is satisfiable, then we can color the literal nodes according to any satisfying assignment, and then (because each clause is satisfied) extend the coloring across every clause gadget. On the other hand, 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.

Because 3Color is a special case of the more general graph coloring problem— What is the minimum number of colors?—the more general optimization problem is also NP-hard.

12.11 Hamiltonian Cycle

A *Hamiltonian cycle* in a graph is a cycle that visits every vertex exactly once. (This is very different from an *Euler circuit*, which is a closed *walk* that traverses every *edge* exactly once; Euler circuits are easy to find and construct in linear time using depth-first search.) Here we consider two different proofs that the Hamiltonian cycle problem in *directed* graphs is NP-hard.

From Vertex Cover

Our first NP-hardness proof reduces from the decision version of the vertex cover problem. Given an *undirected* graph G and an integer k, we construct a *directed* graph H, such that H has a Hamiltonian cycle if and only if G has a vertex cover of size k. As in our previous reductions, the output graph H is composed of several gadgets, each corresponding to certain features of the inputs G and k.
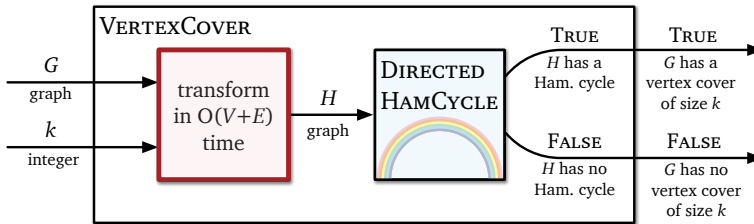
- For each undirected edge uv in G, the directed graph H contains an *edge gadget* consisting of four vertices (u, v, in), (u, v, out), (v, u, in), (v, u, out) and six directed edges

  $(u, v, \text{in}) \rightarrow (u, v, \text{out}) \qquad (u, v, \text{in}) \rightarrow (v, u, \text{in}) \qquad (v, u, \text{in}) \rightarrow (u, v, \text{in})$

  $(v, u, \text{in}) \rightarrow (v, u, \text{out}) \qquad (u, v, \text{out}) \rightarrow (v, u, \text{out}) \qquad (v, u, \text{out}) \rightarrow (u, v, \text{out})$

  as shown in Figure 12.16. Each "in" vertex has an additional incoming edge, and each "out" vertex has an additional outgoing edge. Any Hamiltonian cycle in H must pass through an edge gadget in one of three ways—either straight through on both sides, or with a detour from one side to the other and back. Eventually, these options will correspond to both u and v, only u, or only v belonging to some vertex cover.

- For each vertex u in G, all the edge gadgets for incident edges uv are connected in H into a single directed path, which we call a *vertex chain*. Specifically, suppose vertex u has d neighbors $v_1, v_2, \ldots, v_d$. Then H has $d - 1$ additional edges $(u, v_i, \text{out}) \rightarrow (u, v_{i+1}, \text{in})$ for each i from 1 to $d - 1$.

- Finally, H also contains k *cover vertices* $x_1, x_2, \ldots, x_k$. Each cover vertex has a directed edge to the first vertex in each vertex chain, and a directed edge from the last vertex in each vertex chain.
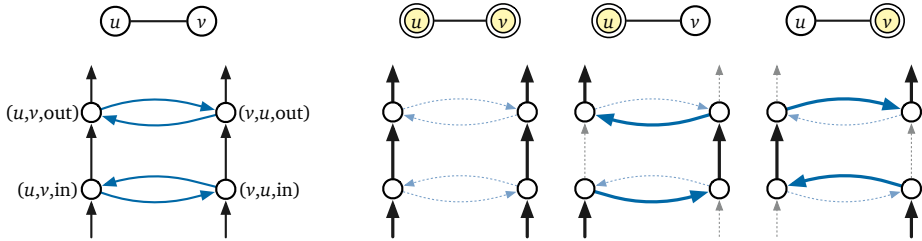
Figure 12.16. An edge gadget and its only possible intersections with a Hamiltonian cycle.

Figure 12.17 shows a complete example of our transformation; each double-arrowed blue segment represents a pair of directed edges.
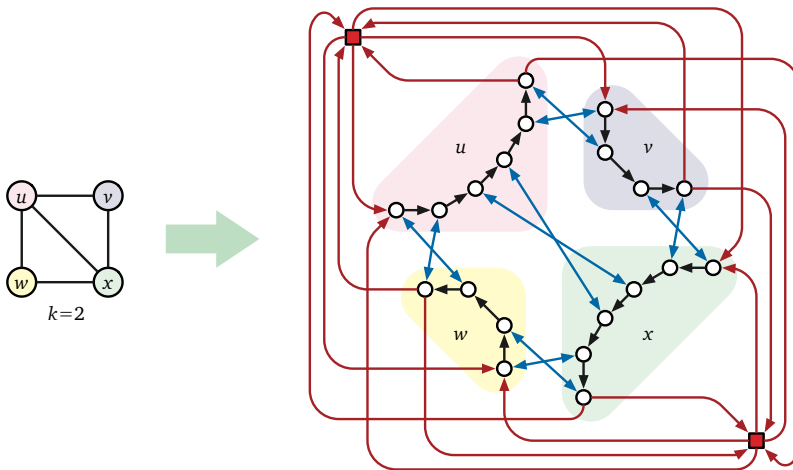

Figure 12.17. Example of our reduction from VERTEXCOVER to DIRECTEDHAMCYCLE.

As usual, we prove our reduction is correct in two stages.

$\Rightarrow$ First, suppose $C = \{u_1, u_2, \ldots, u_k\}$ is a vertex cover of G of size k. We can construct a Hamiltonian cycle in H that "encodes" C as follows. For each index i from 1 to k, we traverse a path from cover vertex x_i, through the vertex chain for u_i, to cover vertex x_{i+1} (or cover vertex x_1 if $i = k$). As we traverse the chain for each vertex u_i, we determine how to proceed from each node (u_i, v, in) as follows:

  – If $v \in C$, follow the edge $(u_i, v, \text{in}) \to (u_i, v, \text{out})$.
  – If $v \notin C$, detour through $(u_i, v, \text{in}) \to (v, u_i, \text{in}) \to (v, u_i, \text{out}) \to (u_i, v, \text{out})$.

Thus, for each edge uv of G, the Hamiltonian cycle visits (u, v, in) and (u, v, out) as part of u's vertex chain if $u \in C$ and as part of v's vertex chain otherwise. See Figure 12.18.

$\Leftarrow$ On the other hand, suppose H contains a Hamiltonian cycle C. This cycle must contain an edge from each cover vertex to the start of some
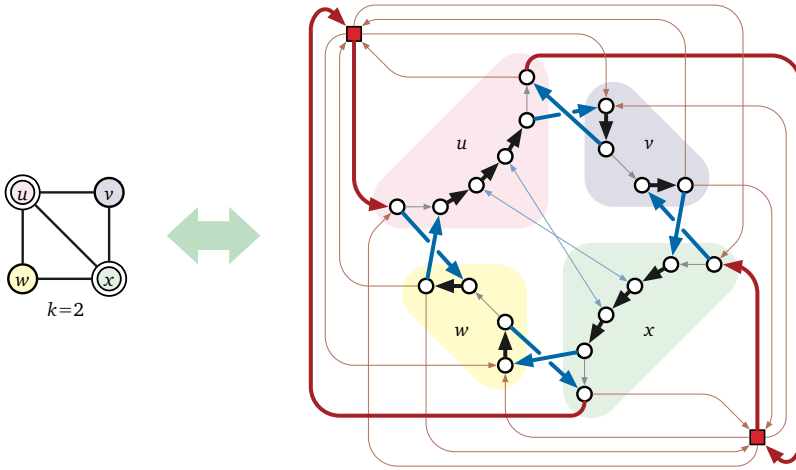
Figure 12.18. Every vertex cover of size k in G corresponds to a Hamiltonian cycle in H and vice versa.

vertex chain. Our case analysis of edge gadgets inductively implies that after C enters the vertex chain for some vertex u, it must traverse the entire vertex chain. Specifically, at each vertex (u, v, in), the cycle must contain either the single edge $(u, v, \text{in}) \rightarrow (u, v, \text{out})$ or the detour path $(u, v, \text{in}) \rightarrow (v, u, \text{in}) \rightarrow (v, u, \text{out}) \rightarrow (u, v, \text{out})$, followed by an edge to the next edge gadget in u's vertex chain, or to a cover vertex if this is the last edge gadget in u's vertex chain. In particular, if C contains the detour edge $(u, v, \text{in}) \rightarrow (v, u, \text{in})$, it cannot contain edges between any cover vertex and v's vertex chain. It follows that C traverses exactly k vertex chains. Moreover, these vertex chains describe a vertex cover of the original graph G, because C visits the vertex (u, v, in) for every edge uv in G.

We conclude that G has a vertex cover of size k if and only if H contains a Hamiltonian cycle. The transformation from G to H takes at most $O(V^2)$ time; it follows that the directed Hamiltonian cycle problem is NP-hard.

From 3Sat

We can also prove that the directed Hamiltonian cycle problem is NP-hard by reducing directly from 3Sat. Given an arbitrary 3CNF formula Φ with n variables $x_1, x_2, \ldots, x_n$ and k clauses $c_1, c_2, \ldots, c_k$, we construct a directed graph H that contains a Hamiltonian cycle if and only if Φ is satisfiable, as follows.

For each variable x_i, we construct a *variable gadget*, which consists of a doubly-linked list of $2k$ vertices $(i, 0), (i, 1), \ldots, (i, 2k)$, connected by edges $(i, j-1) \rightarrow (i, j)$ and $(i, j) \rightarrow (i, j-1)$ for each index j. We connect the first and last nodes in each adjacent pair of variable gadgets by adding edges

$$(i, 0) \rightarrow (i+1, 0) \quad (i, 2k) \rightarrow (i+1, 0) \quad (i, 0) \rightarrow (i+1, 2k) \quad (i, 2k) \rightarrow (i+1, 2k)$$

for each index i; we also connect the endpoints of the first and last variable gadgets with the edges

$$(n,0)\to(1,0) \qquad (n,2k)\to(1,0) \qquad (n,0)\to(1,2k) \qquad (n,2k)\to(1,2k).$$

The resulting graph G has exactly 2^n Hamiltonian cycles, one for each assignment of boolean values to the n variables of Φ. Specifically, for each i, we traverse the ith variable gadget from left to right if $x_i = \text{TRUE}$ and right to left if $x_i = \text{FALSE}$. See Figure 12.19.
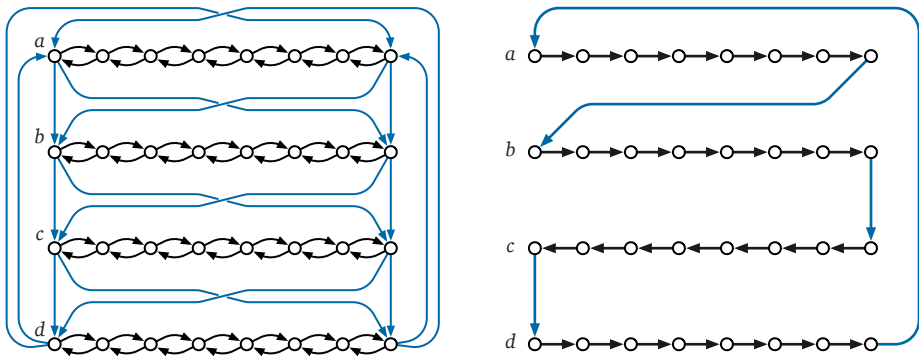

Figure 12.19. Left: Variable gadgets and connectors in G, for any formula with 4 variables and 4 clauses. Right: The Hamiltonian cycle in G corresponding to the assignment $a = b = d = \text{TRUE}$ and $c = \text{FALSE}$

Now we extend G to a larger graph H by adding a *clause vertex* $[j]$ for each clause c_j, connected to the variable gadgets by six edges, as shown in Figure 12.20. For each positive literal x_i in c_j, we add the edges $(i, 2j-1)\to[j]\to(i, 2j)$, and for each negative literal $\bar{x}_i$ in c_j, we add the edges $(i, 2j)\to[j]\to(i, 2j-1)$. The connections to the clause vertices guarantee that a Hamiltonian cycle in G can be extended to a Hamiltonian cycle in H if and only if the corresponding variable assignment satisfies Φ. Exhaustive case analysis now implies that H has a Hamiltonian cycle if and only if Φ is satisfiable.

Transforming the formula Φ into the graph H takes $O(kn)$ time, which is at most quadratic in the total length of the formula; we conclude that the directed Hamiltonian cycle problem is NP-hard.

Variants and Extensions

Trivial modifications of the previous reductions imply that the Hamiltonian *path* problem in directed graphs is also NP-hard. A Hamiltonian path in a graph G is of course a simple path that visits every vertex of G exactly once. In fact, there are simple polynomial-time reductions from the Hamiltonian cycle problem to the Hamiltonian path problem and vice versa. I'll leave the details of these reductions as exercises.
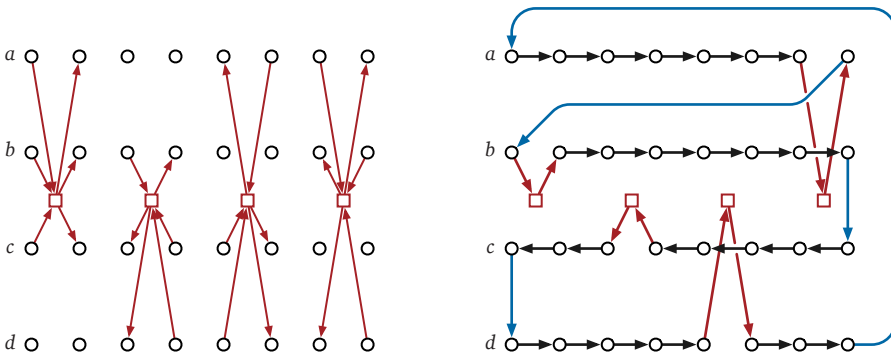
Figure 12.20. Left: Clause gadgets for the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$. Right: A hamiltonian cycle in H corresponding to the satisfying assignment $a = b = d = \text{TRUE}$ and $c = \text{FALSE}$.

Both of the previous reductions deal with directed graphs, but the corresponding question in undirected graph is also NP-hard. In fact, there is a relatively simple reduction from the directed Hamiltonian cycle/path problem to the undirected Hamiltonian cycle/path problem; again, I'll leave the details of this reduction as an exercise.

Finally, the infamous *traveling salesman problem* asks to find the shortest Hamiltonian cycle (or path) in a graph with weighted edges. Since finding the shortest cycle/path is obviously harder than determining if a cycle/path exists at all—Consider a graph where every edge has weight 1!—TRAVELINGSALESMAN is also NP-hard.

12.12 Subset Sum (from Vertex Cover)

The next problem that we prove NP-hard is the SUBSETSUM problem considered in Chapter **??**: Given a set X of positive integers and an integer T, determine whether X has a subset whose elements sum to T.

We once again reduce from VERTEXCOVER. Given a graph G and an integer k, we need to compute a set X of positive integers 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 only two types of "gadgets", which are *integers* representing the vertices and edges in G.

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

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

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 $(E+1)$-digit number written

in base 4. The Eth digit is 1 if the integer represents a vertex, and 0 otherwise; and for each $i < E$, 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

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

Now let's prove that the reduction is correct.

$\Rightarrow$ First, suppose G has a vertex cover C of size k. Consider the subset

$$X' := \{a_v \mid v \in C\} \cup \{b_i \mid \text{edge } i \text{ has exactly one endpoint in } C\}$$

The sum of the elements of X', written in base 4, has most significant digit k and all other digits equal to 2. Thus, the elements of X' sum to exactly T.

$\Leftarrow$ 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 E 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^E, and $\lfloor T/4^E \rfloor = k$, 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 $G = (V, E)$ where $V = \{u, v, w, x\}$ and $E = \{uv, uw, vw, vx, wx\}$, our set X might contain the following base-4 integers:

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

If we are looking for a vertex cover of size $k = 2$, 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. We've already proved that VERTEXCOVER is NP-hard, so it follows that SUBSETSUM is NP-hard.

Caveat Reductor!

One subtle point must be emphasized here. 3oo-something pages ago, back in Chapter **??**, we developed a dynamic programming algorithm to solve SubsetSum in $O(nT)$ time. Isn't this a polynomial-time algorithm? Didn't we just prove that P=NP? Hey, where's my million dollars?!

Alas, life is not so simple. True, the running time is a polynomial function of the variables n and T, but to qualify as a true polynomial-time algorithm, the running time must be a polynomial function of the *input size*—the number of *bits* required to represent 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 a value of T that is exponentially larger than the size of our original input graph, which forces our dynamic programming algorithm to run in exponential time.

Algorithms like this are said to run in **pseudo-polynomial time**, and any NP-hard problem with such an algorithm is called **weakly NP-hard**. Equivalently, a weakly NP-hard problem is a problem that can be solved in polynomial time when all input numbers are represented in *unary* (as a sum of 1s), but becomes NP-hard when all input numbers are represented in *binary*. If a problem is NP-hard even when all the input numbers are represented in unary, we say that the problem is **strongly NP-hard**. A good example of a *strongly* NP-hard problem is TravelingSalesman, which remains NP-hard even if the input graph is complete and all edge weights are equal to 1 or 2.

12.13 Other Useful NP-hard Problems

Literally thousands of problems have been proved to be NP-hard. Here I will list a few NP-hard problems that are useful in deriving reductions.[11] I won't describe the NP-hardness proofs for these problems in detail, but you can find most of them in Garey and Johnson's classic *Scary Black Book of NP-Completeness.*[12] All of the problems I've discussed so far, and most of the problems in the following list, were first proved NP-hard in a single landmark 1972 paper by Richard Karp.[13]

- **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

---

[11]As someday it may happen a reduction must be found, I've got a little list. I've got a little list. Of some Herculean problems we can use to lower bound. Fast solutions don't exist. Our proofs they can assist.

[12]Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., 1979. And yes, it really is black.

[13]Later performed off-Broadway as *Richard Karp and his 21 Assistants*, for which Karp won a well-deserved ~~Tony~~ Turing award.

output TRUE? This problem can be proved NP-hard by reduction from the general circuit satisfiability problem, by replacing each crossing with a small assemblage of gates.

- **1-IN-3SAT:** Given a 3CNF formula, is there an assignment of values to the variables so that each clause contains *exactly* one TRUE literal? This problem can be proved NP-hard by reduction from the usual 3SAT.

- **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 problem 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 problem can be proved NP-hard by reduction from PLANARCIRCUITSAT.[14]

- **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 problem can be proved NP-hard by a reduction from 3SAT, because it has a 3 in it.

- **PARTITION:** Given a set S of n integers, are there subsets A and B such that $A \cup B = S$, $A \cap B = \emptyset$, and

$$\sum_{a \in A} a = \sum_{b \in B} b?$$

  This problem 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 three-element subsets, such that every subset has exactly the same sum? Despite the similar names, this problem is *very* different from PARTITION; sorry, I didn't make up the names. This problem can be proved NP-hard by reduction from X3M, because it has a 3 in it. Unlike PARTITION, the 3PARTITION problem is *strongly* NP-hard; it remains NP-hard even if every input number is at most n^3.

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

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

---

[14]Surprisingly, PLANARNOTALLEQUAL3SAT is solvable in polynomial time!

- **LONGESTPATH:** Given a non-negatively weighted graph G (either directed or undirected) 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 problem is a generalization of the corresponding Hamiltonian path problem. Of course, the corresponding *shortest* path problem can be solved 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 the minimum spanning tree; if exactly two vertices are marked, the minimum Steiner tree is the shortest path between them. This problem can be proved NP-hard by reduction from VERTEXCOVER.

- **MAX2SAT:** Given a Boolean formula in conjunctive normal form, with exactly *two* literals per clause, find a variable assignment that maximizes the number of clauses with at least one TRUE literal. This problem can be proved NP-hard by reduction from 3SAT (yes, even though it doesn't have a 3 in it). The simpler decision problem 2SAT, which asks if there is an assignment that satisfies *every* clause, can actually be solved in polynomial time.

- **MAXCUT:** Given an undirected graph $G = (V, E)$, find a subset $S \subset V$ that maximizes the number of edges with exactly one endpoint in S. Equivalently, find the largest bipartite subgraph of G. This problem can be proved NP-hard by reduction from MAX2SAT.

In addition to these dry but useful problems, 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!) Here are some examples you may find familiar:

- Minesweeper (from CIRCUITSAT)[15]
- Sudoku (utlimately from 3SAT)[16]
- Tetris (from 3PARTITION)[17]
- Klondike, aka "Solitaire" (from 3SAT)[18]

---

[15]Richard Kaye. Minesweeper is NP-complete. *Mathematical Intelligencer* 22(2):9–15, 2000. But see also: Allan Scott, Ulrike Stege, and Iris van Rooij. Minesweeper may not be NP-complete but is hard nonetheless. *Mathematical Intelligencer* 33(4):5–17, 2011.

[16]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.

[17]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.

[18]Luc Longpré and Pierre McKenzie. The complexity of Solitaire. *Proceedings of the 32nd International Mathematical Foundations of Computer Science*, 182–193, 2007.

- Pac-Man (from HAMILTONIANCYCLE)[19]
- Super Mario Brothers (from 3SAT)[20]
- Candy Crush Saga (from a variant of 3SAT)[21]
- Threes/2048 (from 3SAT, of course)[22]
- Trainyard (from DOMINATINGSET; see Exercise 26)[23]
- Shortest $n \times n \times n$ Rubik's cube solution (from 3SAT via a special case of PLANARUNDIRECTEDHAMCYCLE).[24]
- Cookie Clicker (from PARTITION or 3PARTITION)[25]

This list is necessarily incomplete, thanks to a limited footnote budget.[26] As of June 2019, nobody has published a proof that a generalization of Ultimate Paperclips, Line Rider, Twister, or Cards Against Humanity is NP-hard, but I'm sure it's only a matter of time.

12.14 Choosing the Right Problem

One of the most difficult steps in proving that a problem is NP-hard is choosing the best problem to reduce from. The Cook-Levin Theorem implies that if there is a reduction from *any* NP-hard problem to problem X, then there is a reduction from *every* NP-complete problem to problem X, but some problems are easier to work with than others. There's no systematic method for choosing the right problem, but here are a few useful rules of thumb.

- If the problem asks how to assign bits to objects, or to choose a subset of objects, or to partition objects into two different subsets, try reducing from some version of SAT or PARTITION.

[19]Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014. http://giovanniviglietta.com/papers/gaming2.pdf.

[20]Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games Are (computationally) hard. *Theoretical Computer Science* 586:135–160, 2015. http://arxiv.org/abs/1203.1895.

[21]Luciano Gualà, Stefano Leucci, Emanuele Natale. Bejeweled, Candy Crush and other match-three games are (NP-)hard. *Proc. 2014 IEEE Conference on Computational Intelligence and Games*, 2014. http://arxiv.org/abs/1403.5830.

[22]Stefan Langerman and Yushi Uno. Threes!, Fives, 1024!, and 2048 are Hard. *Proc. 8th International Conference on Fun with Algorithms*, 2016. https://arxiv.org/abs/1505.04274.

[23]Matteo Almanza, Stefano Leucci, and Alessandro Panconesi. Trainyard is NP-Hard. *Proc. 8th International Conference on Fun with Algorithms*, 2016. https://arxiv.org/abs/1603.00928.

[24]Erik D. Demaine, Sarah Eisenstat, and Mikhail Rudoy. Solving the Rubik's Cube optimally is NP-complete. *Proc. 35th Symposium on Theoretical Aspects of Computer Science*, 2018. https://arxiv.org/abs/1706.06708.

[25]Erik D. Demaine, Hiro Ito, Stefan Langerman, Jayson Lynch, Mikhail Rudoy, and Kai Xiao. Cookie Clicker. Preprint, August 2018. https://arxiv.org/abs/1808.07540.

[26]See https://xkcd.com/1208/

- If the problem asks how to assign labels to objects from a small fixed set, or to partition objects into a small number of subsets, try reducing from kCOLOR or even 3COLOR.

- If the problem asks to arrange a set of objects in a particular order, try reducing from DIRECTEDHAMCYCLE or DIRECTEDHAMPATH or TRAVELING-SALESMAN.

- If the problem asks to find a *small* subset satisfying some constraints, try reducing from MINVERTEXCOVER.

- If the problem asks to find a *large* subset satisfying some constraints, try reducing from MAXINDSET or MAXCLIQUE or MAX2SAT.

- If the problem asks to partition objects into a large number of small subsets, try reducing from 3PARTITION.

- If the number 3 appears naturally in the problem, try 3SAT or 3COLOR or X3M or 3PARTITION. (No, this is not a joke.)

- If all else fails, try 3SAT or even CIRCUITSAT!

I do not recommend trying to reduce from TETRIS, SUPERMARIOBROS, or TRAINYARD. You really want to choose a starting problem that is as simple as possible, while still capturing *some* feature of your problem that makes it difficult to solve.

12.15 A Frivolous Real-World Example

Draughts is a family of board games that have been played for thousands of years. Most Americans are familiar with the version called *checkers* or *English draughts*, but the most common variant worldwide, known as **international draughts** or **Polish draughts**, originated in the Netherlands in the 16th century. For a complete set of rules, the reader should consult Wikipedia; here a few important differences from the Anglo-American game:

- **Flying kings:** As in checkers, a piece that ends a move in the row closest to the opponent becomes a *king* and gains the ability to move backward. Unlike in checkers, however, a king in international draughts can move any distance along a diagonal line in a single turn, as long as the intermediate squares are empty or contain exactly one opposing piece (which is captured).

- **Forced maximum capture:** In each turn, the moving player must capture as many opposing pieces as possible. This is distinct from the forced-capture rule in checkers, which requires only that each player must capture if possible, and that a capturing move ends only when the moving piece cannot capture further. In other words, checkers requires capturing a *locally maximal* set of opposing pieces on each turn; whereas, international draughts requires a *globally maximum* capture.

- **Capture subtleties:** As in checkers, captured pieces are removed from the board only at the end of the turn. Any piece can be captured at most once. Thus, when an opposing piece is jumped, that piece remains on the board *but cannot be jumped again* until the end of the turn.

For example, in the first position shown below, each circle represents a piece, and doubled circles represent kings. Black *must* make the first indicated move, capturing five white pieces, because it is not possible to capture more than five pieces, and there is no other move that captures five. Black cannot extend his capture further, either northeast or southeast, because the captured White pieces remain on the board until his turn is over. Then White *must* make the second indicated move, thereby winning the game.
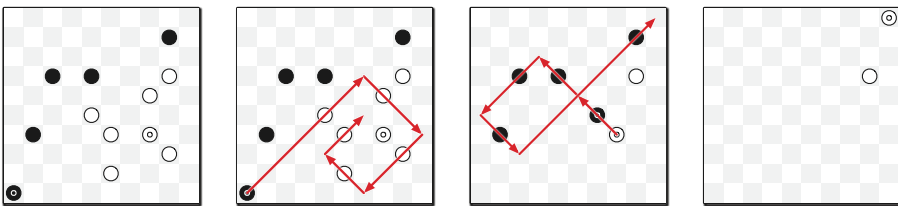

Figure 12.21. Two forced(!) moves in international draughts; doubled circles are kings.

The actual game, which is played on a 10×10 board with 20 pieces of each color, is computationally trivial; we can precompute the optimal move for both players in every possible board configuration and hard-code the results into a lookup table of constant size. Sure, it's a *big* constant, but it's still a constant!

But consider the natural generalization of international draughts to an $n \times n$ board. In this setting, ***finding a legal move is actually NP-hard!*** The following reduction from the Hamiltonian cycle problem in directed graphs was discovered by Bob Hearn in 2010.[27] In most two-player games, finding the *best* move is NP-hard (or worse). This is the only example I know of a game—and moreover a real game played by millions of people for centuries—where *merely following the rules* is NP-hard!

Given an undirected graph G with n vertices, we construct a board configuration for international draughts, such that White can capture a certain number of black pieces in a single move if and only if G has a Hamiltonian cycle. We treat G as a directed graph, with two arcs $u \rightarrow v$ and $v \rightarrow u$ in place of each undirected edge uv. Number the vertices arbitrarily from 1 to n. The final draughts configuration has several gadgets:

- The vertices of G are represented by rabbit-shaped *vertex gadgets*, which are evenly spaced along a horizontal line. Each arc $i \rightarrow j$ is represented by a path of two diagonal line segments from the "left ear" of vertex gadget i to

---

[27]See Theoretical Computer Science Stack Exchange: http://cstheory.stackexchange.com/a/1999/111.

the "right ear" of vertex gadget j. The path for arc $i{\rightarrow}j$ is located above the vertex gadgets if $i < j$, and below the vertex gadgets if $i > j$.
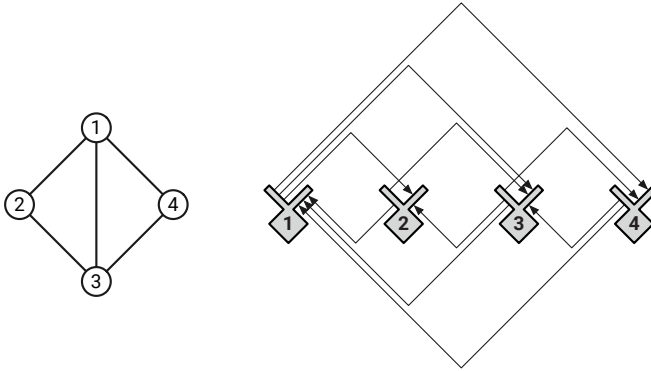

Figure 12.22. A high-level overview of the reduction from Hamiltonian cycle to international draughts.

- The bulk of each vertex gadget is a diamond-shaped region called a *vault*. The walls of the vault are composed of two solid layers of black pieces, which cannot be captured; these pieces are drawn as gray circles in the figures. There are N capturable black pieces inside each vault, for some large integer N to be determined later. A white king can enter the vault through the "right ear", capture every internal piece, and then exit through the "left ear". Both ears are hallways, again with walls two pieces thick, with gaps where the arc paths end to allow the white king to enter and leave. The lengths of the "ears" can easily be adjusted to align with the other gadgets.
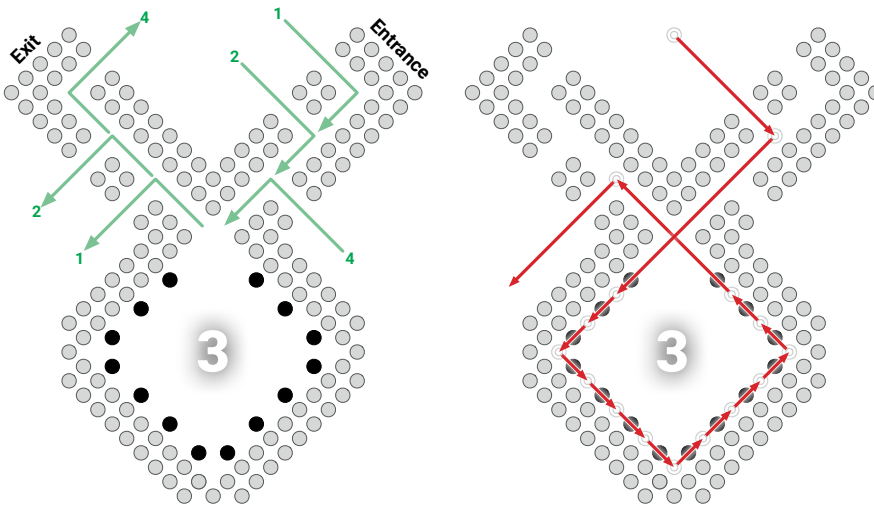

Figure 12.23. Left: A vertex gadget with three entrances and three exits. Right: A white king emptying the vault. Gray circles are black pieces that cannot be captured.

- For each arc $i \rightarrow j$, we have a *corner gadget*, which allows a white king leaving vertex gadget i to be redirected to vertex gadget j.

- Finally, wherever two arc paths cross, we have a *crossing gadget*; these gadgets allow the white king to traverse either arc path, but forbid switching from one arc path to the other.
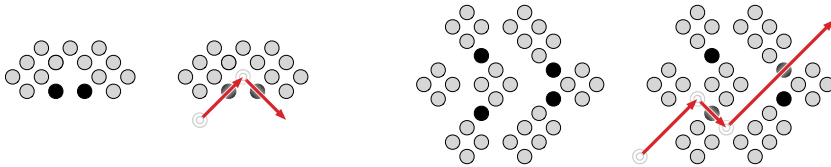

Figure 12.24. Left: One of two paths through a corner gadget. Right: One of two paths through a crossing gadget.

A single white king starts at the bottom corner of one of the vaults. In any legal move, this king must alternate between traversing entire arc paths and clearing vaults. The king can traverse the various gadgets backward, entering each vault through the exit and vice versa. But the reversal of a Hamiltonian cycle in G is another Hamiltonian cycle in G, so walking backward is fine.

If there is a Hamiltonian cycle in G, the white king can capture at least nN black pieces by visiting each of the other vaults and returning to the starting vault. On the other hand, if there is no Hamiltonian cycle in G, the white king can capture at most half of the pieces in the starting vault, and thus can capture at most $(n - 1/2)N + O(n^3)$ enemy pieces altogether. The $O(n^3)$ term accounts for the corner and crossing gadgets; each edge passes through one corner gadget and at most $n^2/2$ crossing gadgets.

To complete the reduction, we set $N = n^4$. Summing up, we obtain an $O(n^5) \times O(n^5)$ board configuration, with $O(n^5)$ black pieces and one white king. We can clearly construct this board configuration by brute force in polynomial time. Figure 12.25 shows a complete example of the construction.

It is still open whether the following related question is NP-hard: Given an $n \times n$ board configuration for international draughts, can (and therefore *must*) White capture *all* the black pieces (thereby winning the game) in a single turn?

♥12.16 On Beyond Zebra

P and NP are only the first two steps in an enormous hierarchy of complexity classes. To close this chapter (and the book), let me describe a few more classes of interest.
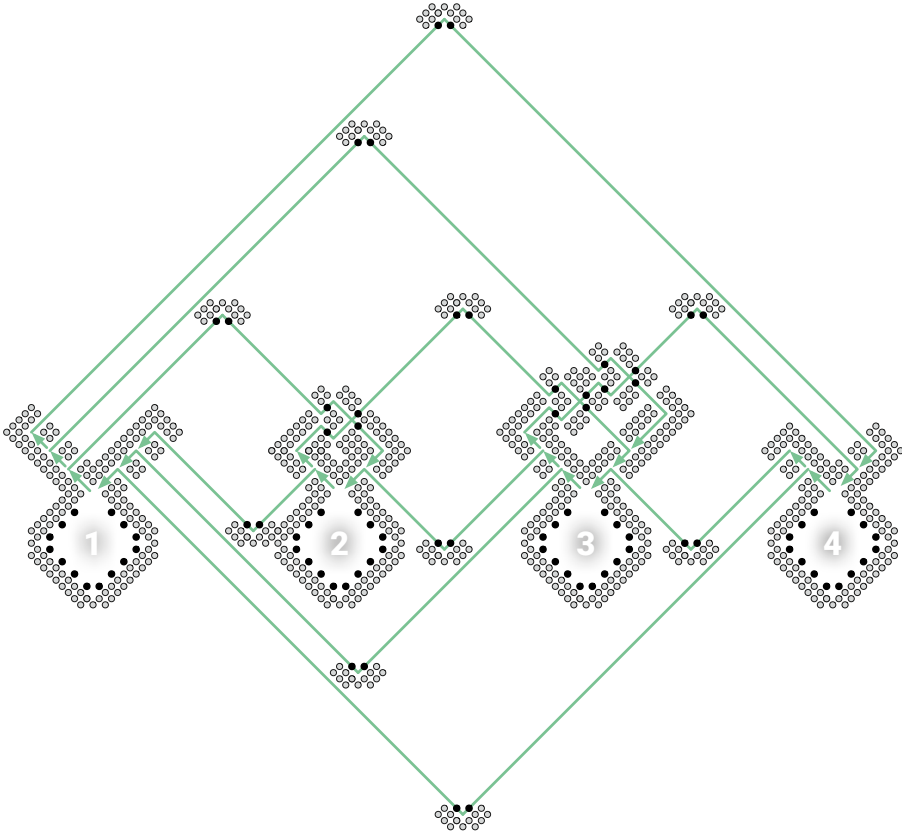
Figure 12.25. The final draughts configuration for the 4-vertex graph in Figure 12.22. (The green arrows are not actually part of the configuration.)

Polynomial Space

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 ≠ PSPACE, but nobody can even prove that P ≠ 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; similarly, if any PSPACE-hard problem is in P, then PSPACE=P.

The canonical PSPACE-hard problem is the *quantified boolean formula* problem, or *QBF*: Given a boolean formula Φ that may include any number of universal or existential quantifiers, but no free variables, is Φ equivalent to TRUE? For example, the following expression is a valid input to QBF:

$$\exists a : \forall b : \exists c : (\forall d : a \vee b \vee c \vee \bar{d}) \Longleftrightarrow ((b \wedge \bar{c}) \vee (\exists e : \overline{(\bar{a} \Rightarrow e)} \vee (c \neq a \wedge e))).$$

Sᴀᴛ is equivalent to the special case of QBF where the input formula contains only existential quantifiers (∃). QBF remains PSPACE-hard even when the input formula must have all its quantifiers at the beginning, the quantifiers strictly alternate between ∃ and ∀, and the quantified proposition is in conjunctive normal form, with exactly three literals in each clause, for example:

$$\exists a : \forall b : \exists c : \forall d : \big((a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d}) \big)$$

This restricted version of QBF can also be phrased as a two-player strategy question. Suppose two players, Alice and Bob, are given a 3CNF predicate with free variables $x_1, x_2, \ldots, x_n$. The players alternately assign values to the variables in order by index—Alice assigns a value to x_1, Bob assigns a value to x_2, and so on. Alice eventually assigns values to every variable with an odd index, and Bob eventually assigns values to every variable with an even index. Alice wants to make the expression Tʀᴜᴇ, and Bob wants to make it Fᴀʟsᴇ. Assuming Alice and Bob play perfectly, who wins this game? Not surprisingly, most two-player games[28] like tic-tac-toe, reversi, checkers, go, chess, and mancala—or more accurately, appropriate generalizations of these constant-size games to arbitrary board sizes—are PSPACE-hard.

Another canonical PSPACE-hard problem is *NFA totality*: Given a non-deterministic finite-state automaton M over some alphabet Σ, does M accept every string in Σ^*? The closely related problems *NFA equivalence* (Do two given NFAs accept the same language?) and *NFA minimization* (Find the smallest NFA that accepts the same language as a given NFA) are also PSPACE-hard, as are the corresponding questions about regular expressions. (The corresponding questions about *deterministic* finite-state automata are actually solvable in polynomial time.)

Exponential Time

The next significantly larger complexity class, ***EXP*** (also called EXPTIME), is the set of decision problems that can be solved in exponential time, that is, using at most 2^{n^c} steps for some constant $c > 0$. Every problem in PSPACE (and therefore in NP (and therefore in P)) is also in EXP. It is generally believed that PSPACE ⊊ EXP, but nobody can even prove that NP ≠ 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; similarly, if any EXP-hard problem is in NP, then EXP=NP. We *do* know that P ≠ EXP; in particular, no EXP-hard problem is in P.

---

[28]For a good (but inevitably aging) overview of known results on the computational complexity of games and puzzles, see Erik Demaine and Bob Hearn's monograph *Games, Puzzles, and Computation* (CRC Press, 2009).

Natural generalizations of many interesting 2-player games—like checkers, chess, mancala, and go—are actually EXP-hard. The boundary between PSPACE-complete games and EXP-hard games is rather subtle. For example, there are three ways to draw in chess (the standard 8×8 game): stalemate (the player to move is not in check but has no legal moves), repeating the same board position three times, or moving fifty times without capturing or moving a pawn. The $n \times n$ generalization of chess is either in PSPACE or EXP-hard depending on how we generalize these rules. If we declare a draw after (say) n^3 capture-free moves, then every game must end after a polynomial number of moves, so we can simulate all possible games from any given position using only polynomial space. On the other hand, if we ignore the capture-free move rule entirely, the resulting game can last an exponential number of moves, so there no obvious way to detect a repeating position using only polynomial space; indeed, this version of $n \times n$ chess is EXP-hard.

Excelsior!

Naturally, even exponential time is not the end of the story. **NEXP** is the class of decision problems that can be solve in *nondeterministic* exponential time; equivalently, a decision problem is in NEXP if and only if, for every Yes instance, there is a *proof* of this fact that can be checked in exponential time. **EXPSPACE** is the set of decision problems that can be solved using exponential *space*. Even these larger complexity classes have hard problems; for example, if we add the intersection operator $\cap$ to the syntax of regular expressions, deciding whether two such expressions describe the same language is EXPSPACE-hard. Beyond EXPSPACE are complexity classes with *doubly*-exponential resource bounds (EEXP, NEEXP, and EEXPSPACE), then *triply* exponential resource bounds (EEEXP, NEEEXP, and EEEXPSPACE), and so on ad infinitum.

All these complexity classes can be ordered by inclusion:

$$P \subseteq NP \subseteq PSPACE \subseteq EXP \subseteq NEXP \subseteq EXPSPACE \subseteq EEXP \subseteq NEEXP \subseteq \cdots$$

Most complexity theorists strongly believe that every inclusion in this sequence is strict; that is, no two of these complexity classes are equal. However, the strongest result that has been proved is that every class in this sequence is strictly contained in the class *three* steps later in the sequence. For example, we have proofs that $P \neq EXP$ and $PSPACE \neq EXPSPACE$, but not whether $P \neq PSPACE$ or $NP \neq EXP$.

The limit of this series of increasingly exponential complexity classes is the class **ELEMENTARY** of decision problems that can be solved using time or space bounded by a function the form $2 \uparrow^k n$ for some constant integer k, where

$$2 \uparrow^k n := \begin{cases} n & \text{if } k = 0, \\ 2^{2 \uparrow^{k-1} n} & \text{otherwise.} \end{cases}$$

For example, $2 \uparrow^1 n = 2^n$ and $2 \uparrow^2 n = 2^{2^n}$.

It may be tempting to conjecture that *every* natural decidable problem can be solved in elementary time, but in fact this conjecture is incorrect. Consider the ***generalized regular expressions*** defined by recursively combining (possibly empty) strings over some finite alphabet by concatenation (xy), union $(x + y)$, Kleene closure (x^*), and negation $(\overline{x})$. For example, the generalized regular expression $\overline{(0 + 1)^*00(0 + 1)^*}$ represents the set of strings in $\{0, 1\}^*$ that do *not* contain two 0s in a row. It is possible to determine algorithmically whether two generalized regular expressions describe identical languages, by recursively converting each expression into an equivalent NFA, converting each NFA into a DFA, and then minimizing the DFA. However, the running time of this algorithm has the non-elementary bound $2 \uparrow^{\Theta(n)} 2$, intuitively because each layer of recursive negation can exponentially increase the number of states. In fact, Larry Stockmeyer proved in 1974 that this problem *cannot* be solved in merely elementary time, even if we forbid Kleene closure.

Exercises

1. (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?

2. 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?

3. 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

$$(\overline{x} \wedge y \wedge \overline{z}) \vee (y \wedge z) \vee (x \wedge \overline{y} \wedge \overline{z})$$

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

   (a) Describe a polynomial-time algorithm to solve DNF-SAT.

(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,
>
> $$(x \vee y \vee \bar{z}) \wedge (\bar{x} \vee \bar{y}) \iff (x \wedge \bar{y}) \vee (y \wedge \bar{x}) \vee (\bar{z} \wedge \bar{x}) \vee (\bar{z} \wedge \bar{y})$$
>
> Now we can use the algorithm 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!

4. The problem ALLORNOTHING3SAT asks, given a 3CNF boolean formula, whether there is an assignment to the variables such that each clause either has three TRUE literals or has three FALSE literals.

   (a) Describe a polynomial-time algorithm to solve ALLORNOTHING3SAT.

   (b) But 3SAT is NP-hard! Why doesn't the existence of this algorithm prove that P=NP?

5. (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 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. Only 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 arbitrary set X of positive integers, whether X can be partitioned into two sets A and B such that $\sum A = \sum B$. Describe and analyze a **polynomial-time** algorithm that either computes an equal partition of a given set of positive integers or correctly reports that no such partition exists, using the magic black box as a subroutine.

♣♥(f) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary generalized regular expression R (as defined just before the Exercises), whether R matches any string. Describe and analyze a **polynomial-time** algorithm that either finds a single string that matches a given generalized regular expression or correctly reports that no such string exists, using the magic black box as a subroutine.

6. **There's something special about the number 3.**

   (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. *[Hint: This problem is strongly connected to topics described in an earlier chapter.]*

7. **There's nothing special about the number 3.**

   (a) The problem 12Partition is defined as follows: Given a set S of $12n$ positive integers, determine whether the elements of S can be split into n subsets, each with 12 elements, whose sums are all equal. Prove that 12Partition is NP-hard. *[Hint: Reduce from 3Partition. It may be easier to consider multisets first.]*

   (b) The problem 12Color is defined as follows: Given an undirected graph G, determine whether we can color each vertex with one of twelve colors, so that every edge touches two different colors. Prove that 12Color is NP-hard. *[Hint: Reduce from 3Color.]*

(c) The problem 12SAT is defined as follows: Given a boolean formula Φ in conjunctive normal form, with exactly twelve literals per clause, determine whether Φ has a satisfying assignment. Prove that 12SAT is NP-hard. *[Hint: Reduce from 3SAT.]*

8. There are two different versions of the Hamiltonian cycle problem, one for directed graphs and one for undirected graphs. Earlier in this chapter you can find two proofs that the *directed* Hamiltonian cycle problem is NP-hard.

   (a) Describe a polynomial-time reduction from the *undirected* Hamiltonian cycle problem to the *directed* Hamiltonian cycle problem. Prove your reduction is correct.

   (b) Describe a polynomial-time reduction from the *directed* Hamiltonian cycle problem to the *undirected* Hamiltonian cycle problem. Prove your reduction is correct.

   (c) Which of these two reductions implies that the *undirected* Hamiltonian cycle problem is NP-hard?

9. (a) Describe a polynomial-time reduction from UNDIRECTEDHAMILTONIAN-CYCLE to DIRECTEDHAMILTONIANCYCLE.

   (b) Describe a polynomial-time reduction from DIRECTEDHAMILTONIAN-CYCLE to UNDIRECTEDHAMILTONIANCYCLE.

10. (a) Describe a polynomial-time reduction from the HAMILTONIANPATH problem to HAMILTONIANCYCLE.

    (b) Describe a polynomial-time reduction from the HAMILTONIANCYCLE problem to HAMILTONIANPATH. *[Hint: A polynomial-time reduction can call the black-box subroutine more than once, but it doesn't have to.]*

11. Consider the following subtle variants of CNFSAT. For each problem, the input is a boolean formula Φ in conjunctive normal form, and the goal is to determine whether Φ has a satisfying assignment.

    (a) Suppose every clause of Φ contains **at most** three literals and each variable appears in at most **three** clauses. Prove that this variant of CNFSAT is NP-hard.

    (b) Suppose every clause of Φ contains **exactly** three literals and each variable appears in at most **four** clauses. Prove that this variant of 3SAT is NP-hard. *[Hint: Solve part (a) first.]*

    ♥(c) Suppose every clause of Φ can contain *any number of* literals, but each variable appears in at most **two** clauses. Describe a polynomial-time algorithm for this variant of CNFSAT.

♥(d) Suppose every clause of Φ contains ***exactly*** three literals and each variable appears in at most ***three*** clauses. Prove that Φ must be satisfiable. (So this variant of 3SAT is completely trivial!)

12. (a) Prove that NOTALLEQUAL3SAT is NP-hard.

   (b) Prove that 1-IN-3SAT is NP-hard.

13. 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 XCNF-SAT is NP-hard. *[Hint: Do not try to do both.]*

♥14. Consider the following variant of 3SAT, called MAJORITY3SAT. Just like 3SAT, the input to MAJORITY3SAT is a boolean formula Φ in conjunctive normal form, with exactly three literals er clause. MAJORITY3SAT asks whether there is an assignment to the variables of Φ, such that every clause contains *at least two* TRUE literals.

   Either describe an algorithm that solves MAJORITY3SAT in polynomial time or prove that MAJORITY3SAT is NP-hard. *[Hint: Do not try to do both.]*

♠♥15. For any subset $X \subseteq \{0, 1, 2, 3\}$, consider the following problem, which I'll call ***X-3SAT***. The input is a boolean formula Φ in conjunctive normal form, with exactly three literals in each clause. The problem is to decide whether there is an assignment to the variables of Φ such that in each clause of Φ, the number of TRUE literals is in the set X. For example:

   • $\{1, 2, 3\}$-3SAT is the standard 3SAT problem.
   • $\{0, 3\}$-3SAT is the same as ALLORNOTHING3SAT. (See Exercise 4.)
   • $\{1, 2\}$-3SAT is usually called NOTALLEQUAL3SAT. (See Exercise 12(a).)
   • $\{1\}$-3SAT is usually called 1-IN-3SAT. (See Exercise 12(b).)
   • $\{1, 3\}$-3SAT is usually called XCNF-3SAT. (See Exercise 13.)
   • $\{2, 3\}$-3SAT is usually called MAJORITY3SAT. (See Exercise 14.)

   Give a complete list of all subsets $X \subseteq \{0, 1, 2, 3\}$ such that X-3SAT is solvable in polynomial time, assuming $P \neq NP$. *[Hint: Don't give 16 different arguments.]*

16. Prove that the following problems are NP-hard.

   (a) Given an undirected graph G, does G contain a simple path that visits all but 17 vertices?

    (b) Given an undirected graph G, does G have a spanning tree in which every node has degree at most 23?

    (c) Given an undirected graph G, does G have a spanning tree with at most 42 leaves?

    (d) Given an undirected graph $G = (V, E)$, what is the size of the largest subset of vertices $S \subseteq V$ such that at most 374 edges in E have both endpoints in S?

    (e) Given an undirected graph $G = (V, E)$, what is the size of the largest subset of vertices $S \subseteq V$ such that each vertex in S has at most 473 neighbors in S?

    (f) Given an undirected graph G, is it possible to color the vertices of G with three different colors, so that at most 31337 edges have both endpoints the same color?

17. Prove that the following variants of the minimum spanning tree problem are NP-hard.

    (a) Given a graph G, compute the *maximum-diameter* spanning tree of G. (The diameter of a tree T is the length of the longest path in T.)

    (b) Given a graph G with weighted edges, compute the minimum-weight *depth-first* spanning tree of G.

    (c) Given a graph G with weighted edges and a subset S of vertices of G, compute the minimum-weight spanning tree all of whose leaves are in S.

    (d) Given a graph G with weighted edges and an integer ℓ, compute the minimum-weight spanning tree with at most ℓ leaves.

    (e) Given a graph G with weighted edges and an integer Δ, compute the minimum-weight spanning tree where every node has degree at most Δ.

18. (a) Using the gadget in Figure 12.26(a), prove that deciding whether a given *planar* graph is 3-colorable is NP-hard. *[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 gadget in Figure 12.26(b), prove that deciding whether a planar graph *with maximum degree 4* is 3-colorable is NP-hard. *[Hint: Replace any vertex with degree greater than 4 with a collection of gadgets connected so that no degree is greater than four.]*

19. Prove that PLANARCIRCUITSAT is NP-hard. *[Hint: Construct a gadget for crossing wires.]*
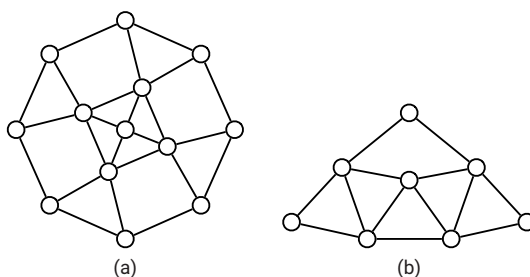
Figure 12.26. (a) Gadget for planar 3-colorability. (b) Gadget for degree-4 planar 3-colorability.

20. (a) Describe a polynomial-time reduction from 3SAT to 4SAT.

    (b) Describe a polynomial-time reduction from 4SAT to 3SAT.

♥21. Describe a direct polynomial-time reduction from 4COLOR to 3COLOR. (This is a lot harder than the opposite direction.)

22. A *domino* is a 1×2 rectangle divided into two squares, each of which is labeled with an integer.[29] In a *legal arrangement* of dominos, the dominos are lined up end-to-end so that the numbers on adjacent ends match.
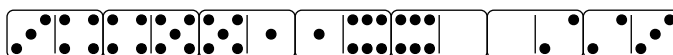

Figure 12.27. A legal arrangement of dominos in which every integer between 0 and 6 appears twice

    For each of the following problems, either describe a polynomial-time algorithm or prove that the problem is NP-hard:

    (a) Given an arbitrary bag D of dominos, is there a legal arrangement of *all* the dominos in D?

    (b) Given an arbitrary bag D of dominos, is there a legal arrangement of a dominos from D in which every integer between 1 and n appears exactly twice?

    ♥(c) Given an arbitrary bag D of dominos, what is the largest number of dominos we can take from D to make a legal arrangement?

23. *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

---

[29]These integers are usually represented by pips, exactly like dice. On a standard domino, the number of pips on each side is between 0 and 6, although one can buy sets with up to 9 or even 12 pips on each side; we will allow arbitrary integer labels. A standard set of dominos contains exactly one domino for each possible unordered pair of labels; we do *not* assume that the inputs to our problems have this property.

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-hard.

24. Recall that a 5-coloring of a graph G is a function that assigns each vertex
of G a "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
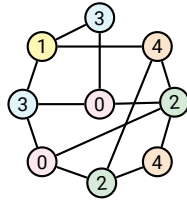is NP-hard. *[Hint: Reduce from the standard 5Color problem.]*

Figure 12.28. A careful 5-coloring.

25. (a) A subset S of vertices in an undirected graph G is *half-independent* if
each vertex in S is adjacent to *at most one* other vertex in S. Prove that
finding the size of the largest half-independent set of vertices in a given
undirected graph is NP-hard.

   (b) A subset S of vertices in an undirected graph G is *sort-of-independent* if
if each vertex in S is adjacent to *at most 374* other vertices in S. Prove
that finding the size of the largest sort-of-independent set of vertices in
a given undirected graph is NP-hard.

   (c) A subset S of vertices in an undirected graph G is *almost independent* if at
most 374 edges in G have both endpoints in S. Prove that finding the size
of the largest almost-independent set of vertices in a given undirected
graph is NP-hard.

26. Let $G = (V, E)$ be a graph. A *dominating set* in G is a subset S of the vertices
such that every vertex in G is either in S or adjacent to a vertex in S. The
DominatingSet problem asks, given a graph G and an integer k as input,
whether G contains a dominating set of size k. Prove that this problem is
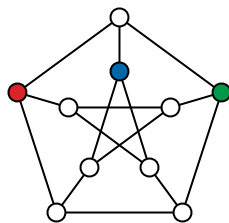NP-hard.

Figure 12.29. A dominating set of size 3 in the Petersen graph.

27. A subset S of vertices in an undirected graph G is *triangle-free* if, for every triple of vertices $u, v, w \in S$, at least one of the three edges uv, uw, vw is *absent* from G. Prove that finding the size of the largest triangle-free subset of vertices in a given undirected graph is NP-hard.
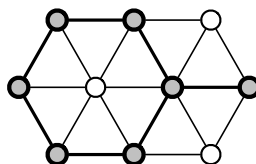

Figure 12.30. A triangle-free subset of 7 vertices. This is not the largest such subset in this graph.

28. The RECTANGLETILING problem is defined as follows: Given one large rectangle and several smaller rectangles, determine whether the smaller rectangles can be placed inside the large rectangle with no gaps or overlaps.

    (a) Prove that RECTANGLETILING is NP-hard.
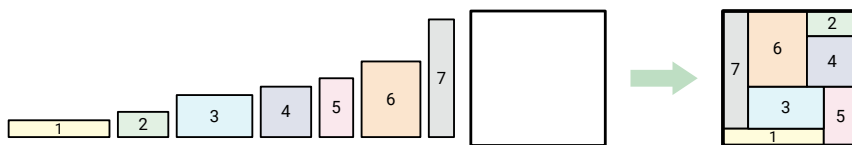
    (b) Prove that RECTANGLETILING is *strongly* NP-hard.

Figure 12.31. A positive instance of the RECTANGLETILING problem.

29. (a) A subset B of vertices in a graph G is a ***Burr set*** if removing every vertex in B from G leaves a subgraph that does not contain a Hamiltonian path. Prove that finding the smallest Burr set in a given graph is NP-hard.

    (b) A subset S of vertices in a graph G is a ***Schuyler set*** if removing every vertex in S from G leaves a subgraph that *does* contain a Hamiltonian path. Prove that finding the smallest Schuyler set in a given graph is NP-hard.

30. (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-hard.

    (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-hard. *[Hint: Use part (a). Or not.]*

31. 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 more than half of the total weight of all edges in G. Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-hard.
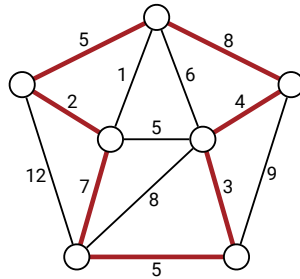

Figure 12.32. A heavy Hamiltonian cycle. The cycle has total weight 34; the graph has total weight 67.

32. For each of the following problems, either describe a polynomial-time algorithm or prove that the problem is NP-hard.

    (a) A *double-Eulerian tour* 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 tour?

    (b) A *double-Hamiltonian tour* 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 tour?

    (c) A *double-Hamiltonian* **circuit** in an undirected graph G is a closed walk that visits every vertex in G exactly twice *and traverses each edge in G at most once*. Given a graph G, does G have a double-Hamiltonian circuit?

    (d) A *triple-Eulerian tour* in an undirected graph G is a closed walk that traverses every edge in G exactly three times. Given a graph G, does G have a triple-Eulerian tour?

    (e) A *triple-Hamiltonian tour* in an undirected graph G is a closed walk that visits every vertex in G exactly three times. Given a graph G, does G have a triple-Hamiltonian tour?

33. This exercise asks you to prove that a certain reduction from VERTEXCOVER to STEINERTREE is correct. Suppose we want to find the smallest vertex cover in a given undirected graph $G = (V, E)$. We construct a new graph $H = (V', E')$ as follows:

    - $V' = V \cup E \cup \{z\}$
    - $E' = \{ve \mid v \in V \text{ is an endpoint of } e \in E\} \cup \{vz \mid v \in V\}$.

    Equivalently, we construct H by subdividing each edge in G with a new vertex, and then connecting all the original vertices of G to a new *apex* vertex z.

    Prove that G has a vertex cover of size k if and only if there is a subtree of H with $k + |E| + 1$ vertices that contains every vertex in $E \cup \{z\}$.

34. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.
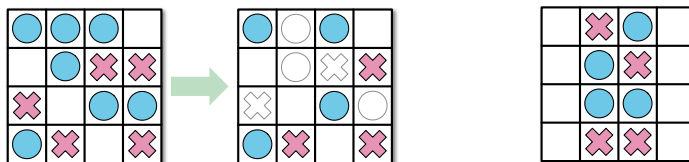
A solvable puzzle and one of its many solutions.   An unsolvable puzzle.

    Prove that it is NP-hard to determine, given an initial configuration of red and blue stones, whether the puzzle can be solved.

35. Each of the following games involves an $n \times m$ grid of squares, where each square is either empty or occupied by a stone. In a single move, you can remove *all* the stones in an arbitrary column.

    (a) Prove that it is NP-hard to find the *smallest* subset of columns that can be cleared so that *at most* one stone remains in each row of the grid.

    (b) Prove that it is NP-hard to find the *largest* subset of columns that can be cleared so that *at least* one stone remains in each row of the grid.

    ♥(c) Prove that it is NP-hard to determine whether *any* subset of columns can be cleared so that *exactly* one stone remains in each row of the grid.

36. 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 at least one of their 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.

37. You're in charge of choreographing a musical for your local community theater, and it's time to figure out the final pose of the big show-stopping number at the end. ("Streetcar!") You've decided that each of the n cast members in the show will be positioned in a big line when the song finishes, all with their arms extended and showing off their best spirit fingers.

    The director has declared that during the final flourish, each cast member must either point both their arms up or point both their arms down; it's your job to figure out who points up and who points down. Moreover, the director has also given you a list of arrangements that will upset his delicate artistic temperament. Each forbidden arrangement is a subset of the cast members paired with arm positions; for example: "Marge may not point her arms up while Ned, Apu, and Smithers point their arms down."

    Prove that finding an acceptable arrangement of arm positions is NP-hard.

38. 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.

39. The party you are attending is going great, but now it's time to line up for *The Algorithm March (*アルゴリズムこうしん*)*! This dance was originally developed by the Japanese comedy duo Itsumo Kokokara (いつもここから)

for the children's television show PythagoraSwitch (ピタゴラスイッチ). The Algorithm March is performed by a line of people; each person in line starts a specific sequence of movements one measure later than the person directly in front of them. Thus, the march is the dance equivalent of a musical round or canon, like "Row Row Row Your Boat" or "Frère Jacques".

Proper etiquette dictates that each marcher must know the person directly in front of them in line, lest a minor mistake lead to horrible embarrassment between strangers. Suppose you are given a complete list of which people at your party know each other. Prove that it is NP-hard to determine the largest number of party-goers that can participate in the Algorithm March. You may assume without loss of generality that there are no ninjas at your party.

♣40. Prove that the following problems about nondeterministic finite-state automata and regular expressions are NP-hard:

(a) Given an NFA M over the alphabet $\Sigma = \{0, 1\}$, is there a string in Σ^* that M does *not* accept?

(b) Given an *acyclic* NFA M over the alphabet $\Sigma = \{0, 1\}$, what is the length of the *shortest* string in Σ^* that M does *not* accept?

(c) Given a regular expression R over the alphabet $\Sigma = \{0, 1\}$, is there a string in Σ^* that R does *not* match?

(d) Given a *star-free* regular expression R over the alphabet $\Sigma = \{0, 1\}$, what is the length of the *shortest* string in Σ^* that R does *not* match?

(In fact, problems (a) and (c) are PSPACE-complete; even proving that these problems are in PSPACE is nontrivial.)

♣41. (a) Describe a polynomial-time algorithm for the following problem: Given an NFA M over the alphabet $\Sigma = \{0, 1\}$, is there a string in Σ^* that M *does* accept?

(b) Describe a polynomial-time algorithm for the following problem: Given a regular expression R over the alphabet $\Sigma = \{0, 1\}$, is there a string in Σ^* that R *does* match?

(c) The complement of any regular language is another regular language. So why don't these two algorithms, together with the NP-hardness results in Problem 40, prove that P=NP?

42. Charon needs to ferry n recently deceased people across the river Acheron into Hades. Certain pairs of these people are sworn enemies, who cannot be together on either side of the river unless Charon is also present. (If two enemies are left alone, one will steal the obol from the other's mouth, leaving

them to wander the banks of the Acheron as a ghost for all eternity. Let's just say this is a Very Bad Thing.) The ferry can hold at most k passengers at a time, including Charon, and only Charon can pilot the ferry.[30]

Prove that it is NP-hard to decide whether Charon can ferry all n people across the Acheron unharmed (aside from being, you know, dead). The input for Charon's problem consists of the integers k and n and an n-vertex graph G describing the pairs of enemies. The output is either TRUE or FALSE.

Please do not write your solution in classical Latin.

---

[30]This problem is a generalization of the well-known wolf-goat-and-cabbage puzzle, whose first known appearance is in the remarkable medieval manuscript *Propositiones ad Acuendos Juvenes* [*Problems to Sharpen the Young*].

XVIII. PROPOSITIO DE HOMINE ET CAPRA ET LVPO.
Homo quidam debebat ultra fluuium transferre lupum, capram, et fasciculum cauli. Et non potuit aliam nauem inuenire, nisi quae duos tantum ex ipsis ferre ualebat. Praeceptum itaque ei fuerat, ut omnia haec ultra illaesa omnino transferret. Dicat, qui potest, quomodo eis illaesis transire potuit?
Solutio. Simili namque tenore ducerem prius capram et dimitterem foris lupum et caulum. Tum deinde uenirem, lupumque transferrem: lupoque foris misso capram naui receptam ultra reducerem; capramque foris missam caulum transueherem ultra; atque iterum remigassem, capramque assumptam ultra duxissem. Sicque faciendo facta erit remigatio salubris, absque uoragine lacerationis.

For those few readers whose classical Latin is a little rusty, here is an English translation:

XVIII. THE PROBLEM OF THE MAN, THE GOAT, AND THE WOLF.
A man needed to transfer a wolf, a goat, and a bundle of cabbage across a river. However, he found that his boat could only bear the weight of two [objects at a time, including the man]. And he had to get everything across unharmed. Tell me if you can: How they were able to cross unharmed?
Solution. In a similar fashion [as an earlier problem], I would first take the goat across and leave the wolf and cabbage on the opposite bank. Then I would take the wolf across; leaving the wolf on shore, I would retrieve the goat and bring it back again. Then I would leave the goat and take the cabbage across. And then I would row across again and get the goat. In this way the crossing would go well, without any threat of slaughter.

The most likely author of the *Propositiones* is the prolific 8th-century English scholar Alcuin of York. The evidence for Alcuin's authorship of this treatise is somewhat circumstantial; however, we do know from his correspondence with Charlemagne that he sent the emperor some "simple arithmetical problems for fun". Most modern scholars believe that even if Alcuin did write the *Propositiones*, he did not invent all of the problems himself, but rather collected them from even earlier sources.

Some things never change.